Notes de cours
TC6 : Science des données pour le Big Data

Eleonore Bartenlian, Adrien Pavao

November 2017

Table des matières

Introduction

La Big Data concerne le stockage, l'accès, le traitement et l'analyse de **très grande quantité** d'information. Depuis 2006, le Big Data a connu un énorme développement d'activité et de recherche.

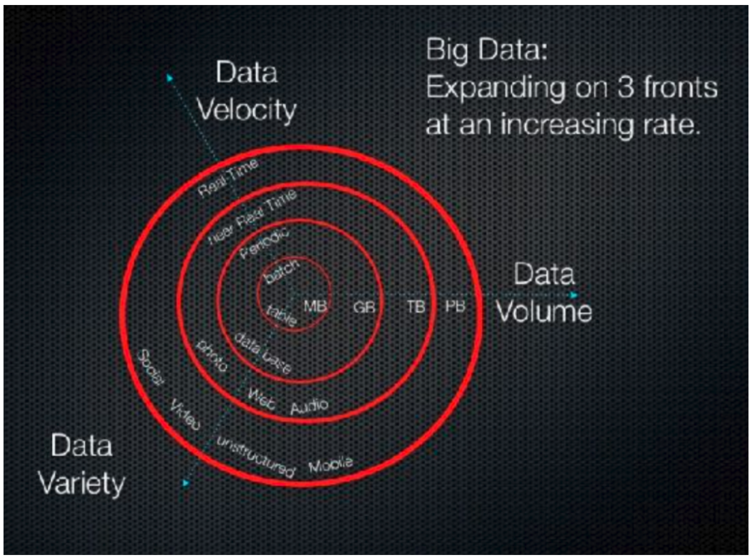


FIGURE 1 – Les trois V du Big Data

Trois joueurs principaux en data et analytics

— **Génération et collection de données :** La source et la plateforme où les données ont été initialement capturées.
— **L'aggrégation de données :** Méthodes et plateforme permettant de combiner les données de différentes sources.
— **L'analyse de données :** Ce que l'on produit à partir des données.

1 Cours 1 : Systems, Paradigms, Algorithms

Enabled thanks to : increasing of storage capacity, increasing of processing power, availability of massive amounts of data.
And **new computing frameworks and paradigms.** Traditional RDMS (SGBD in french) cannot cope with the 3 Vs : need of cluster of commodity computers for distribution and parallelisms ; file systems to distributs huge file storage and manipulation ; programming paradigms to speed up SW development ; data models to cope with Variability ; scalability, flexibiliby and robustness : both at hardware and software levels.

Big Data is still disruptive ; we are only at the beginning of the story. Increase rate of data volume will accelerate in a wide class of crucial tasks of both companies and public administration. There has been important recent advances on techniques and algorithms relying on massive data : ML ; Large scale SQL, Semantic Web and Graph analytics ; Combined real-time analytics of data streams and warehouse.

1.1 Hadoop

Hadoop stack : Applications run naively in hadoop : pig (script), hive(sql), hbase (nosql), accumulo (nosql), storm (stream), soir, spark(in memory), cascading (java), others. All rely on YARN data operating system and HDFS.

1.2 The Hadoop Distributed Filesystem : HDFS

This is the storage component of Hadoop.

— Highly scalable
— Distributed
— Load-balanced (task repartition)
— Portable
— Fault-tolerant (with built-in redundancy at the software level)

It provides a layer for storing Big Data in a traditional, hierarchical file organization of directories and files. It has been designed to run on commodity hardware.

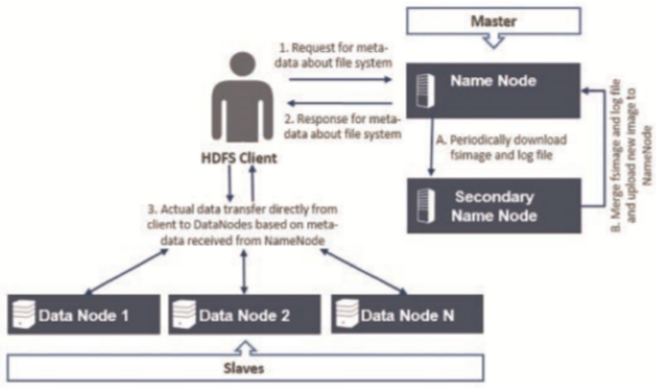
1.3 Main assumptions behind its design

— Horizontal scalability
— Fault tolerance
— Capability to run on commodity hardware
— Write once, read many times
— Data locality
— File system namespace, relying on traditional hierarchical file organization.
— Streaming access and high throughput : reading the data in the fastest possible way (instead of focusing on the speed of the data write) ; reading data from multiple nodes, in parallel

1.4 Typical cluster architecture

— One or more racks.
— Typically 30 to 40 node servers per rack with a 1GB switch for the rack.
— The cluster switch is normally 1GB or 10GB.
— Architecture of single node server can vary. More disk capacity and network throughput for operations like indexing, grouping, data importing/ exporting, data transformation. More CPU capacity for operations like clustering/classification, NLP, feature extraction

1.5 Name nodes and Data nodes


Master
1. Request for meta-data about file system
Name Node
2. Response for meta-data about file system
HDFS Client
A. Periodically download fsimage and log file
Secondary Name Node
B. Merge fsimage and log file and upload new image to NameNode
3. Actual data transfer directly from client to DataNodes based on meta-data received from NameNode
Data Node 1
Data Node 2
Data Node N
Slaves

— Name node = a server node running the Name node daemon (service in Windows)
— Data node = a server node running the data node daemon.
— To store a file, HDFS client asks meta information to the Name node
— The client then interacts only with Data Nodes
— It splits the file into one or more chunks or blocks (64 MB by default, configurable

— And send them to a set of Data Nodes slaves previously indicated by the Name node
— Each block is replicated n times (n=3 by default, configurable)

HDFS has a master/slave architecture.

An HDFS cluster : single NameNode, a master server that manages the file system namespace and regulates access to files by clients + DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on.

HDFS allows user data to be stored in files. A file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.

The NameNode and DataNode are pieces of software designed to run on commodity machines. These machines typically run a GNU/Linux operating system (OS). HDFS is built using the Java language ; any machine that supports Java can run the NameNode or the DataNode software. Usage of the highly portable Java language means that HDFS can be deployed on a wide range of machines. A typical deployment has a dedicated machine that runs only the NameNode software. Each of the other machines in the cluster runs one instance of the DataNode software. The architecture does not preclude running multiple DataNodes on the same machine but in a real deployment that is rarely the case.

The existence of a single NameNode in a cluster greatly simplifies the architecture of the system. The NameNode is the arbitrator and repository for all HDFS metadata. The system is designed in such a way that user data never flows through the NameNode.

The File System Namespace

HDFS supports a traditional hierarchical file organization. A user or an application can create directories and store files inside these directories. The file system namespace hierarchy is similar to most other existing file systems ; one can create and remove files, move a file from one directory to another, or rename a file. HDFS does not yet implement user quotas. HDFS does not support hard links or soft links. However, the HDFS architecture does not preclude implementing these features.

The NameNode maintains the file system namespace. Any change to the file system namespace or its properties is recorded by the NameNode. An application can specify the number of replicas of a file that should be maintained by HDFS. The number of copies of a file is called the replication factor of that file. This information is stored by the NameNode.

Data Replication

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks ; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once and have strictly one writer at any time.

The NameNode makes all decisions regarding replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode.

Replica Placement : The First Baby Steps

The placement of replicas is critical to HDFS reliability and performance. Optimizing replica placement distinguishes HDFS from most other distributed file systems. This is a feature that needs lots of tuning and experience. The purpose of a rack-aware replica placement policy is to improve data reliability, availability, and network bandwidth utilization. The current implementation for the replica placement policy is a first effort in this direction. The short-term goals of implementing this policy are to validate it on production systems, learn more about its behavior, and build a foundation to test and research more sophisticated policies.

Large HDFS instances run on a cluster of computers that commonly spread across many racks. Communication between two nodes in different racks has to go through switches. In most cases, network bandwidth between machines in the same rack is greater than network bandwidth between machines in different racks.

The NameNode determines the rack id each DataNode belongs to via the process outlined in Hadoop Rack Awareness. A simple but non-optimal policy is to place replicas on unique racks. This prevents losing data when an entire rack fails and allows use of bandwidth from multiple racks when reading data. This policy evenly distributes replicas in the cluster which makes it easy to balance load on component failure. However, this policy increases the cost of writes because a write needs to transfer blocks to multiple racks.

For the common case, when the replication factor is three, HDFS's placement policy is to put one replica on one node in the local rack, another on a node in a different (remote) rack, and the last on a different node in the same remote rack. This policy cuts the inter-rack write traffic which generally improves write performance. The chance of rack failure is far less than that of node failure ; this policy does not impact data reliability and availability guarantees. However, it does reduce the aggregate network bandwidth used when reading data since a block is placed in only two unique racks rather than three. With this policy, the replicas of a file do not evenly distribute across the racks. One third of replicas are on one node, two thirds of replicas are on one rack, and the other third are evenly distributed across the remaining racks. This policy improves write performance without compromising data reliability or read performance.

