# Datum Academy

DATUM ACADEMY

# Hadoop, Spark & MapReduce

Benjamin Renaud
Updated by: Sergio Simonian





# Global Table of Contents

# Prerequisites

1. General knowledge in CS
2. Some knowledge of Java
3. Some knowledge of Python

# Big Data, Distributed Computing & MapReduce

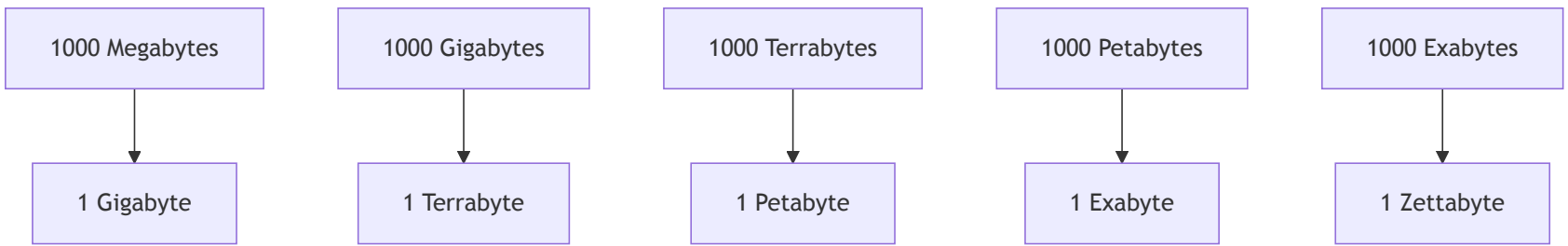
# Introduction

The world is more and more connected:

- In 2016 - 16.1 Zettabytes of data were created or replicated
- In 2020 - 64.2 Zettabytes

(Source: Dave Reinsel, Senior Vice President, IDC Global DataSphere)

=> The need for distributed computing

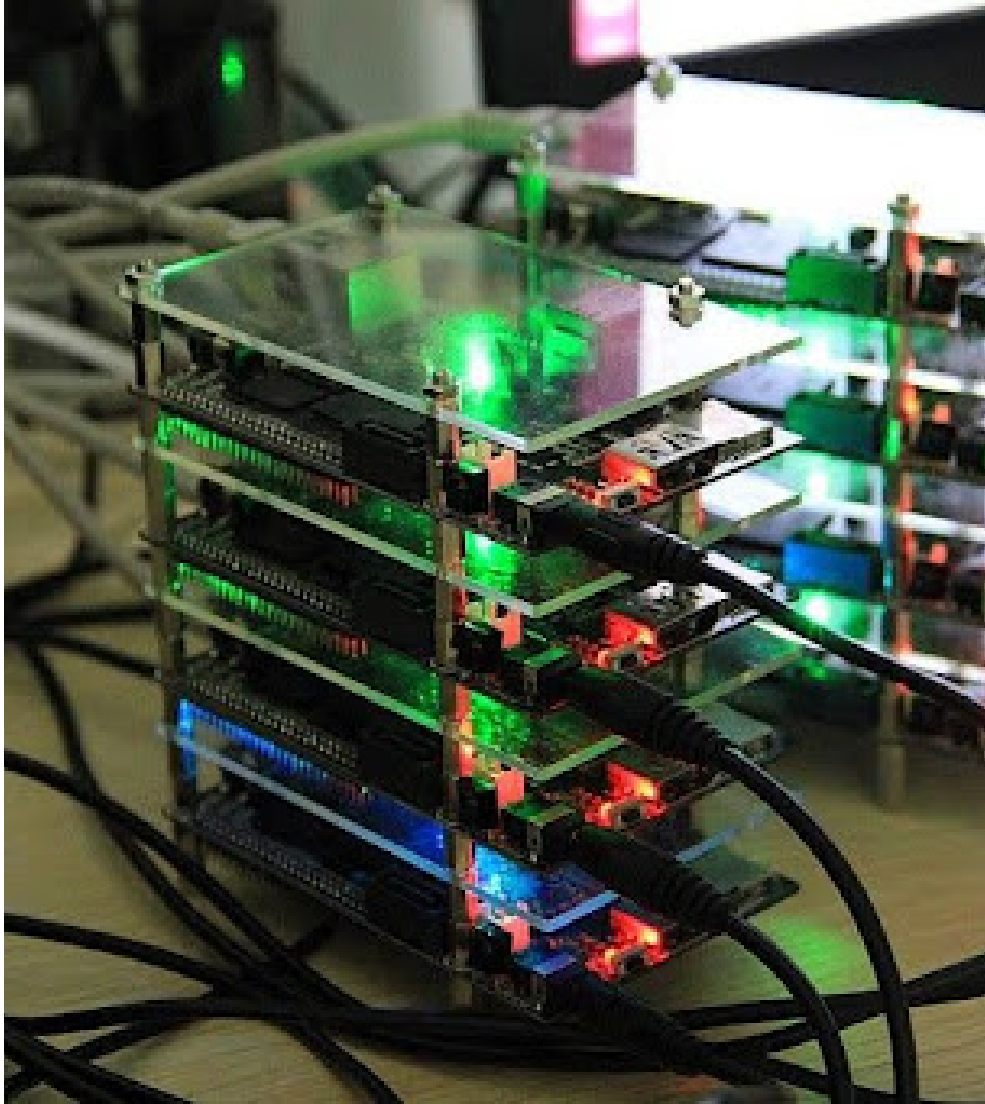


# Distributed Computing

Definition:

> A distributed system is a collection (cluster) of autonomous computing elements (nodes) that appears to its users (people/applications) as a single coherent system

Example of a Cubieboard cluster ->



# Challenges of Distributed Computing

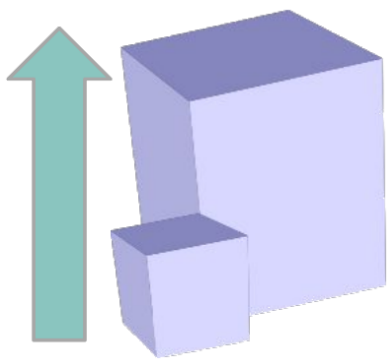
1. Big Data, Distributed Computing & MapReduce
    1. Challenges of Distributed Computing
        1. Scalability
        2. Diversity
        3. Concurrency
        4. Fault Tolerance
        5. Transparency

# Scalability

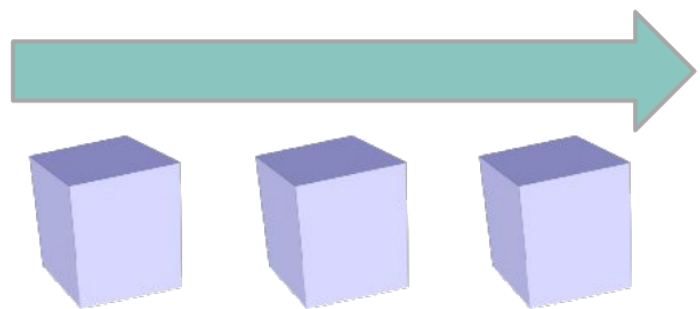
Handling vertical and horizontal scalability.

- Vertical scalability (Scaling Up)
  - Adding more resources to existing nodes
- Horizontal scalability (Scaling Out)
  - Adding more nodes to the cluster

Vertical

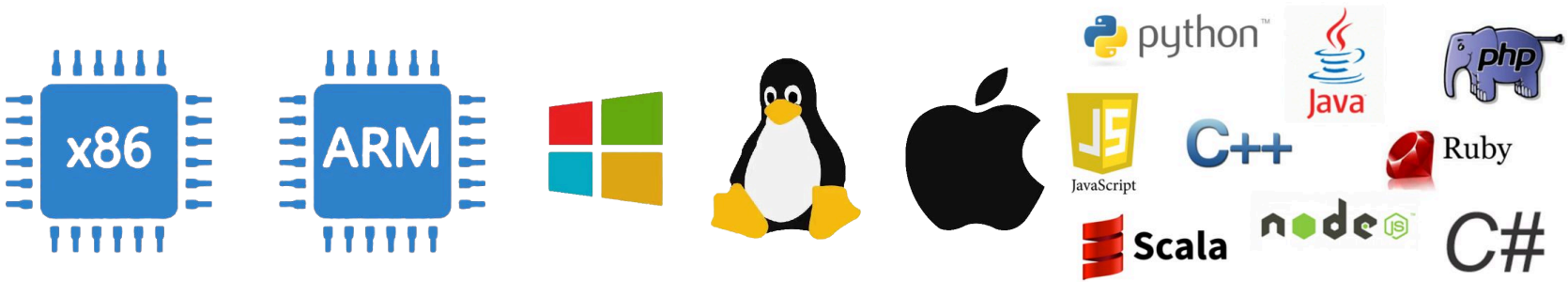


Horizontal



# Diversity

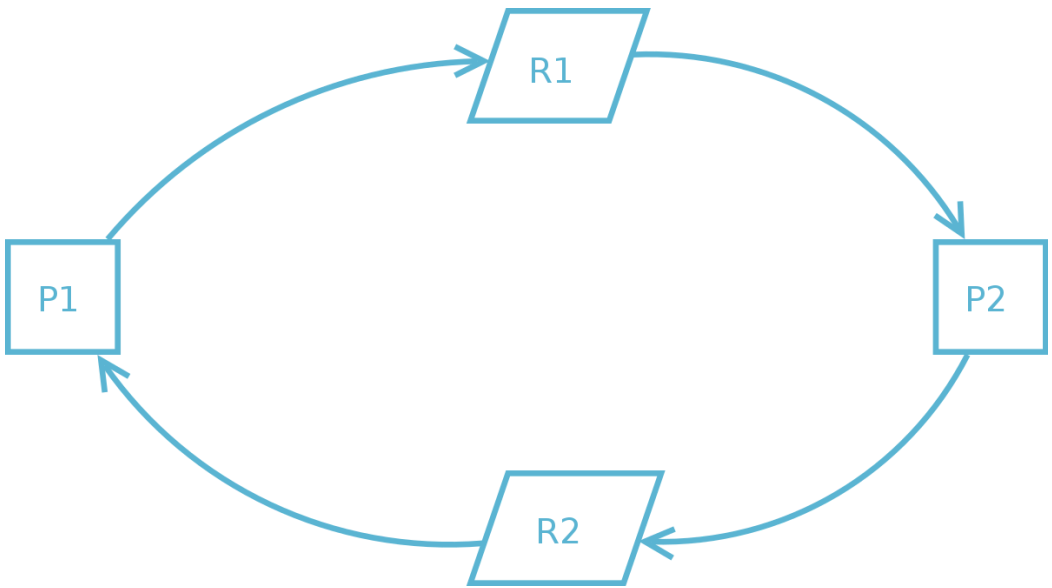
The nodes need to support **different architectures**, **operating systems** and **programming languages**.



# Concurrency

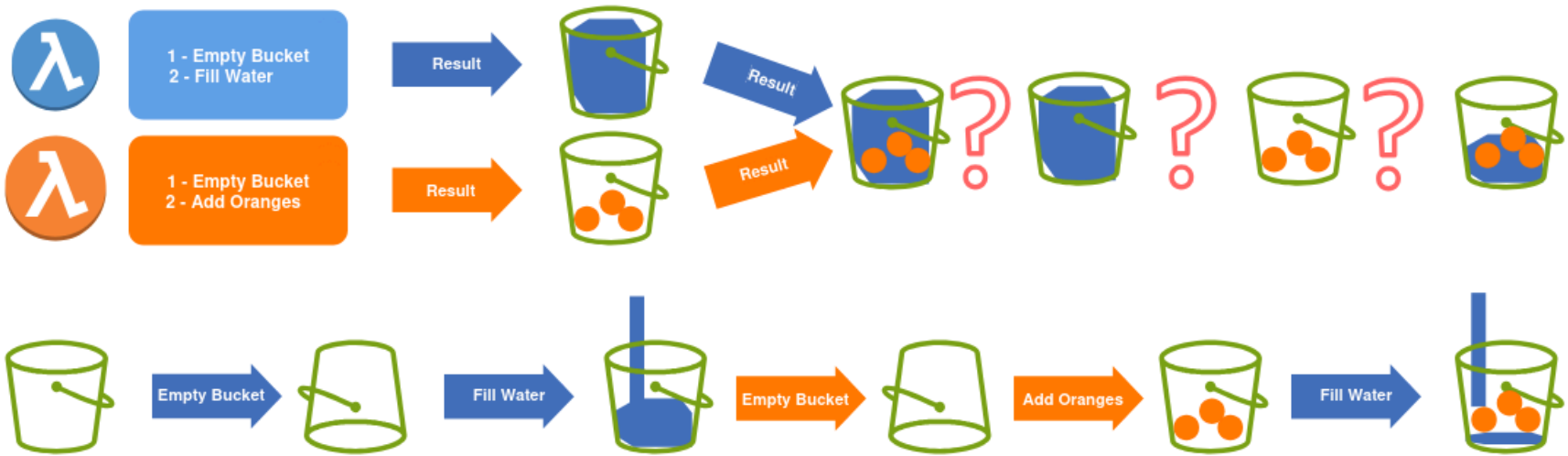
Managing concurrent access to shared resources.

- Race conditions
- Deadlocks



# Concurrency

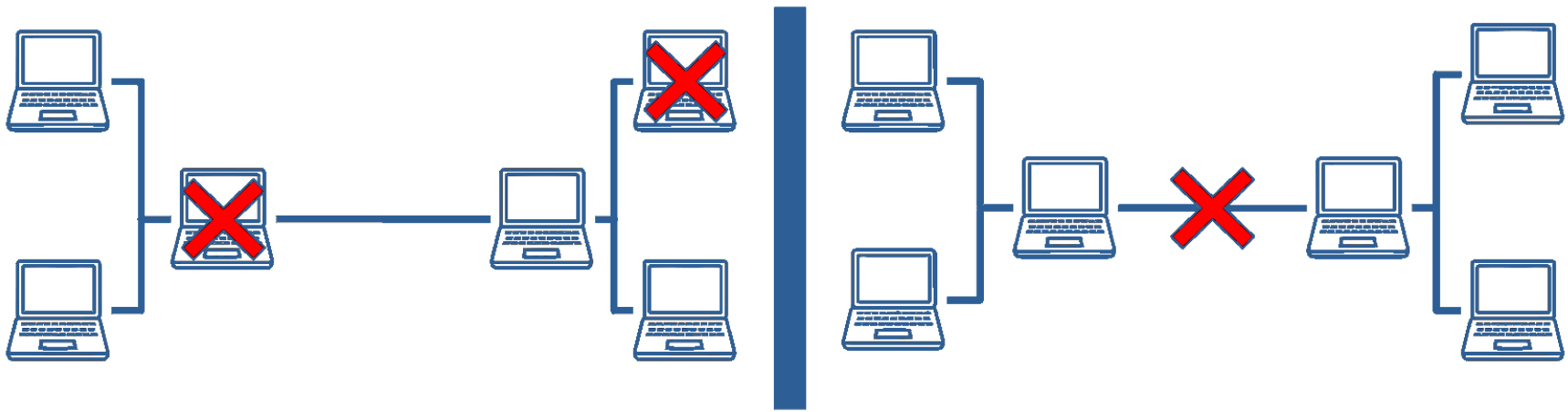
Example of a race condition



# Fault Tolerance

A failing machine/network in the cluster must not lead to errors in the overall computation.

- Hardware or software failures are more likely in distributed systems because multiple components are involved.
- Need to detect, recover, and continue operating despite failures.



# Transparency

The whole cluster must be usable as a single "ordinary" machine.

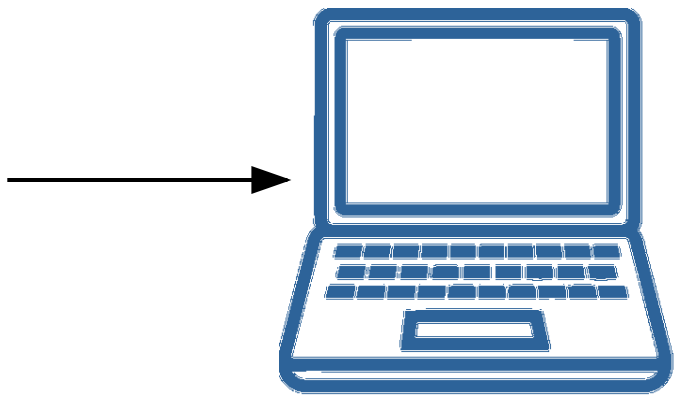


Image: Alexis Lê-Quôc from New York, United States, CC BY-SA 2.0

https://creativecommons.org/licenses/by-sa/2.0, via Wikimedia Commons

# Use cases for Distributed Computing

**Example: Blue Gene (1999) supercomputer**

- Connects 131072 CPUs and 32 terabytes of RAM, all under centralized control for distributed task execution.
- First supercomputer to be marketed and produced (by IBM) in multiple copies.
- Used for medical simulations, studying astronomical radio signals, etc.



Argonne National Laboratory

# Use cases for Distributed Computing

**Example: Folding@home (cluster distributed over the Internet)**

- Equivalent to an exascale supercomputer
- Aims to simulate protein folding to enable the development of new drugs, notably against Alzheimer's disease, certain types of cancer, COVID, etc.

# Use cases for Distributed Computing

**Example: Beowulf cluster**

- Logical architecture defining a means of connecting several personal computers together to perform distributed tasks on a local network.
- A master machine distributes tasks to a series of worker machines.
- Allows high-performance distributed computing from low-cost hardware.

# The problem

Universities and companies need local execution of distributed tasks.

The solutions that were available before:

- **Supercomputers** like Blue Gene: very expensive, often too powerful.
- **In-house solutions**: very substantial initial investment, requires skills in distributed systems.
- **Beowulf architecture**: complex to implement.

# The need

**To have a tool that is:**

- Available
- Easy to deploy, simple to support

**that would:**

- Enable the execution of distributed tasks (with support and monitoring of these tasks)
- Allow the creation of variable-sized clusters that can be expanded at any time
- Handle all challenges of distributed computing

# The MapReduce paradigm

## Presentation

- Algorithmic strategy: **divide and conquer**
- Formal approach to parallel algorithms
- Inspired by existing Map/Reduce functions in functional languages (e.g. Lisp)
- Google research article published in 2004 (“MapReduce: Simplified Data Processing on Large Clusters”).

# Presentation

There are **4 distinct stages** in MapReduce processing:

- **SPLIT** the input data into several fragments.
- **MAP** each of these fragments to obtain (key; value) pairs.
- **SHUFFLE** these (key; value) pairs to group them by key.
- **REDUCE** the key-indexed groups into a final form.

# Presentation

To solve a problem using the Map/Reduce approach with Hadoop:

- Choose a way to slice the input data so that the MAP operation can be parallelized.
- Define which KEY to use for our problem.
- Write the program for the MAP operation.
- Write the program for the REDUCE operation.
- Hadoop will take care of the rest (distributed computing challenges, grouping by distinct key between MAP and REDUCE, etc.).

# Example 1: Word count

- **Objective**: count the number of times each word occurs in a text.
- **Problem**: Imagine a large amount of text, more than a single machine can store.
- A very common Hadoop example

# Example 1: Word count

Input data:

```
Et mes voeux et mes promesses
Ne sont que feintes caresses,
Et mes voeux et mes promesses
Ne sont jamais que du vent.
```

(Pierre Corneille, Chanson, 1626, fragment)

To simplify we are going to:

- Remove punctuation
- Put everything in lower case

# Example 1: Word count

Split by line:

```
et mes voeux et mes promesses
```

```
ne sont que feintes caresses
```

```
et mes voeux et mes promesses
```

```
ne sont jamais que du vent
```

- We obtain 4 fragments from our input data.
- Each fragment will be the input data for our MAP operation.

