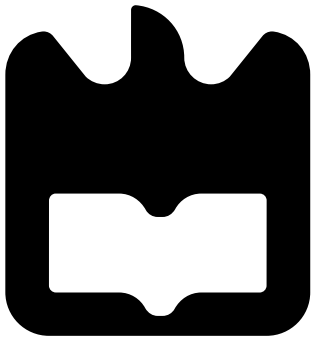
Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2014

Ricardo Jorge de Santiago Marques

Modelo de dados Big Data com suporte a SQL para performance management em redes de telecomunicações.

Big data model with SQL support for performance management in telecommunication networks



Ricardo Jorge de Santiago Marques

Modelo de dados Big Data com suporte a SQL para performance management em redes de telecomunicações

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Sistemas de Informação, realizada sob a orientação científica do Doutor José Luis Guimarães Oliveira, Professor do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro. O trabalho descrito nesta dissertação foi realizado na empresa Nokia Networks, sob a supervisão do Mestre Carlos Filipe Marques Almeida, Engenheiro de Desenvolvimento de Software da Nokia Networks.

Ricardo Jorge de Santiago Marques

Big data model with SQL support for performance management in telecommunication networks

Dissertation submitted to the University of Aveiro to fulfill the requirements to obtain a Master's degree in Information Systems, held under the scientific guidance of Professor José Luis Guimarães Oliveira, Department of Electronics, Telecommunications and Computer Science from the University of Aveiro. The work described in this thesis was carried out in the company Nokia Networks, under the supervision of Master Carlos Filipe Marques Almeida, Software Development Engineer at Nokia Networks.

o júri / the jury

presidente / president	**Professor Doutor Joaquim Manuel Henriques de Sousa Pinto, Professor Auxiliar, Universidade de Aveiro**
vogais / examiners committee	**Professor Doutor José Luis Guimares Oliveira, Professor Associado, Universidade de Aveiro** (Orientador)
	Doutor(a) Rui Pedro Sanches de Castro Lopes, Professor Coordenador, Instituto Politécnico de Bragana (Arguente Principal)

Resumo

Com o tempo, a informação mantida pelas aplicações tem vindo a crescer e espera-se um crescimento exponencialmente na área de banda larga móvel com o surgimento da LTE. Com este crescimento cada vez maior de dados gerados, surge a necessidade de os manter por um período maior de tempo e as RDBMS não respondem rápido o suficiente. Isto fez com que as empresas se se tenham afastado das RDBMS e em busca de outras alternativas.

As novas abordagens para o aumento do processamento de dados baseiam-se no desempenho, escalabilidade e robustez. O foco é sempre o processamento de grandes conjuntos de dados, tendo em mente que este conjunto de dados vai crescer e vai ser sempre necessário uma resposta rápida do sistema.

Como a maioria das vezes as RDBMS já fazem parte de um sistema implementado antes desta tendência de crescimento de dados, é necessário ter em mente que as novas abordagens têm que oferecer algumas soluções que facilitem a conversão do sistema. E uma das soluções que é necessário ter em mente é como um sistema pode entender a semântica SQL.

Abstract

Over time, the information kept by the applications has been growing and it is expected to grow exponantially in the Mobile Broadband area with the emerging of the LTE. This increasing growth of generated data and the need to keep it for a bigger period of time has been revealing that RDBMS are no longer responding fast enough. This has been moving companies away from the RDBMS and into other alternatives.

The new approaches for the increasing of data processing are based in the performance, scalability and robustness. The focus is always processing very large data sets, keeping in mind that this data sets will grow and it will always be needed a fast response from the system.

Since most of the times the RDBMS are already part of a system implemented before this trend of growing data, it is necessary to keep in mind that the new approaches have to offer some solutions that facilitate the conversion of the system. And one of the solutions that is necessary to keep in mind is how a system can understand SQL semantic.

Contents

List of Figures

List of Tables

List of Acronyms

ASCII American Standard Code for Information Interchange

RDBM Relational Database Management System

ORM Object-relational mapping

JDBC Java Database Connectivity

ODBC Open Database Connectivity

XML EXtensible Markup Language

JSON JavaScript Object Notation

BSON Binary JavaScript Object Notation

CAP Consistency, Availability and Partition tolerance

ACID Atomicity, Consistency, Isolation, Durability

API Application Programming Interface

SQL Structured Query Language

GFS Google Distributed File System

HDFS Hadoop Distributed File System

DAG Directed Acyclic Graph

HQL Hive Query Language

CLI Command Line Interface

UDF User Defined Functions

UDAF User Defined Aggregate Function

MPP Massively Parallel Processing

RDD Resilient Distributed Datasets

PDE Partial DAG Execution

JVM Java Virtual Machine

NE Network Element

SAN Storage Area Network

RAC Real Application Clusters

KPI Key Performance Indicator

ETL Extract, Transform and Load

RCFile Record Columnar file

CSV Comma-separated values

JBOD Just a bunch of disks

REST Representational state transfer

Chapter 1

Introduction

1.1 Context

Over the years, internet has become a place where massive amount of data are generated every day. This continuous growing of data and the need to organize and search it have been revealing some lack of performance in the traditional Relational Database Management System (RDBM) implemented so far.

RDBMs are massively used in the world, but the schema-rigidity and their performance for big volumes of data, are becoming a bottleneck. The schema-rigid forces the correct design of the model at the starting point of a project. However, if over time the data stream is a volatile schema, the schema-rigidity can be a problem. If the structure of the data is changed, this can force a migration of data from the RDBM, and possible data losses, at a great cost. In terms of data volume, it is a more complex decision for an organization, especially if the product is already implemented and needs to evolve. The knowledge acquired in the RDBMs is very large and not only are the applications based on it, but also the mindset of people that uses them. So, organizations are trying to reduce costs by shifting to other methods by finding Structured Query Language (SQL) based systems to reduce the impact. RDBMs are well integrated in the market and they are a very good, or even the best, solution for common scenarios. However, for huge amount of data these systems can perform very poorly.

Changing the way on how to keep and interact with data is becoming more and more frequent in the database world, not only because it is a trend in the IT world, but also because now it is possible to save more data and for a longer period of time at a lower price. With these possibilities, the organizations are exploring their data in a way that was not possible a few years ago and can analyze, transform or predict future results based on the amount of data kept. This type of possibilities can give organizations a greater leverage in deciding for future strategies in the market.

For this amount of data, a catch-all term is used to identify it - Big Data. This term defines data that exceed the processing capacities of conventional database systems. Big data have become a viable cost-effective approach that can process the volume, velocity and variability of massive data.

Telecommunication companies have infrastructures, with systems such as 2G, 3G or even 4G LTE, where the activity involved in the daily operations and future network planning require data on which decisions are made. These data refer to the load carried by the network and the grade of a service. In order to produce these data, performance measurements

are executed in the Network Elements (NEs), collecting data that is then transferred to an external system, e.g. a Monitoring System, for further evaluation. In a 2G and 3G system, the size of data generated by the NEs is considerable, but sustainable. With the introduction of 4G LTE systems, and anticipating the future 5G systems, the data growth has been exponential, making storage space and RDBMs performance a problem. Facing this problem, telecommunication companies have turned to Big Data systems trying to find alternatives for the data growth.

1.2 Objectives

Facing the increase of data to store and process, it is necessary to analyze solutions that allow, in a viable and cost-effective way, to process, store and analyze it. For this work, some of the solutions used in Big Data systems were studied. Different solutions to store and analyze the data were studied. The key objectives are namely:

- studying different technologies/frameworks, taking into consideration the support to SQL and its scalability.
- evaluating the data load from the NEs.
- evaluating storage solutions.
- evaluating SQL support and performance over a typical scenario.
- defining specifications of a Big Data system and resources needed.

1.3 Structure of the document

This document is structured in 3 chapters:

- Big data ecosystem - in this chapter are described several types of systems and different practices used on each system.
- Evaluation scenario - in this chapter, a test environment is specified and the results of each evaluation are presented.
- Conclusion and future work - in this chapter is summarized some conclusions and also point out possible future work.

Chapter 2

Big data ecosystem

2.1 Introduction

The term big data has been used to describe data that exceed the processing capacities of conventional database systems. In these data lie valuable patterns and hidden information, which can be analyzed to create valuable information to an organization. The analytics can reveal insights previously hidden due to the cost to process, being only possible by exploration and experimentation.

Most of the organizations have their data in multiple applications and different formats. The data can be collected by smart phones, social networks, web server logs, traffic flow sensors, satellite imagery, broadcast audio streams, banking transactions, MP3s music, Global Positioning System (GPS) trails and so on. These types of collected data contain structured and unstructured data that can be analyzed, explored and experimented for patterns.

There are some aspects that need to be characterized to better understand and clarify the nature of Big Data. These aspects have been identified as the three Vs - **V**ariety, **V**elocity and **V**olume (figure 2.1) [1] [2].

Volume

This is one of the main attractions of big data. The ability to process large amounts of information that, in conventional relational databases infrastructures, could not be handled[1]. It is common to reach Terabytes and Petabytes in the storage system and sometimes the same data is re-evaluated with multiple angles searching for more value in the data. The options to process the large amounts of data are parallel processing architectures such as data warehouses or databases, for instance Greenplum, or platform distributed computing system - Apache Hadoop based solutions[2]. This choice can be mostly defined by the Variety of the data.

Velocity

With the growth of social media, the way we see information has changed. There was a time when data with some hours or days were considered recent. Organizations have changed their definition of recent and have been moving towards the near real-time applications. The terminology for such fast-moving data tends to either streaming data, or complex event processing[2].

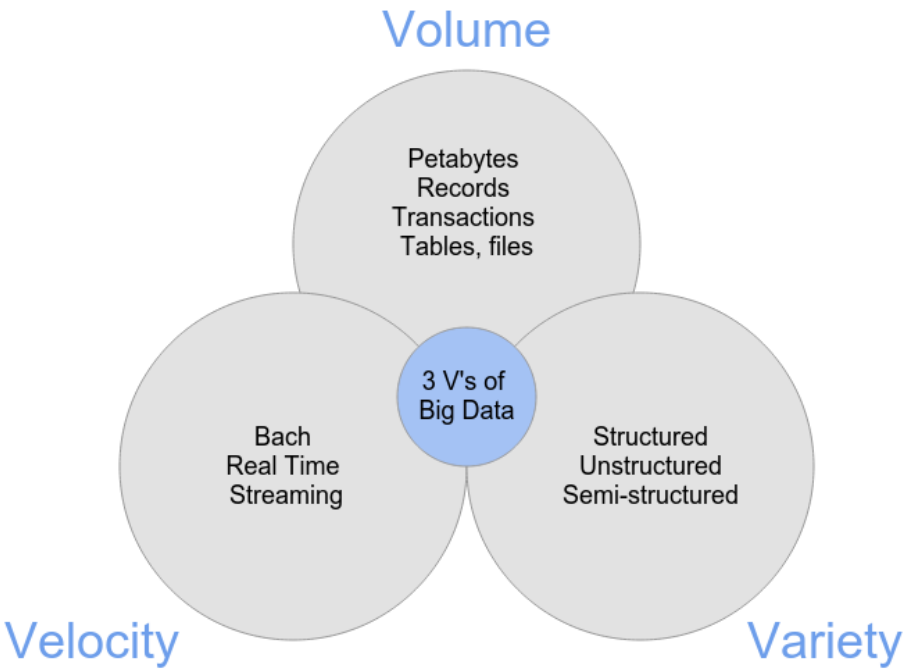


Figure 2.1: The big data 3Vs

Streaming processing has two main reasons to be considered. One main reason is when the input data are too fast to be entirely stored and need to be analyzed as the data stream flows[1]. The second main reason is when the application sends immediate response to the data - mobile applications or on line gaming, for example[1]. However, velocity is not only for the data input but also for the output - as the near real-time systems previously mentioned. This need for speed has driven developers to find new alternatives in databases, being part of an umbrella category known as NoSQL.

Variety

In a Big Data system it is common to gather data from sources with diverse types. These data does not always fit the common relational structures, since it may be text, image data, or raw feed directly from a sensor. Data are messy and the applications can not deal easily with all the types of data. One of the main reasons for the use of Big Data systems is the variety of data and processes to analyze unstructured data, and be able to extract an ordered meaning of it[2].

In common applications, the process of moving data from sources to a structured process to be interpreted by an application may imply data loss. Throwing data away can mean the loss of valuable information, and if the source is lost there is no turning back. "*This underlines a principle of Big Data: when you can, keep it*"[2].

Since data sources are diverse, relational databases and their static schemas nature have a big disadvantage. To create ordered meaning of unstructured or semi-structured data, it is necessary to have a system that can evolve along the way, following the detection and extraction of more valuable meaning of the data over time. To guaranty this type of flexibility, it is

necessary to encounter a solution on how to allow organization of data but without requiring an exact schema of data before storing, the NoSQL databases meet these requirements.

2.2 NoSQL

NoSQL is a term that was originated in a meet-up focused on the new developments made in BigTable, by Google, and Dynamo, by Amazon, and all projects that they inspired. However, the term NoSQL caught out like wildfire and up to now, it is used to define databases that do not use SQL. Some have implemented their query languages, but none fit in the notion of standard SQL. An important characteristic of a NoSQL database is that it is generally open-source, even if sometimes the term is applied to closed-source projects.

The NoSQL phenomenon has emerged from the fact that databases have to run on clusters[3]. This has an effect on their data model as well as their approach to consistency. Using a database in a cluster environment clashes with the Atomicity, Consistency, Isolation, Durability (ACID) transactions used in the relational databases. So NoSQL databases offer some solutions for consistency and distribution. However, not all NoSQL databases are focused on running in clusters. The Graph databases are similar to relational databases, but offer a different model focused on data with complex relationships.

When dealing with unstructured data, the NoSQL databases offer a schemeless approach, which allows to freely adding more records without the need to define or change the structure of data. This can offer an improvement in the productivity of an application development by using a more convenient data interaction style.

The ACID transactions are well known in relational databases. An ACID transaction allows the update of any rows on any tables in a single transaction giving atomicity. This operation can only succeeds or fails entirely, and concurrent operations are isolated from each other so they cannot see a particular update.

In NoSQL databases, it is often said that they sacrifice consistency and do not support ACID transactions. Aggregate-oriented databases have the approach of atomic manipulation in a single aggregate at a time, meaning that to manipulate multiple aggregates in an atomic way it is necessary to add application code. Graph and aggregate-ignorant database usually support ACID transactions in a similar way to relational database.

In NoSQL it is always common the reference to the Consistency, Availability and Partition tolerance (CAP) Theorem. This theorem was proposed by Eric Brewer(may also be known as Brewer's Conjecture) and proven by Seth Gilbert and Nancy Lynch[3]. CAP are the tree properties of the theorem and represent:

· **Consistency** - all nodes see the same data at the same time.

· **Availability** - a guarantee that every request receives a response about whether it was successful or has failed.

· **Partition tolerance** - the system continues to operate despite arbitrary message loss or failure from part of the system.

However, the CAP theorem defines that in a distributed system a database can only offer two out of the three properties (figure 2.2). In other words, you can create a system that is

consistent and partition tolerant, a system that is available and partition tolerant or a system that is consistent and available, but not all the three properties[3].

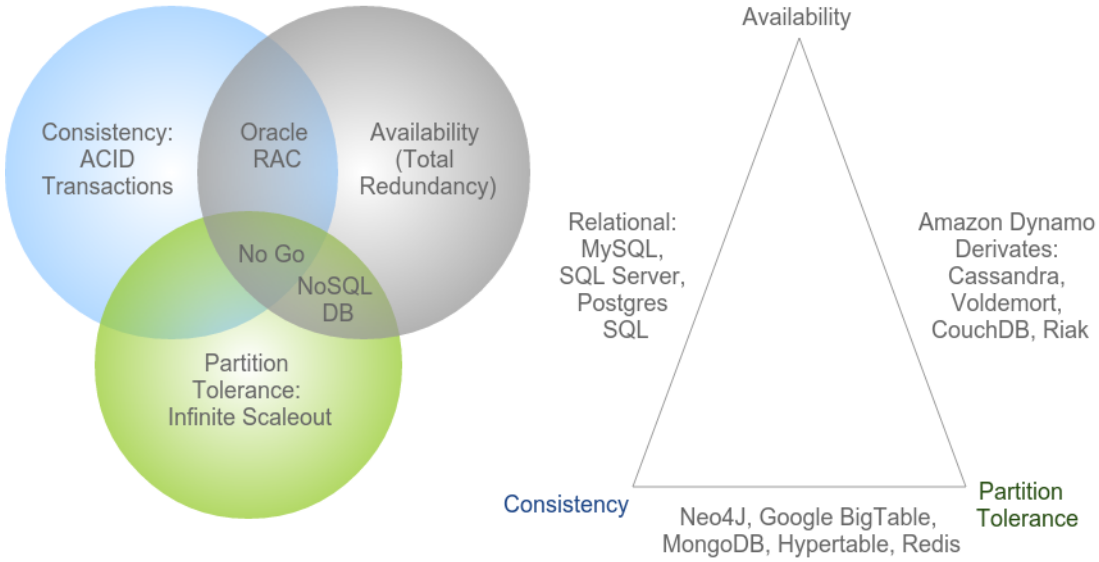


Figure 2.2: The CAP Theorem

As seen, the CAP Theorem is very important on the decision of which distributed system to use. Based on the requirements of the system it is necessary to understand which property is possible to give up. The distributed databases must be partition tolerant, so that the choice is always between availability and consistency, if the choice is availability there is the possibility of *eventual* consistency. In *eventual* consistency the node is always available to requests but, the data modification is propagated in the background to other nodes. At any time the system may be inconsistent, but the data are still accurate.

NoSQL Database Types

For a better understanding of NoSQL data stores, it is necessary to compare their features with RDBMs, understand what is missing and what is the impact in the application architecture. In the next sections it will be discussed the consistency, transactions, query features, structure of the data, and scaling for each type presented.