The current, default replica placement policy described here is a work in progress.

Replica Selection

To minimize global bandwidth consumption and read latency, HDFS tries to satisfy a read request from a replica that is closest to the reader. If there exists a replica on the same rack as the reader node, then that replica is preferred to satisfy the read request. If angg/ HDFS cluster spans multiple data centers, then a replica that is resident in the local data center is preferred over any remote replica.

Safemode

On startup, the NameNode enters a special state called Safemode. Replication of data blocks does not occur when the NameNode is in the Safemode state. The NameNode receives Heartbeat and Blockreport messages from the DataNodes. A Blockreport contains the list of data blocks that a DataNode is hosting. Each block has a specified minimum number of replicas. A block is considered safely replicated when the minimum number of replicas of that data block has checked in with the NameNode. After a configurable percentage of safely replicated data blocks checks in with the NameNode (plus an additional 30 seconds), the NameNode exits the Safemode state. It then determines the list of data blocks (if any) that still have fewer than the specified number of replicas. The NameNode then replicates these blocks to other DataNodes.

The Persistence of File System Metadata

The HDFS namespace is stored by the NameNode. The NameNode uses a transaction log called the EditLog to persistently record every change that occurs to file system metadata. For example, creating a new file in HDFS causes the NameNode to insert a record into the EditLog indicating this. Similarly, changing the replication factor of a file causes a new record to be inserted into the EditLog. The NameNode uses a file in its local host OS file system to store the EditLog. The entire file system namespace, including the mapping of blocks to files and file system properties, is stored in a file called the FsImage. The FsImage is stored as a file in the NameNode's local file system too.

The NameNode keeps an image of the entire file system namespace and file Blockmap in memory. This key metadata item is designed to be compact, such that a NameNode with 4 GB of RAM is plenty to support a huge number of files and directories. When the NameNode starts up, it reads the FsImage and EditLog from disk, applies all the transactions from the EditLog to the in-memory representation of the FsImage, and flushes out this new version into a new FsImage on disk. It can then truncate the old EditLog because its transactions have been applied to the persistent FsImage. This process is called a checkpoint. In the current implementation, a checkpoint only occurs when the NameNode starts up. Work is in progress to support periodic checkpointing in the near future.

The DataNode stores HDFS data in files in its local file system. The DataNode has no knowledge about HDFS files. It stores each block of HDFS data in a separate file in its local file system. The DataNode does not create all files in the same directory. Instead, it uses a heuristic to determine the optimal number of files per directory and creates subdirectories appropriately. It is not optimal to create all local files in the same directory because the local file system might not be able to efficiently support a huge number of files in a single directory. When a DataNode starts up, it scans through its local file system, generates a list of all HDFS data blocks that correspond to each of these local files and sends this report to the NameNode : this is the Blockreport.

The Communication Protocols

All HDFS communication protocols are layered on top of the TCP/IP protocol. A client establishes a connection to a configurable TCP port on the NameNode machine. It talks the ClientProtocol with the NameNode. The DataNodes talk to the NameNode using the DataNode Protocol. A Remote Procedure Call (RPC) abstraction wraps both the Client Protocol and the DataNode Protocol. By design, the NameNode never initiates any RPCs. Instead, it only responds to RPC requests issued by DataNodes or clients.

Robustness

The primary objective of HDFS is to store data reliably even in the presence of failures. The three common types of failures are NameNode failures, DataNode failures and network partitions.

Data Disk Failure, Heartbeats and Re-Replication

Each DataNode sends a Heartbeat message to the NameNode periodically. A network partition can cause a subset of DataNodes to lose connectivity with the NameNode. The NameNode detects this condition by the absence of a Heartbeat message. The NameNode marks DataNodes without recent Heartbeats as dead and does not forward any new IO requests to them. Any data that was registered to a dead DataNode is not available to HDFS any more. DataNode death may cause the replication factor of some blocks to fall below their specified value. The NameNode constantly tracks which blocks need to be replicated and initiates replication whenever necessary. The necessity for re-replication may arise due to many reasons : a DataNode may become unavailable, a replica may become corrupted, a hard disk on a DataNode may fail, or the replication factor of a file may be increased.

Cluster Rebalancing

The HDFS architecture is compatible with data rebalancing schemes. A scheme might automatically move data from one DataNode to another if the free space on a DataNode falls below a certain threshold. In the event of a sudden high demand for a particular file, a scheme might dynamically create additional replicas and rebalance other data in the cluster. These types of data rebalancing schemes are not yet implemented.

Data Integrity

It is possible that a block of data fetched from a DataNode arrives corrupted. This corruption can occur because of faults in a storage device, network faults, or buggy software. The HDFS client software implements checksum checking on the contents of HDFS files. When a client creates an HDFS file, it computes a checksum of each block of the file and stores these checksums in a separate hidden file in the same HDFS namespace. When a client retrieves file contents it verifies that the data it received from each DataNode matches the checksum stored in the associated checksum file. If not, then the client can opt to retrieve that block from another DataNode that has a replica of that block.

Metadata Disk Failure

The FsImage and the EditLog are central data structures of HDFS. A corruption of these files can cause the HDFS instance to be non-functional. For this reason, the NameNode can be configured to support maintaining multiple copies of the FsImage and EditLog. Any update to either the FsImage or EditLog causes each of the FsImages and EditLogs to get updated synchronously. This synchronous updating of multiple copies of the FsImage and EditLog may degrade the rate of namespace transactions per second that a NameNode can support. However, this degradation is acceptable because even though HDFS applications are very data intensive in nature, they are not metadata intensive. When a NameNode restarts, it selects the latest consistent FsImage and EditLog to use.

The NameNode machine is a single point of failure for an HDFS cluster. If the NameNode machine fails, manual intervention is necessary. Currently, automatic restart and failover of the NameNode software to another machine is not supported.

Snapshots

Snapshots support storing a copy of data at a particular instant of time. One usage of the snapshot feature may be to roll back a corrupted HDFS instance to a previously known good point in time. HDFS does not currently support snapshots but will in a future release.

Data Organization

Data Blocks

HDFS is designed to support very large files. Applications that are compatible with HDFS are those that deal with large data sets. These applications write their data only once but they read it one or more times and require these reads to be satisfied at streaming speeds. HDFS supports write-once-read-many semantics on files. A typical block size used by HDFS is 64 MB. Thus, an HDFS file is chopped up into 64 MB chunks, and if possible, each chunk will reside on a different DataNode.

Staging

A client request to create a file does not reach the NameNode immediately. In fact, initially the HDFS client caches the file data into a temporary local file. Application writes are transparently redirected to this temporary local file. When the local file accumulates data worth over one HDFS block size, the client contacts the NameNode. The NameNode inserts the file name into the file system hierarchy and allocates a data block for it. The NameNode responds to the client request with the identity of the DataNode and the destination data block. Then the client flushes the block of data from the local temporary file to the specified DataNode. When a file is closed, the remaining un-flushed data in the temporary local file is transferred to the DataNode. The client then tells the NameNode that the file is closed. At this point, the NameNode commits the file creation operation into a persistent store. If the NameNode dies before the file is closed, the file is lost.

The above approach has been adopted after careful consideration of target applications that run on HDFS. These applications need streaming writes to files. If a client writes to a remote file directly without any client side buffering, the network speed and the congestion in the network impacts throughput considerably. This approach is not without precedent. Earlier distributed file systems, e.g. AFS, have used client side caching to improve performance. A POSIX requirement has been relaxed to achieve higher performance of data uploads.

Replication Pipelining

When a client is writing data to an HDFS file, its data is first written to a local file as explained in the previous section. Suppose the HDFS file has a replication factor of three. When the local file accumulates a full block of user data, the client retrieves a list of DataNodes from the NameNode. This list contains the DataNodes that will host a replica of that block. The client then flushes the data block to the first DataNode. The first DataNode starts receiving the data in small portions (4 KB), writes each portion to its local repository and transfers that portion to the second DataNode in the list. The second DataNode, in turn starts receiving each portion of the data block, writes that portion to its repository and then flushes that portion to the third DataNode. Finally, the third DataNode writes the data to its local repository. Thus, a DataNode can be receiving data from the previous one in the pipeline and at the same time forwarding data to the next one in the pipeline. Thus, the data is pipelined from one DataNode to the next.

Accessibility

HDFS can be accessed from applications in many different ways. Natively, HDFS provides a Java API for applications to use. A C language wrapper for this Java API is also available. In addition, an HTTP browser can also be used to browse the files of an HDFS instance. Work is in progress to expose HDFS through the WebDAV protocol.

