Hadoop

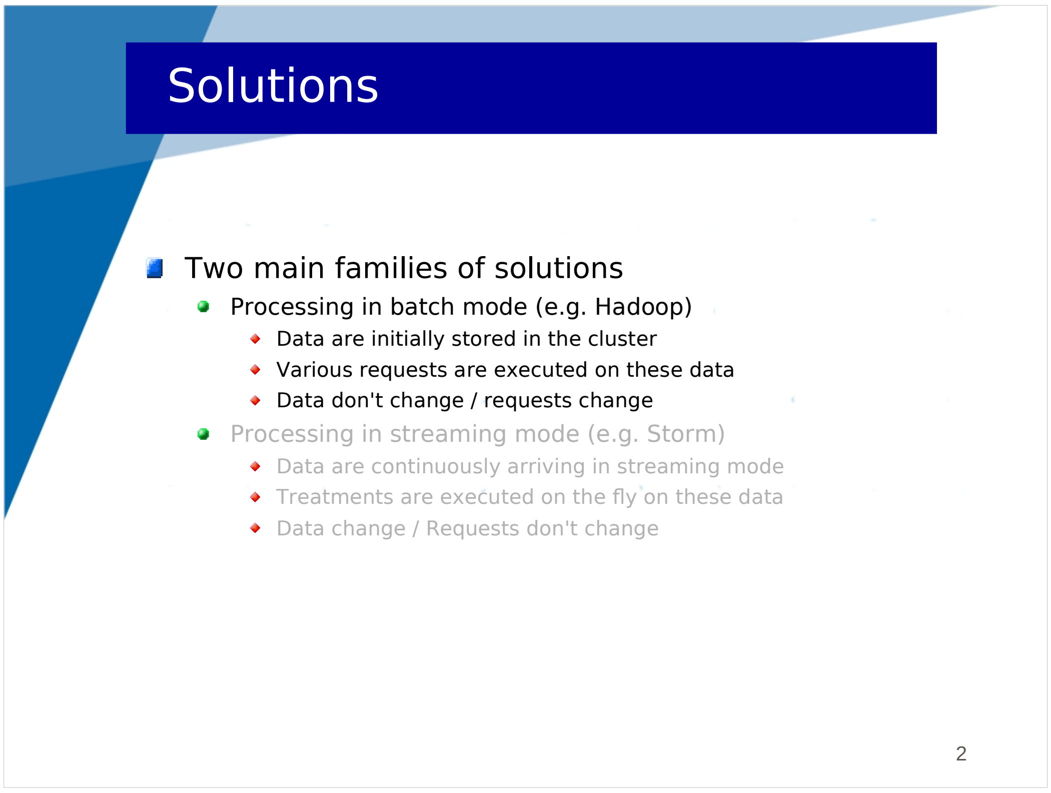
Daniel Hagimont

https://www.google.fr/search?q=daniel+hagimont+home+page

1

The lecture considers the first example of software infrastructure for the treatment of big data. This is HADOOP.

Hadoop was developed and popularized by Google.


Solutions

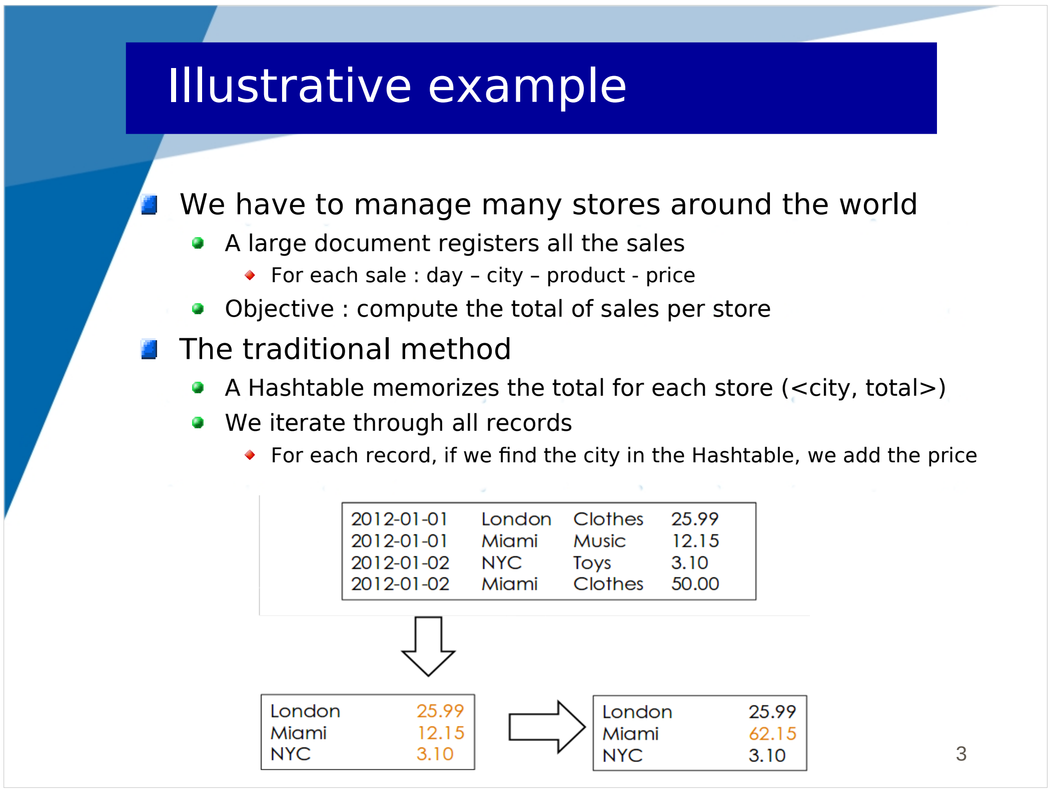
- Two main families of solutions
  - Processing in batch mode (e.g. Hadoop)
    - Data are initially stored in the cluster
    - Various requests are executed on these data
    - Data don't change / requests change
  - Processing in streaming mode (e.g. Storm)
    - Data are continuously arriving in streaming mode
    - Treatments are executed on the fly on these data
    - Data change / Requests don't change