Key-Value

Key-value database are the simplest NoSQL data stores based on a simple hash table. In this type of data stores it is the responsibility of the application to know which type of data is added. The application manages the key and the value without the key-value store caring or knowing what is inside(figure 2.3). This type of data stores always use primary-key access which gives a great performance and scalability.

In key-value database, the consistency is only applicable in a single key because the only operations are 'get', 'put' or 'delete' a single key. There is the possibility of an optimistic write, but the implementation would be expensive, because of the validation of changes in

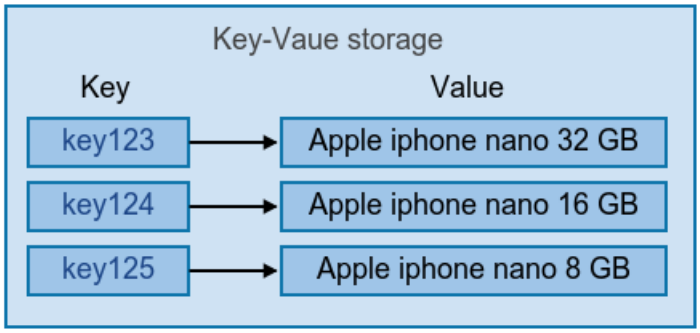


Figure 2.3: NoSQL database type Key-value store

the value from the data store. The data can be replicated in several nodes and the reads are managed by the validation of the most updated writes.

For transactions, the key-value database has different specifications and, in general, there is no guarantee on the writes. One of the concepts is the use of quorums.[1]

The query is by key and quering by value is not supported. For searching by value, it is necessary that the application reads the value and process it itself. This type of database, usually, does not give a list of keys, so it is necessary to give a thought about the pattern/key to use on key-value database[3].

The structure of data for key-value database is irrelevant, the value can be a blob, text, JavaScript Object Notation (JSON), EXtensible Markup Language (XML) or any other type[3]. So, it is the responsibility of the application to know what type of data is in the value.

If there is the need for scaling, the key-value stores can scale by using sharding[2]. The sharding for each node could be determined by the value of the key, but it could introduce some problems. If one of the nodes goes down, it could implye data loss or unavailable and no new writes would be done for keys defined in sharding for that node.

Document

Document databases stores documents in the value part of the key-value store, where the value is examinable (figure 2.4). The type of files can be XML, JSON, Binary JavaScript Object Notation (BSON) and many more self-described type of files[3]. The data schema can differ across documents and belong to the same collection without a rigid schema such as in RDBMs. In documents, if a given attribute is not found, it is assumed that it was not set or relevant, but it can create new attributes without the need to define it or change the existing ones.

Some document databases allow configuration of replica sets choosing to wait for the writes in a number of slaves or all. In this case all the writes need to be propagated before it returns success allowing consistency of data but affecting the performance.

The traditional transactions are generally not available in NoSQL solutions. In document

[1]**Write quorum** is expressed as W>N/2, meaning the number of nodes participating in the write (W). Must be more than a half the number of nodes involved in replication (N)[3].

[2]**Sharding:** puts different data on separate nodes, each of which does its own reads and writes.

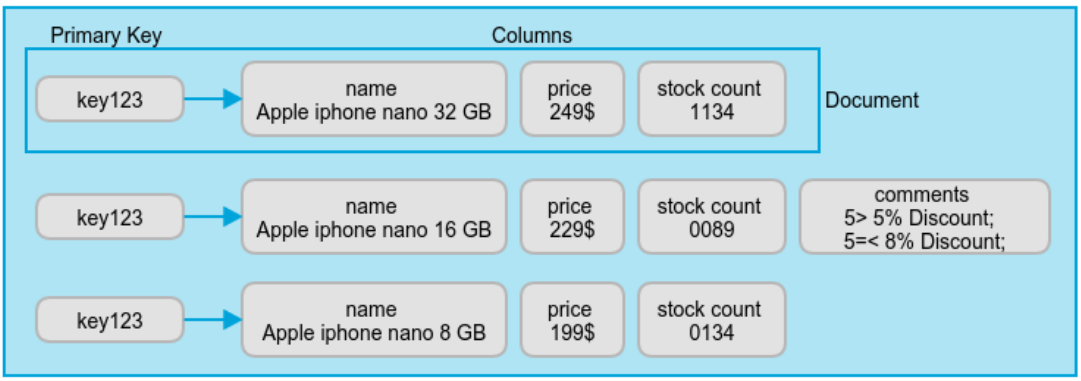


Figure 2.4: Document database store

databases, the transactions are made at single-document level and called 'atomic transactions'. Transactions involving more than one operation are not supported.

To improve availability in document databases, it can be done by replicating data using master-slave setup[3]. This type of setup allows access to data, even if the primary node is down, because data is available in multiple nodes. Some document databases use replica sets where two or more nodes participate in an asynchronous master-slave replication. All the requests go through the master node, but if the master goes down, the remaining nodes in the replica set vote among themselves to elect the new master. The new requests will go through the new master and the slaves will start receiving data from the new master. If the node that failed gets back online, it will join the list of slaves and catch up with other nodes to get the latest information. It is also possible to scale, for example, to archive heavy-reads. It would be a horizontal scaling by adding more read slaves, the addition of nodes can be performed during time the read load increases. The addition of nodes can be performed without the need of application downtime, because of the synchronization made by the slaves in the replica set.

One of the advantages of document databases, when compared to key-value databases, is that we can query the data inside the value. With the feature of querying inside the document, the document databases get closer to the RDBMs query model. The document databases could be closer to the RDBMs, but the query language is not SQL, some use query expressed via JSON to search the database.

Column-Family

Column-family databases allow storing data in a key-value type and the values are grouped in multiple column families. Data in column-family databases is stored in rows with many columns associated to a key. The common data stored in column-family database are groups of related data that are usually accessed together as shown in figure 2.5.

In some column-family databases, the column consists of a name-value pair where the name behaves as key for searches. The key-value pair is a single column and, when stored, it receives a timestamp for data control. It will allow the management of data expiration, resolve data conflicts, stale data, among others[3]. In RDBMs, the key identifies the row and the row consists in multiple columns. The column-family databases have the same principle, but the rows can have different columns during time, without the need to add the columns to

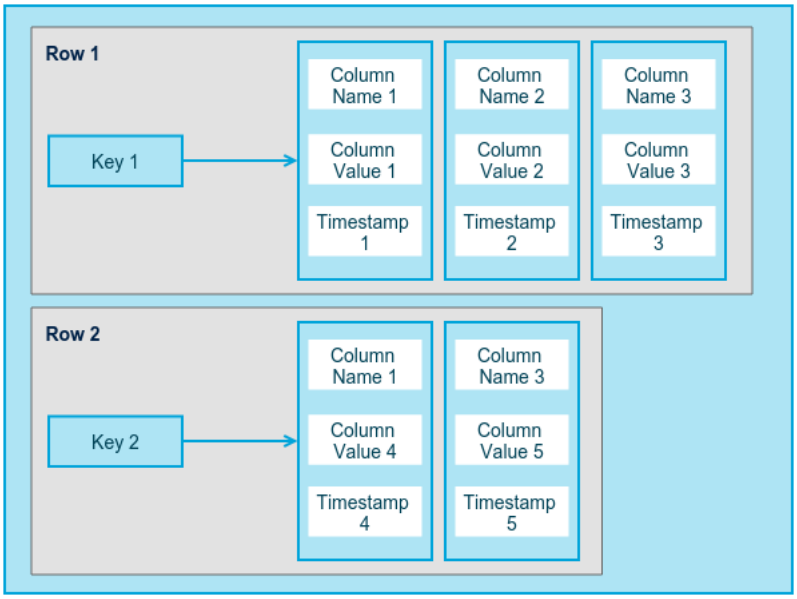


Figure 2.5: Column family store

other rows. In column-family databases there are other type of columns, the super columns. The super columns can be defined as a container of columns being a map to other columns; this is particulary good when the relation of data needs to be established.

Some column-families use quorums to eventually achieve consistency, but their focus are on availability and partition tolerance. Others guaranty consistency and partition tolerance, but lack in availability.

Transactions in column-family databases are not made in the common sense, meaning that it is not possible to have multiple writes and decide if we want to commit or roll-back the changes. In some column-family databases the writes are made in an atomic way, being only possible to write at the row level and will either succeed or fail.

As previously mentioned, the availability in column-family database relies on the choice of the vendor. To improve the availability some column-family databases use quorums formulas to improve availability. The availability improvement in a cluster will decrease the consistency of data.

The design of column-family databases is very important, because the query features for this type of databases is very poor. In this type of databases it is very important to optimize the columns and column families for the data access.

The scaling process depends on the vendor. Some offer a solution to scale with no single point of failure, presenting a solution with no master node, and by adding more nodes, it will improve the cluster capacity and availability. Others offer consistency and capacity by a master node and *RegionServers*. By adding more nodes, it will improve the capacity of the cluster keeping the data consistent.

Graph

Graph databases allow the representation of relationships between entities. Entities, also known as nodes, contain entity properties and they are similar to an object instance in applications. The relationships are known as edges and contain properties and direction significance, as figure 2.6 shows.

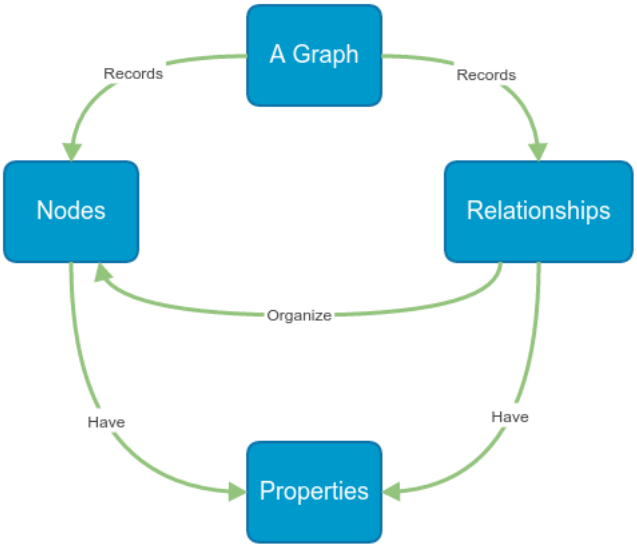


Figure 2.6: Graph database

The graph databases allow relationships as in RDBMs, but if there is a change in a relationship the scheme change will not be necessary. One of the key features of this type of database is the fast traversing of the joins or relationships, and that the relations are not calculated at query time - they are persisted as a relationship. Relationships are first-class citizens and not only have types, but can also have start node, end node, and properties of their own allowing to add intelligence to the relationships[3].

One of the key features of graph databases is the relationships and, therefore, they operate on connected nodes. Because of this type of feature, most graph databases do not support the distribution of nodes on different servers. Graph databases ensure consistency and transactions, but in some vendors they can also be fully ACID compliant. The start or end node does not need to exist and it can only be deleted if no relationship exists.

As previously said, the graph databases can ensure transactions, but it is necessary to keep in mind that the workflow of the transactions can differ from vendor to vendor. So, even in this, there are some characteristics similar to RDBMs. The standard way of RDBMs managing transactions can not be applied to graph databases.

For availability, some vendors offer the solution of replication on nodes to increase the availability on reads and writes. This solution implies a master and slaves to manage replication and management of data. Other vendors provide distributed storage of the nodes.

Querying a graph database requires a domain-specific language for traversing graphs. Graphs are really powerful on traversing at any depth of the graph and, the node search can be improved with the creation of indexes.

Since graph databases are relation oriented, it is difficult to scale them by sharding, but some techniques are used for this type of databases. When the data can entirely fit in RAM, this will give a better performance for the database and for traversing the nodes throw the cluster. Some techniques go through adding slaves, with read-only access, to the data or, through all the writes passing on the master. This pattern is defined by writing once and reading from many. However, it can be applied when the data is large enough to fit in RAM but small enough to be replicated. The sharding technique is only used when the data is too big for replication.

2.3 Streaming data

With the size of data that modern environments deal with, it is required a processing computation on streaming data. To process the data and transform it to the defined structure, it is necessary a distributed real-time computation system which will deal with the data stream. The system must scale and perform Extract, Transform and Load (ETL) over a big data stream.

Storm

Storm is a real-time distributed computation system and it is focused in scalability, resilience (fault-tolerance), extensible, efficient and easy to administer[4]. Storm uses a Nimbus/Supervisors architecture, which is the same definition of master and slaves:

- Nimbus daemon - this is the master node and it is responsible for assigning tasks to the worker nodes, monitoring cluster failures and deploy the topologies (the code developed) around the clusters.
- Supervisors daemon - this is the worker node and listens for work assigned by the Nimbus. Each worker executes a subset of a topology.

Storm uses Zookeeper to coordinate between Nimbus and Supervisors, as represented in figure 2.7, and their state is hold on disk or in Zookeeper.

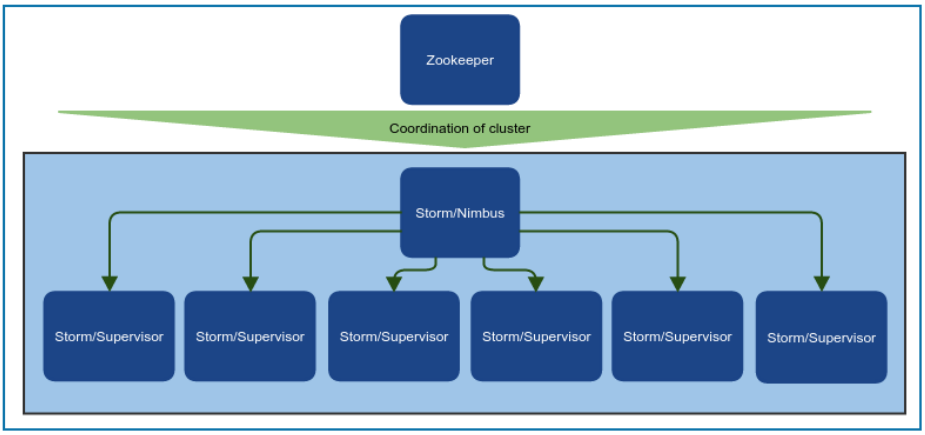


Figure 2.7: Data stream cluster architecture

Zookeeper, a project inside the Hadoop ecosystem (section 2.4), is a centralized service used for maintaining configuration information, naming, providing distributed synchronization, and providing group service. The main focus is to enable highly reliable distributed coordination.

By using Storm, tuples containing objects of any type, can be manipulated and transformed. Storm uses three abstractions (figure 2.8):

- spout - is the source of streams in computations and typically reads from queuing brokers.
- bolt - most of the logic of a computation goes into bolts, and bolts processes input, from the streams, and produces a number of new outputs.
- topology - is the definition of the network made of spouts and bolts, and can run indefinitely when deployed.

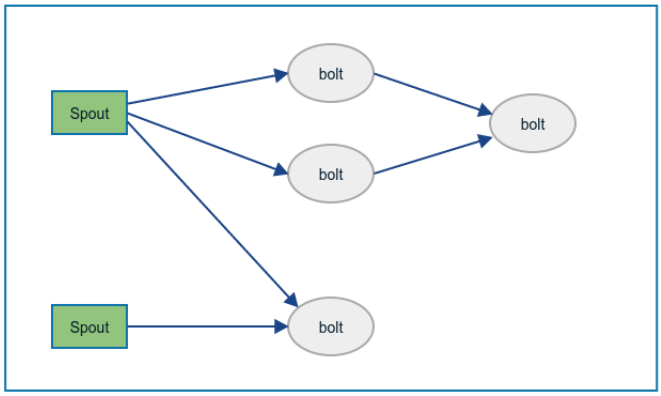


Figure 2.8: Topology example

Storm offers the flexibility to execute ETL and the connection with several sources, such as, a real-time system or a off-line reporting system using common RDBMs or Hadoop Distributed File System (HDFS).

2.4 Storage

Hadoop

Google Distributed File System (GFS) and the Map-Reduce programming model have had enormous impact in parallel computation. Doug Cutting and Mike Cafarella started to build a framework called Hadoop based on GFS and Map-Reduce papers. The framework is mostly known by the HDFS and the Map-Reduce features, and it is used to run applications on large clusters of commodity hardware.

Map-Reduce

Communities, such as the High Performance Computing (HPC) and Grid Computing, have been doing large-scale data processing for years. The large-scale data processing was

done by APIs like Message Passing Interface (MPI), and the approach was to distribute work across clusters of machines, which access a shared file system, hosted by a Storage Area Network (SAN). This type of approach works only for computer-intensive jobs. When there is the need to access large data volume the network bandwidth is a bottleneck and compute nodes become idle.

The Map-Reduce breaks the processing into two phases, the map phase and the reduce phase. The map function is processed by the Map-Reduce framework before being sent to the reduce function, and each phase has a key-value pair as input and output (figure 2.9)[5].

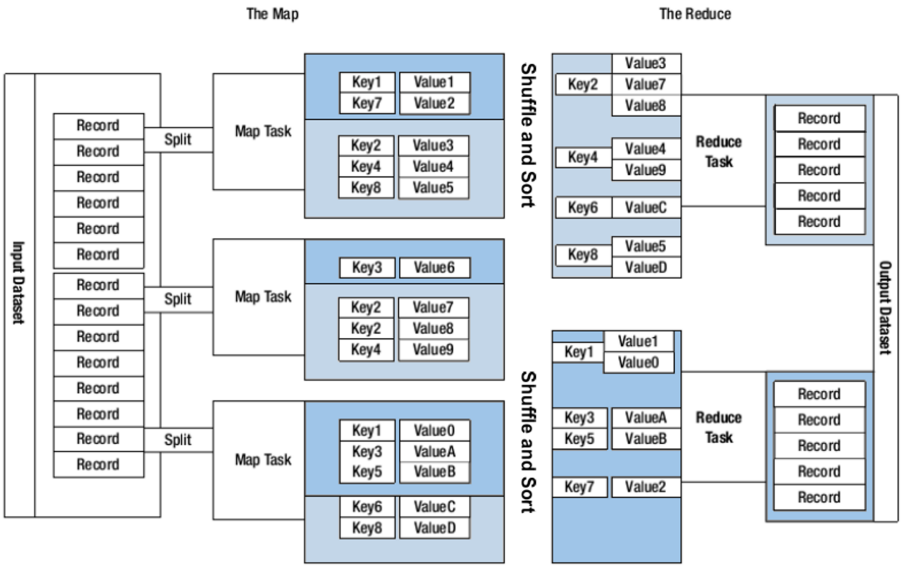


Figure 2.9: Map Reduce framework architecture

This functions and key-value structures are defined by the programmer, and the processing sorts and groups the key-value pairs by key. The framework can detect failure for each phase and, detecting the failure on map or reduce tasks, it will reschedules the jobs on healthier machines. This type of reschedule can be performed because Map-Reduce is a share-noting architecture, meaning that the tasks have no dependencies on one another[5].

