



POLITECNICO DI TORINO

Degree Course in Computer Engineering

Master Thesis

Reengineering of a Big Data architecture for real-time ingestion and data analysis

Advisor

Prof. Elena BARALIS

Candidate

Roberto FORTINO

Company Tutor

Engr. Marco GATTA

April 2018

Summary

This thesis is based on the work done during the internship at *Data Reply*, a consulting company of Reply group, focused on Big Data and Data Analytics.

The thesis shows the study of developed solution and its application in a production environment.

Added value has been focused on reengineering the solution used by the client at the beginning of the activity, with a particular focus on the ingestion layer and the storage layer.

Contents

1	INTRODUCTION	5
1.1	Personal Effort	6
1.2	Thesis Structure	6
2	THE HADOOP ECOSYSTEM	7
2.1	Introduction to Big Data	7
2.2	The Hadoop Framework	9
2.2.1	Hadoop Architecture	11
2.3	Hadoop Distributed File System	13
2.3.1	HDFS Architecture	14
2.3.2	HDFS High Availability	15
2.3.3	Persistence of HDFS metadata	16
2.3.4	Staging	18
2.4	The MapReduce Paradigm	19
2.4.1	Operating Principles	19
2.4.2	MapReduce Limits	22
2.5	Apache YARN	22
2.6	Apache Kafka	24
2.7	Apache Flume	27
2.8	Apache Hive	29
2.9	Cloudera Impala	31
3	STATE OF ART	35
3.1	Logical Architecture	35
3.2	Ingestion Layer	36
3.3	Storage Layer and Analytic Layer	39
4	DESIGN AND DEVELOPMENT	41
4.1	Environment and Versions	41

4.1.1	Versions	42
4.2	Logical Architecture	43
4.3	Ingestion Layer	44
4.3.1	Introduced Tools	44
4.3.2	Implementation	48
4.4	Storage Layer and Analytic Layer	67
4.4.1	Introduced Tool	67
4.4.2	Deployment	70
4.4.3	Performance Test	71
5	CONCLUSIONS AND FUTURE DEVELOPMENTS	75
5.0.1	Future Developments	76
	List of Figures	77
	List of Tables	79
	List of Codes	82
	Bibliography	83

Chapter 1

INTRODUCTION

The work described in this thesis is born from the need to bring innovation to the actual solution, taking in account both the possibility to replace components actually used with more recent ones or that better fit the needs, and the possibility to design them *from scratch*.

Every day new platforms get on the technological stage, offering alternative approaches for solving a problem. If this sentence is typically true, it is truer when we speak about technologies for Big Data: in the last two years data produced in the world have grown by 90%, companies are going to produce *zettabyte* of data in a really short time, from this the need of solutions able to handle huge moles of data and guarantee good performance to sudden load variations.

It is easy to understand that in the Big Data world is really important to move with the times, be updated on upcoming technologies (*technological scouting*) has become one of the phases of solution development, that can lead to the implementation of a cutting-edge product able to satisfy final users' needs.

During the implementation of the project, many softwares available on the Big Data market have been studied in order to evaluate their suitability to improve real-time data ingestion and propose an alternative storage solution to the one actually used.

One of the most known solution for the management and analysis of Big Data is the *Apache Hadoop* framework. Today, The number of companies that develop softwares integrated with this framework is constantly increasing, offering data architects an ever-increasing number of products to choose from. Those products are usually released under commercial distributions with a subscription fee for technical support.

The whole work described in this thesis has been developed on a *Cloudera* distribution, an open source platform, based on Apache Hadoop, built specifically for an enterprise environment. Integrating Hadoop with many other open source projects, Cloudera has created

an advanced system which allows to execute Big Data end-to-end workflows.

This document has been written with the goal to propose a *general purpose* solution to a classic reengineering problem. The whole work has been done on a pre-production cluster where developed solutions are tested before being released to the production environment. In this way we were able to simulate real workloads and evaluate how the developed solution reacts.

1.1 Personal Effort

The proposed solution has been developed dividing the work in the following phases:

- Study of the real-time ingestion layer in place at the beginning of the project, design and development of the new layer based on Kafka Stream and StreamSets technologies.
- Implementation of an analytic layer, on HDFS, CRUD compliant, based on Apache Kudu technology.
- Performance study and comparison of the query layer (Hive vs Impala vs Kudu).

The personal effort covers all design and development phases of the new platform, except of the configuration of Hadoop environment.

1.2 Thesis Structure

The structure of the thesis is the following. In chapter 2 is provided an introduction to the Big Data world and a description of the main tools available in the sector, with a particular attention to the Apache Hadoop framework. In chapter 3 is shown the state of art. In particular will be presented the logical architecture at the beginning of the thesis work. In chapter 4 is described the project design and development, showing the introduced tools, development choices and difficulties encountered. In chapter 5 are described the work results e some possible future development.

Chapter 2

THE HADOOP ECOSYSTEM

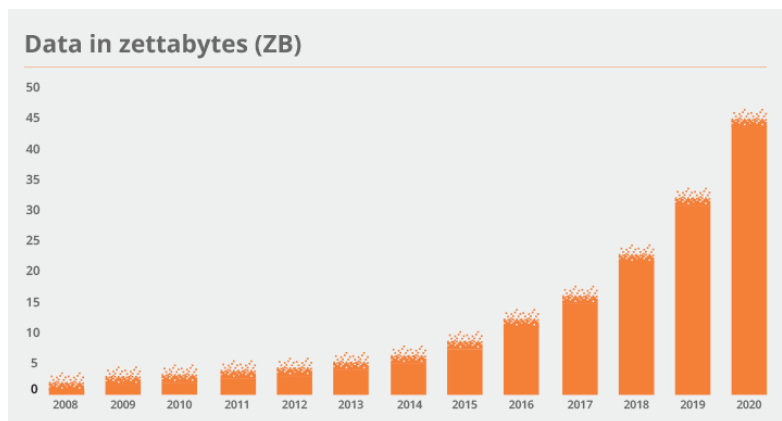
In this chapter are explained the basic concepts about Big Data, aiming to show their main features and scope.

After that, will be introduced the main aspects on which the tools used during the whole development process are based, with a particular focus on the Hadoop framework.

2.1 Introduction to Big Data

Every day the complexity and volume of available data increases. It has been estimated that in less than three year the amount of information will exceed 40 Zettabytes (1 ZB = 2^{21} Byte) [1], thanks to the increasing number of available sources.

Figure 2.1: Growth estimation of data volume produced every year between 2010 and 2020



Even if technology has made giant steps in terms of storage and processing capacity, traditional data systems can not support the information flows coming from modern sources (social network, sensor network, online transaction). One of the worst risks is that this

systems can not satisfy the needs of an ever-connected world, where everything turns around the availability and access speed of information.

Actually there is no formal definition of Big Data, but in order to fully understand the phenomenon, it is usually described through a model called *Big Data 3 Vs* as shown in figure 2.2:

Volume: it refers to the amount of data generated every second. Just think to the number of emails, Twitter messages, photo, videos, data coming from sensors, etc. produced and shared every second. We are not talking about Terabytes , but about Zettabytes or Brontobytes. If we take all the data generated in the world between the beginning of time and 2008, the same amount will be produced soon every minute. This makes data sets too big to be stored and analyzed using traditional databases. With the Big Data technology, we can store and use these data sets with the help of distributed systems, where parts of the same data is stored in different places and then put together by software.

Velocity: it refers to the speed at which new data is generated and to the speed this data is moved. Just think how a message posted on a social network becomes viral in few seconds, or how financial transactions are checked to avoid fraudulent activities. Big Data allows us to analyze data as soon as they are produced, without being put in a database.

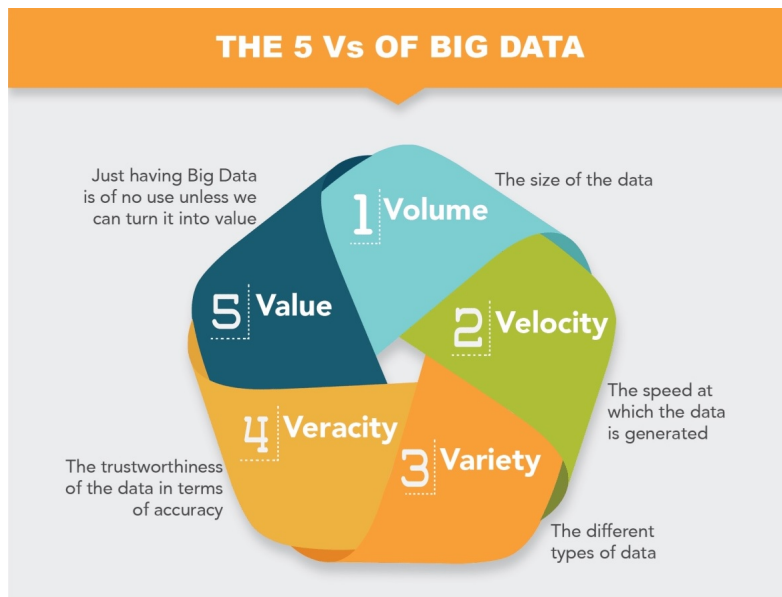
Variety: it refers to the different data types we can access today. In the past everything was all about structured data that could be mapped on tables or inserted in relational databases. Today more than the 80% of the available data are unstructured by nature, and for this reason is difficult to map them on a table. With Big Data we can handle different data types (structured and unstructured) and put them together with the more traditional structured data.

In the last years two more Vs have been added:

Veracity: it refers to reliability and quality of data. Since there are many shapes of Big Data, taking under control the quality and accuracy of data becomes a difficult task (just think to Twitter messages with hashtags, languages slang or to the reliability and accuracy of contents). Big Data and data analytics allows us to work with this kind of data. High volumes fill the gap with low quality and accuracy of the available data.

Value: maybe one of the most important Vs. Having access to Big Data technology is a good starting point but, if we are not able to get value from data, everything is useless.

Figure 2.2: Big Data Vs



This leads us to the most widely used definition in the industry. Gartner (2012) defines Big Data technology in this way.

Big Data is high-volume, high-velocity and/or high-variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making and process automation.

It should be clear now that "big" in Big Data is not just about volume. Big data certainly involves having a huge amount of data, but is not just that. Yes, you are getting a lot of data, but it is also coming at you fast, in a complex format and from a great variety of sources.

This is a rapidly expanding sector whose potential is not fully explored yet, which involves many professional figures with skills that are often very heterogeneous.

2.2 The Hadoop Framework

Apache Hadoop is an open source framework that allows the distributed processing and storage of large data sets across clusters of computers using a simple programming model. It is designed to scale up from single server to thousands of machines, each offering local computation and storage [2].

The framework has many features that have made it one of the reference tools for the

Figure 2.3: Hadoop Logo



management and analysis of Big Data in many companies and research centers all over the world [3]:

Scalability and performance: processing is distributed across several hosts and this allows to store, manage and process data in the order of Terabytes or Petabytes. New nodes can be easily added to the cluster and the framework allows easy integration.

Reliability: large data centers are subject to a great number of failures due to malfunctioning of single nodes. The main idea with which all Hadoop components have been developed is that faults and malfunctions are not exceptions but common events, so they must be automatically managed within the framework. When a node falls, processing is redirected to the other nodes in the cluster and the data are automatically replicated in case of future failures. This approach also allows the use of *commodity hardware*, reducing the total cost.

Flexibility: unlike traditional DBMS, typically is not necessary to define the data structure before storing them. These can be stored in any format, structured or unstructured, and then be managed appropriately during the reading phase.

Low cost: Hadoop is an open source framework released under Apache license [4]. However, there are several commercial implementations with paid supports, such as Cloudera, Hortonworks and MapR.

The main components of Hadoop are a processing infrastructure based on the *MapReduce* paradigm and a distributed file system, called *Hadoop Distributed File System* (HDFS). Its main components are:

Hadoop Common: a common software layer that support the other Hadoop modules.

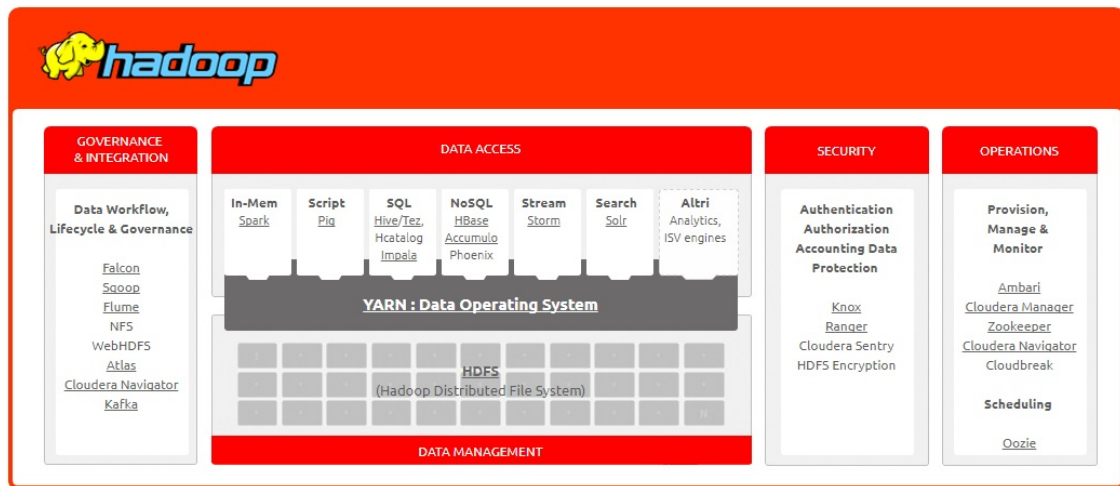
Hadoop Distributed File System: a file system distributed across the nodes inside the cluster. It provides high throughput and data redundancy to make data always available even in case of a node failure.

Hadoop YARN: a framework for job scheduling and cluster resource management.

Hadoop MapReduce: a programming pattern, based on YARN, for parallel processing of large data sets.

The great success of Hadoop has led to the creation of an entire ecosystem (figure 2.4) based on the four component just mentioned, but enriched by many modules designed to address specific needs. Among these we can mention *Apache Hive*, *Apache Flume*, *Apache ZooKeeper* and many others.

Figure 2.4: Hadoop Ecosystem



In the next paragraphs will be described the main components used during the thesis work.

2.2.1 Hadoop Architecture

In the Hadoop terminology, an host is a computer. Usually we refer to a host calling it *node*.

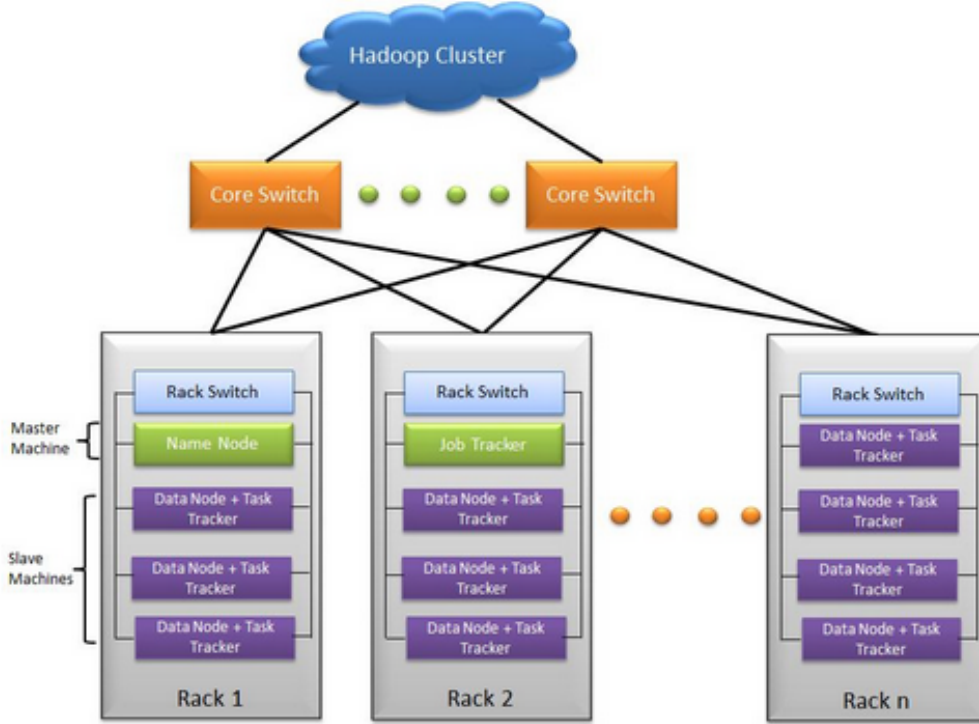
A *cluster* is a set of two or more nodes connected each other by an high-speed local network. Depending on the needs, a cluster can consist of few nodes or thousands. Within large data centers, cluster nodes are organized into multiple *racks* connected by a very high-speed switch. Each rack contains a variable number of servers, which typically varies between 16 and 24, connected to each other by a switch.

Figure 2.5 shows a typical configuration of a cluster.

A Hadoop cluster consists of two different categories of nodes: *master node* and *worker node*.

The master nodes have the task of supervising and coordinating the main operations.

Figure 2.5: Typical configuration of a Hadoop Cluster



Among these we can mention the NameNode, which coordinates the use and access to the distributed file system, or the JobTracker which coordinates the distributed execution of MapReduce applications.

The worker nodes, which make up the majority of nodes, are responsible for executing operations and launching processes, under the directives of master nodes.