FS Shell

HDFS allows user data to be organized in the form of files and directories. It provides a command-line interface called FS shell that lets a user interact with the data in HDFS. The syntax of this command set is similar to other shells (e.g. bash, csh) that users are already familiar with. Here are some sample action/command pairs :

Create a directory named /foodir : bin/hadoop dfs -mkdir /foodir Remove a directory named /foodir : bin/hadoop dfs -rmr /foodir View the contents of a file named /foodir/myfile.txt : bin/hadoop dfs -cat /foodir/myfile.txt

DFSAdmin

The DFSAdmin command set is used for administering an HDFS cluster. These are commands that are used only by an HDFS administrator. Here are some sample action/command pairs :

Put the cluster in Safemode bin/hadoop : dfsadmin -safemode enter Generate a list of DataNodes : bin/hadoop dfsadmin -report Recommission or decommission DataNode(s) : bin/hadoop dfsadmin -refreshNodes

Browser Interface

A typical HDFS install configures a web server to expose the HDFS namespace through a configurable TCP port. This allows a user to navigate the HDFS namespace and view the contents of its files using a web browser.

Space Reclamation

File Deletes and Undeletes
When a file is deleted by a user or an application, it is not immediately removed from HDFS. Instead, HDFS first renames it to a file in the /trash directory. The file can be restored quickly as long as it remains in /trash. A file remains in /trash for a configurable amount of time. After the expiry of its life in /trash, the NameNode deletes the file from the HDFS namespace. The deletion of a file causes the blocks associated with the file to be freed. Note that there could be an appreciable time delay between the time a file is deleted by a user and the time of the corresponding increase in free space in HDFS.

A user can Undelete a file after deleting it as long as it remains in the /trash directory. If a user wants to undelete a file that he/she has deleted, he/she can navigate the /trash directory and retrieve the file. The /trash directory contains only the latest copy of the file that was deleted. The /trash directory is just like any other directory with one special feature : HDFS applies specified policies to automatically delete files from this directory. The current default policy is to delete files from /trash that are more than 6 hours old. In the future, this policy will be configurable through a well defined interface.

Decrease Replication Factor

When the replication factor of a file is reduced, the NameNode selects excess replicas that can be deleted. The next Heartbeat transfers this information to the DataNode. The DataNode then removes the corresponding blocks and the corresponding free space appears in the cluster. Once again, there might be a time delay between the completion of the setReplication API call and the appearance of free space in the cluster.

- NameNode is the centerpiece of HDFS.
- NameNode is also known as the Master
- NameNode only stores the metadata of HDFS – the directory tree of all files in the file system, and tracks the files across the cluster.
- NameNode does not store the actual data or the dataset. The data itself is actually stored in the DataNodes.
- NameNode knows the list of the blocks and its location for any given file in HDFS. With this information
- NameNode knows how to construct the file from blocks.
- NameNode is so critical to HDFS and when the NameNode is down, HDFS/Hadoop cluster is inaccessible and considered down.
- NameNode is a single point of failure in Hadoop cluster.
- NameNode is usually configured with a lot of memory (RAM). Because the block locations are help in main memory.

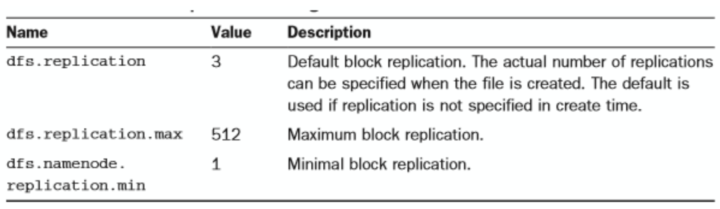
- DataNode is responsible for storing the actual data in HDFS.
- DataNode is also known as the Slave
- NameNode and DataNode are in constant communication.
- When a DataNode starts up it announce itself to the NameNode along with the list of blocks it is responsible for.
- When a DataNode is down, it does not affect the availability of data or the cluster. NameNode will arrange for replication for the blocks managed by the DataNode that is not available.
- DataNode is usually configured with a lot of hard disk space. Because the actual data is stored in the DataNode.

1.6 File split

Plenty of configuration files. For exmaple, block size is set in the hdfs-site.xml.

1.7 Block placement and replication

By default each block is stored 3 times in three different Data nodes, for fault tolerance. When a file is created, an application can specify the number of replicas of each block of the file that HDFS must maintain. The upper bound dfs.replcation.max must be respected. Settings in hdfs-site.xml.



Name	Value	Description
`dfs.replication`	3	Default block replication. The actual number of replications can be specified when the file is created. The default is used if replication is not specified in create time.
`dfs.replication.max`	512	Maximum block replication.
`dfs.namenode.replication.min`	1	Minimal block replication.

1.8 Block placement and replication

For robustness, the ideal approach would be to store block replicas in different racks. For efficiency, it is better to store all replicas in the same rack. Balanced Hadoop approach : store one block on the client Data node where the file originates (or a not too busy node chosen by the Name node) and the two other blocks in a different rack (if any). This requires to configure the cluster for RackAwareness.

1.9 Heartbeats

All data nodes periodically (each 3 seconds) send heartbeat signals to the name node. They contain crucial information about stored blocks, percentage of used storage, current communication load, etc. Hearth beat contents are crucial for the Name Node to build and maintain metadata information. The NameNode does not directly call the Data Nodes. It uses replies to heartbeats to ask replication to other nodes, remove local block replicas, etc.

1.10 Node failure and replication

Assume Data node 4 stops working. This means that no heart beats is sent anymore. The Name node then instructs another living Data Node including blocks B and C to replicate them on other Data Nodes. Data transmission for B and C replication does not involve the Name Node.

1.11 Writing a file to HDFS

This can happen, for instance, by command-line or by means of a client program requesting the writing operation.

First step :

On demande les méta données au name node. Il répond avec les méta données : le namenode dit le client où mettre chaque bloc du chifier en fonctino du 'replication factor' et du contenu des datanodes

Second step, the first block is sent to Data Nodes :

En fonction des méta données reçues, le client HDFS écrit directement les données ou les bloques dans les datanodes assignés.

Third step, Data Nodes notify the Name Node about the stored block replicas : les datadones envoient des confirmations que les blocs ont été reçus au namenode.

Fourth step, the first Data Node storing the block sends conformation to the client.

TODO : Faire schémas à la main

1.12 Reading a file

First step : le clien HDFS demande les méta données au namenode pour lire des données. Il répond en indiquant au client où les blocks du fichier voulu sont enregistrés et quels datanodes peuvent être contactés pour lire les données.

Second step : voilà voilà. Il demande les données aux datanodes qui les envoient TODO vérifier le schéma et pas clair.

1.13 Shell

Creating a directory : hdfs fs -mkdir /example/sampledata

Copying a directory to HDFS (from the local FS) : hdfs fs -copyFromLocal /apps/dist/examples/-data/gutenberg /example/sampledata

Listing content of a directory : hdfs fs -ls /example/sampledata

Copying a fie to FS : hdfs fs -copyToLocal /user/hadoop/filename /apps/dist/examples/data/-gutenberg

1.14 HDFS high availability

The presence of a single name node in Hadoop 1.0 caused many problems, as files could become corrupted, stopping the cluster working.

To overcome this limitation, Hadoop 2.x introduced a Secondary name node, with the following tasks : storing periodically a checkpoint of Name node information ; logging changes after the check point ; taking over in case of failures of the Name node (this requires merging the log and the check point).

Unfortunately this configuration has not been successful in production, so currently the HDFS High Availability (HA) configuration is normally adopted : An active Name Node plus a passive (stand-by) Name Node ; They stay synchronised : if the active Name Node fails the passive Name Node takes over ; See next possible configurations.

1.15 HDFS HA with shared storage

Separate server for storing the Name node image

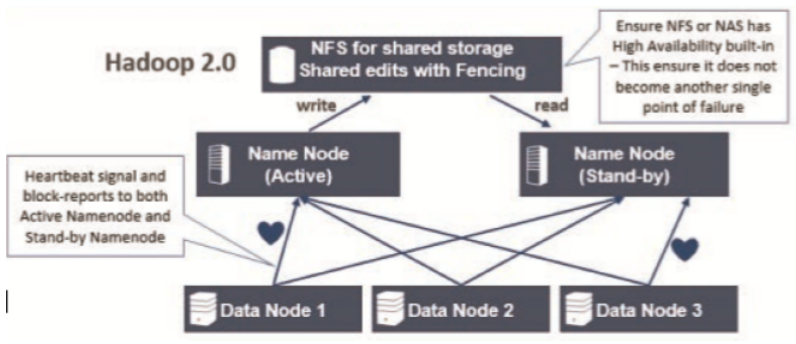
Fencing : active NN writes, passive NN reads

Data nodes send heartbeats to both name nodes

Passive NN is constantly updated

It is important that the server for passive NN is as powerful as that for the active NN

For this HA option configurations are needed.


Hadoop 2.0
NFS for shared storage
Shared edits with Fencing
Ensure NFS or NAS has High Availability built-in – This ensure it does not become another single point of failure
write
read
Name Node (Active)
Name Node (Stand-by)
Heartbeat signal and block-reports to both Active Namenode and Stand-by Namenode
Data Node 1
Data Node 2
Data Node 3

1.16 HA with quorum based storage

Frequently adopted.