# Example 1: Word count

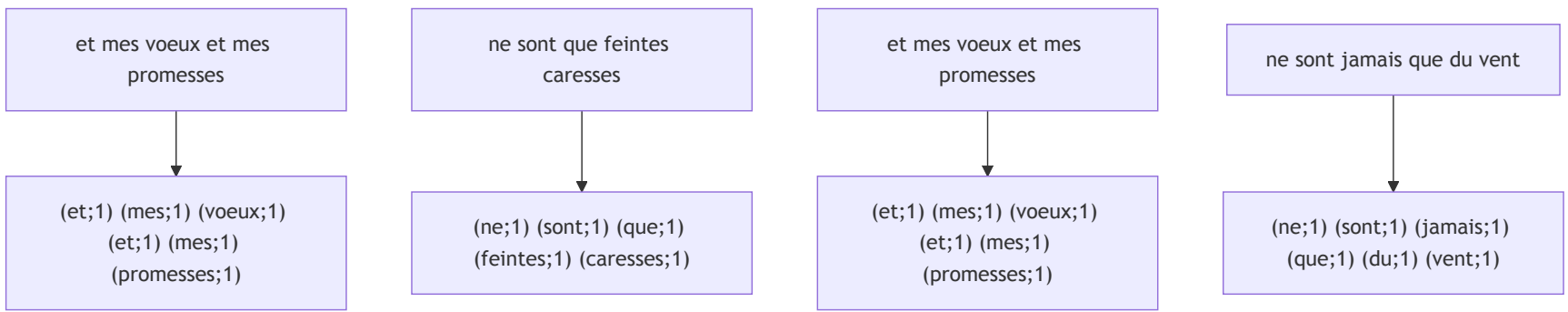
- The Map operation produces a set of (key; value) pairs, which are then grouped by key.
- It’s up to us to implement the MAP operation and determine which key to use.
- To group by word, we simply produce (WORD; 1) pairs for each word in each fragment.

# Example 1: Word count

The code for our MAP operation (in pseudo code):

```
FOR WORD IN FRAGMENT, DO:
    GENERATE (WORD; 1)
```

For each of our fragments, the generated (key; value) pairs will therefore be :



# Example 1: Word count

- Shuffle: Grouping by distinct key.
- Performed automatically by Hadoop (in a distributed manner).
- The following 12 groups are obtained:

```
G1:  (et;1) (et;1) (et;1) (et;1)
G2:  (mes;1) (mes;1) (mes;1) (mes;1)
G3:  (voeux;1) (voeux;1)
G4:  (jamais;1)
G5:  (sont;1) (sont;1)
G6:  (que;1) (que;1)
G7:  (ne;1) (ne;1)
G8:  (feintes;1)
G9:  (caresses;1)
G10: (du;1)
G11: (vent;1)
G12: (promesses;1) (promesses;1)
```

## Example 1: Word count

- The REDUCE operation receives one group at a time.
- In our case, it will sum all the values linked to the specified key and return a single key/value pair (WORD, TOTAL).

Pseudo code:

```
TOTAL = 0
FOR VALUE IN VALUES, DO:
  TOTAL += VALUE
RETURN (KEY; TOTAL)
```

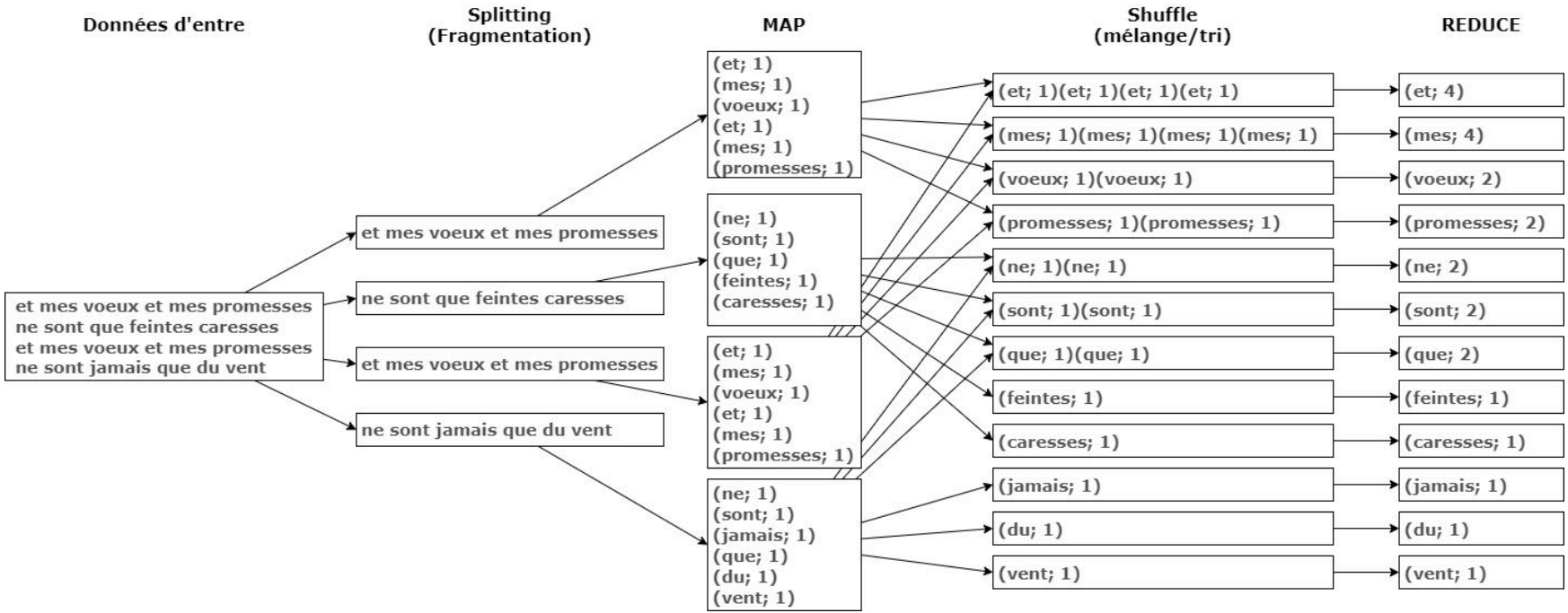
# Example 1: Word count

- **Final result**: one (key; value) pair per distinct word.
- The key is the word and the value is the number of occurrences.

```
et: 4
mes: 4
ne: 2
promesses : 2
que: 2
sont: 2
voeux: 2
du: 1
vent: 1
[...]
```

# Example 1: Word count

Overall diagram



# Exemple 2 : Web Statistics

Example: counting the number of visits per page on a website.

Input data: (web server log files)

```
/index.html [19/Oct/2013:18:45:03 +0200]
/contact.html [19/Oct/2013:18:46:15 +0200]
/news.php?id=5 [24/Oct/2013:18:13:02 +0200]
/news.php?id=18 [24/Oct/2013:18:14:31 +0200]
... etc ...
```

# Exemple 2 : Web Statistics

- SPLIT: per line.
- MAP: delete GET parameters and date. Generates pairs (SITE; 1).
- REDUCE: is identical to the “word count” example.

The MAP operation can also be used to filter by date, IP, etc.

# Example 3: Common friends

- A social network with millions of users.
- Each user has a list of friends.
- **Objective**: Find, for any pair of users, their common friends.

# Example 3: Common friends

Input data : (User => Friends)

```
A ⇒ B, C, D
B ⇒ A, C, D, E
C ⇒ A, B, D, E
D ⇒ A, B, C, E
E ⇒ B, C, D
```

# Example 3: Common friends

- Split: per line.
- MAP:
  - Separate user and friends.
  - For each friend, generate a pair (key; value).
  - The key is the “User - Friend” combination, sorted alphabetically (key “B-A” becomes “A-B”).
  - The value is the list of friends.

# Example 3: Common friends

Map operation pseudo code:

```
USER = [FIRST PART OF THE LINE]
FOR FRIEND IN [LAST PART OF THE LIGNE], DO:
    IF USER < FRIEND:
        KEY = USER + "-" + FRIEND
    ELSE:
        KEY = FRIEND + "-" + USER
    GENERATE (KEY; [REST OF THE LINE])
```

```
"A ⇒ B, C, D"    becomes (“A-B“; “B, C, D”) (“A-C“; “B, C, D”) (“A-D“; “B, C, D”)

"B ⇒ A, C, D, E" becomes (“A-B“; “A, C, D, E”) (“B-C“; “A, C, D, E”) (“B-D“; “A, C, D, E”) (“B-E“; “A, C, D, E”)

"C ⇒ A, B, D, E" becomes (“B-C“; “A, B, D, E”) (“A-C“; “A, B, D, E”) (“C-D“; “A, B, D, E”) (“C-E“; “A, B, D, E”)

"D ⇒ A, B, C, E" becomes (“A-D“; “A, B, C, E”) (“B-D“; “A, B, C, E”) (“C-D“; “A, B, C, E”) (“D-E“; “A, B, C, E”)

"E ⇒ B, C, D"    becomes (“C-E“; “B, C, D”) (“B-E“; “B, C, D”) (“D-E“; “B, C, D”)
```

# Example 3: Common friends

After the SHUFFLE operation we have 9 groups:

```
(“A-B“; “A, C, D, E”)  (“B-C“; “A, B, D, E”)  (“C-D“; “A, B, C, E”)
(“A-B“; “B, C, D”)     (“B-C“; “A, C, D, E”)  (“C-D“; “A, B, D, E”)

(“A-C“; “A, B, D, E”)  (“B-D“; “A, B, C, E”)  (“C-E“; “A, B, D, E”)
(“A-C“; “B, C, D”)     (“B-D“; “A, C, D, E”)  (“C-E“; “B, C, D”)

(“A-D“; “A, B, C, E”)  (“B-E“; “B, C, D”)     (“D-E“; “A, B, C, E”)
(“A-D“; “B, C, D”)     (“B-E“; “A, C, D, E”)  (“D-E“; “B, C, D”)
```

# Example 3: Common friends

For each “USER1-USER2” key, we obtain two lists of friends.

Input data for the REDUCE operation:

```
Key “A-B“: Values “A C D E” and “B C D”
Key “A-C“: Values “A B D E” and “B C D”
Key “A-D“: Values “A B C E” and “B C D”
Key “B-C“: Values “A B D E” and “A C D E”
Key “B-D“: Values “A B C E” and “A C D E”
Key “B-E“: Values “A C D E” and “B C D”
Key “C-D“: Values “A B C E” and “A B D E”
Key “C-E“: Values “A B D E” and “B C D”
Key “D-E“: Values “A B C E” and “B C D”
```

# Example 3: Common friends

Our REDUCE operation will determine which friends appear in both lists.

Returns a key/value pair: (“USER-FRIEND”, FRIENDS_IN_COMMUN)

Pseudo-code:

```
COMMON_FRIENDS_LIST = [] // Empty list at start.
IF LENGTH(VALUES) ≠ 2, THEN: // Should not occur.
    RETURN ERROR
ELSE:
    FOR FRIEND IN VALUES[0], DO:
        IF FRIEND ALSO PRESENT IN VALUES[1], THEN:
            // Present in both friend lists, add it.
            COMMON_FRIENDS_LIST += FRIEND
    RETURN (KEY; COMMON_FRIENDS_LIST)
```

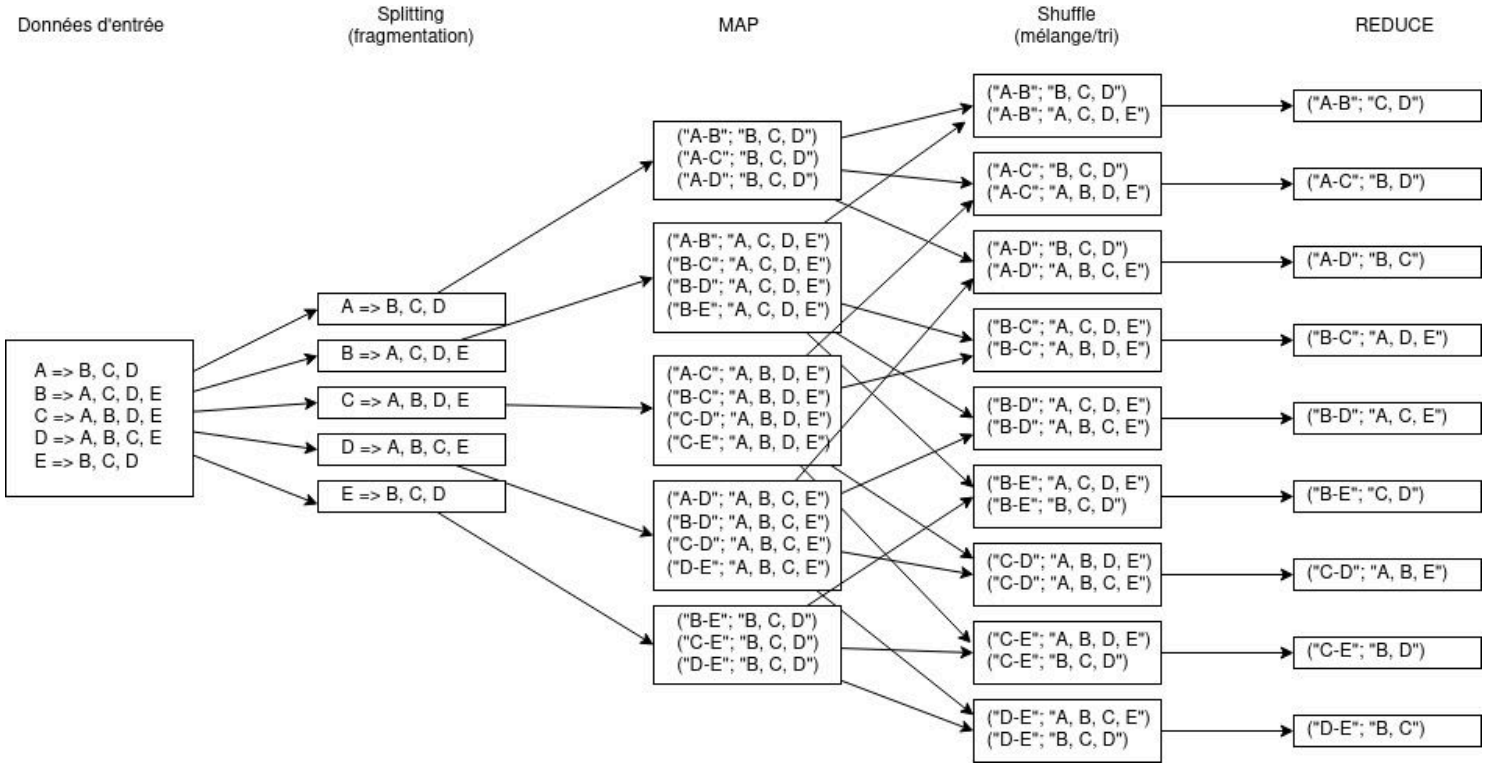
# Example 3: Common friends

**Final result:**

```
“A-B“: “C, D“
“A-C“: “B, D“
“A-D“: “B, C“
“B-C“: “A, D, E“
“B-D“: “A, C, E“
“B-E“: “C, D“
“C-D“: “A, B, E“
“C-E“: “B, D“
“D-E“: “B, C“
```

# Example 3: Common friends

## Overall diagram



# Example 4: Graph traversal

- The aim is to determine the maximum depth of all nodes in a graph, starting from a starting node.
- Breadth-first search algorithm.
- This algorithm is rather problematic because the steps are highly interconnected - but can be solved by making several iterations of the same Map/Reduce program.

## Conclusion

- Using the MapReduce model, we've been able to create four programs with just a few lines of code, which can perform quite complex processing.
- All we need to do is divide the input data and implement the MAP and REDUCE operations.
- Better still, our processing is **distributed**: even with hundreds of gigabytes of data, as long as we have enough machines in the Hadoop cluster, processing will be carried out quickly. To go faster, all we need to do is add more machines.

## Conclusion

- The choice of key is critical in order to exploit shuffle to the full extend.
- A poor choice of key or algorithm causes poor parallelism.
- A key that produces many groups of similar size is preferable to one that creates only a few groups of disproportionate size.
- MapReduce is generally suited to problems where “divide and conquer” approaches make sense.
- MapReduce is less suited to highly connected problems.
- Some problems are harder to implement (graph traversal, etc.).

# Conclusion

Common applications of MapReduce and Hadoop :

- Log analysis and data in general
- Processing massive volumes of data.
- Distributed execution of CPU-intensive tasks (scientific simulations, video encoding, training machine learning models, etc.).

# Hadoop

1. Hadoop
    1. Introduction
    2. Hadoop: HDFS
    3. Hadoop: YARN

# Introduction

- First widely used MapReduce framework.
- Origin: Yahoo / Doug Cutting and Mike Cafarella 2006 (search engine indexing)
- In 2011: fully functional, the Yahoo cluster has 42,000 nodes
- “Hadoop” is not an acronym; named after Cutting’s child’s stuffed elephant.however, it is sometimes called: High Availability Distributed Object Oriented Platform

# Introduction

- Apache Foundation project - Open Source.
- Implements the MapReduce model very strictly.
- Developed in Java; API also in Java.
- Very easy to deploy (preconfigured Linux packages), very easy to configure.
- Handles all challenges of distributed computing.
- Two main components: **HDFS** (File System) and **YARN** (Execution Engine).

# Introduction

- Main means of interacting with the cluster: CLI (command line interface) tool - **hadoop**.
- Allows you to read/write files, submit programs for execution, etc.
- Graphical interfaces are also available.
- The use of the hadoop command will be detailed in the following sections.

# Hadoop: HDFS

# Overview

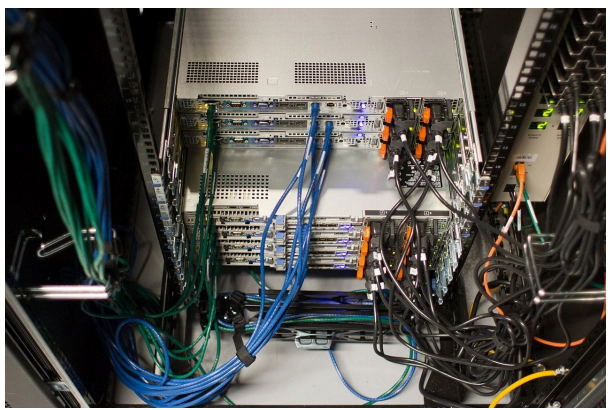
- HDFS: Hadoop Distributed FileSystem.
- Inspired by the research paper: “The Google File System”, Google, 2003.
- Required for concurrent data access by cluster nodes.

> Note: Hadoop can also communicate directly with a database. This mode of integration is achieved via “bridges”.

# Overview

Characteristics of HDFS :

- It is **distributed**: data is spread across the entire cluster of machines.
- It is **replicated**: if one of the machines in the cluster fails, no data is lost.
- It's **rack-aware**: data is replicated on different racks - if an entire rack of servers fails, no data is lost. HDFS can also optimize data transfers to limit the "distance" to be covered for replication.



# Overview

**On HDFS, data is :**

- Structured as in a traditional Unix file system (/ as root).
- Divided into blocks of 128 MB (or 64 MB on older versions) by default (configurable).

# Architecture

**Two main servers (daemons):**

- The **NameNode**: stores file information (such as: file names, their location on HDFS, etc. ). Only one NameNode is active in the entire Hadoop cluster.
- The **DataNode**: stores the data blocks themselves. There is one DataNode for each machine within the cluster, and they are in constant communication with the NameNode to receive new blocks, indicate which blocks are contained on the DataNode, report errors, etc...

# Architecture

**The state of the NameNode is persisted on disk with 2 main file types:**

- **fsimage**: is a backup of the state of the File System at a given moment.
- **edit-logs**: Contain all changes (creation, modification, deletion) made to files after the last **fsimage** file was created.