One of the main problems to be faced in the management of distributed systems is scalability. In fact, at any time, we may need more resources to support an increasing workload. Two types of approach can be adopted:

Vertical Scalability (scale up): it consists of adding more resources to the individual nodes, for example more memory, more CPUs or more disks. This approach is typically adopted in case of use of few high-performance nodes.

Horizontal Scalability (scale out): it consists of adding more nodes to the cluster. Typically it is adopted in case the cluster consists of a large number of commodity servers.

The term *commodity hardware* refers to a low cost and widespread device, replaceable

by another device of the same type in case of need. The term can be put in contrast to *dedicated hardware*, where the cost is typically higher and whose availability is limited due to the lack of diffusion and compatibility with other devices other than those for which it was designed.

The approach used by Hadoop is scale out, making it particularly easy to add servers to the cluster. Furthermore, the framework is designed to hide from the user the complexity introduced by the distributed programming models in terms of synchronization and scheduling of the task, allowing them to concentrate on solving the problem rather than managing the environment.

2.3 Hadoop Distributed File System

Hadoop Distributed File System is the file system on which the Hadoop architecture is based. It is a distributed file system, written in Java, which ensures reliability and scalability.

HDFS was designed having in mind that hardware failures are common events and not exceptions. A typical HDFS instance can in fact be made up of hundreds or thousands of nodes, so the probability that in a certain instant one or more nodes could fail is very high and therefore not negligible.

HDFS is inspired by the Google File System, a proprietary file system described for the first time in a paper released by Google in 2003 [5]. It is not a *general purpose* file system, but it is meant for exclusive use within the Hadoop framework, and it is not fully compliant with POSIX specifications. The designers have decided to relax some specifications in order to optimize the performances of the applications for which the file system was thought [6].

HDFS is designed to support very large files. HDFS-compatible applications typically work with file sizes in the order of GBs or TBs. Internally, a file is divided into smaller blocks, sometimes called *chunk*.

All blocks in a file have the same size except for the last one. The block size is configurable for each individual file, and the typical value of this size is 64-128 MB.

Each data block is replicated on different nodes to guarantee reliability and fault tolerance in the event that one of the nodes becomes unreachable or part of the data is corrupted. The number of replicas is called *replication factor* and is also configurable for each file.

Figure 2.6: Configuration with replication factor equal three

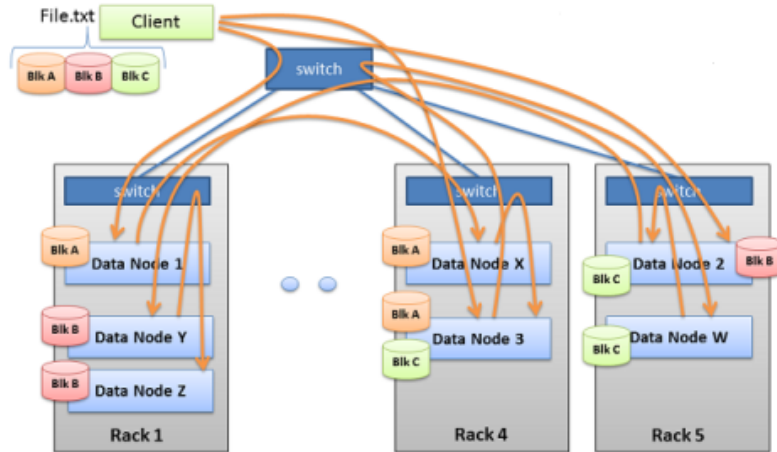


Figure 2.6 shows the possible configuration of a cluster with a replication factor of three. Note the arrangement of replicas in the cluster. After a block has been written to a node, a replica is placed on another node within the same rack. This ensures that data is not lost in the event of a machine failure. In this case, however, if the fault should affect the whole rack, the data is still lost. To avoid this situation, the third replica of the block is placed on a machine inside a different rack than the initial one.

The knowledge of the network topology within the cluster is called *rack awareness* and allows to maximize network performances and optimize data reading operations.

In order to minimize bandwidth consumption and read-only latency, the framework ensures that the reading of data always occurs on the node that is topologically closer to the client that requested it. In the best case, the data is present on the same machine that requested it, so that replica is chosen. In case this is not possible, it is checked that there is a replica on a node inside the same rack of the client, and if so, that copy is selected. If no replicas are available in the requester's rack, data is read from the closest rack.

2.3.1 HDFS Architecture

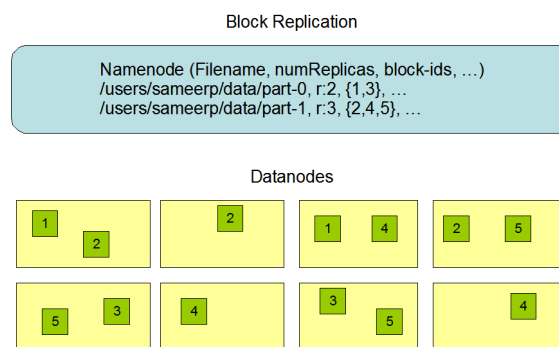
HDFS is based on a master-slave architecture. More precisely, in each cluster there is a single master, called *NameNode*, and a variable number of slaves, called *DataNode*.

NameNode: its role is to regulate client access to files. It also manages the file system

namespace and the operations of opening, closing and renaming of files and folders. Finally, it determines the mapping of file blocks and their respective replicas in the various DataNodes. The NameNode is designed to contain and manage all file system metadata, without ever directly manipulating application data.

DataNode: they manage the data present in the node in which they reside. They are responsible for serving client block read and write requests and, coordinated by the NameNode, perform data creation, deletion and replication operations.

Figure 2.7: NameNode and DataNode in HDFS



HDFS is written entirely in Java and although there are no limits to the number of nodes that can be active on a single machine, typically a single HDFS node is instantiated on each host.

2.3.2 HDFS High Availability

The NameNode is a critical component of HDFS. In fact, without it, clients could not have access to the data stored in the DataNodes.

In the first version of Hadoop, each cluster had a single NameNode. This simplified the overall architecture but made the NameNode a *single point of failure*. Whenever the host node or the NameNode process became unavailable, the entire cluster became unusable as long as the NameNode was not restarted.

Starting with version 2.0, a feature called *HDFS High Availability* has been introduced. This allows to create two redundant NameNode within the cluster. In this configuration, one of the two NameNodes is active, while the other is in a state of *hot standby* which allows fast recovery in case of failure or maintenance of the main node.

In order to let the mechanism work properly, the active NameNode and the standby NameNode must always be synchronized in terms of knowledge of the namespace and the arrangement of the blocks inside the cluster. Every change to the namespace is recorded

in an edit log placed in a shared folder. The standby NameNode periodically reads this log and applies the changes to its namespace, so that it stays in sync with the main NameNode. This allows a quick and correct awakening of the standby NameNode in case of need. Information about the location of blocks and replicas within the cluster is provided directly DataNodes, configured so that such status information is automatically sent to both NameNodes.

The NameNode receives information from the various DataNodes through messages called *Heartbeat* and *Blockreport*.

The Heartbeat is sent periodically from each DataNode. The transmission interval between two Heartbeat messages is configurable and is set to a default of three seconds. Through this short message, the node tells the NameNode to be active and properly working. If the NameNode no longer receives a Heartbeat from a certain node, it then considers it no longer reachable and does not assign work to that node. Since a DataNode is no longer reachable, the NameNode assumes that every block stored in that node is no longer available. If this causes the number of replicas of a block to fall below the value set as replication factor, the NameNode starts the replication process on another node.

Each DataNode saves HDFS files on its local file system, storing a separate file for each block. Moreover, it keeps track of the relevant metadata, like the checksum. At startup, the DataNode scans its local file system and generates the list of HDFS blocks that correspond to each file. Starting from this list, it generates a report called Blockreport. A new Blockreport is generated and sent to the NameNode at predefined intervals, so that NameNode is updated on the status of the replicas on the various nodes.

2.3.3 Persistence of HDFS metadata

The whole HDFS namespace is stored in the NameNode, which handles it through the use of two files, *FsImage* and *EditLog*.

FsImage is a file representing the file system metadata snapshot. Even if it is very efficient to read, the FsImage is not suitable for small updates, such as renaming a single file. In order to avoid writing a new FsImage every time the namespace is changed, the NameNode records the editing operations in the EditLog.

Both are stored in the local file system of the NameNode.

The NameNode keeps in memory the whole image of the namespace and the mapping of the blocks, thanks to the fact that these structures are designed to be compact and to be able to contain information related to a large number of folders and files in a small space. When the NameNode is started, it performs an operation called *checkpoint*, whose purpose is to apply the changes contained in the EditLog to the image contained in the FsImage.

The checkpoint is performed only when the node is started and consists of the following sequence of operations:

1. The NameNode reads EditLog and FsImage from the local file system.
2. It applies the transactions contained in the EditLog to the FsImage cached in memory.
3. It overwrites the version of FsImage file on disk with the updated image just obtained.
4. It truncates the EditLog file because the changes contained in it have been applied to the file system image in memory and on disk.

The EditLog and FsImage files are really important for the functioning of HDFS and it is necessary to make sure that their corruption does not make the system unusable. For this reason the NameNode can be configured in such a way as to create multiple copies and keep them updated over time. This synchronization can cause performance degradation in terms of transactions per second that the NameNode can support.

In the current implementation, the merging between the FsImage and the EditLog is done only when the NameNode is started. For this reason, the EditLog size can become very large over time. In addition, a larger number of edits to be applied makes the node start slower. To allow a quick start, a node called *Secondary NameNode* was introduced. From the name it might seem a sort of backup of the main NameNode, in reality its role is to periodically merge the two files, so that the EditLog keeps its size small.

Figure 2.8: Behaviour of the Secondary NameNode

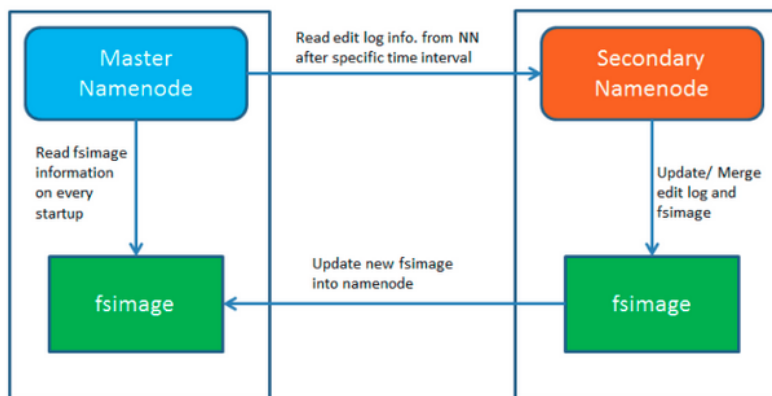


Figure 2.8 shows the behaviour of the Secondary NameNode:

1. The EditLog and the FsImage are retrieved from the Master NameNode at regular time intervals.

2. A new FsImage is created by applying the edits, and then this version is sent to the Master NameNode.
3. The Master NameNode is going to use this new version of FsImage at the next start, reducing restart times.

2.3.4 Staging

When a client requests the creation of a file, the request is not immediately forwarded to the NameNode. The data is initially placed in a temporary location on the client's file system. When enough data is reached to fill a block or the temporary file is closed, the client contacts the NameNode. Only at that point data is written on HDFS, through the following sequence of operations:

1. The NameNode puts a new entry in the file system hierarchy and allocates a block in one of the DataNodes.
2. The NameNode replies to the client's request by specifying the identity of the DataNode is going to receive the data.
3. The client transfers the data from the local file to the DataNode and tells the NameNode to close the file.
4. The NameNode stores the operation persistently by inserting its entry in the EditLog. If the NameNode falls before the EditLog edit is written, the file reference is lost.

In case of a replication factor greater than one, the staging phase is followed by another phase, called *replication pipelining*. In this scenario, when the client requests to create a file, the NameNode sends to the client a list of DataNodes that are going to receive the block. At that point the client, in addition to transmitting data to the first DataNode, it sends also the list of the other DataNodes received by the NameNode. The first DataNode, as it receives portions of files, saves them in its local file system and at the same time sends them according to the DataNode of the list, and so on.

The presence of the staging phase is one of the relaxation introduced to the POSIX specifications during the design of the file system, to better support the features of HDFS applications and avoid a high network congestion due to the direct writing of files by the clients.

2.4 The MapReduce Paradigm

The Hadoop architecture is based on the MapReduce programming paradigm. This model was created with the aim of supporting the creation of distributed applications capable of processing huge amounts of data, in a reliable and fault-tolerant way.

The paradigm was created by Google in early 2000 and described in a paper in 2004 [7]. This work inspired the open source implementation of Apache Hadoop.

The use of MapReduce allows the execution of a large number of applications. Among these we can mention:

- Log Analysis.
- Textual Analysis, indexing and research.
- Analysis of complex data structures such as graphs (for example social network analysis applications).
- Data Mining and Machine Learning.
- Execution of distributed tasks such as complex mathematical calculations and numerical analyzes.

Google itself has used in the past its implementation of MapReduce for the regeneration of search indexes of Web Pages.

2.4.1 Operating Principles

The model requires that the data can be mapped to a *key-value* pair. For example, think about a Web Page: you could consider the URL as the Key and the HTML content of the page as the respective value.

A MapReduce program consists of two basic steps, from which the framework takes its name:

Map: The master node takes the input data, divides it into small sub-problems, and distributes the job to the slave nodes. The single mapper node applies transformations and produces the intermediate result of the `map()` function in the form of a pair (key, value) stored on a distributed file, whose location is notified to the master at the end of the map phase.

Reduce: The master node collects the answers, combines the pairs (key, value) into lists of values that share the same key and sorts them by key (lexicographic order, increasing or defined by the user), *Shuffle Step*. The pairs of the form (key, List(value, value, ...)) are passed to the reducer nodes that perform the `reduce()` function.

More formally, the two phases can be defined as:

Map(k_1, v_1) \rightarrow list(k_2, v_2)

Reduce(k_2 , list (v_2)) \rightarrow list(v_3)

Intermediate operations, such as the extraction of (key, value) pairs from the input data set and the aggregation by key of the values exiting the mapping phase are managed by the framework, so the programmer does not have to deal with it.

When a job is submitted, the application is copied to the different cluster nodes in charge of processing. We distinguish three main actors:

Driver: takes care of configuring the job and running it on the Hadoop cluster. There is only one instance of the driver, and it is performed on the client.

Mapper: implements the mapping phase. This phase is completely parallelizable, therefore multiple instances are executed. For optimal parallelism, the number of mappers should be equal to the number of input blocks. The instances are launched on the various nodes of the cluster and, depending on the capacity of the individual machines, multiple instances can be active on each node. Following the data locality principles of Big Data, the framework makes sure that the instances are executed on the nodes that contain the data to be processed, in order to minimize the network usage.

Reducer: implements the reduce phase. Its instances are executed on the cluster nodes, but their number must be configured by the user and varies depending on the application. Each reducer produces an output file and stores it on HDFS.

The simplest and most famous example of a MapReduce application is the Word Count Problem, where you must count the number of occurrences of words within a text. In figure 2.9 is shown how it works.

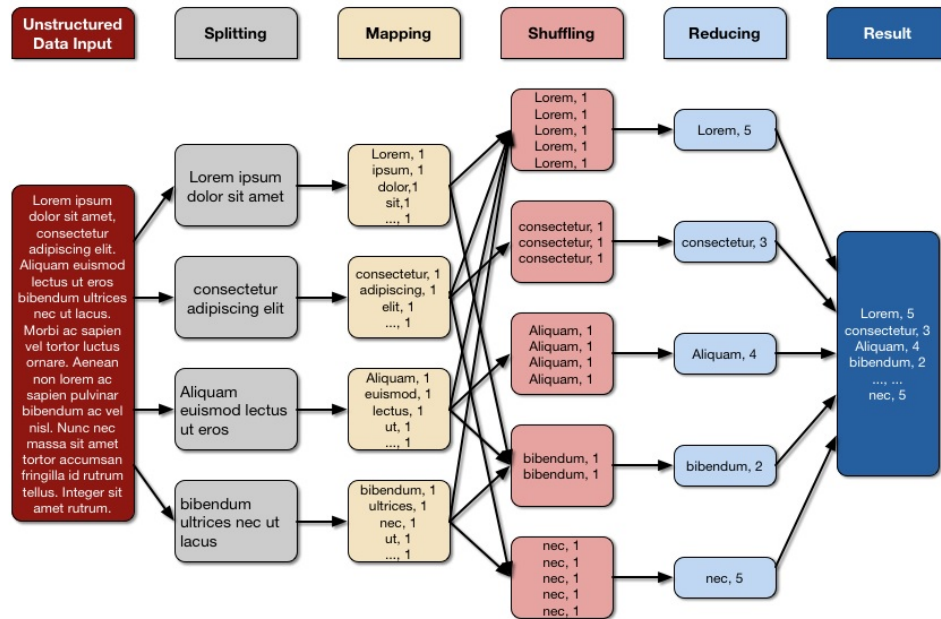
The text file is initially splitted in multiple blocks on HDFS, in this case four. Inside each block are stored a certain number of records, each containing a sequence of words. For each block is created an instance of the Mapper. This receives one line of text at a time, in the format (*NULL*, *line of text*). In this particular example, the key of the elements processed by the Mapper is irrelevant.