It can be deployed on the Hadoop cluster itself

Multiple daemons called Journal nodes are accessed by both NN, for write and read as before

Each change to the active NN file system is logged to the majority of the JNs.

Only when a quorum of the JNs validate the writing the active NN validate the modification to the file system.

As JNs performs light operations (record append), they can run on the cluster nodes.

At least 3 JNs must run, preferably on different machines, in case of HW failures

TODO : ajouter schéma ?

1.17 MapReduce : main principles

Data model : data collections are represented in terms of collections of key-value pairs (k, v)

Paradigm model : a MapReduce algorithm (or job) consists of two functions Map and Reduce

A MapReduce algorithm takes as input a collection of key-value pairs and returns a collection of the same kind

The Map function is intended to be applied to each input pair (k,v) and for this pair returns a, possibly empty, list of pairs [(k1,v1), ... , (kn,vn)]

The Reduce function is applied on pairs of the form (k', [v1', ... , vn']) and for each such pair returns a list of key-value pairs that takes part of the final result.

The behaviour of (how the output is determined for) the Map and Reduce function is determined by the user

Processing model : given a MapReduce algorithm over an input I

— The **Map** phase consist of applying the Map function to each pair in I.
— The collection M of pairs resulted by the Map phase are submitted to a group-by process called shuffle-and-sort aiming at grouping by key the pairs in M, resulting into a collection of pairs of the form $(k', [v1', ..., vn'])$
— The **Reduce** phase consists of iterating the Reduce function application on each pair produced by shuffle-and-sort

1.18 Benefits

Simplicity in parallelism : Parallelism is implicit in the model : Map and Reduce functions can be applied in a shared-nothing parallelism fashion; MapReduce programs can be automatically parallelized over an Hadoop cluster

Fault tolerance : If a slave fails in computing a part of the Map or Reduce phase, the system detects this and re-assign the missing work to another slave in a transparent manner ; HDFS replication is crucial for this

Scalability : As the data load increases, parallelism level increases by adding slaves to the cluster

Data locality : Hadoop maximize proximity between the slave where data is stored and the slave where processing happens. ; In the Map phase, most of Mappers are executed on the nodes storing input blocks

1.19 WordCount

Input : a directory containing all the documents

Pair preparation : starting from documents, pairs (k,v) where k is unspecified and v is a text line of a document are prepared and passed to the Map phase.

Map : takes as input a couple (k,v) returns a pair (w,1) for each word w in v Shuffle&Sort : group all of pairs output by Map and produce pairs of the form (w, [1, . . . ,1])

Reduce : takes as input a pair (w, [1, . . . ,1]), sums all the 1's for w obtaining s, and outputs (w,s)

Pseudo code :

```
Map(k,v) : for each w in v, emit (w,1)

Reduce(k,v) : c=0; for x in v : c=c+1; emit(k,c)
```

1.20 Scalability issues

Ideal scaling characteristics : Twice the data, twice the running time ; Twice the resources, half the time

Difficult to achieve in practice : Synchronisation requires time ; Communication kills performance (networks is slow !)

Thus. . . minimise inter-node communication : Local aggregation can help : reduce size of Map phase output ; Use of combiners can help in this direction

1.21 Combiner

It is an additional function you are allowed to define and adopt in MapReduce. Its input has the shape of that of Reduce and its output has to be compatible with that of Map (*). Like Reduce it performs aggregation. Differently from Reduce it is run locally, on slaves executing Map. Goal : pre-aggregate Map output pairs in order to lower number of pairs to shuffle and sort (and to sent trough the network). Attention : it is up to Hadoop to decide whether a Mapper node runs a Combiner : so some of the Map nodes run the combiner, some do not ; this is why we need (*) above ; we will se when Combiner is triggered.

1.22 Pseudo code with combiner

```
map(k,v) : for each w in v: emit(w,1)

Combine (k,v) : c=0; for x in v: c=c+1; emit(k,c)

reduce(k,v) : c= 0; for x in v: c=c+x; emit(k,c)
```

TODO voir plus en détail dans le cours1 tout ça, shuffle sort...

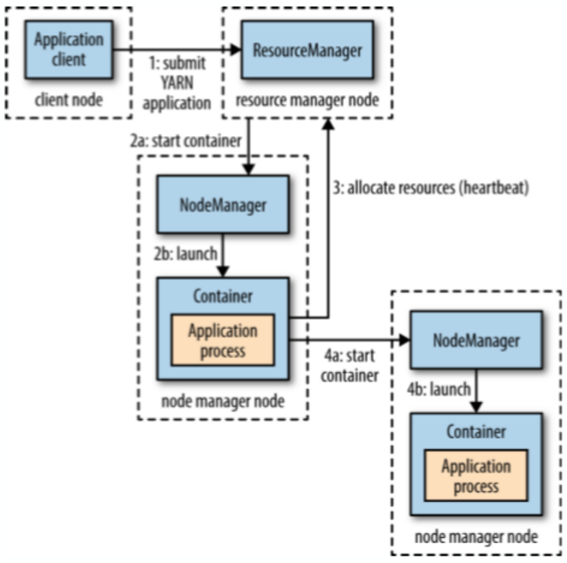
The main function of a Combiner is to summarize the map output records with the same key. The output (key-value collection) of the combiner will be sent over the network to the actual Reducer task as input.

1.23 YARN

Apache Hadoop YARN (Yet Another Resource Negotiator) is a cluster management technology.

YARN is a general purpose data operating systems. It accepts requests of task executions on the cluster and allocate resources for them. For instance YARN can accept requests for executing MR jobs and MPI programs on the same cluster. The set of resources for a task on a given node is called container and it includes given amounts of memory space and CPU power (cores). YARN keeps track of allocated resources in order to schedule container allocation for new requests. Containers can be demanded all in advance like for MR jobs and Spark tasks, or at run time

— Step 1 : a client contact the resource manager (running on the master node)
— Step 2.a : the resource manager then finds a node manager (running on a slave) that can launch and manage running of the application ; this is the application manage.
— Step 2.b : the node manager allocates the container indicated by the resource manager. The application runs within the container
— Step 3 : eventually, the application manager can request new containers to the resource manager
— Step 4 : a parallel container is then started after the acknowledge of the resource manager. Started container informs the application manager about their status upon request.
— Containers can be demanded all in advance like for Spark, or at run time (step 4)
— At the end of the process, the application managers informs the resource manager, which will kill the allocated containers.



1.24 YARN scheduling policies

Set in yarn-default.xml configuration file.

FIFO, for clusters with 'small' workloads

Capacity Scheduler (introduced by Yahoo ! for large clusters) (the administrator configures several queues with a predetermined fraction of the total resources (this ensures minimal resources for each queue) ; each queue is FIFO, and has strict Access Control Policies to determine what user can submit to what queue ; configuration can be changed dynamically, when feedback from the Node managers is available, more starved queues can have more resources ; work best when there is knowledge about the kind of workload, this allows for setting minima resources for queues).

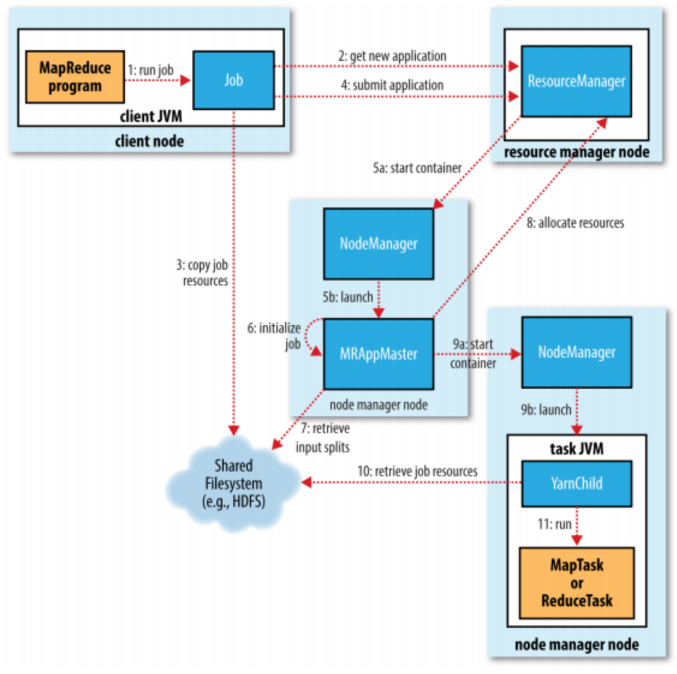
Fair Scheduler (each application is bound to a queue ; YARN containers are given with priority to queues consuming less resources ; within a queue the application having the fewest resources is assigned the asked container ; this can imply resource reduction for running jobs ; attention : in the presence of a single application, that application can ask for the entire cluster).

1.25 MapReduce on YARN

MapReduce job execution relies on YARN.