A Map-Reduce job is a unit of work that we want to be performed over a set of data, and run by dividing it into tasks of map and reduce. All the jobs are coordinated by a jobtracker, and it will schedule tasks to run on tasktrackers. The function of a trasktracker is to run a task and send progress reports to the jobtracker, while the jobtracker keeps a record of the overall progress of each job. Since the tasks can fail, the jobtracker can reschedule it on a different tasktracker.

As seen in the description of the workflow (figure 2.10) Map-Reduce has a master and slaves which collaborate on getting the work done. The definition of master and slave is made by the configuration file and therefore, they can know about each other. The master has an algorithm on how to assign the next portion of the workfigure[5]. This division is called 'split', and it is controlled by the programmer. By default, the input file is split into chunks of about 64 MB in size, preserving the complete lines of the input file. The data resides in HDFS,

which is unlimited and linearly scalable. So, it can grow by adding new servers as needed. This type of scalability solves the problems of central file server with limited capacity.

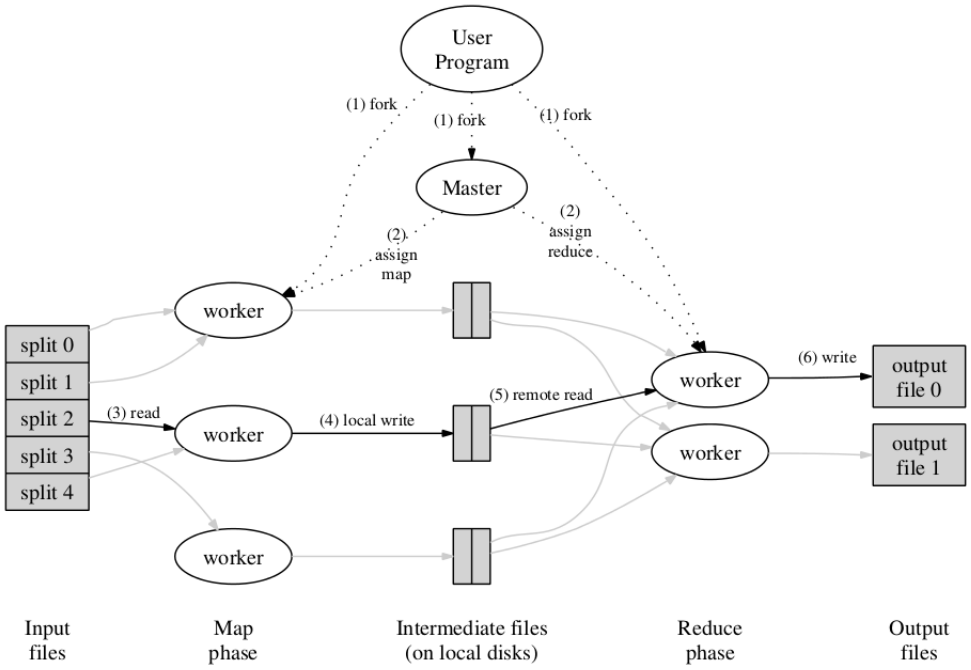


Figure 2.10: Map Reduce execution diagram

The Map-Reduce framework never updates the data. It writes a new output, avoiding update lockups. Moreover, to avoid multiple process writing in the same file, each reducer writes its own output to the HDFS directory designated for the job. Each reducer has its own ID, and the output file is named using that ID to avoid any problem with input or output files in the directory. The map task should run in nodes where the HDFS data is, to improve optimization - called data locality optimization, because it does not use cluster bandwidth[5]. The Map output is written to the disk, since it is an intermediate output and it can be discarded when the job from the reducer is complete.

Sometimes, when looking for a map slot on a node inside the rack, it is necessary to schedule the task outside the rack. An off-rack node is used, which results in an inter-rack network transfer[5]. This is only possible, because Map-Reduce is rack-aware, it knows where the data and tasktracker are over IP (figure 2.11). This awareness allows assigning computation to the server that has data locally. If this is not possible the Map-Reduce will try to select the server closest to the data, reducing the number of hops in the network traffic.

Hadoop Distributed Filesystem

Over the years data have grown to the point of being too big to store in a single physical machine. This created the necessity to partition data across several machines and filesystems that manage storage across a network of machines - distributed filesystems. These type

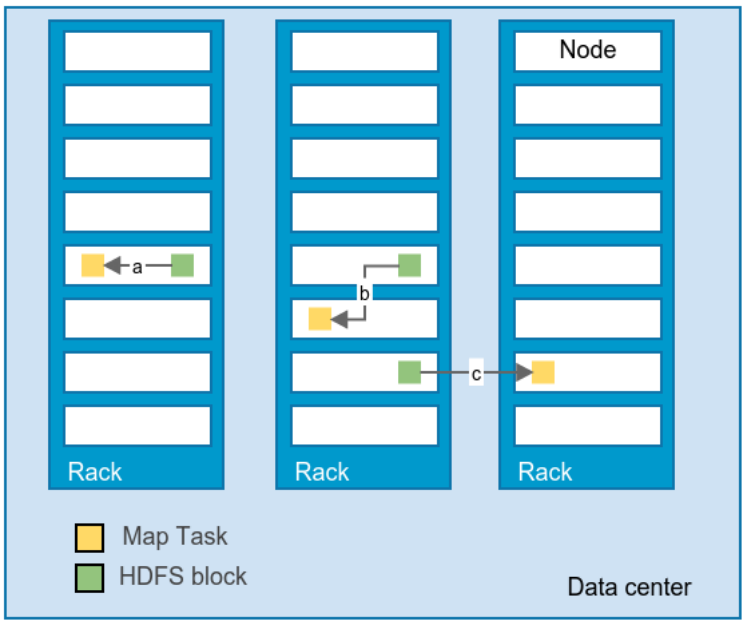


Figure 2.11: Map Reduce example of rack awareness

of filesystems are network-based and more complex than normal filesystems. They create challenges such as tolerance to node failure without suffering data loss. Creating a distributed filesystem over server machines can be costly. Therefore, it is necessary to reduce costs by using commodity hardware.

Hadoop comes with a distributed filesystem called HDFS based on GFS, and it is designed to store very large files with streaming data access patterns. Large files mean files with hundreds of megabytes, gigabytes, or terabytes in size(some clusters today are storing petabytes). Streaming data access in HDFS is based on the data processing pattern write-once, read-many-times[6]. Since the reading is very important, the dataset generated or copied from source is analyzed many times and it will involve a large portion, if not all, of it making very important the time to read all the dataset. HDFS does not need highly reliable hardware to run, since it is design to run on clusters of commodity hardware. Even if a node fails, HDFS is designed to carry on working without interruption.

There are some problems where HDFS does not work so well. When low-latency accessing data is required, HDFS will not work well because it is optimized for delivering a high throughput of data. The limit to the number of files in a filesystem is governed by the amount of memory in the master. So, if an application generates a big amount of small files HDFS may not be the best fit. The HDFS writes are made at the end of the file, and there is no support for multiple writers, or modifications at arbitrary offset in the file[6]. So, if an application needs to perform multiple writes or arbitrary files modification, HDFS it is not the best solution. The information of each file, directory, and block is stored in memory taking about 150 bytes, which can be possible for millions of files but not for billions because of the limits of current hardware.

As previously mentioned, HDFS cluster consists of one master node and many worker nodes. The master is named of Name Node and workers are named of Data Node (figure 2.12).

The Data Nodes are the ones that store the data, and the Name Node is in charge of file system operations. Without the Name Node the cluster will be inoperable, and no one can read or write data.

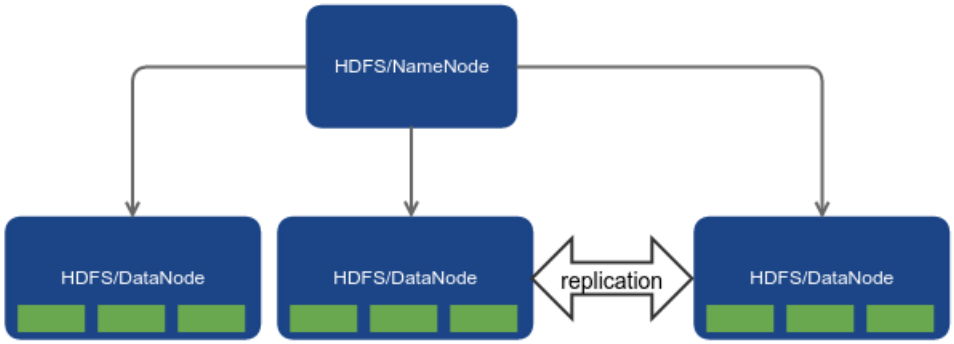


Figure 2.12: HDFS architecture

Since commodity servers will fail, a solution to keep the data resilient is to maintain multiple copies of data across nodes. Keeping several copies, even if a node fails, will maintain data accessible and the system on-line. Data Node store the data, retrieve blocks, and periodically report a list of blocks that they store to the Name Node. The Name Node is a Single Point of Failure, and if it fails, all the files on the filesystem would be lost, and there would be no way of knowing how to reconstruct the files stored in the Data Nodes. For this problem, HDFS provides two mechanisms to keep it resilient to failure. One of the mechanism, is to back up the files that make a persistent state of the filesystem. It can be configured to write, synchronously and atomically, its persistent state to multiple filesystems. The second mechanism, is to run a secondary Name Node. This secondary Name Node only merges periodically the namespace image with the edit log, preventing the edit log from becoming too large. This Name Node has the same hardware characteristics as the primary, and it will be ready for use in case of the fail from the primary.

2.5 Querying Big Data

Hadoop ecosystem solves the cost-effective problem of working with a large data set. It introduces a programming model and a distributed file system, splitting work and data over a cluster of commodity nodes. The Hadoop ecosystem can scale horizontally, making it possible to create bigger data set over time. However the challenge is, how to query all the data from the sources available?

A challenge for companies using, or planning to use, Big Data systems is how to query their data on several infrastructures. The number of users knowing SQL is large, and applications were developed using this knowledge. So, there is a necessity to migrate to Big Data systems with the smaller impact in applications, becoming very important to use query engines that can query on Big Data systems, and with characteristics of being SQL like. This would reduce the cost in a company, using the knowledge already achieved by SQL users, and the time to learn new methods or languages would be unnecessary. The migration of applications would be minimum, creating the possibility to companies to be in the market with new features and

a new infrastructure.

2.5.1 HIVE

Hive was created by the Facebook Data Infrastruture Team in January 2007 and available as open source in August 2008. The main reason for the development of Hive was because of the difficulty of end users on using map-reduce jobs in Hadoop, since it was not such an expressive language as SQL, and it brought that data closer to the users[7]. In Facebook, Hive is used from simple summarization jobs to business intelligence, machine learning and product features. Hive structures data into database concepts like tables, columns, rows, and partitions, but it is possible for users to extend to other types and functions[7].

Hive is separated in some components, each one with a responsibility. In terms of system architecture, Hive is composed by:

- **Metastore** - this component stores the system catalog and metadata of the tables, partitions, columns and types, and more. To interact with this information Hive uses a thrift interface to connect to an RDBM using a Object-relational mapping (ORM) (DataNucleus) to convert object representation into relational schema and vice versa. This is a critical component for Hive and, without it, it will not be possible to execute any query on HDFS. So, for safety measures, it is advised to keep a regular back up of the Metastore or a replicated server to provide the necessary availability for a production environment.

- **Driver** - this component manages the lifecycle of Hive Query Language (HQL) statements, maintains the session handle and any session statistics.

- **Query Compiler** - this component is responsible for the compilation of HQL statements to the Directed Acyclic Graph (DAG) of map-reduce jobs. The metadata in the Metastore is used to generate the execution plan and processes HQL statements in four steps:

  - Parse - uses Antlr to generate an abstract syntax tree for the query
  - Type checking and Semantic Analysis - in this step, the data from the Metastore is used to build a logical plan and check the compatibilities in expressions flagging semantic errors.
  - Optimization - this step consists of a chain of transformations such that the operator DAG is passed as input to the next transformation. The normal transformations made are column pruning, predicate pushdown, partition pruning and map side joins.
  - Generation of the physical plan - after the optimization step, the logical plan is split into multiple map-reduce and HDFS tasks.

- **Execution Engine** - this component executes the tasks generated by the Query Compiler in the order of their dependencies. The executions are made using Hadoop and Map-Reduce jobs, and the final results are stored in a temporary location and returned at the end of the execution plan ends.

- **Hive Server** - this component provides a thrift interface and Java Database Connectivity (JDBC)/Open Database Connectivity (ODBC) server, allowing external applications to integrate with Hive.

- **Clients Components** - this component provides Command Line Interface (CLI), the Web UI and the JDBC/ODBC drivers.

- **Interfaces** - this component contains the SerDe and ObjectInspector interfaces, the UDF and User Defined Aggregate Function (UDAF) interfaces for user custom functions.

Since it is very similar to SQL language it is easily understood by users familiar with SQL. Hive stores data in tables, where each table has a number of rows, each row has a specific number of columns and each column has a primitive or complex type.

Numeric Types

Type	Description
TINYINT	1-byte signed integer, from -128 to 127
SMALLINT	2-byte signed integer, from -32,768 to 32,767
INT	4-byte signed integer, from -2,147,483,648 to 2,147,483,647
BIGINT	8-byte signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
FLOAT	4-byte single precision floating point number
DOUBLE	8-byte double precision floating point number
DECIMAL	Based in Java's BigDecimal which is used for representing immutable arbitrary precision decimal numbers in Java.

Table 2.1: Hive Numeric Types[8]

Data/Time Types

Type	Description
TIMESTAMP	Supports traditional UNIX timestamp with optional nanosecond precision. Supported conversions: Integer numeric types: Interpreted as UNIX timestamp in seconds Floating point numeric types: Interpreted as UNIX timestamp in seconds with decimal precision Strings: JDBC compliant java.sql.Timestamp format "YYYY-MM-DD HH:MM:SS.ffffffff" (9 decimal place precision)
DATE	Describe a particular year/month/day, in the YYYY-MM-DD format.

Table 2.2: Hive Data/Time Types[8]

String Types

Type	Description
STRING	Can be expressed either with single quotes (') or double quotes ("). Hive uses C-style escaping within the strings.
VARCHAR	Are created with a length specifier (between 1 and 65355), which defines the maximum number of characters allowed in the character string.
CHAR	Similar to Varchar but they are fixed-length. The maximum length is fixed to 255.

Table 2.3: Hive String Types[8]

Misc Types

Type	Description
BOOLEAN	true or false
BINARY	Only available starting with Hive 0.8.0

Table 2.4: Hive Misc Types[8]

Complex Types

Type	Description
ARRAYS	ARRAY<data_type>
MAPS	MAP<primitive_type, data_type>
STRUCTS	STRUCT<col_name : data_type [COMMENT col_comment], ...>
UNION	Only available starting with Hive 0.7.0. UNIONTYPE<data_type, data_type, ...>

Table 2.5: Hive Complex Types[8]

Queering complex types can be made by using the '.' operator for fields and '[ ]' for values in associative arrays. Next is shown a simple example of the syntax that can be used with complex types:

```
CREATE TABLE t1(
  st string,
  f1 float,
  li list<map<string,struct<p1:int, p2:int>>>
)
```

In the previous example, to obtain the first element of the *li* list, it is used the syntax *t1.li[0]* on the query. The *t1.li[0]['key']* return the associated *struct* and *t1.li[0]['key'].p2* gives the p2 field value.

These type of tables are serialized and deserialized using the defaut serializeres/desirializeres already present in Hive. But for more flexibility, Hive allows the insertion of data into tables without having to transform it. This can be achieved by developing an implementation of the interface of SerDe in Java, so any arbitrary data format and types can be plugged into Hive. The next example shows the syntax needed to use a SerDe implementation:

```
add jar /jars/myformat.jar
CREATE TABLE t2
ROW FORMAT SERDE 'com.myformat.MySerDe';
```

As seen in the previous examples, traditional SQL features are present in the query engine such as from clause sub-queries, various types of joins, cartesian products, groups by and aggregations, union all, create table as selected and many useful functions on primitive and complex types make the language very SQL like. For some above mentioned it is exactly like SQL. Because of this very strong resemblance with SQL, users familiar with SQL can use the system from the start. It can be performed browsing metadata capabilities such as show tables, describe and explain plan capabilities to inspect query plans.