The Mapper processes the received line of text by separating the words, and issuing for each of them a (*word*, *1*) pair, which indicates that the word appeared once.

The issued pairs are temporarily stored on the local file system of the node where was executed the Mapper.

At this point the framework makes sure that the pairs are sent to the nodes that will carry

Figure 2.9: Word Count in MapReduce



out the reduce phase. All pairs with the same key must be grouped together and sent to the same Reducer. To do this, an operation called *shuffle* is performed, in which a hash function is typically applied to the key, and the pair is routed to a Reducer based on the result of the hash.

Assume that five instances of Reducer were created. The pairs addressed to each of them are grouped by key. At this point the Reducer receives, one at a time, the pairs (word, list(1, 1, 1, ...)).

For each pair, all the unit values in the list are sum up, obtaining the number of occurrences of the word inside the whole text. The *pair* (word, occurrences) is issued and the final result is written on HDFS.

The application output consists of file files, one for each Reducer.

The input files of a MapReduce application and its outputs are typically stored on HDFS. In contrast, the intermediate results of the processing, such as the outputs of the mapping phase, are not placed on the distributed file system but on the local file system of each node that have done the processing.

The pairs emitted by the mappers are transferred to the Reducer host nodes through the network, so the performance of the framework can be influenced by the amount of data exchanged and by the capacity of the network.

In the previous example, it can be seen that the (word, 1) pair is issued by a Mapper as

many times as the occurrences of the word inside the block. This can result in degradation of network performance. For this reason, an optimization phase has been introduced called *combiner*, which performs a similar processing like the Reducer, but acting on local data, before being sent to the Reducer.

2.4.2 MapReduce Limits

MapReduce was one of the first programming paradigms used for Big Data analysis. Even if it is still used for the development of a large number of applications, there are cases in which its use is discouraged. Its limits are mainly due to the lack of flexibility and management of intermediate results:

- Low flexibility: the model is relatively simple, which is why it is often not suitable for more complex problems.
- Designed for batch applications, not streaming or interactive.
- Not suitable for iterative problems, because this involves having to read data from disk at each iteration.

The need to overcome these limits has led to a series of alternative paradigms to MapReduce in recent years. The introduction of Apache YARN into Hadoop has allowed the integration of these new paradigms with the framework.

In recent years, the complexity of problem solving has been masked inside products based on MapReduce, which over time have become more and more widespread: *Apache Hive* and *Apache Pig*, for example, automatically translate sql-like queries in a sequence of MapReduce jobs.

2.5 Apache YARN

YARN (Yet Another Resource Negotiator) is one of the four main components of Hadoop 2.0, Its main job is managing the resources of the whole ecosystem, scheduling and launching processes on the cluster.

The idea on which YARN is based is to divide the functions of resource management, scheduling and monitoring of jobs in multiple processes distributed on the various nodes. In particular, a global *ResourceManager* and a *ApplicationMaster* are instantiated for each submitted job.

YARN provides its services through two demons:

ResourceManager: it is the master. It takes care of receiving requests from the client, managing resources of the cluster, keeping track of them, and assigning tasks to individual workers. The term ResourceManager is also used to indicate the machine hosting the process. There is only one ResourceManager inside the cluster.

NodeManager: it is the worker. It receives the ResourceManager guidelines, launches the required processes, and tracks local resources of the host node, including CPU and memory usage, storage space, and network capacity. Also in this case the term NodeManager is used to indicate both the process and the host node. There is a NodeManager for each node in the cluster.

Figure 2.10: YARN Architecture

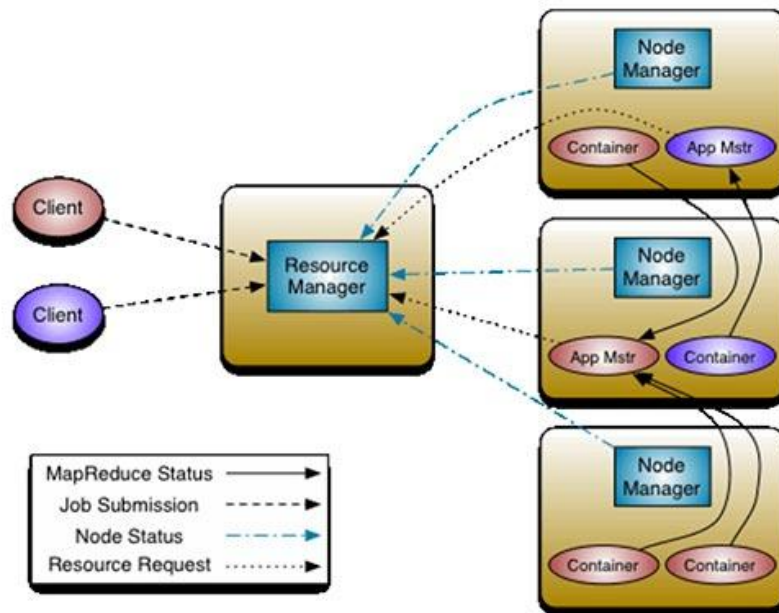


Figure 2.10 shows the architectural scheme of YARN.

Each application is divided into several processes, which are called *task*. One of the tasks defined by the framework is called *ApplicationMaster*. Its role is to negotiate the use of resources with the ResourceManager, launch and monitor the various tasks on the NodeManager.

The ResourceManager consists of two main components, called *Resource Scheduler* and *ApplicationsManager*:

Resource Scheduler: it takes care of allocating the needed resources for applications. It performs its task based on needs of individual applications and the availability

of resources on each node. It does not offer features such as monitoring application status or restarting jobs in case of failure.

ApplicationManager: it takes care of receiving applications submission requests and allocating resources for their ApplicationMaster. It also allows to restart the ApplicationMaster in case of task failure.

Each NodeManager keeps track of its available resources and communicates them to the ResourceManager, which tracks the overall resources available in the cluster and where they are located. In Hadoop 2.0, each application, through the ApplicationMaster, can request the assignment of two types of resources: number of vcores (possibility of using the CPUs) and amount of memory. In the future, support will also be extended to requests regarding the number of GPUs, the amount of bandwidth, and the use of disk.

The Resource Scheduler replies to the ApplicationMaster by assigning the requested resources, grouped into a single logical package called *Container*. Holding a container guarantees an application the ability to use a certain amount of resources in a node of the cluster.

The execution of an application takes place through a series of steps:

1. The application starts and contacts the ResourceManager.
2. The ResourceManager assigns a Container to the application.
3. Once the resources are available through the first Container, the ApplicationMaster is executed.
4. The ApplicationMaster negotiates with the ResourceManager the allocation of new Containers required by the application, which is divided into several processes called tasks. At that point the processing begins.
5. As soon as the tasks end, the Containers are deallocated. When all the tasks have been completed, the ApplicationMaster exits and even the last Container is released.
6. The application exits.

2.6 Apache Kafka

Apache Kafka is an open source distributed streaming platform [8], based on the *publisher/subscriber* pattern. The project was born with the idea of proposing a platform able to manage real-time flows, guaranteeing persistence, high throughput and low latency.

From an architectural point of view, Kafka consists of the following components:

Figure 2.11: Kafka Logo



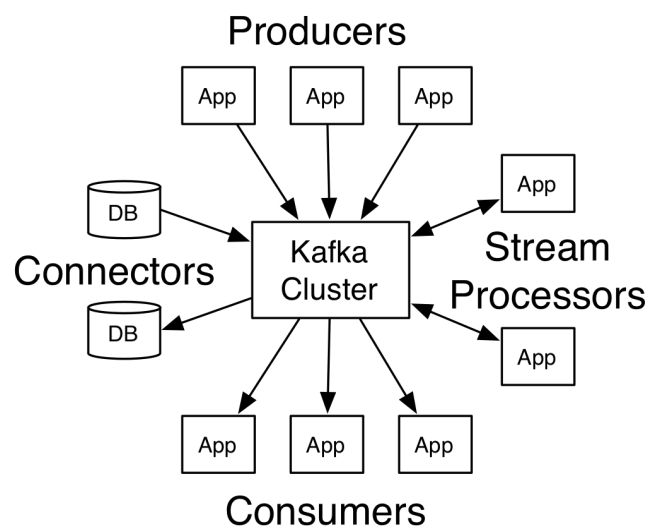
Topic: represents a stream of messages of a particular type. A message is defined as a byte sequence and a topic is the category to which the message belongs.

Producer: is the actor who publishes messages in a topic.

Consumer: can subscribe to one or more topics and consume published messages.

Brokers or Kafka Cluster: are a set of servers where the topic physically resides together with the data published on it.

Figure 2.12: Kafka Architecture



Kafka provides four APIs:

Producer API: which allows an application to publish messages on one or more topics.

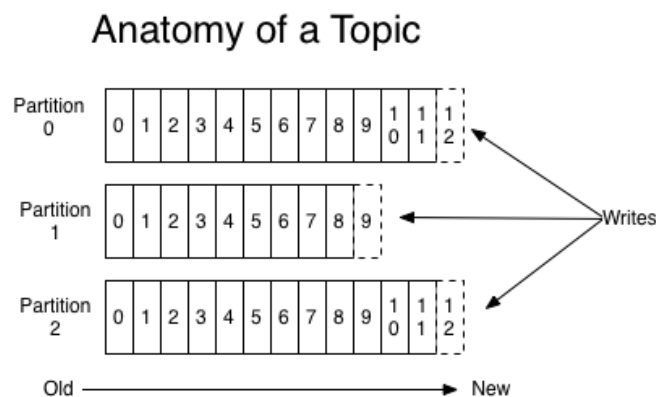
Consumer API: which allows an application to subscribe to one or more topics and process the messages contained in.

Streams API: which allows an application to act as a stream processor, whose task is to consume messages from one or more topics, apply a transformation to the consumed messages and republish them.

Connector API: allows you to develop and run reusable producer or consumer that can link Kafka topics to existing applications or storage systems.

As you can guess, Kafka main component is the topic. For each topic, the Kafka cluster maintains a partitioned log that looks like:

Figure 2.13: Topic Anatomy

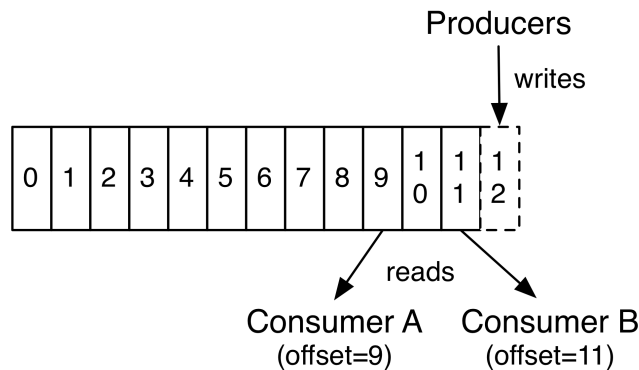


Each partition is an ordered, immutable sequence of records that is continually appended to a structured commit log. The records in the partitions are each assigned a sequential id number called *offset* that uniquely identifies each record within the partition. The Kafka cluster retains all published records, whether or not they have been consumed, using a configurable retention period. Kafka's performance is effectively constant with respect to data size so storing data for a long time is not a problem [8].

In fact, the only metadata retained on a per-consumer basis is the offset or position of that consumer in the log. This offset is controlled by the consumer: normally a consumer will advance its offset linearly as it reads records, but, in fact, since the position is controlled by the consumer it can consume records in any order it likes. For example a consumer can reset to an older offset to reprocess data from the past or skip ahead to the most recent record and start consuming from "now". This combination of features means that Kafka consumers are very cheap, they can come and go without much impact on the cluster or on other consumers.

The partitions of the log are distributed over the servers in the Kafka cluster with each

Figure 2.14: Topic Consumer



server handling data and requests for a share of the partitions. Each partition is replicated across a configurable number of server for fault tolerance. Each partition has one server which acts as the leader and zero or more servers which act as followers. The leader handles all read and write requests for the partition while the followers passively replicate the leader. If the leader fails, one of the followers will automatically become the new leader. Each server acts as a leader for some of its partitions and a follower for others so load is well balanced within the cluster.

Kafka can be used in many situations, but it gives its best when dealing with streaming of data, because it guarantees:

Scalability: it is a distributed system, so easy to scale in case of need without having to stop the flows.

Durability: messages are stored on disks for long periods, moreover they are replicated across the servers inside the cluster.

Reliability: data is replicated, supports multiple subscriptions and automatically balances consumers in case of failure.

Performance: it ensures high throughput for both producers and consumers, with disk-based structures ensuring consistent performance even in case of huge volumes.

2.7 Apache Flume

Apache Flume is a tool/service/data ingestion mechanism for collecting, aggregating and transporting large amounts of streaming data, such as log files or events, from various sources to a centralized data store. Flume is a high reliable, distributed, and configurable

Figure 2.15: Flume Logo



tool. It is principally designed to copy streaming to HDFS.

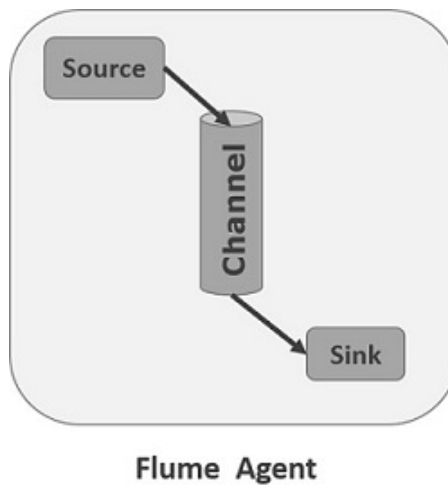
Flume is based on a Java process called *agent*. The agent has the task of running three components:

Source: receives data from the source and transfers them to one or more channels in the form of Flume event.

Channel: is a temporary store that receives events from the source and stores them until they are consumed by sinks. It acts as a bridge between the sources and sinks.

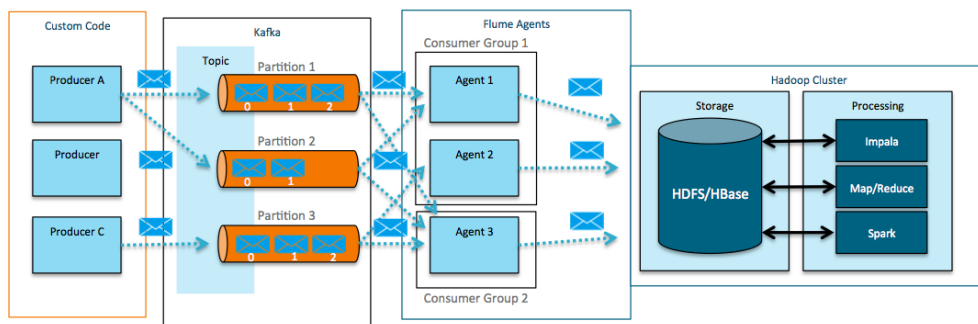
Sink: writes data on HDFS or HBase. It consumes data from the channel and sends it to the destination. The destination of a sink can be also another Flume agent.

Figure 2.16: Flume Agent



Flume has been a great success and its use in Big Data solutions is constantly increasing. This is due to its simplicity in terms of usage and integration. In recent years, Cloudera engineers, along with other members of the open source community, have made possible the integration between Apache Kafka and Flume, informally called *Flafka*. The Flume-Kafka integration offers new features that Kafka, alone, does not offer. In a few simple steps you can create Kafka producers and consumers without having to write code, adding the possibility to process or transform messages coming from Kafka on-the-fly.

Figure 2.17: Flafka Architecture Example



2.8 Apache Hive

Figure 2.18: Hive Logo



Apache Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.

It all started with the need to manage and learn from the huge amount of data that

Facebook was producing every day through its social network. After testing different systems, the team chose Hadoop to store and process data, for its cost and scalability. Hive was created to allow analysts with strong SQL skills to query the huge volume of data that Facebook had on HDFS. Today, hive is a successful Apache project used by many companies to process their own data.

Despite its ability to manage structured data in a table, Hive is not designed to replace transaction-based systems.

The main features offered by Hive are:

- Data access through a dialect of the SQL language, called *HiveQL*, which allows to perform ETL operations, reporting and data analysis.
- Ability to read and write data in various formats.
- Ability to query through Spark or MapReduce applications.
- Possibility to structure data read by HDFS.

The Hive infrastructure is divided into several components:

User Interface: allows the user to submit queries and receive results.

Metastore: is one of the main components. It takes care of storing all information on partitions and tables, such as the names and types of columns. It also keeps track of how to read and write data, as well as the location of data on HDFS.

Compiler: takes care of receiving queries, verifying their semantic correctness and, after retrieving the necessary information from the Metastore, generating an execution plan. This execution plan is represented by an acyclic graph, made by operations such as MapReduce jobs or read and write operations on HDFS.

Execution Engine: receives the execution plan of the queries and assigns the various operations to the individual Hadoop components responsible for executing them, managing their dependencies.