2

Hadoop is an instance of the first category: processing in batch mode.

Repeat from the introduction presentation :

Data are initially stored on the computers' disks. For instance a very large dataset is divided into blocks and the blocks are distributed on the machines. Then, various requests can be issued to analyze the data. Such a request is divided into sub-requests which will handle the blocks on the different machines (in parallel). What is important here is that data are installed in the cluster (installed means here that data don't change, they are here to be read and analyzed, not modified) and that many requests can be issued on the same dataset.



Illustrative example

- We have to manage many stores around the world
  - A large document registers all the sales
    - For each sale : day – city – product - price
  - Objective : compute the total of sales per store
- The traditional method
  - A Hashtable memorizes the total for each store (<city, total>)
  - We iterate through all records
    - For each record, if we find the city in the Hashtable, we add the price

2012-01-01	London	Clothes	25.99
2012-01-01	Miami	Music	12.15
2012-01-02	NYC	Toys	3.10
2012-01-02	Miami	Clothes	50.00

London	25.99
Miami	12.15
NYC	3.10

London	25.99
Miami	62.15
NYC	3.10

3

Let's use an illustrative example.

We consider the management of a set of stores geographically distributed.

A large document gathers records of all the sales, with for each sale : the day, the city, the product code and the price.

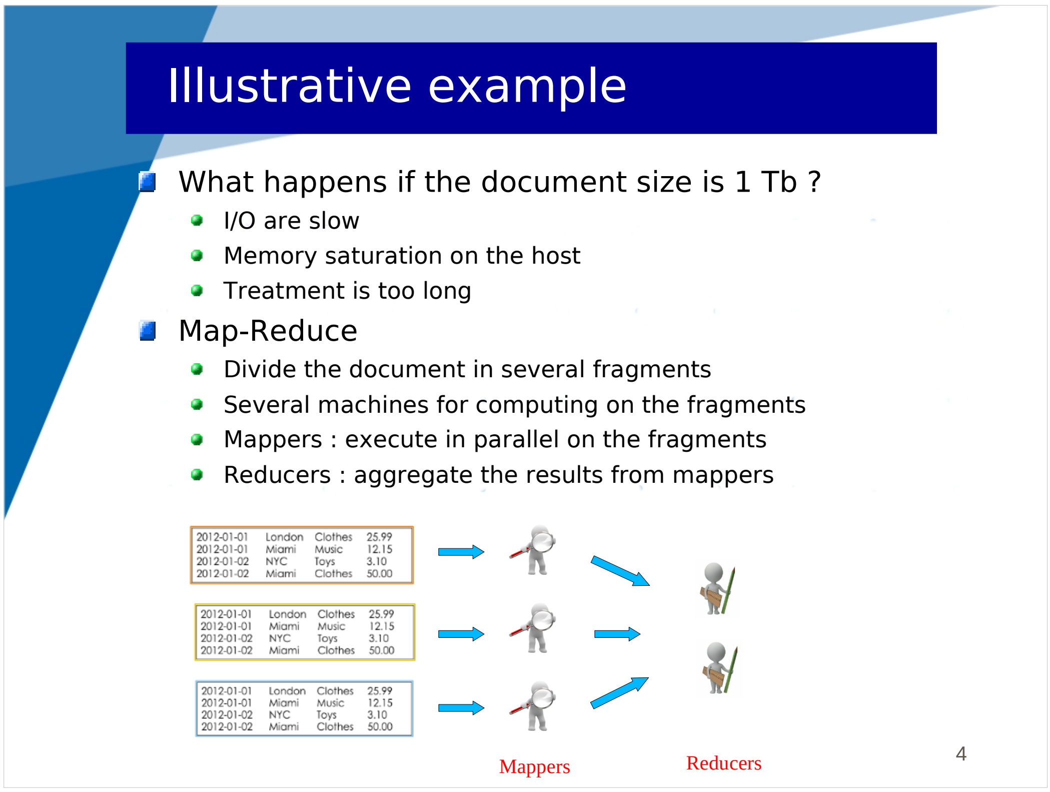
One request we may issue on the document is to compute for each store the total of the sales.

The traditional method is a sequential program which iterates over all the records. A hashtable registers the total of the sales for each store (<city, total>). In the iteration, for each record, we accumulate the prices for each store.

