UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR

INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE MASTER

MASTER UNIVERSITARIO EN INVESTIGACIÓN

EN TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES

HDFS File Formats: Study and Performance Comparison

Autor:

D. Álvaro Alonso Isla

Tutor:

Dr. D. Miguel Ángel Martínez Prieto
Dr. D. Aníbal Bregón Bregón

Valladolid, 29 de Junio de 2018

TÍTULO:	**HDFS File Formats: Study and Performance Comparison**
AUTOR:	**D. Álvaro Alonso Isla**
TUTOR:	**Dr. D. Miguel Ángel Martínez Prieto** **Dr. D. Aníbal Bregón Bregón**
DEPARTAMENTO:	**Departamento de Informática**

Tribunal

PRESIDENTE:	**Dr. D. Pablo de la Fuente Redondo**
VOCAL:	**Dr. D. Alejandra Martínez Monés**
SECRETARIO:	**Dr. D. Guillermo Vega Gorgojo**
FECHA:	**29 de Junio de 2018**
CALIFICACIÓN:	

Resumen del TFM

El sistema distribuido Hadoop se está volviendo cada vez más popular a la hora de almacenar y procesar grandes cantidades de datos (Big Data). Al estar compuesto por muchas máquinas, su sistema de ficheros, llamado HDFS (Hadoop Distributed File System), también es distribuido. Al ser HDFS un sistema de almacenamiento distinto a los tradicionales, se han desarrollado nuevos formatos de almacenamiento de ficheros para ajustarse a las características particulares de HDFS. En este trabajo estudiamos estos nuevos formatos de ficheros, prestando especial atención a sus características particulares, con el fin de ser capaces de intuir cuál de ellos tendrá un mejor funcionamiento según las necesidades de los datos que tengamos. Para alcanzar este objetivo, hemos propuesto un marco teórico de comparación, con el que poder reconocer fácilmente qué formatos se ajustan a nuestras necesidades. Además, hemos realizado un estudio experimental para reforzar la comparación anterior, y poder obtener conclusiones sobre qué formatos son los mejores para los casos expuestos. Para ello se han seleccionado dos conjuntos de datos con distintas características, y un grupo de consultas básicas, que se ejecutan con trabajos MapReduce escritos en Java y a través de la herramienta Hive. El objetivo final de este trabajo será ser capaces de identificar las distintas fortalezas y debilidades de los formatos de almacenamiento.

Palabras clave

Big Data, Hadoop, HDFS, MapReduce, Formatos de Almacenamiento.

Abstract

The distributed system Hadoop has become very popular for storing and process large amounts of data (Big Data). As it is composed of many machines, its file system, called HDFS (Hadoop Distributed File System), is also distributed. But as HDFS is not a traditional storage system, plenty of new file formats have been developed, to take advantage of its features. In this work we study that new formats to find out their characteristics, and being able to decide which ones can be better knowing the needs of our data. For that goal, we have made a theoretical framework to compare them, and easily recognize which formats fit our needs. Also we have made an experimental study to find out how the formats work in some specific situations, selecting two very different datasets and a set of simple queries, resolved with MapReduce jobs, written with Java or run using Hive tool. The final goal of this work is to be able to identify the different strengths and weakenesses of the file formats.

Keywords

Big Data, Hadoop, HDFS, MapReduce, File Formats.

Agradecimientos

Este trabajo ha sido posible gracias a la colaboración de Boeing Research & Technology Europe (BR&T-E) y forma parte del proyecto AIRPORTS. Este proyecto está financiado por CDTI y MINECO (FEDER) ref. IDI-20150616, CIEN 2015. Quiero agradecer a todos los miembros de BR&T-E por dejarnos utilizar su cluster Hadoop para la realización de las pruebas de este proyecto, y en especial a David Scarlatti por facilitarme toda la información que he ido necesitando sobre las características del sistema.

He de hacer un agradecimiento especial a mi compañero Iván García, con el que he compartido largas conversaciones sobre el progreso que íbamos haciendo en nuestros trabajos, que ayuda mucho a sobrellevar los malos momentos. Además, en más de una ocasión le he tenido que pedir consejo sobre ciertos aspectos que él conocía mejor que yo.

También quiero dar las gracias a mis tutores Miguel Ángel Martínez y Aníbal Bregón, por haber dedicado tanto tiempo en ayudarme y llevar un completo seguimiento de mi avance. Gracias a Miguel Ángel, que de nuevo me ha prestado una de sus ideas para llevarla a cabo y hacer un proyecto de ella, aunque en esta ocasión el resultado es bastante distinto de lo que la idea inicial planteaba. Y especialmente agradecer a Aníbal su especial dedicación, que en los momentos finales del trabajo en los que Miguel Ángel no estaba disponible, ha hecho un esfuerzo sobrehumano para sacar tiempo para mi trabajo, aconsejarme y guiarme, a pesar de ser un tema en el que él no está especializado.

Finalmente, y no por ello menos importante, gracias a mi familia y a mi pareja Ainoa, que tantas horas han tenido que escuchar mis problemas, sin saber qué responderme porque no entendían. Gracias por todo el apoyo prestado y por esos momentos en que me han ayudado a desconectar y descansar.

Contents

1 Introduction **1**

1.1 Motivation . . . 2
1.2 Objectives and Results of the Work . . . 3
1.3 Methodology . . . 4
1.4 Document structure . . . 4

2 Distributed file systems **7**

2.1 Basic concepts . . . 7
2.2 Characteristics . . . 8
2.2.1 Transparency . . . 8
2.2.2 Naming . . . 8
2.2.3 Scalability . . . 9
2.2.4 Remote Multiple Access . . . 10
2.3 Architectures . . . 10
2.4 Fault tolerance . . . 11

3 Hadoop Distributed File System **13**

3.1 GFS . . . 14
3.2 HDFS Architecture . . . 15
3.3 HDFS Writing and Reading . . . 17
3.4 HDFS Blocks Robustness . . . 18
3.4.1 Replication . . . 18
3.4.2 Reading/writing Failure . . . 19
3.4.3 Namenode Failure . . . 20

4 HDFS storage formats **21**

4.1 Text files . . . 23
4.1.1 CSV . . . 23
4.1.2 JSON . . . 23
4.2 SequenceFiles . . . 24
4.3 Avro . . . 26
4.4 RCFiles . . . 27
4.5 ORC . . . 29
4.6 Parquet . . . 30
4.7 Comparison . . . 33

5 Experimental results **37**
5.1 Methodology . . . 37
5.1.1 Mapreduce . . . 37
5.1.2 Datasets . . . 38
5.1.3 Considered Formats . . . 39
5.1.4 Queries . . . 40
5.1.5 Experimentation environment . . . 41
5.1.6 Used Parameters . . . 41
5.2 Results . . . 42
5.2.1 Opensky . . . 42
5.2.2 Github log . . . 57

6 Conclusions **79**
6.1 Future work . . . 80

I APPENDICES **85**

A Opensky Schema **87**

B GitHub log Schema **89**

List of Figures

3.1 HDFS architecture . . . 16
3.2 HDFS writing operation . . . 17
3.3 HDFS reading operation . . . 18
3.4 HDFS replication example . . . 19

4.1 SequenceFile record compression . . . 24
4.2 SequenceFile block compression . . . 25
4.3 Avro structure . . . 27
4.4 RCFile structure . . . 28
4.5 ORC structure . . . 30
4.6 Parquet schema example . . . 31
4.7 Parquet definition level example . . . 32
4.8 Parquet repetition level example . . . 32
4.9 Parquet file structure . . . 33

5.1 Opensky format sizes . . . 42
5.2 Opensky query 1 performance . . . 44
5.3 Opensky query 2 performance . . . 47
5.4 Opensky query 3 performance . . . 50
5.5 Opensky query 4 performance . . . 53
5.6 Opensky query 5 performance . . . 55
5.7 GitHub log format sizes . . . 57
5.8 GitHub log query 1 performance . . . 59
5.9 GitHub log query 2.1 performance . . . 62
5.10 GitHub log query 2.2 performance . . . 65
5.11 GitHub log query 3 performance . . . 68
5.12 GitHub log query 4 performance . . . 71
5.13 GitHub log query 5.1 performance . . . 73
5.14 GitHub log query 5.2 performance . . . 76

List of Tables

4.1 File formats comparison table 34

5.1 Opensky format sizes (GB) 42
5.2 Opensky query 1 results 45
5.3 Opensky query 2 results 48
5.4 Opensky query 3 results 51
5.5 Opensky query 4 results 54
5.6 Opensky query 5 results 56
5.7 GitHub log format sizes (GB) 58
5.8 GitHub log query 1 results 60
5.9 GitHub log query 2.1 results 63
5.10 GitHub log query 2.2 results 66
5.11 GitHub log query 3 results 69
5.12 GitHub log query 4 results 72
5.13 GitHub log query 5.1 results 74
5.14 GitHub log query 5.2 results 77

Chapter 1

Introduction

Since several years, the Big Data concept is becoming more and more popular, but It is not something new. We can say that this concept reached maturity in 2013, when it was introduced as a new term in the Oxford English Dictionary (OED) [1], where it is defined as "Extremely large data sets that may be analysed computationally to reveal patterns, trends, and associations, especially relating to human behaviour and interactions". However, this is not a very accurate definition; it only refers to the volume of the data and some fields in which it can be used. Traditionally, Big Data has been characterized with the so called 3 *V*s [2], i.e., volume, velocity and variety.

The **volume** refers to the most famous characteristic of Big Data. It is simply the storage space that the data uses. Years ago, a dataset between 10GB or 100GB could have been considered Big Data. Nowadays the typical size at which a dataset is considered Big Data is about 1TB. This is decided by the actual computation capacity, because Big Data can also be defined as "data that exceeds the processing capacity of conventional database systems" [3]. Thus, if the capacity of conventional databases increases, a dataset considered Big Data some years ago, could be now not included in that group.

The next *V* is the **velocity**. This concept refers to the speed at which the data is generated. In these days, the amount of data produced each second is immense. For example, Youtube generates 300 hours of video every minute [4]. For systems that process the data in real time is very hard to manage these data generation speeds. Because of this, it is necessary to use new techniques to be able to process data as it is generated.

The last *V* is the **variety**. This characteristic can be understood in two different ways. The first one is the variety in the data form and type, being possible to mix natural speech text with multimedia, with structured data, or whichever type of information. And the second aspect of the variety is related with the data structure. Most of the data generated nowadays is produced by people, and it is written it in natural language. Thus, the majority of the data is non structured data, which is much more difficult to process than when it follows a structure.

Some authors add two more *V*s to the previous three, thus having the Big Data 5 *V*s [5]. These new characteristics are Veracity and Value. **Veracity** refers to the data quality or trustworthiness. It is important to take this into account because nowadays the majority of the data is generated by people, and maybe not all the data can be trusted. And the **value** means that the data has to produce a benefit. If there is no possibility of bring value to the companies or the society, it is not worth to process that data.

The last two Vs are the newest ones, and they make reference to the data meaning, not just to its numerical characteristics. They are mostly related with where the data comes from and if it has any utility.

As we have mentioned, the volume is the most significant of the Big Data characteristics, however, the volume has more problems apart from having enough storage capacity. The data is usually queried, and that queries must be resolved as fast as possible. With very large amounts of data, this task is not easy.

One popular solution to solve the problems originated by the volume is to use distributed file systems. A distributed file system is just a set of machines that makes the role of a single file system. Having many machines, the storage capacity is much higher, and it is possible to increase a lot the parallelization of tasks. One distributed file system very common in these days is the one used within Apache Hadoop.

Apache Hadoop [6, 7] is an open-source software framework, written in Java, that allows a distributed processing and storage of very large volumes of data, in a computers cluster composed of commodity hardware[1]. Hadoop can be divided into storage and processing. The storage system of Hadoop is called HDFS (Hadoop Distributed File System) [8, 9], and the processing part MapReduce [10].

1.1 Motivation

In this project we will focus only in the storage part of Hadoop, studying the different file formats that have been developed for this environment. These new formats have different characteristics, and depending on the structure of the data and its final use, one format can be more suitable for some tasks than the others. But, how is it possible to know which format is better in a specific situation? Although here have been different studies to compare the formats between them and answer this question, there is no published work covering and comparing all kind of formats in an exhaustive way.

For example, D. Plase [11] made a comparison between the two most known file formats, Avro and Parquet. In her work the rest of the file formats were discarded because of their theoretical limitations. For the selected ones, there is a comparison of the compression capacity and execution performance for a quite big number of datasets and queries, taken from the TPC-H benchmark [12], and modifying some of them. The execution of the queries have been done using Hive-MapReduce. However, the theoretical comparison is very short, discarding all the other formats without giving them an opportunity in the experimentation.

Another paper comparing two file formats in the Hadoop environment is the Thesis written by Gavin So [13]. In this case, Parquet and JSON formats are compared. For the experimentation part, the processing framework is not the native of Hadoop (MapReduce), but a different one, called Spark [14], is used.

Z. Baranowski [15] makes a comparison including more than just two formats. In this case, there is a comparison between different ways of storing data, and not just file formats, including Hadoop databases like HBase[16]. However, the focus is made on the experimental comparison, not making a big effort in the theoretical part.

[1] Commodity hardware refers to devices or device components that are easy to obtain, widely available, and not expensive. Usually they are easily interchangeable with oher hardware of its type.

Apart from the previously mentioned works, there have been several presentations [17, 18] and some blog posts [19]. In this works the amount of file formats considered is larger and almost all the most relevant formats are taken into account. Nevertheless, there is a very little theoretical base to compare the formats between them without using numbers, and almost all the importance of the content is focused on the experimental results.

With all this information, we can conclude that the trend is to take one or more datasets and try to make an experimental comparison between the formats. But the majority of these studies either are not exhaustive when selecting the compared file formats, or do not make a theoretical analysis that can be reflected the experimental results

As we have previously mentioned, depending on the data and the future use of that data, a format can be better or worse for a particular purpose than the other. Thus, we think that a theoretical comparison is very important, because with the theory is feasible to have a general idea of which formats are better for any possible situation. By making an experimental comparison only that specific scenario (or another very similar) is taken into account.

1.2 Objectives and Results of the Work

In this work we want to cover that lack of a theoretical comparison between all the most important formats. To reach this goal, we have designed a comparison framework, including some of the most important aspects to take into account in a Hadoop environment.

For this purpose we formulate the following research hypothesis:

Hypothesis *It is possible to figure out which file formats have a better performance in a specific situation of data structure and queries.*

However, we think that the experimental part is still very important. For this reason, we have also done an experimental comparison using two very different datasets and a set of basic queries. With this experimentation we can enrich the theoretical comparison with some conclusions, useful for similar scenarios to the one used, and test if the theory is reflected.

To test the hypothesis, there are two main objectives in this project:

- To make a **theoretical comparison** of the formats using a proposed framework. This framework has to cover the most important aspects for HDFS, that can affect the performance. Apart from the comparison, it is also important to make a detailed description of each format characteristics, as they can add new information that can affect the performance.

- To make an **experimental comparison** of that formats with two different datasets. With this second objective we pretend to verify if the conclusions made in the theoretical framework can predict the results obtained in this experimental comparison. Also, it is possible to enrich the theory.

1.3 Methodology

Due to the two objectives that this project has, we can distinguish between two very differentiated methodologies, one qualitative and another quantitative. According to Guba and Lincoln [20], this includes the two possible paradigms of a research methodology, calling the qualitative methodology *naturalistic* paradigm, and the quantitative *rationalistic* paradigm.

The first objective of creating a theoretical framework involves the qualitative methodology. This means that it is not considered a specific scenario, just the characteristics of each format, independently of the type of data stored. The focus is to be able to have a theoretical idea of each format and the differences between them. For this goal, it is very important to make an effort studying the specific particularities of each file format, and in which aspects they differ from the rest.

This comparison can be considered the main goal of the project, as it allows to have a global idea of which format will work better with any specific data and queries. This is useful for every situation, and allows the data scientists to know which format can have better performance on their datasets and their data needs.

After this first theoretical part, we have proposed an experimental environment (quantitative methodology) to test the file formats. Despite we can theoretically suspect which is the better format for a known scenario, in some cases the decision is not so easy and it is possible to have doubts. Thus, making an experimental study, the general knowledge of the formats can be enriched, deducing more particularities of each format, or discovering how these particularities affect the performance in similar scenarios than the ones used.

This second approach is based on the rationalistic paradigm. This means that we will focus on the study of particular scenarios, trying to make generalizations from that specific situations. These generalizations could be less useful than the naturalistic paradigm in some situations, because it is possible to have very different data or queries to the ones used.

With all this, we can propose a research question, that we try to solve in this project. The question would be, ***it is possible to figure out which format is better knowing the scenario?***. This is a very general question that we have divided into two sub-questions, each one related with one of the two objectives. *It is possible to make a theoretical framework to specify the differences between each format?* And, *it is possible to make a generalization of the file formats performance by making experiments in specific scenarios?*.

1.4 Document structure

The present work is divided into 6 different chapters. The first one is the introduction that we have already made.

In Chapter 2 we make a review of the general aspects of a distributed file system, the most typical architectures and how the fault tolerance is managed.

Once it is explained what is a distributed file system, we have to study the Hadoop storage system (HDFS). To do it, we first study a former file system by Google (called the Google File System, GFS), which forms the basis of HDFS. By having a general idea on GFS, is easy to understand the HDFS architecture, how the read and write operations

are done, and how the cluster robustness is managed. All these things are explained in Chapter 3.

Up to this point, all the necessary previous theory is explained, so we focus on the file formats comparison. Firstly, the theoretical comparison is done in Chapter 4. In this chapter all the considered file formats are explained, and at the end of the chapter we summarize their similarities and differences with a proposed framework.

After the theoretical comparison, we present the experimentation part in Chapter 5, where we explain the experimental framework used, including the datasets and queries, as well as the characteristics of our cluster. Once the environment is described, the obtained results are shown.

Finally, we present a summary of all the conclusions that we have obtained along the whole project, including the future work, in Chapter 6.

Chapter 2

Distributed file systems

As we have mentioned at the beginning of Chapter 1.4, the distributed file systems are a solution to the problems derived from the Big Data volume.

A **distributed file system (DFS)** [21] is just a set of remote computers that acts as a single file system. It has advantages, such as the opportunity of easily sharing files, or the scalability capacity by just adding new computers to the system. As a file system, it has to be able to store data and to allow users to retrieve it. A DFS must provide the same operations as a traditional local file system. The primitive operations of a file system are read, write, modify and delete files.

In this chapter we are going to explain the main characteristics of DFSs, and how they work. First, we define some basic concepts related with distributed file systems. After that, in the following section, we will detail their characteristics, including transparency, scalability, remote access, multiple access, naming and fault tolerance. Once we know the main characteristics of the DFS's, we will explain the principal types of architectures they can have, distinguishing between centralized and not centralized architectures. Finally, we focus in one of the main characteristic in a DFS, the fault tolerance.

2.1 Basic concepts

In distributed file systems we can find three main actors, a client, a server and a service. A **service** is a software running on one or more machines, that provides some functionality to a client. The **client** is a process that requests a service to a server. The **server** is a software running in one machine that receives requests from clients and orders the execution of services. Using this terminology we can say that a DFS provides file services to clients. Each machine of the system is usually called **node**. These nodes are the ones responsible of storing the files. A file may not be stored only in one node, it can be replicated, split into different parts, or both things.

The file services are the primitive operations mentioned before (read, write, modify and delete), applied to the files stored in the nodes. Depending on where the files are stored we can talk about local resources or remote resources. A **local resource** means that the files needed to satisfy the client request are stored in the same machine where the service is running. If that is not the case, and the files are stored in another one, we talk about **remote resources**. When dealing with remote resources it is necessary to move the

files between nodes, and this transport means that the performance is worse.

Usually the clients do not know where the remote resources are stored. For this task there is a **namespace**, that is responsible of linking the names of the files with their data location. It can be a file with a direct description of the location for each file name, but there are also different ways to manage the namespace, as we will see later.

2.2 Characteristics

2.2.1 Transparency

We have defined a DFS as a system acting like a conventional, centralized file system. Thus, the multiplicity and dispersion of servers and storage devices have to be transparent to clients.

We can distinguish different types of transparency in a distributed file system. The main one is the **network transparency**, which means that the interaction of clients is the same as in a local file system. The clients do not need to know the details of where the files are stored or even that there are multiple machines

In this dimension there are included the access transparency and location transparency. Access transparency means that resources are requested by clients in a uniform, single way, regardless of how the file access has to be performed on each node. The location transparency allows the clients to be unaware of where the file is stored. Thus, the request made by a client will be always the same, independently of in which machine the file is stored. It is worth noting that there is a strong dependency between these two transparency properties.

The other transparency type is the **user mobility**. The client can access the DFS with any machine belonging to the system. The clients will have the same interaction regardless of the machine they are accessing with.

In addition, all the other problems resulting from the distribution, such as multiple access to a file, or the machine or communication faults, are completely transparent to the clients. All these problems are managed by the DFS, and therefore, the client does not need to take care of them.

2.2.2 Naming

The naming refers to the mapping between the name of the files and their data location. It is the way the DFS knows which data correspond to a certain name. For this task, it is used the namespace. There are different approaches to implement it. Here we are going to explain three of them.

The first technique, and the easiest one, is to directly omit the namespace. With this approach the file name explicitly indicates the location of the data, usually using the syntax *"host:local_file_direction"*. This technique has one main disadvantage, and it is that the location transparency is lost. Thus, we do not consider this approach valid for DFSs, as we have considered the characteristic of transparency as mandatory.

The second one is based on linking directories in each individual machine. The idea is the same as the *mount* command in UNIX. Once a directory is attached locally in a

machine, its files can be named in a local way. Using this approach we can found that the shared namespace is not the same for all the machines. Usually, this fact is perceived as a serious problem for DFSs.

The last technique is to use a single namespace stored in a file. Every machine of the system must have access to this file. The idea is to make a query to this global namespace to find the direction of a resource, knowing its name; just like in a conventional file system. The main problem of this approach is to make visible the same namespace for every node. One possibility is to maintain this whole namespace in one specific machine, and query there whenever it is needed, like in a master-slave architecture.

An alternative for the technique of storing the namespace in a single machine, is to replicate the namespace file to every node of the DFS. With this approach the main problem is to maintain all the replicas updated, and take decisions when some nodes have different values in their namespace.

This last technique has a variation that can avoid problems when the namespace grows too much. It consist of having different levels. The idea is to split the namespace file into parts and store them in different nodes. To know where these parts are stored, a "father" namespace file is created for pointing to the location of the original namespace file parts.

2.2.3 Scalability

One of the main reasons for the creation of DFSs is their scalability, as they can store much more and compute much faster than a single machine.

The **storage scalability** refers to the capacity of storing enormous amounts of data. This is possible because the distributed file system has lot of machines, having each one some storage capacity. Even if the data amount grows enough to take up all the storage capacity, it is always possible to add new machines to the system, adding their storage capacity to the distributed file system.

The only problem regarding storage scalability will appear, in some cases, when managing the namespace. When there is one single namespace file for all the system (third technique explained in previous section) and the number of files grows too much, the namespace will also grow. A very big namespace may be difficult to manage if it is stored only in one machine, or the queries to it can be slow because of its large volume. A solution to this problem is the variation of a "father namespace".

The other type of scalability is the **computational scalability**. The computation may not be a typical problem of file systems (it is more related with higher levels like data analytics). However, the reading and writing tasks can be considered computation. If it is requested to read or write a very big file, the time of this request can be very high using a single machine. But if we spread the file into different machines, it would be possible to read or write in parallel, reducing a lot the time taken to finish the task.

This parallelization is possible if these large files are spread into different machines. This spread makes more difficult the task of namespace, as one file has different directions, and can cause a great increase on the namespace size.

2.2.4 Remote Multiple Access

The other main characteristic of the DFSs is to have the possibility to access from multiple points and the ability of easily sharing files.

Having **remote access** to the file system means that it is not necessary to be physically near the data to access it. When making a request (i.e., read or write) to a remote file, firstly it is necessary to find which machine stores it (using the namespace). When the machine storing the file (server) is known, a data transfer between the client and the server is done, so the request can be satisfied.

There is one alternative to the transfer of the whole file from the server to the client. It consists in calling a *remote service*. Instead of transferring the complete file, a preprocessing in the server is made. This is possible when the requested data is not the entire file; for example when only some records of the file are requested. The idea is to extract only the necessary data of the file and send it to the client.

Regardless of this possibility it is very important to make an appropriate usage of the cache memories. It is possible to avoid lot of file readings from disks using the caches memories, because that files can has being accessed before and the needed data can be still stored locally in the cache. Thus, a reading from disk is avoided and the request is answered quicker since the cache data retrieving is faster.

As well as being able to access from any machine of the system, more than one client or machine can access to the same file at the same time (**multiple access**). This can become a problem when some of these accesses modify the file.

In the case of two simultaneous writings, the first client's modifications done can be lost, because the second one does not take into account the previous changes. And if one client reads a file just before it has been modified by another one it will be working with the file *out-of-date* without the modifications.

These problems can be solved using permissions and blocks. Thus, two or more different processes can read the same file, but if one is writing, the file is blocked and no one can read or write on it until the first process finish the writing. These permissions and blocks can be more or less restrictives, depending on the system necessities.

2.3 Architectures

We have explained what is a DFS and its characteristics. However there are many ways in which a file system can be built, depending on how its architecture its organized. We are going to distinguish two main groups of architectures, the centralized and the decentralized. These architectures refer only to the server side, the clients just access to these systems.

The **centralized architectures** are characterized by having one main node called *master*. This master node is responsible for storing all the metadata of the file system, specially the namespace, and to receive the clients requests. In this type of architecture the master does not store any file data, and the rest of nodes are the ones that do this work. The responsibility of the master is to receive the clients requests and answer them by just indicating the node where the requested data is. Thus, its main job is the naming. Once the client knows in which node is stored the data (or in which it can write the data), it

just makes the request directly to the node. The master can be also responsible of making itself the requests to the nodes, and then answer the client with the requested data; but this approach would lead to a possible overloading of the master node (because it must make much more operations), and an unnecessary moving of data (the data has to be moved from the node where it is stored to the master, and then to the client).

The two main examples of centralized architectures are the *Hadoop Distributed File System* [9] and the *Google File System* [22]. These two file systems will be explained in the next chapter.

There is an alternative inside the centralized architectures, and it lies in having more than one node acting as the master. The concept is the same as when there is one single master, but just making a *sub-distributed system* for the master. This is useful because, in some cases, the namespace can grow a lot and store it in only one node can become a bottleneck. An example of this approach is implemented in the Ceph distributed file system [23].

On the other hand, the **decentralized architectures** are the ones in which there is not a master, and all the nodes are equivalents. In this case, the main problem is where the namespace is stored, and where the client has to make the requests.

An easy option is to not store the namespace and make the user responsible for know where each file is stored. Then it is only needed to make an FTP (File Transfer Protocol) [24] connection to the node. This approach is not suitable, as the transparency is completely lost. Using our definition of DFS we can not consider this option.

The solution to this problem is to store the namespace inside the nodes. It is obvious that the complete namespace can not be stored in every single node, as it can grow large enough to be a problem of space and, what is more important, whenever the namespace changes, that changes must be updated on every node. This makes the alternative of storing the namespace in each node as not viable.

The proper way of storing the namespace inside the nodes is to store only a piece of it in each node. This may not be easy, as it is necessary to find a proper way to split the namespace and when a query is done, know to which part access. An approach of this splitting, based on connecting a directory to only one node and splitting the namespace in a tree form, is done by the Newcastle connection [25].

Finally, there is the option of not storing a namespace and know the files location by other ways, such as calculating a hash code for knowing where to store each file. This approach can be done with entire files, or even with only blocks or pieces of files (in the cases in which the files are large). One example of a DFS that uses this technique is the GlusterFS [26].

2.4 Fault tolerance

Once we know how a DFS architecture works, we have to take into account that a lot of nodes are needed in the system to take advantage of its characteristics. Having lot of machines has the benefit of a great storage scalability, but when this number grows too much, the probability of failure from any of them grows as well. At some point it is sure that the machines failures will occur. Thus, the distributed file systems must be able to maintain the files consistency even if some machines fail.

Svobodova [27] defined two file properties related with the fault tolerance: "A file is *recoverable* if it is possible to revert it to an earlier consistent state when an operation on it fails or is aborted by the client. A file will be called *robust* if it is guaranteed to survive crashes of the storage device and decays of the storage medium". A DFS should be recoverable and robust.

The most probable fault in a distributed file system is the **machines crash**, loosing all their data. As there are a lot of nodes conforming the distributed file system, the crash of any of them is very common. It is not recommended to try to avoid the device crashes, but just assume them. For solving this problem, the main solution is the creation of replicas of data in other nodes. Even the namespace must be replicated, as it can also be lost. This will make the files robust.