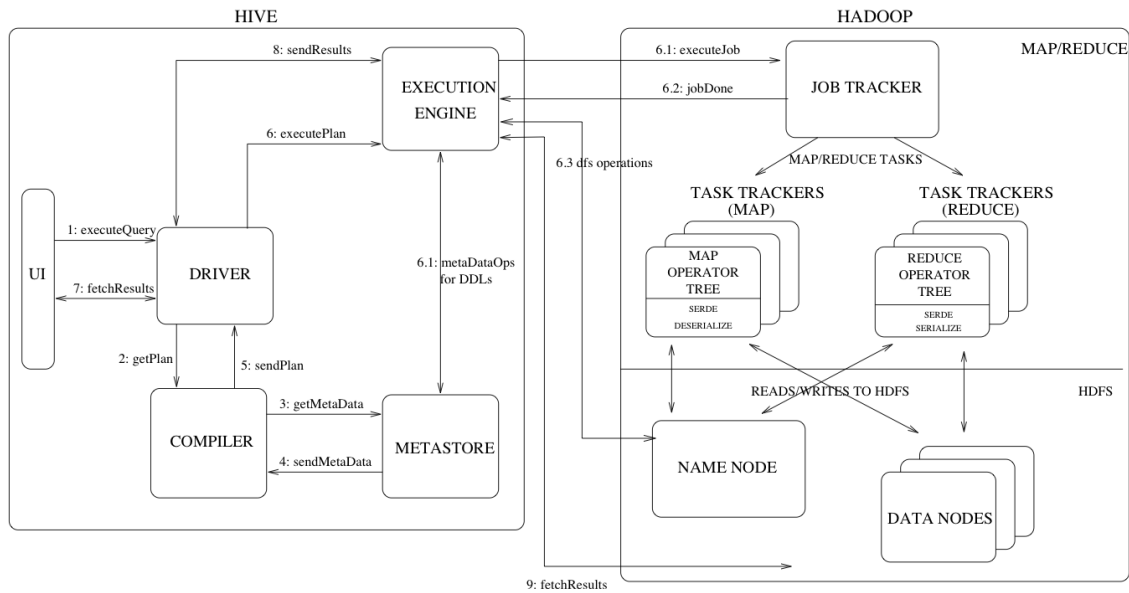
Driver: deals with receiving user queries, creating sessions and managing the communication between Compiler and Execution Engine.

Figure 2.19 shows Hive's architectural scheme.

When a table is created, its schema is not stored in the same Hive blocks where data resides. On the contrary, all the metadata of a table are stored by the Hive Metastore in a relational database (for example MySQL). At each access to the table, its metadata is read from the Metastore.

One of the main consequences of this decoupling is the possibility to create tables from files already stored on HDFS. This fact does not require the modification of the

Figure 2.19: Hive Architecture



blocks on the file system. During the creation of the metadata, the user indicates how these files must be read and modified, and from that moment they will be able to access them through HiveQL queries.

The Hive Metastore represents a *single point of failure*, in fact its unavailability causes the impossibility to access the whole data set. For this reason, the service is often configured in *High Availability* mode within the cluster, with a second node containing a copy of the metastore and placed in standby mode, ready to wake up in case of failure of the main metastore.

2.9 Cloudera Impala

Cloudera Impala is a Massive Parallel Processing (MPP) SQL Engine created to query data in Hadoop. Impala was born with the aim of combining the familiarity of the SQL language with the scalability and flexibility of Apache Hadoop.

Compared to other systems, Impala was developed entirely in C++ and Java. It is perfectly integrated with Hadoop using other components already included in the platform (HDFS, HBase, YARN, Hive) and is able to work with the most common formats.

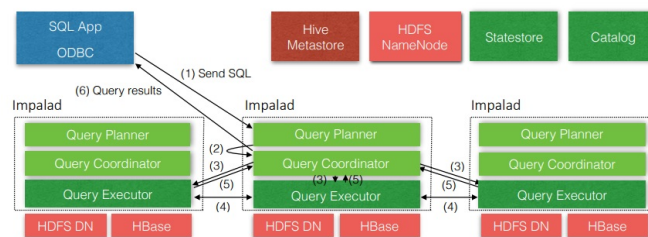
Impala is independent of the underlying storage layer, unlike traditional relational databases where the query and storage engine are integrated into the same system.

Figure 2.20: Impala Logo



The high-level architecture of Impala is shown in figure 2.21.

Figure 2.21: Impala Architecture



Impala is based on three services:

Impala Daemon: (impalad) is responsible for accepting queries from clients and orchestrating their execution on the cluster. When an Impala Daemon takes care of running a query, it is referred as *coordinator* of that query. In any case, all the demons are symmetrical, so they can play all the roles. This is very useful both for balancing the load and for replacing a daemon in case of failure. The Impala Daemon is executed on every machine in the cluster on which the DataNode process is also running, so there is typically an Impala Daemon on each machine. This allows Impala to take advantage of the data locality and to read blocks from the file system without having to use the network.

Statestore Daemon: (statestored) is the service in charge to manage the metadata, and in particular to send them to all the other processes in the cluster during the executions. The Statestore Daemon is also responsible for verifying the status of Impala Daemons dispatched on the DataNodes. In the event that one of the

Impala Daemons could no longer be reached due to some failure, the Statestore Daemon informs all other Impala Daemons such that the offline node is not used for future query execution. Since the goal of the Statestore Daemon is to help Impala Daemons when something goes wrong, its presence within the cluster is not essential. In the event that it is no longer reachable, Impala Daemons would work anyway.

Catalog Daemon: (catalogd) is responsible for sending metadata to Impala Daemons, when the metadata are modified by a query. Since requests come from the Statestore Daemon, generally both the statestore and the catalogd reside on the same machine.

One of the main purposes for which Impala was designed is to make SQL-on-Hadoop operations fast and efficient. In Practice, Impala makes use of the infrastructure set up by Apache Hive, usually present because it is used for the execution of very long or batch operations. In particular, Impala uses the same metastore used by Hive, but unlike the latter, it tries to reduce accesses to the metastore caching metadata on the Impala Daemons distributed in the cluster. SQL queries are translated into MapReduce jobs, like Hive, but everything is done in-memory in such way to ensure reduced response times.

Chapter 3

STATE OF ART

In this chapter will be shown an overview of the architecture in place at the beginning of the thesis work.

For each layer will be explained how it works, trying to highlight strengths and weaknesses occurred during the study of the solution.

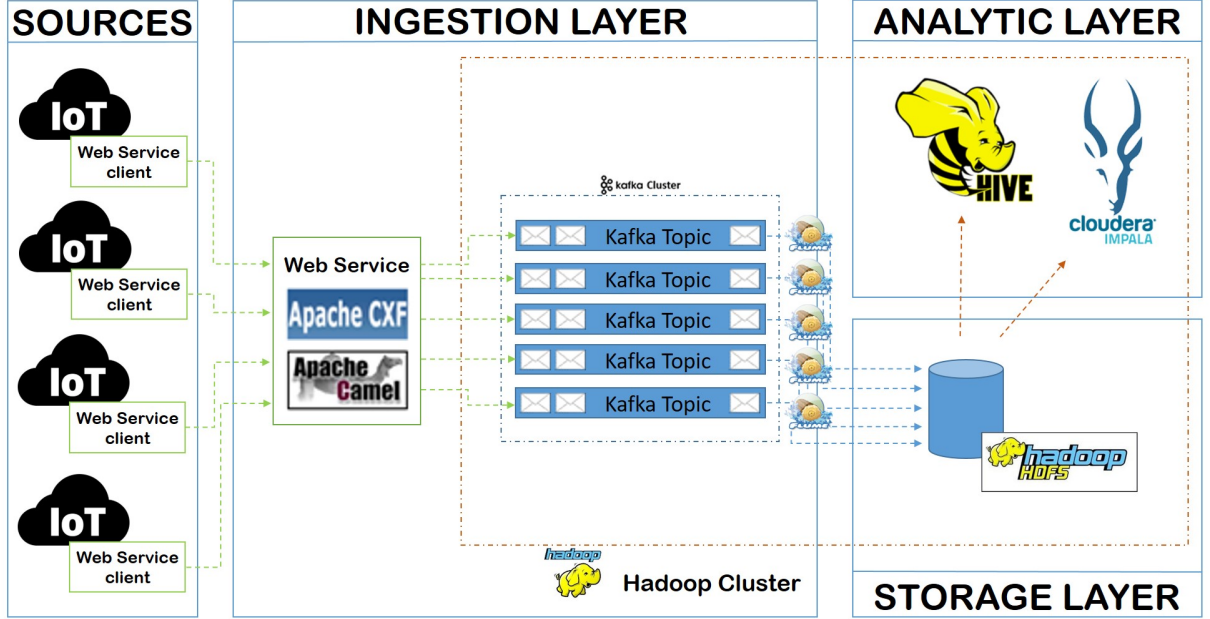
3.1 Logical Architecture

The solution was developed in the late 2015, commissioned by one of the main clients of Data Reply. Initially the project specifications were relatively simple: the sources were limited in number and belonging to the same category, the messages sent by the latter were well specified and subject to few variations. Currently the number of sources is considerably greater than when the system was developed, moreover the types of messages sent by the latter is increased.

Figure 3.1 illustrates the logical architecture developed in 2015.

The image, in addition to showing the logical architecture, also allows us to understand its operating principle: the sources, similar to IoT devices, are equipped with a WS client able to contact the single access point of the layer ingestion. The devices periodically send messages to the WS server, which, after processing the message, dispatches information on different Kafka topics. These topics have been linked to a Flume agent whose job is to consume the data and store them in the destination folders on HDFS. On the destination folders, tables have been created that can be queried through Hive or Impala, depending on the needs.

Figure 3.1: Logical Architecture



In the following paragraphs, the ingestion layer, the storage layer and the analytical layer will be analyzed in detail, in order to provide a clearer image of the various components and highlight those that have been defined as the weak points of the whole architecture.

3.2 Ingestion Layer

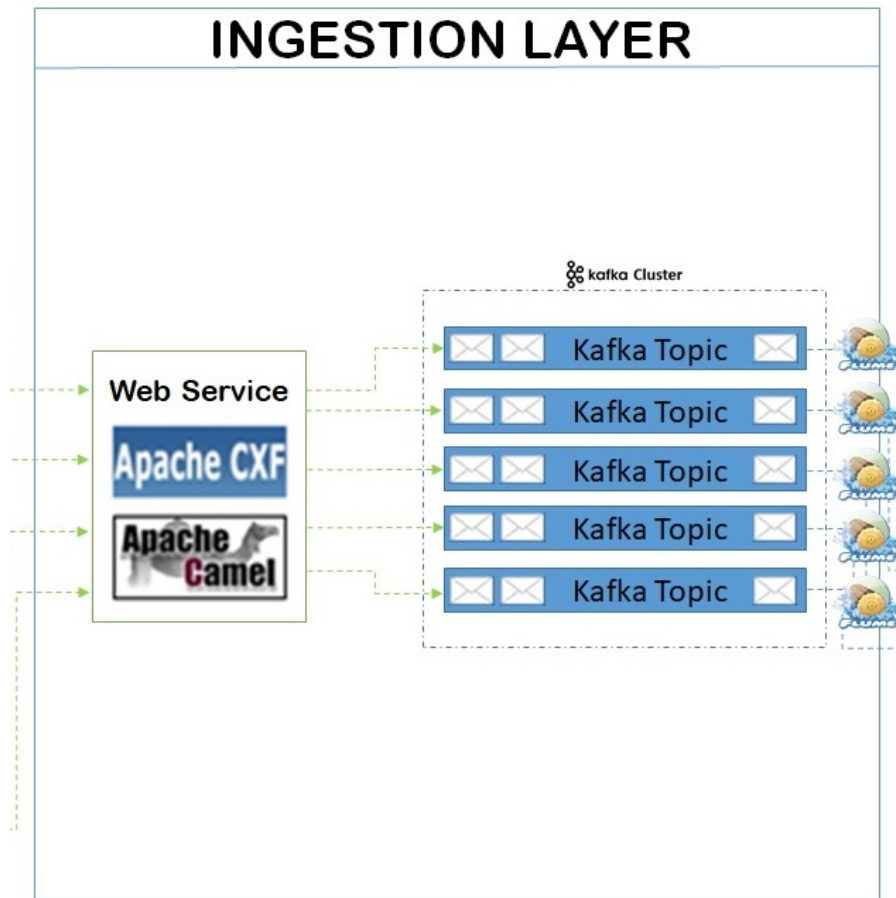
The ingestion layer, shown in figure 3.2, is the part that most needed to be redesigned. This consists of:

- A Web Service server.
- Apache Kafka.
- Apache Flume.

The Web Service server is the only point of access to the entire Big Data infrastructure. Based on SOAP protocol, this exposes an interface that devices can contact to send their own messages.

The server plays a fundamental role in the entire architecture as, in addition to being the access point to the infrastructure, it also takes care of processing the messages before being stored on HDFS.

Figure 3.2: Ingestion Layer



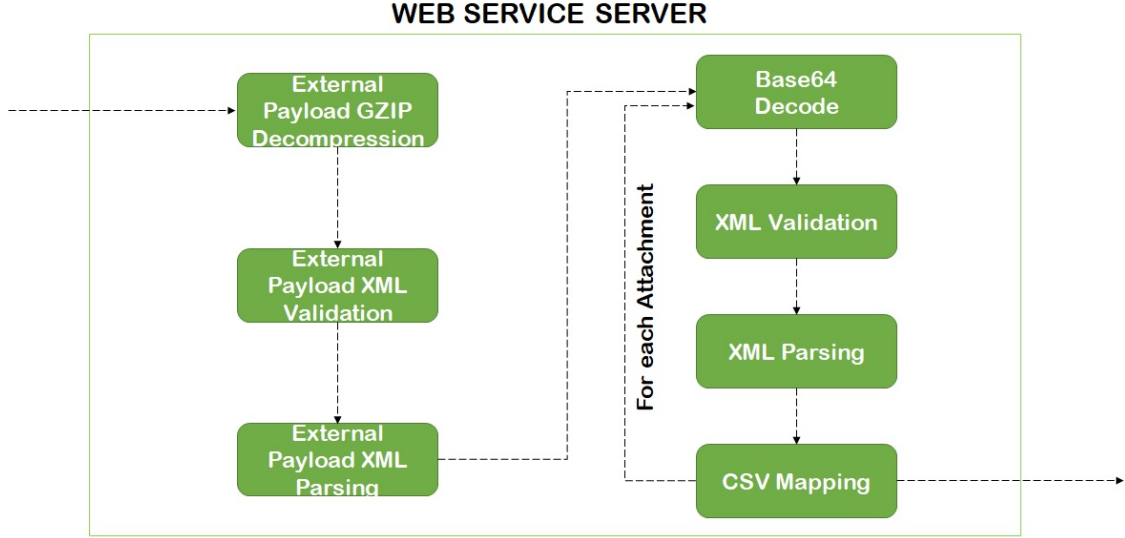
In figure 3.3 is shown the transformation chain to which messages are subject, in particular::

External Payload GZIP Decompression: Devices send messages containing a list of attachments in XML format. To reduce the amount of data sent to each transmission, the SOAP envelope payload is compressed. In this phase messages are decompressed; in the event that the payload is corrupt or can not be unzipped, the whole message is discarded.

External Payload XML Validation: The decompressed payload is validated against the WSDL schema. Even at this stage, if the validation fails, the whole message would be discarded.

External Payload XML Parsing: The payload is mapped to Java classes generated by the WSDL schema itself.

Figure 3.3: Web Service Server Internal



At this point we have obtained a list of attachments containing the data we are interested in. For each of them the following transformations are applied:

Base64 Decode: The attachments are further compressed by encoding them in Base64. Before being interpreted, these are decoded in order to obtain raw XML. An error in this step translates into the deviation of the single attachment.

XML Validation & Parsing: As in the case of the external payload, the decoded attachments are validated against the WSDL schema and subsequently mapped onto the generated Java classes.

CSV Mapping: Starting from the Java classes, complex data structures are created that can structure the data in tabular form. This phase became mandatory as Apache Camel was used for the integration between the server and Kafka.

During the study of the Web Service server, the ones that can be defined the weak points have been identified:

1. *Single Point of Failure:* the server was designed and developed as a single monolithic block. All the transformations described above are carried out by the same Java program. In the event that the server goes down, the entire infrastructure would no longer be reachable.
2. There is no error queue where you can temporarily store badly formatted messages.