Figure top : the content of the document

Figure bottom left : the hashtable state after handling of the 3 first lines

Figure bottom right : the hashtable state after handling of the last line



Illustrative example

- What happens if the document size is 1 Tb ?
  - I/O are slow
  - Memory saturation on the host
  - Treatment is too long
- Map-Reduce
  - Divide the document in several fragments
  - Several machines for computing on the fragments
  - Mappers : execute in parallel on the fragments
  - Reducers : aggregate the results from mappers

2012-01-01	London	Clothes	25.99
2012-01-01	Miami	Music	12.15
2012-01-02	NYC	Toys	3.10
2012-01-02	Miami	Clothes	50.00

2012-01-01	London	Clothes	25.99
2012-01-01	Miami	Music	12.15
2012-01-02	NYC	Toys	3.10
2012-01-02	Miami	Clothes	50.00

2012-01-01	London	Clothes	25.99
2012-01-01	Miami	Music	12.15
2012-01-02	NYC	Toys	3.10
2012-01-02	Miami	Clothes	50.00

4

The traditional sequential execution is not satisfactory if the dataset to handle is very large (e.g. 1 TB).

- data are stored on disk and have to be loaded to be processed, and IO are slow

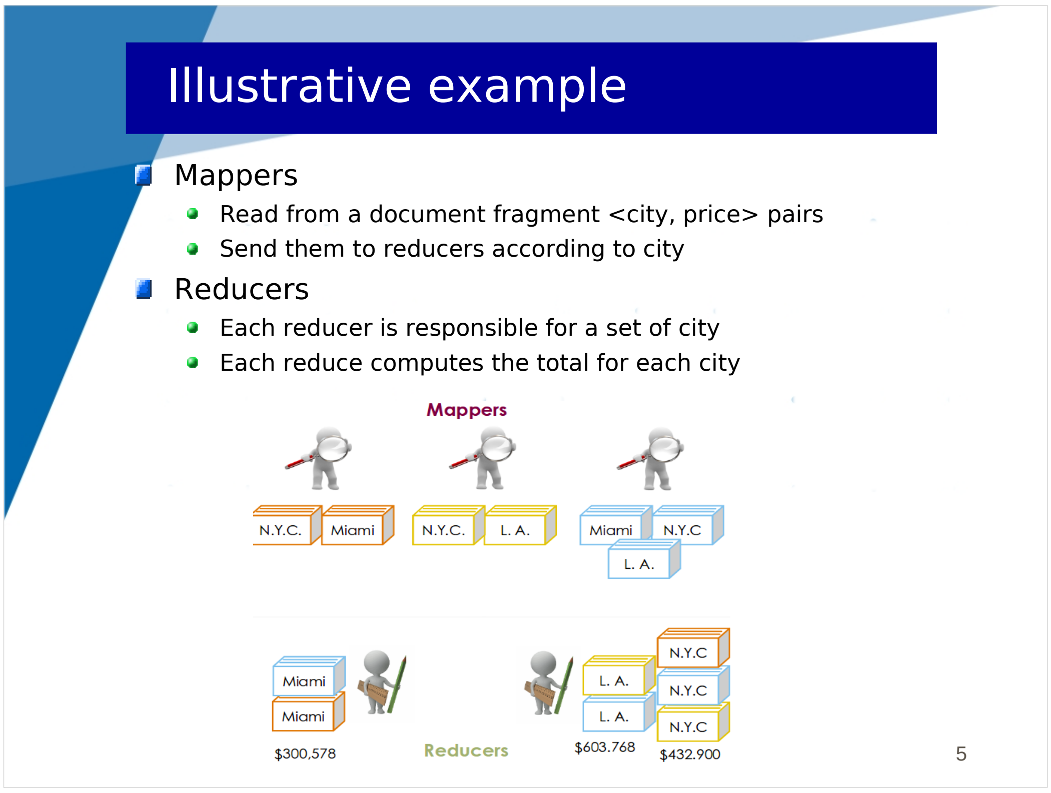
- memory can be a bottleneck as it is used to load data and also to store the hashtable

The sequential processing of the whole document may take a very long time depending on the size of the document.

The strategy introduced by Google is called map-reduce. It is a way to divide a request into many sub-requests. It is also a programming model which, when it is followed, helps the division into several sub-requests.

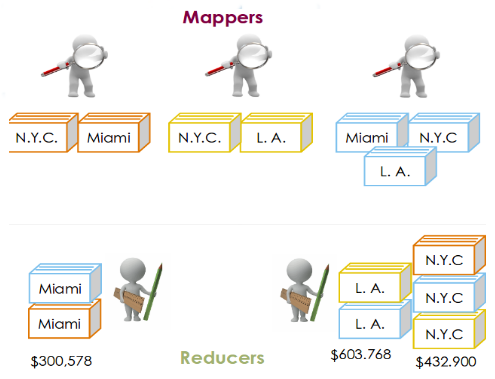
The principle is to divide the document (the data) into several fragments which are stored on different machines. Then there are 2 types of tasks executed on the machines. Mappers execute on each machine where a fragment is stored and process the local fragment. They generate results that are sent to Reducers which are responsible for aggregating the results.

Notice that Mappers and Reducers execute in parallel, thus improving performance for big documents.