# Architecture

**When the NameNode is launched :**

- It loads the latest **fsimage** into memory.
- Applies the changes found in the **edit-logs** to its **fsimage**.
- Future changes made on HDFS will be persisted in **edit-logs** and applied to the **fsimage** in memory.

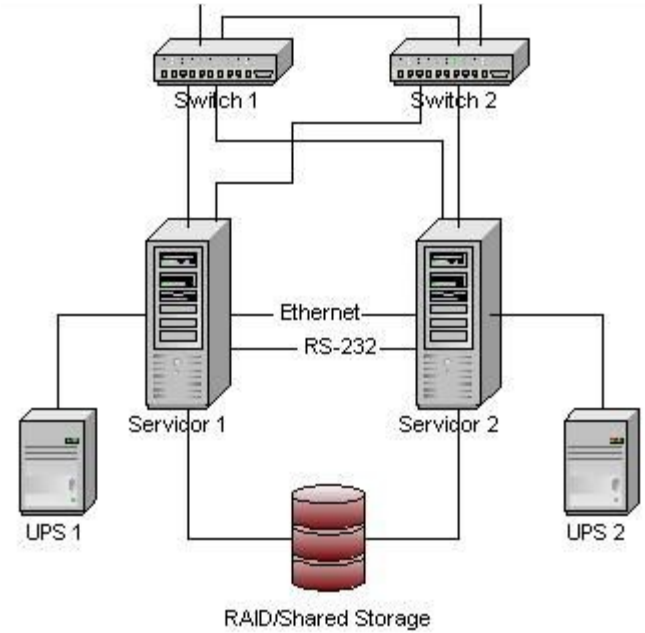
# Architecture

**There are also 2 additional daemons:**

- The **SecondaryNameNode**: Helps the NameNode produce “checkpoints”.
  - It applies changes recorded in edit-logs to the last fsimage and persists the new, updated fsimage on disk.
  - Speeds up NameNode launch time
  - But it is not a backup NameNode.
- **JournalNode**: Can be used in a high-availability configuration to efficiently share edit-logs and metadata between two NameNodes.

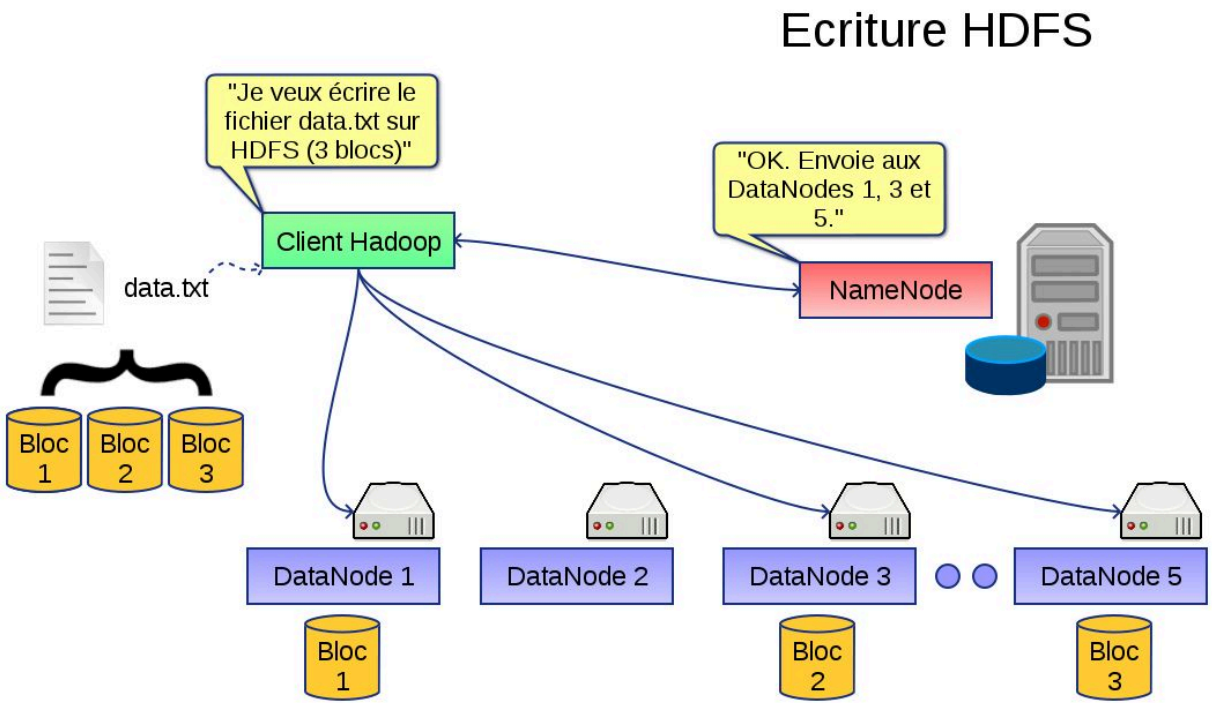
# Architecture

Example: High-availability architecture



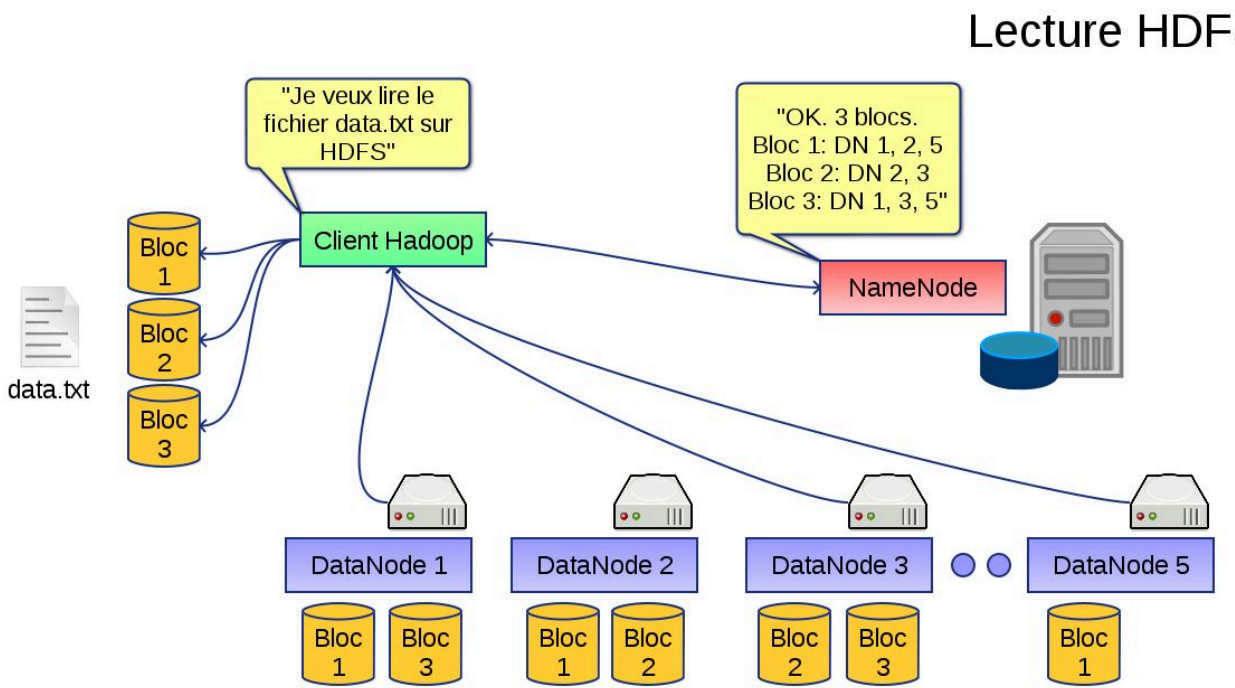

Nuno Tavares

# Writing a file




- Le client indique au NameNode qu'il souhaite écrire un bloc.
- Celui-ci lui indique le DataNode à contacter.
- Le client envoie le bloc au Datanode.
- Les DataNodes répliquent le bloc entre eux.
- Le cycle se répète pour le bloc suivant.

# Reading a file




- Le client indique au NameNode qu'il souhaite lire un fichier.
- Celui-ci lui indique sa taille et les différents DataNode contenant les N blocs.
- Le client récupère chacun des blocs à un des DataNodes.
- Si un DataNode est indisponible, le client le demande à un autre.

# The hadoop fs command

The “hadoop” command with the “fs” option can be used to store or extract files from HDFS.

Usage :

```
hadoop fs -UNIX_COMAND ARGUMENTS
```

- Most unix file system commands are supported: -mkdir, -ls, etc.
- Can be run on any node in the cluster.

There is also an alternative command (less generic than “hadoop fs”):

```
hdfs dfs -UNIX_COMMAND ARGUMENTS
```

# The hadoop fs command

Examples:

```
# Store the book.txt file on HDFS in the /user/john directory.
hadoop fs -put livre.txt /user/john

# Display the contents of the HDFS /user directory
hadoop fs -ls /user

# Retrieve the /results/part-r-0001 file from HDFS.
hadoop fs -get /results/part-r-0001

# Delete the /results folder:
hadoop fs -rm -r /results
```

# The hadoop fs command

```
hdoop@hp:~$
hdoop@hp:~$ hadoop fs -ls /user/wolf/
hdoop@hp:~$ echo "Hello World" > test.txt
hdoop@hp:~$ cat test.txt
Hello World
hdoop@hp:~$ hadoop fs -put test.txt /user/wolf/
hdoop@hp:~$ hadoop fs -ls /user/wolf/
Found 1 items
-rw-r--r--   3 hdoop supergroup         12 2021-08-30 10:31 /user/wolf/test.txt
hdoop@hp:~$ hadoop fs -cat /user/wolf/test.txt
Hello World
hdoop@hp:~$ hadoop fs -mv /user/wolf/test.txt /user/wolf/renamed.txt
hdoop@hp:~$ hadoop fs -ls /user/wolf/
Found 1 items
-rw-r--r--   3 hdoop supergroup         12 2021-08-30 10:31 /user/wolf/renamed.txt
hdoop@hp:~$ hadoop fs -rm /user/wolf/renamed.txt
Deleted /user/wolf/renamed.txt
hdoop@hp:~$ hadoop fs -ls /user/wolf/
hdoop@hp:~$
```

## Notes

- HDFS can be used as a scalable storage system independent of Hadoop.
- Hadoop can also be used without HDFS.
- HDFS is optimized for concurrent reads: writing concurrently is significantly less efficient.
- There are also alternatives to HDFS: databases, network protocols, proprietary alternatives (e.g. MongoDB, Amazon S3, FTP, NFS, MapR-FS).

# Hadoop: YARN

1. Hadoop
    1. Hadoop: YARN
        1. Overview
        2. YARN - ResourceManager
        3. YARN - NodeManager
        4. YARN - Program submission lifecycle
        5. YARN - Architecture
        6. YARN - Comments
        7. User interfaces

# Overview

- YARN: Yet Another Resource Negociator
- Is the **resource management system** and **execution engine** of Hadoop.
- Integrated starting with version 2.x of Hadoop.

# Overview

Two main servers (daemons):

- **ResourceManager**: one per cluster.
- **NodeManager**: one per node.

# YARN - ResourceManager

- The main daemon is the **ResourceManager**.
- It is **unique** on the cluster.
- It manages the concept of **available "resources"** on the cluster: machines, execution slots, memory/CPU, etc.
- It receives program submissions (applications) and starts their execution.

# YARN - ResourceManager

**The ResourceManager is divided into two main components:**

- The **Scheduler**:
  - Plans and distributes tasks to the various execution slots in the cluster based on available resources.
- The **ApplicationsManager**:
  - Receives programs to be executed on the cluster from clients.
  - When a program is received, it launches a first task on the cluster where the driver class (the main part of the program) is executed.
    - This first task is special and is called **ApplicationMaster**.
    - **It becomes the "coordinator"** of the program's execution and will itself submit new tasks to be performed to the Scheduler.

# YARN - NodeManager

- The second daemon, **NodeManager**, runs on each machine in the cluster.
- It maintains **local execution slots** (execution containers).
- It receives and executes tasks in these containers assigned to applications by the **Scheduler**.
- These tasks to be executed can be simple Map or Reduce operations, but also the core of the application itself — the **ApplicationMaster**.
- The **NodeManager** sends **heartbeats** to the **ResourceManager** at regular intervals
  - these contain metrics on available resources and information on tasks in progress, and enable failures to be detected.

# YARN - Program submission lifecycle

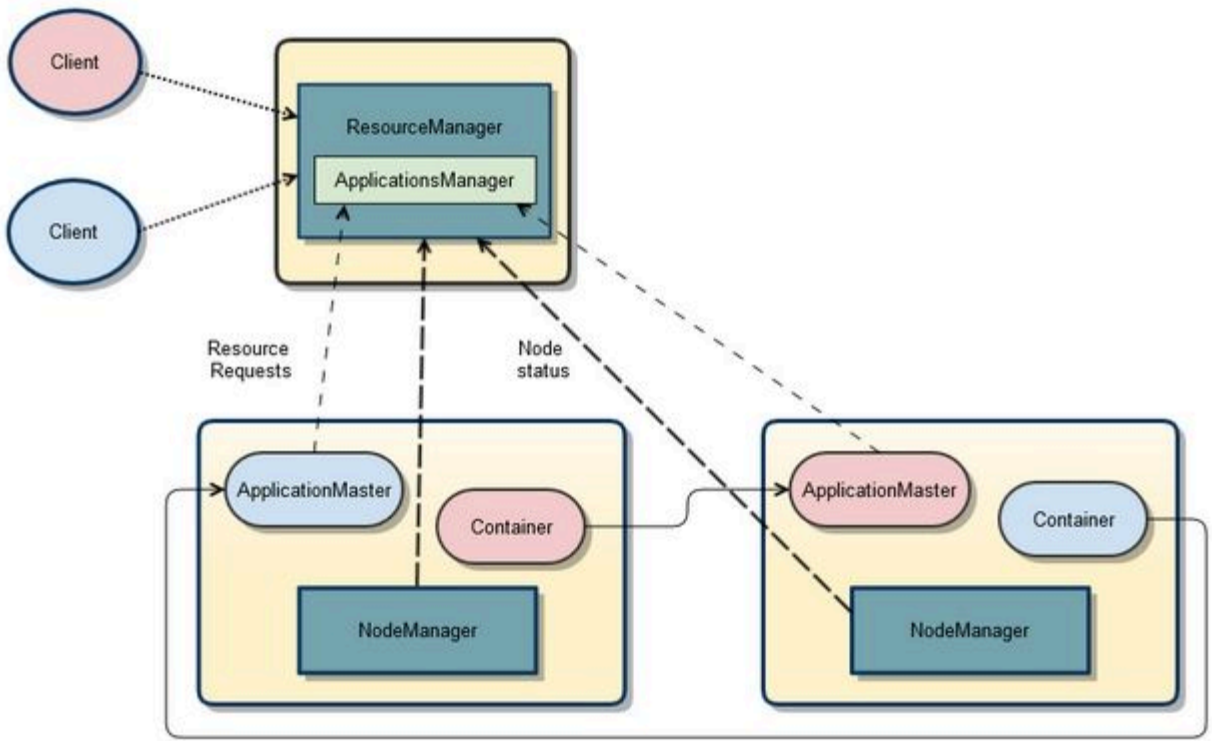
**The steps involved in running a programme via YARN:**

- A **client** connects to the **ResourceManager** and announces the programme's execution/provides its code (jar).
- The **ResourceManager**'s **ApplicationsManager** prepares a free container to run the programme's **ApplicationMaster** (its main).
- The programme's **ApplicationMaster** is launched; it immediately registers with the **ResourceManager** and makes its resource requests (containers to execute Map and Reduce tasks, for example).
- It then contacts the **NodeManagers** corresponding to the containers assigned to it directly to submit the tasks to them.

# YARN - Program submission lifecycle

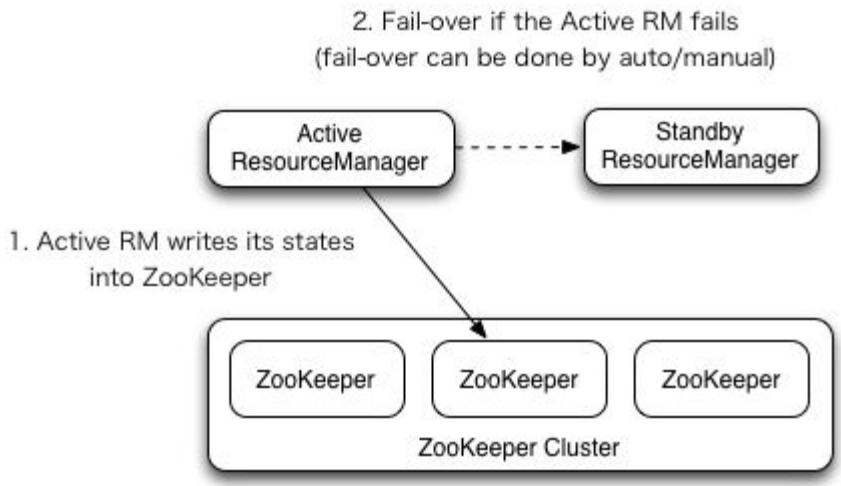
- While the various **tasks** are being executed, the containers (via the NodeManager daemons) continuously update the **ApplicationMaster** on their status. If a problem occurs, the **ApplicationMaster** can request new containers from the **ResourceManager** to re-execute a task.
- The **user** can also obtain an update on the execution status either by contacting the **ResourceManager** or by contacting the **ApplicationMaster** directly.
- Once all **tasks** have been executed, the **ApplicationMaster** notifies the **ResourceManager** and shuts down.
- The **ResourceManager** then releases the remaining occupied container, which was executing the **ApplicationMaster** code.

# YARN - Architecture



# YARN - Comments

The **ResourceManager** is a '**single point of failure**': it is necessary to ensure its high availability through conventional failover mechanisms. (Active/Standby, often implemented with Zookeeper)



# YARN - Comments

- When **YARN** is used with **HDFS**, the **ResourceManager** attempts to launch tasks on nodes that have the required data.
  - This speeds up execution by minimising data transfer between nodes in the cluster.
- Direct communication between the **ApplicationMaster** and **NodeManagers** is a notable difference from version 1 of Hadoop.

## User interfaces

- Hadoop is primarily controlled by console tools (CLI), but a few user interfaces are available to users:
  - **NameNode** provides a web interface (via an integrated web server) that allows users to browse the files and directories stored on HDFS.
  - **RessourcesManager** provides a similar web interface, which allows you to view the available resources of the cluster and the applications currently running.

In addition, there are also third-party user interfaces that have been developed to simplify the use of Hadoop, for example the **Hue** project (Open Source).

# Hadoop development

# Hadoop development overview

As mentioned above, Hadoop is developed in Java.

- **MapReduce** tasks can therefore be implemented using Java interfaces.
- However, there is a very simple tool that allows you to implement these tasks in any language (**Hadoop Streaming**).