3. Consequence of the previous point, if one of the first transformations fails, the whole message is discarded.

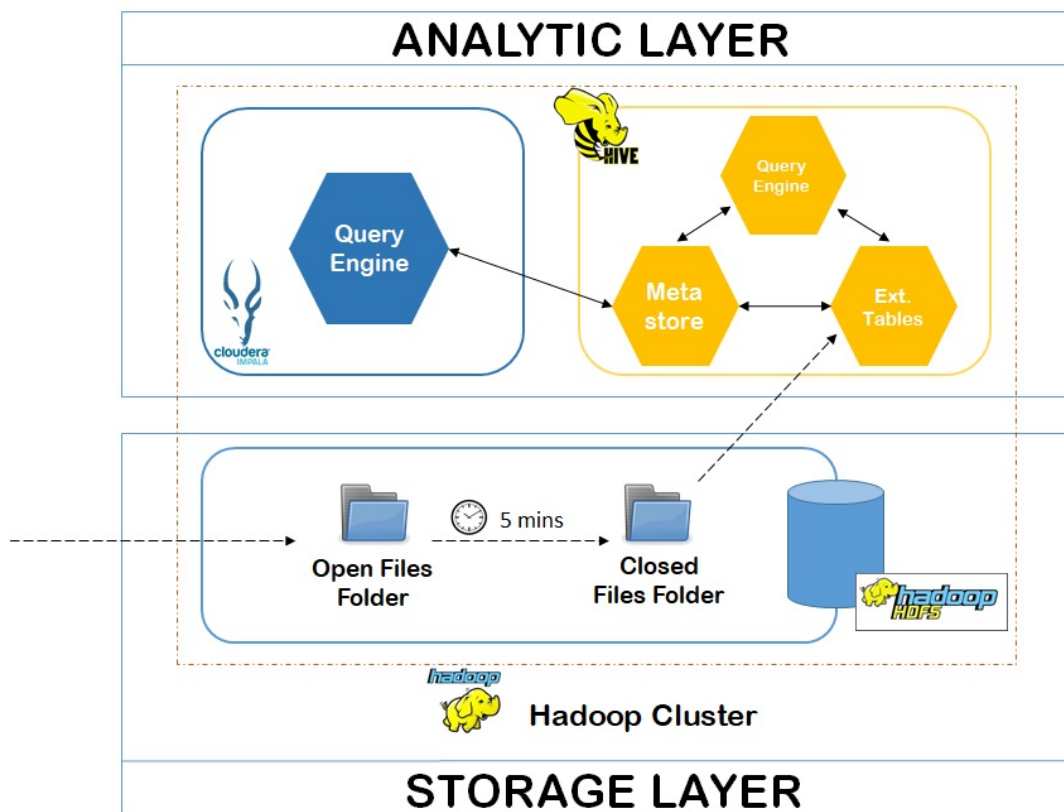
Leaving aside Kafka whose operation is known, Apache Flume takes care to consume the records published on the topic and write them in CSV files stored on HDFS.

The problems highlighted by the use of Flume, if we can define them so, are collected in the following two points:

1. Absence of a graphical interface for flow monitoring.
2. Operation based on a configuration file. As the flows increase, the configuration file becomes longer and more difficult to manage.

3.3 Storage Layer and Analytic Layer

Figure 3.4: Storage and Analytic Layer Internal



As mentioned in the previous paragraph, Flume takes care of importing data on HDFS; these will then be queried through tables.

Figure 3.4 shows a lower level architecture of the storage layer and the analytical layer.

The records, as soon as they are consumed from Kafka, are written on CSV files. These files are stored in a folder different from the final one: in this folder, in fact, there are all the files opened in writing that are waiting to be finalized. Every five minutes an automatic routine takes care of moving the closed files in what will be the final destination folder. The reason for this process will be clarified later.

Once the files have been moved you can query them: for this purpose, there are external tables, created through Hive, that point directly to the final folder.

The reasons that led to revisit the Storage and Analytic layers are summarized below:

1. Hive and Impala do not support all CRUD operations. The only operations allowed are, in fact, Create (INSERT) and Read (SELECT).
2. Impala is affected by some problems in the querying phase when the underlying files are still open. This forces the closed CSV files to be kept separate from those opened for writing operations: hence the two folders described above. Even if the operations of moving from one folder to another have been automated, it is still difficult to maintain these processes when the number of different flows to be managed is very high.

Chapter 4

DESIGN AND DEVELOPMENT

In this chapter will be presented the final architecture and how it has been developed. After a brief introduction about the environment used for the development, for each layer involved into the reengineering process will be shown the introduced tools and how they have been used.

4.1 Environment and Versions

The project was developed on a cluster consisting of three physical machines and two virtual machines, with the characteristics described in the table 4.1.

Table 4.1: Cluster Configuration

HOST	ROLE	TYPE	CPU	RAM	STORAGE
host01	Management	Virtual	8	16 GB	300 GB
host02	Management	Virtual	8	16 GB	300 GB
host03	Worker	Physical	40	189 GB	1 TB + 10 x 3,6 TB
host04	Worker	Physical	40	189 GB	1 TB + 10 x 3,6 TB
host05	Worker	Physical	40	189 GB	1 TB + 10 x 3,6 TB

The configuration of the nodes follows the master-slave model of the main Hadoop components.

The HDFS service is configured in *High Availability* mode. In particular, host01 is the active NameNode instance, while host02 is the standby instance.

The total capacity of HDFS is around 110 TB, distributed over three DataNodes. Each of them has a 1 TB hard disk designed to host the node local file system and ten 3.7 TB disks dedicated to HDFS. Is preferred to use many small disks, rather than a single large disk, mainly for reliability and performance reasons:

- The breakdown of a single large disk would result in the loss of a lot of data. The replication process of files blocks on other nodes in the cluster would negatively impact performances.
- I/O operations performed on different physical disks increase parallelism.

Notice that the overall capacity of the file system does not take into account the replication factor of the blocks, in this case three. The available capacity of HDFS is therefore equal to one third of the total capacity.

Communication to the outside takes place through 1 Gbps network interfaces, while in the internal network is designed to support a 10 Gbps communication.

The cluster management interface is available, in Cloudera distribution, through the *Cloudera Manager* service, accessible via a web browser at host01 address on port 7180. Through this service, it is possible to monitor the status of nodes, view information and statistics about resources in use and configure most of the services. The interface is shown in figure 4.1.

Finally another machine, outside the Hadoop cluster, has been used as a web server. It is a small Linux virtual machine with 8 cores and 8 GB of memory, enough to host the service.

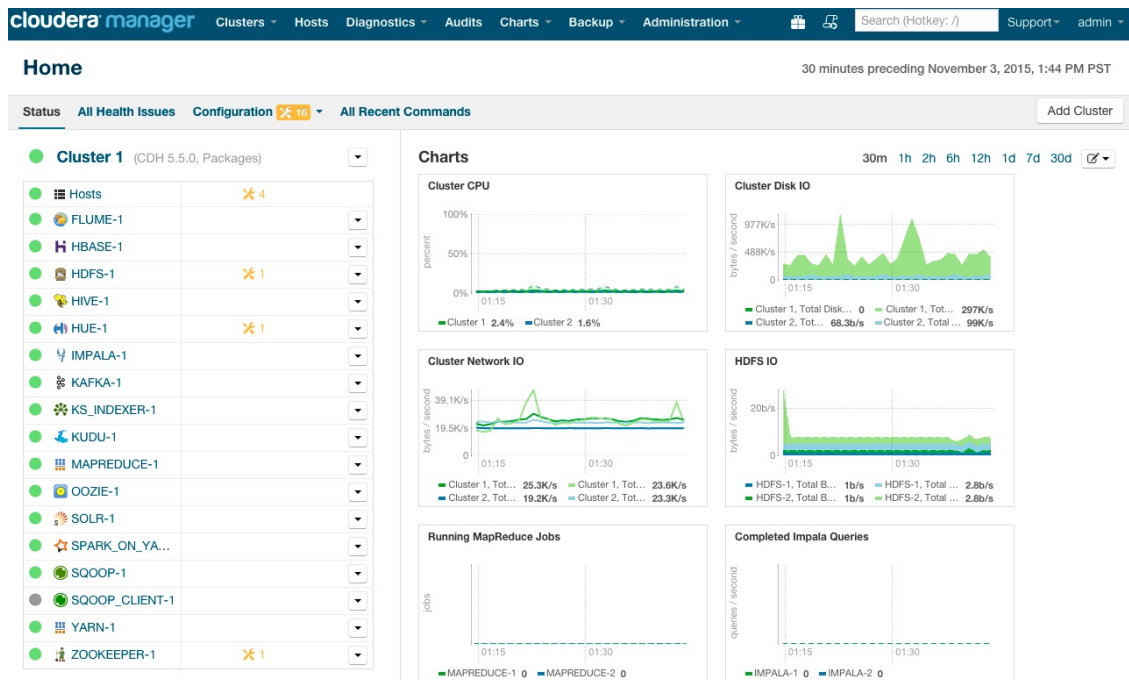
4.1.1 Versions

The Hadoop distribution used is the Cloudera distribution. In particular, the version installed on the cluster is *Cloudera CDH 5.13.1*.

In order to ensure compatibility between the components and to be able to obtain adequate support in case of need, Cloudera recommends to use, for each service, the version released with the distribution. For this reason, the following versions have been used:

- Apache Hadoop 2.6.0
- Apache Hive 1.1.0

Figure 4.1: Cloudera Manager Interface



- Cloudera Impala 2.10.0
- Apache Kudu 1.5.0
- Apache Kafka 2.1.1
- Apache ZooKeeper 3.4.5
- StreamSets 3.1.0.0

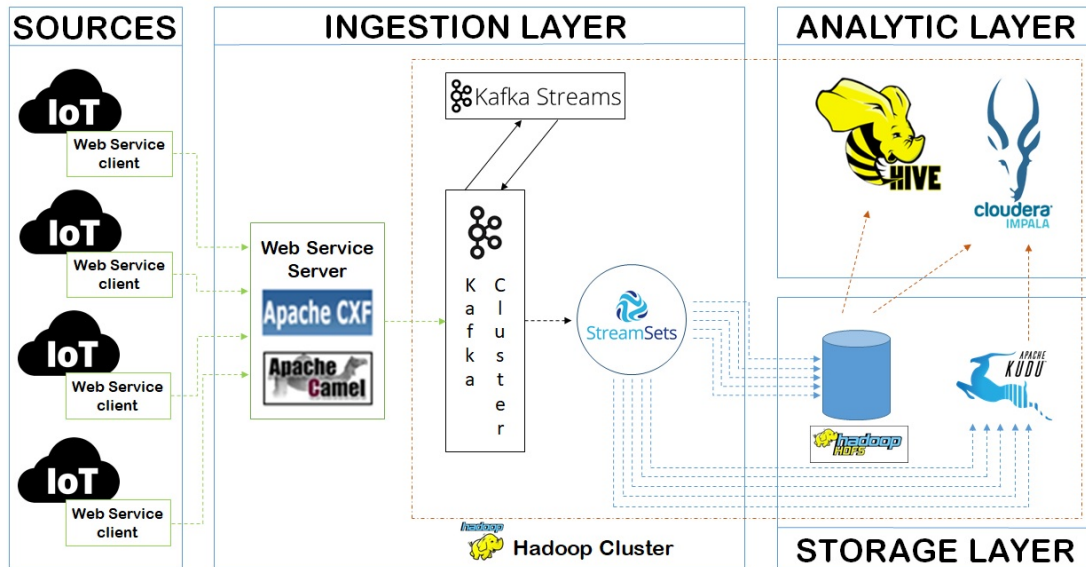
Some services are affected by known bugs, fixed in later versions. However, due to the inability to upgrade or install patches, it was necessary to use workarounds or relax project specifications to get around these issues.

4.2 Logical Architecture

In figure 4.2 is shown the logical architecture of the new system. From the image you can see where the new tools are located.

In the following paragraphs will be analyzed in detail the work done: for the re-designed layers, will be done an introduction of the new tools first and then will be shown the implementation.

Figure 4.2: Logical Architecture



4.3 Ingestion Layer

4.3.1 Introduced Tools

Kafka Streams API

Figure 4.3: Kafka Streams Logo

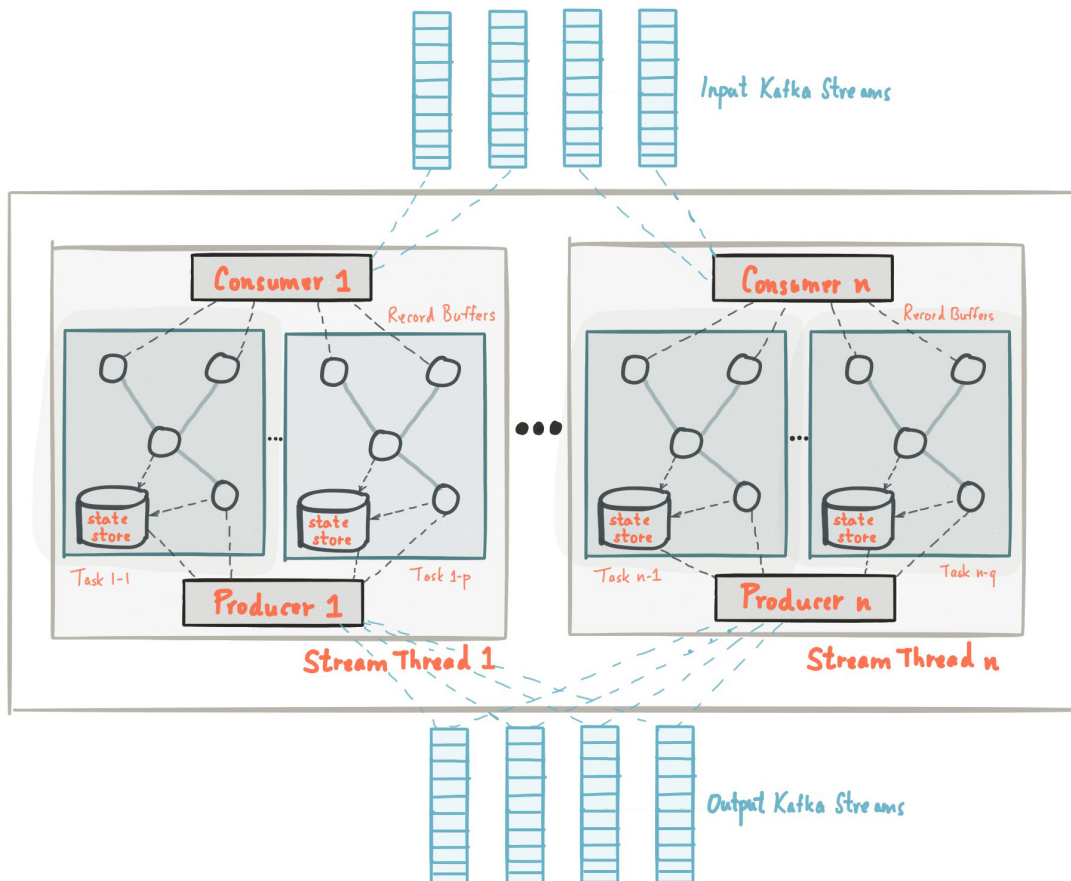


Kafka Streams is a library for building streaming applications, specifically applications that transform input Kafka topics into output Kafka topics. It lets you do this with concise code in a way that is distributed and fault-tolerant. Stream processing is a computer programming paradigm, equivalent to data-flow programming, event stream processing, and reactive programming, that allows some applications to more easily exploit a limited form of parallel processing [9].

Kafka Streams simplifies application development by building on the Kafka producer

and consumer APIs, and leveraging the native capabilities of Kafka to offer data parallelism, distributed coordination, fault tolerance, and operational simplicity. In figure 4.4 is shown the anatomy of an application that uses the Kafka Streams API. It provides a logical view of a Kafka Streams application that contains multiple stream threads, that each contain multiple stream tasks.

Figure 4.4: Kafka Streams Application Anatomy



The first aspect of how Kafka Streams makes building streaming services simpler is that it is cluster and framework free, it is just a library. Kafka Streams is one of the best Apache Storm alternatives. You don't need to set up any kind of special Kafka Streams cluster and there is no cluster manager, nimbus, daemon processes, or anything like that. If you have Kafka there is nothing else you need other than your own application code. The app code will coordinate with Kafka to handle failures, divvy up the processing load amongst instances, and rebalance load dynamically if more instances start up.

All this is accomplished by using the exact same group management protocol that

Kafka provides for normal consumers. The result is that a Kafka Streams app is just like any other service. It may have some local state on disk, but that is just a cache that can be recreated if it is lost or if that instance of the app is moved elsewhere.

The next way Kafka Streams simplifies streaming applications is that it fully integrates the concepts of *tables* and *streams*. The *stream-table duality* describes the close relationship between streams and tables:

Stream as Table: A stream can be considered a changelog of a table, where each data record in the stream captures a state change of the table. A stream is thus a table in disguise, and it can be easily turned into a "real" table by replacing the changelog from beginning to end to reconstruct the table. Similarly, aggregating data records in a stream will return a table.

Table as Stream: A table can be considered a snapshot, at a point in time, of the latest value for each key in a stream (a stream's data records are key/value pairs). A table is thus a stream in disguise, and it can be easily turned into a "real" stream by iterating over each key/value entry in the table.

By modeling the table concept in this way, Kafka Streams lets you compute derived values against the table using just the stream of changes. In other words it lets you process database change streams just as you would a stream of clicks.

StreamSets

Figure 4.5: StreamSets



Streamsets is a cloud native collection of products to control data drift. It provides two products, the *Data Collector* and the *Dataflow Performance Manager*. In this project has been used the Streamsets Data Collector.

The Data Collector is an open source application which allows users to build platform agnostic data pipelines. They are optimized for continuous ingestion and no data latency.

A pipeline describes the flow of data from the origin system to destination systems and defines how to transform the data along the way. A pipeline is made of stages, which are divided into three types:

Origin: an origin stage represents the source for the pipeline. Only one origin per pipeline is allowed.

Processor: a processor stage represents a type of data processing that you want to perform. You can use as many processors in a pipeline as you need.

Destination: a destination stage represents the target for a pipeline. You can use one or more destinations in a pipeline.

StreamSets Data Collector is released with many built-in stages, but allows also to develop custom stages in case of needs.

Building a pipeline is very simple: just drag-and-drop needed stages in the workspace, link them by an arrow and set few parameters Data Collector provides a graphical user interface (shown in figure 4.6) which lets you build and monitor batch and streaming data flows.

Figure 4.6: StreamSets Data Collector GUI



4.3.2 Implementation

The reengineering process of the ingestion layer was the activity that required more time and effort. Initially, a study phase of the solution in use was necessary to fully understand the functioning of the individual components and how they interacted with each other.