Illustrative example

- Mappers
  - Read from a document fragment <city, price> pairs
  - Send them to reducers according to city
- Reducers
  - Each reducer is responsible for a set of city
  - Each reduce computes the total for each city



5

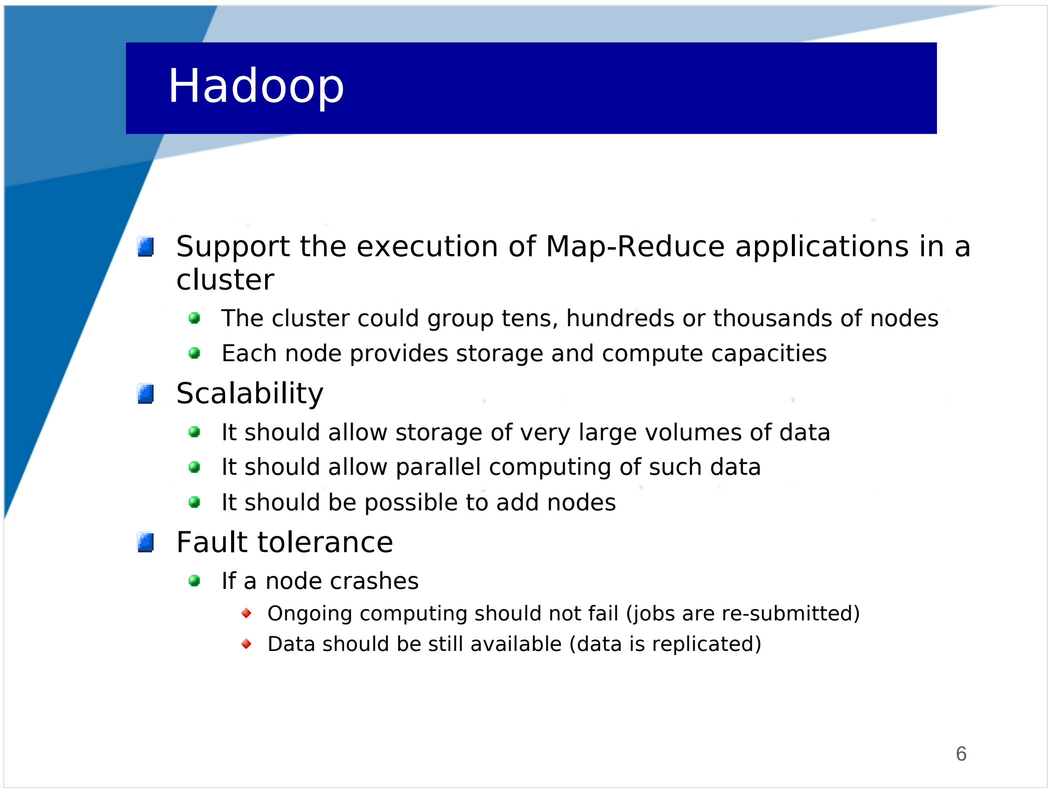
In our illustrative example, each Mapper executes on one fragment. It reads the fragment from disk, constructs <city, price> pairs and sends them to the Reducers.

Each reducer is responsible for generating a part of the final result, independently from other reducers. To enforce this independence between Reducers, it guarantees that a pair with a given city always goes to the same Reducer. This Reducer computes the total for that city. Therefore, the final result is the concatenation of the results (a table of <city, total>) from the different Reducers.

In the figure, each Mapper constructs and sends <city, price> pairs which are sent to the Reducers according to the city. Here, the first reducer receives pairs with the Miami city, while the second reducer receives pairs with the LA and NYC cities.

The first reducer generates a result : <Miami, 300>

The second reducer generates a result : <LA, 603>, <NYC, 432>



Hadoop

- Support the execution of Map-Reduce applications in a cluster
  - The cluster could group tens, hundreds or thousands of nodes
  - Each node provides storage and compute capacities
- Scalability
  - It should allow storage of very large volumes of data
  - It should allow parallel computing of such data
  - It should be possible to add nodes
- Fault tolerance
  - If a node crashes
    - Ongoing computing should not fail (jobs are re-submitted)
    - Data should be still available (data is replicated)