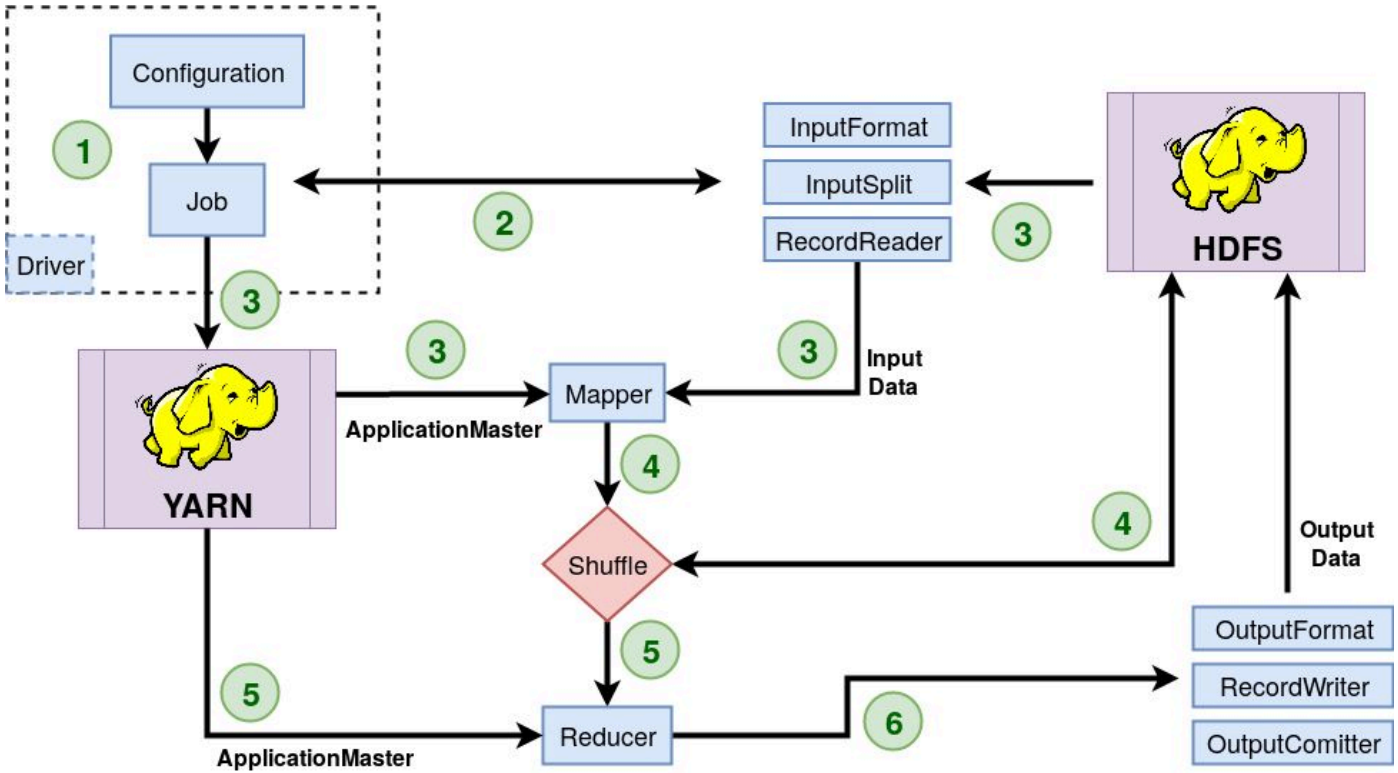
# Hadoop development overview

A Hadoop program is compiled within a .jar file.

**To develop a Hadoop program, we will create three separate classes:**

- A class called **Driver**, which contains the program's **main function**. This class will be responsible for informing Hadoop of the **key/value** data types used, the classes responsible for MAP and REDUCE operations MAP and REDUCE operations, and the HDFS files to be used for input/output.
- A **MAPPER** class (which will perform the MAP operation). A **REDUCER** class (which will perform the REDUCE operation).

# Hadoop development overview



# Hadoop development environment

- For development, a **Java IDE** is highly recommended.
- For example: IntelliJ, Eclipse, Netbeans, emacs, etc.
- For dependency management, you can use **Gradle, Maven**, or do it manually.

# Hadoop development environment

**Supported Java versions:**

- Hadoop 2.X – Java 7 and 8
- Hadoop 3.X – Java 8 and 11

# Hadoop development environment

**There are two versions of the Hadoop API:**

The **new API**: import org.apache.hadoop.**mapreduce**.Mapper ➔ The **old API**: import org.apache.hadoop.**mapred**.Mapper

The old API is much less readable and generates warning messages.

# Hadoop Driver Class

**The Driver class contains the main entry point of our programme.**

- It manages the configuration, sets up one or more MapReduce tasks (**Jobs**) and launches them on the cluster.
- It can run the Hadoop programme in the background or in a blocking manner for the client.

# Hadoop Driver Class

**The main function should perform at least the following operations:**

- Create a Hadoop **Configuration** object.
- Create a Hadoop **Job** object, which represents a MapReduce task.
- Inform Hadoop about the **Driver**, **Mapper**, and **Reducer** classes.
- Inform Hadoop about the **data types** used for the (key; value) pairs between MAP and REDUCE and at the end of REDUCE.
- Specify the input data and the destination for the results.
- Launch the task.

# Hadoop Driver Class

Declaration of the main method in the Driver class:

```
public static void main(String[] args) throws Exception
```

- args is used to retrieve command line arguments.
- It is recommended to let Hadoop retrieve its parameters from args.

# Hadoop Driver Class

**Creating a configuration object:**

```
import org.apache.hadoop.conf.Configuration;

Configuration conf = new Configuration();
```

- The default constructor will automatically find the local Hadoop cluster and obtain its configuration.
- This object can also be adjusted to connect to a different cluster.

# Hadoop Driver Class

Passing command line arguments to Hadoop:

```
import org.apache.hadoop.util.GenericOptionsParser;

String[] ourArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
```

- **GenericOptionsParser** allows Hadoop to obtain its own arguments, such as:
  - -D key=value
  - -conf <CONF_FILE>
  - -fs <local|NAMENODE_LOCATION>
  - etc...
- **getRemainingArgs()** allows you to retrieve arguments not used by Hadoop.

# Hadoop Driver Class

Creating a Hadoop Job object:

```
import org.apache.hadoop.mapreduce.Job;

Job job = Job.getInstance(conf, "Word count v1.0");
```

- Applies the **factory** design pattern - uses **getInstance** instead of a constructor.
- Accepts two arguments – the **Configuration** object and a text description.
- Allows you to **configure your MapReduce task** and start its execution.

# Hadoop Driver Class

Configuring Driver, Mapper, and Reducer classes with the Job object:

```
job.setJarByClass(Driver.class);
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);
```

# Hadoop Driver Class

**Configuring the data types used for pairs (key; value):**

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
```

- Sets the types of keys and values used **between Map and Reduce and at the output of Reduce**.
- Classic Java types such as Integer and String must not be used.
- Requires **Writable** types — special **serializable** types from Hadoop.

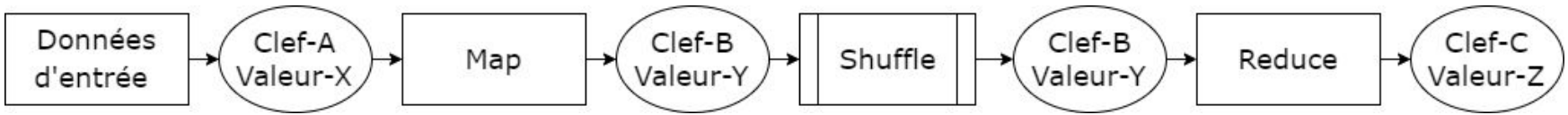
# Hadoop Driver Class

Standard Writable types in Hadoop: (There are many others in org.apache.hadoop.io.*).

- **Text** (instead of String)
- **IntWritable** (instead of Integer)
- **LongWritable** (instead of Long)
- **FloatWritable** (instead of Float)
- **DoubleWritable** (instead of Double)

# Hadoop Driver Class

There are six types of data used for pairs (key; value):



# Hadoop Driver Class

If the types of keys and values output by Map are the same as those used in the output of Reduce:

- Example: Map (Text; IntWritable) and Reduce (Text; IntWritable)

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
```

If the types of keys and values output by Map are different from those used in the output of Reduce:

- Example : Map (Text; IntWritable) and Reduce (Text; Text)

```
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
```

# Hadoop Driver Class

Indicating input and output data on HDFS:

```
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileOutputFormat;
import org.apache.hadoop.fs.Path;

FileInputFormat.addInputPath(job, new Path("/users/john/poeme.txt"));
FileOutputFormat.setOutputPath(job, new Path("/users/john/output"));
```

- These methods update the Job object.

# Hadoop Driver Class

- **Hadoop Path** specifies the path to files or directories.
  - (Example: “hdfs://namenode:9000/user/john/books/*”, *“/user/john/books/*”, “file:///home/tom/*”).
- If the **Schema** (such as: “hdfs://”, “file://”) is not provided, Hadoop uses “hdfs://” by default.
- We can specify as many **addInputPath** entries as we want.
- It is possible to use the **setInputPaths** method to define all entries at once.
- The output must be unique — a (new) directory.

# Hadoop Driver Class

Examples:

```
FileInputFormat.setInputPaths(
    job,
    new Path("hdfs:///poeme.txt"),
    new Path("/poeme2.txt")
);
FileInputFormat.addInputPath(job, new Path("file:///opt/bigdata/ref.txt"));
FileInputFormat.addInputPath(job, new Path("/user/john/input_poeme"));
FileOutputFormat.setOutputPath(job, new Path("/users/john/output"));
```

# Hadoop Driver Class

Task execution (Approach 1/2):

```
job.waitForCompletion(true)
```

- **Synchronous** mode: waits for execution to complete.
- The argument controls verbosity. If it is **true**, Hadoop will display more logs on standard output.
- Returns **true** on success and **false** on failure.

# Hadoop Driver Class

Task execution (Approach 2/2):

```
job.submit();
```

- **Asynchronous** mode: submits the task to the cluster, then returns immediately.
- There are many methods for obtaining information about the execution or influencing the execution:
  - getJobState()
  - getStatus()
  - kilUob()
  - getTaskCompletionEvents()
  - ... and others ...
- Both approaches raise exceptions in case of problems.

# Hadoop Driver Class

The complete code for our Driver class (1/2):

```
package org.mbds.hadoop.wordcount;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
```

# Hadoop Driver Class

The complete code for our Driver class (2/2):

```
/public class WCount {
    public static void main(String[] args) throws Exception {
        // Creates a Hadoop configuration object.
        Configuration conf = new Configuration();
        // Retrieves the remaining arguments in ourArgs.
        String[] ourArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
        // Creates a Job object. Provides the configuration and a description of the task.
        Job job = Job.getInstance(conf, "Compteur de mots v1.0");
        // Defines the Driver, Mapper, and Reducer classes.
        job.setJarByClass(WCount.class);
        job.setMapperClass(WCountMap.class);
        job.setReducerClass(WCountReduce.class);
        // Defines the types of keys/values in our Hadoop program.
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        // Defines the program's input files and the results directory.
        FileInputFormat.addInputPath(job, new Path(ourArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(ourArgs[1]));
        // Start the Hadoop task.
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

# Hadoop Mapper Class

- The **Mapper** class is responsible for the Map operation. It must:
- Extend the Hadoop class **org.apache.hadoop.mapreduce.Mapper**.
- Redefine the **map** method that performs the Map task.

# Reminder - Java Generic Classes

Often used in types such as ArrayList:

```
ArrayList<Integer> list = new ArrayList<>();
```

- Example:

```
public class Variable<T> {
    private T value;

    public void setValue(T t) {
        this.value = t;
    }

    public T getValue() {
        return(this.value);
    }
}
```

```
public static void main(String[] args) {
    Variable<Integer> varInt = new Variable<Integer>();
    Variable<String> varStr = new Variable<String>();

    varInt.setValue(42);
    varStr.setValue("Hello");

    System.out.println("Int: " + varInt.getValue());
    System.out.println("String: " + varStr.getValue());
}
```

# Hadoop Mapper Class

The Mapper class is a generic class that is parameterised with four types:

- The type for the input key
- The type for the input value
- The type for the output key
- The type for the output value

# Hadoop Mapper Class

The key/value pairs for Map are constructed from fragments of input data.

**By default, for text files on HDFS:**

- The Map operation splits the fragment per line.
  - The line **number** is passed as the input **key** (type: **LongWritable**).
  - The line **content** is passed as the input **value** (type: **Text**)
  - If you are not interested in the line number, you can specify Object as the input key type.
- The default behaviour can be adjusted by the **InputFormat** / **RecordReader** classes.

# Hadoop Mapper Class

Example of a Mapper class declaration:

```
public class Map extends Mapper<Object, Text, Text, IntWritable>
```

Using types:

- Object for the input key
- Text for the input value
- Text for the output key
- IntWritable for the output value

# Hadoop Mapper Class

**The map method takes three arguments:**

- The input key.
- The input value.
- A Context object that is used to interact with Hadoop and return the (key; value) pairs resulting from the Map operation.

Example of a **map** method declaration:

```
protected void map(Object key, Text value, Context context)
    throws IOException, InterruptedException
```

# Hadoop Mapper Class

- The **map** method implements the Map task.
- The **write** method of the **Context** object allows you to return a key/value pair.
- **write** can be called multiple times to return multiple key/value pairs.

```
context.write(new Text("ciel"), new IntWritable(1));
```

# Hadoop Mapper Class

**Important:**

- The generic types of the input **key** and **value** of our Mapper class must match the arguments of the map method:

```
public class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    //                          ^^^^^^^^^^^^  ^^^^
    protected void map(LongWritable lineno, Text line, Context context)
        //             ^^^^^^^^^^^^         ^^^^
    throws IOException, InterruptedException {
```

- The key and value types returned by the map method must also match the generic types of the output **key** and **value** of our Mapper class:

```
public class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    //                                              ^^^^  ^^^^^^^^^^^
    protected void map(LongWritable lineno, Text line, Context context)
    throws IOException, InterruptedException {
        context.write(new Text(“ciel”), new IntWritable(5));
        //                ^^^^              ^^^^^^^^^^^
```

# Hadoop Mapper Class

**Important:**

- The **key** and **value** types of our Mapper class must also match the types specified in our Driver class:

```
// In the main Driver method
job.setMapOutputKeyClass(Text.class);
//                       ^^^^
job.setMapOutputValueClass(IntWritable.class);
//                         ^^^^^^^^^^^^
```

```
public class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    //                                              ^^^^  ^^^^^^^^^^^^
    protected void map(LongWritable lineno, Text line, Context context)
    throws IOException, InterruptedException {
        context.write(new Text(“ciel”), new IntWritable(5));
        //                ^^^^              ^^^^^^^^^^^^
```

# Hadoop Mapper Class

- The complete code for our Mapper class (1/2):

```
package org.mbds.hadoop.wordcount;

import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.IntWritable;
import java.util.StringTokenizer;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;
```

# Hadoop Mapper Class

- The complete code for our Mapper class (2/2):

```
public class WCountMap extends Mapper<Object, Text, Text, IntWritable>
{
    protected void map(Object offset, Text value, Context context)
    throws IOException, InterruptedException {
        // Scans each word in the line.
        StringTokenizer tok = new StringTokenizer(value.toString(), " ");
        while(tok.hasMoreTokens() {
            Text word = new Text(tok.nextToken());
            // Returns our (key; value) pair.
            context.write(word, new IntWritable(1));
        }
    }
}
```

# Hadoop Reducer Class

The **Reducer** class is responsible for the Reduce operation. It must:

- Extend the Hadoop class **org.apache.hadoop.mapreduce.Reducer**.
- Redefine the **reduce** method that performs the Reduce task.

# Hadoop Reducer Class

The **Reducer** class is also a generic class that can be parameterised with four types:

- The type for the **input key**
- The type for the **input value**
- The type for the **output key**
- The type for the **output value**

# Hadoop Reducer Class

Example of declaring a Reducer class:

```
public class Reduce extends Reducer<Text, IntWritable, Text, Text>
```

Using the following types:

- **Text** for the **input key**
- **IntWritable** for the **input value**
- **Text** for the **output key**
- **Text** for the **output value**

# Hadoop Reducer Class

**The reduce method also takes three arguments:**

- The input key.
- An iterable object containing the values associated with the input key.
- A Context object that is used to interact with Hadoop and return the (key; value) pairs resulting from the Reduce operation.

Example of a reduce method declaration:

```
protected void reduce(Text key, Iterable<IntWritable> values, Context context)
    throws IOException, InterruptedException
```

# Hadoop Reducer Class

- The key/value pairs for Reduce are constructed from **groups** of key/value pairs resulting from the **Shuffle** operation.
- To return a pair (key; value) as a result, Reduce uses the same principle as in Map.

```
context.write(new Text("ciel"), new Text("5 occurrences"));
```

# Hadoop Reducer Class

**By default, the output of Reduce is persisted in a folder on HDFS:**

- One text file per Reduce execution (per group of common keys from Shuffle).
- One key/value pair per line, with a tab character separating the key from the value.
- The default final write behaviour can be adjusted by the OutputFormat / RecordWriter classes.

Example of a result file on HDFS named “/resulat/part-r-00000”:

```
Key_1 Value_1
Key_2 Value_2
Key_3 Value_3
Key_4 Value_4
```

# Hadoop Reducer Class

**Important:**

- The generic types of the **key** and **value inputs** of our Reducer class must match the arguments of the reduce method:

```
public class Reduce extends Reducer<Text, IntWritable, Text, Text> {
    //                              ^^^^  ^^^^^^^^^^^
    protected void reduce(Text key, Iterable<IntWritable> values, Context context)
        //                ^^^^              ^^^^^^^^^^^
    throws IOException, InterruptedException {
```

- The key types and value returned by the reduce method must also match the generic types of the **key** and **output value** of our Reducer class:

```
public class Reduce extends Reducer<Text, IntWritable, Text, Text> {
    //                                                 ^^^^  ^^^^
    protected void reduce(Text key, Iterable<IntWritable> values, Context context)
    throws IOException, InterruptedException {
        context.write(new Text("ciel"), new Text("5 occurrences"));
        //                ^^^^              ^^^^
```

# Hadoop Reducer Class

**Important:**

- The types of **key** and **output value** in our Reducer class must also match the types specified in our Driver class.