In the next sections the individual components of the ingestion layer will be analyzed showing how these have been modified. The main differences with the old version and the advantages that the new version has brought will be highlighted.

Web Service Server

As already mentioned in the chapter 3, the Web Service server is the only access point to the entire Big Data infrastructure. This, in addition to receiving messages from IoT devices, also took care of processing the messages before they were imported into HDFS.

During the study of the old server it was discovered that it managed about fifteen different types of messages, divided into three macro-groups. Keeping in mind the multitude of source devices, it is easy to imagine the complexity of the work done and the workloads to which it was subjected.

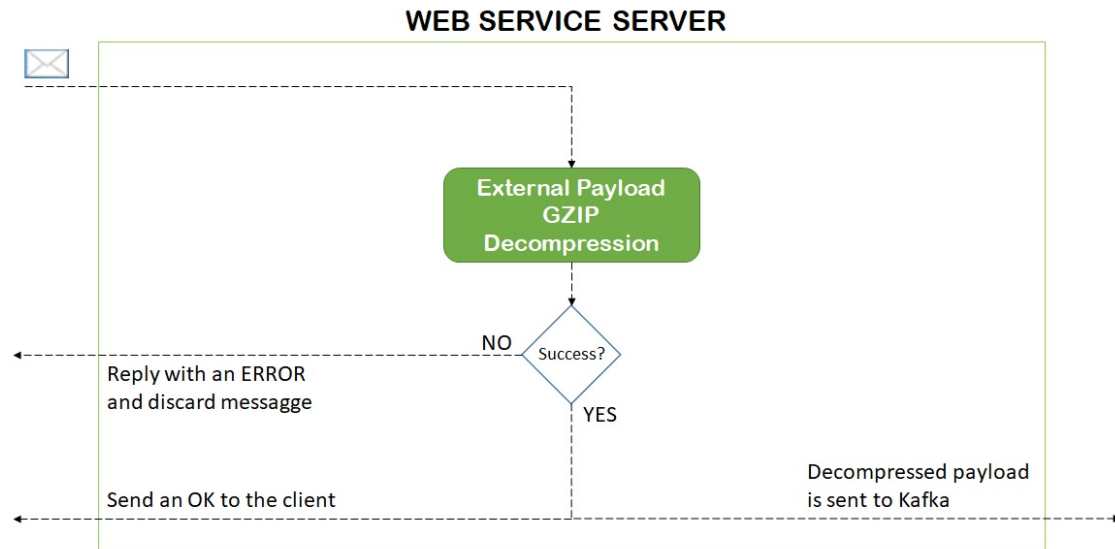
For the reasons mentioned above, it was decided to base the design of the new system on the decoupling of the role of access point from the role of processor of received messages.

Figure 4.7 shows the operation of the new Web Service server.

In the new implementation it was decided to keep only the decompression of the external payload inside the server. The reasons for this choice are essentially two and dependent on each other:

- The first reason is to reduce the number of discarded messages. In the old system, an error at any point in the elaboration was translated into the discard of the message, with the impossibility of knowing the reason for the latter. In the new system we thought to discard only the messages for which the decompression process fails, because if I can not decompress, the message is completely unusable and impossible to process.
- It was decided to introduce queues where to temporarily store the messages that can not be successfully imported on HDFS. The decoupling of the tasks was

Figure 4.7: New Web Service Server Internal



the pretext to review the error management system too: while before in case of an error the message was discarded, now, in every processing step, errors are managed with more accuracy and it is possible to know the reason why a message is not imported correctly.

With the reduction of the computational load the possibility of the server reacting unexpectedly to a request has also been reduced, reducing the possibility of sudden crashes that would make the service unavailable.

Successfully decompressed messages are mapped on a key/value pair and published to a Kafka topic. The generated pair is shown in the table 4.2.

Table 4.2: Web Service Server Output

	TYPE	CONTENT
Key	String	<messageID>_<messageTYPE>
Value	String	<decompressedPayload>

Kafka and Kafka Streams Applications

As mentioned in the previous section, the processing of the received messages is no longer carried out by the Web Service server.

During the study of the new solution different tools were analyzed. Of these the

pros and cons were evaluated trying to understand if they satisfied the following requirements:

Modularity: because in the old system the processing of the messages took place in a single point, the new tool had to allow us to split the processing into a series of independent modules and consequently to better balance the computational load.

Performance: given the volume of data, the tool had to guarantee high throughput and moderate consumption of resources. Not least the possibility of scaling in case of occurrence.

Integration with other components: the instrument had to be easily integrated with the existing components, in particular with Apache Kafka.

Among all the tools evaluated, it was decided to use Kafka Streams. In addition to being designed by the same company that owns Kafka and therefore perfectly integrated with it, being a Java library, Kafka Streams does not require the installation of any software or hardware components. Plus it's an easy-to-use and lightweight tool (the whole library consists of about ten thousand lines of code).

Figure 4.8: Kafka and Kafka Stream Applications

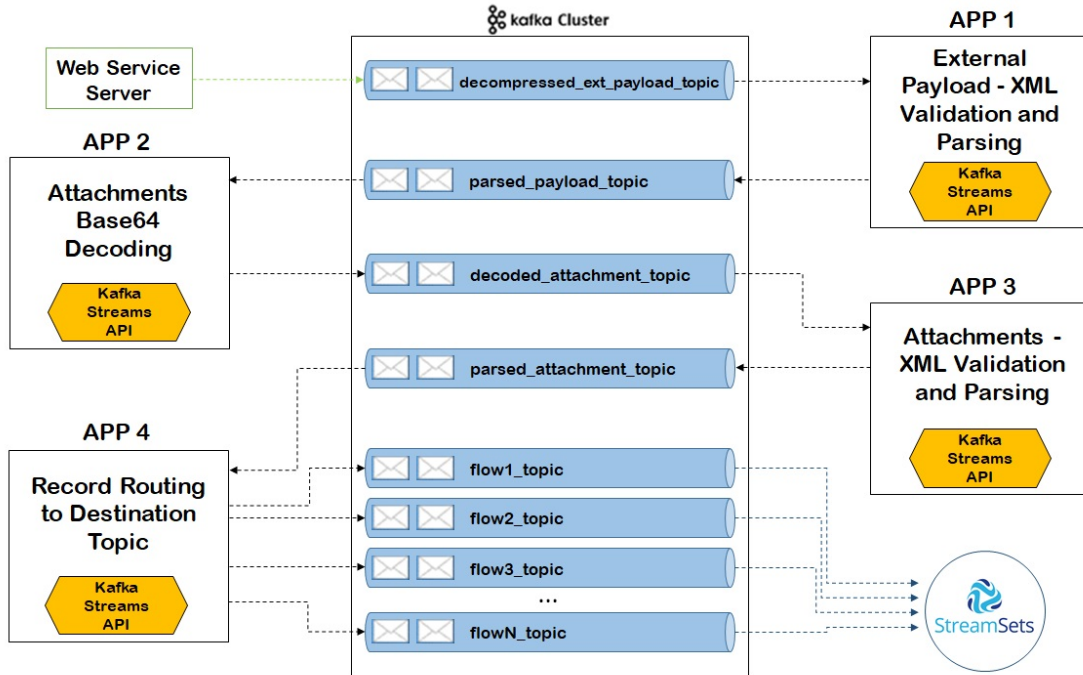


Figure 4.8 shows the new block, made by Kafka and Kafka Streams, responsible for processing the messages.

After a careful analysis of the whole message processing process, it was decided to split the processing into a chain of four applications based on the Kafka Streams APIs. Let's analyze in detail what they are dealing with and how they have been implemented.

Application 1: the application at the top of the chain takes care of validating and parsing the external payload. This consumes the payloads published by the Web Service server on the topic *decompressed_ext_payload_topic*, processes them and publishes the result on the destination topic.

The payloads are in XML format and follow this schema:

```
<ArrayOfUpload>
  <Upload>
    <ReportInfo>...</ReportInfo>
    <Attachments>
      <FileName>filename</FileName>
      <File>Base64 Encoded Attachment</File>
    </Attachments>
  </Upload>
</ArrayOfUpload>
```

Code 4.1: External Payload Sample

ArrayOfUpload, as the name implies, is a list of *Upload*. Each Upload is characterized by a header, contained in the *ReportInfo* tag, and by a list of attachments, where for each one is specified the name (tag *FileName*) and the content encoded in Base64 (tag *File*).

Once the message from the topic is consumed, the application validates the XML payload against the corresponding scheme. While payloads that fail validation are published on an error topic, the properly validated payloads are parsed, obtaining the Java class representative of XML.

```
1 public class Application1
2 {
3     @SuppressWarnings("unchecked")
4     public static void main( String[] args )
5     {
```

```
6      Properties config = new Properties();
7      config.put(StreamsConfig.APPLICATION_ID_CONFIG,
8          "application1");
9      config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
10         "localhost:9092");
11      config.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
12         Serdes.String().getClass());
13      config.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
14         Serdes.String().getClass());
15
16      KStreamBuilder builder = new KStreamBuilder();
17      KStream<String, String> decompressedExtPayload
18          =
19          builder.stream("decompressed_ext_payload_topic");
20
21      KStream<String, String> validatedExtPayload =
22          decompressedExtPayload.map(new
23              KeyValueMapper<String, String,
24                  KeyValue<String, String>>(){
25
26          @Override
27          public KeyValue<String, String> apply(String
28              key, String value) {
29              try {
30                  // Validation procedure against schema
31              } catch (JAXBException|SAXException e) {
32                  // Validation fails -> prepend "Error_" to
33                  // the key and return
34                  String newKey = "Error_" + key;
35                  return new KeyValue<String, String>(newKey,
36                      value);
37              }
38
39              // Validation success -> Just return the
40              // KeyValue as it is
41              return new KeyValue<String, String>(key,
42                  value);
43          }
44      });
```

```
30
31     KStream<String, String>[] branches =
        validatedExtPayload.branch(new
        Predicate<String, String>(){
32 @Override
33 public boolean test(String key, String value) {
34     if(key.startsWith("Error_"))
35         return true;
36     else
37         return false;
38 }
39     }, new Predicate<String, String>(){
40 @Override
41 public boolean test(String key, String value) {
42     if(!key.startsWith("Error_"))
43         return true;
44     else
45         return false;
46 }
47     });
48
49     KStream<String, String> validPayload =
        branches[1];
50
51     KStream<String, ArrayOfUpload> parsedPayload =
        validPayload.map(new KeyValueMapper<String,
        String, KeyValue<String, ArrayOfUpload>>(){
52 @Override
53 public KeyValue<String, ArrayOfUpload>
        apply(String key, String value) {
54     ArrayOfUpload uploads = // Parsing Procedure
        generating the Java object
55     return new KeyValue<String,
        ArrayOfUpload>(key, uploads);
56 }
57     });
58
```

```
59      // Send failed payloads to the error topic,  
        while the others to the destination topic  
60      KStream<String, String> errorPayload =  
          branches[0];  
61      errorPayload.to(Serdes.String(),  
          Serdes.String(), "error_payload_topic");  
62  
63      final Serializer<ArrayOfUpload> aouSerializer  
        = new ArrayOfUploadSerializer();  
64      final Deserializer<ArrayOfUpload>  
        aouDeserializer = new  
          ArrayOfUploadDeserializer();  
65      final Serde<ArrayOfUpload> aouSerde =  
        Serdes.serdeFrom(aouSerializer,  
          aouDeserializer);  
66      parsedPayload.to(Serdes.String(), aouSerde,  
          "parsed_payload_topic");  
67  
68      KafkaStreams streams = new  
        KafkaStreams(builder, config);  
69      streams.start();  
70    }  
71  }
```

Code 4.2: Application 1 Pseudocode

In the listing 4.2 the code executed by the application 1 is shown, showing only the transformations that are applied to the streams.

The first applied transformation is the *map* transformation (line 26). For each input record, validation against the reference schema is performed: if successful, the key/value pair is returned without changing anything, otherwise the string *Error_* is added at the beginning of the key.

Once the payload has been validated, you must separate the valid ones from the invalid ones. To do this, it has been used the *branch* transformation (line 42). This transformation allows to specify a list of predicates that the incoming records must satisfy and, based on what happens, to sort them in a certain number of output streams. The predicates are evaluated in order and at the end the record is assigned to one and only one output stream.

Once the records are separated, the valid ones are parsed by using *amap* transformation again (line 62).

The last step is to publish the records on the destination topic: invalid records are published on an error queue (line 72), while those correctly validated and parsed are published on the topic that will use application 2 (line 76).

Application 2: the elaboration carried out by the second application consists in decoding from Base64 the attachments of the various Uploads. Since a Upload can contain one or more attachments, a record is issued for each of these by duplicating the information contained in the header where necessary.

```
1 public class Application2
2 {
3     @SuppressWarnings("unchecked")
4     public static void main( String[] args )
5     {
6         final Serializer<ArrayOfUpload> aouSerializer =
7             new ArrayOfUploadSerializer();
8         final Deserializer<ArrayOfUpload>
9             aouDeserializer = new
10             ArrayOfUploadDeserializer();
11         final Serde<ArrayOfUpload> aouSerde =
12             Serdes.serdeFrom(aouSerializer,
13                             aouDeserializer);
14
15         Properties config = new Properties();
16         config.put(StreamsConfig.APPLICATION_ID_CONFIG,
17                     "application2");
18         config.put(StreamsConfig.BootstrapServersConfig,
19                     "localhost:9092");
20         config.put(StreamsConfig.KeySerdeClassConfig,
21                     Serdes.String().getClass());
22         config.put(StreamsConfig.ValueSerdeClassConfig,
23                     aouSerde);
24
25         KStreamBuilder builder = new KStreamBuilder();
26         KStream<String, ArrayOfUpload> parsedPayload =
27             builder.stream("parsed_payload_topic");
28     }
29 }
```

```
19      KStream<String, Attachment> attachment =
        parsedPayload.flatMap(new
            KeyValueMapper<String, ArrayOfUpload,
                KeyValue<String, Attachment>>(){
20      @Override
21      public KeyValue<String, Attachment> apply(String
            key, ArrayOfUpload value) {
22      List<KeyValue<String, Attachment>> result =
            new LinkedList<>();
23
24      for(Upload upload : value) {
25          // attachment(attachments) is(are) decoded
            from Base64
26          // decoded attachment is mapped on an
            Attachment object, containing header
            information and the attachment content
27
28          result.add(KeyValue.pair(key, attachment));
29      }
30
31      return result;
32      }
33      });
34
35      final Serializer<Attachment> aSerializer = new
        AttachmentSerializer();
36      final Deserializer<Attachment> aDeserializer =
        new AttachmentDeserializer();
37      final Serde<Attachment> aSerde =
        Serdes.serdeFrom(aSerializer,
            aDeserializer);
38      attachment.to(Serdes.String(), aSerde,
        "decoded_attachment_topic");
39
40      KafkaStreams streams = new
        KafkaStreams(builder, config);
41      streams.start();
42      }
```


43 }

Code 4.3: Application 2 Pseudocode

The listing 4.3 shows the operations performed by the application 2. Compared to the first application, the whole processing is performed using a single transformation.

Through the *flatMap* transformation (line 19), for each incoming record we can generate a variable number of output records. In our case for each upload we generate as many records as the number of attachments that the Upload carries with it. Each attachment is mapped to an *Attachment* object, which in addition to transporting the contents of the attachment itself, also includes the information contained in the header, common to each attachments.