6

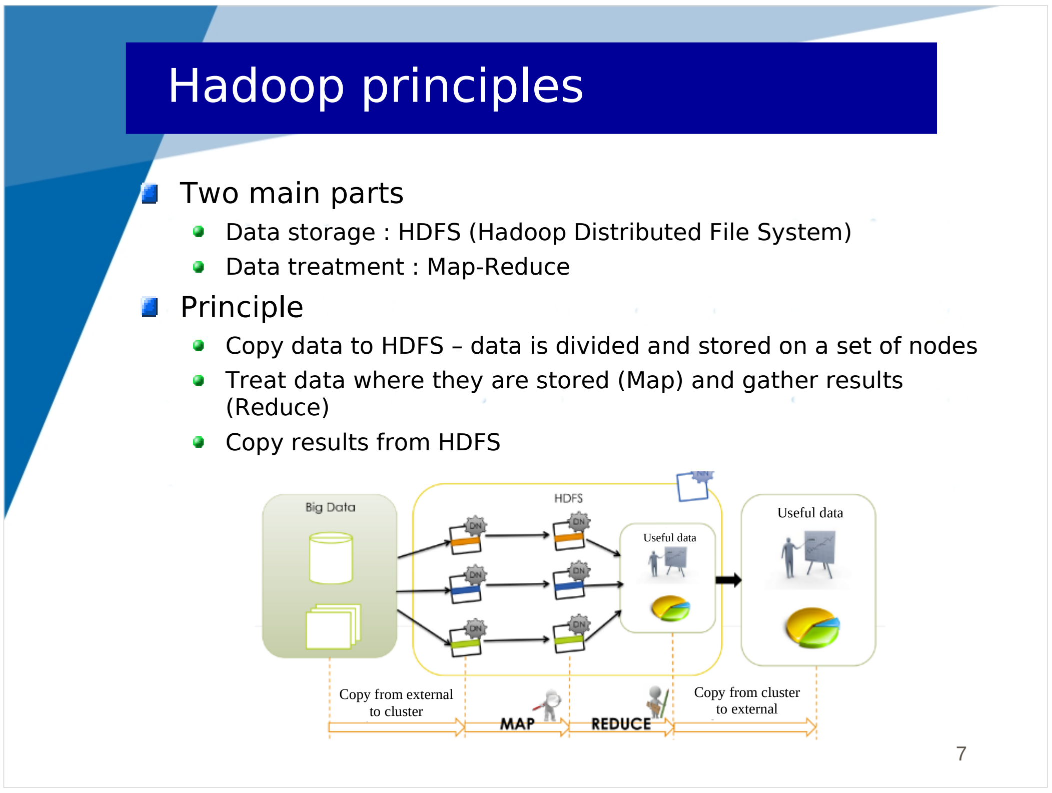
Hadoop is a framework (software infrastructure) introduced by google, which implements the map-reduce model. It allows the execution of map-reduce applications on a cluster of hundreds or thousands of machines. Each machine is supposed to have a independent storage (disk) to host fragments and a compute capacity for handling fragments.

Hadoop is scalable as it allows storing very large datasets and processing such data in parallel for reducing execution time. Scalability means here that by adding new nodes (machines), you increase the storage and computing capacity of the platform.

Hadoop is also fault tolerant :

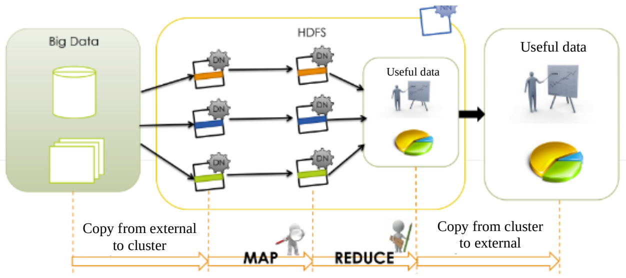
- regarding storage, fragments are replicated on several nodes, so that data is still available if a node crashes

- regarding computation, a task (mapper of reducer) which fails can be re-submitted, potentially on a different machine.



Hadoop principles

- Two main parts
  - Data storage : HDFS (Hadoop Distributed File System)
  - Data treatment : Map-Reduce
- Principle
  - Copy data to HDFS – data is divided and stored on a set of nodes
  - Treat data where they are stored (Map) and gather results (Reduce)
  - Copy results from HDFS



7

Hadoop is composed of 2 main parts :

- HDFS which is the Hadoop Distributed File System, allowing to store data (fragments) on the cluster's machines.

- Hadoop which is the engine allowing to execute map-reduce jobs.

The general usage principle :

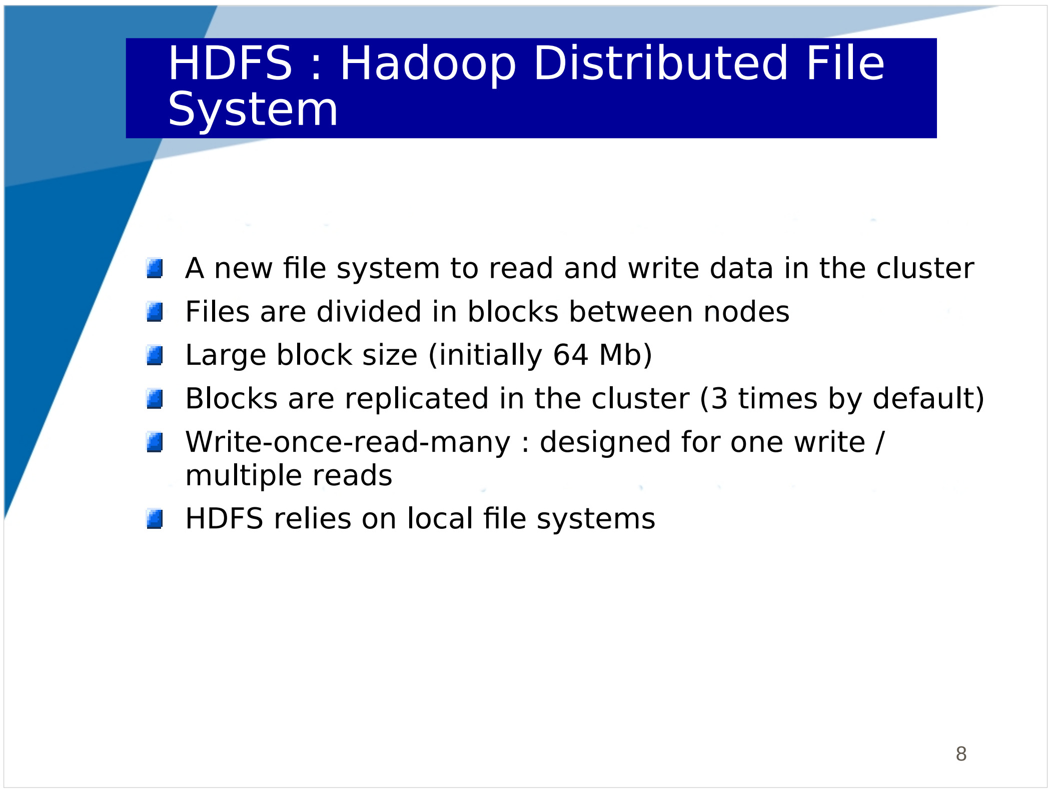
- your data are initially in an external storage (out of HDFS)