In a first phase (steps 1-3) the client node asks the RM for a job ID and then sends all the information needed to start the JOB. The RM then starts a container (steps 5-6) for the the MR Application manager, which initialise the jobs and decide whether to run the job locally (uber execution, input size ¡ size of a block) or on the cluster in parallel. In case of distributed execution, the AP asks for containers for Map and Reduce tasks (a container for each task). Only Map tasks container are allocated keeping into account locality when possible The number of running Map tasks depends on the number of input blocks. The number of Reduce tasks is set by the user.

Task execution is made within a dedicated Java Virtual Machine. If task succeeds it commits and inform the AM. According to configuration, the same task can be run in parallel in order to augment robustness and efficiency (speculative task execution). Only one of speculative tasks commits, the other ones abort. The job succeeds if all the to be executed tasks succeed. Several statistics about execution are made available by Counters



1.26 Hadoop streaming

Although Hadoop is a Java-based system it can support execution of MapReduce jobs written in other languages.This is particularly important for us, as we are going to implement MapReduce jobs in Python. To this end we will rely on Hadoop streaming. In a nutshell, the task JVM runs all the auxiliary operations (split and record reading) output writing, etc. The Map/Reduce algorithms are executed on the node manager node and can read records (pairs) on the Linux/Unix standard input stream (stdin) and on the standard output stream (stdout). For instance the Map task running on the JVM reads records and put them on the stdin stream so that the Map program can read and process them. The Map program emits pairs by performing simple print operations, that put pairs on the stdout stream, that are in turn read by the Map task and sent to the Shuffle&Sort phase. So streaming implies an overhead, that is negligible for complex (time consuming) tasks.

TODO mettre tous les games de spark

2 Cours 2 : Big Data Systems, Paradigms, Algorithms

Spark has Efficient primitives for data sharing. In MapReduce, the only way to share data across processing step is stable storage (disk) Replication also makes the system slow, but it is necessary for fault tolerance. Solution : In memory data processing and sharing.

Spark réalise une lecture des données au niveau du cluster (grappe de serveurs sur un réseau), effectue toutes les opérations d'analyse nécessaires, puis écrit les résultats à ce même niveau. De ce fait, là où le MapReduce de Hadoop travaille par étape, Spark peut travailler sur la totalité des données en même temps. Il est donc jusqu'à dix fois plus rapide pour le traitement en lots et jusqu'à cent fois plus rapide pour effectuer l'analyse en mémoire. Spark exécute la totalité des opérations d'analyse de données en mémoire et en temps réel. Il s'appuie sur des disques seulement lorsque sa mémoire n'est plus suffisante. À l'inverse, avec Hadoop les données sont écrites sur le disque après chacune des opérations. Ce travail en mémoire permet de réduire les temps de latence entre les traitements, ce qui explique une telle rapidité.

2.1 RDD

Un RDD est une collection de données calculée à partir d'une source et conservée en mémoire vive (tant que la capacité le permet). L'un des avantages apporté par RDD se trouve dans sa capacité à conserver suffisamment d'informations sur la manière dont une partition RDD a été produite. En cas de perte d'une partition il est donc en mesure de la recalculer.

Solution : Resilient Distributed Datasets (RDD). A distributed main-memory abstraction. Immutable collections of objects spread across a cluster. Lineage among RDDs to enable their re-evaluation in case of cluster node failures.

An RDD is a collection which divided into a number of partitions, which can be independently processed.

2.2 Programming model

A data flow is composed of any number of data sources and data sinks by connecting their inputs and outputs by means of data operators.

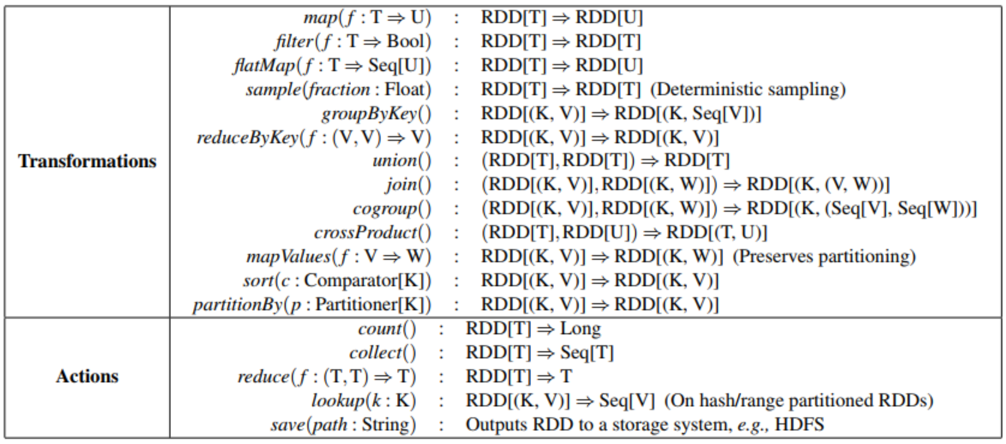
Based on parallelizable operators. Parallelizable operators are higher-order functions that execute user-defined functions in parallel, on each partition of an RDD. There are two types of RDD operators : transformations and actions.

Transformations : lazy operators that create new RDDs.

Actions : lunch a computation and return a value to the program driver or write data to the external storage

2.3 Example

Suppose that a web service is experiencing errors and an operator wants to search terabytes of logs in the Hadoop filesystem (HDFS) to find the cause.



Transformations	*map*(f : T ⇒ U)	RDD[T] ⇒ RDD[U]
	filter(f : T ⇒ Bool)	RDD[T] ⇒ RDD[T]
	flatMap(f : T ⇒ Seq[U])	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(f : (V, V) ⇒ V)	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(f : V ⇒ W)	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(c : Comparator[K])	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(p : Partitioner[K])	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	RDD[T] ⇒ Long
	collect()	RDD[T] ⇒ Seq[T]
	reduce(f : (T, T) ⇒ T)	RDD[T] ⇒ T
	lookup(k : K)	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	Outputs RDD to a storage system, *e.g.*, HDFS

2.4 RDD transformations : map

All pairs are independently processed

```
# passing each RDD element trough a function
nums = sc.parallelize([1,2,3])
squares = nums.map(lambda x: x * x)

# selecting elements making a boolenba function returning true
even = squares.filter(lambda x : x % 2 ==0)
# map + flattening
m = nums.map(lambda x: range(x))
# [[0], [0, 1], [0, 1, 2]]
fm = nums.flatMap(lambda x: range(x))
# [0, 0, 1, 0, 1, 2]
```

2.5 RDD transformations : Reduce

Pairs with identical key are grouped. Each group is independently processed

```
pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2), ("dog", 3) ])

pets.reduceByKey(lambda x, y : x +y)
# [('dog', 4), ('cat', 3)]

pets.groupByKey()
pets.groupByKey().map(lambda x : (x[0], list(x[1])))
# [('dog', [1,3]), ('cat', [1, 2])]
```

2.6 RDD transformations : Join

Equi-join on the key

```
visits = sc.parallelize( [("h", \1.2.3.4"), ("a", "3.4.5.6"), ("h", \1.3.3.1")] )

pageNames = sc.parallelize( [("h", \Home"), ("a", \About")] )

visits.join(pageNames)
```

```
# [('a', ('3.4.5.6', 'About')), ('h', ('1.2.3.4', Home')),
 ('h', ('1.3.3.1', 'Home'))]
```

2.7 RDD transformations : CoGroup

Groups each input on key, Groups with identical keys are processed together

```
visits = sc.parallelize([("h", \1.2.3.4"), ("a", "3.4.5.6"), ("h", "1.3.3.1")] )

pageNames = sc.parallelize([("h", "Home"), ("a", "About"), ("o", "Other")])

visits.cogroup(pageNames)
visits.cogroup(pageNames).map(lambda x :(x[0], ( list(x[1][0]), list(x[1][1]))))

# [('a', (['3.4.5.6'], ['About'])), ('h', (['1.2.3.4', '1.3.3.1'], ['Home'])),
('o', ([], ['Other']))]
```

2.8 Remarks

MapReduce makes important abstraction step that greatly helps rapid development of efficient and robust Big Data data flows. But : we still need some 'acking' to ensure good performances ; problems with iterative analyses ; MapReduce programming is not easy.

Spark overcomes these limitations in a large extent, at the cost of more RAM needed. Makes a one more step towards 'The data center is the computer' scenario.

3 Cours 3 : Spark

3.1 Set operators

Union : merges two RDDs and returns a single RDD using bag semantics, i.e., duplicates are not removed.

Intersection : performs intersection, using set semantics, i.e. duplicates are eliminated.

```
    rdd1.union(rdd2) rdd1.intersection(rdd2)
    rdd1.subtractByKey(rdd2)
```

Attention : RDDs do not need to be homogeneous in order to be unioned/intersected.

Difference : can be done by means of subtractByKey on RDDs of key-value pairs.