Application 3: in this application the processing is similar to the one carried out in application 1. The attachments obtained from the various uploads are validated against the reference scheme and subsequently mapped onto the appropriate Java objects. Also in this case the attachments that do not pass the validation are published on an error topic, different from the one used in the first application.

```
1 public class Application3
2 {
3     @SuppressWarnings("unchecked")
4     public static void main( String[] args )
5     {
6         final Serializer<Attachment> aSerializer = new
              AttachmentSerializer();
7         final Deserializer<Attachment> aDeserializer =
              new AttachmentDeserializer();
8         final Serde<Attachment> aSerde =
              Serdes.serdeFrom(aSerializer,
                              aDeserializer);
9
10        Properties config = new Properties();
11        config.put(StreamsConfig.APPLICATION_ID_CONFIG,
12                  "application3");
13        config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
14                  "localhost:9092");
15        config.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
16                  Serdes.String().getClass());
```

```
14         config.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
15                     aSerde);
16
17         KStreamBuilder builder = new KStreamBuilder();
18         KStream<String, Attachment> decodedAttachment =
19             builder.stream("decoded_attachment_topic");
20
21         KStream<String, Attachment>
22             validatedAttachment =
23                 decodedAttachment.map(new
24                     KeyValueMapper<String, Attachment,
25                     KeyValue<String, Attachment>>(){
26 @Override
27 public KeyValue<String, Attachment> apply(String
28     key, Attachment value) {
29     try {
30         // Validation procedure against schema
31     } catch (JAXBException | SAXException e) {
32         // Validation fails -> prepend "Error_" to
33         // the key and return
34         String newKey = "Error_" + key;
35         return new KeyValue<String, String>(newKey,
36             value);
37     }
38
39     // Validation success -> Just return the
40     // KeyValue as it is
41     return new KeyValue<String, String>(key,
42         value);
43 }
44     });
45
46 // Divide well formatted records from the badly
47 // formatted ones
48 KStream<String, Attachment>[] branches =
49     validatedAttachment.branch(new
50         Predicate<String, Attachment>(){
51 @Override
```

```
38     public boolean test(String key, Attachment
        value) {
39         if(key.startsWith("Error_"))
40             return true;
41         else
42             return false;
43     }
44     }, new Predicate<String, Attachment>(){
45     @Override
46     public boolean test(String key, Attachment
        value) {
47         if(!key.startsWith("Error_"))
48             return true;
49         else
50             return false;
51     }
52     });
53
54     KStream<String, Attachment> validPayload =
        branches[1];
55
56     KStream<String, ParsedAttachment>
        parsedPayload = validPayload.map(new
        KeyValueMapper<String, Attachment,
        KeyValue<String, ParsedAttachment>>(){
57     @Override
58     public KeyValue<String, ParsedAttachment>
        apply(String key, Attachment value) {
59         ParsedAttachment parsedAttachment = // Parsing
        Procedure generating the Java object
60         return new KeyValue<String,
        ParsedAttachment>(key, parsedAttachment);
61     }
62     });
63
64     // Send failed payloads to the error topic,
        while the others to the destination topic
```

```
65         KStream<String, String> errorPayload =
            branches[0];
66         errorPayload.to(Serdes.String(), aSerde,
            "error_attachment_topic");
67
68         final Serializer<ParsedAttachment>
            paSerializer = new
            ParsedAttachmentSerializer();
69         final Deserializer<ParsedAttachment>
            paDeserializer = new
            ParsedAttachmentDeserializer();
70         final Serde<ParsedAttachment> paSerde =
            Serdes.serdeFrom(paSerializer,
            paDeserializer);
71         parsedPayload.to(Serdes.String(), paSerde,
            "parsed_attachment_topic");
72
73         KafkaStreams streams = new
            KafkaStreams(builder, config);
74         streams.start();
75     }
76 }
```

Code 4.4: Application 3 Pseudocode

The flow of transformations performed by the application 3 is shown in the listing 4.4. The transformations applied are the same applied by the application 1. The *ParsedAttachment* object is a wrapper class that contains the header information and the object representing the attachment.

Application 4: the last application in the chain simply acts as a router. Starting from the *ParsedAttachment* object are generated the records that will be written in the CSV files on HDFS. Then, using a field in the header, the records are published on the appropriate destination topic.

```
1 public class Application4
2 {
3     @SuppressWarnings("unchecked")
4     public static void main( String[] args )
5     {
```

```
6      final Serializer<ParsedAttachment> paSerializer
          = new ParsedAttachmentSerializer();
7      final Deserializer<ParsedAttachment>
          paDeserializer = new
          ParsedAttachmentDeserializer();
8      final Serde<ParsedAttachment> paSerde =
          Serdes.serdeFrom(paSerializer,
          paDeserializer);
9
10     Properties config = new Properties();
11     config.put(StreamsConfig.APPLICATION_ID_CONFIG,
          "application4");
12     config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
          "localhost:9092");
13     config.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
          Serdes.String().getClass());
14     config.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
          paSerde);
15
16     KStreamBuilder builder = new KStreamBuilder();
17     KStream<String, ParsedAttachment>
          parsedAttachmet =
          builder.stream("parsed_attachment_topic");
18
19     KStream<String, String> record =
          parsedAttachment.flatMap(new
          KeyValueMapper<String, ParsedAttachment,
          KeyValue<String, String>>(){
20     @Override
21     public KeyValue<String, String> apply(String
          key, ParsedAttachment value) {
22     List<KeyValue<String, String>> records = new
          LinkedList<>();
23
24     // from the attachment object contained into
          ParsedAttachment are generated the final
          records
```

```
25         // the Key of the new KeyValue object is set
           using a specific field contained in the
           header included in the ParsedAttachment
           object
26
27         return records;
28     }
29     });
30
31     // Split records in N branches based on the
       destination flow
32     KStream<String, String>[] branches =
       validatedAttachment.branch(new
       Predicate<String, String>(){
33     @Override
34     public boolean test(String key, String value) {
35         if(key.startsWith("flow1"))
36             return true;
37         else
38             return false;
39     }
40     },
41     new Predicate<String, String>(){
42     @Override
43     public boolean test(String key, String value) {
44         if(!key.startsWith("flow2"))
45             return true;
46         else
47             return false;
48     }
49     },
50     . . . ,
51     new Predicate<String, String>(){
52     @Override
53     public boolean test(String key, String value) {
54         if(!key.startsWith("flowN"))
55             return true;
56         else
```

```
57         return false;
58     }
59     });
60
61     KStream<String, String> flow1Records =
62         branches[0];
63     KStream<String, String> flow2Records =
64         branches[1];
65     . . .
66     KStream<String, String> flowNRecords =
67         branches[N];
68
69     // Send records to the destination topic
70     flow1Records.to(Serdes.String(),
71                    Serdes.String(), "flow1_topic");
72     flow2Records.to(Serdes.String(),
73                    Serdes.String(), "flow2_topic");
74     . . .
75     flowNRecords.to(Serdes.String(),
76                    Serdes.String(), "flowN_topic");
77
78     KafkaStreams streams = new
79         KafkaStreams(builder, config);
80     streams.start();
81 }
82 }
```

Code 4.5: Application 4 Pseudocode

In the listing 4.5 the code executed by the application 4. Initially, through a *flatMap* transformation (line 19) the final records are generated, then from the header is taken the information necessary to split records between the various streams.

Also in this case the split of the records between the flows is done through the *branch* transformation (line 31). Finally, the records are published on the destination topics where StreamSets will read.

StreamSets

Once the records have been published on the Kafka topics, we need a tool that allows them to be consumed and written on HDFS. In the old system this task was carried out by Flume. The only problem with Flume is that it is based on a configuration file, which, in the case of a large number of flows, can become very big and difficult to manage.

In the new implementation it was decided to replace Flume with StreamSets. The latter, in addition to performing the same functions as Flume, also allows data to be processed as they pass through the pipeline. Moreover, everything is made through a web graphical interface, which makes the work simpler and more intuitive.

The implementation of one of the sixteen created pipelines will be illustrated below. The others differ only in destination paths and tables in which the records are inserted.

Figure 4.9: StreamSets Pipeline Sample

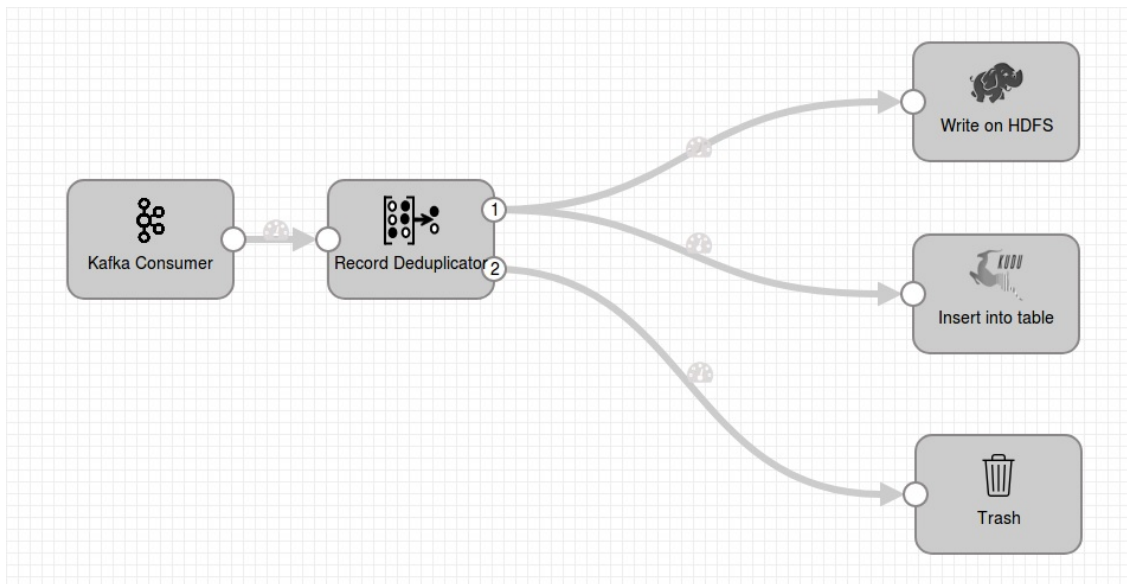


Figure 4.9 shows one of the pipelines that have been created. The pipeline composition was done simply by dragging the blocks into the workspace and connecting them with arrows. Let's see their configuration.

Kafka Consumer: as the name implies, the block implements a Kafka consumer. This is subscribed to one of the topics where the records were published.

As you can see from the figure 4.10, configuring a Kafka consumer in StreamSets is really immediate: in the *Broker URI* field you have to enter the addresses of the Kafka brokers followed by the port where the service is available; in the

Figure 4.10: Kafka Consumer Configuration

The screenshot shows the StreamSets Kafka Consumer configuration interface. The left sidebar has tabs for Info, Configuration, and Raw Preview. The main area has tabs for General, Kafka, and Data Format. The Kafka tab is active, showing the following configuration fields:

- Broker URI:** 10.240.136.193:9092,10.240.136.194:9092
- ZooKeeper URI:** 10.240.136.193:2181
- Consumer Group:** streamsetsDataCollector
- Topic:** flow1_topic
- Produce Single Record:** ☐
- Max Batch Size (records):** 1000
- Batch Wait Time (ms):** 2000
- Rate Limit Per Partition (Kafka messages):** 1000
- Kafka Configuration:** +

At the bottom right, there is a link to "Switch to bulk edit mode".

ZooKeeper URI field, enter the address of the server hosting the service; the *Consumer Group* field contains the group to which this consumer belongs; it is used for all the operations related to the offset with which the messages are identified; the *Topic* field simply contains the name of the topic to which the consumer subscribes. The values of the other fields are those of default and can be changed to increase or decrease the number of records that every second the consumer reads from the topic.

Record Deduplication: this block has the task of deleting duplicates. It caches record information for comparison until it reaches a specified number of records. The Record Deduplicator can show the entire records or subset of fields. To enhance the performance pipeline, the Record Deduplicator hashes comparison fields and uses the hashed values to evaluate for duplicates.

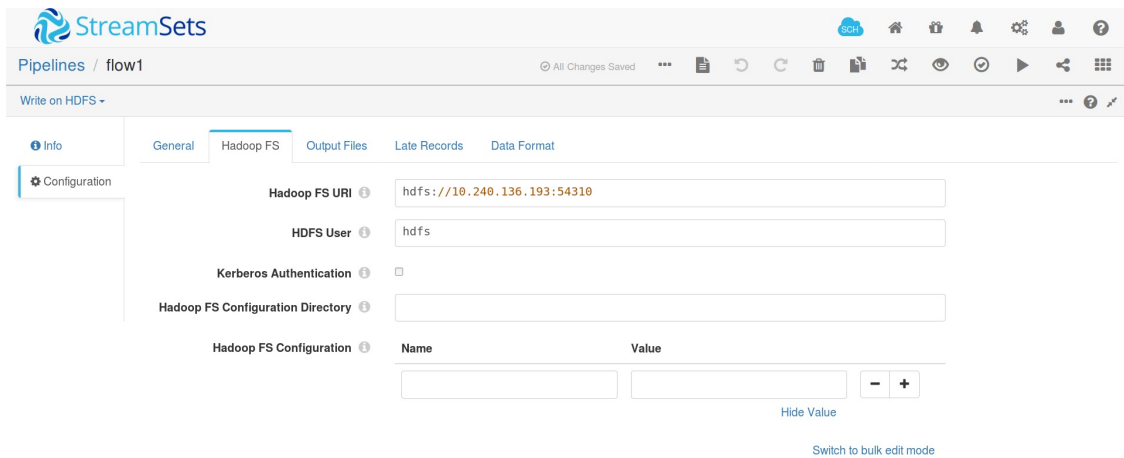
HDFS Sink: it writes data to the Hadoop Distributed File System.

In the *Hadoop FS* panel, shown in the figure 4.11, you need to set the address where you can contact HDFS (*Hadoop FS URI* field) and the user performing the operation (*HDFS user* field).

In the *Output Files* panel in figure 4.12, instead, you can specify the destination path where the records will be written (*Directory Template* field), and other information about the file that will be created, like the type, the prefix and the suffix of the file name and the desired size.

Kudu Sink: it writes data to a Kudu cluster.

Figure 4.11: HDFS Sink Configuration 1



StreamSets

Pipelines / flow1

Write on HDFS -

Info General **Hadoop FS** Output Files Late Records Data Format

Configuration

Hadoop FS URI

HDFS User

Kerberos Authentication ☐

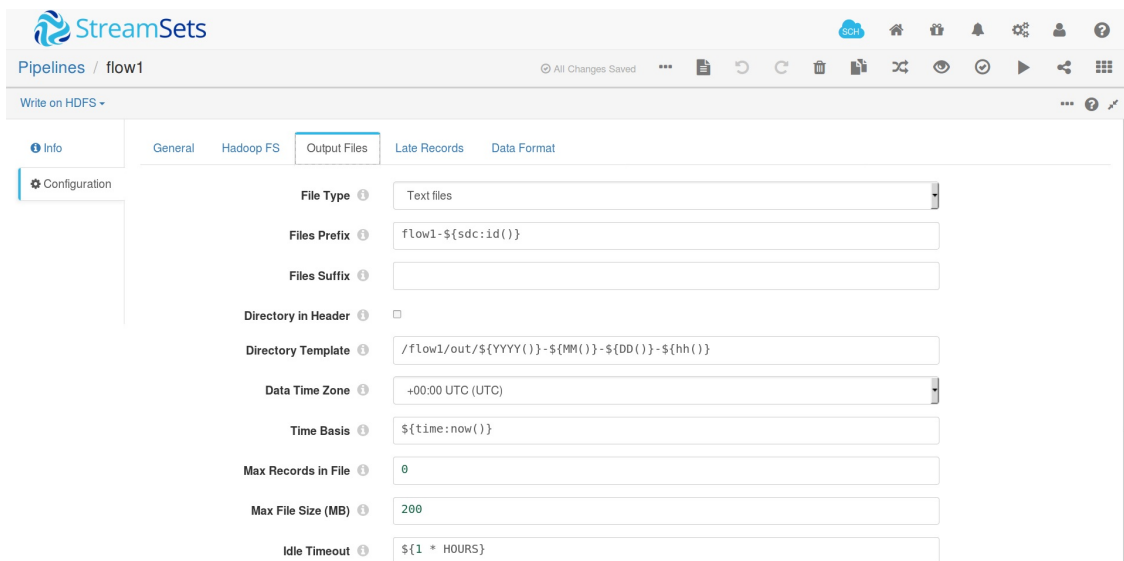
Hadoop FS Configuration Directory

Hadoop FS Configuration	Name	Value
	<input type="text"/>	<input type="text"/>

Hide Value

Switch to bulk edit mode

Figure 4.12: HDFS Sink Configuration 2



StreamSets

Pipelines / flow1

Write on HDFS -

Info General Hadoop FS **Output Files** Late Records Data Format

Configuration

File Type

Files Prefix

Files Suffix

Directory in Header ☐

Directory Template

Data Time Zone

Time Basis

Max Records in File

Max File Size (MB)

Idle Timeout

Its configuration, shown in figure 4.13, is very simple: after specifying the address of the master nodes of the Kudu cluster (*Kudu Masters* fields), you must indicate the destination table (*Table Name* field) and the operation you want to perform (*Default Operation* field).

Figure 4.13: Kudu Sink Configuration

The screenshot shows the StreamSets configuration page for a Kudu Sink. The 'Kudu' tab is selected. The configuration fields are as follows:

- Kudu Masters:** 10.240.136.193:8051
- Table Name:** \${record:attribute('flow1')}
- Field to Column Mapping:** A table with two columns: 'SDC Field' and 'Column Name'. It contains one row with empty fields and buttons for adding or removing rows.
- Default Operation:** INSERT
- Change Log Format:** None

There is a 'Switch to bulk edit mode' link next to the Field to Column Mapping table.

4.4 Storage Layer and Analytic Layer

4.4.1 Introduced Tool

Apache Kudu

Figure 4.14: Apache Kudu Logo

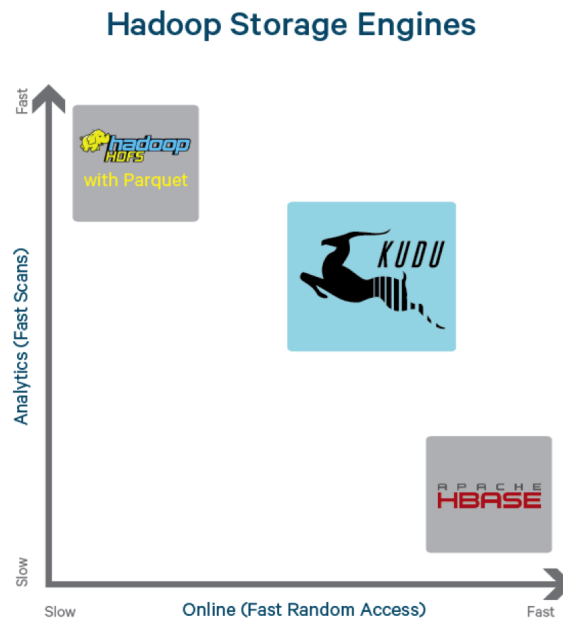


Kudu is a columnar storage manager developed for the Hadoop platform. Kudu shares the common technical properties of Hadoop ecosystem applications: it runs on commodity hardware, is horizontally scalable, and supports highly available operation [10].