Other possible fault is the **corruption of files**. As the previous problem, it is commonly resolved using replicas, the corruption of files usually affects only to one of the replicas. The corruption is solved by just removing the corrupted replica and creating a new one. Another possibility is to add additional information to the files to make possible the recovery of the files to a previous stable state. This alternative is very useful when there are more than one replica corrupted and it is not possible to ensure which of the replicas is the good one. These solutions ensure the files to be robust and recoverable.

Finally, there can be another problem in a DFS, the **communication faults**. This problem occurs when it is made a request and there is no answer. It can occur because the receiver machine has crashed or because of connection problems. Whichever the problem is, there are two main solutions, to resend the request or to try to connect to another machine. The first solution is obvious, just resend and wait. If there is still no answer, the second solution is used, and the request is sent to another node. If the request is a read one, it is sent to a machine that stores a replica of the file requested; and if the request is a write one, it can be sent to any machine that can stores it; in this case the namespace must be updated.

Chapter 3

Hadoop Distributed File System

Up to now we have explained the general characteristics of a DFS. In this chapter we will focus on a specific system, the **Hadoop Distributed File System (HDFS)** [8, 9]. HDFS is a distributed file system based on a centralized architecture, in which there is one single master node and a lot of slave nodes that store the data.

It is a system based on the Google File System (GFS). Thus, many characteristics are shared for both file systems. The main shared characteristic is that they are designed to run using *commodity hardware*. Commodity hardware refers to devices that are inexpensive and easy to obtain; the main example are the personal computers (PC).

In this chapter we are going to explain the architecture of HDFS, how the reading and writing operations are done, and how it maintains the robustness of the files. But, since HDFS is strongly based on GFS, we will first explain GFS prior going to the details of HDFS. This first file system has some assumptions that HDFS has also inherit. Most of them are related with the general DFS characteristics:

- It is assumed that there will be hardware failures in the nodes. As these file systems use many of devices, it is usual that some of them fail.It is not worthy to make an effort trying to solve this problem, it is just assumed.

- As they are DFS, there will be multiple reads and writes at the same time. This means that two or more different clients can be modifying the same file simultaneously needing to manage this simultaneous accesses. Even knowing this, both file systems do not try to solve the problem of loosing data or read out-dated files; this can be managed by upper layers or tools running on top of the file system. This is a hard problem for these file systems to solve on their own, so they do not try to do it.

- The files stored in the file systems will be usually very large files, meaning by ”very large” of the order of gygabytes or even terabytes in size. This lead to two different problems, the simply fact that there are very big files difficult to store in one single node; and that there will be required a great disk storage capacity. For example, there are clusters that stores petabytes of data [28].

- It is better to move computation rather than moving data. That is, it is preferred to make the computation in a far away node that has the data locally rather than moving the data to the nearest node. This is because for moving computation there

is very few data moving throw the net, but moving the data (taking into account that files are very big) is very expensive.

- The data reads will be for large pieces of data in streaming or very short pieces of random data. The streaming read is not a problem, because the data is normally allocated contiguous. For the writings, it is supposed that the majority of them will be similar to the large streaming reads, and very few will be small writes. Once a file is written, it is very strange to modify it; the modification is possible, but as it is very uncommon it is not optimized.

3.1 GFS

It is well known that Google is one of the biggest companies around the world, and some of its products provides storing and processing via Internet (such as Google Drive). Because of the amount of users Google has, they must deal with an enormous data processing demand. A distributed file system called Google File System (GFS) [22] is the solution to this problem. GFS provides very scalable storing and processing capacities. Nowadays there are deployed many GFS clusters for different purposes. The biggest ones have over 1000 nodes and more than 300TB of disk storage.

As mentioned in the previous chapter, a distributed file system must provide sharing capacities. GFS supports simultaneous connections of clients to the same data. For concurrent reads and writes on files GFS uses *file snapshots* and atomic *record append* operations.

GFS is designed as a master-slave architecture with two types of nodes: one **master** node and many **chunkservers**. The master is only one and does not contain any data; it only has metadata, including the namespace. The chunkservers are the ones which store the data. Typically the chunkservers are Linux running a user-level server process.

Data files are divided into **chunks** of fixed size (usually 64MB), that are stored only in the chunkservers local disk. The chunks size is not necessary the size of the files or the operative system blocks. Each chunk has an identifier called chunk handler, of 64 bit size. GFS assumes that hardware fails are common, so it is possible that a chunkserver fails and loose all its data. Because of this possibility every chunk must have different **replicas**, typically 3. A replica is a copy of the chunk that is stored in a different node than the original chunk or any other replica. The replicas must be updated if the chunk data changes.

As GFS architecture is master-slave, all the requests go to the master, and there is only one. Because of this, it is important to give the less work as possible to this node. For example, the master never does read or write tasks. For this read and write operations the client requests to the master the identity of the chunkservers that store the data (read) or where the data can be written (write); so the master just need to access the namespace and answer with the direction of the corresponding nodes.

As we have mentioned before, the master node only contains metadata of the file system. There are three types of metadata inside the master: the namespace and file-to-chunk mapping, chunkservers status, and a log. The **namespace** contains the information about where each file, and its chunks, is allocated to make possible the reads. It is necessary to

know where the file is to order the chunkserver to respond the read task. This namespace is stored in memory, so the master can locate the chunkservers as quick as possible, without reading from disk. Because of this in memory storing, the size of the namespace is limited by the master's memory size (once the memory is full it is not possible to store more namespace information). This is not considered a big problem because of the facility to compress this type of metadata.

In the namespace metadata the master must store some specific information about chunks, in order to provide data consistency. For each file, it is saved the information of every chunk and in which chunkserver it is stored, in the order of the replicas. Each chunk has a version number (if two chunks of the same data have different version number, then something is wrong). For chunks modification (or chunk mutation) all the changes done are temporally saved in an ordered list, and these modifications are applied to all the replicas in the same order.

The master node also stores the **chunkservers status**. This job is done using *heartbeat* messages every few minutes. Each chunkserver sends a heartbeat message periodically to the master node. With this messages, the master can detect when a chunkserver fails (if it does not receive heartbeats from that node). This data is also stored in main memory, and there is no size problems because it is usually very little data and has a small storage volume.

The **log** is the only persistent data stored in the master node. Here it is stored all the critical metadata changes, so it is possible to know all changes done and, if necessary, make a metadata backup. The read and write operations are not stored in this log, only the metadata changes.

Finally, to verify the **data consistency**, in a similar way as the heartbeats, there is a periodically handshaking process with the master for every chunkserver. In his handshake the master can check if the chunks are synchronized or not (if every chunk of the replicas makes the same handshake). If the handshaking determines that the data is not consistent it is necessary to solve it. In the case of not being able to solve the data consistency, it is made a backup to a consistent previous state (just in extreme situations).

3.2 HDFS Architecture

Once we know how the architecture of GFS is, it will be easy to understand HDFS's architecture [29], since it is strongly based on the distributed file system of Google.

HDFS has a master-slave architecture. There is a single *Namenode*, that stores the file system namespace and receives clients requests; and a large number of *Datanodes*. As in GFS, the Namenode (master) does not store the files data, and the Datanodes are the ones in which the files are stored. The machines typically run a GNU/Linux operating system.

The **Datanodes** are responsible for storing the data. The same way as the chunks of GFS, this data is divided into blocks, by default of 128MB. Originally the block size was of 64MB, like in GFS, but from Hadoop 2.7.3 it was changed to 128MB. This large block size is normally used because of the assumption of large reads of contiguous data, favoring the seek time, but slowing small reads, as the blocks are read entire. Also this large block size is because of the initial assumption of having very large files in the file system, and very few small ones. And what is more, one of the main reasons of enlarging

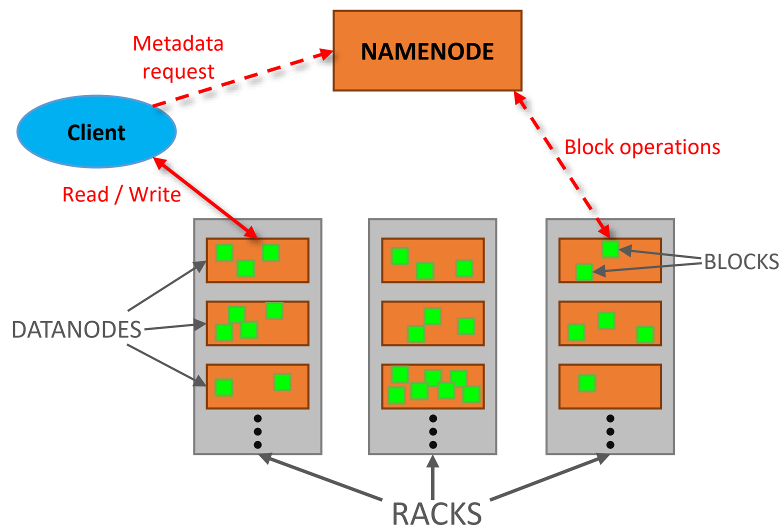


Figure 3.1: HDFS architecture

the block size is to improve the performance of the Namenode, as it has to manage a smaller number of blocks.

As in the GFS architecture, the blocks must be replicated on different Datanodes (by default 3 replicas per block). This is done because of the high probability of hardware failure in the nodes. Note that this replication is only useful for Datanodes; we will see what happens when the Namenode fails in section 3.4.3.

The HDFS namespace is designed as a hierarchy of files and directories. These files and directories are represented in the Namenode by *inodes*, which record attributes like permissions, modification and access times, and disk space quotas. The Namenode maintains the namespace tree and the mapping between file blocks and Datanodes (the physical location of file data).

The namespace in HDFS is stored entirely in main memory, and it is only allocated in the Namenode. This means that the namespace, and therefore the number of files, is limited by the amount of memory the Namenode has. The inodes data and the list of the blocks belonging to each file are called the *image*. The image is saved on persistent storage in the Namenode periodically. This copy of the image is called *checkpoint*.

The Namenode also stores a modification log of the image, called the *journal*, in its local disk. The location where the block replicas are stored is not part of the image, because this data is updated on real time by heartbeats, sent from the Datanodes to the Namenode periodically.

Apart from the Datanodes and the Namenode it is important to notice that there are **HDFS clients** that request reading and writing operations. Again like in GFS, the Namenode does not deal with read and write requests from the clients. Instead of reading and writing, it only answers in which Datanodes the data is stored (in a reading request), or in which Datanode the client can write, including where to allocate the replicas (in writing request).

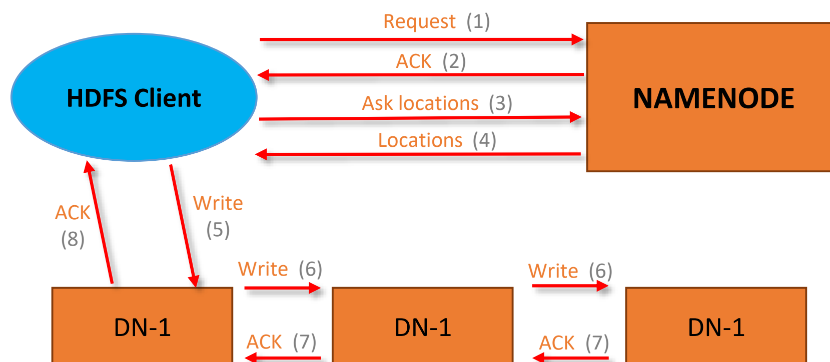


Figure 3.2: HDFS writing operation

An overview of the architectures of HDFS is shown in Figure 3.1. It is important to remember again that the clients directly request readings and writings to the Datanodes freeing the Namenode from that responsibility. Although in the figure the master is out of any rack, it can be inside one; there is no restriction on where it must be located.

3.3 HDFS Writing and Reading

If we look carefully to Figure 3.1 we can see that the client is the one in charge of requesting the writing and reading tasks. As mentioned in the previous section, the Namenode is not able to do these tasks and the Datanodes are the ones that receive the reading and writing requests.

The **writing** task is illustrated in Figure 3.2. First, the client sends a message to the Namenode requesting the writing of a new file (1). The Namenode performs various checks to ensure that the file does not already exist, and that the client has permissions to write; if it does not have permissions or the file exists, the writing process fails. If this step does not fail, the Namenode sends an acknowledgement message to the client (2). The Namenode makes a record of the new file with a new *inode*.

Once the client has the approval of the Namenode, it splits the file into blocks and, for each one, requests to the Namenode for the directions of the Datanodes where the blocks can be written (3). This request is made to the Namenode, because it is stored there the location and information about the Datanodes and how many blocks they are storing, and the client does not know where is better to write the data.

When the Namenode sends to the client the direction of all the Datanodes where the data will be stored (one Datanode for each replica of the block) (4), the client sends a writing request to the nearest node (5), including the direction of the rest of the nodes where the replicas will be stored. This first Datanode saves the block in its local disk and sends a writing request to the next Datanode of the list to store the next replica (6). This process is repeated with the next Datanode until all the replicas has been saved in the Datanodes. When the last Datanode stores the last block replica, it sends an acknowledgement message to the previous node (7), and this one another to the previous, and so on. When the first Datanode receives its acknowledgement message, it sends another one to the client

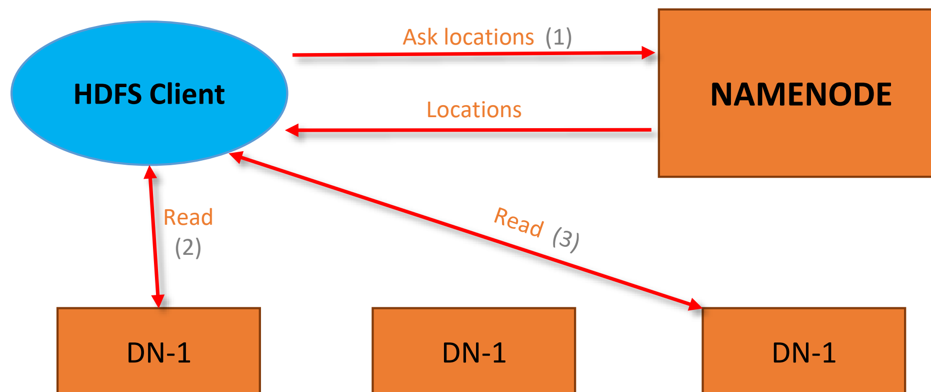


Figure 3.3: HDFS reading operation

to inform that the writing process has been done successfully (8), and the writing task is finished. Steps (3) to (8) are repeated for every block of the file.

The **reading** task is easier than the writing one. It is shown an example of a 2 blocks file reading in figure 3.3. As in the writing operation, the first step is a request from the client to the Namenode asking for which are the blocks of the file, and where they are stored (1). The Namenode sends the location of the blocks (including all the replicas) to the client.

Once the client has which blocks it needs and where to find them, it sends a reading request to the nearest Datanode that contains a replica of the first block (2). When the client receives the data of the first block, it repeats the same process with the next blocks in order (3).

A reading or writing operation can fail, because of multiple reasons. What is done in these situations is explained in Section 3.4.2.

3.4 HDFS Blocks Robustness

Knowing that Hadoop is built assuming that there will be hardware failures, it is very important to make sure that the data will be always available, even if any node fails. The main solution to this problem is to replicate everything.

3.4.1 Replication

As explained in the previous section, HDFS uses blocks to store the data files, and these blocks are equivalent to the chunks of GFS, but instead of being of 64MB, by default they are of 128MB. The files are broken into pieces of this block size and spread throw the cluster. But we know that a node storing a block can fail. Because of this, the data blocks must be replicated; typically each block has 3 replicas (as in GFS).

The Namenode makes all decisions regarding replication of blocks. The replication philosophy in HDFS is to store the replicas the nearest as possible (trying to reduce the data flow through the cluster), but guaranteeing the fewer replica loss when a hardware

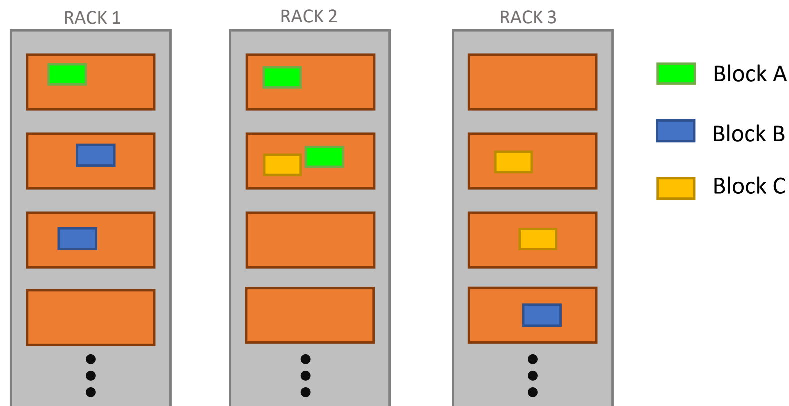


Figure 3.4: HDFS replication example

failure occur. By default, the first replica is stored in the client node (the Datanode where the client is), as it is the nearest one; the second one in a different Datanode of the same rack; and the last one in a Datanode of a different rack. Thus, if a entire rack fails, there will be at least one replica available. If there are additional replicas they are randomly placed. For example, in Figure 3.4 it is shown how three different blocks would be stored using 3 replicas each.

The Namenode must know the location of all the replicas, but it does not ask for this directly to the Datanodes. The Datanodes are the ones that send periodically *heartbeat* messages to the Namenode with the information of the blocks they are storing. If one Datanode does not send any heartbeat for a long time, the Namenode assumes that that Datanode has fail and orders to write another replica of the blocks that the broken node was storing.

The heartbeat messages do not only contain which blocks the Datanodes are storing, they also include a checksum for each block, to guarantee that they are not corrupted. If the Namenode discovers that one checksum is different to the other replicas ones, it orders to the Datanode with the wrong checksum to delete the block, and orders to write another replica of the block (not necessarily in the same Datanode).

If one Datanode has several wrong checksums in its blocks it is considered as broken and it is sent a shut down order to it. In this case, all the blocks of the Datanode are rewritten in other nodes, using their replicas to not rewrite corrupted blocks.

3.4.2 Reading/writing Failure

As we have seen in Section 3.3, the read and write operations can fail, usually because a Datanode does not answer. **Reading failures** occur when requesting one block from a Datanode, and it does not answer. This has an easy solution. Just read the block from another Datanode that contains a replica of the block. If the Datanode is broken, the Namenode will notice this when it does not receive any heartbeat message from that node. The client does not need to report anything to the Namenode, because the problem could be a network failure, and does not mean that he Datanode is broken.

The **writing failure** is not so easy. When a Datanode fails, it is just skipped and the

block is only written into the rest of Datanodes. In this case, the block that is being written gets a new identity, and this new identity is notified to the Namenode, so it can change the block identity in the namespace. With this new identity, if the failed Datanode has really stored the block (it is not broken, but a connection problem with the acknowledgement happened), the Namenode will notice this when it receives the corresponding heartbeat message. When this happens, the block with old identity will be ordered to be deleted.

3.4.3 Namenode Failure

We have seen what happens when a Datanode fails. But what happens if it is the Namenode the one which fails? It is possible to restore a Namenode by using another node. This auxiliary Namenode is called **Backup node**. This Backup node receives a real time copy of the checkpoint and journal mentioned in Section 3.2, and they are stored into persistent memory (disk). This real time copy is not very difficult to store when all the changes done by the Namenode are notified to de Backup node.

Using the checkpoints and journal it is possible to restore the Namenode by copying to main memory the checkpoint to restore the image and applying the changes stored in the journal. The replicas placement of the blocks are restored easily because the Datanodes will be sending the blocks they have, periodically, using the heartbeats.

Even this last part of restoring the location of the blocks can be avoided by sending the heartbeats both to the Namenode and to the Backup node. This means that the Datanodes and the clients send their messages to the Namenode and to the Backup node. Thus, the Backup node will have the same data as the Namenode, and when the Namenode fails, the Datanodes and clients can continue acting the same way without knowing that the Namenode has fail (it is transparent for them).

Chapter 4

HDFS storage formats

Because of HDFS characteristics, new formats for storing files have been developed, trying to use the distributed file system features in their advantage. The main facts to take into account are the large size of the files, and that they are split into blocks spread around different nodes.

These file formats are designed to store structured data that can be described with a **schema** (more or less complex). The storage of unstructured data (natural language) is not so common and has been less studied in this environment. Thus, we can assume that, in these file formats, data can be described with rows (or records) and columns. The mentioned schema describes the structure of the rows. One simple example of this type of data is a plain table, but it can be more complex, such as data with nested structures.

Knowing that structured data can be described as rows and columns, there are two possibilities when storing it; writing one row after another, or store each complete column together. The traditional way is storing by rows, and the formats that do this are classified as **row formats**. However, in many cases better reading performance can be obtained if the data is organized in a **columnar format**, where the values of the columns are stored together. A columnar format can read just the required columns without passing through the unnecessary ones. And what is more, storing by columns allows to have together values of the same type, that probably are homogeneous; thus, is easier to have a more efficient storage and get better results with compression. Nevertheless, it has some disadvantages, like having difficulties for adding new records to the file after it has been written, or taking a lot of time for a complete scan of the data (data scanning is usually required by rows).

Each format storage orientation can work better than the other one depending on the situation, being very good and fast for some operations and very slow for others. For a medium term there is a mixed possibility, that consists of making a split of the file by rows, and for each group of rows make a columnar storage. This alternative takes advantage of the columnar format inside each row group, and lowers its problems due to the row splits. Because of this balanced performance, this mixed approach is the one used by the column-oriented formats in HDFS.

In this chapter we are going to study the main file formats used in HDFS. After explaining each one we will compare them. To make this comparison, we have designed a **framework** with some important features for HDFS formats. This framework includes the following aspects:

- Storage orientation. The orientation refers to the type of format, row or columnar.
- Splitability. If the format allows to split the file into different blocks.
- Compression. The types of compression that the file format allows. In Hadoop the most common compression codecs are Gzip [30], Deflate [31], Bzip2 [32], Snappy[33] and LZO [34].
- Human readable. This property refers to the formats that store the data in a way that humans can read it. The opposite is a binary serialization, not possible to read for humans, but readable for machines. The binary serialization has the advantage that can avoid unnecessary text, having less storage volume for the file.
- Schema storage. As we have mentioned, the structured data can be described with a schema. Some of the formats explicitly stores this schema, but some others do not, and the client must know the schema for making queries.
- Schema evolution. The schema does not need to be fixed. It can change over time. For some formats it is possible to change the schema. With the changed schema new records can be added with the new row structure. This possibility of changing the schema is called *schema evolution*.
- Complex data types. We have mentioned that the schema can be more or less complex. The easiest schema would be one with all primitive type values (numeric types, string, boolean and byte), and one more complicated would be one with complex data in a column (inner structs, arrays,...). There are formats that natively can store data with complex types; others can not.
- Data append. This characteristic refers to the possibility of adding new data to the file without rewriting it entirely. Updating or removing records is not taken into account because in this environment, to modify the data is usually done by processing the old one and creating a complete new data.
- Random access. Some queries to the data just need to access to some specific values, and some formats allow to know in which blocks that values can be. Thus, it is avoided reading unnecessary blocks.
- MapReduce compatibility. As MapReduce is the original computation framework of Hadoop, it is important for the file formats to be able to use them with this framework. There are more computation frameworks that can be used in Hadoop, but MapReduce is the most important and the most used one.

At the end of the chapter there is a final table to compare all the formats with this framework.

4.1 Text files

The most used file formats for every platform are the text files. These text format files are the ones that store the data in a human readable way. In this section we are going to explain two of the most popular text file formats, CSV and JSON. Both of them can be used with MapReduce easily.

4.1.1 CSV

A CSV file (comma-separated values) is a delimited text format. It uses the comma to separate the different columns of each row. A file stored with CSV can be represented in a plain table.

It is a row-oriented format, and can be easily split by dividing the file into groups of rows. Each block (row group) can be compressed with any compression codec supported by Hadoop. Some compression codecs allow splits, so the entire CSV file can be compressed and after that split it.

In some cases the schema is defined in the first row of the file, but just the names of the columns are represented (not the type). In many cases this information does not appear. Thus, we do not consider that the CSV explicitly stores the schema. The data structure can not change over the time, the number and type of the columns must be always the same.

The data stored in a CSV file must have primitive types. Complex types are not supported by default (but it is possible to represent *enum* values).

The access to the data have to be done in a sequential way, not being able to skip any block. There is no metadata to know anything about the values inside the blocks.

Despite all these disadvantages, CSV format has some advantages. Data appends are possible. Adding new rows is possible, with the only restriction of not changing the number of columns, just by adding them at the end of the file.

4.1.2 JSON

JSON is another human readable file format. It is a row-oriented, key-value text format. Again it can be easily split by making row groups. JSON files can be compressed and then split, or compress each split individually. As CSV, the compression codecs that can be used with this format are any allowed in Hadoop.

JSON files do not have any explicit schema stored, but it is possible to know the columns names as they are written in every row of the file (the key of that key-value pairs is the column name). Even the data do not have to follow a specific schema. This means that the *schema evolution* is possible without any restriction. But this has the disadvantage that the data can loose the structure and be very heterogeneous (there can be different records with no similarities in their structure). To add new records, they are just added at the end of the file.

JSON allows any data type, including complex data like structs or arrays. However, there are no metadata in these files. Thus, the access to specific records or values has to be doing a sequential read. There is no possibility of skipping unnecessary blocks.

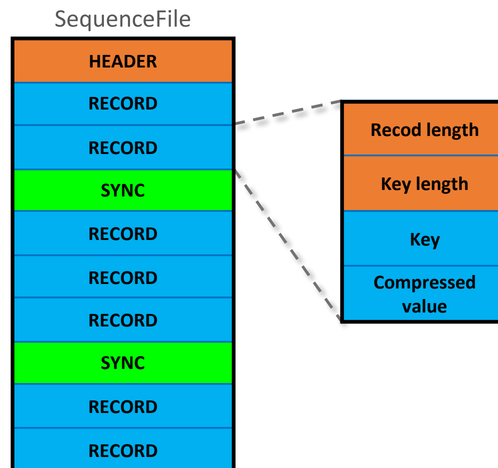


Figure 4.1: SequenceFile record compression

4.2 SequenceFiles

A SequenceFile [35, 36] is a flat file format based on key-value pairs, with one key and value per row. One of the main uses of SequenceFiles is to put small files into a larger single file, using one key-value pair to store each small file (key the file name, and value the file content). For example, if there are 100,000 files of 10KB, it is possible to store all them into one single SequenceFile. This is a big advantage as the Hadoop Namenode does not need to store the metadata of all that small files, and it only has information about the SequenceFile. Besides this use, the temporary outputs of MapReduce jobs are stored using SequenceFiles. Thus, it is obvious that it is possible to use this format with the MapReduce framework.

A SequenceFile has a header with the file information, such as the compression type (or if it is compressed), the version number, the classes of the keys and values, etc. After that header it is written the list of records, and a sync-marker every few ones. These records contains metadata about the length of the whole record and of the key, and the raw data of the key and value. This structure can be seen in Figure 4.1.

The sync marker permits seeking to a random point in a SequenceFile. This is important to make the SequenceFiles splitable. The splits can not break records.

For the compression there are two possibilities, the record compression and the block compression. The record compression just compress the value of each record, as the example of Figure 4.1 shows. The compression techniques can be any supported by Hadoop.

In the block compression a set of records are compressed into a single block. Each block stores the number of records, the length of all the keys, the compressed keys, the length of the values, and the compressed values. Thus, the block compression can be seen as something similar to a column-orientated storage, as all the keys are stored together, the same as the values. These blocks are not related with the HDFS blocks. Between each block a sync marker is written. This structure is shown in Figure 4.2.

The splitability using block compression is the same as the record one, just taking into account to not divide any block. A block must be stored in a single split.

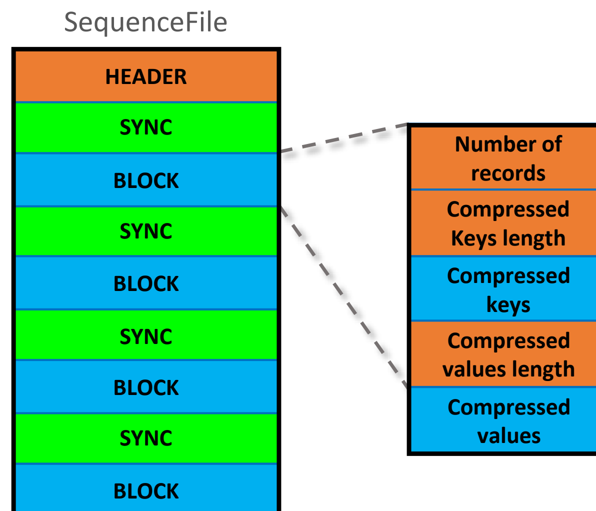


Figure 4.2: SequenceFile block compression

The block compression provides better compression ratio tha then record one. Therefore, block compression is generally preferred when using SequenceFiles.

SequenceFile is considered a row-oriented format. However,in the very special situation in which the data schema has only two columns (one stored in the key and the other in the value) and it is used with block compression, the storage would be like a columnar format. But as this is a very strange and uncommon situation, SequenceFile is just considered a row format.