- you copy the data to HDFS

- the mappers and reducers are executed on the fragments

- the results (useful data) are available in HDFS

- you can copy the results from HDFS to the external storage



HDFS : Hadoop Distributed File System

- A new file system to read and write data in the cluster
- Files are divided in blocks between nodes
- Large block size (initially 64 Mb)
- Blocks are replicated in the cluster (3 times by default)
- Write-once-read-many : designed for one write / multiple reads
- HDFS relies on local file systems

8

HDFS is a file system distributed over the cluster.

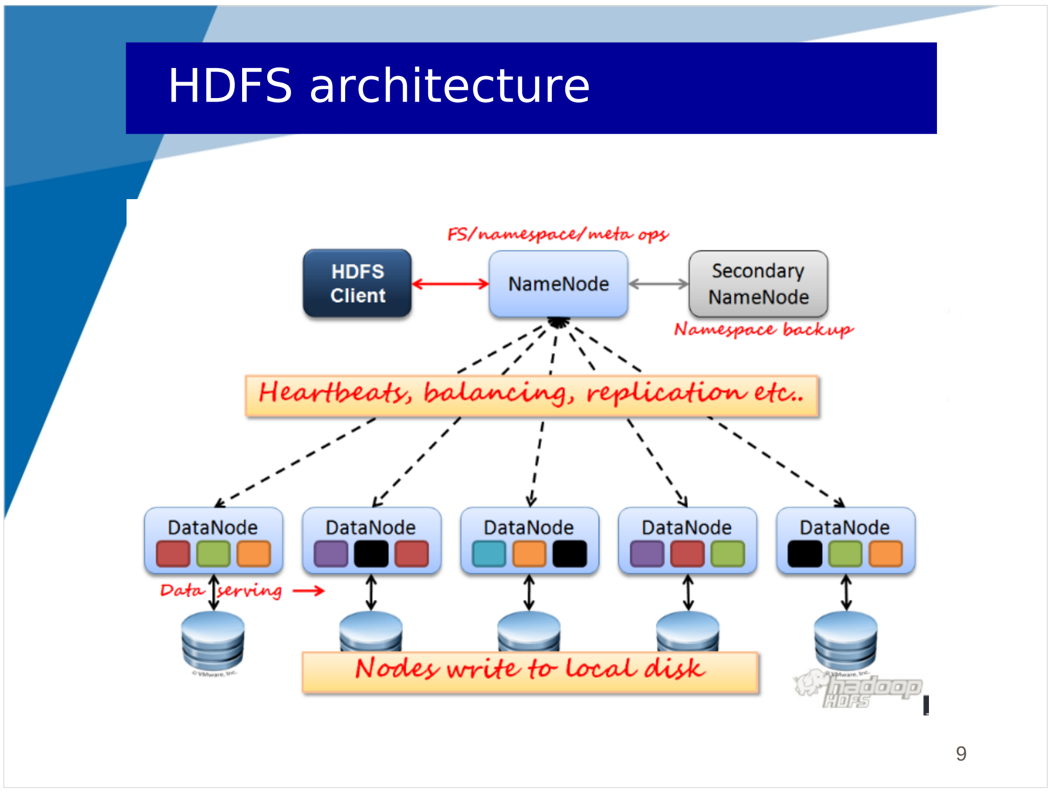
When you copy a large file to HDFS, it is divided into blocks (fragments) distributed over the cluster' nodes.

Block size is significantly large, so that mappers execution time is also significant. Initially, the default block size in Hadoop was 64 MB, it's currently 128 MB.

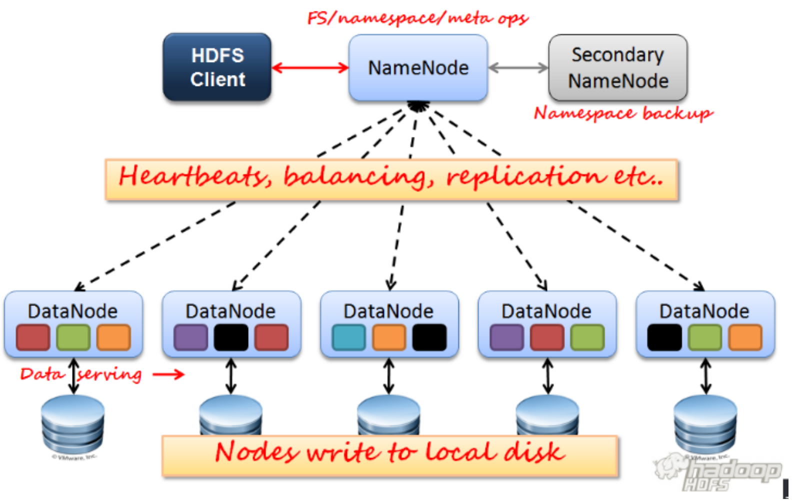
Each block is replicated (3 times by default) on several nodes, so that a node failure does not compromise the blocks availability.