3.2 Sampling and repartitioning

TODO : mieux expliquer tout ça

Sample : returns a sample of the input RDD, takes as argument a boolean indicating whether the same element can be re-sampled, the percentage of the sample, and a seed for random number generation.

```
>>> rdd = sc.parallelize(range(100), 4)
>>> 6 <= rdd.sample(False, 0.1, 81).count() <= 14
True
```

Repartition : performs RDD repartitioning by possibly lowering/increasing the number of parts. Attention : shuffle is used. So you may want to use coalesce in case of lowering number of parts.

```
    >>> rdd.repartition(10)
```

Glom : return an RDD created by coalescing all elements within each partition into a list (below note the use of take(n), returning the first n elements of an RDD)

```
>>> rdd.glom().take(1)
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24]]
```

3.3 Making RDDs persistent

Crucial for fast iterative and interactive data processing They allow for persisting an RDD in a memory level (RAM, DISK) : the RDD is computed once and re-used many times.

```
>>> rdd = sc.parallelize(["b", "a", "c"])
>>> rdd.persist().is_cached
True
```

3.4 Actions

We also have count(). Reduce can also be performed as an action. The passed binary operation must be associative and commutative : so it is first evaluated locally on each partition and then globally (local aggregation).

```
>>> from operator import add
>>> sc.parallelize([1, 2, 3, 4, 5]).reduce(add)
15
```

saveAsTextFile is used to save an RDD as a text file

```
    >>> rdd1.saveAsTextFile(file: or hdfs: path ...)
```

3.5 Spark execution model : Process distribution

A Spark application consists of a driver program that runs the user's main function and executes various parallel tasks on a cluster. A task is a transform/action operation performed on a single partition.

In the case of shell-based use the driver role is played by the shell itself.

3.6 Stages and scheduling

When a user runs an action on an RDD : the scheduler builds a DAG of stages from the RDD lineage graph. A stage contains as many as possible pipelined transformations with narrow dependencies Between stages we have wide dependencies : those involving a shuffle operation over the cluster.

The scheduler launches tasks to compute missing partitions from each stage until it computes the target RDD. Tasks are assigned on machines based on data locality : if a task needs a partition, which is available in the memory of a node, the task is sent to that node.

TODO wtf?

3.7 RDD fault tolerance

Logging lineage rather than the actual datakllk. No replication (unless specified in persist action). Recompute only the lost partitions of an RDD.

4 Cours 4 : Spark k-means

4.1 Accumulators and Broadcast variables : Not totally shared-nothing...

When Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task.

Sometimes, a variable needs to be shared across tasks, or between tasks and the driver program. General read-write shared variables across tasks is inefficient. Two types of shared variables : accumulators and broadcast variables.

4.2 Accumulators

Aggregating values from worker nodes back to the driver program.

Example : counting events that occur during job execution. Worker code can add to the accumulator with its += method. The driver program can access the value by calling the value property on the accumulator

```
>>> accum = sc.accumulator(0)
>>> accum
Accumulator<id=0, value=0>
>>> >>> sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
>>> accum
Accumulator<id=0, value=10>
>>> accum.value
10
```

TODO : voir ça plus en détails

4.3 Example

How many lines from a file were blank ?

```
>>> t = sc.textFile(\file:/home/dario.colazzo/data/JSON/people.txt",4)
>>> t.collect()
[u'Michael, 29', u'', u'', u'Andy, 30', u'', u'Justin, 19']
>>> blankLines = sc.accumulator(0)
>>> blankLines
Accumulator<id=4, value=0>
>>> def f(x):
...     if x=="" :
...         blankLines.add(1)
...         return []
...     return [y.encode('utf-8') for y in x.split(" ")]
...
>>> words = t.flatMap(f)
>>> blankLines.value
0
>>> words.collect()
['Michael,', '29', 'Andy,', '30', 'Justin,', '19']
>>> blankLines.value
3
```

4.4 Broadcast Variables

Recall that each time Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task.

The broadcast values are sent to each node only once, rather than shipping a copy of it whenever a task is activated. Broadcast variables are cached in main memory, and are read-only. The task using broadcast variables can access its value with the value property

```
>>> rdd = sc.parallelize(range(100))
>>> broadcastVar = sc.broadcast([1, 2, 3])
>>> broadcastVar.value
[1, 2, 3]
>>> rdd.flatMap(lambda y : broadcastVar.value).count()
300
```

4.5 Example : Map joins

Let's first see standard join :

```
>>>>>> bigCollection =
[(random.randint(1,3) ,random.randint(1,100) ) for x in range(1000)]
>>> smallCollection=[(1, 'a'),(2, 'b'), (3,'c')]
>>> bigc = sc.parallelize(bigCollection)
>>> smallc = sc.parallelize(smallCollection)
>>>
>>> smallc.join(bigc).take(5)
[(1, ('a', 38)), (1, ('a', 11)), (1, ('a', 67)), (1, ('a', 89)), (1,
('a', 71))]
```

Map join : smallc can be broadcasted :

```
>>> smallDict=dict( (x[0], x[1]) for x in smallc.collect() )
>>> smallDict[1]
'a'
>>> bc=sc.broadcast(smallDict)
>>> mapJoined = bigc.map(lambda x : (x[0], (bc.value[x[0]], x[1])))
>>> mapJoined.take(5)
[(3, ('c', 60)), (2, ('b', 26)), (3, ('c', 96)), (3, ('c', 29)), (2,
('b', 63))]
>>>bigc.take(5)
[(3, 60), (2, 26), (3, 96), (3, 29), (2, 63)]
>>>
```

TODO ça aussi pas compris

4.6 Conclusion on spark

Spark enables dataflow programming. RDD as a new type added to existing Programming Languages. RDD distribution in main memory. Two types of operations : transformations and actions. Execution engine based on stages, task and lineage. Spark SQL : Dataframes, also used in ML. Counters and broadcast variables.

5 Some program exmaples

5.1 K-Means

With Iris database :

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
...
```

Initialising variables and RDDs :

```
    data = lines.map(lambda x: x.split(’,’))\
 .map(lambda x: [float(i) for i in x[:4]] + [x[4]])\
 .zipWithIndex()\
 .map(lambda x: (x[1], x[0]))
# zipWithIndex allows us to give a specific index to each point
# (0, [5.1, 3.5, 1.4, 0.2, ‘Iris-setosa’])
clusteringDone = False
number_of_steps = 0
```

Initialising the centroids :

```
# In the same manner, zipWithIndex gives an id to each cluster

centroides =
sc.parallelize(data.takeSample(’withoutReplacment’,nb_clusters
))\
.zipWithIndex()\
.map(lambda x: (x[1],x[0][1][:-1]))
# (0, [4.4, 3.0, 1.3, 0.2])
```

Points repartition :

```
joined = data.cartesian(centroides)
# ((0, [5.1, 3.5, 1.4, 0.2, ’Iris-setosa’]), (0, [4.4, 3.0, 1.3, 0.2]))

# We compute the distance between the points and each cluster
dist = joined.map(lambda x: (x[0][0],(x[1][0], computeDistance(x[0][1][:-1], x[1]
[1]))))

def computeDistance(x,y):
 return sqrt(sum([(a - b)**2 for a,b in zip(x,y)]))

# (0, (0, 0.866025403784438))
dist_list = dist.groupByKey().mapValues(list)
# (0, [(0, 0.866025403784438), (1, 3.7), (2, 0.5385164807134504)])
```

Points repartition

```
def closestCluster(dist_list):
 cluster = dist_list[0][0]
 min_dist = dist_list[0][1]
 for elem in dist_list:
    if elem[1] < min_dist:
        cluster = elem[0]
        min_dist = elem[1]
 return (cluster,min_dist)

# We keep only the closest cluster to each point.
min_dist = dist_list.mapValues(closestCluster)
# (0, [(0, 0.866), (1, 3.7), (2, 0.538)])
# (0, (2, 0.538))

# assignment will be our return value : It contains the datapoint,
# the id of the closest cluster and the distance of the point to the centroid

assignment = min_dist.join(data)
# (0, ((2, 0.538), [5.1, 3.5, 1.4, 0.2, 'Iris-setosa']))
```

New centroids

```
clusters = assignment.map(lambda x: (x[1][0][0], x[1][1]
[:-1]))
# (2, [5.1, 3.5, 1.4, 0.2])

count = clusters.map(lambda x: (x[0],1)).\reduceByKey(lambda x,y: x+y)

somme = clusters.reduceByKey(sumList)

newCentroides = somme.join(count).map(lambda x :
(x[0],moyenneList(x[1][0],x[1][1])))

def moyenneList(x,n):
 return [x[i]/n for i in range(len(x))]
```

End condition :

```
# Based on counting the number of point moves
if number_of_steps > 0:
    switch = prev_assignment.join(min_dist)\
        .filter(lambda x: x[1][0][0] != x[1][1][0])\
        .count()
else:
    switch = 150

if switch == 0 or number_of_steps == 100:
    clusteringDone = True
    error = sqrt(min_dist.map(lambda x: x[1][1]).\
 reduce(lambda x,y: x + y))/nb_elem.value
else:
    centroides = centroidesCluster
    prev_assignment = min_dist
    number_of_steps += 1
```

5.1.1 Programme complet