Talking about a schema in SequenceFiles is not so obvious. It just stores a key and a value, if there is a structured data inside one of them, it is not known by this format. The only "schema" stored by SequenceFiles is the key and value classes. In some cases, if we are storing only a 2 columns data, this key and value classes can be acting as a complete data schema; but this is not the usual scenario. Most of the times it is necessary to know how the data is structured, and there is no schema stored in the SequenceFile. Thus, we can not consider a schema evolution, not having either a schema. Even if we consider the key and value classes as a schema, this classes can not be changed after being established. Also, we will not consider that SequenceFiles support complex data types, because the classes allowed for key and value are just simple types (such a string, bytes, integer values).

Similar as JSON and CSV files, data can be added without any problem. Any key-value pair can be written, just appending it at the end.

However, unlike text files, SequenFiles do store metadata. It is few, but there are sync markers that can be used to jump to a specific record or block. This can be useful if the keys are ordered and can be used as an index. Thus, in some very specific cases SequenceFiles can help random access queries to data. However, we will not consider that they provide random access to data, as these situations are very uncommon.

The SequenceFile is the base for other type of files such as the **MapFiles**. A MapFile

is a directory that contains two SequenceFiles, a data file and an index file. The idea is to have the data in one SequenceFile and an index for faster access in another SequenceFile. The data file contains all the key-value records; each key must be greater than, or equal, to the previous one. And the index is just a list of the keys with the starting byte position of the corresponding key. The index is loaded entirely into main memory, but if it is too large, it is possible to load only a fraction of it.

4.3 Avro

Apache Avro [37, 38] is a data serialization system, but it is also considered a file format.

Similar to JSON format, Avro is prepared to store more complex data than simply comma-separated values (CSV) or key-value pairs (SequenceFiles). It supports fields with complex types such as structs or arrays. Avro is a nice file format to represent complex data structures within a MapReduce job.

The records structure is defined using a schema, that is present at every moment. This schema is usually written in JSON and it is stored in the metadata header of the Avro files, making them self-describing. Therefore an Avro file can be divided into schema and data.

Avro was one of the first formats to include schema evolution. The schema of an Avro file can be changed to a new one, after data has been written to disk using an older version of that schema. This is very useful when appending new records, being able to add new columns easily not needing to rewrite the whole file. There are some restrictions for the schema changes, such as that data types can not be changed. The idea is to be able to add new fields when writing new records, or read old ones but without some unnecessary columns. To append new records is easy, just adding them at the end of the file.

Avro serializes the data using a binary encoding. The data is not tagged with any type information, because the writing schema is always available. The schema is required to parse data. It is possible to use a JSON serialization on the data, but it is only used for testing as it is slower and consumes more storage space. The details of the encoding are described in the Apache Avro Documentation [38].

Apart from the binary encoding, it is possible to compress the data of an Avro file. The compression codecs allowed by Avro are Deflate and Snappy. Avro files are also splittable by dividing the data into blocks (groups of rows). Each block is compressed, if selected, independently. Each one has a header including the number of records inside it and the block length. The structure of Avro is the one shown in Figure 4.3. There is a file header that contains firstly four bytes (ASCII 'O', 'b', 'j', followed by 1), the file metadata, such as the schema or the compression used, and a sync marker. After the header the blocks are stored, with their respective block header.

Although Avro stores metadata in its files, there is no information about the values of the data. Thus, it is not possible to skip blocks when just a few specific records are requested. It is not possible to know if a record fits a query or not, it is necessary to read all the blocks.

Inside the blocks the records are stored using a row orientation. This means that a entire record is stored one after another. Avro stores the rows in a similar way as JSON does, but changing the text with a binary serialization. Thus, it is possible to store complex

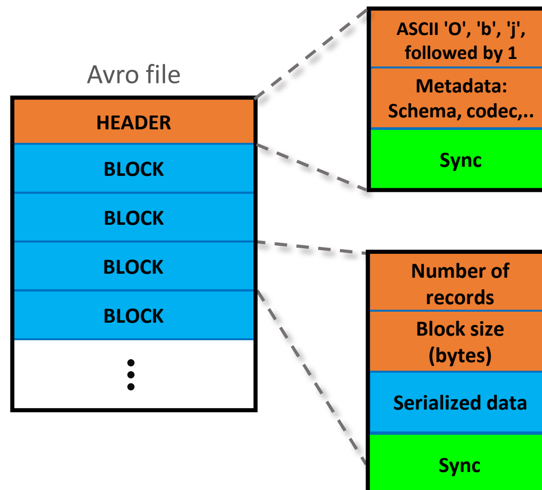


Figure 4.3: Avro structure

types, like nested structures or arrays. Avro also has the possibility of using columnar a storage by using its variation Trevni.

Trevni [39] is the columnar format of Avro. To allow scalable, distributed query evaluation, datasets are partitioned into row groups, containing distinct collections of rows. Each row group is organized in column-major order, while row groups form a row-major partitioning of the entire dataset. Each row group is a separate file, and all values of a column are stored together.

By default, each Trevni file has the same size as a HDFS block. This moderates the memory impact of the Namenode since no small files are stored.

Each column in Trevni is composed of blocks of around 64KB. The sequence of blocks is indexed, and thus it is possible a random access within the column. A header with metadata is stored at file, column and block levels.

This format is not very used, as other columnar formats are preferred, because they have been designed from the beginning for being a column-oriented format. Trevni is just an adaptation from Avro.

4.4 RCFiles

RCFile (Record Columnar File) [40] is one of the firsts column-oriented formats of the Hadoop environment, designed for systems using MapReduce jobs. It is considered a columnar format, but as Trevni does, it applies the concept of making first an horizontal partition and then storing that partitions by columns (in general, all the column-oriented formats uses this split by rows). Thus, it combines the advantages of both store types. The row partitions are the equivalent to the Trevni Blocks, so it is guaranteed that any partition is stored entirely in the same node.

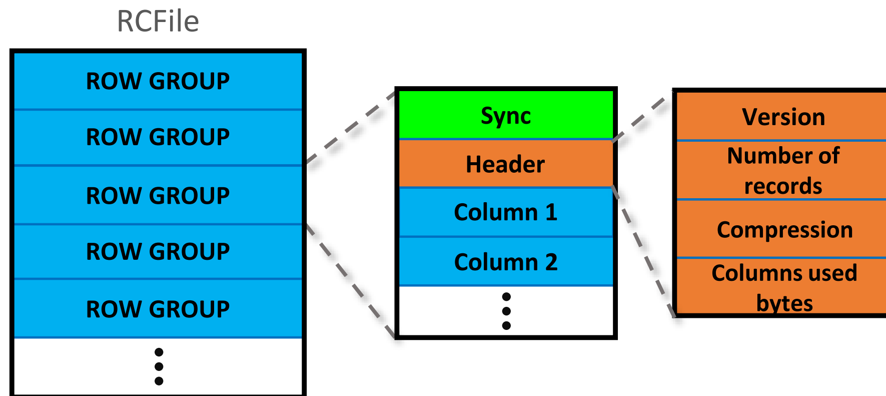


Figure 4.4: RCFile structure

As there are row partitions, it is obvious that the RCFiles are splittable. These partitions are called *row groups*. That is to say, all the records stored in RCFile are divided into row groups, all of the same size. It is possible to store just one or multiple row groups in a HDFS block.

A row group is composed of a sync marker, to separate contiguous row groups inside a HDFS block; a metadata header with the version, the number of records stored in the row group, the compression used, and how many bytes there are in each column; and finally the data of the records, stored by columns. This is shown in Figure 4.4.

RCFiles allow compression. The header and the data are compressed independently. The header is encoded using RLE (Run Length Encoding) algorithm [41]. The data part is not compressed as a whole unit. To take advantage of the columnar storage, each column is compressed independently using the Gzip algorithm. Even it is a heavy algorithm, by dividing the file into row groups makes the decompression phase faster. The Gzip algorithm can be changed by another of the ones supported by Hadoop, but by default this is de compression codec used. It is also possible to store the columns without compression, which means that they will be written in plain text.

The column concept of RCFile is very simple. Even, there is no information about the data type they are storing. All the values of a column are treated as a stream of bytes, it does not matter their original types. Complex types, like arrays or structs, are not managed by RCFiles and are stored as if they were a simple string. Thus, RCFile does not support complex type data.

Besides not knowing the original data type and store the data as a stream of bytes, there is no other information about the schema, not even the columns name; they are managed using their order. For example, the first column in one row group is always the first column in every row group.

RCFile does not allow arbitrary data writing operations. It is only possible to add records at the end of the file. When a record is added, it is firstly stored in memory. When there are enough records in memory, each column of them is written at the end of each column holder, and the file header is updated. When to write the records from memory depends on the number of records and the limit size of the memory buffer. It is to be

expected that schema evolution is not considered in RCFiles, the number of columns is fixed.

When processing a row group, RCFile does not need to fully read the whole content of the row group into memory. Rather, it only reads the metadata header and the needed columns of the row group. Thus, it can skip unnecessary columns and take advantage of column storage. However, RCFile does not decompress all the loaded columns, it uses lazy decompression. This means that a column will not be decompressed until it is sure that its data is necessary for the execution. It is useful due to the *WHERE* conditions in a query. If a condition cannot be satisfied by all the records in a row group, then RCFile does not decompress any additional column to resolve the query.

Even though, RCFiles are not the better alternative for random access queries with very few records affected, due to the fact that the entire columns in each row partition have to be decompressed and can not access directly to only one value of a column. Also, there is no metadata information about what are the values of the columns, and so it is not possible to skip row groups when searching a specific record or few records.

4.5 ORC

Apache ORC (Optimized Row Columnar) [42], as its name says, is an optimization of the RCFile format. RCFiles manages all data as bytes, whichever the data type is. Thus, it is not possible to take advantage of the type to make storage more efficient. Also, RCFiles do not allow schema evolution. And what is more, the RCFiles decompression can be very slow, as it is necessary to decompress a whole column of a row partition to access any data of it. ORC files were design to solve these handicaps. It was also developed with the idea of being efficient when using it with Hive and MapReduce.

The same as RCFile, it is a column-oriented format. But it also makes row partitions to make the file easily splittable. These row splits of ORC files are called *stripes*. Inside each stripe, the storage is columnar. The stripes are, by default, of 256MB. This size is much larger than the RCFile row groups to avoid many small reads when analyzing a whole column or a big amount of data.

A complete ORC file can be divided into stripes, a footer and a postscript, as shown in Figure 4.5. Each stripe has an index, the data and a stripe footer. The stripe index stores statistics about the values in each column of the stripe. The data are just the columns, each one divided into a set of streams. One column inside a stripe can have multiple streams. In each column there is an index entry every 10,000 rows to being able to jump to a particular row very quickly. It stores statistics for those rows, specifically, the minimum and maximum values, making easier queries requesting a specific record. Finally, there is a stripe footer that stores the location where each stream is located, and the encoding used for each column (as each column can use a different encoding depending on its type).

A the end of the file, after all the stripes, there is a file footer. This footer has the location of each stripe, the types description (the schema), and the *count*, *min*, *max* and *sum* statistics for every column across the whole file. The columns types can be complex, allowing struct, list, map and union types.

All the stripes and the footer can be compressed, and to know which is the compression technique used there is a postscript, stored after the footer. This postscript contains the

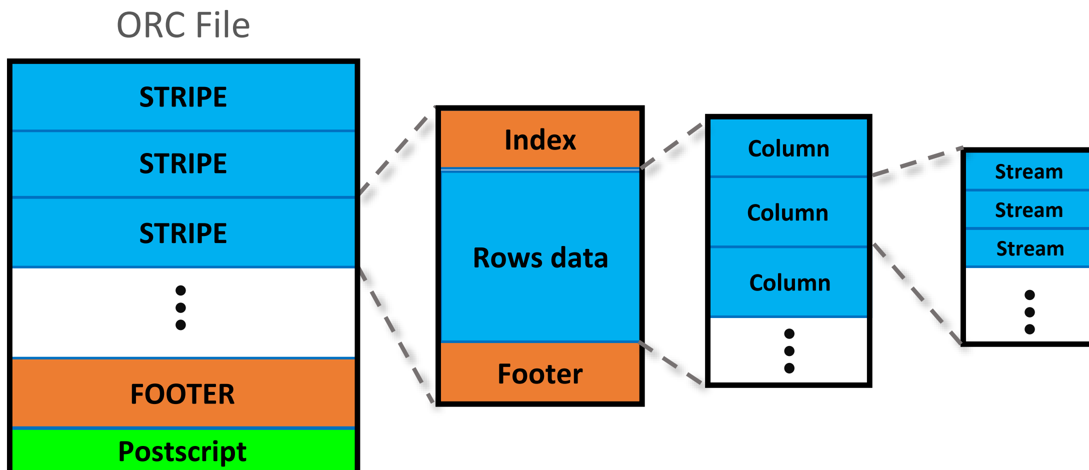


Figure 4.5: ORC structure

compression codec used and the compressed footer size. The compression codec used can be Snappy, Zlib [43] or none. Every part of the ORC file except for the Postscript is compressed with that codec.

In addition to this generic file compression, there is a lightweight compression over each stream. This compression depends on the data types, and can be configured. Thus, it is possible to use different compression techniques on integer columns and string ones, for example, to improve the compression efficiency.

Apart from the lightweight compression, each column is serialized according to its type. For example, the integer columns are serialized using RLE; string columns use dictionaries; struct columns are not directly serialized and creates a child column for each field of the struct; the list columns have one child column to store all the list values. This serialization is made at stream level, storing the data in a non human-readable way. Over this serialization, it is made the lightweight compression depending on the column data type.

For it last versions, ORC allows some *schema evolution*, such as deleting columns, reorder them, or change some columns types. Adding columns is possible, but by appending the new columns data at the end of the file. However, this format was developed thinking on a write-once file format, so edits were implemented by using auxiliary files where insert, update, and delete operations are recorded. Adding new records is also possible using the same technique used in RCFiles, store first the records in memory, and when there are several records add them to the file with a new stripe.

4.6 Parquet

The previous seen columnar formats where developed thinking only in the MapReduce framework, but there are plenty more tools and computation frameworks to query data in the Hadoop environment. Apache Parquet [44] is another columnar format that has as one

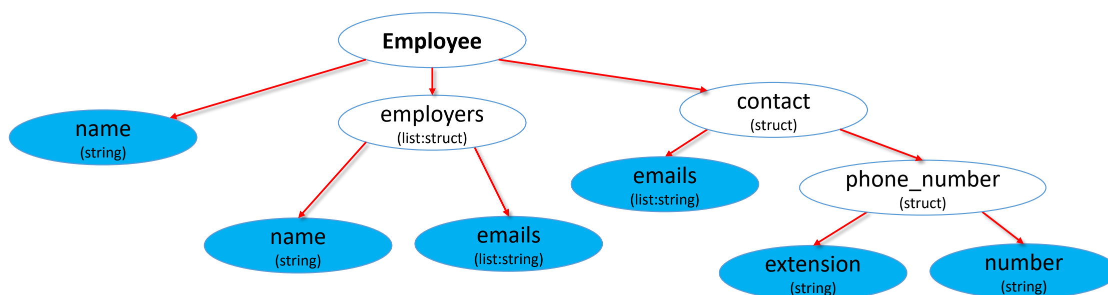


Figure 4.6: Parquet schema example

of its main characteristics the interoperability. Parquet can store nested data structures in a flat columnar format by using the technique explained on the Dremel paper from Google [45].

Parquet is **language independent**, so it can be used both by Java and C++. And what is more, there are converters to a lot of tools such as Hive, Hbase, Impala, Pig, etc. These converters are just to pass from the Parquet format to the tool models and vice versa.

Parquet is an excellent format to store nested data, meaning by nested data that which has not a flat schema and has complex types such as lists or structs. This nested schema can be represented graphically as a tree. The columns that are finally stored are the same as the leaf nodes of that tree. It does not matter if a leaf node is representing a list type field. In the example of Figure 4.6 the columns would be *name*, *employers.name*, *employers.emails*, *contact.emails*, *contact.phone_number.extension* and *contact.phone_number.number*.

With this representation there are two main problems, what happens when there are nested lists? (a list of lists), and what if a field has *NULL* value and it is not a leaf node? (for example, using the schema of the Figure 4.6, we can a have *contact* with a *NULL* value).

For both situations the solution is to store two integers for each value, called *definition level* and *repetition level* [46].

The **definition level** indicates at which level of the schema the *NULL* value occurs. It can take values from 0 up to the maximum level of the column. To explain it is easier to use the example of Figure 4.6, with the *contact.phone_number.number* column. If the *NULL* value is at *contact*, the definition level will be 0, if the *NULL* value is at *contact.phone_number* the definition level will be 1, and if the *NULL* value is *contact.phone_number.number* it will be 2. If the value is not *NULL* the definition level will be 3 (the maximum level of the column). Thus, it is possible to know at which level the *NULL* field is. This example is shown in Figure 4.7.

And for the other case, in which there are nested lists the repetition level indicates when to start a new list and at which level. Suppose we have two nested lists (level1 that contains level2). A 0 at the repetition level indicates the first element of level1 and a new level2 list (it represent the first element of a record). A value of 1 marks a new element of level1, and a new level2 list. And finally, if the repetition level is 2 means that is just a new element of the level2 list. This is extensible for any level of nesting. This example with *employers.emails* column is shown in Figure 4.8 using *employers* as level1 list and

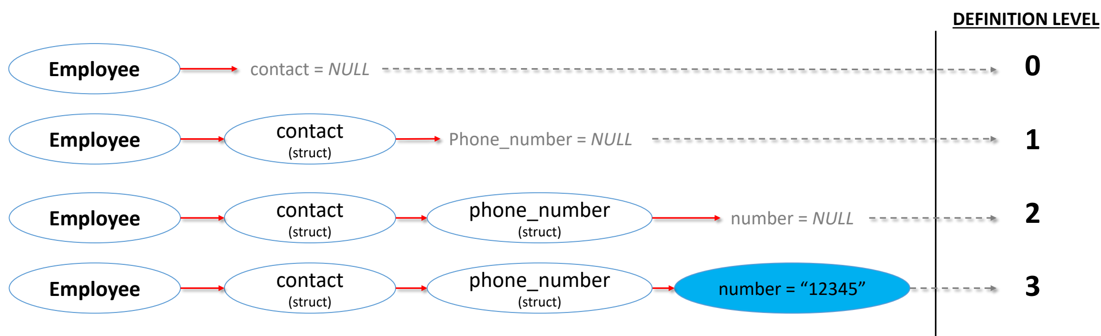


Figure 4.7: Parquet definition level example

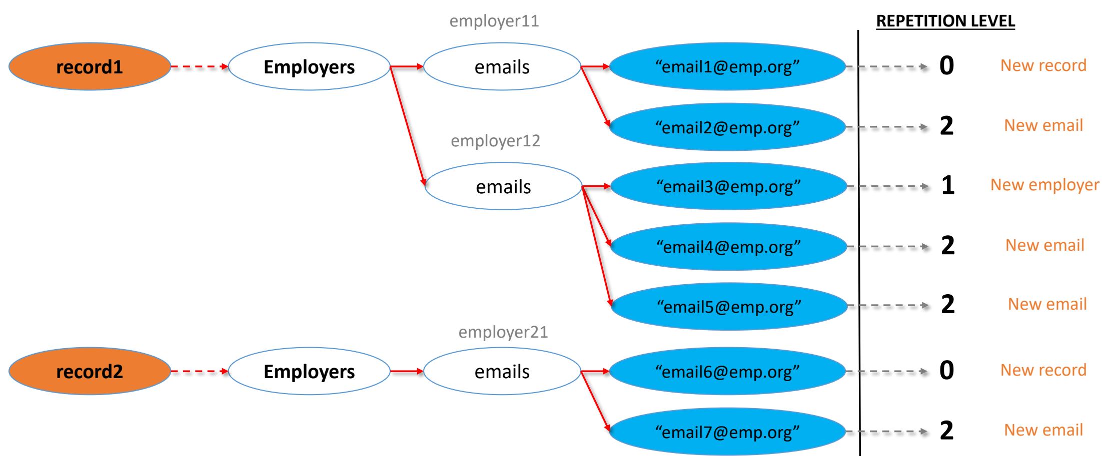


Figure 4.8: Parquet repetition level example

emails as level2.

Both, the definition and repetition levels are optional. In some cases they are not needed. In a flat table they are not necessary and can be omitted to avoid storing that extra space.

Once this definition and repetition levels are designed, it is possible an ease complex columns storage. However, as the other columnar formats seen, Parquet does not make a pure columnar storage. Instead firstly makes row groups, that internally stores data by columns. The row groups size is configurable, and they are usually between 50MB and 1GB. Each of the columns in a row group is called *column chunk*. And inside each one, the data is stored using smaller blocks called *pages*. This pages are compressed independently (like the ORC streams), so they have to be large enough for the compression to be efficient. The pages are the minimum unit to read when accessing a single record. A page contains a metadata header (with the number of records, maximum and minimum values,...), the repetition and definition levels, and the data. To find the columns start locations, the schema and other metadata, there is a footer at the end of the Parquet file. This whole file structure can be seen in Figure 4.9

As it can be deduced, Parquet files are splittable by just dividing the file by the row groups, having each partition one or more entire row groups.

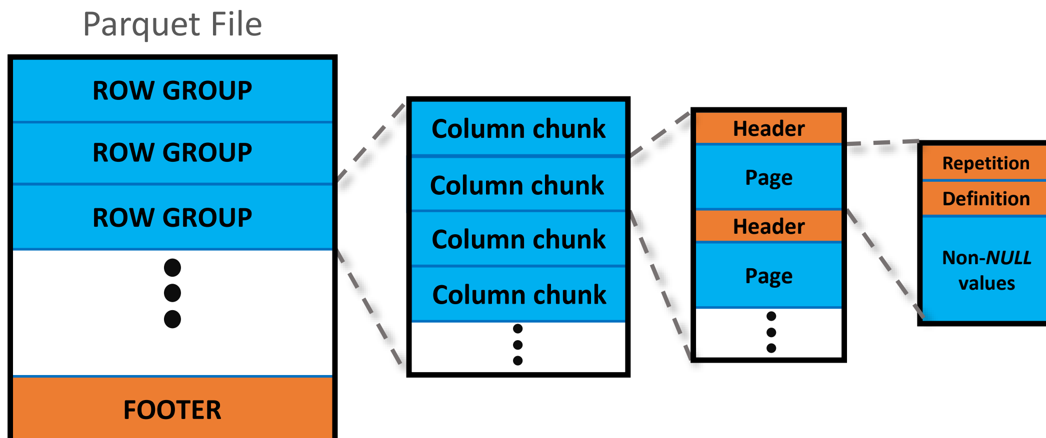


Figure 4.9: Parquet file structure

About the encoding, each page is encoded independently, just to improve the random access to the data. The definition and repetition levels inside the pages are encoded using RLE. For the data inside the pages there are used different techniques depending on the data type (as done in ORC files). Thus, the serialization is optimized for the data type. The encoding used for each page is marked in the page header.

Even each page is encoded individually, it is possible to compress entire blocks (row groups) using any compression codec allowed by Parquet. It is possible to use Gzip, LZO and Snappy codecs, or not use any compression codec above the Parquet internal serialization.

As the design of Parquet can be seen very similar to the ORC files, it is easy to deduce that the *schema evolution* of Parquet is limited, as it is also developed thinking just in write-once data. A Parquet file can only change its schema by adding new columns at the end of the file. However, adding new records to the file is not taken into account, so for append new data it is necessary to create a new file.

4.7 Comparison

Up to now, we have explained each HDFS file format in an individual way. In this section we will make comparisons between them, using the framework described at the beginning of the chapter. An overview of the comparison is shown in Table 4.1.

First of all, it is very important to know that there is not a perfect format, better than any other. Depending on the data structure and the queries done to the data, one format could be better or worse.

As we are working in a Hadoop environment, it is obvious that we will need to split the files and spread them through different nodes. Fortunately, all the considered formats are splittable, and can be partitioned without any problem.

Once we know that all of our file formats can be split, maybe one of the main aspects to take into account is if we prefer columnar or row storage. If the data has a lot of

	CSV	JSON	SequeceFile	Avro	RCFile	ORC	Parquet
Storage orientation	Row	Row	Row[1]	Row	Column	Column	Column
Splitability	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Compression	All	All	All	Deflate, Snappy	All	Zlib, Snappy	Gzip, LZO, Snappy
Human readable	Yes	Yes	Yes	No	No[2]	No	No
Schema storage	No	No	No	Yes	No	Yes	Yes
Schema evolution	No	Yes	No	Yes	No	Limited	Limited
Complex data types	No	Yes	No	Yes	No	Yes	Yes
Data append	Yes	Yes	Yes	Yes	Yes	Yes	No
Random access	No	No	No	No	No	Yes	Yes
MapReduce	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 4.1: File formats comparison table

columns, and very few of them are involved in most of the queries, it would be better a column-oriented format; that can be RCFile, ORC or Parquet. Otherwise, it could be better a row-oriented format, because columnar formats are slower when a lot (or all) the columns are read.

Another very important aspect when working with Big Data is the compression, because with a good compression it is possible to reduce greatly the files storage volume, without loosing a lot of querying performance. Most of the formats allow all the typical Hadoop compressions; except Avro, ORC and Parquet. In Avro and ORC there are only 2 possible compression codecs allowed, Snappy and another codec based on Deflate (Zlib for ORC and directly Deflate for Avro). Parquet allows Gzip, LZO and Snappy. This is not a very big disadvantage, as they are good compression options. But if another codec is preferred or even needed, there is no possibility to use them with these file formats.

Without taking into account the compression, each format makes a serialization to store the data. As it was mentioned at the beginning of the chapter, the data can be serialized in human-readable way or as binary data. The human-readable serialization is not recommended as it is necessary to write a lot of unnecessary information, not needed when it is read by a machine. The text file formats (CSV, JSON) and SequenceFile do not make a binary serialization of data. All the other formats make by default a binary serialization trying to reduce the amount of written data, and save storage space.

Regardless of how the data is stored, it is important to know which is the data structured to make queries to it. In many situations, the client would not know which is exactly the data schema, and it is a big advantage to have that schema explicitly stored in the file. Otherwise the client is forced to know the data structure previously to make the queries. The text formats and SequenceFile do not store information about the data schema, or the

[1]In some very specific situations SequenceFile can be stored as a columnar format. Specifically, when it has a two columns schema (one column in the key, and the other one in the value), and selecting a block compression. However this is a very uncommon situation and thus SequenceFile is not considered as a column-oriented format

[2]RCFile is not considered human readable, because columns are compressed by default with Gzip codec. But if it is configured to not compress the columns, they will be written in plain text, which means it will be human readable.

types. RCFile stores the columns data type, but there is no more information. Thus, we do not consider that RCFile stores a schema. In Avro, Parquet and ORC files it is stored a complete explicit schema with all the necessary information about the data structure; and if we do not want the client to be bound to know the data schema in advance we should use one of these three formats.

In this mentioned schema the columns types are specified. Depending the format we use, we can store complex data types, like inner structures or arrays; or just be able to use primitive types columns. The formats that do not allow complex data types are CSV, SequenceFile and RCFile. If we want to store complex columns we would have to use JSON, Avro, ORC or Parquet.

Related with the schema, in some formats, it is possible to change it without rewriting the entire file. This concept, called schema evolution, was firstly done by Avro, that allows to change the schema as we want. The only restriction is not to change a column type. JSON can make a similar schema evolution as each row is totally independent from the rest; but as there is no column types specification, they can also be changed. Parquet and ORC (in its last versions) also allow some schema evolution, with not so much freedom as in Avro, because it is not possible o remove or change data, just add new columns. This restriction is just because these columnar formats were designed thinking in write-once data.

This though of write-once data is stronger in Parquet. This format does not allow to add new records to the file. It just allows to add new columns, adding them at the end of the file (not very efficient). The rest of columnar formats do allow to add new records by storing them in main memory until there are enough to write a new block. But it is not efficient if a client just want to add one or very few rows. To store data that is going to increase little by little is better to use a row-oriented format.

We refer now to data access, and taking into account that one of the assumptions of Hadoop is that data reads are of very large pieces of data, one aspect that it is not usually optimized is the access to a specific record, or very few records. This is not considered in many cases, because the typical request is to process big amounts of data that are written contiguous. Thus, to access to one single record usually needs to scan sequentially the entire file. This type of queries usually have better performance in column-oriented formats, as they only need to search throw the requested columns, and have more facilities for adding indexes. Among the seen formats, the ones that makes easier these specific records searching are ORC and parquet, that include metadata about the values of the columns inside each partition, telling the maximum and minimum value of the columns in that piece of data. This metadata allows to skip reading entire blocks if the searched value is not in that range.

Finally, all the file formats studied are compatible with MapReduce, and can be used as input or output of MapReduce jobs. However, it is important to highlight that RCFile and ORC have been specifically designed for MapReduce, and they would probably have better performance than other formats designed for much more frameworks. Thus, if we need to store data that will be accessed using different computation frameworks and tools, it could be better to use other formats that allow more interoperability, like Parquet.