There are some limitations such as only equality predicates are supported in a join which has to be specified using ANSI join syntax in SQL:

```
SELECT t1.a1 as c1, t2.b1 as c2
FROM t1, t2
WHERE t1.a2 = t2.b2;
```

In Hive syntax can be as next:

```
SELECT t1.a1 as c1, t2.b1 as c2
FROM t1 JOIN t2 ON (t1.a2 = t2.b2);
```

HQL can be extended to support map-reduce programs developed by users, in the programming language of choice. If the map-reduce program is developed in python:

```
FROM(
  MAP doctext USING 'python wcP_mapper.py' AS (word, cnt)
  FROM docs
  CLUSTER BY word
) a
REDUCE word, cnt USING 'python wc_reduce.py'
```

MAP clause indicates how the input columns can be transformed using the 'wc_mapper.py', the *CLUSTER BY* specifies the output columns that are hashed on to distributed data to the reducers, that are specified by *REDUCE* clause and it is using the 'wc_reduce.py'. If distribution criteria between mappers and reducers must be defined the *DISTRIBUTED BY* and *SORT BY* clauses are provided by Hive.

```
FROM (
  FROM session_table
  SELECT sessionid, tstamp, data
  DISTRIBUTED BY sessionid SORT BY tstamp
) a
REDUCE sessionid, tstamp, data USING 'session_reducer.sh';
```

The first *FROM* clause is another deviation from SQL syntax. Hive allows interchange on the order of the clauses *FROM/SELECT/MAP/REDUCE* within a given sub-query and supports inserting different transformations results into different tables, HDFS or local directories as part of the same query, ability that allows the reduction of scans done on the input data.

In terms of Data Storage, Hive contains metadata table that associates the data in a table with HDFS directories:

- Tables - stored in HDFS as a directory. The table is mapped into the directory defined by hive.metastore.warehouse.dir in hive-site.xml.
- Partitions - several directories within the table directory. A partition of a table my be specified by the *PARTITION BY* clause.

```
CREATE TABLE test_part(c1 string, c2 int)
PARTITION BY (ds string, hr int);
```

  There is a partition created for every distinct value of *ds* and *hr*.

- Buckets - it is a file within the leaf level directory of a table or partition. Having a table with 32 buckets:

```
SELECT * FROM t TABLESAMPLE(2 OUT OF 32)
```

It is also possible, in terms of Data Storage, to access data stored in other locations in HDFS by using *EXTERNAL TABLE* clause.

```
CREATE EXTERNAL TABLE test_extern(c1 string, c2 int) LOCATION '/usr/mytables/mydata';
```

Since Hadoop can store files in different files, Hive have no restriction on the type of file input format and can be specified when the table is created. In the previous example, it is assumed that the type of access to the data is in Hive internal format. If not, a custom *SerDe* has to be defined. The default implementation of *SerDe* in Hive is the *LazySerDe* and assumes that a new line is American Standard Code for Information Interchange (ASCII) code 13 and each row is delimited by ctrl-A the ASCII code 1.

Hive is a work in progress and currently only accepts a subset of SQL as valid queries. Hive contains a naive rule-based optimizer with a small number of simple rules and supports JDBC/ODBC drivers[7][8].

2.5.2 Shark and Spark

Since the adoption of Hadoop and the Map-Reduce engine the data stored have been growing, creating the need to scale out across clusters of commodity machines. But this growing has shown that Map-Reduced engines have high latencies and have largely been dismissed for interactive speed queries. On the other hand, Massively Parallel Processing (MPP) analytic databases lack in the rich analytic functions that can be implemented in Map-Reduce[9]. It can be implemented in UDFs but these algorithms can be expensive.

Shark uses Spark to execute queries over clusters of HDFS. Spark is implemented based on a distributed shared memory abstraction called Resilient Distributed Datasets (RDD).

RDDs perform most of the computations in memory and can offer fine-grained fault tolerance, being the efficient mechanism for fault recovery one of the key benefits. Contrasting with the fine-grained updates to tables and the replications across the network for fault tolerance used by the main-memory databases, operations that are expensive on commodity servers, RDDs restrict the programming interface to coarse-grained deterministic operators affecting multiple data items at once[10]. This approach works well for data-parallel relational queries, recovering from failures by tracking the lineage of each data set and recover from it, and even if a node fails, Spark can recover mid-query.

Spark is based in RDD, which are immutable partitioned collections created through various data-parallel operators[10]. The queries, made by Shark, use three-step process: query parsing, logical

plan generations and physical plan generation[9]. The tree is generated parsing the query with Hive query compiler, adding rule based optimizations, such as pushing LIMIT down to individual partitions, generating the physic plan consisting of transformations on RDD instead of map-reduce jobs. Then, the master executes the graph in map-reduce scheduling techniques, resulting in placing the tasks near to the data, rerunning lost tasks, and performing straggler mitigation.

Shark is compatible with Hive queries and, by being compatible with Hive, Shark can query any data that is in the Hadoop storage API[11]. Shark supports text, binary sequence files, JSON and XML for data formats and inherit Hive schema-on-read and nested data types, allowing to choose which data will be loaded to memory. It can coexist with other engines such as Hadoop because Shark can work over a cluster where resources are managed by other resource manager.

The engine of Shark is optimized for an efficient execution of SQL queries. The optimizations are in DAG, improving joins and parallelism, columnar memory store, distribution of data loading, data co-partition and partition statistics and Map pruning. Next are described in more details the optimizations made in Shark to improve query times:

- **Partial DAG Execution (PDE)**
  The main reason is to support dynamic query optimization in a distributed setting. With it, it is possible to dynamically alter query plans based on data statistics collected at runtime. Spark materializes the output of each map in memory before shuffle, but if necessary it splits it on disk and reduce task will fetch this output later on. The main use of PDE is at blocking "shuffle" operator, being one of the most expensive tasks, where data is exchanged and repartitioned. PDE gathers customized statistics at global and per-partition, some of them are:
  - partition size and record counts to detect skew;
  - list of items that occurs frequently in the data set;
  - approximate histograms to estimate data distribution over partitions;

  PDE gathers the statistics when materializing maps and it allows the DAG to be altered for better performance.

  Join Optimization In the common map-reduce joins there are shuffle joins and map joins. In shuffle joins, both join tables are hash-partitioned by the join key. In map joins, or broadcast joins, a small input is broadcasted to all nodes and joined with the large tables for each partition. This approach avoids expensive repartition and shuffling phases, but map joins are only efficient if some join inputs are small.

  Shark uses PDE to select the strategy for joins based on their inputs for exact size and it can schedule tasks before other joins if the optimizer, with the information from run-time statistics, detects that a particular join will be small. This will avoid the pre-shuffling partitioning of a large table over the map-join task decided by the optimizer.

  Skew-handling and Degree of Parallelism Launching too few reducers may overload the network connection between them and consume their memories, but launching too many could extend the job due to task overhead. In Hadoop's clusters the number of reducer tasks can create large scheduling overhead and for that Hive can be affected it.

  PDE is used in Shark to use individual size partitions and determine the number of reducers at run-time by joining many small, fine-grained partitions into fewer coarse partitions that are used by reduce tasks.

- **Columnar Memory Store**
  Space footprint and read throughput are affected by in-memory data representation and one of the approaches is to cache the on-disk data on its naive format. This demands deserializations when using the query processor, creating a big bottleneck.

  Spark, on the other hand, stores data partitions as collections of Java Virtual Machine (JVM) objects avoiding deserialization by the query processor, but this has an impact on storage space and on the JVM garbage collector.

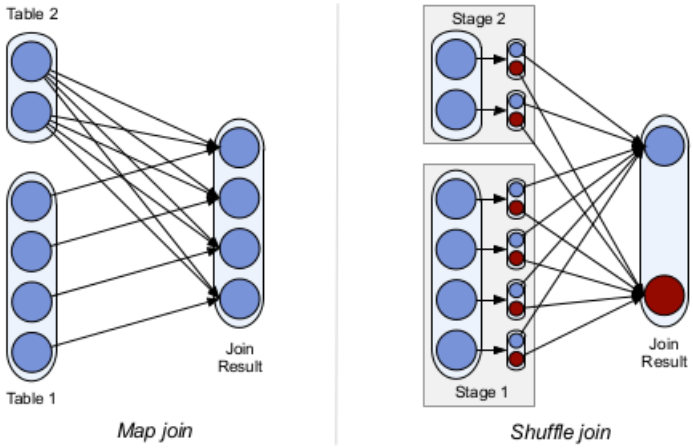


Figure 2.13: Data flows for map an shuffle joins[9]

Shark stores all columns of primitive types as JVM primitive arrays and all complex types supported by Hive, such as map and array, are serialized and concatenated into a single byte array. The JVM garbage collector and space footprint are optimized because each column creates only one JVM object.

- **Distributed Data Loading**
  Shark uses Spark for the distribution of data loading and, in each data loading task, metadata is tracked to decide if each column in a partition should be compressed. With this decision, by tacking each task it can choose the best compression scheme for each partition, rather than a default compression default scheme that is not always the best solution for local partitions. The loading tasks can achieve a maximum degree of parallelism requiring each partition to maintain its own compression metadata. It is only possible, because these tasks do not require coordination, being possible, for Shark, to load data into memory at the aggregated throughput of the CPUs processing incoming data.

- **Data Co-partitioning**
  In MPPs, the technique of co-partitioning two tables based on their join key in the data loading process is commonly used. However, in HDFS, the storage system is schema agnostic, preventing data co-partitioning. By using Shark, it will be possible to co-partitioning two tables on a common key for faster joins in subsequent queries.

```
CREATE TABLE l_mem TBLPROPERTIES ("shark.cache"=true)
AS SELECT * FROM lineitem DISTRIBUTED BY L_ORDERKEY;

CREATE TABLE o_mem TBLPROPERTIES ("shark.cache"=true, "copartition"="l_mem")
AS SELECTED * FROM order DISTRIBUTED BY O_ORDERKEY;
```

By using the co-partitioning in Shark, the optimizer constructs a DAG that avoids the expensive shuffle and uses map tasks to perform the joins.

- **Partition Statistics and Map Pruning**
  Map pruning is a process where the data partitions are pruned based on their natural clustering columns and Shark, in its memory store, splits these data into small partitions, each block containing logical groups on such columns. This avoids the scan over certain blocks where their

values fall out of the query filter range. The information collected for each partition has the range and the distinct values of the column. This information is sent to the master and kept in memory for pruning partitions during query executions. During the query, Shark evaluates the query's predicate against the partition statistics, That is, if there is a partition that does not satisfy the predicate this partition is pruned and no task is launched to scan this partition.

Map-Reduce like systems lacks on query time speed and Shark has enhanced times. The use of column-oriented storage and PDE gave the dynamic re-optimization of the queries needed at runtime.

2.6 Summary

In this chapter it was presented several existing solutions for Big Data. Some of the solutions used in the NoSQL systems can scale and process a big stream of data, but most of them lack of a query language that can be near the SQL standard. To use a SQL like engine in a Big Data system, the solutions best fit are Hive or Shark. Since Hive and Shark uses HDFS and Hive uses Hadoop Map-Reduce jobs, Hadoop system must be used. For the data stream, Storm is a flexible solution to feed real-time and off-line systems. In the next chapter is presented some scenarios using some of the solutions mentioned.

Chapter 3

Evaluation scenario

The analysis is focused on examining alternatives to the storage of generated data from the NE and the querying possibilities over that data. Usual industrial storage systems like SANs can be expensive over time as data grows in size and processing, demanding faster solutions. The main requirements for the system are defined by reducing costs in exotic hardware, increasing storage to a minimum of two years and query data no matter the level of aggregation within a reasonable time. This type of requests are mostly made by companies using Real Application Clusters (RAC) solutions from Oracle who pretend to reduce costs.

One of the main concerns is the retro-compatibility with the SQL standards. This is mainly because of the customization that the reporting application allows. These customizations are made by teams that have knowledge on SQL and use it to create the reports with counters and Key Performance Indicator (KPI) formulas, adding improvement on performance of the ad-hoc queries and keeping time results of common queries in an acceptable time.

Since the solutions that supports the most of the SQL standards, presented in the previous chapter, uses HDFS to store and Map-Reduce jobs to query the choice passes by using a Hadoop cluster, allowing the query engines to execute the Map-Reduce jobs over the cluster.

In this chapter it will be presented scenarios for the data stream, using Storm. It will be presented some approaches to store data and their impact on the final results over the queries, using solutions supporting SQL like engines. Based on the results collected over the evaluated scenarios, it will be defined a plan for two years of collected data.

3.1 Loading data

The core information is delivered by measurement files generated from NEs. Those files are designed to exchange performance measurement and are in XML format. These data is streamed to the node responsible for ETL over the measurements files and inserts the data into the tables of raw data in Oracle. The raw data can also suffer some aggregations, in time for example, to optimize querying the data.

This type of aggregations defines how data should be stored. To help understand which is the best solution for the reporting system, it is important to have a system that can simulate a real world scenario.

Since the data will be stored in HDFS, the final result will be in CSV format. As shown in figure 3.1, it is necessary a system to process all the data and a system to store it. A component, the *Simulator*, is used to simulate the NEs data and send that data to the *Data stream cluster*. After the data arrives, *Data stream cluster* analyze all the NEs data and transformed to the CSV format, sending that data to the *Storage cluster* using WebHDFS, which is a Representational state transfer (REST) Application Programming Interface (API) available in HDFS.

Next it will be presented, in more detailed description, the several components.

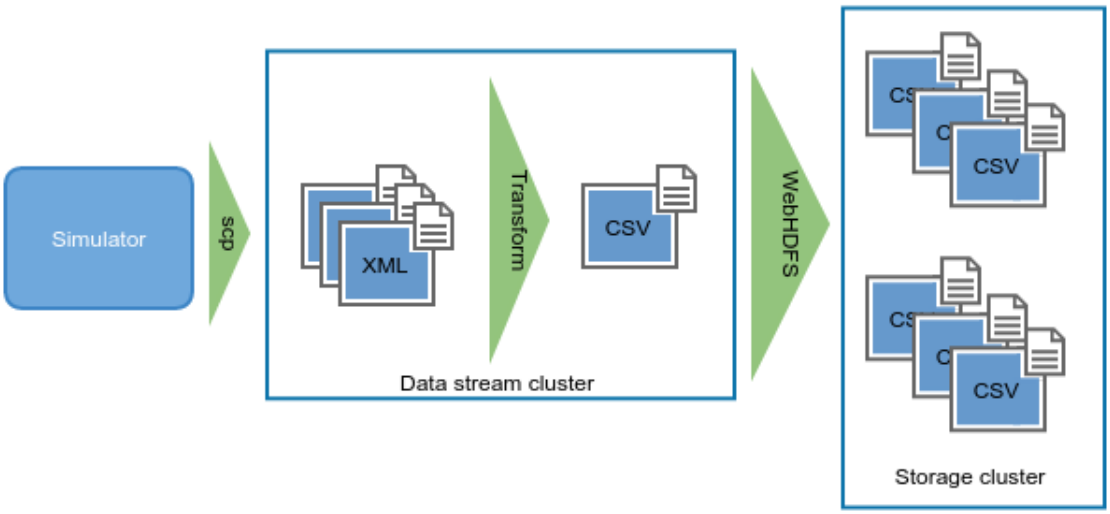


Figure 3.1: High level of data flow and formats in streaming and storage clusters

3.1.1 Network Element data simulation

The measurement files are generated by the NEs and, to simulate a real world scenario, a data simulation system is used. The data simulator generates data based on topology templates. As the following example shows, the *cbt* defines the date when the measurements are generated, and an *interval* is set to define the frequency of generation of the file with a *conjob*. As an example, the frequency of 15 minutes indicates that the file will arrive every 15 minutes with the measurements. The *nedn* is the distinguished name of the element, for example *System=UTRANNetwork,RNC=123, Cell=997*, to identify the object and the counters are defined in *mt* and *mv*. The values for this scenario are generated in ranges of 1 to 100.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="MeasDataCollection.xsl" ?>
<!DOCTYPE mdc SYSTEM "MeasDataCollection.dtd" >
<mdc xmlns:HTML="http://www.w3.org/TR/REC-xml">
    <mfh>
        <ffv>1</ffv>
        <sn>System=UTRANNetwork,RNC=123</sn>
        <st>RNC</st>
        <vn>Telecom corp.</vn>
        <cbt>20000301140000</cbt>
    </mfh>
    <md>
        <neid>
            <neun>RNC Telecomville</neun>
            <nedn>System=UTRANNetwork,RNC=123</nedn>
        </neid>
        <mi>
            <mts>20000301141430</mts>
            <gp>900</gp>
            <mt>attTCHSeizures</mt>
            <mt>succTCHSeizures </mt>
            <mt>attImmediateAssignProcs</mt>
            <mt>succImmediateAssignProcs</mt>
```

```
<mv>
    <moid>Cell=997</moid>
    <r>34</r>
    <r>45</r>
    <r>67</r>
    <r>89</r>
    <sf>FALSE</sf>
</mv>
<mv>
    <moid>Cell=998</moid>
    <r>90</r>
    <r>01</r>
    <r>23</r>
    <r>34</r>
    <sf>FALSE</sf>
</mv>
<mv>
    <moid>Cell=999</moid>
    <r>56</r>
    <r>67</r>
    <r>78</r>
    <r>89</r>
    <sf>FALSE</sf>
```

The data simulator requires Red Hat Linux 5.8, or above, and Perl 5.8.8 and it is installed in a single machine. The simulator is structured with some folders for the configuration needed.