Files in HDFS are read-only. The goal is to analyze data from these files and generates new data, not to modify the initial data. We say that HDFS is write-once-read-many.

HDFS relies on the local file system to store data on machines.



HDFS architecture



9

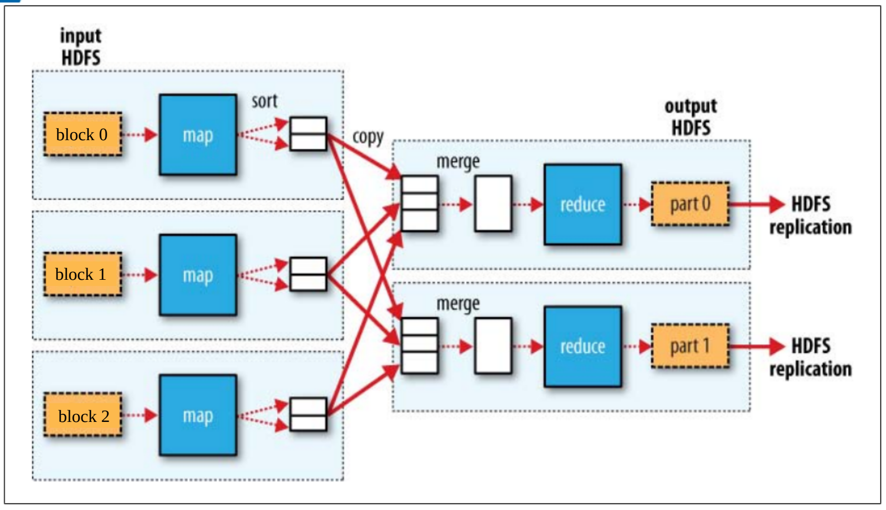
This figure illustrates the architecture of HDFS.

At the bottom, DataNodes are daemons that run on each node of the cluster. A DataNode allows reading and writing blocks on the local machine.

A global daemon called NameNode allows writing/reading files to/from HDFS. The NameNode is the entry point used by clients of HDFS. When a client invokes (write) the copy of a file to HDFS, the file is split into several blocks which are copied to DataNodes. Each block is replicated in 3 different DataNodes. The NameNode registers for each file its pathname in HDFS (in a logical hierarchy) and the location of its blocks in the cluster (the DataNodes where the blocks are stored).When a client invokes (read) the copy of a file from HDFS, the NameNode knows the location of the blocks which compose the file. It can then read the blocks from the DataNodes and reconstruct the file to be returned to the client.

For fault tolerance, a SecondaryNameNode is launched and can replace the NameNode in case of failure.

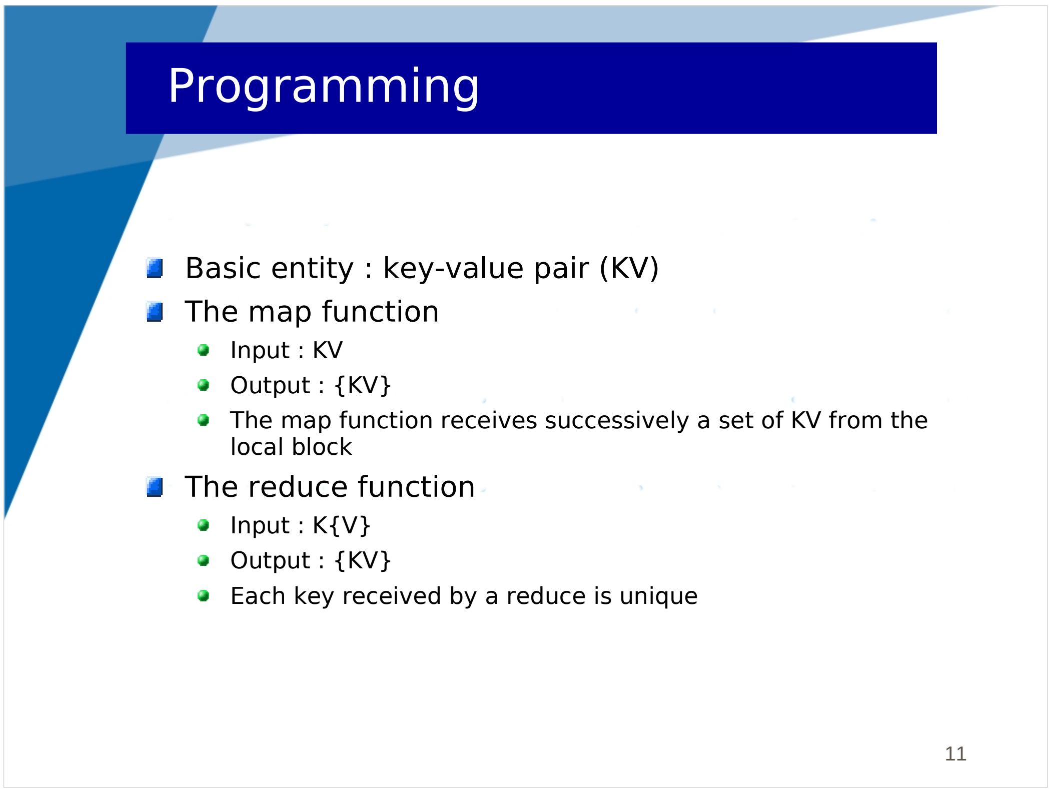
Execution scheme



10

Here is a more precise illustration of the execution scheme of a Hadoop application.