With all this, we can conclude what we have said at the beginning of the section. There is no a perfect format. Depending on the data structure and for what that data will

be used, one format can be better than another. For example, if we want to storage a file with a very complex structure (nested columns, arrays, etc.) that will be just written once, and we want to search for very specific records often, we will probably choose ORC or Parquet.

Once the theoretical study and comparison of the HDFS file formats has been done, it is time to start with the experimental one. The next chapter presents the results obtained with the experimentation, and we can check if the conclusions obtained here are correct.

Chapter 5

Experimental results

Now that we have a general idea on how each file format works, tests will be performed in different scenarios to contrast the experimental results against our theoretical conclusions. In this chapter we will show how such experimentation has been carried out, the results we have obtained and the main conclusions.

5.1 Methodology

Before starting with the numerical results, it is necessary to explain how the experimentation have been done. First of all, the computation framework selected for resolving the queries has been MapReduce, as it is the main framework in Hadoop.

5.1.1 Mapreduce

Hadoop MapReduce [10] is a software framework designed to ease the development of applications that have to process large amounts of data (Big Data) in big clusters of commodity hardware, like Hadoop. For this goal, it is based on parallelization.

First, the input data is divided into independent pieces (for example HDFS blocks). Such blocks of data are processed in parallel, in a first phase called **map**. The processing made to each piece of data is called *map task*. Once all the map tasks have finished, the results of this phase are sorted (merging all the outputs), and after that, this data is once again split into pieces. The next phase, called **reduce**, takes these splits as input. Each one of these pieces of data is processed independently by a *reduce task*. The outputs of this phase are written into the file system (HDFS). This complete process is called MapReduce **job**.

The main idea of the MapReduce framework is the independence of each map or reduce task. Thus, it is possible to completely parallelize the map and the reduce phases. Different map tasks can be executed in distinct nodes; and the same happens with the reduce tasks.

Since we are working on a Hadoop cluster, the nodes that execute the map and reduce tasks are the same ones that store the data, the Datanodes. And all the entire MapReduce job is managed by a single master, called **TaskTracker**. This TaskTracker is run in a Datanode. Managing the TaskTrackers it is the **JobTracker**, that is a master that creates

and runs the jobs (it creates the TaskTrackers). There is only one single JobTracker in the cluster.

The JobTracker is responsible of the resources management, tracking resources availability and TaskTrackers life cycle. It is also in charge of rescheduling failed tasks (map or reduce tasks that have failed) and monitoring them. The TaskTracker executes the tasks as scheduled by the JobTracker.

A **map task** has as input a list of key-value pairs. That list is processed in the map phase, and it is returned as another list, also of key-value pairs. The output pairs can be of different type than the input ones. By default, a map task executes one function (called map function) for each pair. Its output is a set of key-value pairs; not necessarily only one.

The output of all the maps is grouped by lists of values that have the same key. Then, a reduce function is executed for each of that groups. Each reduce function produces a list of values. The output of the reduce tasks is the final output of the MarReduce job, and it is written into HDFS. Thus, a MapReduce job transforms a list of key-value pairs into a list of values.

The most typical example of MapReduce is the ”word count”. In this example we have a text, and we want to know the number of occurrences of each word in that text. The map inputs are the lines as value, and numerical key (the key can take any value, as we will not use it in this case). Each map function breaks the line into words, and outputs a key-value pair for each word, with the word as key, and the number of occurrences of the word in the line as value. For the reduce input, the pairs are grouped by words, and each reduce function just needs to sum all the values it has as input. Finally we would have that each reduce obtains the number of occurrences of a word, having one function for each different word.

Hive

Programming MapReduce tasks is not straightforward, as it is necessary to think with key-value pairs, and how the data is managed with that pairs. The map output and reduce input sometimes can be a bit tricky. Also, it is required to write low level programs in Java to use this framework. A solution to these difficulties is Apache Hive.

Apache Hive [47, 48] is a data warehouse software that makes easier the data reading, writing and management with MapReduce, by using a SQL-like syntax (called HQL). Thus, it is not necessary to be aware of the framework details, and just by knowing SQL it is possible to take advantage of all its benefits.

For this reason, Hive has been selected for the majority of the queries used in this project's experimentation. The rest ones are written with a custom Java MapReduce script, as it has been more convenient. The reasons of this decision are explained later on.

5.1.2 Datasets

After explaining the computation framework in which the experimentation will be done, we are going to describe the the benchmark we have used. We decided to use two datasets with very different characteristics.

The first of the datasets selected is one composed of ADS-B (Automatic Dependent Surveillance - Broadcast) [49] messages. ADS-B information messages are sent by air-

crafts to determines their position (and some more data) via satellite navigation. The aircraft periodically broadcasts that messages, and thus it is possible to track it. Currently, we can find many ADS-B data providers, both open and private. One of the most important providers is **Opensky**[1] [50], since it offers free access to large amounts of data, collected through its network, and makes them accessible for researchers worldwide. The ADSB messages used in this project have been collected from Opensky.

The data collected from Opensky has been parsed to a CSV format. Originally it is stored using JSON, but it has a plain table structure and it is easily transformable to CSV. We have used the data from the month of January of 2018. The entire data in CSV without compression has a volume of 364.16GB, with 2,150,817,499 rows, divided into files of approximately 1GB. The schema of this dataset is very simple, with very few columns (just 20). There is no complex types in any column, all of the types are primitive. This schema is available in Appendix A.

The other dataset is the **GitHub log** [51]. This log archives the public GitHub timeline in JSON files (one file per hour). Basically, it contains all the public actions done in GitHub. It has a much more complex structure than the Opensky data. Originally it had more than 700 columns (considering a column each field with a primitive type), but since we have carried out our research using a proprietary cluster, as we will describe later, with a default and fixed configuration of Hive, we had no possibility of storing a column with more than 40000 characters in its type (a column called "payload" stores almost all the columns inside it and it is not possible to store its type metadata). Thus, we have made a schema reduction to allow Hive to store that very complex column. The final result is a schema with 236 different columns, including nested structures and arrays of structures. This schema is shown in Appendix B.

Using this reduction in the schema, we have used the log of the months from January to April of 2018. This data has been "cut" to fix it to our reduced schema. And the final result is a dataset of 195.46GB without compression and 163,939,857 records. This dataset is divided into a large number of small files, each one of approximately 200MB.

Both datasets do not have a very big size, but we have considered their size enough for a proof of concept. The Opensky dataset is larger as it has more simple data, and GitHub log is smaller because it has a more complex structure.

5.1.3 Considered Formats

Knowing the datasets, we have to decide which formats we are going to use. As we are trying to make a complete comparison, we have selected the most popular formats, that are text, SequenceFile, Avro, RCFile, ORC and Parquet. For the text format, it is used CSV with Opensky and JSON with the GitHub log. Thus we do not need to transform from CSV to JSON or vice versa.

Another aspect to consider is the compression. As there are many possibilities, we have to select only some of them to be able to make the experimentation in a reasonable time. Thus we have decided to use the Gzip and Snappy compressions, and also the formats without any compression. Gzip is one of the most important and used compression codecs, and Snappy is available for every file format. Also it is important to know the

[1] https://opensky-network.org/

difference between using or not compression, and thus we use the formats without any codec. It is worth noting that Avro and ORC do not allow to compress their format with Gzip. The only compression codec used for these cases is Snappy.

Thus, for each of the two datasets we would have 16 different combinations of file format and compression codec. Three for text files, SequenceFile, RCFile and Parquet (Snappy, Gzip and no compression), and two for Avro and ORC (Snappy and no compression).

5.1.4 Queries

Once we have selected the datasets and formats that we are going to use, its time to define the queries. For each of the combinations we will make more or less the same queries, varying very little between the two datasets.

We can differentiate two different types of queries, table scans and SELECT queries. The table scans make a complete dataset read, row by row, or an entire column scan. These queries are made with MapReduce scripts, as making them with Hive would produce a very big output, and Hive stores it as temporary data. This means that the data size can be even duplicated. With a custom MapReduce script we are able to just read the data and make null output for the MapReduce job. Thus, we avoid writing very large of unnecessary data.

The SELECT queries are queries made in SQL language, and have a very small output. These queries are done using Hive, as heir output is very small and the written temporary files do not have a big impact.

Including both types, it is possible to differentiate five queries. Query 1 is a complete scan of the dataset, made row by row (script in MapReduce). Query 2 is a entire column read (script in MapReduce). Query 3 is a count of how many records does the dataset have (Hive). Query 4 is a search of a specific record, selecting it by its id (Hive). And query 5 is a search of the ids of the records that fulfills a simple condition of "greater than" in a column, being the condition very restrictive (Hive).

The GitHub log data have two versions of queries 2 and 5. This is because in these queries there is a column read, and it is possible to test what happens if this column is a very nested one. GitHub log has many nested structures and allows to makes this test.

Opensky

- Query 1: Complete dataset read.
- Query 2: Read the column '*id*'.
- Query 3: *SELECT count(*) FROM opensky*
- Query 4: *SELECT * FROM opensky WHERE id="osky-abf2ea-1522188075"*
- Query 5: *SELECT id,altitude FROM opensky WHERE altitude>30000*

GitHub log

- Query 1: Complete dataset read.
- Query 2.1: Read the column '*id*'.
- Query 2.2: Read the column '*payload.pull_request.base.repo.owner.login*'.
- Query 3: *SELECT count(*) FROM githublog*
- Query 4: *SELECT * FROM githublog WHERE id="7044822816"*
- Query 5.1: *SELECT id FROM githublog WHERE id>"7607000000"*
- Query 5.2: *SELECT id,payload.pull_request.base.repo.owner.id FROM githublog WHERE payload.pull_request.base.repo.owner.id>37000000*

5.1.5 Experimentation environment

Finally, it is important to describe where the experimentation has been done, so the results can be contextualized.

The environment used is a Cloudera Hadoop system, property of Boeing Research and Technology Europe. The versions of the used software are **Cloudera 5.14.2**, **HDFS 2.6.0**, **MapReduce 2.6.0** and **Hive 1.1.0**.

The cluster uses for MapReduce jobs 7 different nodes (we will call them node0 to node6). Node0, node1, node2 and node3 have each 128GB of main memory, a hard disk with 8 SDD of 1TB and 5 HDD of 5TB, and 40 cores. Node4 has 256GB of main memory, a hard disk with 8 SDD of 1TB and 5 HDD of 5TB, and 40 cores. Node5 and node6 have 256GB of main memory, a hard disk with 8 SDD of 4TB, and 48 cores.

5.1.6 Used Parameters

The performance aspects that we have considered are the **execution time** of the entire MapReduce job, the **number of map tasks** launched, and the add of all the map tasks memory peak, that would be equivalent to the job **memory peak** value. We do not consider the reduce tasks, as the selected queries only need the map phase and do not require reduce.

Each query is executed three times for each of the combinations of dataset, file format and compression codec, and the file performance results is the average value of the three executions. To guarantee the experiment validity, we calculate the variation coefficient (VC), that is a percentage of the standard deviation, for each of the considered measures. If the VC is greater than 10% for any of them, the query is repeated again three times, and the previous values are discarded.

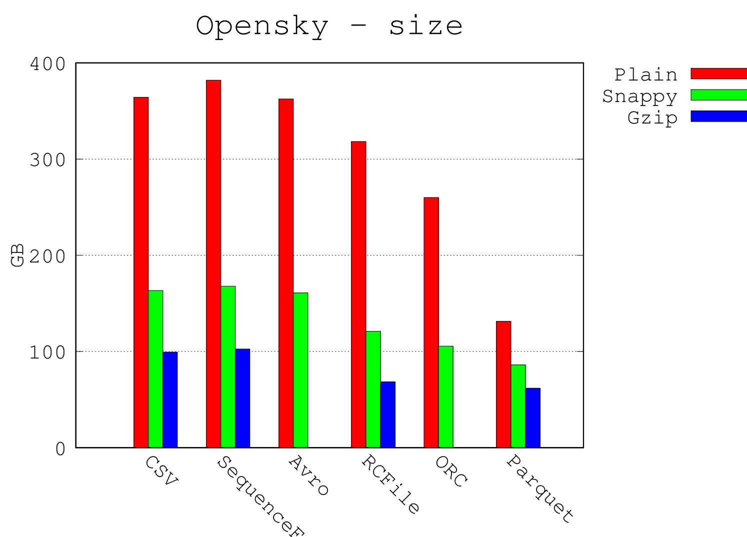


Figure 5.1: Opensky format sizes

	Plain	Snappy	Gzip
CSV	364.16	163.32	99.333
SequenceFile	381.99	167.87	102.57
Avro	362.48	160.94	-
RCFile	318.25	120.99	68.413
ORC	260.02	105.53	-
Parquet	131.27	86.01	61.918

Table 5.1: Opensky format sizes (GB)

5.2 Results

Now that we know all the experimental environment, and all the proves that we are making, it is time to see the performance results for each of the queries and datasets.

5.2.1 Opensky

Before starting with the queries, it is very important to take a look to the **storage size** that each file format use. Depending on the format serialization and the selected compression codec we can store the same amount of data with different sizes.

The different storage volumes of each file format and compression is represented in the graphic of Figure 5.1 and Table 5.1. In this graphic we take as reference the CSV format, that represents the original data.

We can appreciate that the best compression rates are obtained with the column-oriented file formats. This is because storing all column values together ease a better compression rate, because within a same column values are similar.

The dataset stored with SequenceFile requires a 4.89%, 2.78% and 3.25% more storage space than CSV (in plain, Snappy and Gzip respectively). This is because SequenceFile stores the same rows adding one key per each one. Thus, we have the same data plus

keys, which adds even more volume to the files.

Avro requires 0.47% less size with no compression and 1.46% less with Snappy than CSV; this is more or less the same size as the text format. This is due to the fact that Avro serialization does not compress too much, and it even adds keys inside each row. We can guess that Avro was designed not thinking in make smaller CSV files, but more prepared for JSON style structures.

RCFile does not provide a very big advantage when using it without any compression codec, as it is only 12.61% smaller. But we have to take into account that its default configuration uses Gzip codec. With Snappy or Gzip compression we obtain much better rates than in the row-oriented formats, having a compression of 25.92% with Snappy and 31.13% with GZip. Here it is possible to see that column-oriented formats ease the compression by storing similar values together.

ORC serialization makes a the lightweight compression before the general one. This adds the advantage of reaching better compression ratios, having files 28.6% smaller with no compression. But it is necessary to use a compression codec over the ORC serialization to reduce notably the files volume, as we obtain better ratios (35.38% smaller files with Snappy).

The last column-oriented format, Parquet, is the one that reaches the smallest sizes, having files 63.93% smaller with no compression, 47.34% with Snappy, and 37.67% with Gzip. In the theory we saw that it uses also a previous compression for each page depending on the data type, as ORC does. But comparing he compression rates, we can see that without any compression codec, Parquet obtains much better storage results.

Finally, talking about compression codecs in general, we can observe that using a compression over the file formats allows to have much smaller files, reaching up to a quarter of the original size using Gzip, or almost one third with Snappy. Thus, if the storage capacity of our system is not very large, it is a very good idea to use compression codecs over the files.

Now that we have seen the storage sizes of each of the file formats and compression codecs, we start with the performance of the queries for each of them.

Query 1

The first of the queries is the complete scan of the files, made by rows. This query is made using a MapReduce script. The performances of this query are shown in the three graphics of Figure 5.2 and Table 5.2. The first graphic shows the average execution time for each file format and compression, the second one the average memory peak, and the third one the number of map tasks. In the table there are shown all the results obtained from the three executions of each format, including execution time, memory peak and number of map tasks. Also it is shown the average value, the standard deviation and the variation coefficient.

We can observe that the most restrictive compression codecs provide much better performance than the file formats without any compression, being the formats compressed with Gzip the ones getting better results; for example, with CSV format Gzip is 53.37% faster than with no compression. This may seem not obvious, because having compression over some data implies to decompress before accessing it. But MapReduce uses a parallelization process, in which the data is split into different map tasks; and by having

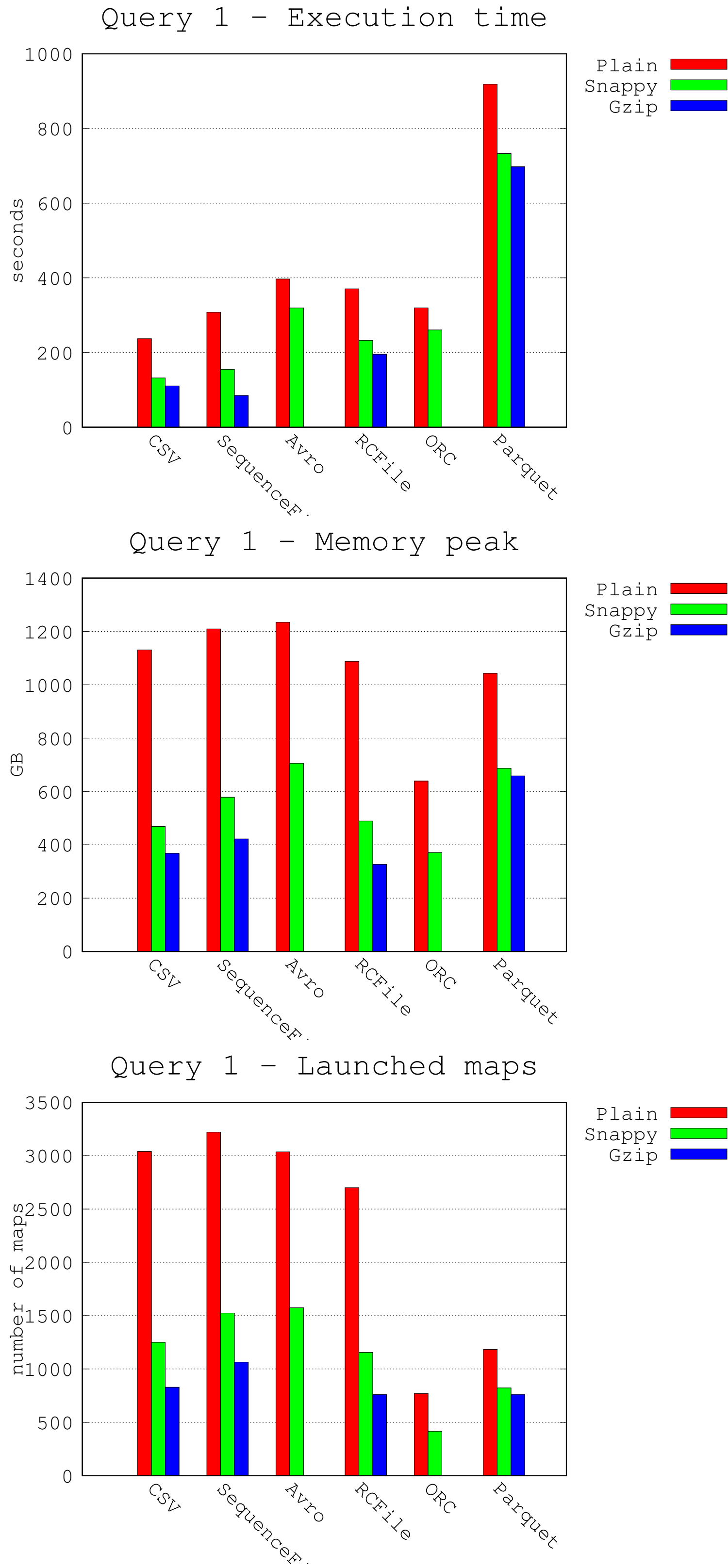


Figure 5.2: Opensky query 1 performance

		run 1	run 2	run 3	Average	Desv.	VC
CSV-Plain	**time (sec)**	243	236	233	237.3333	5.1316	0.0216
	mem.(GB)	1130.7030	1137.0915	1154.0796	1140.6247	12.0822	0.0106
	maps	3040	3040	3040	3040	0	0
CSV-Snappy	**time (sec)**	441	430	425	432	8.1854	0.032
	mem.(GB)	168.6873	165.7842	163.1847	165.8854	2.7527	0.0059
	maps	1250	1250	1250	1250	0	0
CSV-Gzip	**time (sec)**	109	109	114	110.6667	2.8868	0.0261
	mem.(GB)	367.8700	370.6483	373.2353	370.5845	2.6832	0.0072
	maps	830	830	830	830	0	0
SeqF-Plain	**time (sec)**	324	290	310	308	17.088	0.0555
	mem.(GB)	1209.2886	1221.0791	1197.7321	1209.3666	11.6737	0.0097
	maps	3221	3221	3221	3221	0	0
SeqF-Snappy	**time (sec)**	158	153	154	155	2.6458	0.0171
	mem.(GB)	578.4998	568.7232	561.7734	569.6654	8.4029	0.0148
	maps	1524	1524	1524	1524	0	0
SeqF-Gzip	**time (sec)**	83	85	87	85	2	0.0235
	mem.(GB)	422.0018	426.9561	427.5538	425.5039	3.0476	0.0072
	maps	1064	1064	1064	1064	0	0
Avro-Plain	**time (sec)**	420	375	396	397	22.5167	0.0567
	mem.(GB)	1234.7048	1212.8505	1214.4151	1220.6568	12.1910	0.0100
	maps	3036	3036	3036	3036	0	0
Avro-Snappy	**time (sec)**	317	319	322	319.3333	2.5166	0.0079
	mem.(GB)	704.6404	700.2435	694.5995	699.8278	5.0333	0.0072
	maps	1575	1575	1575	1575	0	0
RCFile-Plain	**time (sec)**	357	370	385	370.6667	14.0119	0.0378
	mem.(GB)	1088.0207	1102.2412	1122.5886	1104.2835	17.3742	0.0157
	maps	2700	2700	2700	2700	0	0
RCFile-Snappy	**time (sec)**	235	229	234	232.6667	3.2146	0.0138
	mem.(GB)	488.9639	498.2054	506.6350	497.9348	8.8386	0.0178
	maps	1156	1156	1156	1156	0	0
RCFile-Gzip	**time (sec)**	213	183	190	195.3333	15.695	0.0803
	mem.(GB)	326.7258	324.3799	328.4282	326.5113	2.0326	0.0062
	maps	760	760	760	760	0	0
ORC-Plain	**time (sec)**	320	317	322	319.6667	2.5166	0.0079
	mem.(GB)	638.9058	648.3744	651.2856	646.1886	6.4729	0.01
	maps	770	770	770	770	0	0
ORC-Snappy	**time (sec)**	256	265	261	260.6667	4.5092	0.0173
	mem.(GB)	370.9429	374.0291	370.618	371.8633	1.8827	0.0051
	maps	416	416	416	416	0	0
Parquet-Plain	**time (sec)**	929	921	906	918.6667	11.6762	0.0127
	mem.(GB)	1042.7736	1053.024	1051.2971	1049.0316	5.4879	0.0052
	maps	1183	1183	1183	1183	0	0
Parquet-Snappy	**time (sec)**	726	738	735	733	6.245	0.0085
	mem.(GB)	716.6175	726.9941	738.6478	727.4198	11.0213	0.0152
	maps	823	823	823	823	0	0
Parquet-Gzip	**time (sec)**	706	697	690	697.6667	8.0208	0.0115
	mem.(GB)	658.3433	670.2001	660.4688	663.0041	6.3219	0.0095
	maps	759	759	759	759	0	0

Table 5.2: Opensky query 1 results

smaller data, we have less map tasks, needing less time to schedule them or repeating the failed ones (less map tasks implies less failed tasks). Also we have blocks with more data inside them, and it takes less time to read a entire block decompressing it, than reading many small blocks without decompressing them.

Another remarkable aspect is that Parquet is the format that performed the worst with this query with an execution time 3.87 times slower than CSV, without using compression. This is because the query is resolved reading by rows and being a column-oriented format is a handicap for this type of queries. However, RCFile and ORC do not take so much time to execute the job.

After Parquet, Avro is the next worse format looking at the execution time, being 1.67 times slower than the text format when no compression is used. It seems that reading a file in Avro takes more time than reading it using the plain text format. And it does not provide a lower memory peak or less map tasks than the rest of formats.

The SequenceFile format has the expected results, as it just takes a little bit more time (29.95% more with no compression); slightly greater memory peak and few more map tasks. This is because it is the same as the CSV format, just adding a key that only implies more storage volume.

The other two column-oriented formats, RCFile and ORC have had a very good performance, taking into account that they are column-oriented formats, and this query is hard for them, as they need to search every row field inside each column. They just have an execution 1.56 times slower for RCFile and 1.34 times for ORC, without using compression. It is also remarkable the very few resources used by ORC, that has launched very few maps (only 770 with no compression, having CSV 3040) and has the lowest memory peak values (56.65% lower than CSV without compression).

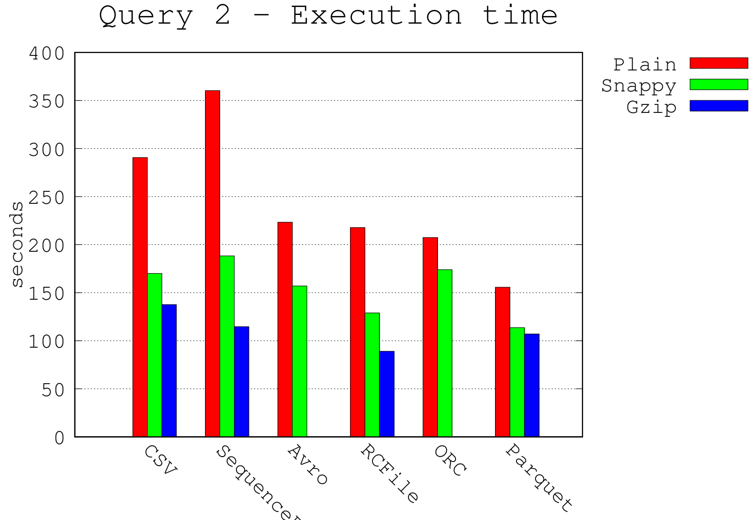
Finally, we would like to emphasize that the format that had a smaller execution time in this full scan query has been the CSV format, with an average of 237.33 seconds; despite it is supposed to be one of the worst formats, as it is human readable. Moreover, it does not consume more resources than the majority of the other formats. Nevertheless, we have to take into account that a full scan without doing anything else is not a very common action.

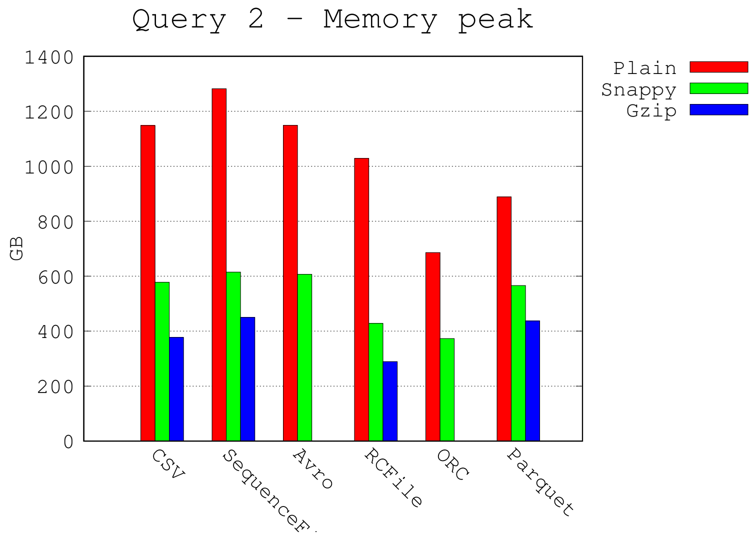
Query 2

Now is time for the other MapReduce scripted query, the complete read of a single column. In this query the best formats should be the column-oriented ones, as the values of a column are stored together. The results of this query are shown in Figure 5.3 and Table 5.3.

Again we see that the better performance results are the ones of the compressed datasets, having an execution time in CSV 41.51% and 52.63% faster with Snappy and Gzip compressions than with no compression. Thus we can guess that in MapReduce jobs it is more important the the size and number of blocks than the time taken to decompress the data.

In this query the results are the expected ones. The fastest execution times have been obtained by RCFile, ORC and Parquet, reaching RCFile with Gzip an execution time of only 89 seconds; and the slowest ones have been CSV and SequenceFile, having CSV with Gzip an execution time of 137.67 seconds.