```
bin - the binaries to generate the measurements
config - the configuration folder with the supported intervals
log - logs of the application
output - the measurements files are saved in this folder
program - scripts to start the program
templates - the template with all the topologies used for tests
```

Over the configuration folder, it is defined the host to transfer data, the topologies to be generated and the type of protocol (scp, ftp, etc...) to be used to transfer data. This configuration is then added to a configuration file that has an interval defined, for example *theconfiguration_{quarter or hour or day}*. To start generating the measurements it is necessary to add the information to the *crontab* the file configured by the specified interval.

3.1.2 Data stream

In order to transform the data in the *Data stream cluster*, is necessary to take in consideration some of the requirements for the report queries, such as KPI formulas and aggregations of data.

KPIs are mathematical formulas that are applied over the NE counters. On a high-level view, it can be said that the KPIs are not based in measurements but rather calculations for a specific time period, network level or a set of network objects. When calculating the KPIs there are three aggregation levels specified.

- Time level aggregations - is a set of measurement values for a network object within a time period. In all the measurement values for network the objects are listed for the specific time period defined.
- Network level aggregations - the network objects can have child objects and form a tree-graph. As an example, the object PLMN-PLMN/RNC-{id}/WBTS-{id}/WCEL-{id} in Nokia 3G format has WBTS-{id} as a parent. The input of the network level aggregation, for a network

object in a specific time stamp, gathers the measurement values for that time stamp of all child objects, and then applies the aggregation function.

- Object level aggregations - it is selected a subset of the object within a network level aggregation and applied in them.

Based on the aggregation examples, to store the data for future readings, it is necessary to define the structure of the data. This structure should take in consideration, besides the measurements, the time when the measurements where generated, from which topology and the associated object (figure 3.2).

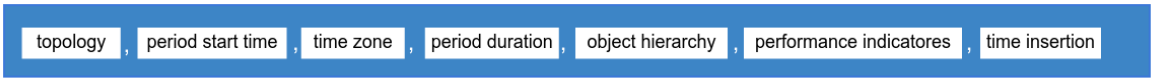


Figure 3.2: High-level structure of a CSV file

Storm is used to manage the data stream from the NEs simulator. It is already used in the parsing of measurement files, helping on the calculation of KPIs in real-time reporting system. This system had a topology already defined, but since its definition is a very simple network structure, a new *bolt* was added to manage the stream for off-line reporting system. This *bolt*, as is shown in figure 3.3, will create a block file of configured size.

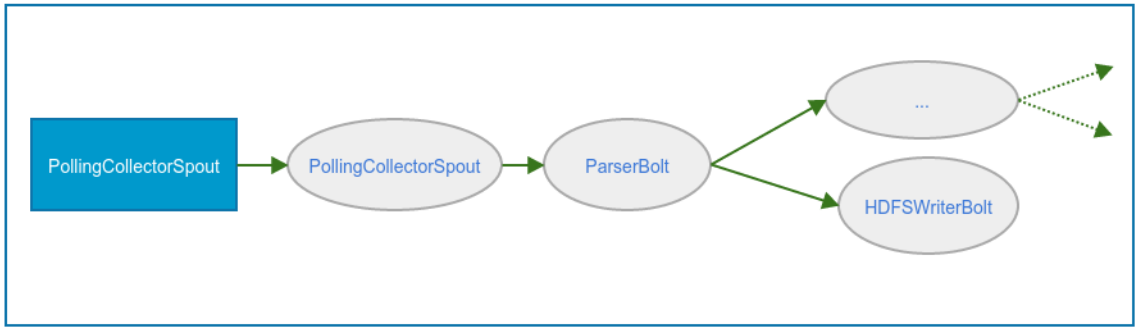


Figure 3.3: Topology implementation

These definitions are made in a configuration file that is deployed with the topology in the Nimbus node. In these configurations, it is possible to define the size of the block:

```
<blockSize>64000000</blockSize>
```

It is also define the URI of the NameNode, the measurement files and where it will be saved after the transformation:

```
<rootpath>http://<NAMENODE>:<PORT>/webhdfs/v1/</rootpath>
<sourcepath>/path/to/measurements/files/temp/</sourcepath>
<targetpath>/path/to/transformed/data/</targetpath>
```

The connection with the NameNode is made with WebHDFS. With this interface, it is possible to create new files on HDFS using a *PUT* request and defining the type of operations that we want to execute. To create a new file, it is necessary to execute two steps. This is a workaround for a software library bug[11]. The next two steps are presented in a Java example. The first step is a request to the NameNode for the creation of a new file and is defined as:

```
//curl -i -X PUT "http://<NAMENODE>:<PORT>/webhdfs/v1/<PATH>?op=CREATE
//[&overwrite=<true|false>][&blocksize=<LONG>][&replication=<SHORT>]
//[&permission=<OCTAL>][&buffersize=<INT>]"
String nnUrl = rootPath + filePath + "?op=CREATE&overwrite=true&permission=" + permission
    + "&user.name=" + user;
URL url = new URL(nnUrl);
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setRequestMethod("PUT");
conn.setInstanceFollowRedirects(false);
responseCode = conn.getResponseCode();
```

By using this first command the NameNode will send a response with a redirect (307 Temporary Redirect) to a DataNode, as shown in the next example:

```
HTTP/1.1 307 TEMPORARY_REDIRECT
Location: http://<DATANODE>:<PORT>/webhdfs/v1/filepath?op=CREATE...
Content-Length: 0
```

In the response, by using the *Location* returned in the REDIRECT header, it is necessary to execute another request with the file:

```
//curl -i -X PUT -T <LOCAL_FILE> "http://<DATANODE>:<PORT>/webhdfs/v1/<PATH>?op=CREATE..."
if (responseCode == 307) {
    // TEMPORARY_REDIRECT (307) towards appropriate data node
    dnUrl = conn.getHeaderField("Location");
    url = new URL(dnUrl);
    conn = (HttpURLConnection) url.openConnection();
    conn.setRequestMethod("PUT");
    conn.setRequestProperty("Content-type", "application/octet-stream");
    conn.setDoOutput(true);
}
```

The final result, after adding the file, will be:

```
HTTP/1.1 201 Created
Location: webhdfs://<DATANODE>:<PORT>/<PATH>
Content-Length: 0
```

After all the steps done the data is now inserted in HDFS and replicated by the factor defined. The cluster with all the components together is represented in figure 3.4. The flow starts in the *Simulator*, passing NEs measurements to *Nimbus*. After receiving the data, *Nimbus* will coordinate, through *Zookeeper*, the execution of the topologies on each *Supervisor*. When the file built reaches, for example, 64Mb in size, it will send the CSV to HDFS cluster though the *WebHDFS* API.

In the next section, it will be explained the HDFS system deployment and the inputs obtained in simple tests over several formats or sizes of data.

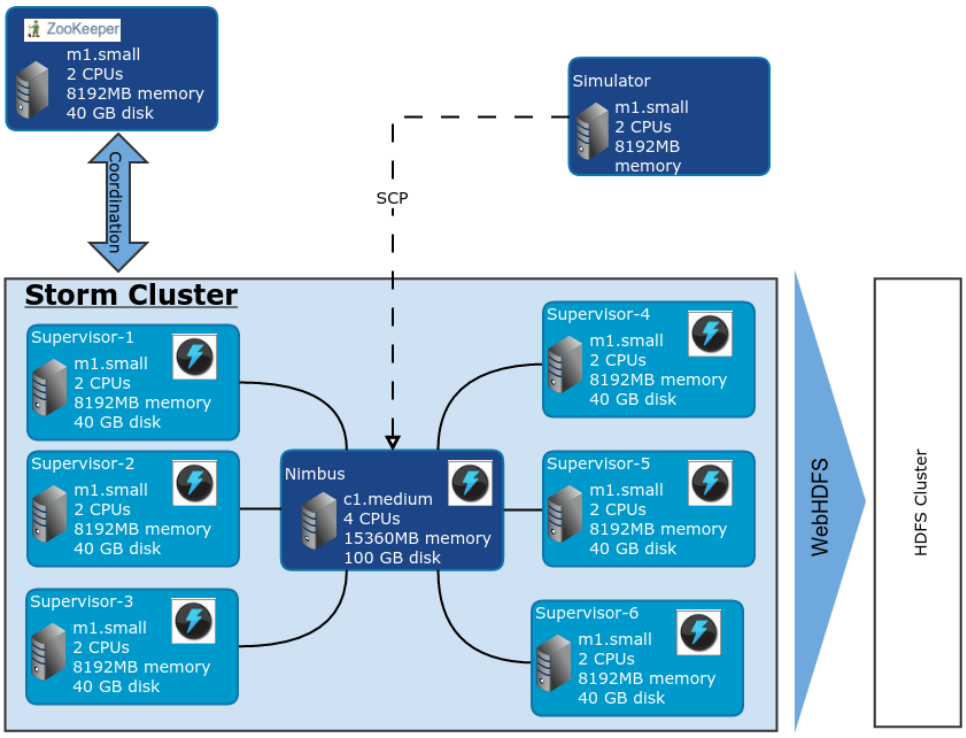


Figure 3.4: Data stream cluster(all together) specifications

3.2 Storing data

As previously seen, the information generated is based on small files from some bytes to some kilobytes, but one of the default configurations in a HDFS system is a block size of 64Mb in the data nodes. The block size is the smallest unit of data that a file system can store and, up to 64Mb, it will take up one block[11]. After this boundary, a second block will be needed. For each block, it is created a task to process the data. The block size should be as large as possible (this means that it cannot affect parallelization, so it cannot be to big). The Name Node serves all of its metadata directly from RAM that contains the filename, permissions, owner and group data, list of blocks that make each file and the current known location of each replica of the blocks, in the Data Nodes (figure 3.5)[14].

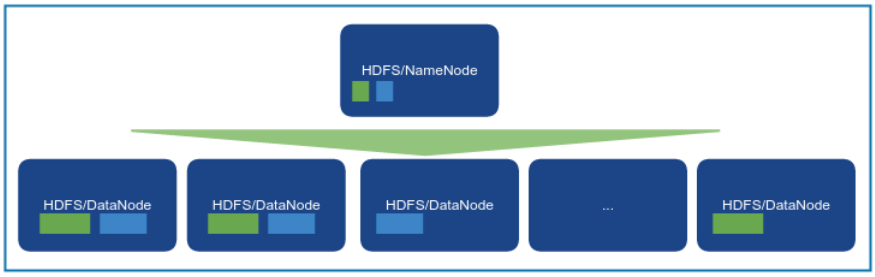


Figure 3.5: Data storage cluster with master/slave block files metadata

Pointing this out, at scale, there are some decisions that can affect the Name Node, for instance the length of the filenames starts to matter, and the longer the filename, the more bytes will occupy

in memory, or small files can generate more blocks and associated metadata. As a rule of thumb, the NameNode consumes roughly 1Gb for every 1 million blocks[14]. Since the metadata must fit in memory, the Name Node normally does not take more than that on disk.

HDFS allows to store data in blocks with different sizes or formats. In the next scenarios is tested the block size, the file formats and the partitioning of the blocks in a HDFS cluster. It is analyzed their influence in the results of simple queries over the data and, in some cases, the space used. The specifications of the cluster where the tests were made are represented in the figure 3.6.

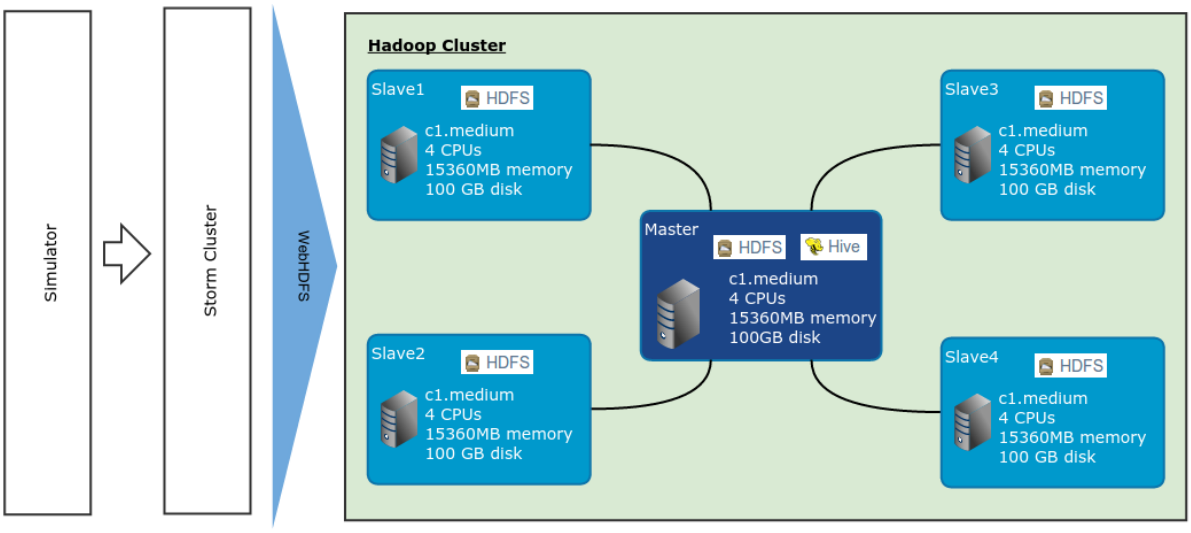


Figure 3.6: Data storage cluster(all together) specifications for storage scenario

This cluster is installed with Hadoop, for the Map-Reduce jobs and HDFS, and with Hive to allow the use of SQL over Hadoop cluster.

Blocks

In this scenario, it is shown how Hadoop and Hive behaves with the storage of blocks in different sizes. Next is an example of simple queries, and simple format file, run against two tables. One is stored as 1Mb blocks and another as 64Mb blocks:

```
SELECT * FROM test_1mb_blocks WHERE ts>0;
```

```
SELECT * FROM test_64mb_blocks WHERE ts>0;
```

Each block in the HDFS is processed in a separated task. When there are more blocks, more tasks will be scheduled. When a huge number of tasks are defined, this can create latency problems in the Map-Reduce jobs. By storing block files of 1Mb, there will be more tasks assign to be executed and, therefore, more time will be needed to execute a query. Hive creates Map-Reduce splits at the block boundary, and excessive number of splits prevents workers of reading large chunks of the files sequentially. To optimize query times, large blocks of files should be used, but keeping in mind that it can not compromise the parallelism of the map-reduce framework. The figure 3.7 shows the performance of each query over different sizes of data stored. In this case, the data stored in total is of 1Gb, 10Gb and 100Gb.

The scenario shows, as expected, an improvement of performance in every total amount of data stored. The 64MB blocks scenario shows an improvement of performance relative to 1Mb blocks in

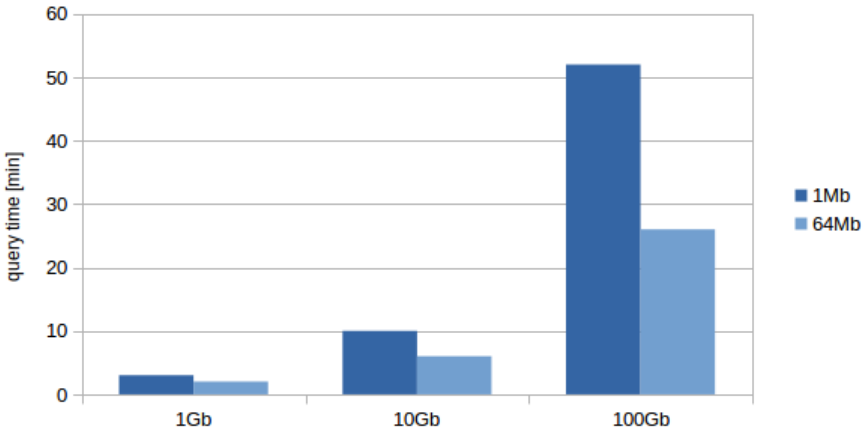


Figure 3.7: Block size queries

every total amount of data stored. For 1Gb of total data case a 33.3% improvement achieved, for 10Gb an improvement of 40% and for the total of 100Gb of data the improvement was 50%. Next will be evaluated some file formats and the space used by them.

Files format

The file format is another concern to improve performance over query times and, obviously, the space used in the cluster. In this scenario it will be shown some file format types that can be used, such as plain text, sequence or RCFile.

In the scenario using plain text the CSV format can be used as an example. This format allows a better parsing over the data and if necessary a fast concatenation of small files, reducing the number of small blocks and improve performance. The schema of the file is represented bellow maintaining the the format defined in figure 3.2:

```
Topology,timestamp,time_zone,period_duration,PLMN-ID,RNC-ID,WBTS-ID,WCEL-ID,counter1,
    counter2,...,countern,insert_time
```

In this type of format, Hive's metadata will be less flexible when it is necessary to add more counters, from NEs, to the KPIs formulas. The table created for this scenario would be:

```
CREATE TABLE IF NOT EXISTS Traffic (
    topology STRING,
    timestamp TIMESTAMP,
    time_zone STRING,
    period_duration TIMESTAMP,
    object STRING,
    counter1 DOUBLE,
    counter2 DOUBLE,
    ...
    countern DOUBLE,
    insert_time TIMESTAMP
)
```

```
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','

LOCATION '/path/to/traffic'
STORED AS TEXTFILE;
```

For a more flexible format Hive has Complex Data Types, such as *Structs*, *Maps* or *Arrays*. The most important to test is the *Map* Complex Data Type, because it will solve repetitive data pairs (a problem encountered in the present NEs stream). The *Map* will be flexible in the number of counters and their values. An example of usage is:

```
Topology,timestamp,time_zone,period_duration,PLMN-ID,RNC-ID,WBTS-ID,WCEL-ID,counter1=0|
    counter2=0|...|countern=0,insert_time
```

This type of notation can be parsed with a Hive table, the defined table is according to the following:

```
CREATE TABLE IF NOT EXISTS Traffic (
    name STRING,
    timestamp TIMESTAMP,
    time_zone STRING,
    period_duration TIMESTAMP,
    mo STRING,
    counter MAP<STRING, DOUBLE>,
    insert_time TIMESTAMP
)

ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
COLLECTION ITEMS TERMINATED BY '|'
MAP KEYS TERMINATED BY '='