```
// In the Driver main method.
job.setOutputKeyClass(Text.class);
//                    ^^^^
job.setOutputValueClass(Text.class);
//                      ^^^^
```

```
public class Reduce extends Reducer<Text, IntWritable, Text, Text> {
    //                                                 ^^^^  ^^^^
    protected void reduce(Text key, Iterable<IntWritable> values, Context context)
    throws IOException, InterruptedException {
        context.write(new Text("ciel"), new Text("5 occurences"));
        //                ^^^^              ^^^^
```

# Hadoop Reducer Class

**Important:**

- Finally, the generic types of the **key** and **value output** of the **Mapper** class must also match the generic types of the **key** and **value input** of our **Reducer** class:

```
public class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    //                                          ^^^^  ^^^^^^^^^^^
    protected void map(LongWritable lineno, Text line, Context context)
    throws IOException, InterruptedException {
        context.write(new Text(“ciel”), new IntWritable(5));
        //                ^^^^              ^^^^^^^^^^^
```

```
public class Reduce extends Reducer<Text, IntWritable, Text, Text> {
    //                              ^^^^  ^^^^^^^^^^^
    protected void reduce(Text key, Iterable<IntWritable> values, Context context)
    //                    ^^^^              ^^^^^^^^^^^
    throws IOException, InterruptedException {
```

# Hadoop Reducer Class

The complete code for our Reducer class (1/2):

```
package org.mbds.hadoop.wordcount;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Reducer;
import java.util.Iterator;
import java.io.IOException;
```

# Hadoop Reducer Class

The complete code for our Reducer class (2/2):

```
public class WCountReduce extends Reducer<Text, IntWritable, Text, Text> {
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
    throws IOException, InterruptedException {
        // Iterates through all values associated with the provided key.
        Iterator<IntWritable> i = values.iterator();
        int count = 0;
        while(i.hasNext()) {
            count += i.next().get();
        }
        // Returns our pair (key; value)
        context.write(key, new Text(count + " occurrences."));
    }
}
```

# Hadoop Program compilation

# Hadoop Program execution

# Hadoop Streaming

# Hadoop Streaming - MAP

# Hadoop Streaming - REDUCE

# Hadoop Streaming - Example (Python)

# Hadoop HDFS API

# Advanced Hadoop development

# Passing information to Mappers and Reducers

Let’s take the word count example again: we want to add an optional new optional feature - capitalize words.

- In this case one way to proceed would be:
  - Create 2 “Mapper” classes - one for lowercase letters and one for uppercase letters.

# Passing information to Mappers and Reducers

**Solution: 2 Mapper classes in the Driver:

```
String[] ourArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
Job job = Job.getInstance(conf, "WordCount v1.0");
// ...
if(ourArgs[0].equals("--uppercase"))
    job.setMapperClass(WordCountMapUpperCase.class);
else
    job.setMapperClass(WordCountMap.class);
// ...
FileInputFormat.addInputPath(job, new Path(ourArgs[1]));
FileOutputFormat.setOutputPath(job, new Path(ourArgs[2]));
// ...
```

# Passing information to Mappers and Reducers

**Solution: 2 Mapper classes**

- Not ideal: most of the Mapper code is repeated.
- Very limited: what if the optional parameter is not Boolean?

**Solution: Configuration properties**

- Configuration properties allow us to define values in the Driver class and retrieve them in the Mapper/Reducer classes.
- So our Mapper class would be aware of whether we wanted to capitalize words or not.

# Passing information to Mappers and Reducers

To define a value:

```
conf.setInt("org.ebihar.hadoop.wordcount.uppercase", 1);
```

- There are many functions for setting other data types:
  - setLong
  - setFloat
  - ... and set for the String type.

To retrieve a value:

```
Configuration conf = context.getConfiguration();
conf.getInt("org.ebihar.hadoop.wordcount.uppercase", 0)
```

- The configuration object is retrieved with the Context object in the map/reduce functions.
- There are many other functions: **getFloat**, **getLong**, etc. and **get** for the String type.
- The second argument specifies a default value if the property is not found.

# Passing information to Mappers and Reducers

Solution: configuration properties - in the Driver class:

```
String[] ourArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
// ...
int uppercase = 0;
if(ourArgs[0].equals("--uppercase"))
    uppercase = 1;

conf.setInt("org.embds.hadoop.wordcount.uppercase", uppercase);
Job job = Job.getInstance(conf, "WordCount v1.0");
// ...
FileInputFormat.addInputPath(job, new Path(ourArgs[1]));
FileOutputFormat.setOutputPath(job, new Path(ourArgs[2]));
// ...
```

# Passing information to Mappers and Reducers

Solution: configuration properties - in the Mapper class:

```
protected void map(Object offset, Text value, Context context)
throws IOException, InterruptedException {
    StringTokenizer tok;
    Configuration conf = context.getConfiguration();
    int uppercase = conf.getInt("org.embds.hadoop.wordcount.uppercase", 0);
    if(uppercase =/= 0)
        tok = new StringTokenizer(value.toString().toUpperCase(), " ");
    else
        tok = new StringTokenizer(value.toString(), " ");

    while(tok.hasMoreTokens()) {
        Text word = new Text(tok.nextToken());
        context.write(word, new IntWritable(1););
    }
}
```

# Hadoop Counters

- Let’s take the graph traversal example again.
- Our stopping condition was: the color state of all nodes is BLACK.
- One way of implementing this condition would be :
  - After each run - Search the output dataset for nodes that are not black.

# Hadoop Counters

Example:

```
private static Boolean stoppingCondition(Configuration conf, String output_path) {
    FileSystem fs = FileSystem.get(conf);
    FileStatus[] list = fs.globStatus(new Path(output_path));
    for(int i = 0; i < list.length; ++i) {
        BufferedReader br = new BufferedReader(new InputStreamReader(fs.open(list[i].getPath())));
        String line = br.readLine();
        while(line ≠ null) {
            if(!line.contains("BLACK")) {
                br.close();
                return(false);
            }
            line=br.readLine();
        }
        br.close();
    }
    return(true);
}
```

# Hadoop Counters

**This solution is problematic:**

- The graph could contain millions of nodes.
- Our program reads the output twice (in `reduce` and `stoppingCondition`).

**A better solution would be:**

- Check the stop condition during reduce execution.
- Transmit this information to the Driver.
- Hadoop **Counters** will be useful for this.

# Hadoop Counters

A Hadoop Counter:

- Contains a numerical value: of type Long.
- Can only be incremented.
- In Mapper/Reducer: can be read and incremented.
- In Driver: can be read.

Use cases:

- Communication between Driver and Mappers/Reducers.
- Collect statistics on a job in progress.
- Detect problems (e.g. count the number of errors).

# Hadoop Counters

To create a Counter, declare an `Enum`. (for example, in the Driver):

```
public enum GRAPH_COUNTERS {
    NB_NODES_UNFINISHED // Amount of non-black nodes after one execution cycle
};
```

The Counter can then be retrieved from either the Job object (in the Driver) or the Context object (in Mapper/Reducer):

```
Counters cn = job.getCounters();
Counter cnt = cn.findCounter(GRAPH_COUNTERS.NB_NODES_UNFINISHED);
```

```
Counter cnt = context.getCounter(GRAPH_COUNTERS.NB_NODES_UNFINISHED);
```

# Hadoop Counters

To read a counter value (in Driver/Mapper/Reducer):

```
Counter cnt;
// ...
Long value = cnt.getValue();
```

To increment (only in Mapper/Reducer):

```
Counter cnt;
// ...
cnt.increment(1);
```

Note: Can also be incremented by more than 1.

# Hadoop Counters

Our stopping condition function in Driver becomes :

```
private static Boolean stoppingCondition(Job previous_job) {
    Counters cn = previous_job.getCounters();
    Counter cnt = cn.findCounter(GRAPH_COUNTERS.NB_NODES_UNFINISHED);
    // If the counter has never been incremented:
    // we have not encountered a node that was not black
    // in the Reducers of the specified Job execution.
    return(cnt.getValue() == 0);
}
```

# Hadoop Counters

And at the end of our Reducer, let's add something like :

```
if(!new_node_color.equals("BLACK"))
    context.getCounter(Graph.GRAPH_COUNTERS.NB_NODES_UNFINISHED).increment(1);
```

In this way, the Reducer communicates the stopping condition to the Driver and we avoid having to re-read the output.

# Hadoop Counters

Hadoop also has internal counters; most of them can be found in TaskCounter and JobCounter (package org.apache.hadoop.mapreduce).

Here are a few examples:

- TOTAL_LAUNCHED_MAPS: The number of MAP tasks that have been launched. Includes tasks that have been launched speculatively.
- TOTAL_LAUNCHED_REDUCES: The number of REDUCE tasks that have been launched. Includes tasks that have been launched speculatively.
- NUM_FAILED_MAPS: The number of failed MAP tasks.
- NUM_FAILED_REDUCES: The number of failed REDUCE tasks.
- REDUCE_INPUT_GROUPS: The number of distinct key-value groups sent to the Reducers.

# InputFormat Class

- Until now: the input was always a text file on HDFS, divided by line.
- Produces key/value pairs: the key passed to the Mapper is the **row number**, the value is the **row itself**.
- Default Hadoop behavior - can be modified.
- Reading, splitting and interpreting input data is handled by a class called **InputFormat**.

# InputFormat Class

The InputFormat class influences :

- Source data **retrieval**: HDFS, databases, etc.
- **Splitting** of input data by the **InputSplit** class.
- **Reading** of input data by the **RecordReader** class.
- Types of key/value pairs passed to the Mapper (from read input data).

# InputFormat Class

- Hadoop provides several InputFormat classes.
- To use a different InputFormat class instead of the default:

```
job.setInputFormatClass(class);
```

The default class is:

```
job.setInputFormatClass(TextInputFormat.class);
```

Behavior can often be influenced (by the configuration object or static methods).

# InputFormat Class

**KeyValueTextInputFormat:**

- Cuts input file(s) by line.
- Expects to find (key; value) pairs within each line.
- Default separator for key and value is the tab character.

Example data:

```
1   2,5|GREY|0
2   3|WHITE⊢1
```

The character used for separation can be modified as follows:

```
conf.set("mapreduce.input.keyvaluelinerecordreader.key.value.separator", ";");
```

# InputFormat Class

**FixedLengthInputFormat:**

- Trims input file(s) to a specified fixed size.
- The key is the fragment number, the value is the fragment itself.
- Particularly suitable for binary data where a series of “blocks” of equal size follow one another.

Example: read RNA code by codon (three characters).

```
GCUACGGAGCUUCGGAGCUAGGCUACGGAGCUUCGGAGCUAG
```

Block size can be set with:

```
conf.setInt(FixedLengthInputFormat.FIXED_RECORD_LENGTH, 3);
```

or

```
FixedLengthInputFormat.setRecordLength(conf, 3);
```

# InputFormat Class

**NLineInputFormat:**

- Cuts the input file(s) into groups of N lines.
- The key is the number of the block of N lines.
- The value is the line content.

Example: read in groups of two lines:

```
It's a two lines
long line.
And here
is another one.
```

The number N of lines can be set with:

```
conf.setInt(“mapreduce.input.lineinputformat.linespermap”, 2);
```

```
NLineInputFormat.setNumLinesPerSplit(job, 2);
```

# InputFormat Class

**Sequence Files:**

- Apache format for storing key-value pairs.
- Supports binary format.
- Supports compression.
- More efficient; avoids overload problems with many small files.
- Often used with Hadoop and Big Data frameworks in general.

# InputFormat Class

**SequenceFileInputFormat:**

- Reads “Sequence File” files.
- Keys and values are extracted from the sequence file; their data types vary according to the file.
- Often used in production, although text format is also common.
- Sequence files are commonly created with Hadoop programs; often via a map operation only: no shuffle, no reduce - key/value pairs produced by map are written directly to the output. Can be achieved with :

```
// In the Driver
job.setNumReduceTasks(0);
```

# InputFormat Class

- Many other InputFormat classes are available.
- Not all are based on HDFS files; for example, MongoInputFormat with the MongoDB connector.
- For specific use cases, you can create your own InputFormat.

# Custom InputFormat Class

**To create your own InputFormat:**

- The new class must inherit from an existing **InputFormat: FileInputFormat, DBInputFormat, CompositeInputFormat ...**
- Create a class inheriting **RecordReader**.
- (Re)implement the relevant methods in both.

# Custom InputFormat Class

Example:

- An InputFormat to read the data from our "Common Friends" example as appropriate key/value pairs right from the start.
- As a reminder, the input data is :

```
A ⇒ B, C, D
B ⇒ A, C, D, E
C ⇒ A, B, D, E
D ⇒ A, B, C, E
E ⇒ B, C, D
```

# Custom InputFormat Class

Main Hadoop objects for working with input data from files:

- **InputSplit** is a block containing data; it is returned by the InputFormat's getSplits() method. For FileInputFormat, by default: InputSplit ~= a block on HDFS.
- **RecordReader** reads **InputSplit**; generates tuples from it.
- **InputFormat** encapsulates and creates both.

# Custom InputFormat Class

Our custom **InputFormat** class for the "Common Friends" example:

```
public class FriendsInputFormat extends FileInputFormat<Text, Text> {
    public RecordReader<Text, Text> createRecordReader(InputSplit split, TaskAttemptContext context)
    throws IOException, InterruptedException {
        return new FriendsRecordReader();
    }
}
```

# Custom InputFormat Class

Our custom **RecordReader** for the "Common Friends" example: (1/3)

```
public class FriendsRecordReader extends RecordReader<Text, Text> {
    private LineRecordReader lineRecordReader = null;
    private Text key = null;
    private Text value = null;

    public void initialize(InputSplit split, TaskAttemptContext context)
    throws IOException, InterruptedException {
        close();
        lineRecordReader = new LineRecordReader();
        lineRecordReader.initialize(split, context);
    }
```

# Custom InputFormat Class

Our custom **RecordReader** for the "Common Friends" example: (2/3)

```
    public boolean nextKeyValue() throws IOException, InterruptedException {
        if(!lineRecordReader.nextKeyValue()) {
            key = null;
            value = null;
            return false;
        }
        Text line = lineRecordReader.getCurrentValue();
        String str = line.toString();
        String[] arr = str.split("⇒");
        key = new Text(arr[0].trim());
        value = new Text(arr[1].trim());
        return true;
    }
    public Text getCurrentKey() throws IOException, InterruptedException {
        return key;
    }
```

# Custom InputFormat Class

Our custom **RecordReader** for the "Common Friends" example: (3/3)

```
    public Text getCurrentValue() throws IOException, InterruptedException {
        return value;
    }
    public float getProgress() throws IOException, InterruptedException {
        return lineRecordReader.getProgress();
    }
    public void close() throws IOException {
        if(lineRecordReader != null) {
            lineRecordReader.close();
            lineRecordReader = null;
        }
        value = null;
    }
}
```

# Custom InputFormat Class

Notes:

- We have access to the configuration object through the context; we can allow the end user to adjust the behavior of our InputFormat using configuration variables.
- We re-used the LineRecordReader = the RecordReader of the default InputFormat - TextInputFormat.
- This is one of the most common approaches; it allows us to easily read by line. Alternatively, we can implement the InputSplit reading manually, using the HDFS API for example.

# Custom InputFormat Class

An example of manual InputSplit management:

```
public void initialize(InputSplit isplit, TaskAttemptContext ctx) throws IOException {
    FileSplit split = (FileSplit) isplit;
    Configuration conf = context.getConfiguration();
    start = split.getStart();
    end = start + split.getLength();
    Path file = split.getPath();
    FileSystem fs = file.getFileSystem(job);
    fileIn = fs.open(file);
    fileIn.seek(start);
}
```

# Custom InputFormat Class

And in the same class, to read one line and save the key/value pair:

```
public boolean nextKeyValue() throws IOException {
    // ...
    Text line = new Text();
    size = fileIn.readLine(line, MAX_LENGTH);
    String[] arr = line.toString().split("⇒");
    key = new Text(arr[0].trim());
    value = new Text(arr[1].trim());
    // ...
}
```

# Custom InputFormat Class

**And concerning the (input) splits:**

- Behavior can be modified by overloading `getSplits()` in InputFormat.
- Returns a list of InputSplit objects.
- Rarely necessary for file input formats (default behavior is usually fine).

# OutputFormat Class

- Equivalent to the InputFormat class, but for output data.
- Default behavior: one output tuple per line, key then tab then value.
- Can be modified in the same way as the input format:

```
job.setOutputFormatClass(class);
```

For example (default output format class):

```
job.setOutputFormatClass(TextOutputFormat.class);
```

# OutputFormat Class

There are a number of standard **OutputFormat** classes provided by Hadoop:

- **SequenceFileOutputFormat**: writes sequence files.
- **MultipleOutputFormat**: writes output key/value pairs to different locations on HDFS according to tuple keys or values.
- There are others, whose behavior can also be modified via configuration parameters.

# OutputFormat Class

For example, to adjust the separator in the default **TextOutputFormat** class (to use semicolons instead of tabs):

```
conf.set("mapreduce.output.textoutputformat.separator", ";");
```

- As with **InputFormat**, for more complex needs, we can create our own OutputFormat.
- A very similar approach: simply create the **OutputFormat** and **RecordWriter** classes.

# OutputFormat Class

**The OutputFormat must mainly:**

- Open an output file on HDFS for the results of a Reduce operation.
- Instantiate and return a RecordWriter object.
- The **RecordWriter** writes key/value pairs to the output file.

# Custom OutputFormat Class

Example:

- We now want to create our own OutputFormat for our "Common Friends" example.
- We want the final data to be written as follows:

```
A-B ⇒ C, D
A-C ⇒ B, D
A-D ⇒ B, C
B-C ⇒ A, D, E
B-D ⇒ A, C, E
...
```

# Custom OutputFormat Class

Our custom OutputFormat class:

```
public class FriendsOutputFormat extends FileOutputFormat<Text, Text> {
    public RecordWriter<Text, Text> getRecordWriter(TaskAttemptContext context)
    throws IOException, InterruptedException {
        Path path = getDefaultWorkFile(context, "");
        FileSystem fs = path.getFileSystem(context.getConfiguration());
        FSDataOutputStream fileOut = fs.create(path, context);
        return(new FriendsRecordWriter(fileOut));
    }
}
```

# Custom OutputFormat Class

Our RecordWriter class:

```
public class FriendsRecordWriter extends RecordWriter<Text, Text> {
    private DataOutputStream out;

    public FriendsRecordWriter(DataOutputStream stream) {
        out = stream;
    }

    public void write(Text k, Text val) throws IOException, InterruptedException {
        out.writeBytes(k.toString() + " ⇒ " + val.toString() + "\n");
    }

    public void close(TaskAttemptContext ctx) throws IOException, InterruptedException {
        out.close();
    }
}
```

# Custom OutputFormat Class

We can also alter the name of the resulting files:

```
public class FriendsOutputFormat extends FileOutputFormat<Text, Text> {
    public RecordWriter<Text, Text> getRecordWriter(TaskAttemptContext context)
    throws IOException, InterruptedException {
        Path path = FileOutputFormat.getOutputPath(context);
        Path fullPath = new Path(
                path, FileOutputFormat.getUniqueFile(context, "RESULTS", ".txt")
        );
        FileSystem fs = path.getFileSystem(context.getConfiguration());
        FSDataOutputStream fileOut = fs.create(fullPath, context);
        return(new FriendsRecordWriter(fileOut));
    }
}
```

# Custom OutputFormat Class

Before changing the name of the files, the results were:

```
/out/_SUCCESS
/out/part-r-00000
/out/part-r-00001
/out/part-r-00002
...
```

And now, with our change:

```
/out/_SUCCESS
/out/RESULTS-r-00000.txt
/out/RESULTS-r-00001.txt
/out/RESULTS-r-00002.txt
...
```

# Writable Types

- So far, we've used "simple" types for keys and values in the examples: Text or IntWritable.
- Hadoop also allows you to define your own **Writable types**, for the key and value.
- If we wish to use our own types as the program's initial input or final output, we need to implement a dedicated InputFormat or, respectively, OutputFormat.

# Writable Types

Two options:

- If our type would be used only as a tuple value => implement the **Writable** interface.
- If our type would be used as both key and value of tuples => implement the **WritableComparable** interface.
- In both cases, serialization and deserialization must be handled by the specific class.

# Writable Types

**Example:**

- We want to create a Writable type to store our values in the "Common Friends" example.
- The class will be called `FriendsListWritable` and will store the list of friends linked to a social network user.

# Writable Types

## Example of a custom writable type: (1/2)

```
public class FriendsListWritable implements Writable {
    public ArrayList<String> friends;

    public void write(DataOutput out) throws IOException {
        Text line = new Text(get_serialized());
        line.write(out);
    }
    public FriendsListWritable(String datatxt) {
        unserialize(datatxt);
    }
    public void unserialize(String datatxt) {
        String[] data = datatxt.split(",");
        friends = new ArrayList<String>(Arrays.asList(data));
    }
```

# Writable Types

## Example of a custom writable type: (1/2)

```
    public FriendsListWritable() { }

    public String get_serialized() {
        String line = String.join(",", friends);
        return(line);
    }
    public void readFields(DataInput in) throws IOException {
        Text line = new Text();
        line.readFields(in);
        unserialize(line.toString());
    }
}
```

# Writable Types

To use our custom writable type as our output value we need to specify it in our **Driver**:

```
job.setInputFormatClass(FriendsInputFormat.class);
job.setOutputFormatClass(FriendsOutputFormat.class);
job.setOutputValueClass(FriendsListWritable.class);
```

And in our Mapper and Reducer:

```
public class FriendsMap extends Mapper<Text, FriendsListWritable, Text, FriendsListWritable>
```

```
public class FriendsReduce extends Reducer<Text, FriendsListWritable, Text, FriendsListWritable>
```

# Writable Types

We need to update our **InputFormat**:

```
public class FriendsInputFormat extends FileInputFormat<Text, FriendsListWritable> {
```

And also the **RecordWriter**:

```
public boolean nextKeyValue() throws IOException, InterruptedException {
    // ...
    String[] arr = str.split("⇒");
    key = new Text(arr[0].trim());
    value = new FriendsListWritable(arr[1].trim()));
    // ...
}
```

# Writable Types

To use the new Writable type as a key too, we need to implement the `WritableComparable` interface containing 2 additional methods:

- compareTo() :
  - similar to the usual Java compare function; compares two instances.
- hashCode() :
  - must return a value whose similarity is guaranteed for two identical objects - even across JVMs and different executions.

# Writable Types

- The hash code is used during the shuffle stage, to produce the partitions that are passed on to the reduce stage.
- Two identical hashcodes enable the shuffle operation to group identical objects (identical keys).
- Java objects have a hashcode() function; however, we can’t use it because according to the Java doc :
  - This hashcode does not need to remain consistent from one run of an application to another run of the same application.

# Writable Types

- As we're running our program on several machines, the hashcode() function will be called on identical objects but on different machines.
- As a result, although the keys are equal, they may not be grouped together.
- **Solution**: generate a hashcode ourselves, based on the object data
- In addition, the hashcode must be uniformly distributed.
- Standard Hadoop Writable types (Text, etc.) already offer such a hashcode function.

# Writable Types

In our "Common Friends" custom writable type example, we could update it as follows:

```
public class FriendsListWritable implements WritableComparable<FriendsListWritable> {
// ...
```

And:

```
    public int hashCode() {
        return(new Text(get_serialized()).hashCode());
    }
    public int compareTo(FriendsListWritable o) {
        int mysize = friends.size();
        int theirsize = o.friends.size();
        return (mysize < theirsize ? -1 : (mysize == theirsize ? 0 : 1));
    }
// …
```

# Spark

1. Spark
    1. Presentation
    2. High-level Architecture
    3. Architecture – RDDs
    4. Execution
    5. Python API
    6. Python API – Transformations
    7. Python API – Actions
    8. Python API – Broadcast Variables
    9. Python API – Accumulators
    10. Python API – Examples
    11. Python API – Conclusion
    12. Development environment
    13. Usage

# Limitations of Hadoop

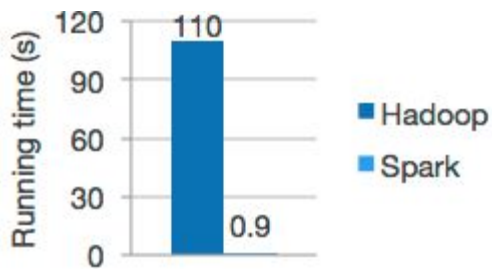
- Not interactive
  - Scheduling jobs takes time
- Only 2 main stages per job (Map/Reduce)
  - Intermediate results are written to HDFS = slow
- Complex development
  - Lot’s of Java boiler plate
  - Time consuming implementation

# Presentation

- A framework for distributed computing.
- Originally (2014) a project of the University of California, Berkeley
- Now open source software from the Apache Foundation.
- Official website: https://spark.apache.org/

# Presentation

- Significantly faster than Hadoop. Especially for tasks involving multiple Map and/or Reduce executions.
- Avoids repeated reads and writes to HDFS.
- Much more flexible than Hadoop. No rigid "map+shuffle+reduce" framework. Well suited to Machine Learning.
- Easy to use and develop.
- Highly integratable with other solutions. Can easily read data from various sources.



# Presentation

Developed in Scala (an object-oriented language derived from Java and including many aspects of functional languages).

Five supported languages :

- Scala
- Java
- Python (PySpark)
- R (SparkR) - deprecated in favor for `sparklyr`
- SQL

Performance and functionality more or less equivalent for all 4.

# Presentation

Consists of various libraries:

- Spark Core: Spark's core framework.
- Spark SQL: structured data component; unified data access and manipulation.
  - Includes the Datasets (typed) and DataFrames (untyped) APIs
- Spark MLlib: distributed Machine Learning library.
- Spark GraphX: distributed graph computation library.
  - deprecated / not maintained - alternatives: GraphFrames / openCypher
- Spark Structured Streaming: streaming processing library (continuous/near real-time).

This module is dedicated to Spark Core.

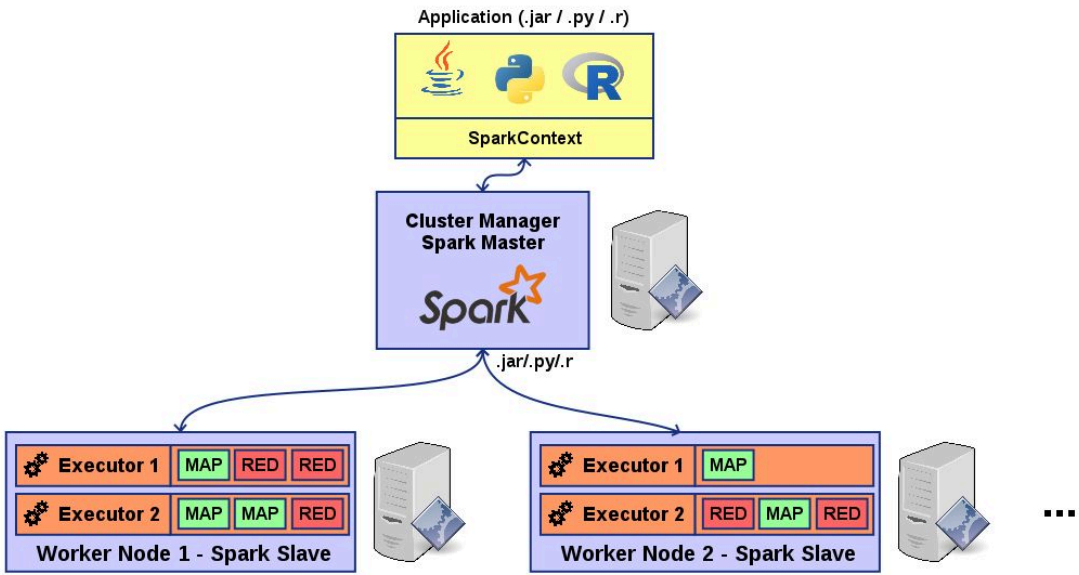
# Presentation

4 execution modes:

- Spark Standalone Cluster.
- Spark on YARN.
- Spark on Mesos. (Deprecated & Removed)
- Spark on Kubernetes (since december 2019 / Spark 2.3)

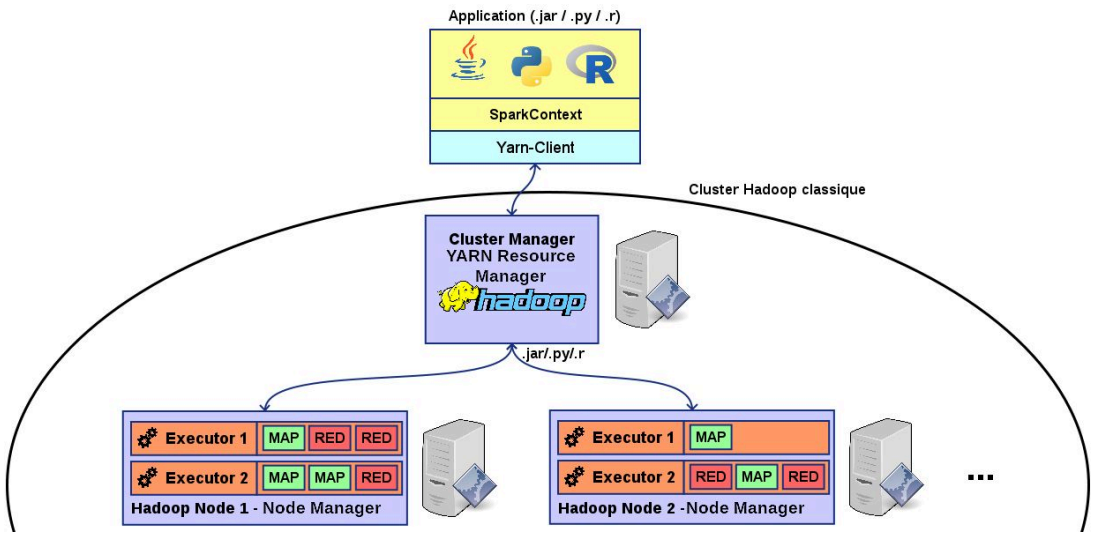
# High-level Architecture - Standalone Cluster

- One Spark Worker service per node; N Executor execution slots per worker.
- They all have a cache.
- The single cluster manager, Spark Master, is a single point of failure.
- High-availability configuration required.



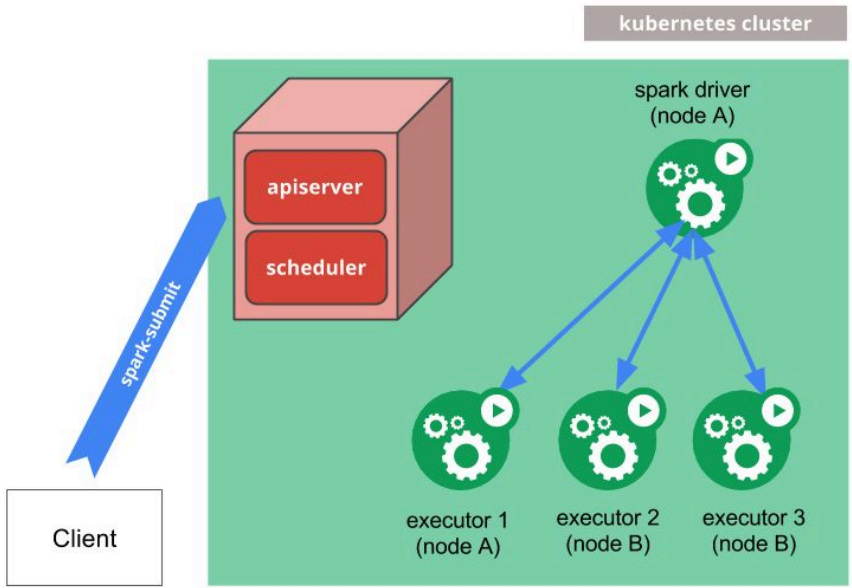
# High-level Architecture - YARN

- Spark Executor tasks are launched on YARN's NodeManagers.
- They remain running throughout execution, and maintain a similar cache.
- Spark often complements existing Hadoop infrastructures, and is currently the most common deployment mode in production.



# High-level Architecture - Kubernetes

- The cluster manager is the Kubernetes Master (kube-scheduler+kube-controller-manager).



# High-level Architecture

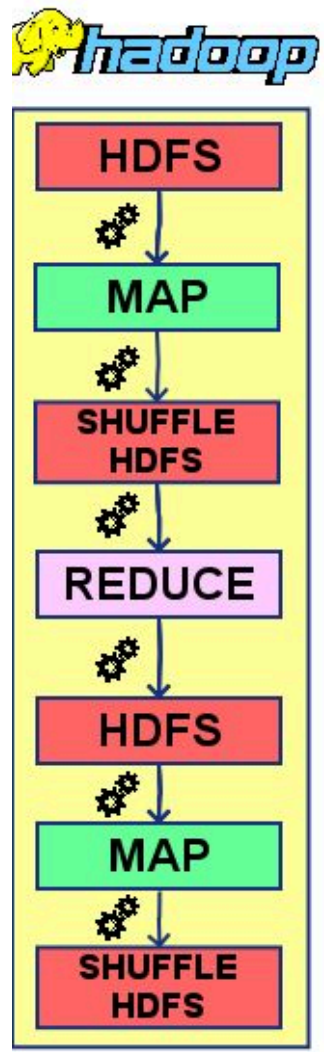
- Similar architectures - one resource manager per cluster, at least one 'executor' service on each node.
- The main code of a Spark program is called the `Driver`.
- There are two execution modes:
  - Client mode: the Driver runs where the program is initially launched.
  - Cluster mode: the Driver runs in a node on the cluster.

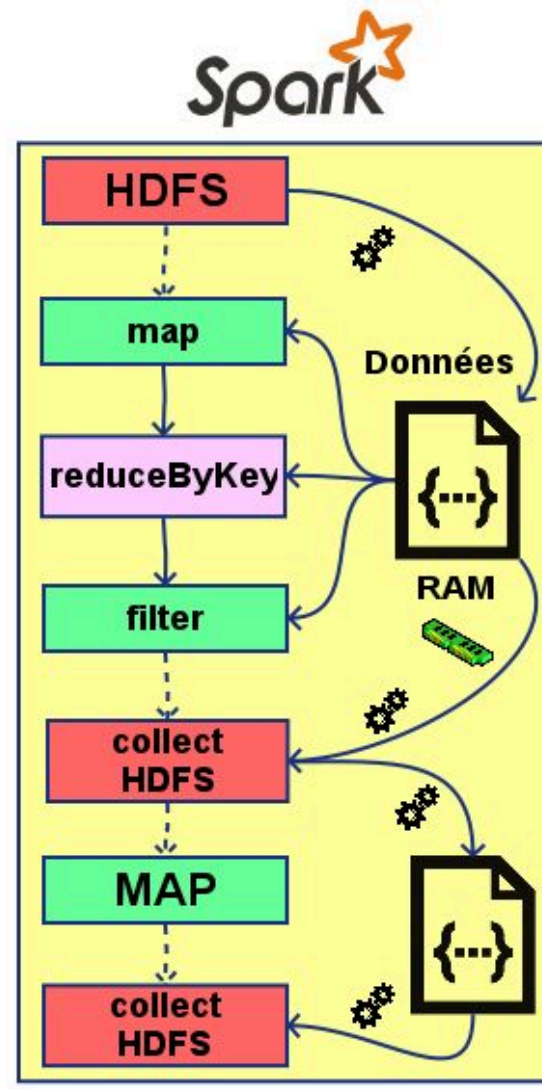
# High-level Architecture

- The Driver communicates with the cluster manager, requesting resources and execution slots.
- Fault tolerance is managed by the cluster manager.
- No dedicated persistent distributed storage; can use HDFS, or many other alternatives (other distributed file systems, databases, etc.).

# High-level Architecture

- Hadoop needs to serialize/deserialize data before, during and after each MapReduce run.
- Writing to HDFS is a slow operation.
- Many programs need to run multiple iterative MapReduce executions.
- Spark stores data in RAM and is able to determine when it needs to serialize/reorganize data, and only does so when necessary.
- It can be explicitly asked to keep data in RAM, when it's needed between several write steps.



# High-level Architecture - Notes

- Spark consumes much more RAM than Hadoop. The machines in the cluster therefore require more RAM.
- The cluster manager (“Spark Master”) is a single point of failure and requires a high-availability architecture.

Spark is preferred for :

- Complex tasks requiring multiple MapReduce iterations.
- Real-time/continuous data processing.

Hadoop is preferred for :

- Simple tasks requiring a single iteration of MapReduce.
- Cluster creation using low-end heterogeneous hardware (low-cost).

# Architecture – RDDs

Spark's main data abstraction: RDDs (Resilient Distributed Datasets)

These are large collections of various elements.

They are :

- Distributed (to enable multiple nodes to process the data)
- Partitioned (each node owns one or more parts of the RDD).
- Fault-tolerant (to limit the risk of data loss, RDDs are able to recalculate missing or damaged partitions due to node failures).
- Read-only (any change applied to an RDD results in the creation of a new RDD).

# Architecture – RDDs

Two operation types are available on RDDs:

- Transformation:
  - Creates a new, modified RDD.
  - Lazy evaluation: only executed when data access is required.
  - An Map operation on an RDD is a transformation.
- Action:
  - Accesses RDD data.
  - Causes its evaluation (application of all transformations one after the other).
  - Reading an RDD is an action.

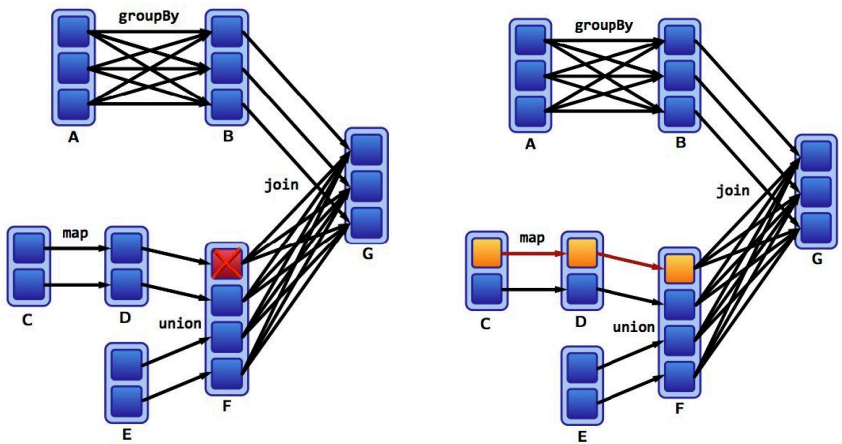
# Architecture – RDDs

Operations can be divided into wide and narrow operations:

- A narrow operation :
  - Performs an operation on the RDD “isolated by partition”. Does not change its partitioning.
  - Rather fast, as it does not require data to be moved over the cluster network.
- Wide operation:
  - Performs an operation on the RDD that requires access to more than one RDD partition.
  - Requires partial or complete shuffling of data between nodes.
  - Slower than narrow transformations.

# Architecture – RDDs

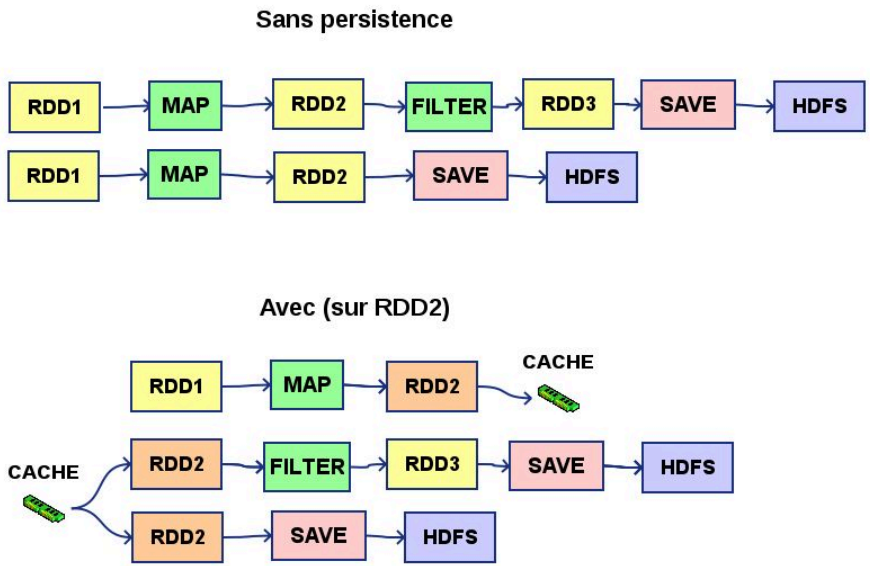
- Spark maintains the chronology of operations for RDD partitions; it can evaluate them, if necessary, several times for fault tolerance.
- No redundancy required (but possible): Spark recalculates lost data based on the chronology of operations from the source data.




Spark recalculates F from the original partition C

# Architecture – RDDs

- Two different tasks applying an operation to the same RDD cause this transformation to be executed twice.
- To avoid this, an RDD can be persisted in memory, so that it is calculated only once.
- When an RDD is persisted, it is kept in memory after the first evaluation.



# Execution

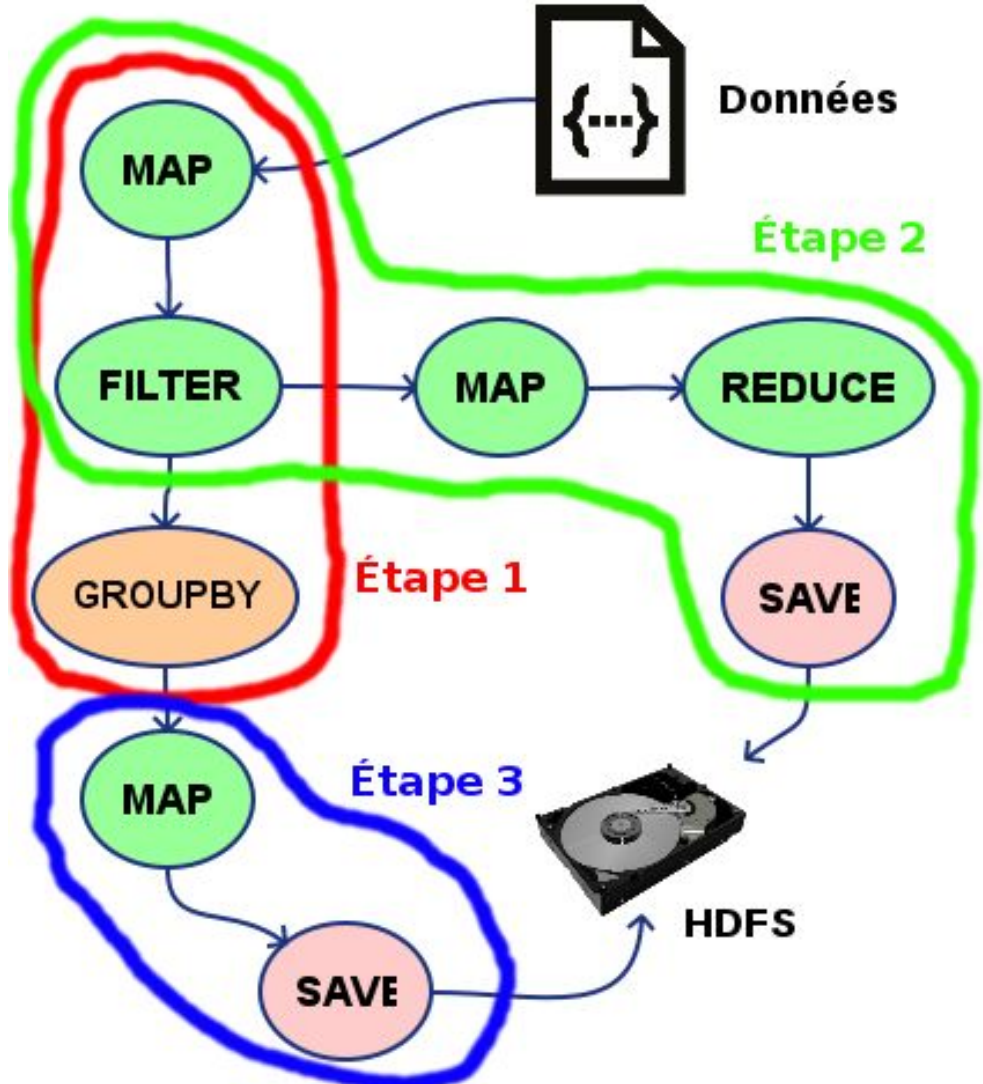
- Every Spark program consists of a Driver (a program containing a SparkContext object).
- The SparkContext interacts with the cluster. It has two main components: the Tasks Scheduler and the DAG Scheduler.
- When executing a program :
  - The DAG Scheduler constructs a Directed Acyclic Graph (DAG) with the actions and transformations defined in the program, and then divides the graph into stages, each containing a number of tasks: actions or transformations.
  - The Tasks Scheduler then submits the tasks to the cluster manager.

# Execution

The stages of the DAG are divided by :

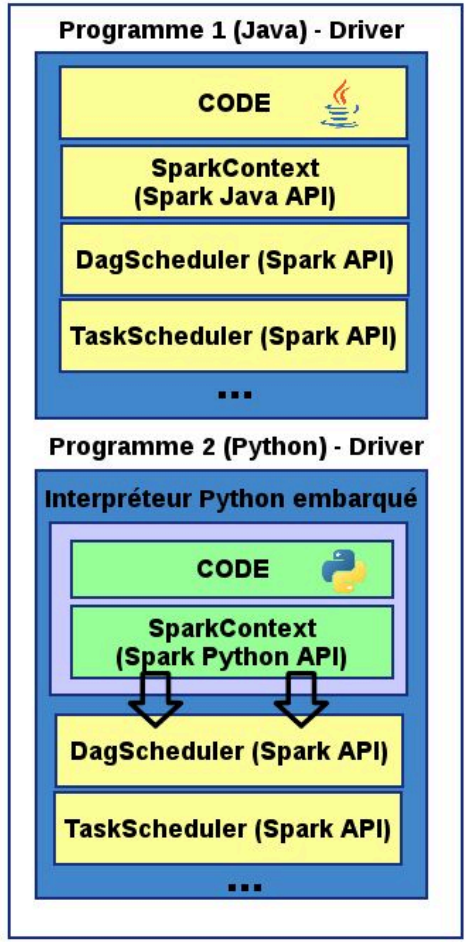
- Actions.
- Wide transformations.

The DAG lets Spark know when it should persist on disk.



# Execution

- Programs using the Spark API have performances close to native execution.
- Spark executes Python/R code via a corresponding embedded interpreter.



# Python API

- The Spark API is globally unified: the functions and similar from one language to another.
- The most important thing, therefore, is to be familiar with the various operations.
- The API presented here is that of Python; but the Scala, Java and (to a lesser extent) R APIs are similar.
- Most operations take an environment-independent function as a parameter (closure); in practice, this will often be reflected in the code by the use of lambda functions.

# Reminder - Lambda functions

Syntax of a lambda function in Python:

```
lambda ARG1, ARG2, ARG3, ... : RETURN_VALUE
```

More examples :

```
lambda x: x + 1
lambda a, b: a + b
lambda mot: (mot, 1)
```

# Reminder - Lambda functions

Example in Java (since version 8):

```
(x) → return(x+1)
(a, b) → { return(a+b)}
(mot) → { System.out.println(mot); return(Arrays.asList(mot, 1)) }
```

Example in Scala:

```
(x:Int) ⇒ (x+1)
(_:Int)+(_:Int)
(mot:String) ⇒ { println(mot); return(Seq(mot, "1")) }
```

# Python API

- The main class for accessing the Spark API is `SparkContext`.
- It is contained in the **pyspark module**.
- A Spark program generally starts with :

```
from pyspark import SparkContext
```

- **Note:** if you're using the interactive Shell `pyspark`, this import is already done and you already have a SparkContext instance with the `sc` variable.

# Python API

- To instantiate a `SparkContext`, the basic syntax is :

```
SparkContext(master=MASTER_URL, appName=DESCRIPTION)
```

- The `master` option specifies the cluster manager to contact to submit tasks for execution.
- The `appName` option provides a textual description of the application.
- Note: these options are **optional**. If not specified: defaults will be used from the spark configuration in `$SPARK_HOME/conf/spark-defaults.conf`, where `$SPARK_HOME` is the path to Sprak's installation directory.

Examples:

```
# Spark on YARN:
sc = SparkContext(master="yarn", appName="WordCount")
# Spark Standalone Cluster:
sc = SparkContext("spark://10.0.0.1/")
# Spark in local mode (simulates a cluster with N=2 nodes):
sc = SparkContext("local[2]", "WordCount")
# Spark on Kubernetes :
sc = SparkContext("k8s://https://192.168.1.254", "WordCount")
```

# Python API

- Alternative method to instantiate a `SparkContext`:

```
from pyspark import SparkContext, SparkConf

conf = SparkConf().setAppName("WordCount").setMaster("yarn")
sc = SparkContext.getOrCreate(conf=conf)
```

# Python API

> Note: SparkContext can only be instantiated once per JVM.

We can also retrieve an instance of SparkContext via Spark's DataFrame API:

```
from pyspark.sql import SparkSession
spark = (
  SparkSession.builder
  .master("local[*]")
  .appName("tp1")
  .getOrCreate()
)
sc = spark.sparkContext
```

# Python API – Reading data

- SparkContext has several functions for reading data and loading it into a first RDD.
- These functions all return a (pointer to) RDD.

A non-exhaustive list (to be called on the SparkContext object):

- **textFile(URL)** Loads one or more text files; the resulting RDD will have one element per line.
- **binaryFiles(URL)** Loads one or more binary files; the resulting RDD will have one element per file.
- **sequenceFile(URL)** Loads one or more sequence files. The resulting RDD will have one element per object in the file.
- **wholeTextFiles(URL)** Load entire text files. The resulting RDD will have one element per file.
- **pickleFile(URL)** Loads one or more serialized Python 'pickle' files containing an RDD.

# Python API – Reading data

- The URL argument can take the following forms, for example:

```
hdfs:///input/poeme.txt
file:///home/john/data.txt
hdfs:///results/part-r-*
```

- Note: all functions have an optional `numPartitions` argument - allowing you to specify the minimum number of partitions you wish to have within the RDD.
- Some of these functions also have other arguments, not described here, to fine-tune their operation.

# Python API – Creating data

- We can also create an RDD using the `parallelize` method of the SparkContext object.

```
parallelize(LIST, numSlices=NUM_PARTITIONS)
```

- The method converts the list passed as a parameter into an RDD (by performing partitioning), and returns a pointer to this RDD.
- For example:

```
rdd = sc.parallelize([0, 1, 2, 3, 4], numSlices=5)
rdd = sc.parallelize([f"item-#{x}" for x in range(0, 100)])
```

- Particularly useful for development and rapid testing.

# Python API – Note on Partitioning

- Partitions determine the level of parallelism of your program.
  - Too few partitions = low parallelism.
  - Too much partitions = slow shuffle process / network congestion.
- Spark tries to set the number of partitions automatically, but the general rule of thumb is to have **2-4 partitions for each CPU** in your cluster.

# Python API – Saving data

- RDDs have a number of functions for writing data to HDFS or disk.
- The contents of an RDD would be stored in a directory in several files
  - one per partition (named part-XXXXX, where XXXXX is an incremental number).

A non-exhaustive list of save functions (to be called on RDDs):

- **saveAsTextFile(URL)** Saves results in text files.
- **saveAsPickleFile(URL)** Saves results in Python 'pickle' files.
- **saveAsSequenceFile(URL)** Saves results in sequence files.

... there's more.

# Python API – Transformations

A non-exhaustive list of RDD transformations:

- **coalesce(numPartitions)** Returns an RDD resulting from re-partitioning the RDD with the specified number of partitions, which must be less than the current number of partitions.
- **distinct(numPartitions=None)** Returns an RDD where all duplicate elements of the original RDD are removed.
- **filter(FUNCTION(1 arg))** Apply the specified function to each element and, if true, keep the element in the return RDD.

# Python API – Transformations

- Example of `filter`:

```
rdd = sc.parallelize([1, 2, 3, 4, 5, 6])
rdd2 = rdd.filter(lambda x: x > 3)
rdd2.collect()
[4, 5, 6]
```

Example without using a lambda function:

```
def filtre(x):
  return(x % 2 == 0)

rdd = sc.parallelize([1, 2, 3, 4, 5, 6])
rdd2 = rdd.filter(filtre)
rdd2.collect()
[2, 4, 6]
```

# Python API – Transformations

- **map(FUNCTION(1 arg))** Returns an RDD resulting from the execution of the specified function on each element of the source RDD. Examples :

```
rdd = sc.parallelize(["celui", "ciel", "celui", "qui"])
rdd2 = rdd.map(lambda x: (x, 1))
rdd2.collect()
[("celui", 1), ("ciel", 1), ("celui", 1), ("qui", 1)]
rdd = sc.parallelize([1, 2, 3, 4])
rdd2 = rdd.map(lambda x: x*2)
rdd2.collect()
[2, 4, 6, 8]
```

# Python API – Transformations

- **flatMap(FUNCTION(1 arg))** Works like the map transformation, but if the function passed as an argument returns a list, each of its values becomes a distinct element of the resulting RDD. This allows the flatMap function to return fewer or more values. Example:

```
rdd = sc.parallelize(["un mot", "deux"])
rdd2 = rdd.map(lambda x: x.split())
rdd3 = rdd.flatMap(lambda x: x.split())
rdd2.collect()
[["un", "mot"], ["deux"]]  # map
rdd3.collect()
["un", "mot", "deux"]  # flatMap
```

# Python API – Transformations

- **mapValues(FONCTION(1 arg))** Applies the function passed as an argument to all the values of the RDD's key/value pairs. Requires that the source RDD contains key/value pairs. Example:

```
rdd = sc.parallelize([("qui", 20), ("ciel", 12)])
rdd.mapValues(lambda x: x*2).collect()
[('qui', 40), ('ciel', 24)]
# Ce qu'est d'ailleurs équivalent à :
rdd.map(lambda x: (x[0], x[1]*2)).collect()
[('qui', 40), ('ciel', 24)]
```

# Python API – Transformations

- **flatMapValues(FONCTION(1 arg))** This is the equivalent of flatMap but with the behavior of mapValues.

  Requires the source RDD to contain key/value pairs. Example:

```
rdd = sc.parallelize([("qui", 20), ("ciel", 12)])
rdd.flatMapValues(lambda x: (x - 5, 5)).collect()
[('qui', 15), ('qui', 5), ('ciel', 7), ('ciel', 5)]
#  Ce qu'est d'ailleurs équivalent à :
rdd.flatMap(lambda x: [(x[0], x[1] - 5), (x[0], 5)]).collect()
[('qui', 15), ('qui', 5), ('ciel', 7), ('ciel', 5)]
```

# Python API – Transformations

- **reduceByKey(FONCTION(2 args))** First groups RDD key/value pairs by key. Then applies the function passed as an argument to reduce each group and returns (a pointer to) the resulting RDD. Requires the source RDD to contain key/value pairs. This is the equivalent of a Hadoop Shuffle + Reduce. Example:

```
rdd = sc.parallelize([("qui", 1), ("ciel", 1), ("qui", 1), ("ciel", 1), ("qui", 1)])
rdd.reduceByKey(lambda a, b: a + b).collect()
[("qui", 3), ("ciel", 2)]
```

# Python API – Transformations

- **groupBy(FONCTION(1 arg))** Applies the function supplied as an argument to each element of the source RDD to generate a key. Then groups the RDD elements by the generated keys. Returns (a pointer to) the resulting RDD containing key/value pairs where the key is the one produced by the supplied function and the value is an iterable over the values of the group. Example:

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
res = rdd.groupBy(lambda x: x % 2).collect()
[(0, <pyspark.resultiterable ... >),
 (1, <pyspark.resultiterable ... >)]
print([(x, sorted(y)) for (x, y) in res])
[(0, [2, 4]), (1, [1, 3, 5])]
```

# Python API – Transformations

- **groupByKey()** Groups RDD elements by distinct key. Returns (a pointer to) the resulting RDD in a format similar to that of groupBy(). Requires the source RDD to contain key/value pairs.Example:

```
rdd = sc.parallelize(
    [("qui", 1), ("ciel", 1), ("qui", 1), ("ciel", 1), ("qui", 1), ("qui", 1)]
)
res = rdd.groupByKey().collect()
[("qui", <pyspark.resultiterable... >),
 ("ciel", <pyspark.resultiterable... >)]
print([(x, sorted(y)) for (x, y) in res])
[('qui', [1, 1, 1, 1]), ('ciel', [1, 1])]
```

... it's the equivalent of Hadoop's Shuffle stage (before Reduce).

# Python API – Transformations

- **intersection(RDD)** Returns an RDD created from the intersection with another RDD. Example:

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd2 = sc.parallelize([2, 4, 8])
rdd.intersection(rdd2).collect()
[2, 4]
```

- **union(RDD)** Returns an RDD created from the union with another RDD. Example:

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd2 = sc.parallelize([2, 4, 8])
rdd.union(rdd2).collect()
[1, 2, 3, 4, 5, 2, 4, 8]
```

# Python API – Transformations

- **subtract(RDD)** Returns an RDD created from subtraction with another RDD. Example:

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd2 = sc.parallelize([2, 4, 8])
rdd.subtract(rdd2).collect()
[1, 3, 5]
```

- **sortBy(FONCTION(1 arg))** Returns an RDD after sorting the source RDD; the function returns the key to be used for sorting. Example:

```
rdd = sc.parallelize([('z', 1), ('y', 2), ('m', 3), ('e', 4)])
rdd.sortBy(lambda x: x[0]).collect()
[('e', 4), ('m', 3), ('y', 2), ('z', 1)]
rdd.sortBy(lambda x: x[1]).collect()
[('z', 1), ('y', 2), ('m', 3), ('e', 4)]
```

# Python API – Transformations

- join(RDD)
- fullOuterJoin(RDD)
- leftOuterJoin(RDD)
- rightOuterJoin(RDD)

Performs joins between the current RDD and the specified RDD.

Requires source RDD to contain key/value pairs.

Joins are performed on keys. They return key/value pairs whose key is the one used for the join and whose value is a tuple composed of the values found in the first and second RDDs.

For fullOuterJoin, leftOuterJoin and rightOuterJoin, if a value has not been found in the other RDD, `None` is used.

# Python API – Transformations

- Exemples :

```
rdd = sc.parallelize([('a', '1'), ('b', 4), ('c', 9)])
rdd2 = sc.parallelize([('a', '0'), ('c', 2), ('d', 7)])

rdd.join(rdd2).collect()
[('a', ('1', '0')), ('c', (9, 2))]

rdd.fullOuterJoin(rdd2).collect()
[('a', ('1', '0')), ('c', (9, 2)), ('b', (4, None)), ('d', (None, 7))]

rdd.leftOuterJoin(rdd2).collect()
[('a', ('1', '0')), ('c', (9, 2)), ('b', (4, None))]

rdd.rightOuterJoin(rdd2).collect()
[('a', ('1', '0')), ('c', (9, 2)), ('d', (None, 7))]
```

# Python API – Transformations

- **setName(STR)** Allows you to give the RDD a name; useful for finding associated traces in Spark logs.
- **cache()** Requests to persist that RDD in memory. This call does not immediately trigger evaluation;
- **unpersist()** Requests to no longer persist that RDD in memory.

# Python API – Actions

- All actions trigger an RDD evaluation.

A non-exhaustive list of actions applicable to RDDs :

- **collect()** Retrieves the contents of an RDD in the form of a list, not recommended for large RDDs. Useful for development.
- **count()** Returns the number of elements in the RDD.
- **collectAsMap()** Like collect(), but returns a dictionary associating keys and values. Requires source RDD to contain key/value pairs.

# Python API – Actions

- **countByKey()** Returns the number of elements in the RDD for each distinct key as a dictionary. Requires source RDD to contain key/value pairs.

```
rdd=sc.parallelize([('qui', 1), ('qui', 3), ('ciel', 2), ('qui', 2)])
rdd.countByKey()
defaultdict(<type 'int'>, {'qui': 3, 'ciel': 1})
```

- **countByValue()** Returns the number of elements in the RDD for each distinct value as a hashmap.

```
rdd=sc.parallelize(["qui", "qui", "ciel", "qui", "croyait"])
rdd.countByValue()
defaultdict(<type 'int'>, {'qui': 3, 'croyait': 1, 'ciel': 1})
```

# Python API – Actions

- **first()** Returns the first element of the RDD.
- **getNumPartitions()** Returns the number of partitions in the RDD.
- **glom()** Returns an RDD where each element is a list containing all the values of a partition. Example:

```
rdd = sc.parallelize([1, 2, 3, 4, 5], 2)
rdd2 = sc.parallelize([1, 2, 3, 4, 5], 3)
rdd.glom().collect()
[[1, 2], [3, 4, 5]]
rdd2.glom().collect()
[[1], [2, 3], [4, 5]]
```

# Python API – Actions

- **max(key=FUNCTION(1 arg))** Returns the maximum element in the RDD; optionally with a function to call to return a numerical value for comparison.
- **min(key=FUNCTION(1 arg))** The inverse of max().