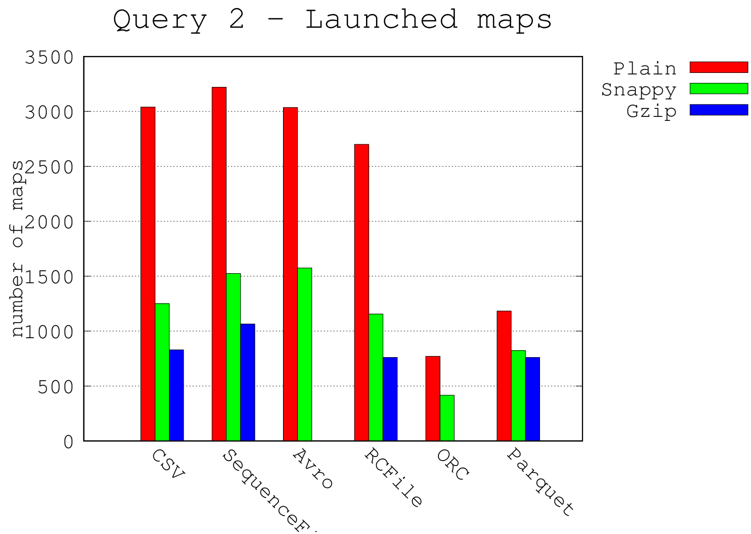


Figure 5.3: Opensky query 2 performance

		run 1	run 2	run 3	Average	Desv.	VC
CSV-Plain	**time (sec)**	294	288	290	290.6667	3.0551	0.0105
	mem.(GB)	1147.9906	1154.9474	1137.8888	1146.9422	8.5775	0.0075
	maps	3040	3040	3040	3040	0	0
CSV-Snappy	**time (sec)**	181	168	161	170	10.1489	0.0597
	mem.(GB)	577.8688	579.0908	581.0590	579.3395	1.6096	0.0028
	maps	1250	1250	1250	1250	0	0
CSV-Gzip	**time (sec)**	136	143	134	137.6667	4.7258	0.0343
	mem.(GB)	377.0761	377.8835	381.0979	378.6858	2.1276	0.0056
	maps	830	830	830	830	0	0
SeqF-Plain	**time (sec)**	379	335	367	360.3333	22.745	0.0631
	mem.(GB)	1281.8952	1285.2409	1273.3653	1280.1671	6.1235	0.0048
	maps	3221	3221	3221	3221	0	0
SeqF-Snappy	**time (sec)**	188	187	190	188.3333	1.5275	0.0081
	mem.(GB)	614.7622	617.4241	613.275	615.1538	2.1021	0.0034
	maps	1524	1524	1524	1524	0	0
SeqF-Gzip	**time (sec)**	121	112	111	114.6667	5.5076	0.048
	mem.(GB)	449.9415	455.8492	446.7915	450.8607	4.5983	0.0102
	maps	1064	1064	1064	1064	0	0
Avro-Plain	**time (sec)**	209	233	228	223.3333	12.6623	0.0567
	mem.(GB)	1148.8208	1126.8438	1131.6216	1135.7621	11.5587	0.0102
	maps	3036	3036	3036	3036	0	0
Avro-Snappy	**time (sec)**	151	160	160	157	5.1962	0.0331
	mem.(GB)	606.7438	607.2778	603.7252	605.9156	1.9156	0.0032
	maps	1575	1575	1575	1575	0	0
RCFile-Plain	**time (sec)**	220	205	228	217.6667	11.6762	0.0536
	mem.(GB)	1028.62	1042.6504	1042.6504	1037.9736	8.1004	0.0078
	maps	2700	2700	2700	2700	0	0
RCFile-Snappy	**time (sec)**	129	125	133	129	4	0.031
	mem.(GB)	428.0213	435.2849	439.3113	434.2058	5.7218	0.0132
	maps	1156	1156	1156	1156	0	0
RCFile-Gzip	**time (sec)**	88	91	88	89	1.7321	0.0195
	mem.(GB)	289.0431	285.7162	284.6591	286.4728	2.2878	0.008
	maps	760	760	760	760	0	0
ORC-Plain	**time (sec)**	209	205	208	207.3333	2.0817	0.01
	mem.(GB)	685.8854	675.7823	665.0035	675.5571	10.4427	0.0155
	maps	770	770	770	770	0	0
ORC-Snappy	**time (sec)**	173	169	180	174	5.5678	0.032
	mem.(GB)	372.9662	368.6809	374.2737	371.9736	2.9256	0.0079
	maps	416	416	416	416	0	0
Parquet-Plain	**time (sec)**	165	145	157	155.6667	10.0664	0.0647
	mem.(GB)	888.5021	885.6411	894.2849	889.476	4.4035	0.005
	maps	1183	1183	1183	1183	0	0
Parquet-Snappy	**time (sec)**	109	116	116	113.6667	4.0415	0.0356
	mem.(GB)	565.8508	557.397	558.5341	560.594	4.5879	0.0082
	maps	823	823	823	823	0	0
Parquet-Gzip	**time (sec)**	101	103	117	107	8.7178	0.0815
	mem.(GB)	437.0639	435.556	438.5178	437.0459	1.481	0.0034
	maps	759	759	759	759	0	0

Table 5.3: Opensky query 2 results

However, it is remarkable that Avro has a good performance in this single column read; taking into account the bad performance in the previous full scan query. Avro is even 9.77% faster than ORC, using both of them the Snappy compression.

If we may select a single format with a better performance would be RCFile when using Gzip compression, because it has the fastest execution time, with 89 seconds, only 760 map tasks, and it have a low memory peak, under 300GB. Also ORC has a very low usage of resources, but using the Snappy compression codec does not have a very good execution time, being even 2.35% slower than the CSV format.

It is worth noting that the number of map tasks is the same as in query 1, so this number is probably related with the files size and the used format, and not with the query. This in theory seems obvious, as the processing of the queries is made inside the map tasks, and the number of these tasks is defined in advance.

Query 3

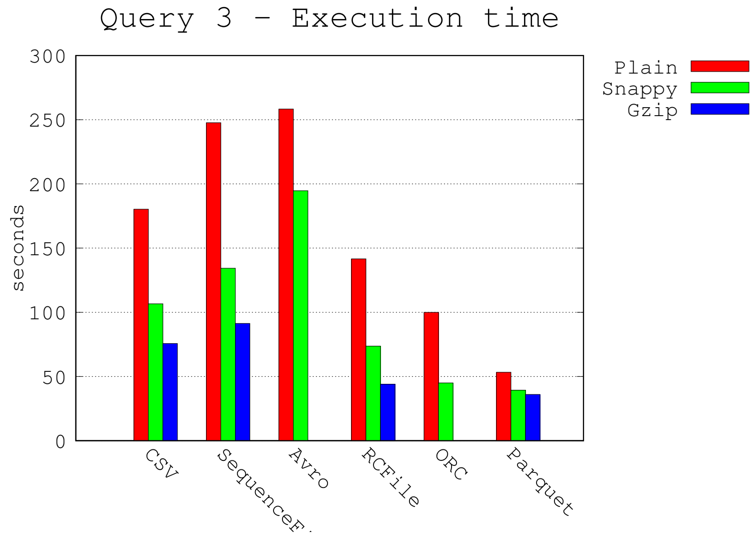
The next query is the count. This query is made with Hive, and it is just a count of all the rows that the dataset has. For this query it is very important the metadata stored by the format, as it allows to not read any data and just read the blocks metadata to know how many rows there are inside. The performance of this query is shown in Figure 5.4 and Table 5.4.

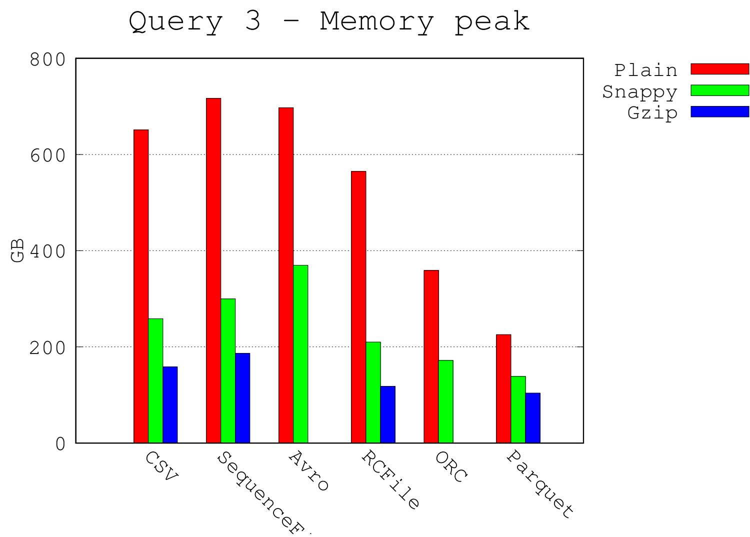
For this query there is a single format better than any other, that is Parquet. It has the fastest execution time, with no more than 53 seconds for any compression option, the lowest memory peak, with only 225.84GB when no compression is used, and a very low number of map tasks. This means that Parquet in this case makes a very good use of its metadata, not reading the entire blocks and just taking into account the headers.

ORC and RCFiles also have a good execution time, being ORC 54.44% and RCFile 21.44% faster than CSV without compression. However, it looks like they use too much resources for just reading the metadata. They have a memory peak 59.5% and 152.8% higher (ORC and RCFile respectively) than Parquet without compression. We presume this could be because the entire blocks are loaded into main memory, but just the metadata is read, not doing anything with the the blocks data. It would be better if there was no necessity of loading the entire blocks into memory and just loading the metadata.

Now talking about the row-oriented formats, CSV is the fastest of the three, being Avro the slowest one, having an execution time 43% higher than the text format without compression. All of them have a much higher execution time than the column-oriented formats. This is probably because of the lack of metadata, needing to read each record just to add one to the count. However, Avro contains metadata of how many records there are in each of its blocks; but seems that this data is not used with this query. SequenceFile, as before, has a slightly worse performance than the text format in every aspect.

In this query the number of maps is different from the first two queries, probably because Hive configuration is different from MapReduce default one.





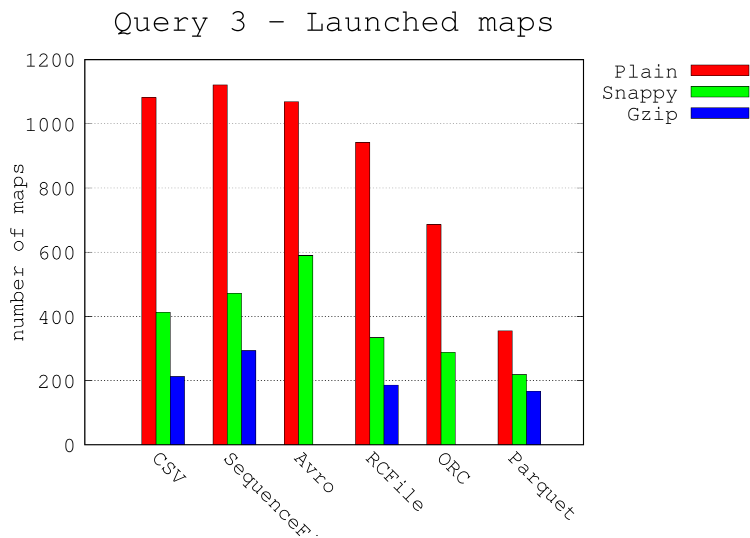


Figure 5.4: Opensky query 3 performance

		run 1	run 2	run 3	Average	Desv.	VC
CSV-Plain	**time (sec)**	186	179	176	180.3333	5.1316	0.0285
	mem.(GB)	651.1885	643.8105	633.4194	642.8061	8.927	0.0139
	maps	1082	1082	1082	1082	0	0
CSV-Snappy	**time (sec)**	136	138	130	134.6667	4.1633	0.0309
	mem.(GB)	258.5566	261.7808	262.1054	260.8143	1.9619	0.0075
	maps	413	413	413	413	0	0
CSV-Gzip	**time (sec)**	79	72	76	75.667	3.5119	0.0464
	mem.(GB)	158.3127	161.384	164.1259	161.2742	2.9082	0.018
	maps	213	213	213	213	0	0
SeqF-Plain	**time (sec)**	248	236	259	247.6667	11.5036	0.0464
	mem.(GB)	716.5535	702.294	716.0801	711.6425	8.0995	0.0114
	maps	1121	1121	1121	1121	0	0
SeqF-Snappy	**time (sec)**	105	110	104	106.3333	3.2146	0.0302
	mem.(GB)	299.8765	305.5052	304.4909	303.2909	3.0001	0.0099
	maps	472	472	472	472	0	0
SeqF-Gzip	**time (sec)**	90	90	94	91.3333	2.3094	0.0253
	mem.(GB)	186.4595	185.3464	183.1129	184.9729	1.7043	0.0092
	maps	293	293	293	293	0	0
Avro-Plain	**time (sec)**	260	254	261	258.3333	3.7859	0.0147
	mem.(GB)	697.4	698.209	702.9777	699.5289	3.014	0.0043
	maps	1069	1069	1069	1069	0	0
Avro-Snappy	**time (sec)**	184	191	209	194.6667	12.8970	0.0663
	mem.(GB)	369.623	371.083	376.7791	372.495	3.7813	0.0102
	maps	590	590	590	590	0	0
RCFile-Plain	**time (sec)**	135	150	140	141.6667	7.6376	0.0539
	mem.(GB)	564.6362	572.208	572.7287	569.8576	4.5294	0.0079
	maps	942	942	942	942	0	0
RCFile-Snappy	**time (sec)**	74	72	75	73.6667	1.5275	0.0207
	mem.(GB)	210.0078	210.138	210.3691	210.1716	0.183	0.0009
	maps	334	334	334	334	0	0
RCFile-Gzip	**time (sec)**	41	46	45	44	2.6458	0.0601
	mem.(GB)	117.5749	119.4525	117.5031	118.1768	1.1054	0.0094
	maps	186	186	186	186	0	0
ORC-Plain	**time (sec)**	94	104	102	100	5.2915	0.0529
	mem.(GB)	358.7742	361.9996	357.4637	359.4125	2.3343	0.0065
	maps	686	686	686	686	0	0
ORC-Snappy	**time (sec)**	50	43	42	45	4.3589	0.0969
	mem.(GB)	171.8806	171.2257	170.8405	171.3156	0.5259	0.0031
	maps	288	288	288	288	0	0
Parquet-Plain	**time (sec)**	50	57	53	53.3333	3.5119	0.0658
	mem.(GB)	225.6203	227.6035	224.2918	225.8385	1.6666	0.0074
	maps	355	355	355	355	0	0
Parquet-Snappy	**time (sec)**	40	41	37	39.3333	2.0817	0.0529
	mem.(GB)	138.715	140.4669	141.8997	140.3605	1.595	0.0114
	maps	219	219	219	219	0	0
Parquet-Gzip	**time (sec)**	35	36	37	36	1	0.0278
	mem.(GB)	104.1195	104.258	103.5866	103.988	0.3545	0.0034
	maps	167	167	167	167	0	0

Table 5.4: Opensky query 3 results

Query 4

The query 4 is the single record search. This means that it is made a query by the column "*id*" that only returns one single record. Even only one column is required for the restriction, all of them are returned in the resulting record. Again in this query it is important to have metadata to be able to skip entire blocks if it is sure that the record can not be in them. The results of this query are shown in Figure 5.5 and Table 5.5.

Again, Avro is the one with worst performance, having the highest execution time (241.67 seconds without compression) and a big use of resources, specially with Snappy compression, reaching almost the 600GB. This is probably because it needs to read the entire file to know if a record accomplish the restriction used (in this case to have a specific value in the column "*id*"). CSV and SequenceFile also have to do this entire read, but they are faster making a complete scan, as we saw in query 1.

In theory the formats that should be faster in this query are ORC and Parquet, that have metadata about the maximum and minimum values of their column blocks. However RCFile is as fast as ORC and Parquet are in this query, being even 44.91% faster than Parquet with Gzip compression. This could be because it has not been possible to skip many blocks with their metadata.

With all this, we can say that the best format for this query in this dataset has been RCFile with the Gzip compression. It is not an expected result, as we had considered it not to ease the random access to data. Also, the use of memory of this format has been very low, being the only one under 100GB.

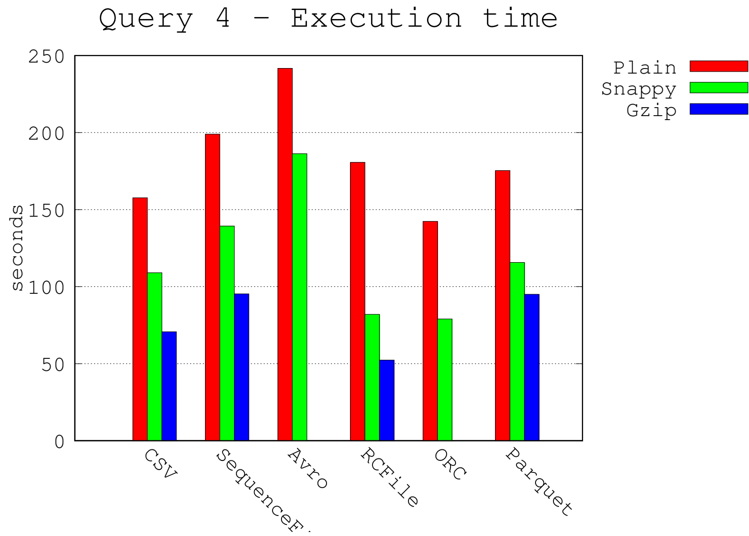
Here the number of map tasks are the same as in query 3, so we can now be almost sure that the number of map tasks depends just on the files size.

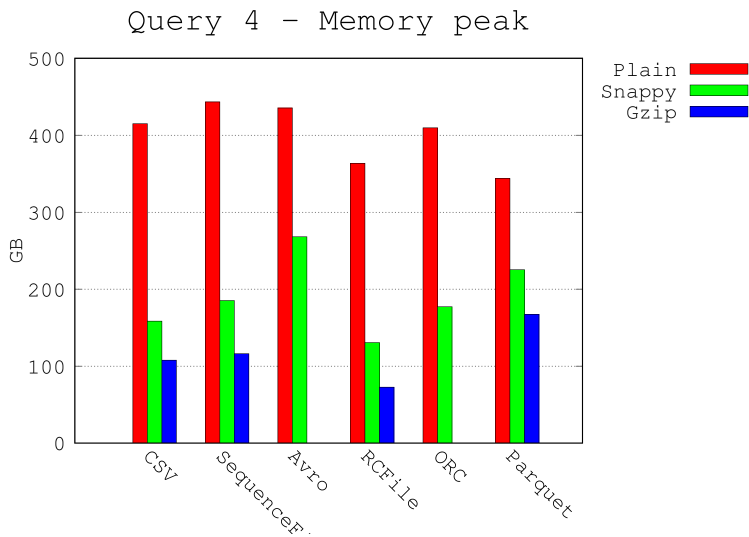
Query 5

The last query done for this dataset has been the query 5. This query requests the id of the records that have an altitude higher than 30000 meters. This is a extremely high value, that only very special flights can reach. Thus, very few records would accomplish this restriction. The performance results for his query are shown in Figure 5.6 and Table 5.6. For this query it is important the formats metadata to be able to skip blocks.

Once again we can see that Avro is the one having the worst performance, reaching the highest execution time, with almost 250 seconds with no compression, and more than 150 with Snappy; and also has a high usage of resources. The CSV and SequenceFile formats are better than Avro, using less resources (37.17% and 31.22% less respectively, when using Snappy), except SequenceFile without compression, and doing the job faster (48.36% CSV and 44.26% SequenceFile, with Snappy).

As expected, the formats with better performance are the column-oriented ones. Now the RCFile format is 43.75% slower than ORC and 21.05% slower than Parquet, using Snappy compression. This is probably because now ORC and Parquet have made a better use of their metadata, being able to skip more blocks and avoid unnecessary readings. However, The usage of memory do not reflect this fact, as Parquet consumes 67.17% and 89.74% more memory than RCFile when Snappy and Gzip compressions are used. This should be because now Parquet is loading the entire blocks into memory, even if they are skipped later.





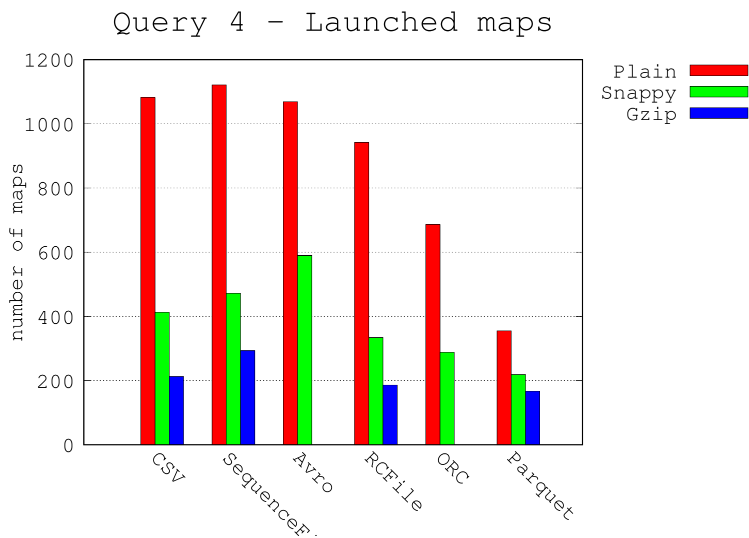
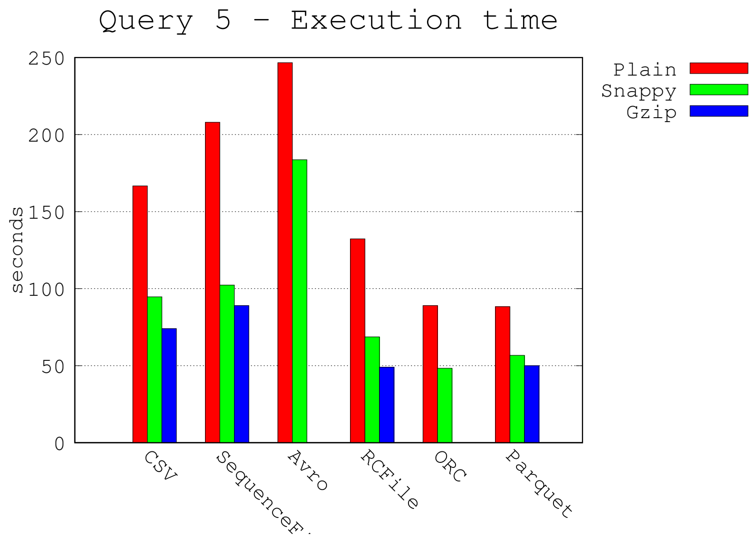
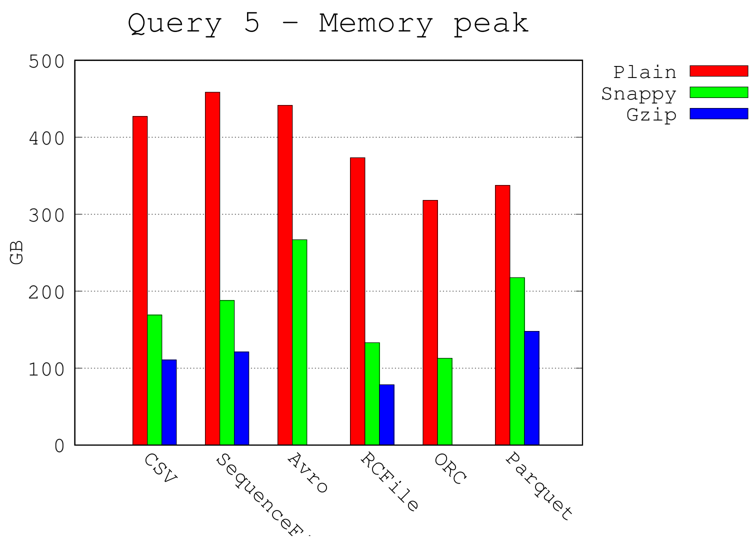


Figure 5.5: Opensky query 4 performance

		run 1	run 2	run 3	Average	Desv.	VC
CSV-Plain	**time (sec)**	157	153	163	157.6667	5.0332	0.0319
	mem.(GB)	415.0875	421.8078	425.9711	420.9555	5.4916	0.013
	maps	1082	1082	1082	1082	0	0
CSV-Snappy	**time (sec)**	104	109	114	109	6.0277	0.0433
	mem.(GB)	158.4278	160.0691	162.7903	160.429	2.2034	0.0137
	maps	413	413	413	413	0	0
CSV-Gzip	**time (sec)**	69	68	75	70.667	3.7859	0.0536
	mem.(GB)	107.7545	105.9577	104.9237	106.212	1.4324	0.0135
	maps	213	213	213	213	0	0
SeqF-Plain	**time (sec)**	198	200	199	199	1	0.005
	mem.(GB)	443.2609	441.6829	449.1871	444.7103	3.9565	0.0089
	maps	1121	1121	1121	1121	0	0
SeqF-Snappy	**time (sec)**	145	133	140	139.3333	5	0.0459
	mem.(GB)	185.2279	186.6246	186.5723	186.1416	0.7917	0.0043
	maps	472	472	472	472	0	0
SeqF-Gzip	**time (sec)**	91	96	99	95.3333	4.0415	0.0424
	mem.(GB)	116.1417	116.669	114.9504	115.9204	0.8804	0.0076
	maps	293	293	293	293	0	0
Avro-Plain	**time (sec)**	245	244	236	241.6667	4.9329	0.0204
	mem.(GB)	435.4667	427.367	427.4054	430.0797	4.6653	0.0108
	maps	1069	1069	1069	1069	0	0
Avro-Snappy	**time (sec)**	194	183	182	186.3333	6.6583	0.0357
	mem.(GB)	268.0768	271.1302	275.6012	271.6028	3.7844	0.0139
	maps	590	590	590	590	0	0
RCFile-Plain	**time (sec)**	187	176	179	180.6667	5.6862	0.0315
	mem.(GB)	363.5963	360.1676	359.4725	361.0788	2.2078	0.0061
	maps	942	942	942	942	0	0
RCFile-Snappy	**time (sec)**	80	81	85	82	2.6458	0.0323
	mem.(GB)	130.5852	128.2059	128.7931	129.1947	1.2394	0.0096
	maps	334	334	334	334	0	0
RCFile-Gzip	**time (sec)**	53	50	54	52.3333	2.0817	0.0398
	mem.(GB)	72.5933	71.7396	70.3801	71.571	1.1162	0.0156
	maps	186	186	186	186	0	0
ORC-Plain	**time (sec)**	140	143	144	142.3333	2.0817	0.0146
	mem.(GB)	409.6368	413.5612	409.5248	410.9076	2.2987	0.0056
	maps	686	686	686	686	0	0
ORC-Snappy	**time (sec)**	79	75	83	79	4	0.0506
	mem.(GB)	177.1493	178.9722	177.0178	177.7131	1.0924	0.0061
	maps	288	288	288	288	0	0
Parquet-Plain	**time (sec)**	169	182	175	175.3333	6.5064	0.0371
	mem.(GB)	343.8616	346.7982	342.921	344.5269	2.0224	0.0059
	maps	355	355	355	355	0	0
Parquet-Snappy	**time (sec)**	111	122	114	115.6667	5.6862	0.0492
	mem.(GB)	225.1929	220.7882	222.9916	222.9909	2.2024	0.0099
	maps	219	219	219	219	0	0
Parquet-Gzip	**time (sec)**	92	95	98	95	3	0.0316
	mem.(GB)	166.9893	165.3912	167.1807	166.5204	0.9826	0.0059
	maps	167	167	167	167	0	0