```
    from pyspark import SparkContext, SparkConf
from math import sqrt
```

```
def computeDistance(x,y):
    return sqrt(sum([(a - b)**2 for a,b in zip(x,y)]))


def closestCluster(dist_list):
    cluster = dist_list[0][0]
    min_dist = dist_list[0][1]
    for elem in dist_list:
        if elem[1] < min_dist:
            cluster = elem[0]
            min_dist = elem[1]
    return (cluster,min_dist)

def sumList(x,y):
    return [x[i]+y[i] for i in range(len(x))]

def moyenneList(x,n):
    return [x[i]/n for i in range(len(x))]

def simpleKmeans(data, nb_clusters):
    clusteringDone = False
    number_of_steps = 0
    current_error = float("inf")
    # A broadcast value is sent to and saved  by each executor for further use
    # instead of being sent to each executor when needed.
    nb_elem = sc.broadcast(data.count())

    ##############################
    # Select initial centroides #
    ##############################

    centroides = sc.parallelize(data.takeSample('withoutReplacment',nb_clusters))\
              .zipWithIndex()\
              .map(lambda x: (x[1],x[0][1][:-1]))
    # (0, [4.4, 3.0, 1.3, 0.2])
    # In the same manner, zipWithIndex gives an id to each cluster

    while not clusteringDone:

        ##############################
        # Assign points to clusters #
        ##############################

        joined = data.cartesian(centroides)
        # ((0, [5.1, 3.5, 1.4, 0.2, 'Iris-setosa']), (0, [4.4, 3.0, 1.3, 0.2]))

        # We compute the distance between the points and each cluster
        dist = joined.map(lambda x: (x[0][0],(x[1][0], computeDistance(x[0][1][:-1], x[1][1]))))
        # (0, (0, 0.866025403784438))

        dist_list = dist.groupByKey().mapValues(list)
        # (0, [(0, 0.866025403784438), (1, 3.7), (2, 0.5385164807134504)])

        # We keep only the closest cluster to each point.
        min_dist = dist_list.mapValues(closestCluster)
        # (0, (2, 0.5385164807134504))
        # assignment will be our return value : It contains the datapoint,
        # the id of the closest cluster and the distance of the point to the centroid
        assignment = min_dist.join(data)

        # (0, ((2, 0.5385164807134504), [5.1, 3.5, 1.4, 0.2, 'Iris-setosa']))

        ############################################
        # Compute the new centroid of each cluster #
        ############################################

        clusters = assignment.map(lambda x: (x[1][0][0], x[1][1][:-1]))
        # (2, [5.1, 3.5, 1.4, 0.2])

        count = clusters.map(lambda x: (x[0],1)).reduceByKey(lambda x,y: x+y)
        somme = clusters.reduceByKey(sumList)
        centroidesCluster = somme.join(count).map(lambda x : (x[0],moyenneList(x[1][0],x[1][1])))

        ############################
        # Is the clustering over ? #
        ############################

        # Let's see how many points have switched clusters.
        if number_of_steps > 0:
            switch = prev_assignment.join(min_dist)\
                                    .filter(lambda x: x[1][0][0] != x[1][1][0])\
                                    .count()
        else:
            switch = 150
        if switch == 0 or number_of_steps == 20:
            clusteringDone = True
            error = sqrt(min_dist.map(lambda x: x[1][1]).reduce(lambda x,y: x + y))/nb_elem.value
        else:
            centroides = centroidesCluster
            prev_assignment = min_dist
            number_of_steps += 1

    return (assignment, error, number_of_steps)


if __name__ == "__main__":

    conf = SparkConf().setAppName('exercice')
    sc = SparkContext(conf=conf)

    lines = sc.textFile("hdfs://hadoopmaster:9000/home/dcolazzo/data/kmeans/iris.data.txt")
    data = lines.map(lambda x: x.split(','))\
            .map(lambda x: [float(i) for i in x[:4]]+[x[4]])\
            .zipWithIndex()\
            .map(lambda x: (x[1],x[0]))
    # zipWithIndex allows us to give a specific index to each point
    # (0, [5.1, 3.5, 1.4, 0.2, 'Iris-setosa'])

    clustering = simpleKmeans(data,3)

    clustering[0].saveAsTextFile("hdfs://hadoopmaster:9000/home/dcolazzo/output")
    # clustering[0].coalesce(1).saveAsTextFile("hdfs:/user/dario.colazzo/data/output")
    print(clustering)
```

5.2 Pi estimation

```
def inside(p):
    x, y = random.random(), random.random()
    return x*x + y*y < 1

count = sc.parallelize(xrange(0, NUM_SAMPLES)) \
             .filter(inside).count()
print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)
```

5.3 Text search

In this example, we search through the error messages in a log file.

```
textFile = sc.textFile("hdfs://...")

# Creates a DataFrame having a single column named "line"
df = textFile.map(lambda r: Row(r)).toDF(["line"])
errors = df.filter(col("line").like("%ERROR%"))
# Counts all the errors
errors.count()
# Counts errors mentioning MySQL
errors.filter(col("line").like("%MySQL%")).count()
# Fetches the MySQL errors as an array of strings
errors.filter(col("line").like("%MySQL%")).collect()
```

5.4 Simple data operation

In this example, we read a table stored in a database and calculate the number of people for every age. Finally, we save the calculated result to S3 in the format of JSON. A simple MySQL table ”people” is used in the example and this table has two columns, ”name” and ”age”.

```
# Creates a DataFrame based on a table named "people"
# stored in a MySQL database.
url = \
  "jdbc:mysql://yourIP:yourPort/test?user=yourUsername;password=yourPassword"
df = sqlContext \
  .read \
  .format("jdbc") \
  .option("url", url) \
  .option("dbtable", "people") \
  .load()

# Looks the schema of this DataFrame.
df.printSchema()

# Counts people by age
countsByAge = df.groupBy("age").count()
countsByAge.show()

# Saves countsByAge to S3 in the JSON format.
countsByAge.write.format("json").save("s3a://...")
```

5.5 Wordcount with local aggregation by map

MAPPER

```
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # initialise the dictionary for the word
    dict={}
    for word in words :
        if wordc in dict :
            dict[wordc] = dict[wordc] + 1
            else : dict[wordc] =1
    for x in dict.keys(): print '%s\t%s' % (x, dict[x])
```

REDUCER

```
current_word = None
current_count = 0
word = None


for line in sys.stdin:
    # parse the input we got from mapper.py
    pair = line.split('\t')
    word = pair[0]
    count = pair[1]

    # here count can be >1 since the mapper has performed local aggregation

    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    # recall that no ordering is ensured by deafult Hadoop on the value component (here count)
    if current_word == word:
    current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

# do not forget to output the last word if needed!
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

Il y a un pb : local aggregation performed only at the line level and not at the split level. The combiner is still needed and reduce can be considered as a combiner as it is (counting is an associative operation)

5.6 URL

Design a MapReduce job without Combine that takes as input a file where each line contains a string indicating a Web URL and a natural number indicating the amount of time (in seconds) a user has spent on that Web page during a visit session. Of course you can have multiple lines for the same URL. The job is expected to return the list of unique URLs together with the average visit time.

```
MAPPER
for line in sys.stdin:
    line = line.strip()
    pair = line.split('\t')
    url= pair[0]
    time = pair[1]
    print '%s\t%s' % (url, time)

REDUCER

current_url = None
current_count = 0
current_sum =0
url = None


for line in sys.stdin:
    pair = line.split('\t')
    url = pair[0]
    time = pair[1]

    # here count can be >1 since the mapper has performed local aggregation
    # convert time (currently a string) to int
    try:
        time = int(time)
    except ValueError:
        continue
    if current_url == url:
        current_sum += time
        current_count+=1
    else:
        if current_url:
            print '%s\t%s' % (current_url, current_sum/float(current_count))
        current_sum = time
        current_count = 1
        current_url = url

# do not forget to output the last url if needed!
if current_url== url:
    print '%s\t%s' % (current_url, current_sum/float(current_count))
```

SI on veut ajouter un Combine, the main idea is to have a Map that emits a pair (url, (1,1)) and a combiner that emits (url, (s,c)) where s is the local sum of times for url, while c is the number of times url has been met.

```
    MAPPER
for line in sys.stdin:
    line = line.strip()
    pair = line.split('\t')
    url= pair[0]
    time = pair[1]
    print '%s\t%s\t%s' % (url, time, 1)

    COMBINER

current_url = None
current_count = 0
current_sum =0
url = None

for line in sys.stdin:
    triple = line.split('\t')
    url = triple[0]
    time_sum = triple[1]
    count = triple[2]
# here count can be >1 since the mapper has performed local aggregation
    # convert time (currently a string) to int
    try:
        time_sum = int(time_sum)
    except ValueError:
        continue
    try:
        count = int(count)
    except ValueError:
        continue
    if current_url == url:
        current_sum += time_sum
        current_count+=count
    else:
        if current_url:
            #write result to STDOUT
            print '%s\t%s\t%s' % (current_url, current_sum, current_count)
        current_sum = time_sum
        current_count = count
        current_url = url
# do not forget to output the last url if needed!
if current_url== url:
    print '%s\t%s\t%s' % (current_url, current_sum, current_count)

    REDUCER
Comme le mapper avec print '%s\t%s' % (current_url, current_sum/float(current_count))
à la place de print '%s\t%s\t%s' % (current_url, current_sum, current_count)
```

5.7 SQL

On a Customer(cid, startDate, name) et Order(#cid, total).