Designed and developed by Cloudera, it is licensed under the Apache license. Since it was developed by Cloudera, it is natively integrated with the Hadoop environment, and can therefore be used by all of its components.

The objectives for which Kudu has been developed are manifold. First of all, to

Figure 4.15: Hadoop Storage Engines



create a storage layer that guarantees performance halfway between those provided by HDFS and HBASE. In fact, HDFS provides high performance on large datasets, especially through batch operations, which are potentially long. HBASE, on the other hand, has low latency in providing small query results, and allows data update, which does not allow HDFS.

Kudu presents itself as a hybrid choice between HDFS and HBASE, ensuring high throughput on large amounts of data (lower than HDFS but higher than HBASE), a low latency data access (lower than HDFS but higher than HBASE) and data update operations. The data are modeled in relational structures, accessible through an SQL-like language, guaranteeing ACID properties.

It is important to understand what Kudu is not. First of all it is not a SQL interface, in fact Kudu offers only a storage layer, like HDFS. If you want to integrate an SQL interface to query the data, you must use an additional tool such as Impala or Spark. Kudu does not use HDFS as a storage layer, being an alternative to it. Finally, Kudu is not a replacement for HDFS or HBASE. It is a product that goes to the other two already existing. Depending on the situation, you can decide which one is the most appropriate.

Before showing how Kudu works internally, we must introduce some fundamental concepts on which it is based:

Tablet: A tablet is a contiguous segment of a table. A given tablet is replicated on

multiple tablet servers, and one of these replicas is considered the leader tablet. Any replica can service reads, and writes require consensus among the set of tablet servers serving the tablet.

Tablet Server: A tablet server stores and serves tablets to clients. For a given tablet, one tablet server serves the lead tablet, and the others serve follower replicas of that tablet. Only leaders service write requests, while leaders or followers each service read requests. Leaders are elected using Raft consensus. One tablet server can serve multiple tablets, and one tablet can be served by multiple tablet servers.

Master: The master keeps track of all the tablets, tablet servers, the catalog table, and other metadata related to the cluster. At a given point in time, there can only be one acting master (the leader). If the current leader disappears, a new master is elected using the Raft Consensus Algorithm. The master also coordinates metadata operations for clients. All the master's data is stored in a tablet, which can be replicated to all the other candidate masters. Tablet servers heartbeat to the master at a set interval (the default is once per second).

Raft Consensus Algorithm: provides a way to elect a leader for a distributed cluster from a pool of potential leaders, or candidates. Other cluster members are followers, who are not candidates or leaders, but always look to the current leader for consensus. Kudu uses the Raft Consensus Algorithm for the election of masters and leader tablets, as well as determining the success or failure of a given write operation.

Now let's see what happens when a client wants to perform an update operation on a record in a table. In figure 4.17 the whole process is schematised.

1. the client asks the Master Server in which tablet of table T resides the record for todd@cloudera.com
2. the Master Server replies by indicating the tablet containing the searched record and the related servers holding the tablet. The Master Server also sends information about other tablets to which the client might be interested in.
3. the client caches the information received from the Master Server, so it does not need to request it again in case of subsequent operations on the same table.
4. The client performs the update operation of the record by sending the request to the server which holds the leader tablet. At the same time, the update operation is also replicated on the servers in possession of the tablet replicas. If the quorum is reached the operation ends successfully, otherwise the operation is aborted.

Figure 4.16: Kudu Architectural Overview

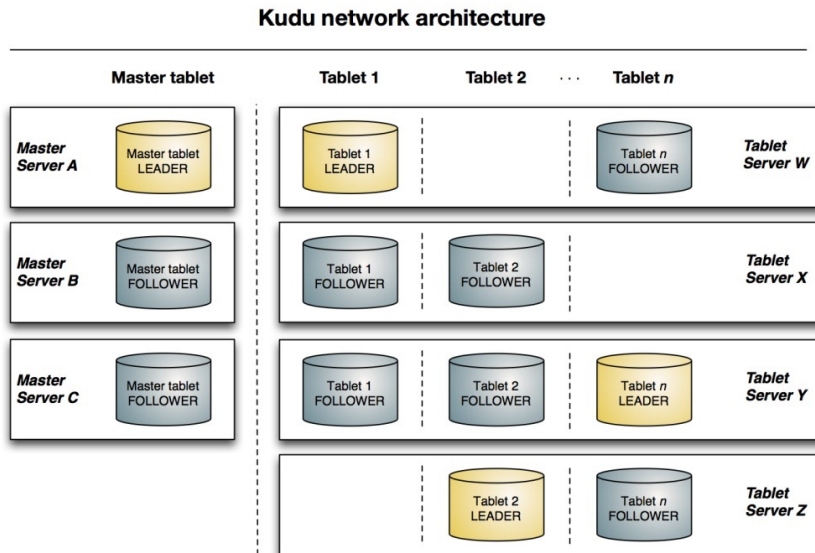
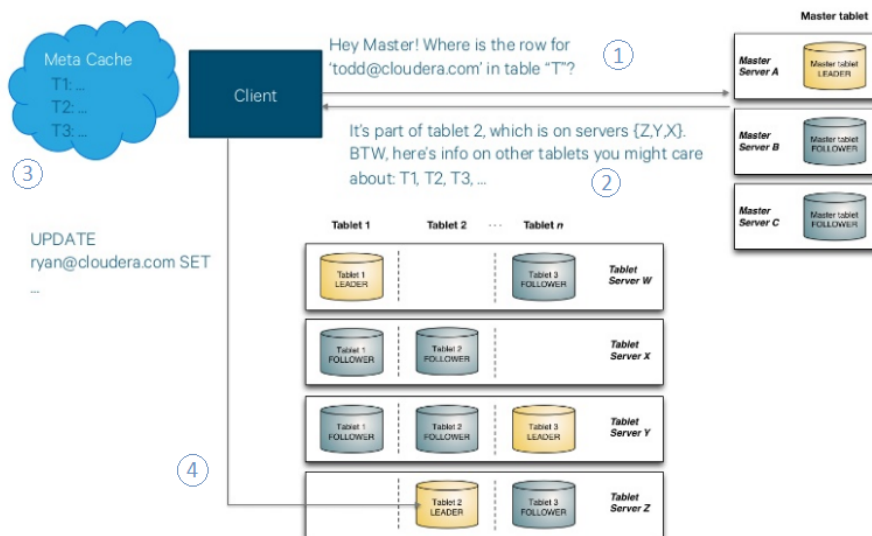


Figure 4.17: Update Operation Steps



4.4.2 Deployment

Starting with version 5.13 of Cloudera CDH, Kudu is part of the CDH parcel rather than being a separate parcel. The deployment was carried out through the Cloudera Manager service. The Kudu cluster configuration is as follows:

Table 4.3: Kudu Cluster Configuration

HOST	ROLE
host01	Master Server and Tablet Server
host02	Tablet Server
host03	Tablet Server

The configuration adopted consists of 1 Master Server and 3 Tablet Servers. One of the 3 physical machines that make up the cluster used for development, hosts both the service of the Master Server and the Tablet Server.

4.4.3 Performance Test

In this section we want to present the results obtained from a series of experiments, with the aim of showing how Kudu could fit in a real-time environment. For this purpose, the experiments were conducted comparing the performances obtained by Hive, Impala on HDFS and Impala on Kudu on a set of queries.

Data

For the experiment, four tables stored as parquet files were used. External tables have been created on these files using Hive, without performing partitioning. These tables have a variable number of records between 1 million to 3 million.

Data Modification (Insert/Update/Delete)

In this case random tests were performed. In particular, it was first tried to insert or delete a record from a table, then to insert and delete a few hundred records. In this situation Kudu was pretty fast with less than one second latency.

In the case of Hive/Impala the only way to delete a record is to reprocess the entire table, which, in the case of tables with billions of records, can also take several hours.

Random Look-ups

Also in this case simple tests were carried out. Random look-ups were performed on the primary key and the result was obtained within 5 seconds.

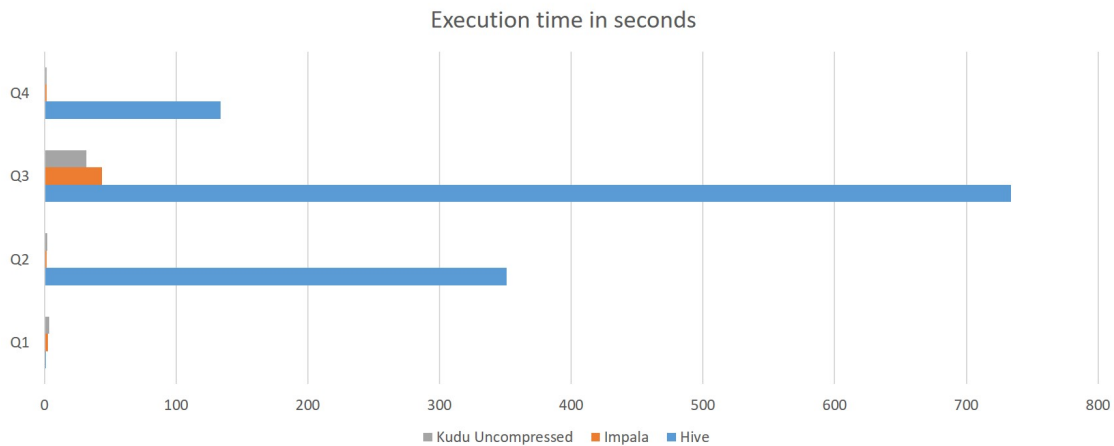
The same test was then carried out using Impala on parquet and on average the waiting time was between 120 and 200 seconds. However, keep in mind that the

tables are not partitioned, so Impala must scan the entire table to find only one line. Back to Kudu, look-ups on non-primary key fields are just as fast thanks to its columnar storage.

Queries

For this experiment, four queries were prepared that were made first with Hive, then with Impala on HDFS and finally with Impala on Kudu. The first two queries involve only one table, while the second two perform join operations between multiple tables. In figure 4.18 the execution time of the four queries is shown in the three cases mentioned above.

Figure 4.18: Hive vs Impala on HDFS vs Impala on Kudu



As you can see, the graph is dominated by the long execution times required by Hive. Only in one case Hive has execution times comparable to those of Impala on HDFS and Impala on Kudu, in particular when the query is a selection of all the fields of all the records contained in the table. The long run times of Hive are mainly due to the fact that the operations are not performed entirely in the main memory as Impala does, moreover, more complex the query is, greater is the number of map-reduce jobs that Hive has to instantiate. For better readability, in the figure 4.19 the results are shown excluding Hive.

The graph shows that Kudu does not differ so much from Impala and even, in the longer query, Kudu beats Impala.

From the first tests it was noticed that the first time a query was executed, it was

Figure 4.19: Impala on HDFS vs Impala on Kudu



about two times slower than the following executions. This could be due to the operating system that caches data once it has been accessed. Queries that query unread tables cause these to be read from disk, while queries that access cached tables are faster because they are read from RAM.

In figure 4.20 it is possible to observe the difference in terms of execution time between a "cold" and a "worm" run.

Figure 4.20: Cold Run vs Worm Run



Note that for the longest queries, Q3, the time difference is not as significant as for fast queries Q1, Q2 and Q4, where the execution time is almost halved.

Chapter 5

CONCLUSIONS AND FUTURE DEVELOPMENTS

The work presented in this thesis shows a possible solution to a classic reengineering problem. The long process of design and development of the new architecture has allowed us to know and explore the technologies that in recent years have entered the Big Data world.

Kafka Streams proved to be a very valuable tool for creating applications that can process real-time data flows. Thanks to its native integration with Apache Kafka, the developer has only the task of establishing the transformation chain that the flow must undergo. In fact, the developer does not have to worry about managing the offset committing phase, as it is all managed within the Kafka Streams library using the same internal messaging layer used by Kafka. Kafka Streams is a tool destined to grow a lot in the near future thanks to the continuous increase of devices connected to the Internet and the need for tools able to manage the huge amount of data generated by the latter.

A special mention goes to StreamSets, maybe the most surprising tool among all those used to implement the new solution. Its strong point is certainly the simplicity of use: the graphical interface, in addition to making the user experience more pleasant, allows you to create pipelines quickly and easily thanks to the drag-and-drop system. Furthermore, creating a pipeline does not require the knowledge of a programming

language, but only to properly configure the various stages that comprise it. This is a very important factor, as it allows the developer to immediately access the full potential of the platform. StreamSets is perhaps the most promising tool, among all those present, for ingestion and on-the-fly data processing, thanks also to the great community behind it.

During the tests performed, Kudu proved to be a good tool for supporting HDFS. We have been surprised by its ability to modify and update the database quickly, which is very important in a context where data is rapidly changing. We have seen how, through Impala, it is possible to query data in few seconds, allowing fast analytics on fast data.

Currently the new solution is being tested to verify that all the various types of messages are handled correctly. In the coming months it is planned to carry out a load test to evaluate the behavior of the solution when subjected to real workloads, in order then to release it in the production environment.

5.0.1 Future Developments

As anticipated in the chapter 4, in the new solution was decided to manage more accurately the messages that, for one reason or another, were discarded. In particular, two Kafka topics have been added where are published messages that are not considered compliant with the specifications. Currently the only way to access information on discarded messages is through the appropriate Hive tables.

To make access to this information faster, it was decided to implement a web interface. This interface allows you to quickly know the reason why a message has been rejected, leaving to the developer only the task of investigating and possibly correcting the problem.

Once the problem is corrected, the discarded messages must be re-processed to be imported to HDFS. Currently the process is managed manually by the developer who sends back discarded messages to the Web Service server via a series of SOAP calls. For the reason just mentioned, it was decided to automate the entire process by implementing a module that would query the tables mentioned in the previous paragraph and send the messages back by contacting the Web Service server. The application would be appropriately scheduled to be run periodically so as to relieve the developer of this task.

List of Figures

2.1	Growth estimation of data volume produced every year between 2010 and 2020	7
2.2	Big Data Vs	9
2.3	Hadoop Logo	10
2.4	Hadoop Ecosystem	11
2.5	Typical configuration of a Hadoop Cluster	12
2.6	Configuration with replication factor equal three	14
2.7	NameNode and DataNode in HDFS	15
2.8	Behaviour of the Secondary NameNode	17
2.9	Word Count in MapReduce	21
2.10	YARN Architecture	23
2.11	Kafka Logo	25
2.12	Kafka Architecture	25
2.13	Topic Anatomy	26
2.14	Topic Consumer	27
2.15	Flume Logo	28
2.16	Flume Agent	28
2.17	Flafka Architecture Example	29
2.18	Hive Logo	29
2.19	Hive Architecture	31
2.20	Impala Logo	32
2.21	Impala Architecture	32

3.1	Logical Architecture	36
3.2	Ingestion Layer	37
3.3	Web Service Server Internal	38
3.4	Storage and Analytic Layer Internal	39
4.1	Cloudera Manager Interface	43
4.2	Logical Architecture	44
4.3	Kafka Streams Logo	44
4.4	Kafka Streams Application Anatomy	45
4.5	StreamSets	46
4.6	StreamSets Data Collector GUI	47
4.7	New Web Service Server Internal	49
4.8	Kafka and Kafka Stream Applications	50
4.9	StreamSets Pipeline Sample	64
4.10	Kafka Consumer Configuration	65
4.11	HDFS Sink Configuration 1	66
4.12	HDFS Sink Configuration 2	66
4.13	Kudu Sink Configuration	67
4.14	Apache Kudu Logo	67
4.15	Hadoop Storage Engines	68
4.16	Kudu Architectural Overview	70
4.17	Update Operation Steps	70
4.18	Hive vs Impala on HDFS vs Impala on Kudu	72
4.19	Impala on HDFS vs Impala on Kudu	73
4.20	Cold Run vs Worm Run	73

List of Tables

4.1	Cluster Configuration	41
4.2	Web Service Server Output	49
4.3	Kudu Cluster Configuration	71

List of Codes

4.1	External Payload Sample	51
4.2	Application 1 Pseudocode	51
4.3	Application 2 Pseudocode	55
4.4	Application 3 Pseudocode	57
4.5	Application 4 Pseudocode	60

Bibliography

- [1] John Gantz and David Reinsel, *THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East*, 2012.
- [2] Apache Software Foundation, *Apache Hadoop*, <http://hadoop.apache.org>, 2018.
- [3] Apache Software Foundation, *Apache Hadoop powered by*, <https://wiki.apache.org/hadoop/PoweredBy>, 2018.
- [4] Apache Software Foundation, *Apache license-2.0*, <http://www.apache.org/licenses/LICENSE-2.0>, 2004.
- [5] Sanjay Ghemawat and Howard Gobioff and Shun-Tak Leung, *The Google File System*, 2003.
- [6] Tom White, *Hadoop: The Definitive Guide, 4th Edition*, O'Reilly Media, 2004.
- [7] Jeffrey Dean and Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, 2004.
- [8] Apache Software Foundation, *Apache Kafka*, <https://kafka.apache.org/intro.html>, 2018
- [9] Jay Kreeps, Confluent, *Introducing Kafka Streams: Stream Processing Made Simple*, <https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/>, 2016
- [10] Cloudera, *Kudu User Guide* <https://www.cloudera.com/documentation/kudu/0-5-0/PDF/cloudera-kudu.pdf>, 2016