Table 5.5: Opensky query 4 results





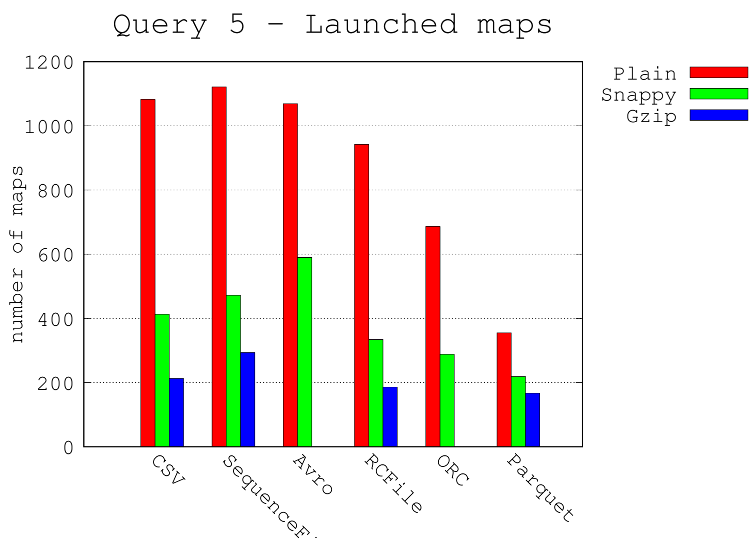


Figure 5.6: Opensky query 5 performance

		run 1	run 2	run 3	Average	Desv.	VC
CSV-Plain	**time (sec)**	164	174	162	166.6667	6.4291	0.0386
	mem.(GB)	426.9127	433.009	437.2222	432.3813	5.1833	0.012
	maps	1082	1082	1082	1082	0	0
CSV-Snappy	**time (sec)**	94	90	100	94.6667	5.0332	0.0532
	mem.(GB)	168.9436	170.1211	169.2552	169.4399	0.6101	0.0036
	maps	413	413	413	413	0	0
CSV-Gzip	**time (sec)**	68	79	75	74	5.5678	0.0752
	mem.(GB)	110.6496	113.3134	110.3602	111.4411	1.628	0.0146
	maps	213	213	213	213	0	0
SeqF-Plain	**time (sec)**	204	212	208	208	4	0.0192
	mem.(GB)	458.5225	465.2582	470.2504	464.6770	5.8855	0.0127
	maps	1121	1121	1121	1121	0	0
SeqF-Snappy	**time (sec)**	100	108	99	102.3333	4.9329	0.0482
	mem.(GB)	187.7981	185.6609	183.4868	185.6486	2.1556	0.0116
	maps	472	472	472	472	0	0
SeqF-Gzip	**time (sec)**	90	84	93	89	4.5826	0.0515
	mem.(GB)	120.9506	119.0239	116.7041	118.8929	2.1263	0.0179
	maps	293	293	293	293	0	0
Avro-Plain	**time (sec)**	248	246	246	246.6667	1.1547	0.0047
	mem.(GB)	441.3357	437.3284	434.7875	437.8172	3.3014	0.0075
	maps	1069	1069	1069	1069	0	0
Avro-Snappy	**time (sec)**	192	179	180	183.6667	7.2342	0.0394
	mem.(GB)	266.9053	272.0406	269.5324	269.4928	2.5679	0.0095
	maps	590	590	590	590	0	0
RCFile-Plain	**time (sec)**	135	130	132	132.3333	2.5166	0.019
	mem.(GB)	373.4748	379.118	375.2776	375.9568	2.8823	0.0077
	maps	942	942	942	942	0	0
RCFile-Snappy	**time (sec)**	67	69	70	68.6667	1.5275	0.0222
	mem.(GB)	133.0986	130.9384	129.6539	131.2303	1.7408	0.0133
	maps	334	334	334	334	0	0
RCFile-Gzip	**time (sec)**	50	49	48	49	1	0.0204
	mem.(GB)	78.4674	78.2689	79.2707	78.669	0.5305	0.0067
	maps	186	186	186	186	0	0
ORC-Plain	**time (sec)**	84	89	94	89	5	0.0562
	mem.(GB)	317.9462	317.5583	315.0083	316.8376	1.596	0.005
	maps	686	686	686	686	0	0
ORC-Snappy	**time (sec)**	48	44	53	48.3333	4.5092	0.0933
	mem.(GB)	112.8209	112.109	111.4543	112.1281	0.6835	0.0061
	maps	288	288	288	288	0	0
Parquet-Plain	**time (sec)**	92	87	86	88.3333	3.2146	0.0364
	mem.(GB)	337.4675	331.0759	334.3138	334.2858	3.1959	0.0096
	maps	355	355	355	355	0	0
Parquet-Snappy	**time (sec)**	56	59	55	56.6667	2.0817	0.0367
	mem.(GB)	217.6367	221.3561	218.3700	219.1210	1.9701	0.009
	maps	219	219	219	219	0	0
Parquet-Gzip	**time (sec)**	49	52	49	50	1.7321	0.0346
	mem.(GB)	147.6998	147.4945	149.0786	148.091	0.8614	0.0058
	maps	167	167	167	167	0	0

Table 5.6: Opensky query 5 results

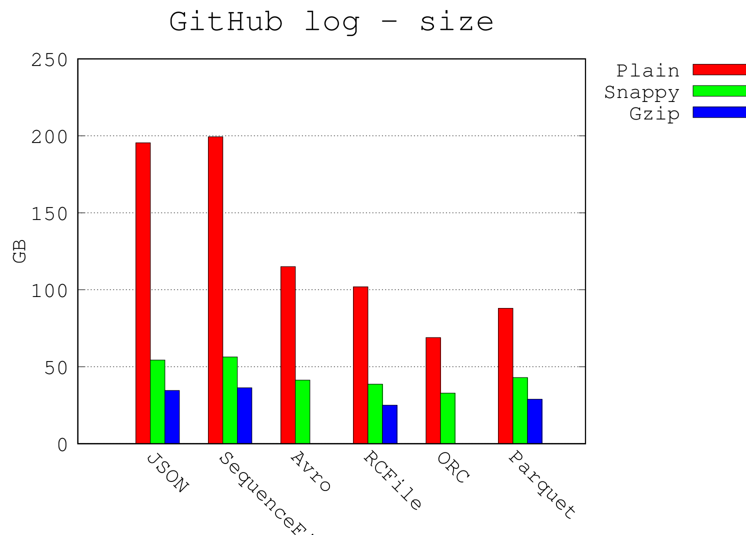


Figure 5.7: GitHub log format sizes

This worse performance of Parquet has not an obvious reason, as we saw in query 3 that parquet was the better managing metadata and not loading unnecessary blocks. This contradiction may be because of the different types of metadata it is managing.

Again the number of maps is the same as in query 4, so we can conclude that the number of map tasks of a query depends only on the configuration and the files size.

5.2.2 Github log

Opposite to the Opensky dataset, we have considered the GitHub log one, that is smaller, but with a much more complex structure. It has many columns, and lot of them have complex types, including inner structures and arrays of structures.

We must take into account that some of the formats do not allow complex types in their columns, so we need to make some processing to be able to perform read and write operations with them. With SequenceFile the solution is very simple; just use the entire row as value and treat it as a string (that is the same thing done with Opensky). But with RCFile the solution is not so easy. As this format only allows to store plain tables, we have had to flatten the structure. Thus we loose the nested structure and just manage a plain table. We need to have this fact present during all the results.

To analyze the results obtained with this dataset we will follow the same dynamic as with Opensky. First of all, the storage volume of each format is one of the main aspects that we have to care about. The data sizes for each format are shown in Figure 5.7 and Table 5.7.

We can observe that the graphic is quite distinct than the one showing the sizes of Opensky dataset. Here we have very differentiated JSON and SequenceFile formats, that for every compression option have the biggest storage volumes (almost 200GB with no compression, over 50GB for Snappy, and over 30GB with Snappy). SequenceFile is slightly bigger than the text file, because the first one is stored the same way as JSON, but adding a numeric key for each row; for example, with no compression SequenceFile has 2% more storage volume than JSON.

	Plain	Snappy	Gzip
CSV	195.46	54.339	34.604
SequenceFile	199.38	56.334	36.321
Avro	115.06	41.287	-
RCFile	101.86	38.679	24.996
ORC	68.976	32.829	-
Parquet	87.95	42.945	28.893

Table 5.7: GitHub log format sizes (GB)

Avro has much better compression rates than with CSV data, reducing the original data 41.02% with no compression and 24.03% with Snappy. This is possible because the Avro structure is very similar o a JSON one. Thus, it is easier for this format to take advantage of its binary serialization, maintaining the same structure and compressing the data. For example, Avro does not need to write the entire key names each time they occur.

However, the smallest sizes are reached by the column-oriented formats. From among all of them, ORC has the best rate without using any compression codec (reaching a volume 64.71% smaller), unlike what happened with the Opensky data, in which Parquet was the best one. It seems that ORC has a better lightweight compression performance when it deals with many columns, in contrast to Parquet, that is 27.51% larger than ORC. This can happen because of the definition and repetition levels stored by Parquet; or a better columns serialization techniques by ORC.

When using compression codecs, RCFile is the one that reaches the smallest size, when it is used with Gzip, having a size of less than 25GB. Nevertheless, it is probable that ORC could reach better compression rates if it were possible to use it with Gzip, as with Snappy compression ORC is 15.12% smaller than RCFile.

It is worth noting that there is a huge difference of storage volume between using or not compression codecs, as it happened with Opensky; for example, with JSON the Gzip compression obtains a size 5.64 times smaller than without using any compression. Thus, we can say that if the storage capacity is a problem, the usage of a compression codec is almost compulsory.

Query 1

First we tested the query 1, that is the full scan of the entire dataset, made by rows. The performance of this query for each format is represented graphically in the figure 5.8.

The most flashy aspect that we can see in the graphics is the fact that for all of the formats there is a same number of map tasks. This is probably because we are working with a dataset composed of many small files, and each file is processed entirely with a single map. Thus, we have 2880 map tasks for every format and compression codec. This is a good opportunity to get a better idea of which format makes a bigger usage of memory inside each map task.

The highest memory peaks are reached by ORC and Parquet, being 132.38% and 160.36% larger, respectively, than the one obtained by JSON, when no compression is used. Even they are the ones with smallest file sizes, the usage of memory is very high.

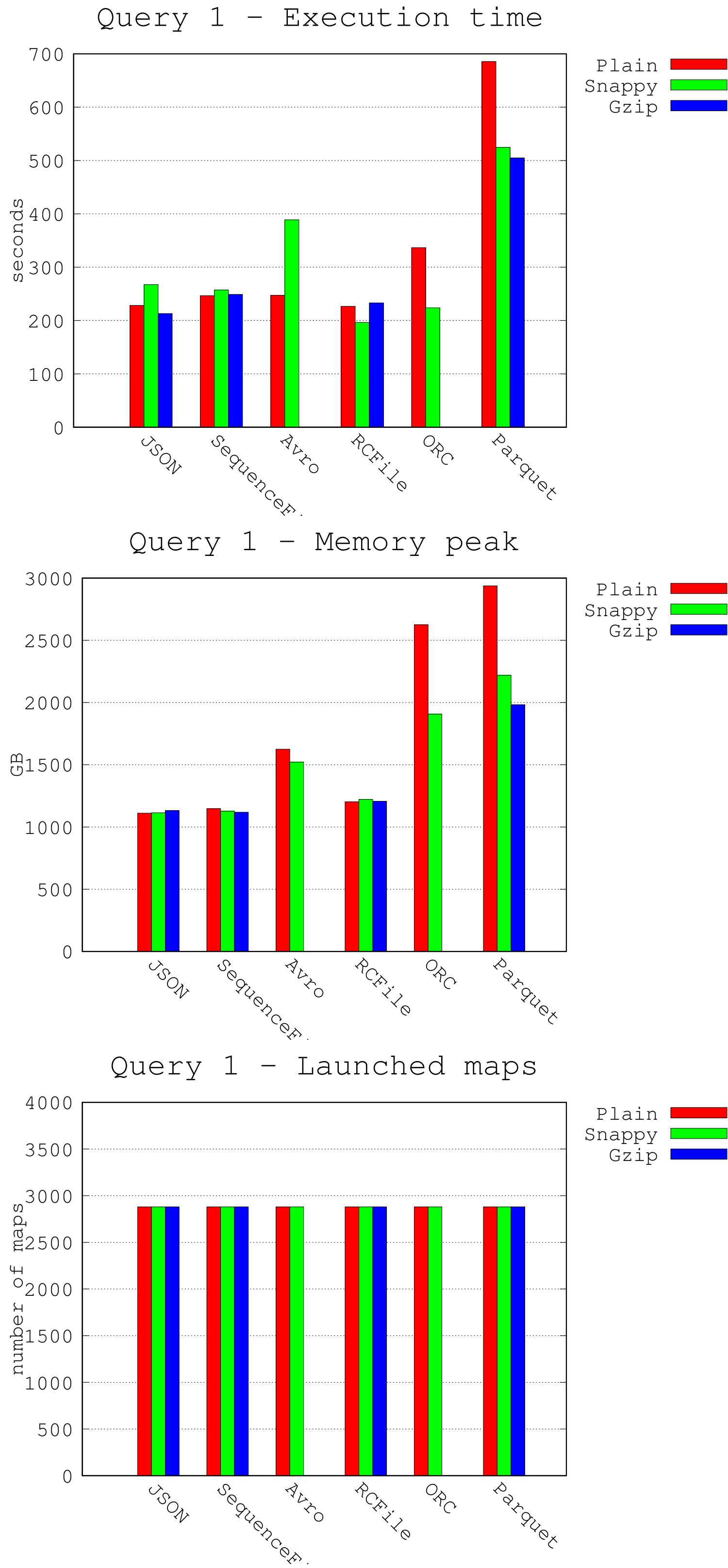


Figure 5.8: GitHub log query 1 performance

		run 1	run 2	run 3	Average	Desv.	VC
JSON-Plain	**time (sec)**	229	228	228	228.3333	0.5774	0.0025
	mem.(GB)	1111.3218	1079.638	1080.7392	1090.5664	17.9832	0.0165
	maps	2880	2880	2880	2880	0	0
JSON-Snappy	**time (sec)**	277	261	264	267.3333	8.5049	0.0318
	mem.(GB)	1114.6334	1093.8567	1102.531	1103.6737	10.4354	0.0095
	maps	2880	2880	2880	2880	0	0
JSON-Gzip	**time (sec)**	155	141	163	153	11.1355	0.0728
	mem.(GB)	1132.434	1197.8774	1235.2272	1188.5129	52.0325	0.0438
	maps	2880	2880	2880	2880	0	0
SeqF-Plain	**time (sec)**	254	233	253	246.6667	11.8462	0.048
	mem.(GB)	1146.8443	1149.677	1130.1325	1142.2179	10.5617	0.0092
	maps	2880	2880	2880	2880	0	0
SeqF-Snappy	**time (sec)**	249	261	262	257.3333	7.2342	0.0281
	mem.(GB)	1126.719	1144.6001	1102.639	1124.6527	21.0567	0.0187
	maps	2880	2880	2880	2880	0	0
SeqF-Gzip	**time (sec)**	249	251	247	249	2	0.008
	mem.(GB)	1118.4939	1084.2121	1075.7878	1092.8313	22.6202	0.0207
	maps	2880	2880	2880	2880	0	0
Avro-Plain	**time (sec)**	257	244	241	247.3333	8.5049	0.0344
	mem.(GB)	1624.933	1672.9498	1664.6185	1654.1671	25.6579	0.0155
	maps	2880	2880	2880	2880	0	0
Avro-Snappy	**time (sec)**	397	374	396	389	13	0.0334
	mem.(GB)	1521.3261	1537.2088	1485.7575	1514.7641	26.3458	0.0174
	maps	2880	2880	2880	2880	0	0
RCFile-Plain	**time (sec)**	237	223	220	226.6667	9.0738	0.04
	mem.(GB)	1202.5244	1239.0932	1282.3127	1241.3101	39.9403	0.0322
	maps	2880	2880	2880	2880	0	0
RCFile-Snappy	**time (sec)**	214	180	196	196.6667	17.0098	0.0865
	mem.(GB)	1221.6956	1270.0992	1290.8145	1260.8698	35.4717	0.0281
	maps	2880	2880	2880	2880	0	0
RCFile-Gzip	**time (sec)**	245	234	220	233	12.53	0.0538
	mem.(GB)	1207.1254	1225.5703	1194.8697	1209.1885	15.4539	0.0128
	maps	2880	2880	2880	2880	0	0
ORC-Plain	**time (sec)**	352	336	322	336.6667	15.0111	0.0446
	mem.(GB)	2624.4902	2546.989	2429.3436	2533.6076	98.2591	0.0388
	maps	2880	2880	2880	2880	0	0
ORC-Snappy	**time (sec)**	225	228	219	224	4.5826	0.0205
	mem.(GB)	1907.1739	1948.1781	1961.3867	1938.9129	28.2691	0.0146
	maps	2880	2880	2880	2880	0	0
Parquet-Plain	**time (sec)**	681	692	684	685.6667	5.6862	0.0083
	mem.(GB)	2937.2068	2836.5781	2741.6662	2838.4837	97.7842	0.0344
	maps	2880	2880	2880	2880	0	0
Parquet-Snappy	**time (sec)**	539	513	522	524.6667	13.2035	0.0252
	mem.(GB)	2218.6109	2295.641	2316.6691	2276.9737	51.6256	0.0227
	maps	2880	2880	2880	2880	0	0
Parquet-Gzip	**time (sec)**	491	510	514	505	12.2882	0.0243
	mem.(GB)	1982.775	2079.1378	2170.308	2077.4069	93.7785	0.0451
	maps	2880	2880	2880	2880	0	0

Table 5.8: GitHub log query 1 results

Also, these columnar formats have not very good execution times, being Parquet the one taking more time to make the full scan, with a very big difference, having its lower value over 500 seconds, using Gzip codec. As it is a column format, this is something understandable.

RCFile, despite being also a column-oriented format, has a very good performance, with a low execution time, around the 200 seconds for each compression option, and not very high memory usage, under the 1500GB. This is something not very common, as the columnar formats do not have good performance when reading by rows. Something similar happened with Opensky, having both ORC and RCFile good execution times. Thus, we can intuit that they have been optimized for not being so much harmed by row scans. It is important to remember that RCFile do not manage complex columns and all them are stored as simple ones.

The row-oriented formats have more predictable results, being the fastest ones the human-readable, as they do no need to deserialize the data. Avro has a worse execution time when it is used with Snappy codec, lasting 57.48% more than when no compression is used.

In contrast to the results obtained with Opensky, the usage of compression with the formats do not make a reduction of time in the executions. Now that we are dealing with very small files, it is not possible to have a smaller number of map tasks, and thus, there are not better execution times. Nevertheless, the compression codecs do not have a worse performance (except with Avro), so it remains beneficial to use compression codecs, as the reduction of files size is very significant.

Also, it is important to remark that the memory peaks are even higher than the ones seen in the Opensky dataset, even being he GitHub log data much smaller. This can be because of the large number of map tasks.

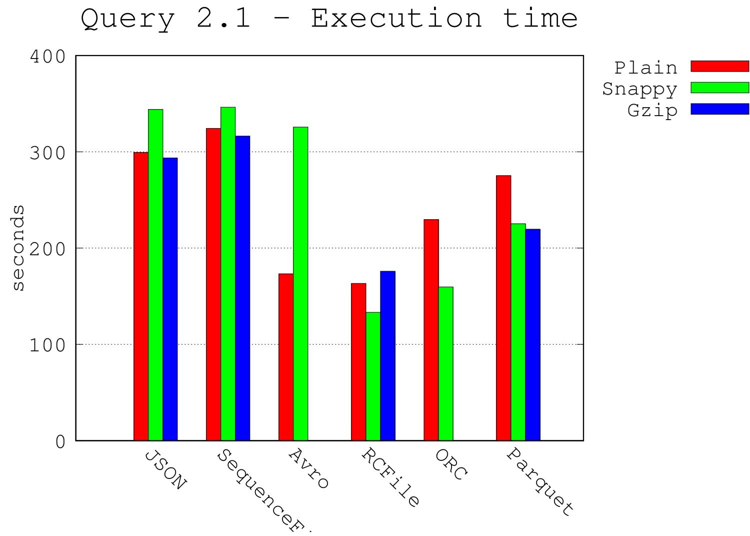
Query 2.1

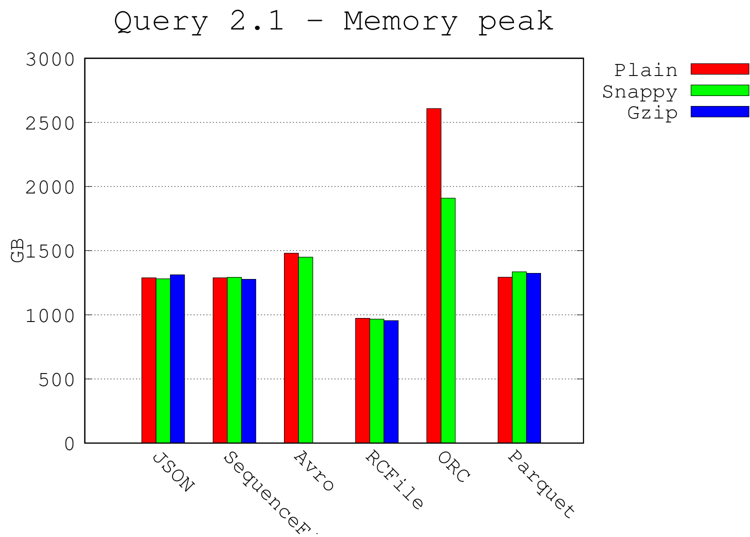
The next query is the query 2, that reads a complete column. For this dataset we have made two variations, requesting first one simple column, and for the next query a very nested one. Firstly we are going to see the simplest one, that we have called query 2.1. The column we are reading now is "*id*". In Figure 5.9 and Table 5.9 we can see the results obtained for this query.

Again we have the same number of map tasks for all of the formats (2880), as we are working with very small files, and with the default MapReduce configuration, there is only one file for each map task.

As in the previous query, we can see that using a compression codec does not makes the execution time shorter. But as we said previously, it is still recommended to use a compression codec because there is a great reduction of storage resources, and the consumption of time and memory is not increased.

Looking at the execution times, the results are the expected ones. The fastest formats are the columnar ones, always under 300 seconds, and the row-oriented are the slowest, that last always more than 300 seconds, except plain Avro. It is necessary to highlight that Avro has a good performance when using it without any compression codec. It is even the second fastest without compression, just a 6.13% slower than RCFile. But when using it with Snappy, the time taken to resolve the query is 87.86% slower, almost doubling





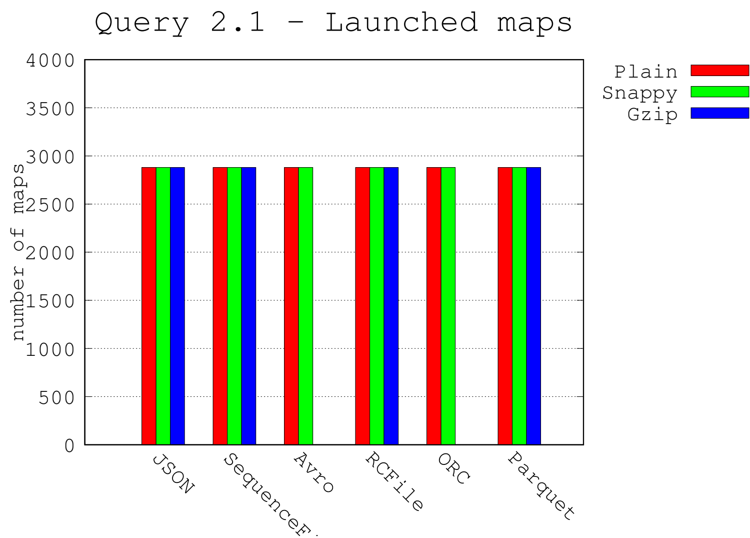


Figure 5.9: GitHub log query 2.1 performance

		run 1	run 2	run 3	Average	Desv.	VC
JSON-Plain	**time (sec)**	303	305	290	299.3333	8.1445	0.0272
	mem.(GB)	1288.1599	1329.4583	1304.3581	1307.3254	20.8085	0.0159
	maps	2880	2880	2880	2880	0	0
JSON-Snappy	**time (sec)**	360	372	360	364	6.9282	0.019
	mem.(GB)	1280.8716	1277.9128	1221.7229	1260.1691	33.3282	0.0264
	maps	2880	2880	2880	2880	0	0
JSON-Gzip	**time (sec)**	222	206	213	213.6667	8.0208	0.0375
	mem.(GB)	1311.7926	1311.0449	1331.235	1318.0242	11.447	0.0087
	maps	2880	2880	2880	2880	0	0
SeqF-Plain	**time (sec)**	307	307	299	304.3333	4.6188	0.0152
	mem.(GB)	1288.3797	1343.3806	1378.107	1336.6224	45.2438	0.0338
	maps	2880	2880	2880	2880	0	0
SeqF-Snappy	**time (sec)**	324	328	327	326.3333	2.0817	0.0064
	mem.(GB)	1291.4937	1338.2846	1306.139	1311.9724	23.9346	0.0182
	maps	2880	2880	2880	2880	0	0
SeqF-Gzip	**time (sec)**	319	313	317	316.3333	3.0551	0.0097
	mem.(GB)	1276.848	1272.9409	1275.4613	1275.0834	1.9808	0.0016
	maps	2880	2880	2880	2880	0	0
Avro-Plain	**time (sec)**	173	174	173	173.3333	0.5774	0.0033
	mem.(GB)	1480.1266	1484.6262	1434.2083	1466.3203	27.9007	0.019
	maps	2880	2880	2880	2880	0	0
Avro-Snappy	**time (sec)**	331	313	333	325.6667	11.0151	0.0338
	mem.(GB)	1449.9317	1478.4229	1464.3875	1464.2474	14.2461	0.0097
	maps	2880	2880	2880	2880	0	0
RCFile-Plain	**time (sec)**	166	161	163	163.3333	2.5166	0.0154
	mem.(GB)	972.533	965.4238	923.09	953.6823	26.7311	0.028
	maps	2880	2880	2880	2880	0	0
RCFile-Snappy	**time (sec)**	144	121	135	133.3333	11.5902	0.0869
	mem.(GB)	966.1714	923.5923	905.8316	931.8651	31.0089	0.0333
	maps	2880	2880	2880	2880	0	0
RCFile-Gzip	**time (sec)**	195	170	163	176	16.8226	0.0956
	mem.(GB)	954.704	977.5978	940.6935	957.6651	18.6295	0.0195
	maps	2880	2880	2880	2880	0	0
ORC-Plain	**time (sec)**	234	245	210	229.6667	17.8979	0.0779
	mem.(GB)	2608.1496	2717.9267	2599.5066	2641.861	66.0164	0.025
	maps	2880	2880	2880	2880	0	0
ORC-Snappy	**time (sec)**	150	162	167	159.6667	8.7369	0.0547
	mem.(GB)	1909.6842	1975.4155	1878.6202	1921.24	49.4215	0.0257
	maps	2880	2880	2880	2880	0	0
Parquet-Plain	**time (sec)**	279	269	278	275.3333	5.5076	0.02
	mem.(GB)	1293.4583	1342.8166	1277.1932	1304.4894	34.1742	0.0262
	maps	2880	2880	2880	2880	0	0
Parquet-Snappy	**time (sec)**	236	223	217	225.3333	9.7125	0.0431
	mem.(GB)	1334.9071	1369.9484	1331.3432	1345.3996	21.3344	0.0159
	maps	2880	2880	2880	2880	0	0
Parquet-Gzip	**time (sec)**	212	222	225	219.6667	6.8069	0.031
	mem.(GB)	1323.4625	1332.5017	1320.4293	1325.4645	6.2803	0.0047
	maps	2880	2880	2880	2880	0	0

Table 5.9: GitHub log query 2.1 results

it. Thus we do not consider Avro a good option, as it is necessary to use it with no compression to have a good performance.

The rest of row-oriented formats have the expected performance. They have an execution time longer than the sequential reading, as they have to read the entire file and just take the requested column; for example, plain JSON is 31.14% slower than in query 1. Similarly to what happened in all the previous queries, the SequenceFile has the same results as the text file, but slightly worse, being a little bit slower and using almost the same memory.

Now focusing on the column-oriented formats, Parquet is the slowest, with more than 200 seconds for every compression option, and RCFile the fastest, that has an execution time of 133.33 seconds with Snappy compression. This is probably because both RCFile and ORC have been designed just thinking in MapReduce, and Parquet has, as one of its main characteristics, compatibility with many more tools and frameworks. Also, RCfile is the simplest format, and in some cases, like these simple queries of reading one column, the simpler the file format is, the better performance it has.

ORC with Snappy compression seems to be also a good option for this query if we just look at the execution time (159.67 seconds). But the memory used by this format is much larger than the rest, reaching a peak of almost 2000GB, while no other format have even 1500GB of memory peak.

The formats that seems to still take advantage of the usage of compression codecs, referring just to execution time and memory peaks, are the columnar ones, especially with Snappy, that has an execution time 18.19% and 20.45% faster with Snappy and Gzip respectively.

Query 2.2

After making a query that just read a very simple column, it is interesting to make another query that reads a column that is not at a first level, but in a very nested column. The selected column for this query has been "*payload.pull_request.base.repo.owner.login*". This column has a simple type (string), but it is in a sixth nested level. It is also a column that contains a lot of null values, even in higher levels (many records have *NULL* "*pull_request*"). Figure 5.10 and Table 5.10 show the performance of this query 2.2.

The results we have obtained are very similar to the ones seen in the previous query. Thus we can assume that the location of a column, referring to the level of nesting, is not something that affects the performance of the query. As could be expected, the use of resources is the same as in the query 2.1.

It is easy to guess that in the column-oriented formats there is no difference between requesting a simple column or a very nested one, as every column is stored together and they do not need to search through different levels of nested structures to find the values.

The row-oriented formats do have a slightly worse performance than in the simple column read; for example plain JSON is 6.24% slower than in query 2.1. This may be because they need to search the requested value through the data structures, going level by level. JSON (and so, SequenceFile storing JSON) and Avro are based on a DOM tree, and to find one value inside lot of structures is not so straightforward. However, the difference is not very significant.