```
rddstr=sc.parallelize(["qui", "qui", "ciel", "qui", "croyait"])
rddint=sc.parallelize([4, 8, 15, 16, 23, 42])
rddint.max()
42
rddint.min()
4
rddstr.max(lambda x: len(x))
'croyait'
rddstr.min(lambda x: len(x))
'qui'
```

# Python API – Actions

- **reduce(FONCTION(2 args))** Recursively applies the supplied function to the RDD. Returns a single final value as the result. Example:

```
rdd2 = sc.parallelize([1, 2, 3, 4, 5])
print(rdd2.reduce(lambda a, b: a + b))
15

def red(a, b):
    if a > b:
        return(a)
    return(b)
print(rdd2.reduce(red))
5
```

# Python API – Actions

- **foreach(FUNCTION(1 arg))** Applies a function to each element of the RDD.
- **foreachPartition(FUNCTION(1 arg))** Applies a function to each of the RDD's partitions. Example :

```
rdd = sc.parallelize([4, 8, 15], 2)
rdd.foreach(lambda x: print(f"Value: {x}"))
Value: 4
Value: 8
Value: 15

rdd.foreachPartition(lambda x: print(f"Partition has {len(list(x))} elements."))
Partition has 1 elements.
Partition has 2 elements.
```

# Python API – Actions

- **isEmpty()** Returns true if the RDD is empty.
- **repartition(numPartitions)** Increases or decreases the number of partitions in the RDD. To decrease the number of partitions, use **coalesce** (less costly).
- saveAsTextFile
- saveAsPickleFile
- saveAsSequenceFile

Already discussed in the “Saving data” section.

# Python API – Broadcast Variables

- When a Spark program is run, the execution steps are sent to the various work nodes.
- All relevant data required is also sent. This includes local variables.
- For example:

```
multiplier = 2
rdd = sc.parallelize([4, 8, 15])
rdd.map(lambda x: x * multiplier).collect()
```

The 'multiplier' variable will be serialized and sent with the execution step for execution.

# Python API – Broadcast Variables

- If a local variable is used in several execution steps, it will be transferred several times.
- And if the variable is large, this causes high bandwidth utilization.
- To distribute such a variable efficiently and cache it on all the worker nodes in the cluster, Spark introduces the concept of broadcast variables.
- The use of broadcast variables avoids repeated transfers.
- The broadcast variables are, however, read-only.

# Python API – Broadcast Variables

- To create a broadcast variable: ('sc' = SparkContext object)

```
sc.broadcast(VARIABLE)
```

- The function returns the broadcast variable; its value can be accessed via `.value`.
- Example :

```
capitals = {"FR": "Paris", "DE": "Berlin", "UK": "London"}
capitals_bc = sc.broadcast(capitals)
rdd = sc.parallelize(["FR", "DE", "UK"])
rdd.map(lambda x: capitals_bc.value[x]).collect()
['Paris', 'Paris', 'Berlin’]
```

# Python API – Accumulators

- Transformations and actions in Spark are performed in a distributed manner.
- Updating local variables in transformations/actions will not update the local variable in the Driver: it will only update its copy on a worker node running the operation.
- To communicate information to the Driver, Hadoop offers counters.
- The equivalent in Spark: accumulators.

# Python API – Accumulators

- Like Hadoop Counters, Spark accumulators can only be incremented.
- The Driver can also increment accumulators.
- To declare an accumulator: sc.accumulator(INITIAL_VALUE)
- To increase its value: .add(INCREMENT)

```
count = sc.accumulator(0)
# ...
count.add(1)
```

# Python API – Accumulators

- Example :

```
invalid  = 0
valid = sc.accumulator(0)

def mapfunc(x):
    global invalid
    global valid
    invalid = invalid + 1
    valid.add(1)
    return(len(x))

data = sc.parallelize(["lake", "leak", "kale", "boat", "car"])
result = data.map(mapfunc)
print(f"invalide={invalid} ; valide={valid.value}")  # invalid=0 ; valid=0
# Here, ‘valid’ equals 0 because no evaluation has taken place.
result.collect()  # [4, 4, 4, 4, 3]
print(f"invalide={invalid} ; valide={valid.value}")  # invalid=0 ; valid=5
```

# Python API – Examples

Word count example using Spark Python RDD API

```
from pyspark import SparkContext

sc = SparkContext("local[2]", "Wordcount")
lines = sc.textFile("hdfs:///poeme.txt")
words = lines.flatMap(lambda x: x.split())
tuples = words.map(lambda x: (x, 1))
res = tuples.reduceByKey(lambda a, b: a + b)
res.saveAsTextFile("hdfs:///res")
```

# Python API – Examples

Anagram detector example:

Note the `.cache` usage to avoid double evaluation of the groupped RDD.

```
from pyspark import SparkContext

sc = SparkContext("local[2]", "Anagrammes")
words = sc.textFile("hdfs:///common_words_en_subset.txt")
tuples = words.map(lambda x: (''.join(sorted(list(x))), x))
grouped = tuples.groupByKey().mapValues(lambda x: list(x))
grouped.cache()
filtered = grouped.filter(lambda x: len(x[1]) > 1)
res = filtered.mapValues(lambda x: ", ".join(x))
res.saveAsTextFile("hdfs:///res-filtered")
res2 = grouped.mapValues(lambda x: ", ".join(x)) res2.saveAsTextFile("hdfs:///res-unfiltered")
```

# Python API – Examples

Word count example using Spark Scala RDD API

```
object WordCount {
    def main(args: Array[String]) {
        val conf = new SparkConf().setAppName("Wordcount")
        val sc = new SparkContext(conf)
        val lines = sc.textFile("hdfs:///poeme.txt")
        val words = lines.flatMap(line ⇒ line.split())
        val tuples = words.map(word ⇒ (word, 1))
        val res = words.reduceByKey(_+_)
        res.saveAsTextFile("hdfs:///res")
    }
}
```

# Python API – Conclusion

- There are, of course, many other functions, transformations and actions.
- There are other Spark APIs we haven't covered: SQL, MLlib, etc.
- For more information, please refer to the documentation: https://spark.apache.org/docs/latest/api/python/

# Development environment

- Java Spark API: same type of IDE as for Hadoop; with Gradle or Maven for dependencies.
- Scala Spark API: same thing.
- Spark Python API: a simple text editor may suffice; or an advanced Python IDE (PyCharm, Atom...).
- Spark R API: a simple text editor or an advanced R IDE (RStudio...).

# Development environment

During development:

- Using small subsets of large production data sets enables the use of collect(), glom() and similar.
- Such a representative subset can be extracted using the Spark command takeSample().
- For testing purposes, you can use the interactive Spark Shell.

# Usage

To submit an entire program to Spark (Java/Scala/Python/R):

```
spark-submit
```

To access the interactive Spark Shell (useful for quickly testing small Spark programs) :

- spark-shell (Scala)
- pyspark (Python)
- sparkR ®
- If you're using the Interactive Shell, you already have a `SparkContext` instance with the variable `sc`.
- By default, both commands run in local mode: they will execute the program on a simulated Spark cluster.

# Usage

The two Spark commands have two important arguments:

–master URL --deploy-mode client|cluster

–master allows you to specify the URL of the cluster manager.

- Accepts the same syntax used when instantiating the SparkContext object.
- Examples: 'spark://...' yarn', 'mesos://...', 'localN'. –deploy-mode allows you to specify the execution mode.
- Client mode: the Driver runs at the location where the program is initially launched.
- Cluster mode: the Driver runs in a node on the cluster.

# Usage

To submit a Spark Java or Scala program :

```
spark-submit [OPTIONS] --class DRIVER_CLASSPATH JARFILE
```

For example:

```
spark-submit --master yarn --deploy-mode cluster --class org.embds.wcount WordCount.jar
```

# Usage

To submit a Python program, we can also include additional libraries using the :

```
--py-files ZIPFILE
```

For example:

```
spark-submit --master yarn --deploy-mode cluster --py-files opencv.zip word_count.py
```

# Usage

- There are, of course, many other arguments to these two commands.
- For more information on their use, please refer to the documentation:https://spark.apache.org/docs/latest/

# Hadoop/Spark ecosystem

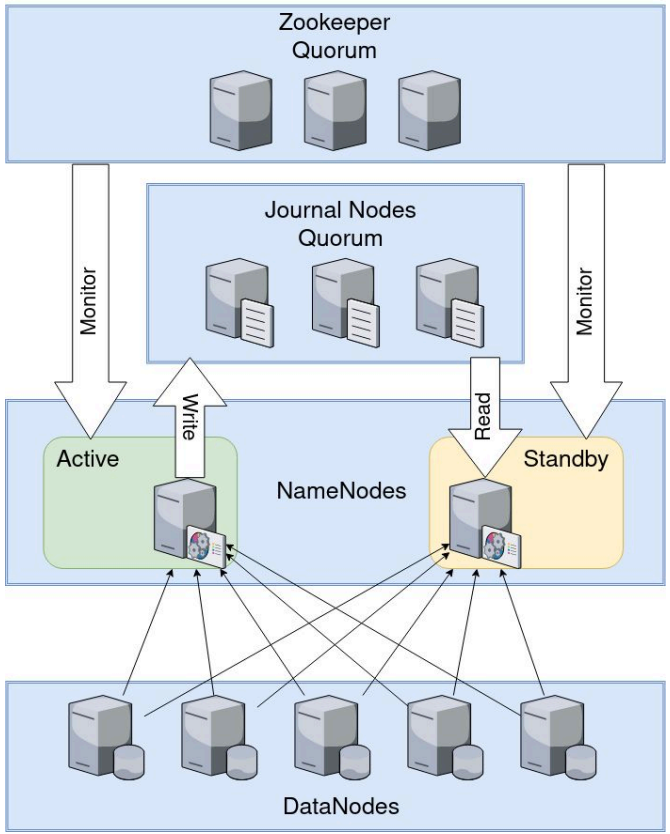
# Zookeeper

- Highly available service.
- Maintains coordination data. (Like a very reliable small distributed file system).
- Notifies customers of changes in this data.
- Monitors client failures.
- Can ensure high availability of HDFS and YARN services.
- Can do much more: implement consensus protocols, group management, leader election, etc.
- For more information: https://zookeeper.apache.org/

# Zookeeper Example HDFS High Availability

- There are several Namenodes deployed (but only one is active and the others are on standby).
- The active Namenode persists all changes made on HDFS on a shared NFS directory or via the Quorum Journal Manager service.
- In this example, Zookeeper is in charge of:
- Fault detection (Namenode operation monitoring).
- Election of a new Namenode as active in the event of a failure.

# Zookeeper Example HDFS High Availability



# Apache PIG

- Originally created at Yahoo.
- Is a platform for analyzing and processing massive data.
- Works with a high-level language (Pig Latin) that compiles into Map-Reduce tasks and executes them (in parallel) on a cluster (Hadoop or Spark).
- In this way, it abstracts the Java MapReduce programming part.
- For more information - see the official documentation: https://pig.apache.org/docs/r0.18.0/
- Note: Apache PIG is increasingly being replaced by Apache Hive and Apache Spark.
  - ~~(The latest PIG release 0.17 dates from 2017)~~ New PIG release 0.18 in 2025 🎉

Apache Pig

# Apache PIG - Word Count example (PigLatin)

Example :

```
-- Load the ‘poeme.txt’ file from HDFS line by line.
data = LOAD 'poeme.txt’ AS line;
-- Split each line by word.
words = FOREACH data GENERATE FLATTEN(TOKENIZE(line, ' ‘)) AS word;
-- Group by word.
grouped = GROUP words BY word;
-- Count words in group.
wordcount = FOREACH grouped GENERATE group, COUNT(words);
-- Save results in ‘wordcount-output’ directory on HDFS.
STORE wordcount INTO 'wordcount-output';
```

# Apache Hive

- Originally developed by Facebook.
- Is data management software that makes it easy to read and write large datasets residing in distributed storage using SQL.
- Works with Hive Query Language (HiveQL) - very similar to SQL.
- Often used for tasks such as extract/transform/load (ETL), reporting and data analysis.

# Apache Hive

- Interacts with files stored on HDFS using SQL. (users define the data schema for files)
- Supports many file formats - TXT, CSV, TSV, ORC, Parquet, AVRO, etc. but also custom formats.
- Hive offers 2 different types of tables:
  - External: The source data does not belong to Hive and if you delete the tables in Hive - its data will be retained.
  - Internal: The source data belongs to Hive, and if you delete the tables in Hive - its data will also be deleted.
- Can be interconnected with other data sources (e.g. Mysql, MongoDB, etc.) using ODBC (Open Database Connectivity) or JBDC (Java Database Connectivity) connectors.
- For further information: https://hive.apache.org/

# Apache Hive - HiveQL

```
$ # beeline est un CLI pour interagir avec Hive.
$ beeline
beeline> -- Connexion a Hive.
beeline> !connect jdbc:hive2://{HIVESERVER}:{PORT} {USERNAME}
beeline> -- Création d'une table Hive externe avec 2 colonnes pour des fichiers TSV (séparés par tab).
beeline> -- Chaque ligne dans ses fichiers dans le dossier "/dictionary" est sous forme "MOT  DESCRIPTION".
beeline>CREATE EXTERNAL TABLE dictionary (word STRING, description STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION '/dictionary';
beeline> -- Par exemple: Cherchons la description du mot "SWEET" dans notre table "dictionary".
beeline>SELECT description FROM dictionary WHERE word = "SWEET";
```

# Apache HBase

- Is an open source, NoSQL, distributed and fault-tolerant database, modeled after Google's BigTable.
- Provides BigTable-like capabilities on top of HDFS*.
- (can also work with local file system, S3 and other file systems BUT implicitly assumes data is stored reliably).
- Well-suited to real-time data processing or random, consistent read/write access to large volumes of data.
- Works well with Hive.
- Can be used as both input and output for map/reduce tasks.
- For further information: https://hbase.apache.org/

# MongoDB

- MongoDB is a document-oriented NoSQL DBMS (database management system).
- Initially proprietary (2007), it was later released under the AGPLv3 license (2009).
- Stores data in the form of documents written in a language close to JSON: BSON (Binary JSON).
- Extremely scalable, capable of storing very large quantities of data.
- It is developed and maintained primarily by MongoDB Inc. but is also used by numerous volunteers and third-party entities.
- A Hadoop connector is also available, making it possible to read the input data of a Hadoop program directly from MongoDB, and to write the results of such a program directly to MongoDB as well.
- Official website: https://www.mongodb.com/

# Apache Oozie

- Is a job scheduling system - manages and executes jobs on Hadoop.
- Can execute multiple Hadoop jobs in sequential and/or parallel order.
- 3 types of jobs (abstraction level):
  - Oozie Workflow jobs - represent sequences of actions to be executed.
  - Oozie Coordinator Jobs - trigger Oozie Workflow jobs according to time, available data or external events.
  - Oozie Bundles - represent a set of “Oozie Coordinator Jobs”.
- For more information: https://oozie.apache.org/

# Apache Airflow

- Airflow began as an open source project at Airbnb, then joined the official Apache Foundation in 2016.
- Is a platform for programmatically creating, scheduling and monitoring workflows (written as python scripts).
- Provides a powerful web interface for monitoring, scheduling and managing workflows.
- Highly extensible: integrations exist for many commonly used data engineering tools.
- For further information: https://airflow.apache.org/

# Apache Mahout

- Apache Mahout is an open-source library dedicated to linear algebra and Machine Learning on a distributed scale.
- Offers various algorithms for collaborative filtering, clustering, classification, etc.
- Is extensible, allowing rapid development of distributed algorithms using a mathematically expressive DSL (domain-specific language), similar to the R language.
- Initially ran on top of Hadoop. Now uses Apache Spark as its distributed computing kernel and can be extended to other distributed computing kernels.
- For further information: https://mahout.apache.org/

# Apache Ozone

- Highly scalable distributed File System (alternative to HDFS).
- Provides:
  - HDFS compatible API.
  - S3 compatible API.
- Scales to Exabyte & billions of objects.
- Organize data into Namespaces (Volumes) > Buckets > Keys.
- A bit slower than HDFS - but promising alternative (active development)

# Thank you for your attention

# Advanced Spark Development

1. Advanced Spark Development
    1. Spark SQL, DataFrames and Datasets
        1. Spark SQL, DataFrames and Datasets - Overview
    2. Spark Structured Streaming
    3. Spark MLlib

# Quick Recap on RRDs

- RDDs enable efficient data reuse (keeping intermediate data in memory)
- RDDs provide an interface based on coarse-gained transformations (map/filter)
- Fault tolerance is ensured by keeping track of the transformations applied on RRDs (allowing Spark to recompute lost RDD partitions)
- To learn more about RDDs you can checkout the RDD programming guide

# Spark SQL, DataFrames and Datasets

1. Advanced Spark Development
    1. Spark SQL, DataFrames and Datasets
        1. Spark SQL, DataFrames and Datasets - Overview

# Spark SQL, DataFrames and Datasets - Overview

- Dedicated for structured data processing.
- Build on top of the **RDD API**
- Provide more information about the structure of the data and the computation being performed
  - Leads to **more optimized processing** than RDDs

# Spark SQL, DataFrames and Datasets - Overview

- Higher level API
  - With **RDDs** you specify how you want to do your computation.
  - With **DataFrames/Datasets/SQL** you specify what result you want to get.

# Spark SQL, DataFrames and Datasets - Overview

A **Dataset**:

- Represents a distributed two-dimensional labeled data structure (Like a table with rows & columns).
- Provides a **strongly-typed** API (available with Scala/Java).
- Lazy evaluation (like in RDDs).
- Supports schema evolution and dynamic typing.

# Spark SQL, DataFrames and Datasets - Overview

A **Dataframe**:

- Is an **untyped** equivalent of a **Dataset** (available with Scala/Java/python/R)

# Spark SQL, DataFrames and Datasets - Overview

**Spark SQL**:

- Allows to run SQL queries on Spark DataFrames.
- Can be configured to read Apache Hive Tables as well.

# Spark SQL, DataFrames and Datasets - Overview

**Pandas on Spark API:**

- A Pandas API to interact with DataFrames as if they where pandas.DataFrame
- Inspired by Dask
- Aims to make the transition from pandas to Spark easy for data scientists.
- Does not officially support Spark Structured Streaming.

# Spark SQL, DataFrames and Datasets - Overview

- Sparks allows seamless switching between the SQL and DataFrame/Datasets APIs.

Example:

```
# Chaining DataFrame operations on SQL results
spark.sql("SELECT name, age FROM people").filter("age > 21").show()

# Changing from pandas-on-spark to Spark DataFrame.
import pyspark.pandas as ps
psdf = ps.range(10) # pandas-on-spark DataFrame
sdf = psdf.to_spark() # spark DataFrame
psdf = sdf.pandas_api() # back to pandas-on-spark DataFrame
```

# Spark SQL, DataFrames and Datasets - Overview

- Sparks allows also switching between RDDs and DataFrames/Datasets.

Example:

```
# Load a text file and convert each line to a Row.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))

# Infer the schema, and register the DataFrame as a table.
people_df = spark.createDataFrame(people)
schemaPeople.createOrReplaceTempView("people")

# SQL can be run over DataFrames that have been registered as a table.
teenagers = spark.sql("SELECT name FROM people WHERE age ⩾ 13 AND age ⩽ 19")

# Getting back access to the RRD.
people_df.rdd.map(lambda p: "Name: " + p.name).collect()
```

# Spark SQL, DataFrames and Datasets

**Useful links:**

- Getting Started with Spark's python API
- User Guide
- Programming Guide
- Data Sources
- API Reference

# Spark Structured Streaming

### What is streaming data?

> Continuously generated and unbounded data

Examples:

- DB change data feeds
- Clickstreams
- Application logs
- Application events
- IoT data

### Useful links:

- Programming Guide
- API Reference

# Spark MLlib

**Useful links:**

- Programming Guide
- API Reference