LOCATION '/path/to/traffic'
STORED AS TEXTFILE;
```

Defining the separation between the fields and the collection of counters, it is possible to load the information to Hive table. As seen in the example, this type of solution tends to occupy more space compared to the previous schema. It adds more characters to the file, creating bigger files but creating more flexibility over the addition of new counters.

The sequence files are traditionally used by Hadoop and it is a way of saving flat sequence files internally. This is a binary storage format for key value pairs and it has the benefit of being a more compact option then plain text files. With sequence files, it is possible to compress on value, or block level, to improve IO profile[11].

A sequence file has a header containing information on the key/value class names, version, file format, metadata about the file and sync marker to denote the end of the header. On the record area there is the record length, key length, key value, and, at the end, there is a sync-marker every few 100 bytes. A Sequence file can have three different formats: an Uncompressed format, a Record Compressed format, where the value is compressed, and a Block Compressed format, where entire records are compressed. The representation of figure 3.8 is in an uncompressed format.

By using Hive query engine, the sequence files are not an optimal solution. The reason is that files are saved in a complete row as a single binary value. The query engine would need to read the full row and decompress it, even if only a reduce number of columns was needed for the query.

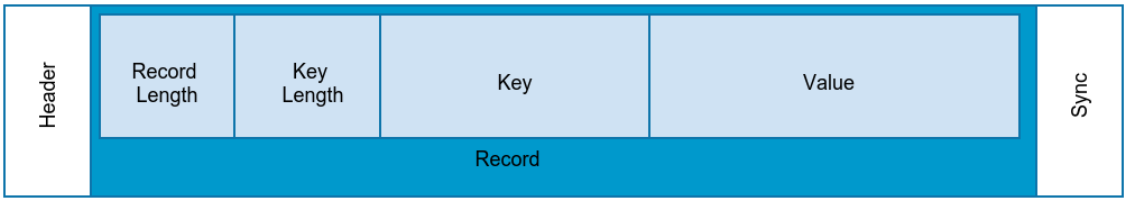


Figure 3.8: Sequence file structure

RCFiles format was co-developed by Facebook with the goal of fast data loading, fast query processing, highly efficient storage space utilization and strong adaptation to highly dynamic workload patterns[18]. The RCFile structure is based on a table stored first horizontally, partitioned into multiple groups, and each row group is vertically partitioned making possible storing each column independently. Then, RCFile uses a column-wise data compression for each row group, providing a lazy decompression to avoid unnecessary column decompression during queries[18]. With this structure, it is possible to create a flexible row group size. Although a default size is given for compression performance and query performance, users can select the row group size for a given table.

Figure 3.9 shows how the RCFile splits data horizontally into row groups. For example, rows 101 to 145 are stored in one group, and rows 106 to 149 in the next, and so on. One or several groups are stored in a HDFS file and the RCFile saves the row group data in a columnar format. So, instead of storing row one and then row two, it stores column one across all rows, then column two across all rows, and so on.

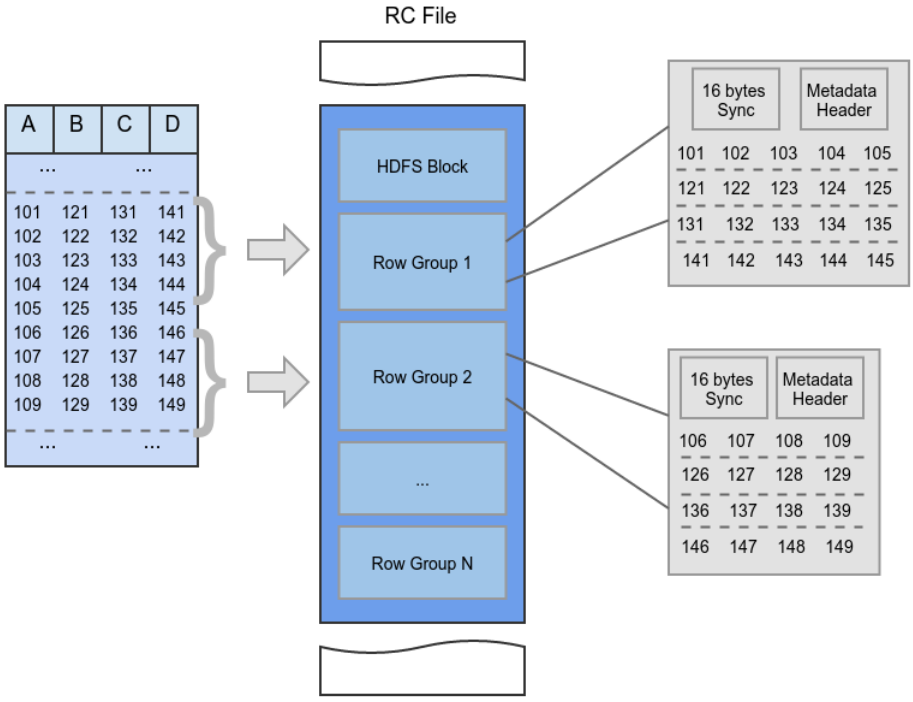


Figure 3.9: RCFile structure [18]

The benefit of this data organization is that parallelism still applies, since the row groups in different files are distributed redundantly across the cluster and processed at the same time. Subsequently,

each processing node reads only the columns relevant to a query from a file and skips irrelevant ones. Additionally, compression on a column base is more efficient. It can take advantage of the similarity of the data in a column.

One of the problems in this type of file is that, as in the first option of plain text files, columns must be defined in the metadata.

```
CREATE TABLE IF NOT EXISTS Traffic_staging (
    topology STRING,
    timestamp TIMESTAMP,
    object STRING,
    counter1 DOUBLE,
    counter2 DOUBLE,
    ...
    countern DOUBLE,
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/path/to/traffic/plain_text'

CREATE TABLE IF NOT EXISTS Traffic (
    topology STRING,
    timestamp TIMESTAMP,
    object STRING,
    counter1 DOUBLE,
    counter2 DOUBLE,
    ...
    countern DOUBLE,
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.columnar.ColumnarSerDe'
STORED AS RCFILE
LOCATION '/path/to/traffic/rc_files';

INSERT OVERWRITE table Traffic SELECT * from Traffic_staging;
```

There is an optimization in the conversion of the data to RCFile, because it creates a smaller file and, as seen before, this affects the execution of the map-reduce jobs.

To reduce more the amount of space it can be used compression methods, and Map-Reduce intermediate compression can make jobs run faster without you having to make any application changes. Snappy is ideal in this case, because it compresses and decompresses very fast compared to other compression algorithms, such as Gzip. Snappy is a compression/decompression library developed by Google and it is not focused on maximum compression. Instead, it aims for high speeds and reasonable compression.

To activate snappy compression in Hadoop it is necessary to change the configuration file:

```
<property>
        <name>mapreduce.map.output.compress</name>
        <value>true</value>
</property>
<property>
        <name>mapred.map.output.compress.codec</name>
        <value>org.apache.hadoop.io.compress.SnappyCodec</value>
</property>
```

To activate in Hive for output files:

```
SET hive.exec.compress.output=true;
SET mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
SET mapred.output.compression.type=BLOCK;
```

Figure 3.10 shows some size scenarios for file formats and the difference between compressed and uncompressed format. Not all the possible scenarios were tested. The plain text(CSV) format scenario is represented as a starting point for the analyses. Sequence file format shows an increase in size of 15% comparing with plain text. Using the snappy compression defined at block level, the reduction of space improved 94% compared with sequence files with no compression. After the results obtained with the compression in the sequence files, the RCFile format was only tested with compression. RCFiles shows an improvement of 13% compared with sequence file using compression. The results were the same for 1Gb and 10Gb of data.

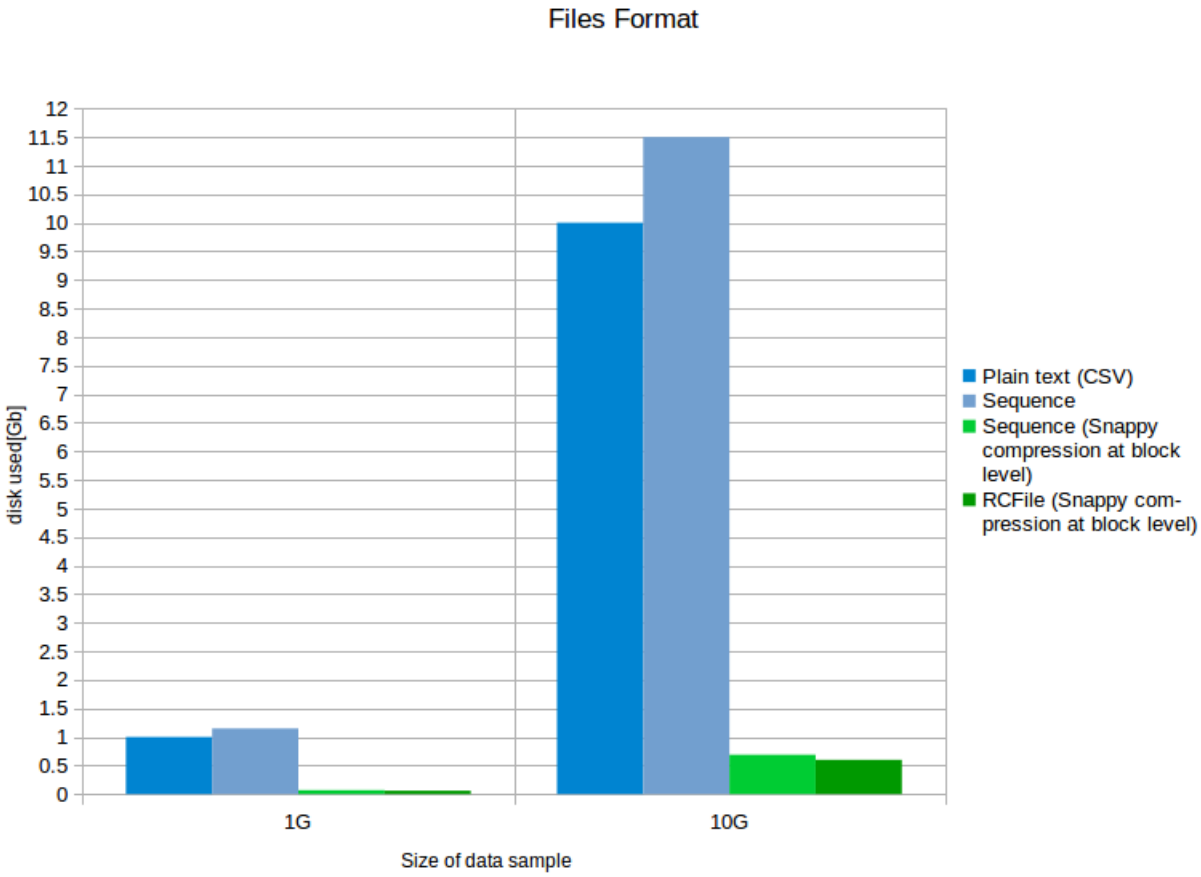


Figure 3.10: Space used for different file formats with and without compression

One of the key factors to improve disk space is the use of compression, independently of the file format. The best results were obtained in RCFiles because of the structure defined for it. Using compression it is necessary to take in consideration that is necessary a decompression to query the data, adding one more task for every query performed. Next it will be evaluated the partition of data and the influence in the queries.

Partitions

In some cases the file format is focused on the performance of Map-Reduce jobs. For this, formats are created with the objective of avoiding full search over data. Therefore, Hive engine allows the creation of partitions, as a mechanism to separate data, and could be defined when loading data:

```
CREATE TABLE IF NOT EXISTS ne_counters_table (
       period_start_time BIGINT
       ...
)
PARTITIONED BY (year INT, month INT)
...
```

In this example, the data would be partitioned by the time when the data was collected by the NEs, optimizing queries with aggregation of years and months. As an example, the next block of query represents the *where* clause using partition for months. The *insert_time* is in milliseconds and to obtain the month value is used some UDFs available in Hive.

```
WHERE month( from_unixtime(floor( (td.insert_time / 1000) )) ) = {month}
```

Figure 3.11 shows the results of the scenario with partitions and the improvement obtained. In this scenario the partitions created are based on the *period_start_time*. To test the scenario it was used the *WHERE* clause in order to define a portion of data to search.

```
SELECT * FROM traffic t
WHERE t.period_start_time >= unix_timestamp('2014-08-15 10:00:00')
     AND t.period_start_time < unix_timestamp('2014-08-15 15:00:00')
```

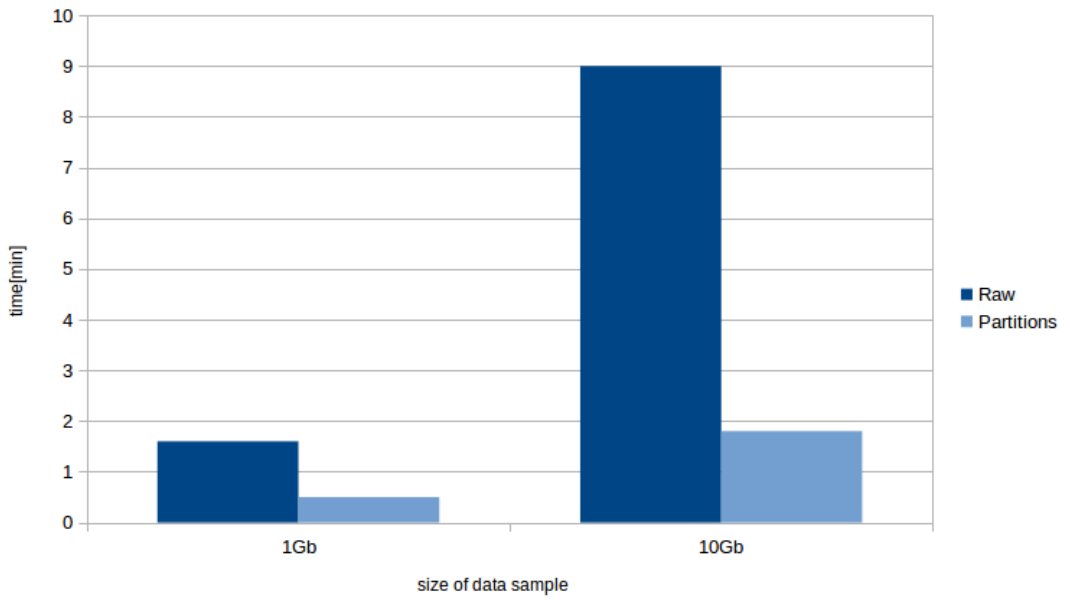


Figure 3.11: Partitioning data and the improvement obtained

The partition scenario shows an improvement of performance relative to raw data. For 1Gb of total data case a 68.75% improvement achieved and for 10Gb an improvement of 80% relative to raw data.

3.3 Queries

In this section it will be tested the scenario of common queries used to calculate the KPIs formulas. In order to test the use of legacy SQL, it was created a database to test performance results and identify differences between SQL and HQL. It will be analyzed solutions to the UDFs used, for example, in Oracle. This database is created in the Hive metastore since it can be used from Hive or Shark. Hive's metadata will be used to test the performance of Shark and Hive using HQL.

HQL vs SQL

Based on a query used in the reporting, the next SQL represents a common use of the data in the reporting system. The counters and some names where altered for example purposes:

```
SELECT
    trunc( p.period_start_time, 'dd' ) period_start_time,
    plmn.co_gid plmn_gid,
    SUM(COUNTER1) COUNTER1,
    SUM(COUNTER1) COUNTER2,
    SUM(COUNTER3) COUNTER4,
    SUM(COUNTER4) COUNTER5,
    SUM(COUNTER5) COUNTER5,
    SUM(COUNTER6) COUNTER6,
    SUM(COUNTER7) COUNTER7,
    SUM(COUNTER8) COUNTER8,
    SUM(COUNTER9) COUNTER9,
    SUM(COUNTER10) COUNTER10,
    SUM(COUNTER11) COUNTER111,
    SUM(COUNTER12) COUNTER12
FROM
    objects plmn,
    objects mrbts,
    objects lnbts,
    objects lncel,
    lncelho_raw p
WHERE
    plmn.co_gid in ('1001')
    and period_start_time >= to_date('2013/01/15 00:00:00', 'yyyy/mm/dd hh24:mi:ss')
    and period_start_time < to_date('2013/01/16 00:00:00', 'yyyy/mm/dd hh24:mi:ss')
    and lncel.co_oc_id=3130
    and lncel.co_gid = p.lncel_id
    and lncel.co_parent_gid = lnbts.co_gid
    and lnbts.co_oc_id=3129
    and lnbts.co_gid = p.lnbts_id
    and lnbts.co_parent_gid = mrbts.co_gid
    and mrbts.co_oc_id=3128
    and mrbts.co_gid = p.mrbts_id
    and mrbts.co_parent_gid = plmn.co_gid
    and plmn.co_oc_id=16
GROUP BY
    trunc( p.period_start_time, 'dd' ),
    plmn.co_gid;
```

Based on the type of query that will be tested in HQL, the schema for the metadata was defined as in figure 3.12. This schema is defined only to allow querying data on a similar to the SQL above. The *Raw data* is the staging area where all the data generated by the network elements will be formatted and stored. The *Aggregated data* is a simulation of a schema from relational database to allow the simulation of queries with joins such as the ones used in SQL.

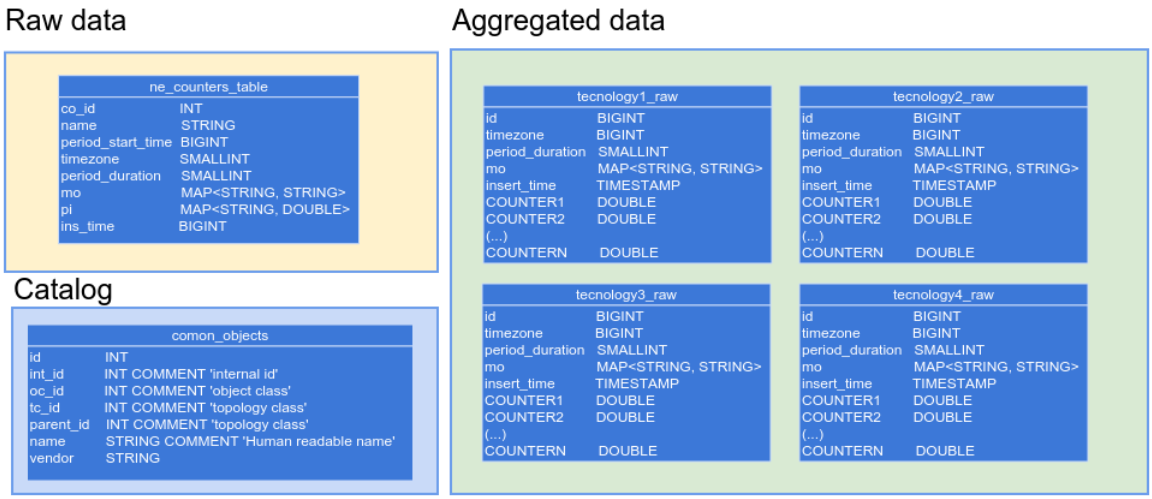


Figure 3.12: Hive Metadata schema

The schema of Hive's metadata only represents a portion of data to store. All the data that is generated by the network elements were not simulated, since the resources were limited.

The possible HQL, to query data in the table *ne_counters_table*, to generate the same result as the query SQL previously shown is represented below. This example is not intended to be optimized, it only intents to show how small the differences between SQL and HQL can be:

```
SELECT
    day(p.period_start_time) AS period_start_time,
    co.parent_ id AS plmn_id,
    sum(p.pi["COUNTER1"]) AS COUNTER1,
    sum(p.pi["COUNTER2"]) AS COUNTER1,
    sum(p.pi["COUNTER3"]) AS COUNTER3,
    sum(p.pi["COUNTER4"]) AS COUNTER4,
    sum(p.pi["COUNTER5"]) AS COUNTER5,
    sum(p.pi["COUNTER6"]) AS COUNTER6,
    sum(p.pi["COUNTER7"]) AS COUNTER7,
    sum(p.pi["COUNTER8"]) AS COUNTER8,
    sum(p.pi["COUNTER9"]) AS COUNTER9,
    sum(p.pi["COUNTER10"]) AS COUNTER10,
    sum(p.pi["COUNTER11"]) AS COUNTER11,
    sum(p.pi["COUNTER12"]) AS COUNTER12
FROM
    ne_counters_table p
JOIN
    catalog.common_objects co ON p.co_id = co.id AND co.oc_id = 3128
WHERE
    co.parent_id = '1001'
    AND p.period_start_time >= unix_timestamp('2013-01-15 00:00:00')
    AND p.period_start_time < unix_timestamp('2015-01-16 00:00:00')
```

```
GROUP BY
    day(p.period_start_time),
    co.parent_id;
```

The previous HQL is not optimized for fast response. By analyzing the table, it is clear that it uses every data collected from the network elements, making a full search since the table is not partitioned. This example only shows how to use *Maps* in the HQL. In order to improve performance and more approximated type of schema used in the SQL query, the data from the *ne_counters_table* is passed to the aggregation tables. Since all the aggregations follow the same process only one example will be showed below, creating partitions from the field *period_start_time*:

```
FROM ne_counters_table td

INSERT INTO TABLE tecnology1_raw
PARTITION (period_start_time)
SELECT
    td.timezone AS timezone,
    td.period_duration AS period_duration,
    td.mo AS mo,
    from_unixtime(floor((td.insert_time / 1000))) AS insert_time,
    td.pi["COUNTER1"] AS COUNTER1,
    td.pi["COUNTER2"] AS COUNTER2,
    td.pi["COUNTER3"] AS COUNTER3,
    td.pi["COUNTER4"] AS COUNTER4,
    td.pi["COUNTER5"] AS COUNTER5,
    td.pi["COUNTER6"] AS COUNTER6,
    td.pi["COUNTER7"] AS COUNTER7,
    from_unixtime(floor((td.period_start_time / 1000))) AS period_start_time
WHERE
    td.insert_time=${insert_time} AND td.name='TECNOLOGY1'
```

With this aggregation the data will be optimized and improve the query search. Next is the HQL used to query the new aggregation table:

```
SELECT
    day(p.period_start_time) AS period_start_time,
    co.parent_id AS plmn_gid,
    sum(p.COUNTER1) AS COUNTER1,
    sum(p.COUNTER2) AS COUNTER2,
    sum(p.COUNTER3) AS COUNTER3,
    sum(p.COUNTER4) AS COUNTER4,
    sum(p.COUNTER5) AS COUNTER5,
    sum(p.COUNTER6) AS COUNTER6,
    sum(p.COUNTER7) AS COUNTER7,
    sum(p.COUNTER8) AS COUNTER8,
    sum(p.COUNTER9) AS COUNTER9,
    sum(p.COUNTER10) AS COUNTER10,
    sum(p.COUNTER11) AS COUNTER11,
    sum(p.COUNTER12) AS COUNTER12
FROM
    tecnology1_raw p
JOIN
    catalog.common_objects co ON p.co_id = co.id AND co.oc_id = 3128
WHERE
    co.parent_id = '1001'
```

```
    AND p.period_start_time >= unix_timestamp('2013-01-15 00:00:00')
    AND p.period_start_time < unix_timestamp('2015-01-16 00:00:00')
GROUP BY
    day(p.period_start_time),
    co.parent_gid;
```

With aggregation tables it is possible to create HQL that is very similar with the first SQL showed. Next it is presented a solution for UDF used in SQL to HQL.

User Defined Functions

As seen in the previous queries the *sum* function is used, but in some cases there are functions that are not defined by Hive UDFs. For these cases, Hive has an API that allows the creation of custom functions. The following example shows how to use the API:

```
import java.util.Date;
import java.text.SimpleDateFormat;
import org.apache.hadoop.hive.ql.exec.UDF;

@Description(
    name = "nvl",
    value = "nvl(expr1, expr2) - Returns expr2 if expr1 is null",
    extended = "SELECT nvl(dep, 'Not Applicable') FROM src; 'Not Applicable' if dep is
        null."
)
public class NVL extends UDF {
    public NVL( InputDataType InputValue ){ ..; }
    public String evaluate( InputDataType InputValue ){ ..; }
}
```

To use these type of functions, it is necessary to set them before the execution of the query. The next example shows how to declare them in Hive CLI:

```
hive> add jar /path/to/udf-functions.jar;
hive> CREATE TEMPORARY FUNCTION nvl AS 'the.package.udf.NVL';
hive> SELECT nvl(p.pi["COUNTER1"], 0 ) AS COUNTER1 FROM ne_counters_table p;
```

These functions can also be defined in *hive-xml* configuration file, as defined in the next example:

```
<property>
    <name>hive.aux.jars.path</name>
    <value>file:///path/to/udf-functions.jar</value>
</property>
```

There are some UDFs already developed and that can be included as a *jar*, as seen in the previous example, and that have some of the common used functions used from RDBMs, such as Oracle. These functions are in GitHub and were developed by KTnexR[16]. The functions supported are:

Function	**Description**
nvl(expr1, expr2)	Returns expr2 if expr1 is null.
decode(value1, valuen, ... defaultValue)	Returns value3 if value1=value2 otherwise defaultValue.

nvl2(string1, value_if_not_null, value_if_null)	Returns value_if_not_null if string1 is not null, otherwise value_if_null.
str_to_date(dateText,pattern)	Convert time string with given pattern to time string with 'yyyy-MM-dd HH:mm:ss' pattern.
to_char(date, pattern)	Converts a string with yyyy-MM-dd HH:mm:ss pattern to a string with a given pattern.
to_char(datetime, pattern)	Converts a string with yyyy-MM-dd pattern to a string with a given pattern.
to_char(number [,format])	Converts a number to a string.
date_format(dateText,pattern)	Return time string with given pattern. Convert time string with 'yyyy-MM-dd HH:mm:ss' pattern to time string with given pattern.
instr4(string, substring, [start_position, [nth_appearance]])	Returns the index of the first occurrence of substr in str.
chr(number_code)	Returns the character based on the NUMBER code.
last_day(dateString)	Returns the last day of the month based on a date string with 'yyyy-MM-dd HH:mm:ss' pattern.
greatest(value1, value2, value3, ...)	Returns the greatest value in the list.
to_number(value, format_mask)	Returns the number converted from string.
substr(str, pos[, len])	returns the substring of str that starts at pos and is of length len.
substr(bin, pos[, len])	returns the slice of byte array that starts at pos and is of length len.
trunc(date, [format_mask])	Returns a date in string truncated to a specific unit of measure.
sysdate()	Returns the current date and time as a value in 'yyyy-MM-dd HH:mm:ss'
lnnvl(condition)	Evalutates a condition when one of the operands may contains a NULL value.

Table 3.1: UDF functions

Hive UDFs solves most of the cases, in case of legacy SQL UDFs Hive offers a very useful API. These type of functions, when developed, can cause performance degradation since it is dependent on the quality of the code in each function.

Performance

In this scenario it is analyzed the performance of Hive and Shark over some data samples. The queries will be executed on the *ne_counter_tables* and in *tecnology1_raw* tables, and the files are stored with Snappy compression in 64Mb blocks and partitioned by month. The tests will be executed in the cluster with the specification shown in the figure 3.13, showing the differences between Hive and Shark.

Since the tests are made with Hive and Shark in the same cluster using the same metadata, some attention is needed for unsupported Hive features by Shark:

- It does not support inserting of tables using dynamic partitioning.

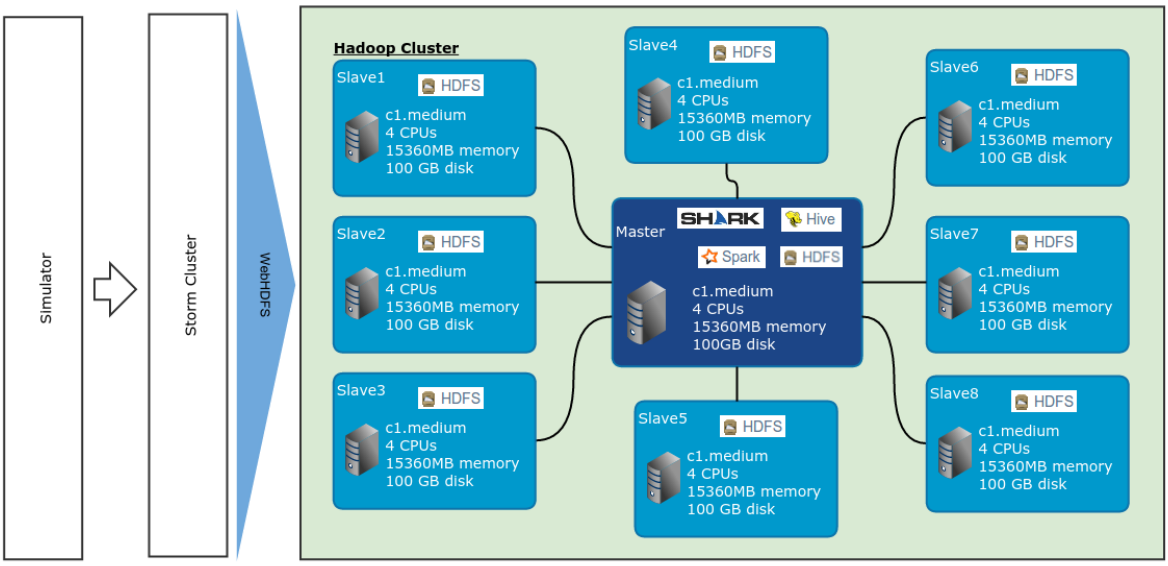


Figure 3.13: Data storage cluster(all together) specifications for queries scenario

- Tables with buckets is the hash partitioning within a Hive table partition and is not supported.
- It does not support different input format on partitions of a table.
- It does not support merge of multiple small files for query results avoiding overflowing the HDFS metadata.

Since Shark uses Spark, and it is a memory demanding engine, it is recommended to allocate 75% of memory from each node, in this case it will be allocated 6Gb of RAM. To allocate memory for Shark/Spark, it is necessary to configure it in *conf/shark-env.sh* and define:

```
export SPARK_MEM=6g
```

Raw table

For the scenario where the data is all grouped together in *ne_counter_tables*, the full query search performs as seen in the figure 3.14. The query are executed against samples of 1Gb, 10Gb and 100Gb using Hive and Shark. The result is an average of three executions on the same sample.

Shark optimized the map-reduce execution reducing the latency in the DAG and optimizing execution times. As it can be seen in figure 3.14 Shark performs faster queries comparing to Hive. For 1Gb of total data case a 74% improvement is achieved, for 10Gb an improvement of 25% and for the total of 100Gb of data the improvement was 46%. In addition, figure 3.14 suggests that in Shark the execution times have a more linear behavior through the amount of data.

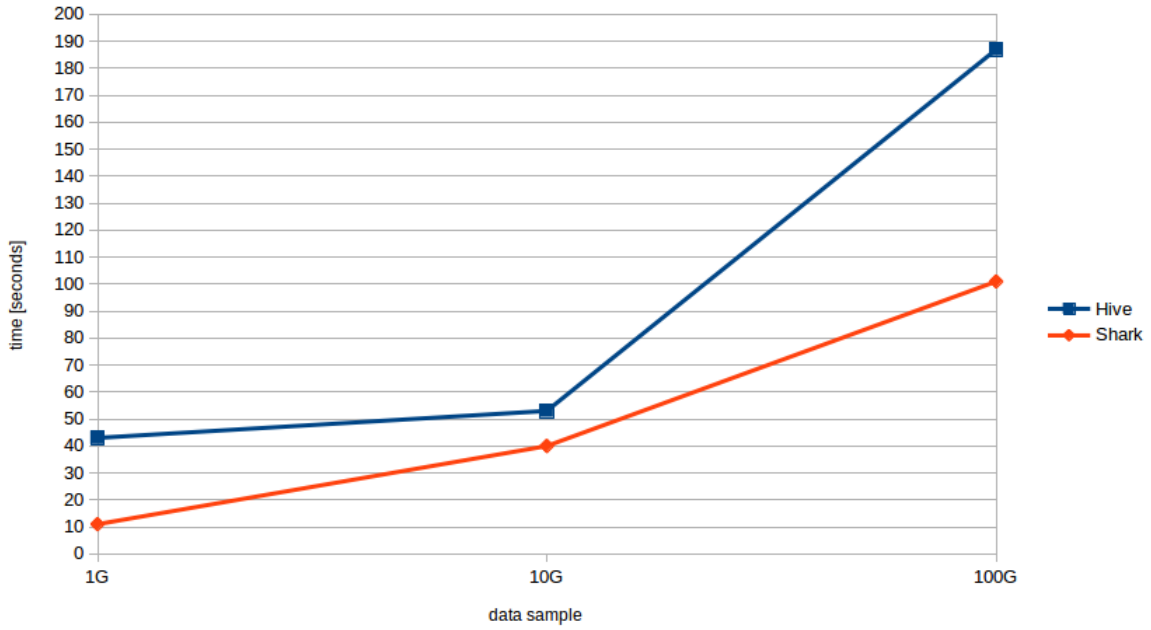


Figure 3.14: Hive vs Shark in ne_counter_tables

Aggregation

The scenario of aggregation tables is made by month. The sample loaded to aggregation tables is based on the 10Gb of total data and it is also saved as RCFiles with Snappy compression. Since the data is loaded from the *ne_counter_tables*, the insertion of data is executed as shown in the next example:

```
FROM ne_counters_table td

INSERT INTO TABLE tecnology1_raw
PARTITION (month)

SELECT
    day(p.period_start_time) AS period_start_time,
    month(p.period_start_time) AS month,
    co.parent_id AS plmn_gid,
    sum(p.{COUNTER}) AS {COUNTER}
    sum(p.{COUNTER}) AS {COUNTER}
    ...
FROM
    tecnology1_raw p
WHERE
    year={year} AND month={month} AND p.id={id}
GROUP BY
    year, month, p.id;
```

After the insertion of data into the aggregation tables, the query executed over the new data is a full search on that data. With this scenario is showed how the aggregation of data can improve query times, seeing the different times obtained in Hive and Shark(figure 3.15).

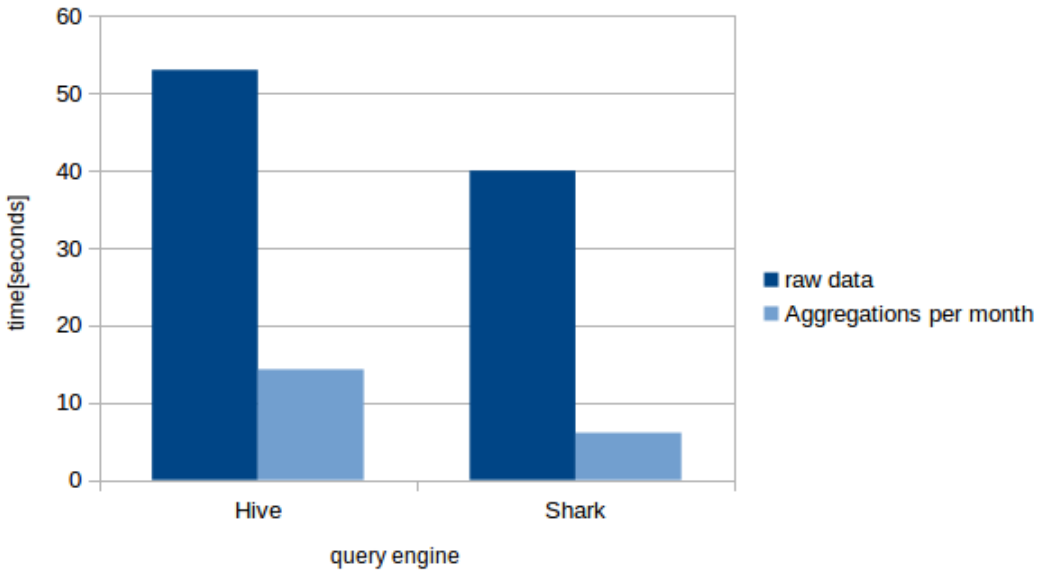


Figure 3.15: Aggregation scenario with Hive and Shark

In this scenario, as figure 3.15 shows, is compared the performance against the *ne_counter_tables* and the aggregation table. Shark once more, display better results in the aggregation table comparing with Hive. For Hive scenario a 73% improvement is achieved and in Shark a 85% improvement is achieved with the aggregation tables. In the aggregation tables a 57% improvement is achieved by Shark comparing with Hive.

Joins

For this scenario the aggregation tables are used to simulate joins over the data. Joins are very expensive in execution times and must be avoided joins in *reduce* tasks, especially in Hive. Because of being slower, Hive has a property that allows this definition:

```
SET hive.auto.convert.join=true;
```

In some versions, it is already defined as true, but there are some buggy versions. So it is better to always define if necessary. The test for joins will be made with this *true* and *false* property for Hive. The objective of the test is to verify how much influence the joins have in the query result times(3.16). The joins will be added and validated one by one, next is an example with all joins:

```
SELECT
  day(p.period_start_time) AS period_start_time,
  a.counter
  b.counter
  c.counter
  ...
FROM (
    SELECT id, period_start_time, sum(COUNTER1) AS counter FROM tecnology1_raw
    WHERE id = {id} GROUP BY id, period_start_time
) a
```

```
JOIN (
    SELECT period_start_time, sum(COUNTER1) AS counter FROM tecnology2_raw
    WHERE id = {id} GROUP BY period_start_time
) b
ON a.period_start_time = b.period_start_time

JOIN (
    SELECT period_start_time, sum(COUNTER1) AS counter FROM tecnology3_raw
    WHERE id = {id} GROUP BY period_start_time
) c
ON a.period_start_time = b.period_start_time
...
;
```

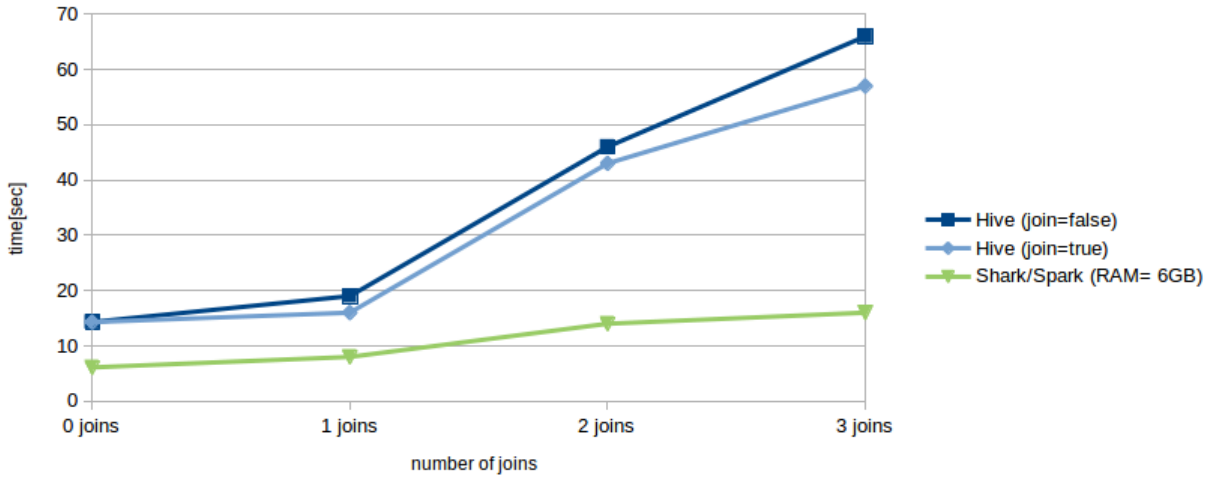


Figure 3.16: Scenario with Joins in Hive and Shark

As verified, Hive stills gets slower than Shark, but with the property of joins defined as *true* Hive can get better results along the increment of joins. Shark optimized the joins using PDE, reducing the latency of the normal DAG, which translates HQL to map-reduce jobs and, as seen in the previous example, it can perform better results with joins. Hive shows a increase of 78% on the time response without the join optimization and a 75% increase on time response with optimization property for the 3 joins comparing with to the query without joins. Shark shows a increase of 62% for 3 joins comparing to the query without joins. Shark shows a 72% improvement on joins comparing to Hive using the optimization property.

3.4 Hadoop Cluster specifications

In this section it is defined the resources to a Hadoop cluster based on some previous values obtained. In Hadoop, each node in the cluster must be configured according to its purpose. This is a process that, even when the system is up, requires analysis over time. It will always depend on the size of the data, the size of the blocks created and the type of processing needed to get a Map-Reduce job done. In the next sections it is defined the size of storage needed for the cluster, for plain text and RCFiles. The requirements for the Master and Slaves with the specifications for each case.

Data size

Starting with the space required for the data streamed and based on the sample of 1Gb of data, the number of records is 2.7 million in the raw data stored. One of the specifications for the cluster is to consume 1 billion counters in an hour and the number of columns corresponding to counters has an average of 96 columns, on the raw tables.

The next storage plan will be made for plain text files with no compression:

Formula	**Result**
$size\_per\_sample = \frac{sample\_size * size\_of\_records}{number\_of\_records}$	370.37 bytes; if $sample\_size$ equals 1
$counter\_size = \frac{size\_per\_sample}{number\_of\_counter\_columns}$	3.85 bytes; corresponding to 1 counter
$storage\_hour = counter\_size \times number\_counters\_per\_hour$	3.85 Gb; for 1 billion counters
$storage\_day = storage\_hour \times 24$	92.4 Gb; for 1 billion counters
$storage\_month = storage\_day \times 31$	2.864 Tb; for 1 billion counters
$storage\_year = storage\_day \times 365$	33.726 Tb; for 1 billion counters
$storage\_2years = storage\_year \times 2$	67.452 Tb; for 1 billion counters

Table 3.2: Text files space used for a plan of 2 years

With this type of data, it is possible to define the number of nodes needed to store text files in HDFS, on disks of 1 Tb:

Formula	**Result**	**Description**
Average daily ingest rate	92.4 Gb	
Replication factor	3	copies of each block
Daily raw consumption	277,2 Gb	$Ingest \times replication$
Node raw storage	1 Tb	8×100

MapReduce temp space reserve	25%	
Node-usable raw storage	750Gb	Node raw storage - MapReduce reserve
1 year (flat growth)	135 Nodes	$\frac{Ingest \times replication \times 365}{Node\_usable\_raw\_storage}$
1 year (5% growth per month)	276 Nodes	$(\frac{Ingest \times 365}{12}\%5) + 1\_year\_flat$
1 year (10% growth per month)	416 Nodes	$(\frac{Ingest \times 365}{12}\%10) + 1\_year\_flat$
2 year (flat growth)	270 Nodes	
2 year (5% growth per month)	552 Nodes	
2 year (10% growth per month)	832 Nodes	

Table 3.3: Growth plan for text files space

As it can bee seen, the plain text files demands several nodes because of the space they use. Next it is presented a storage plan for RCFiles with snappy compression:

Formula	**Result**
$size\_per\_sample = \frac{sample\_size * size\_of\_records}{number\_of\_records}$	22.2 bytes; if $sample\_size$ equals 1
$counter\_size = \frac{size\_per\_sample}{number\_of\_counter\_columns}$	0.23 bytes; corresponding to 1 counter
$storage\_hour = counter\_size \times number\_counters\_per\_hour$	230 Mb; for 1 billion counters
$storage\_day = storage\_hour \times 24$	5.520 Gb; for 1 billion counters
$storage\_month = storage\_day \times 31$	171.12 Gb; for 1 billion counters
$storage\_year = storage\_day \times 365$	2 Tb; for 1 billion counters

$storage\_2years = storage\_year \times 2$	4 Tb; for 1 billion counters

Table 3.4: RCFile with snappy compression space used

With this type of data it is possible to define the number of cluster needed for RCFiles in HDFS, on disks of 1 Tb:

Formula	**Result**	**Description**
Average daily ingest rate	5.520 Gb	
Replication factor	3	copies of each block
Daily raw consumption	16,56 Gb	$Ingest \times replication$
Node raw storage	1 Tb	1 x 1 Tb8 $\times$ 100
MapReduce temp space reserve	25%	
Node-usable raw storage	750 Gb	Node raw storage - MapReduce reserve
1 year (flat growth)	8 Nodes	$\frac{Ingest \times replication \times 365}{Node\_usable\_raw\_storage}$
1 year (5% growth per month)	17 Nodes	$(\frac{Ingest \times 365}{12}\%5) + 1\_year\_flat$
1 year (10% growth per month)	25 Nodes	$(\frac{Ingest \times 365}{12}\%10) + 1\_year\_flat$
2 year (flat growth)	16 Nodes	
2 year (5% growth per month)	34 Nodes	
2 year (10% growth per month)	50 Nodes	

Table 3.5: Growth plan for RCFile with snappy compression space

Comparing with plain text, RCFiles with compression are a good solution demanding less nodes to the cluster. Next, it is analyzed the configurations that are needed for the Master node.

Master

As seen in previous sections, the Name Node and the jobtracker are installed in the master node. They both use memory, the Name Node to store metadata about the HDFS cluster and the jobtracker to keep metadata information about the last 100 (by default) jobs executed on the cluster. These definitions will be equally necessary to the Secondary Name Node.

In order to define the amount of memory in the Name Node and Secondary Name Node, follow a rule of thumb considering that a Name Node needs about 1GB for 1 million blocks[14]. The following equations are used:

$$metadata\_memory = \frac{size\ of\ totalsize\ in\ Mb}{size\ of\ blocks} = \frac{750000 * 8}{64} < 1 million blocks \tag{3.1}$$

Since the 1 million blocks is not achieved, the default value will be 1 Gb.

$$namenode\_memory = metadata\_memory + 2Gb\ for\ Name\ node\ process + 4GB\ for\ OS = 7Gb \tag{3.2}$$

In terms of cores, on both Name Node and Secondary Name Node, four physical cores running at 2Ghz with Hyper-threading should be reasonable[14]. And since the Name Node uses disk to store the metadata and logs 1 Tb of the disk should be enough.

Jobtracker memory requirements can not be calculated from the cluster size. Small clusters that handle many jobs, or jobs with many tasks, could require more memory than expected. This is not easy to predict, because the variation in the number of tasks can be much greater than the metadata in the Name Node, from file to file. It should be assigned, as a precaution, as much memory as possible to the Master and collect data from the jobtracker over time, with the use of the cluster, to achieve the optimum definition of memory size need.

The common configuration used, for small clusters, is:

- a dual quad-core 2.6 Ghz CPU,
- 64 GB of DDR3 RAM,
- dual 1 Gb Ethernet NICs,
- a SAS drive controller,
- at least two SATA II drives in a JBOD configuration,
- dual power supplies,

In some cases the storage device for Name Node are with RAID 10 and the OS with RAID 1 for high availability (the secondary Name Node has the same configurations)[14]. Next, it is analyzed the configurations needed for the Slaves.

Slaves

The Data Nodes and the tasktracker run in the slaves. Given that each slave node in a cluster is responsible for both storage and computation, it is important to ensure not only that there is enough storage capacity, but also that the slave has the CPU and memory to process those data.

The memory is determined depending on the profile of jobs which runs on it. The common defined values for I/O bound jobs is between 2GB and 4GB per physical core and is between 6GB and 8GB per physical core for CPU bound jobs[14].

Defining a quad-core as a physical core, to determine the Data Node memory for I/O bound profile:

$$2Gb \times 4\ physical\ cores + 2Gb\ Data\ Process + 2Gb\ Tasktracker\ process + 4Gb\ OS = 16Gb \quad (3.3)$$

The data Node memory for CPU bound profile:

$$6Gb \times 4\ physical\ cores + 2Gb\ Data\ Process + 2Gb\ Tasktracker\ process + 4Gb\ OS = 32Gb \quad (3.4)$$

As seen, the slaves should have, as common configuration used for small clusters, a quad-core 2.9 Ghz, 48 Gb of memory and dual 1 Gb Ethernet NICs[14].

Final definition

Next is presented the definition for the Hadoop Cluster based on previous sections:

Master - Name Node and Jobtracker	
CPU	dual quad-core 2.6 Ghz
Memory	64 GB
DISK	1 Tb
Network Controller	1 Gb Ethernet NICs
Slave - Data Nodes and Tasktrackers	
CPU	quad-core 2.9 Ghz
Memory	48 Gb
DISK	1T Gb
Network Controller	2 x 1 Gb Ethernet NICs

Table 3.6: Specifications for an Hadoop clusters (small)

In case of a configuration of $2 \times 1Tb$ of disks or more, it should always be in a Just a bunch of disks (JBOD) configuration[14].

Chapter 4

Conclusion and future work

4.1 Conclusion

In this evaluation scenario, there were assessed some of the existing solutions to process large sets of data, such as Hive and Shark, over an Hadoop cluster. Furthermore, over the tested solutions, there were evaluated some techniques, such as storing and SQL like queries. Those solutions fit well on large data sets, were it is demanded lots of processing power, using the distribution of work by several machines. However, in terms of low latency or simple and small reports, these Hadoop solutions are not a very good option. Shark offers, in this case, a better response, but never near the response of an RDBM for aggregated data in time, for example. This is an assumption based on the know-how of RDBMs behavior, since it was not possible to have resources to test the same samples in a RDBM. The best fit to store the data is the use of some compression technique such as Snappy, and the use of sequence files or RCFiles, increasing the performance of the queries and storage.

To use this type of solutions, knowledge of Linux systems is necessary. In some cases, these solutions can be unstable and demand some configurations for fail-fast techniques, preventing data loss or performance degradations. Since most of the solutions are bash based, the installations imply hard work and the maintenance of the cluster demands some monitoring applications, such as Ganglia or Nagios. The use of an installation manager, such as Cloudera, to reduce the complexity of the installation and configuration is advised.

The solution with Hive is best suited for data warehousing, where large data sets need to be queried and fast response times are not required. Since horizontal scaling is possible, it can handle data sets that contain hundreds of millions of records, and that can grow over time in a cost-effective way. Hive presents an approximation of SQL dialect, but, over the continuing releases, it has been closer to the SQL standards. This makes Hive SQL engine an adopted option by other solutions. The schema and queries can be optimized, but Hive is insufficient when a latency lower then a second is necessary.

The solution with Shark/Spark improves the queries time, using Hive's metadata and SQL dialect. It was proved that both solutions can coexist without the need to refactor queries. However, it is necessary to take into consideration the configurations for the memory used by Shark/Spark, assigning as much memory as practically possible for the jobs. As seen in the presented performance tests, and with the schema defined, it is possible to obtain a low latency query results. Nevertheless, with the proper pre-computation and organization of the data it could be possible to obtain near second latency, in some cases.

4.2 Future work

Since the cluster and the storage used was in a virtual environment, the resource handling are not controlled only by Hadoop affecting the executions. An evaluation scenario over physical machines with JBOD configuration should be done. This type of environment should improve Hadoop performance.

It also should be analyzed the scenario of a replication factor of more then three nodes, verifying if affects the performance of query responses.

Over new releases of Hive, it was given support for the Optimized Row Columnar (ORC) file. This type of files provides a highly efficient way to store data, and it is designed to solve some limitations of other file formats. Using ORC files should improve performance for reading, writing and processing data.

One of the evaluations was to verify if it was possible to use SQL in Big Data solutions. However, for that, it was created a similar schema representation of an RDBM engine used. This approach only allows showing if the queries can be similar, but a denormalization should be made to improve performance over querying data using the best features that solutions such as Hive or Shark can offer. To denormalize the data, Storm topology should be analyzed and re-designed to transform data for a schema better fitted to Big Data solutions.

It was seen that, for low latency or for simple and small reports, the solutions evaluated were not the best fit options. It should be tested a hybrid solution, joining RDBMs with aggregations for low latency queries, and Hadoop for big data sets queries. One of the options is Pesto, allowing querying data over several sources. This type of distributed query engine allows querying data over several sources, such as RDBMs, Hive or Cassandra, for example. This type of evaluation would show if it was possible to have the best of both worlds.

Bibliography

[1] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24. ACM, New York, NY, USA, 2013.

[2] Yin Huai Zheng Shao Namit Jain Xiaodong Zhang Zhiwei Xu Yongqiang He, Rubao Lee. rcfiles. In *RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems*. ICDE Conference, 2011.

[3] Apache hive. http://hive.apache.org/. [Online; accessed 29-Set-2014].

[4] Inc O'Reilly Media. *Big Data Now: 2012 Edition.* O'Reilly Media, Inc, 2012.

[5] Edd Dumbill. *Planning for Big Data - The CIO's handbook to the changing data landscape.* O'Reilly Media, Inc, 2012.

[6] Martin Fowler Pramod J. Sadalage. *NoSQL Distilled - A brief Guide to the emerging world of polyglot persistence.* Addison-Wesley, 2013.

[7] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. volume 51, pages 107–113. ACM, New York, NY, USA, January 2008.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43. ACM, New York, NY, USA, 2003.

[10] A. Thusoo, J.S. Sarma, N. Jain, Zheng Shao, P. Chakka, Ning Zhang, S. Antony, Hao Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. March 2010.

[11] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[12] Apache hadoop. http://hadoop.apache.org/. [Online; accessed 29-Set-2014].

[13] Neil Spencer. How much data is created every minute? http://www.visualnews.com/2012/06/19/how-much-data-created-every-minute/, 2012. [Online; accessed 02-Jan-2014].

[14] Jim R. Wilson Eric Redmond. *Seven Databases in Seven Weeks - A guide to modern databases and NoSQL movement.* Pragmatic Bookshelf, 2012.

[15] Eric Sammer. *Hadoop Operations.* O'Reilly Media, Inc, 2012.

[16] Shark. http://shark.cs.berkeley.edu. [Offline (subsumed by Spark SQL); accessed 30-Ago-2014].

[17] Udf functions. https://github.com/nexr/. [Online; accessed 15-Ago-2014].

[18] Tathagata Das Ankur Dave Justin Ma Murphy McCauley Michael J. Franklin Scott Shenker Ion Stoica Matei Zaharia, Mosharaf Chowdhury. Spark. In *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.* University of California, Berkeley, 2012.