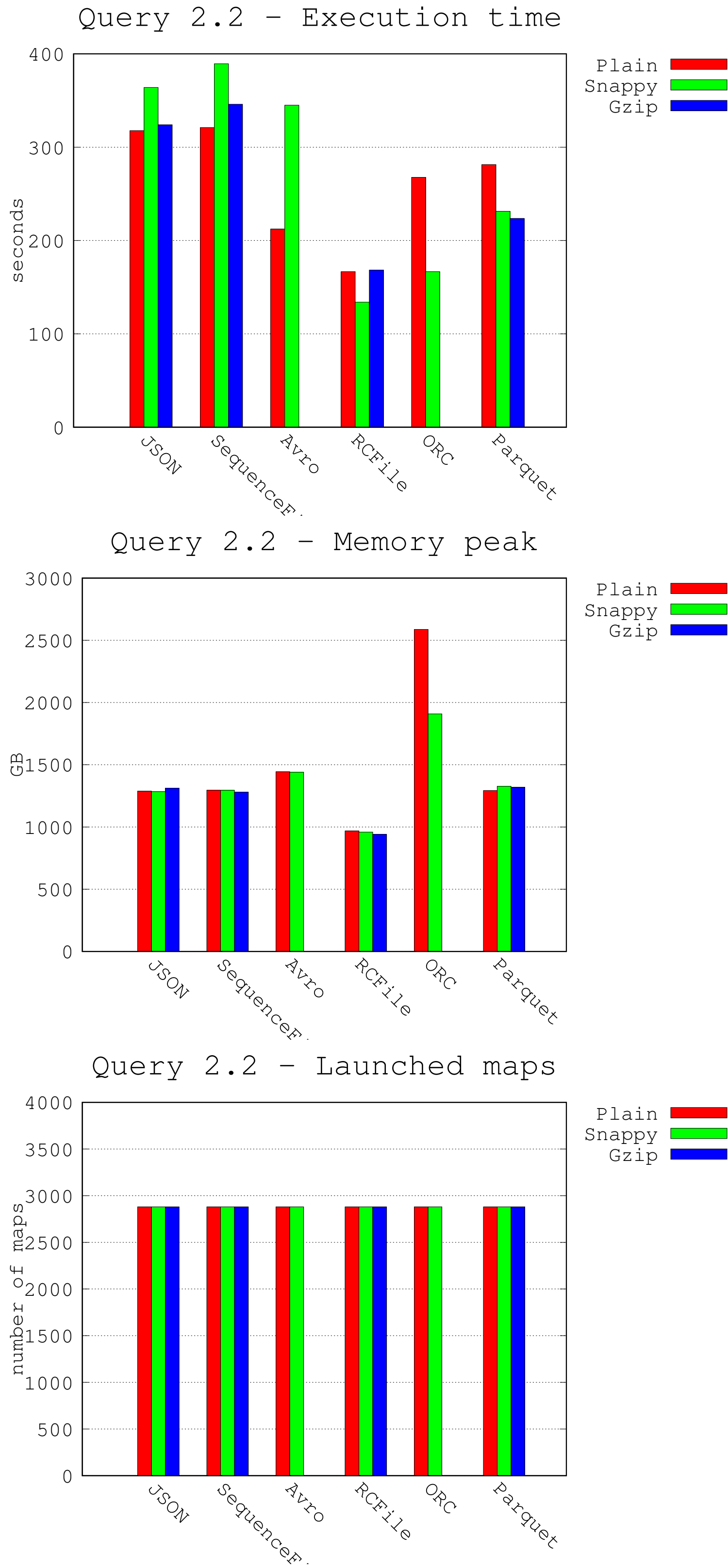


Figure 5.10: GitHub log query 2.2 performance

		run 1	run 2	run 3	Average	Desv.	VC
JSON-Plain	**time (sec)**	328	316	309	317.6667	9.609	0.0302
	mem.(GB)	1288.0167	1317.7313	1276.5917	1294.1132	21.2366	0.0164
	maps	2880	2880	2880	2880	0	0
JSON-Snappy	**time (sec)**	367	364	361	364	3	0.0082
	mem.(GB)	1283.982	1238.0154	1296.4993	1272.8322	30.7949	0.0242
	maps	2880	2880	2880	2880	0	0
JSON-Gzip	**time (sec)**	218	218	236	224	10.3923	0.0464
	mem.(GB)	1312.5817	1314.1962	1386.8844	1337.8874	42.4403	0.0317
	maps	2880	2880	2880	2880	0	0
SeqF-Plain	**time (sec)**	323	312	298	311	12.53	0.0403
	mem.(GB)	1295.4929	1274.8298	1252.6477	1274.3235	21.4271	0.0168
	maps	2880	2880	2880	2880	0	0
SeqF-Snappy	**time (sec)**	329	327	332	329.3333	2.5166	0.0076
	mem.(GB)	1294.8008	1244.692	1267.6565	1269.0498	25.0834	0.0198
	maps	2880	2880	2880	2880	0	0
SeqF-Gzip	**time (sec)**	344	316	318	326	15.6205	0.0479
	mem.(GB)	1280.12	1267.562	1273.152	1273.6114	6.2916	0.0049
	maps	2880	2880	2880	2880	0	0
Avro-Plain	**time (sec)**	197	212	228	212.3333	15.5027	0.073
	mem.(GB)	1443.8832	1485.8713	1434.3562	1454.7036	27.4092	0.0188
	maps	2880	2880	2880	2880	0	0
Avro-Snappy	**time (sec)**	347	343	345	345	2	0.0058
	mem.(GB)	1440.9226	1371.1675	1489.986	1434.0253	59.7088	0.0416
	maps	2880	2880	2880	2880	0	0
RCFile-Plain	**time (sec)**	173	165	162	166.6667	5.6862	0.0341
	mem.(GB)	968.4505	1017.2895	1049.3545	1011.6982	40.7407	0.0403
	maps	2880	2880	2880	2880	0	0
RCFile-Snappy	**time (sec)**	140	124	138	134	8.7178	0.0651
	mem.(GB)	959.1721	1003.8599	956.4677	973.1666	26.6156	0.0273
	maps	2880	2880	2880	2880	0	0
RCFile-Gzip	**time (sec)**	175	165	165	168.3333	5.7735	0.0343
	mem.(GB)	940.3637	979.2196	996.5811	972.0548	28.7854	0.0296
	maps	2880	2880	2880	2880	0	0
ORC-Plain	**time (sec)**	280	267	256	267.6667	12.0139	0.0449
	mem.(GB)	2587.2373	2529.7489	2491.4485	2536.1449	48.2136	0.019
	maps	2880	2880	2880	2880	0	0
ORC-Snappy	**time (sec)**	170	168	162	166.6667	4.1633	0.025
	mem.(GB)	1908.429	1867.2642	1940.1249	1905.2727	36.5327	0.0192
	maps	2880	2880	2880	2880	0	0
Parquet-Plain	**time (sec)**	279	275	290	281.3333	7.7675	0.0276
	mem.(GB)	1292.8724	1260.7316	1279.8947	1277.8329	16.1693	0.0127
	maps	2880	2880	2880	2880	0	0
Parquet-Snappy	**time (sec)**	228	229	237	231.3333	4.9329	0.0213
	mem.(GB)	1326.3562	1367.3937	1406.5012	1366.7504	40.0763	0.0293
	maps	2880	2880	2880	2880	0	0
Parquet-Gzip	**time (sec)**	227	225	219	223.6667	4.1633	0.0186
	mem.(GB)	1319.0871	1380.6884	1432.5885	1377.4547	56.8198	0.0412
	maps	2880	2880	2880	2880	0	0

Table 5.10: GitHub log query 2.2 results

Query 3

Now we start with the Hive queries. The first one is the query 3. This is the same count query done in the Opensky dataset. For having good performance it is important to have metadata in the file header. The results of this query are shown in Figure 5.11 and Table 5.11.

First of all we can notice that the number of map tasks is now different from the ones obtained in the MapReduce queries. This is because Hive do not use the same configuration as the default MapReduce. Here, each map task processes more than one single file. Thus the number of map tasks is reduced under 200 for almost all the formats when using compression.

Considering general performance, once again the best formats for this counting query are the columnar ones. This is because of their metadata. The human-readable formats do not have any information about the number of rows inside each block. Thus, they need to read the records to count how many there are.

We can notice that the human readable formats have a better performance, in execution time, memory peak and number of maps, when they are used with a compression codec. This is because the number of map tasks is much lower when using compression codecs. In the previous queries this did not happen because the number of map tasks was the same for all. Also, the data does not need to be decompressed. It is only necessary to count the records, we do not want to know what is inside them.

Avro is again different from the rest of formats, as it has a very similar execution time when it uses Snappy compression than when it uses no compression. Although, the memory usage and the number of maps are much smaller when using the compression codec. Even it has information about the number of records inside each block, it seems that it does not use that metadata, as it has almost the same execution time as JSON without using compression (just 5.37% faster), and 95.34% slower when using Snappy.

The best formats in every aspect for this query, are the column-oriented ones. They have the smallest files and they have information about the number of records inside their metadata. The slowest of the three is Parquet (over 100 seconds with no compression), unlike when we were using the Opensky data, when it was the fastest. This could be because the number of records information is inside the metadata of each page; and there is, at least, one page for every column chunk, and one column chunk for each column inside a row group. ORC and RCFile have this information in the row group metadata. Thus, Parquet has to read more than 200 headers metadata for each row group, while RCFile and ORC just read one value.

If we have to select one best format for this query would be ORC. It is the fastest one, lasting less than 50 seconds with or without compression, so it has a good management of the metadata, and it uses very few memory (the only one under 200GB with no compression, and under 100GB for Snappy).

The use of resources is much smaller than the one seen in the three previous queries, having all memory peaks under 600GB, unlike the MapReduce queries, in which the memory peaks were even higher than 2000GB. This is mostly because of the much smaller number of map tasks.

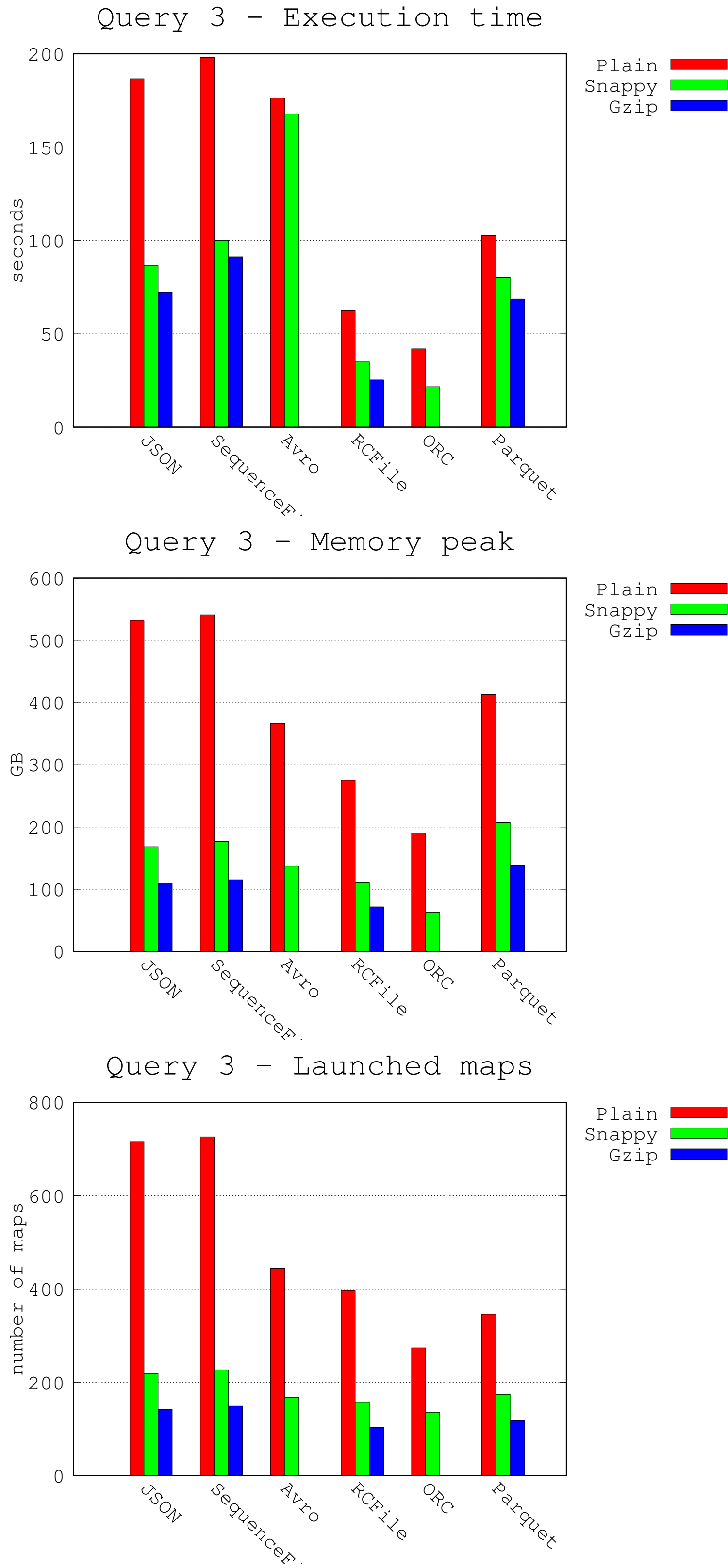


Figure 5.11: GitHub log query 3 performance

		run 1	run 2	run 3	Average	Desv.	VC
JSON-Plain	**time (sec)**	184	181	195	186.6667	7.3711	0.0395
	mem.(GB)	531.8244	514.0508	511.7376	519.2043	10.9904	0.0212
	maps	716	716	716	716	0	0
JSON-Snappy	**time (sec)**	93	86	81	86.6667	6.0277	0.0696
	mem.(GB)	168.3842	162.8932	168.184	166.4871	3.114	0.0187
	maps	219	219	219	219	0	0
JSON-Gzip	**time (sec)**	64	63	60	62.3333	2.0817	0.0334
	mem.(GB)	109.563	111.2262	108.2453	109.6782	1.4938	0.0136
	maps	142	142	142	142	0	0
SeqF-Plain	**time (sec)**	199	200	195	198	2.6458	0.0134
	mem.(GB)	540.9447	547.1223	526.058	538.0417	10.828	0.0201
	maps	726	726	726	726	0	0
SeqF-Snappy	**time (sec)**	107	93	100	100	7	0.07
	mem.(GB)	176.6223	180.9496	190.7824	182.7848	7.2562	0.0397
	maps	227	227	227	227	0	0
SeqF-Gzip	**time (sec)**	70	68	68	68.6667	1.1547	0.0168
	mem.(GB)	115.3382	116.5712	117.1774	116.3623	0.9372	0.0081
	maps	149	149	149	149	0	0
Avro-Plain	**time (sec)**	181	171	177	176.3333	5.0332	0.0285
	mem.(GB)	366.4475	357.301	363.5788	362.4424	4.6779	0.0129
	maps	444	444	444	444	0	0
Avro-Snappy	**time (sec)**	169	169	165	167.6667	2.3094	0.0138
	mem.(GB)	136.9774	139.9114	143.6536	140.1808	3.3463	0.0239
	maps	168	168	168	168	0	0
RCFile-Plain	**time (sec)**	66	59	62	62.3333	3.5119	0.0563
	mem.(GB)	275.4747	290.7746	293.6242	286.6245	9.7605	0.0341
	maps	396	396	396	396	0	0
RCFile-Snappy	**time (sec)**	33	37	35	35	2	0.0571
	mem.(GB)	110.262	109.5541	114.2463	111.3542	2.5296	0.0227
	maps	158	158	158	158	0	0
RCFile-Gzip	**time (sec)**	27	26	23	25.3333	2.0817	0.0822
	mem.(GB)	71.6812	69.3358	67.5511	69.5227	2.0714	0.0298
	maps	103	103	103	103	0	0
ORC-Plain	**time (sec)**	44	44	38	42	3.4641	0.0825
	mem.(GB)	190.3954	184.8663	179.1133	184.7917	5.6414	0.0305
	maps	274	274	274	274	0	0
ORC-Snappy	**time (sec)**	22	22	21	21.6667	0.5774	0.0266
	mem.(GB)	62.7575	64.3835	65.5450	64.2287	1.4002	0.0218
	maps	135	135	135	135	0	0
Parquet-Plain	**time (sec)**	159	158	141	152.6667	10.116	0.0663
	mem.(GB)	413.0379	398.437	403.3696	404.9482	7.4273	0.0183
	maps	346	346	346	346	0	0
Parquet-Snappy	**time (sec)**	103	99	102	101.3333	2.0817	0.0205
	mem.(GB)	207.1419	210.1475	210.3051	209.1982	1.7825	0.0085
	maps	174	174	174	174	0	0
Parquet-Gzip	**time (sec)**	89	88	97	91.3333	4.9329	0.054
	mem.(GB)	138.6923	134.3401	139.7849	137.6058	2.8804	0.0209
	maps	119	119	119	119	0	0

Table 5.11: GitHub log query 3 results

Query 4

The next query is the query 4. This is a single record search. To find that row we use the column "*id*", more specifically the record with the value "7044822816". The performance results of this query can be seen in Figure 5.12 and Table 5.12.

We can observe that, as there are very few columns affected, the most efficient usage of resources and time is reached by the column-oriented formats. It is obvious that if there are very few columns needed to find the record, in these formats it can be avoided the unnecessary read of all the other columns; even if for the final record all the fields are requested.

Again, the number of map tasks is directly proportional with the execution time and with the memory usage. Except for Avro, that last 6% more using Snappy codec than when it uses no compression. Taking into account all the previous queries, this is probably because the Snappy decompression is very slow for the Avro serialized data in a many complex columns situation; we do not know if this statement can be generalized for other scenarios as in the Opensky dataset this performance worsening with compression did not happen.

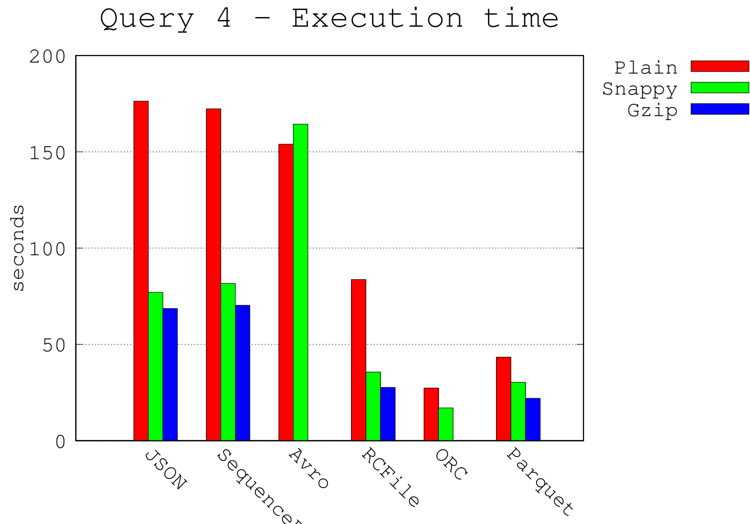
The row-oriented formats are much slower than the column-oriented ones. The columnar formats have an execution time under than 50 seconds (except for RCFile with no compression), and the row ones do not lower that time in any case. This is because the columnar files do not need to read more than 200 columns just to access to one or two of them. Thus, RCFile, ORC and Parquet read only the column "*id*", and only if they find the requested record, read the rest of columns to return the entire row. This allows to skip a great number of unnecessary data reads. The row formats can not skip columns, because of their storage orientation.

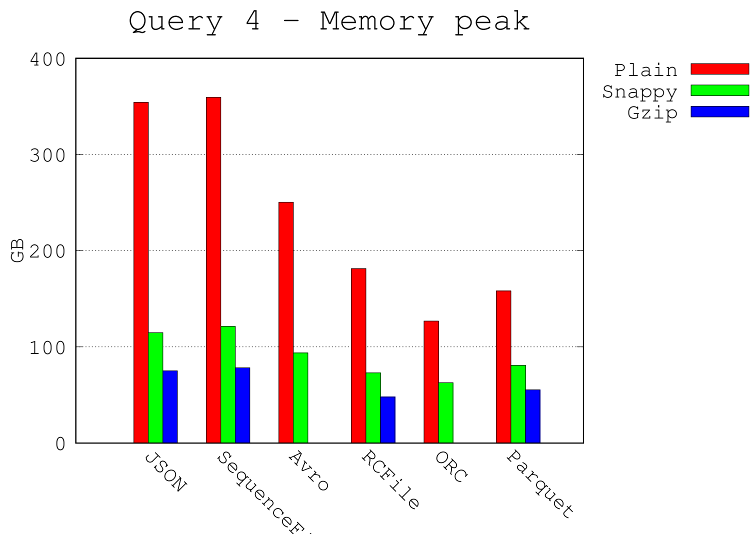
Inside the columnar formats group, ORC is fastest one, being 36.92% faster than Parquet, and 66.33% faster than RCFile, both without using compression. Even, ORC using Snappy compression is faster than any other combination (with only 17 seconds). The reason is probably that it was specifically designed for Hive and MapReduce, so it is feasible that ORC is faster than Parquet. And also, as it is an optimization of RCFile, it is not surprising that ORC is faster.

Query 5.1

The query 5 is divided into two different queries, as we have previously done with query 2. We will make a filter by the column "*id*" in this query 5.1, and in the next one we will filter by a very nested column. The filter done here is to select the *ids* higher than "7607000000". This is a very restrictive query, that very few records accomplish. The rest of the columns are not requested. We can find the results of this query in Figure 5.13 and Table 5.13.

The performances of this query are very similar to the ones seen in previous ones. The best execution times and usage of resources are obtained in the column-oriented formats, and the row ones are up 5.63 times slower, comparing, for example ORC and JSON with Snappy compression. Avro continues having a worse execution time with Snappy (9.52% slower), so the theory of slow decompression for its serialization in this type of data is stronger.





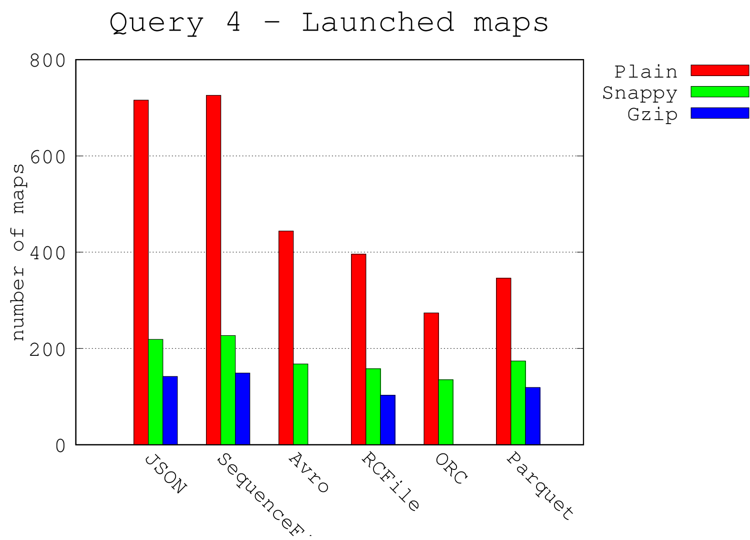


Figure 5.12: GitHub log query 4 performance

		run 1	run 2	run 3	Average	Desv.	VC
JSON-Plain	**time (sec)**	175	172	182	176.3333	5.1316	0.0291
	mem.(GB)	354.1893	347.7608	353.4189	351.7897	3.5103	0.01
	maps	716	716	716	716	0	0
JSON-Snappy	**time (sec)**	81	79	71	77	5.2915	0.0687
	mem.(GB)	114.8108	111.9635	107.5566	111.4436	3.6549	0.0328
	maps	219	219	219	219	0	0
JSON-Gzip	**time (sec)**	63	56	57	58.6667	3.7859	0.0645
	mem.(GB)	75.0492	75.2893	73.9718	74.7701	0.7017	0.0094
	maps	142	142	142	142	0	0
SeqF-Plain	**time (sec)**	175	175	167	172.3333	4.6188	0.0268
	mem.(GB)	359.5041	360.9134	345.506	355.3078	8.5178	0.024
	maps	726	726	726	726	0	0
SeqF-Snappy	**time (sec)**	84	80	81	81.6667	2.0817	0.0255
	mem.(GB)	121.1713	127.3365	128.4838	125.6639	3.9327	0.0313
	maps	227	227	227	227	0	0
SeqF-Gzip	**time (sec)**	60	61	60	60.3333	0.5774	0.0096
	mem.(GB)	78.2657	75.4434	74.3065	76.0052	2.0385	0.0268
	maps	149	149	149	149	0	0
Avro-Plain	**time (sec)**	150	156	156	154	3.4641	0.0225
	mem.(GB)	250.3505	249.6621	247.8245	249.279	1.3058	0.0052
	maps	444	444	444	444	0	0
Avro-Snappy	**time (sec)**	162	168	163	164.3333	3.2146	0.0196
	mem.(GB)	93.7144	94.5494	91.3228	93.1955	1.6747	0.018
	maps	168	168	168	168	0	0
RCFile-Plain	**time (sec)**	85	82	84	83.6667	1.5275	0.0183
	mem.(GB)	181.4096	189.798	197.7733	189.6603	8.1827	0.0431
	maps	396	396	396	396	0	0
RCFile-Snappy	**time (sec)**	33	37	37	35.6667	2.3094	0.0647
	mem.(GB)	72.9717	73.7518	78.1289	74.9508	2.7798	0.0371
	maps	158	158	158	158	0	0
RCFile-Gzip	**time (sec)**	29	28	26	27.6667	1.5275	0.0552
	mem.(GB)	47.912	50.2697	52.4891	50.2236	2.2889	0.0456
	maps	103	103	103	103	0	0
ORC-Plain	**time (sec)**	27	28	27	27.3333	0.5774	0.0211
	mem.(GB)	126.8697	130.6313	131.8044	129.7685	2.5781	0.0199
	maps	274	274	274	274	0	0
ORC-Snappy	**time (sec)**	18	17	16	17	1	0.0588
	mem.(GB)	62.7628	59.9316	59.3725	60.689	1.8177	0.03
	maps	135	135	135	135	0	0
Parquet-Plain	**time (sec)**	39	46	45	43.3333	3.7859	0.0874
	mem.(GB)	158.1578	150.8525	155.8548	154.955	3.7348	0.0241
	maps	346	346	346	346	0	0
Parquet-Snappy	**time (sec)**	31	32	28	30.3333	2.0817	0.0686
	mem.(GB)	80.8347	80.5275	80.4824	80.6149	0.1917	0.0024
	maps	174	174	174	174	0	0
Parquet-Gzip	**time (sec)**	23	22	21	22	1	0.0455
	mem.(GB)	55.3364	56.1145	58.3085	56.5865	1.5413	0.0272
	maps	119	119	119	119	0	0