SELECT name FROM Customer WHERE month(startDate)=7

```
for line in sys.stdin:
    line = line.strip()
    record = line.split(',')
    if record[1][3:5]=='07' :
        print '%s\t%s' % (record[2], '1')
```

SELECT DISTINCT name FROM Customer WHERE month(startDate)=7

Mapper : comme avant,

```
    REDUCER
current_key = None
key = None
# input comes from STDIN
for line in sys.stdin:
    pair = line.split('\t')
    key = pair[0]
    if current_key != key :
        print '%s\t%s' % (key, '')
        current_key = key
```

SELECT O.cid, SUM(total), COUNT(DISTINCT total) FROM Order O GROUP BY O.cid

```
    MAPPER
for line in sys.stdin:
    line = line.strip()
    record = line.split(',')
    print '%s\t%s' % (record[0], record[1])

    REDUCER
current_key = None
key = None
current_sum =0
current_dict={}
for line in sys.stdin:
    pair = line.split('\t')
    key = pair[0]
    value = int(pair[1])
    if current_key == key :
        current_sum+=value
        if value not in current_dict :
            current_dict[value]=1
    else:
        if current_key :
            print '%s\t%s\t%s' % (current_key, current_sum, len(current_dict))
        current_key = key
        current_sum = value
        current_dict={}
        current_dict[value]=1
if current_key == key :
    print '%s\t%s\t%s' % (key, current_sum, len(current_dict))
```

SELECT C.cid, O.total FROM Customer C, Order O WHERE month(startDate)=7 and C.cid=O.cid

```
MAPPER
for line in sys.stdin:
    line = line.strip()
    attributes = line.split(",")
    # read name of file from os environment
    a = os.environ['mapreduce_map_input_file']
    l = a.split("/")
    namefile = l[len(l)- 1]
    if namefile !='Customer.txt' :
        print '%s\t%s' % ( attributes[0], namefile +","+ attributes[1])
    elif attributes[1][3:5]=="07" :
        print '%s\t%s' % (attributes[0], namefile )


    REDUCER
current_cid = None
current_table=None
cid = None
lvalues =[]

# input comes from STDIN
for line in sys.stdin:
    pair = line.strip().split('\t')
    cid = pair[0]
    rest = pair[1]

    if current_cid == cid:
        lvalues.append(rest)
    else:
        if current_cid:
            if 'Customer.txt' in lvalues :
                for x in lvalues :
                    if x != 'Customer.txt' :
                        print '%s\t%s' % (current_cid, x.split(",")[1])
        current_cid = cid
        lvalues=[rest]
if current_cid == cid:
    if 'Customer.txt' in lvalues :
        for x in lvalues :
            if x != 'Customer.csv' :
                    print '%s\t%s' % (cid, x.split(",")[1])
```

SELECT C.cid, O.total FROM Customer C LEFT OUTER JOIN ON Order O ON C.cid=O.cid WHERE month(startDate)=7

```
    MAPPER
for line in sys.stdin:
    line = line.strip()
    attributes = line.split(",")
    a = os.environ['mapreduce_map_input_file']
    l = a.split("/")
    namefile = l[len(l)- 1]
    if namefile !='Customer.txt' :
        print '%s\t%s' % ( attributes[0], namefile +","+ attributes[1])
    elif attributes[1][3:5]=="07" :
        print '%s\t%s' % (attributes[0], namefile )

    REDUCER
current_cid = None
current_table=None
cid = None
```

```
lvalues =[]

for line in sys.stdin:
    pair = line.strip().split(’\t’)
    cid = pair[0]
    rest = pair[1]
    if current_cid == cid:
        lvalues.append(rest)
    else:
        if current_cid:
            # write result to STDOUT
            if ’Customer.txt’ in lvalues :
                if len(lvalues)>1 :
                    for x in lvalues :
                        if x != ’Customer.txt’ :
                            print ’%s\t%s’ % (current_cid, x.split(",")[1])
                else : print ’%s\t%s’ % (current_cid, "NULL")
        current_cid = cid
        lvalues=[rest]
if current_cid == cid:
     if ’Customer.txt’ in lvalues :
                if len(lvalues)>1 :
                    for x in lvalues :
                        if x != ’Customer.txt’ :
                            print ’%s\t%s’ % (current_cid, x.split(",")[1])
                else : print ’%s\t%s’ % (current_cid, "NULL")
```

5.8 Graphs

Assume that a graph is a set of edge pairs (v,w) indicating that an edge exists from node v to node w.

5.8.1 Universal sink

Given a directed graph, find the set of nodes having an incoming edge from all the remaining nodes, and having no outgoing edges.

We first load the data into an RDD, by assume that the graph file is on the local file system (hence the file : pre fix below) :

```
     g = sc.textFile(  "file:/home/dario.colazzo/data/graphs/graph.txt")
```

On ne va ensuite garder que les première et 4ème colonnes (les autres ne servent à rien)

```
    graph = g.map(lambda x: (   x.split(’\t’)[0].encode(’utf-8’),   x.split(’\t’)[2].encode(’utf-
```

IMPORTANT : assume in the initial graph file we have multiple edges from v to w with, of course, different labels (for instance : “2 5 7 7” and “2 9 7 7” meaning from node “2” we have two outgoing edges whose labels are “5” and “7”, respectively). The preprocessing above will drop information about edges “5” and “9”, so it will create an RDD where the string (2, 7) is repeated twice. So we eliminate potential duplicates from graph :

```
    graph_unique = graph.distinct()
```

We need to count how many nodes we have in graph_unique :

```
    nodes = graph_unique.flatMap(lambda x: [x[0]]+ [x[1]]).distinct().count()
```

Now, as discussed in class we eliminate from graph_unique all the pairs (v,w) such that (w,u) exists in graph_unique, as in this case w has not to be considered as a potential sink (recall that a sink can not emit any edges). We first need to compute the inverse of graph_unique

```
    graph_unique_inv = graph_unique.map(lambda x : (x[1], x[0]))
```

And then we compute the graph gss on which we will look for sinks

```
    gss = graph_unique_inv.subtractByKey(graph_unique).map(lambda x : (x[1], x[0]))
```

Concerning the previous example on (v,w), it is first inverted, so to obtain (w,v), and then it is eliminated from graph_unique_inv by subtractByKey since (w,u) is in graph_unique. Observe that, after subtraction we need to inverse again to obtain gss, the part of graph_unique that is needed to look for sinks

```
    pre_sinks = gss.map(lambda x: (x[1], 1)).reduceByKey(lambda x,y :    x+y).filter(lambda x : x
    sinks = pre_sinks.map(lambda x: x[0])
```

As you will see by means of sinks.collect() the graph contains no sink. Replay now all the sequence of RDD manipulations, by making the following RDD creation for graph.

```
    graph = sc.parallelize([('1','4'), ('2','4'), ('3','4'), ('3','2') ])
```

Here there is one sink, the node '4'.

5.8.2 Triangle enumeration

Given a directed graph, enumerate all directed triangles, with no duplicates.

```
>>> g = sc.parallelize([(1,4), (4,2), (2,1), (3,2), (2,5), (5,3)])
>>> g.collect()
[(1, 4), (4, 2), (2, 1), (3, 2), (2, 5), (5, 3)]
>>> gInverse = g.map(lambda x: (x[1],x[0]))
>>> gInverse.collect()
[(4, 1), (2, 4), (1, 2), (2, 3), (5, 2), (3, 5)]

>>> opentriangles=g.join(gInverse)
>>> opentriangles.collect()

[(1, (4, 2)), (2, (1, 4)), (2, (1, 3)), (2, (5, 4)), (2, (5, 3)), (3, (2, 5)), (4, (2, 1)), (5, (

>>> trianglesToClose = opentriangles.map(lambda x : ( (x[1] , x)))

>>> trianglesToClose.collect()

[((4, 2), (1, (4, 2))), ((1, 4), (2, (1, 4))), ((1, 3), (2, (1, 3))), ((5, 4), (2, (5, 4))), ((5,

>>> triangles = trianglesToClose.join(g.map(lambda x : (x, 1)))

>>> triangles.collect()
[((4, 2), ((1, (4, 2)), 1)), ((2, 5), ((3, (2, 5)), 1)), ((3, 2), ((5, (3, 2)), 1)), ((1, 4), ((2,

>>> triangles.count()
6
```