Table 5.12: GitHub log query 4 results

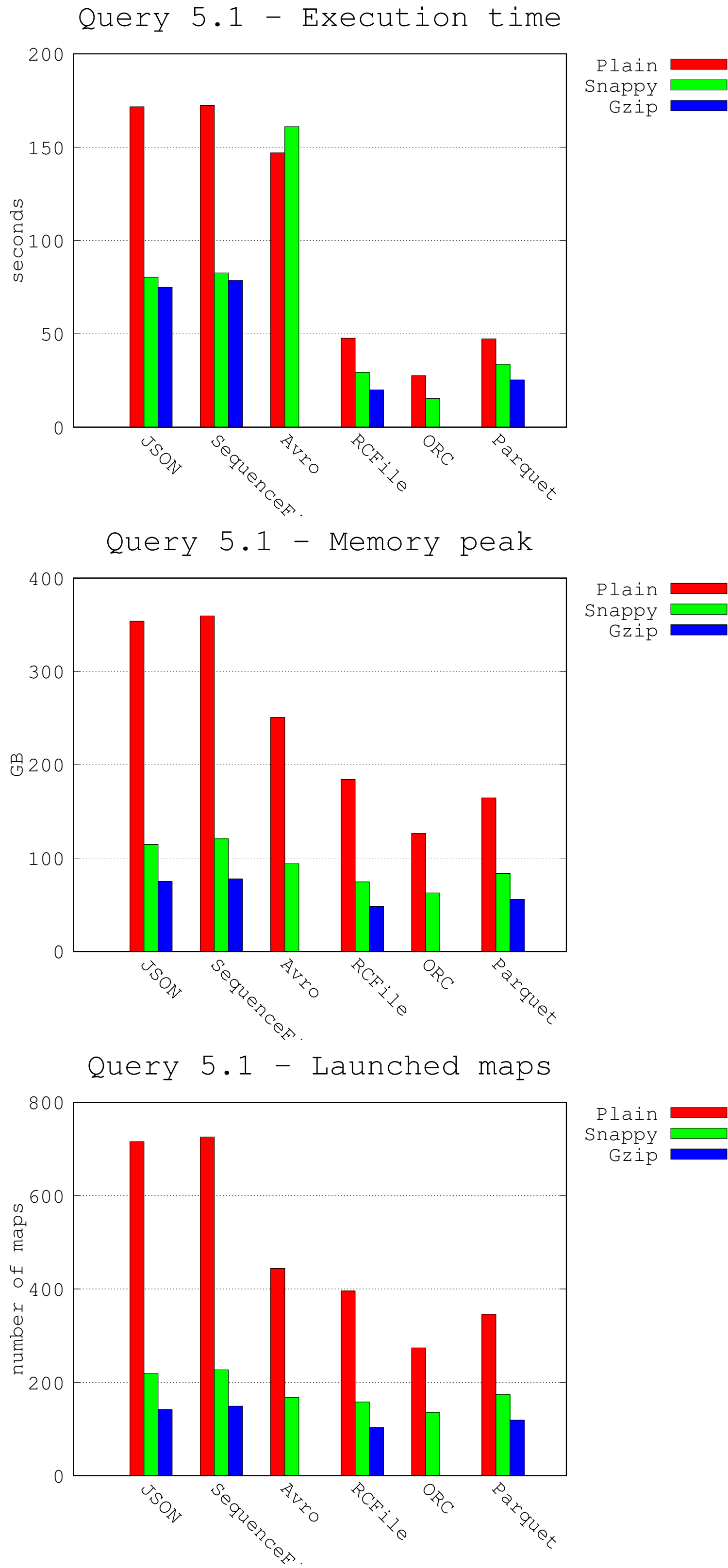


Figure 5.13: GitHub log query 5.1 performance

		run 1	run 2	run 3	Average	Desv.	VC
JSON-Plain	**time (sec)**	178	173	170	173.6667	4.0415	0.0233
	mem.(GB)	353.9083	366.3234	357.8907	359.3742	6.3391	0.0176
	maps	716	716	716	716	0	0
JSON-Snappy	**time (sec)**	87	93	79	86.3333	7.0238	0.0814
	mem.(GB)	114.6645	111.7646	117.1561	114.5284	2.6983	0.0236
	maps	219	219	219	219	0	0
JSON-Gzip	**time (sec)**	59	56	50	55	4.5826	0.0833
	mem.(GB)	75.0104	77.2089	81.3612	77.8602	3.2251	0.0414
	maps	142	142	142	142	0	0
SeqF-Plain	**time (sec)**	174	170	173	172.3333	2.0817	0.0121
	mem.(GB)	359.4693	357.388	370.5398	362.4657	7.0694	0.0195
	maps	726	726	726	726	0	0
SeqF-Snappy	**time (sec)**	82	88	78	82.6667	5.0332	0.0609
	mem.(GB)	120.6574	118.2539	115.2089	118.0401	2.7306	0.0231
	maps	227	227	227	227	0	0
SeqF-Gzip	**time (sec)**	62	63	63	62.6667	0.5774	0.0092
	mem.(GB)	77.8691	81.5406	82.9186	80.7761	2.6101	0.0323
	maps	149	149	149	149	0	0
Avro-Plain	**time (sec)**	147	146	148	147	1	0.0068
	mem.(GB)	250.5735	246.5192	238.638	245.2436	6.0691	0.0247
	maps	444	444	444	444	0	0
Avro-Snappy	**time (sec)**	158	164	161	161	3	0.0186
	mem.(GB)	93.9889	92.6881	95.6732	94.1168	1.4966	0.0159
	maps	168	168	168	168	0	0
RCFile-Plain	**time (sec)**	46	52	45	47.6667	3.7859	0.0794
	mem.(GB)	184.2929	180.8098	177.4051	180.836	3.444	0.019
	maps	396	396	396	396	0	0
RCFile-Snappy	**time (sec)**	24	28	36	29.3333	6.1101	0.2083
	mem.(GB)	74.4698	78.2291	76.6113	76.4367	1.8857	0.0247
	maps	158	158	158	158	0	0
RCFile-Gzip	**time (sec)**	21	20	19	20	1	0.05
	mem.(GB)	48.0471	46.7311	48.0045	47.5942	0.7478	0.0157
	maps	103	103	103	103	0	0
ORC-Plain	**time (sec)**	25	28	30	27.6667	2.5166	0.091
	mem.(GB)	126.4434	126.0325	122.2162	124.8974	2.331	0.0187
	maps	274	274	274	274	0	0
ORC-Snappy	**time (sec)**	15	15	16	15.3333	0.5774	0.0377
	mem.(GB)	62.7809	65.0781	67.8062	65.2217	2.5157	0.0386
	maps	135	135	135	135	0	0
Parquet-Plain	**time (sec)**	46	45	51	47.3333	3.2146	0.0679
	mem.(GB)	164.5961	163.7599	163.0394	163.7985	0.7791	0.0048
	maps	346	346	346	346	0	0
Parquet-Snappy	**time (sec)**	34	34	33	33.6667	0.5774	0.0171
	mem.(GB)	83.546	80.5893	76.9982	80.3778	3.279	0.0408
	maps	174	174	174	174	0	0
Parquet-Gzip	**time (sec)**	25	26	25	25.3333	0.5774	0.0228
	mem.(GB)	55.7879	57.4599	60.0766	57.7748	2.1616	0.0374
	maps	119	119	119	119	0	0

Table 5.13: GitHub log query 5.1 results

Once again, ORC is the format with better overall performance, with the best execution time and one of the lowest memory peaks and number of map tasks. However, RCFile also has a good performance, taking into account that this format do not have metadata to know the range of the values inside a column, being almost as fast as Parquet, with only 0.7% more seconds with no compression. Parquet and ORC do have information about maximum and minimum values of a column inside each block, but Parquet is 71.08% slower than ORC with no compression.

Query 5.2

Now that we have proved what happens with a simple column, we will make the filter by a very nested one. In this query we have requested the id for those records with a "*payload.pull_request.base.repo.owner.id*" value over "37000000". Again this is a very restrictive query, with very few records that fulfill it. Also, in this column there are a lot of null values. It is worth noting that that for this query we are requesting two columns (the *id* and *payload.pull_request.base.repo.owner.id*), but as there are more than 200, the overall results differences should not be significant. This results are shown in Figure 5.14 and Table 5.14.

The obtained performances are slightly different to the ones seen in the query 5.1. Even though the row-oriented formats numbers are roughly the same, the columnar ones are not.

The first fact to highlight is that Parquet has a much worse performance, being 2.79 times slower and using 139.74% more memory than in the previous query, when no compression is used. Even parquet is supposed to work well with complex data schemas, when we request a very nested column the time taken to resolve it is doubled, and the same happens with the memory peaks.

The other important fact that we can observe is that RCFile has been faster than ORC in this case (16.12% and 34.69% faster for plain and Snappy respectively), and it has also used less memory (34.82% and 47.36% less with no compression and with Snappy respectively). This is because of the mentioned situation of this format when storing complex types, that simulates a plain table with no complex type data. Thus, RCFile does not need to deal with a very nested column, for this format it is just a simple one.

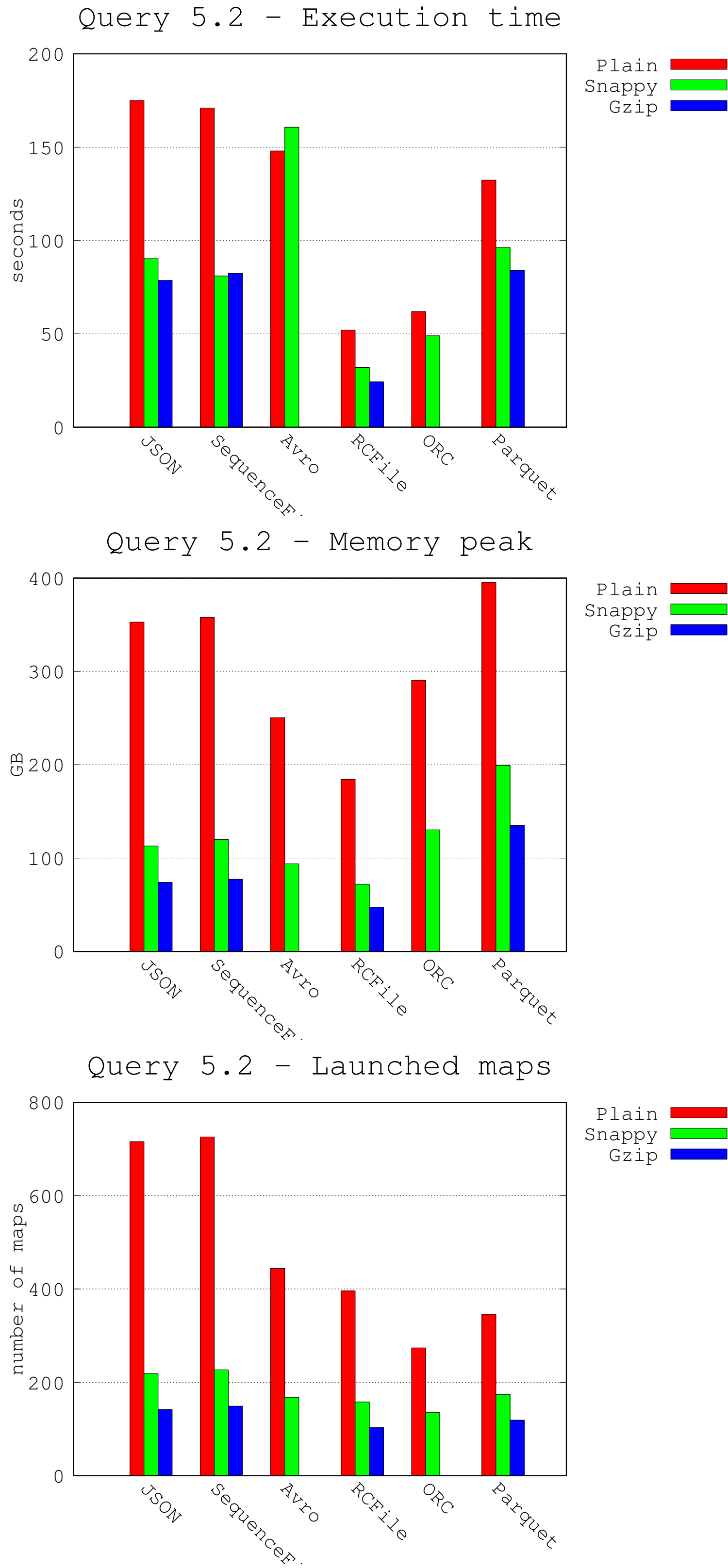


Figure 5.14: GitHub log query 5.2 performance

		run 1	run 2	run 3	Average	Desv.	VC
JSON-Plain	**time (sec)**	178	174	173	175	2.6458	0.0151
	mem.(GB)	352.8366	355.1195	354.0435	353.9999	1.1421	0.0032
	maps	716	716	716	716	0	0
JSON-Snappy	**time (sec)**	88	85	98	90.3333	6.8069	0.0754
	mem.(GB)	112.9644	113.4298	109.5652	111.9865	2.1097	0.0188
	maps	219	219	219	219	0	0
JSON-Gzip	**time (sec)**	60	57	59	58.6667	1.5275	0.026
	mem.(GB)	73.8045	73.986	75.3829	74.3911	0.8637	0.0116
	maps	142	142	142	142	0	0
SeqF-Plain	**time (sec)**	174	169	170	171	2.6458	0.0155
	mem.(GB)	357.8866	368.2331	377.542	367.8872	9.8323	0.0267
	maps	726	726	726	726	0	0
SeqF-Snappy	**time (sec)**	87	76	80	81	5.5678	0.0687
	mem.(GB)	119.8389	119.0264	121.3271	120.0641	1.1668	0.0097
	maps	227	227	227	227	0	0
SeqF-Gzip	**time (sec)**	61	64	62	62.3333	1.5275	0.0245
	mem.(GB)	77.4336	75.9879	76.0396	76.487	0.8202	0.0107
	maps	149	149	149	149	0	0
Avro-Plain	**time (sec)**	148	146	150	148	2	0.0135
	mem.(GB)	250.549	247.8456	237.9565	245.4504	6.6291	0.027
	maps	444	444	444	444	0	0
Avro-Snappy	**time (sec)**	158	160	164	160.6667	3.0551	0.019
	mem.(GB)	93.8958	90.8912	90.7165	91.8345	1.7873	0.0195
	maps	168	168	168	168	0	0
RCFile-Plain	**time (sec)**	56	52	48	52	4	0.0769
	mem.(GB)	184.434	190.1238	193.932	189.4966	4.7799	0.0252
	maps	396	396	396	396	0	0
RCFile-Snappy	**time (sec)**	32	31	33	32	1	0.0313
	mem.(GB)	72.0013	69.8125	69.8907	70.5682	1.2418	0.0176
	maps	158	158	158	158	0	0
RCFile-Gzip	**time (sec)**	24	26	23	24.3333	1.5275	0.0628
	mem.(GB)	47.4683	46.7259	49.35	47.8481	1.3527	0.0283
	maps	103	103	103	103	0	0
ORC-Plain	**time (sec)**	64	61	61	62	1.7321	0.0279
	mem.(GB)	290.5569	286.5443	295.2008	290.7673	4.3321	0.0149
	maps	274	274	274	274	0	0
ORC-Snappy	**time (sec)**	48	49	50	49	1	0.0204
	mem.(GB)	130.3437	135.6395	133.2441	133.0758	2.6519	0.0199
	maps	135	135	135	135	0	0
Parquet-Plain	**time (sec)**	129	123	145	132.3333	11.3725	0.0859
	mem.(GB)	395.1833	382.8418	400.0735	392.6995	8.8803	0.0226
	maps	346	346	346	346	0	0
Parquet-Snappy	**time (sec)**	101	95	93	96.3333	4.1633	0.0432
	mem.(GB)	199.4257	205.0834	198.0962	200.8684	3.7103	0.0185
	maps	174	174	174	174	0	0
Parquet-Gzip	**time (sec)**	87	82	83	84	2.6458	0.0315
	mem.(GB)	134.7944	133.2604	139.501	135.8519	3.2519	0.0239
	maps	119	119	119	119	0	0

Table 5.14: GitHub log query 5.2 results

Chapter 6

Conclusions

In this work we have studied the distributed file systems, focusing in HDFS and its file formats. In particular, we have made a study of the characteristics of each of the main formats of HDFS, comparing each one with a proposed framework. With this theoretical study, we are able to guess which format is better for a specific situation of data and queries. To check if the conclusions obtained are valid, we have done a experimental study, using two different datasets; one very simple, and another one with complex data types and nested structures. For each dataset we have made a set of simple queries using each different format with some compression options (Gzip and Snappy). With this experimentation we have enriched and verified the conclusions obtained with the theory.

First of all, the most important thing is to know that **there is there is no perfect format**. No one is better than the rest in all the situations for all kind of data, usage, and data structure.

In the first part of this work, in which we have studied the characteristics of the file formats, we have learned to make a first filter of which ones can fit our needs. For example, if we have a data with complex type columns, RCFile is not suitable for it, as it does not support complex types. Also, such theoretical study allowed us to make guesses about the expected performance of the different formats. For example, if we are going to store data to be processed with MapReduce jobs, and only few columns are needed for each process, the theoretically best formats should be ORC and RCFile. But if we want to use that data with many frameworks and tools, we would probably prefer Parquet, as it is easily usable with plenty of tools and frameworks.

However, this qualitative study is not enough in many situations. The theory in some cases does not reflect the reality, because very little details are omitted or not considered; or even there is a theoretical tie between some formats. Thus, it is helpful to have some performance number for some specific situations. With that results we generalized some conclusions to enrich the previous theoretical information.

With the experimentation we have learned some new things about the performance of the formats in a MapReduce environment. First of all, it is very important the number of map tasks. Despite having many maps allows a big parallelization, the number of maps should not be much greater than the parallelization capacity. If we have too many map tasks, we will loose lot of time with scheduling and starting/ending of tasks. Thus, the compression is crucial when working with big amounts of data. By using a compression codec we will have less map tasks, and thus, faster jobs with less usage of resources;

and also big saves of disk storage. Our conclusion is to **always use compression** when working with MapReduce.

Another very important aspect to consider is the number of columns, and how many of them would be usually requested when querying the data. If we need very few columns for our queries, the columnar formats have a better performance than the row-oriented ones. However, if the data is stored for making full scans of it, it is probably better to use a row-oriented format.

Focusing now in specific formats, it is important to highlight that in our experimentation Avro do not have a very good performance in many cases, not making a big compression with its serialization, and having execution times even slower than text formats. And what is more, it seems that the blocks metadata is not used, as for the count queries the execution times has been very slow; even it has stored the number or records per block.

On the opposite side, ORC has reached maybe the best results when using it with Hive. Nevertheless, when using it with our MapReduce scripts the memory peaks were much higher. This means that the configuration of Hive with ORC may be optimized. Therefore, if we want to use a column-oriented format with Hive, ORC could be our best option.

This change of performance between Hive and MapReduce default scripts led us to another conclusion. In many cases, the **configuration of a MapReduce** job can change completely the performance. An example of this is the difference between just processing one small file per map task, or process several. With several files per map task, the number of tasks would be smaller, and that can mean a better performance and faster queries.

6.1 Future work

Despite all the conclusions we have made, there are a lot of scenarios that we have not considered and we do not know which format is better in that cases.

We have selected relatively small datasets, that do not even reach 1TB of data, that is what we have considered Big Data nowadays. The decision of the data size has been made due to existing restrictions of time and cluster resources consumption, as several projects, not only ours, cohabit in the same cluster. A new experimental study with larger datasets will be made soon on a dedicated cluster with larger resources in order to, first, confirm our current conclusions, and second, the perform scalability experiments with different sizes on the same datasets.

Also, we have selected very simple queries, that in the majority of cases only affects one or two columns. Some other more complex queries cloud be of interest, as not always simple queries are needed. For example the usage of more than one restriction could lead to different performance results.

Another characteristic of the selected scenario for the experimentation is that have just focused on MapReduce and Hive. However, there are more computation frameworks that could be included in the study. For example, Spark [14] has become popular, as it is better when making iterative jobs or interactive analytics.

Bibliography

[1] Oxford english dictionary (oed). http://www.oed.com/. Accessed: 2018-06-26.

[2] Doug Laney. 3d data management: Controlling data volume, velocity and variety. *META Group Research Note*, 6(70), 2001.

[3] Edd Dumbill. What is big data? an introduction to the big data landscape. *oreilly. com, https://www.oreilly.com/ideas/what-is-big-data*, 2012.

[4] Youtube. Press - youtube. https://www.youtube.com/yt/about/press/. Accessed: 2018-06-26.

[5] Bernard Marr. Big data: The 5 vs everyone must know. *LinkedIn Pulse*, 6, 2014.

[6] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[7] Apache hadoop. http://hadoop.apache.org/. Accessed: 2018-06-26.

[8] Konstantin V Shvachko. Hdfs scalability: The limits to growth. *; login:: the magazine of USENIX & SAGE*, 35(2):6–16, 2010.

[9] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] Daiga Plase, Laila Niedrite, and Romans Taranovs. A comparison of hdfs compact data formats: Avro versus parquet. *Mokslas–Lietuvos ateitis/Science–Future of Lithuania*, 9(3):267–276, 2017.

[12] Transaction Processing Performance Council. Tpc-h benchmark specification. *Published at http://www. tcp.org/hspec.html*, 21:592–603, 2008.

[13] Gavin So. *Comparing Performance of Parquet and JSON Files for a Particle Swarm Optimization Algorithm on Apache Spark*. PhD thesis, California State University, Northridge, 2016.

[14] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[15] Zbigniew Baranowski, Luca Canali, Rainer Toebbicke, Julius Hrivnac, and Dario Barberis. A study of data representation in hadoop to optimize data storage and search performance for the atlas eventindex. In *Journal of Physics: Conference Series*, volume 898, page 062020. IOP Publishing, 2017.

[16] Lars George. *HBase: the definitive guide: random access to your planet-size data*. " O'Reilly Media, Inc.", 2011.

[17] Owen O'Malley. File format benchmark - avro, json, orc and parquet. https://www.slideshare.net/oom65/file-format-benchmarks-avro-json-orc-parquet. Accessed: 2018-06-26.

[18] Stephen O'Sullivan. Choosing an hdfs data storage format: Avro vs parquet and more. https://www.slideshare.net/StampedeCon/choosing-an-hdfs-data-storage-format-avro-vs-parquet-and-more-stampedecon-2015. Accessed: 2018-06-26.

[19] S. Oliveros S. O'Sullivan. How to choose a data format. https://www.svds.com/how-to-choose-a-data-format/. Accessed: 2018-06-26.

[20] Egon G Guba and Yvonna S Lincoln. Epistemological and methodological bases of naturalistic inquiry. *ECTJ*, 30(4):233–252, 1982.

[21] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys (CSUR)*, 22(4):321–374, 1990.

[22] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[23] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[24] Jon Postel and Joyce Reynolds. File transfer protocol. 1985.

[25] David R. Brownbridge, Lindsay F Marshall, and Brian Randell. The newcastle connection or unixes of the world unite! *Software: Practice and Experience*, 12(12):1147–1162, 1982.

[26] Alex Davies and Alessandro Orsaria. Scale out with glusterfs. *Linux Journal*, 2013(235):1, 2013.

[27] Liba Svobodova. File servers for network-based distributed systems. *ACM Computing Surveys (CSUR)*, 16(4):353–398, 1984.

[28] KV Shvachko and AC Murthy. Scaling hadoop to 4000 nodes at yahoo! *Yahoo! Developer Network Blog,*, 2008.

[29] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[30] L Peter Deutsch. Gzip file format specification version 4.3. 1996.

[31] L Peter Deutsch. Deflate compressed data format specification version 1.3. 1996.

[32] Julian Seward. bzip2 and libbzip2. http://www.bzip.org, 1996. Accessed: 2018-06-26.

[33] Snappy: A fast compressor/decompressor. http://google.github.io/snappy/. Accessed: 2018-06-26.

[34] M. F. Oberhumer. Lzo. http://www.oberhumer.com/opensource/lzo/. Accessed: 2018-06-26.

[35] Cloudera Engineering Blog. Hadoop i/o: Sequence, map, set, array, bloommap files. http://blog.cloudera.com/blog/2011/01 /hadoop-io-sequence-map-set-array-bloommap-files. Accessed: 2018-06-26.

[36] Hadoop Wiki. Sequencefile. https://wiki.apache.org/hadoop/SequenceFile. Accessed: 2018-06-26.

[37] Apache avro. https://avro.apache.org/. Accessed: 2018-06-26.

[38] Apache avro 1.8.2 specification. http://avro.apache.org/docs/current/spec.html. Accessed: 2018-06-26.

[39] Trevni specification. http://avro.apache.org/docs///1.8.1/trevni/spec.html. Accessed: 2018-06-26.

[40] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1199–1208. IEEE, 2011.

[41] Dick Pountain. Run-length encoding. *Byte*, 12(6):317–319, 1987.

[42] Apache orc. https://orc.apache.org/. Accessed: 2018-06-26.

[43] Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. Technical report, 1996.

[44] Apache parquet. https://parquet.apache.org/. Accessed: 2018-06-26.

[45] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[46] Twitter Blog. Dremel made simple with parquet. https://blog.twitter.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet.html. Accessed: 2018-06-26.

[47] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[48] Apache hive. https://hive.apache.org/. Accessed: 2018-06-26.

[49] Ads-b technologies. http://www.ads-b.com/. Accessed: 2018-06-26.

[50] Matthias Schäfer, Martin Strohmeier, Vincent Lenders, Ivan Martinovic, and Matthias Wilhelm. Bringing up opensky: A large-scale ads-b sensor network for research. In *Proceedings of the 13th international symposium on Information processing in sensor networks*, pages 83–94. IEEE Press, 2014.

[51] Gh archive. https://www.gharchive.org/. Accessed: 2018-06-26.

Part I

APPENDICES

Appendix A

Opensky Schema

```
opensky
{
    string id;
    string time;
    string icao24;
    string callsign;
    string origin_country;
    double time_position;
    double time_velocity;
    double longitude;
    double latitude;
    double altitude;
    boolean on_ground;
    double velocity;
    double heading;
    double vertical_rate;
    string sensors;
    double baro_altitude;
    double squawk;
    boolean spi;
    string position_source;
    string date;
}
```

Appendix B

GitHub log Schema

```
message github
{
    struct actor
    {
        string avatar_url;
        string display_login;
        string gravatar_id;
        string id;
        string login;
        string url;
    }
    string created_at;
    string id;
    struct org
    {
        string avatar_url;
        string gravatar_id;
        string id;
        string login;
        string url;
    }
    struct payload
    {
        string action;
        array commits
        {
            struct author
            {
                string name;
            }
            string message;
        }
        struct forkee
```

```
{
    string archive_url;
    string assignees_url;
    string blobs_url;
    string branches_url;
    string clone_url;
    string collaborators_url;
    string comments_url;
    string commits_url;
    string compare_url;
    string contents_url;
    string contributors_url;
    string created_at;
    string default_branch;
    string deployments_url;
    string description;
    string downloads_url;
    string events_url;
    boolean fork;
    int32 forks;
    int32 forks_count;
    string forks_url;
    string full_name;
    string git_commits_url;
    string git_refs_url;
    string git_tags_url;
    string git_url;
    boolean has_downloads;
    boolean has_issues;
    boolean has_pages;
    boolean has_projects;
    boolean has_wiki;
    string homepage;
    string hooks_url;
    string html_url;
    string id;
    string issue_comment_url;
    string issue_events_url;
    string issues_url;
    string keys_url;
    string labels_url;
    string language;
    string languages_url;
    string merges_url;
    string milestones_url;
```

```
        string mirror_url;
        string name;
        string notifications_url;
        int32 open_issues;
        int32 open_issues_count;
        struct owner
        {
                string avatar_url;
                string events_url;
                string followers_url;
                string following_url;
                string gists_url;
                string gravatar_id;
                string html_url;
                string id;
                string login;
                string organizations_url;
                string received_events_url;
                string repos_url;
                boolean site_admin;
                string starred_url;
                string subscriptions_url;
                string type;
                string url;
        }
        boolean private;
        boolean public;
        string pulls_url;
        string pushed_at;
        string releases_url;
        int32 size;
        string ssh_url;
        string stargazers_url;
        string statuses_url;
        string subscribers_url;
        string subscription_url;
        string svn_url;
        string tags_url;
        string trees_url;
        string updated_at;
        string url;
        int32 watchers;
        int32 watchers_count;
}
string head;
```

```
struct issue
{
	struct assignee
	{
		string avatar_url;
		string events_url;
		string gists_url;
		string html_url;
		string id;
		string login;
		string organizations_url;
		string repos_url;
		boolean site_admin;
		string starred_url;
		string subscriptions_url;
		string type;
		string url;
	}
	array assignees
	{
		string avatar_url;
		string events_url;
		string followers_url;
		string following_url;
		string gists_url;
		string gravatar_id;
		string html_url;
		string id;
		string login;
		string organizations_url;
		string received_events_url;
		string repos_url;
		boolean site_admin;
		string starred_url;
		string subscriptions_url;
		string type;
		string url;
	}
	string author_association;
	string body;
	string closed_at;
	int32 comments;
	string comments_url;
	string created_at;
	string events_url;
```

```
string html_url;
string id;
array labels
{
        string color;
        boolean default;
        string id;
        string name;
        string url;
}
string labels_url;
boolean locked;
struct milestone
{
        string closed_at;
        int32 closed_issues;
        string created_at;
        struct creator
        {
                string avatar_url;
                string events_url;
                string followers_url;
                string following_url;
                string gists_url;
                string gravatar_id;
                string html_url;
                string id;
                string login;
                string organizations_url;
                string received_events_url;
                string repos_url;
                boolean site_admin;
                string starred_url;
                string subscriptions_url;
                string type;
                string url;
        }
        string description;
        string due_on;
        string html_url;
        string id;
        string labels_url;
        int32 number;
        int32 open_issues;
        string state;
```

```
            string title;
            string updated_at;
            string url;
        }
        int32 number;
        struct pull_request
        {
            string diff_url;
            string html_url;
            string patch_url;
            string url;
        }
        string repository_url;
        string state;
        string title;
        string updated_at;
        string url;
        struct user
        {
            string avatar_url;
            string events_url;
            string followers_url;
            string following_url;
            string gists_url;
            string gravatar_id;
            string html_url;
            string id;
            string login;
            string repos_url;
            boolean site_admin;
            string starred_url;
            string subscriptions_url;
            string type;
            string url;
        }
    }
    string master_branch;
    int32 number;
    array pages
    {
        string action;
        string title;
    }
    struct pull_request
    {
```

```
        struct base
        {
                string ref;
                struct repo
                {
                        string id;
                        struct owner
                        {
                                string id;
                                string login;
                        }
                        string url;
                }
                struct user
                {
                        string id;
                        string login;
                }
        }
        string id;
        struct user
        {
                string id;
                string login;
        }
}
string push_id;
struct release
{
        array assets
        {
                string id;
                string label;
                string name;
                int32 size;
                struct uploader
                {
                        string id;
                        string login;
                        string url;
                }
                string url;
        }
        struct author
        {
```

```
                        string id;
                        string login;
                    }
                    string id;
                }
                int32 size;
            }
            boolean public;
            struct repo
            {
                string id;
                string name;
                string url;
            }
            string type;
        }
```