For each block of a file to handle, a mapper (map operation, we say a map) is executed on one node where the block is located. This map generates results (actually pairs, remember <city, price> in the previous example). These pairs generated by each map are sorted and sent to the reducers, all pairs with the same key (the first field of the pair) going to the same reducer. Each reducer (we say a reduce) aggregates all the pairs it receives and generates a fragment (part) of the final result. Each fragment of the final result is a block in HDFS.



Programming

- Basic entity : key-value pair (KV)
- The map function
  - Input : KV
  - Output : {KV}
  - The map function receives successively a set of KV from the local block
- The reduce function
  - Input : K{V}
  - Output : {KV}
  - Each key received by a reduce is unique

11

An application which processes a large dataset with Hadoop has to be programmed following the Hadoop programming model.

In Hadoop, every handled data is a key-value pair (KV).

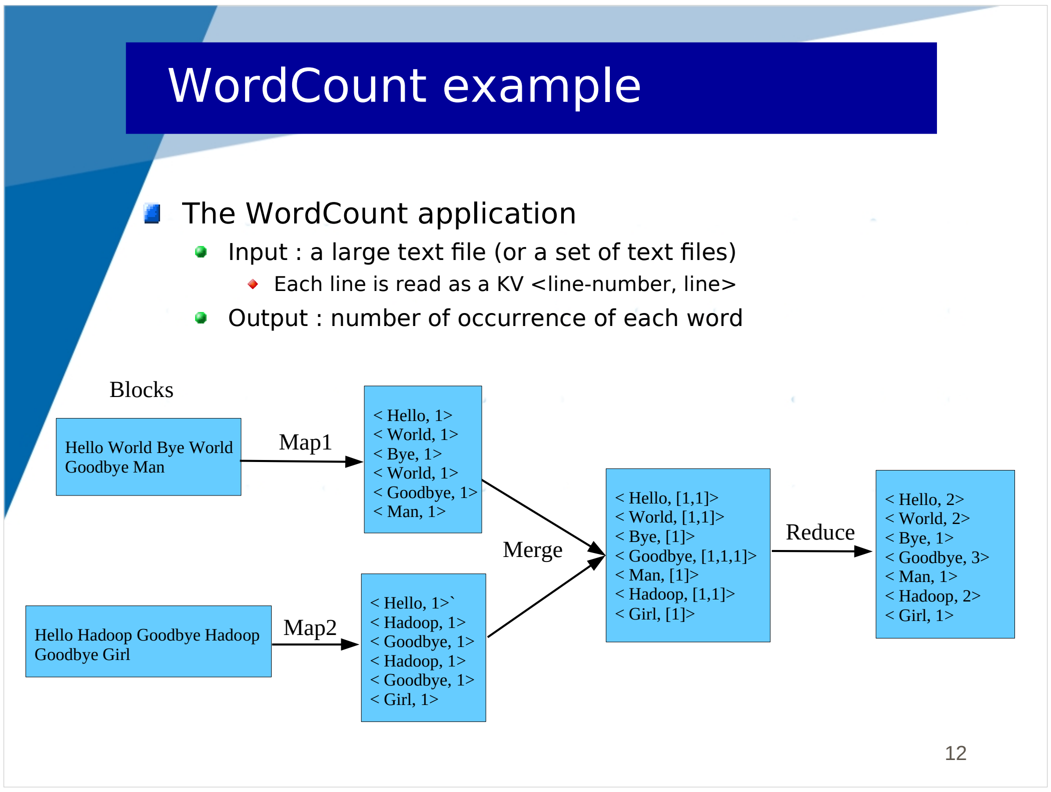
A map reads a block from HDFS locally. The block is supposed to include KV. Either the file is a KV file and then the map reads these KV, or the file is a text file and then the map reads lines returned as KV like <line-number, line>

The map executes a map() function (programmed by the developer) for each KV it reads. Therefore, the map() function receive one KV and it may generate any number of KV.

A reduce should receive a set of KV, but remember that for one key K, all the KV are going to the same reduce. Actually, the Hadoop system aggregates all the KV with the same key K into a unique pair <K,{V}>.

For each different K that a reduce receives, it invokes a reduce() function (programmed by the developer). This function receives <K,{V}> and may generate any number of KV.

(K1,V1)(K2,V2) (K1, V3) (K2, V4) >>> (K1, {V1, V3}) (K2, {V2,V4})



WordCount example

- The WordCount application
  - Input : a large text file (or a set of text files)
    - Each line is read as a KV <line-number, line>
  - Output : number of occurrence of each word

12

Let's see an example. This example is the most popular. It is used in any Big Data tools as a demonstrator.

This example is the WordCount application. The goal is to count the number of occurrence of each word in a text.

For instance, if the document to treat is

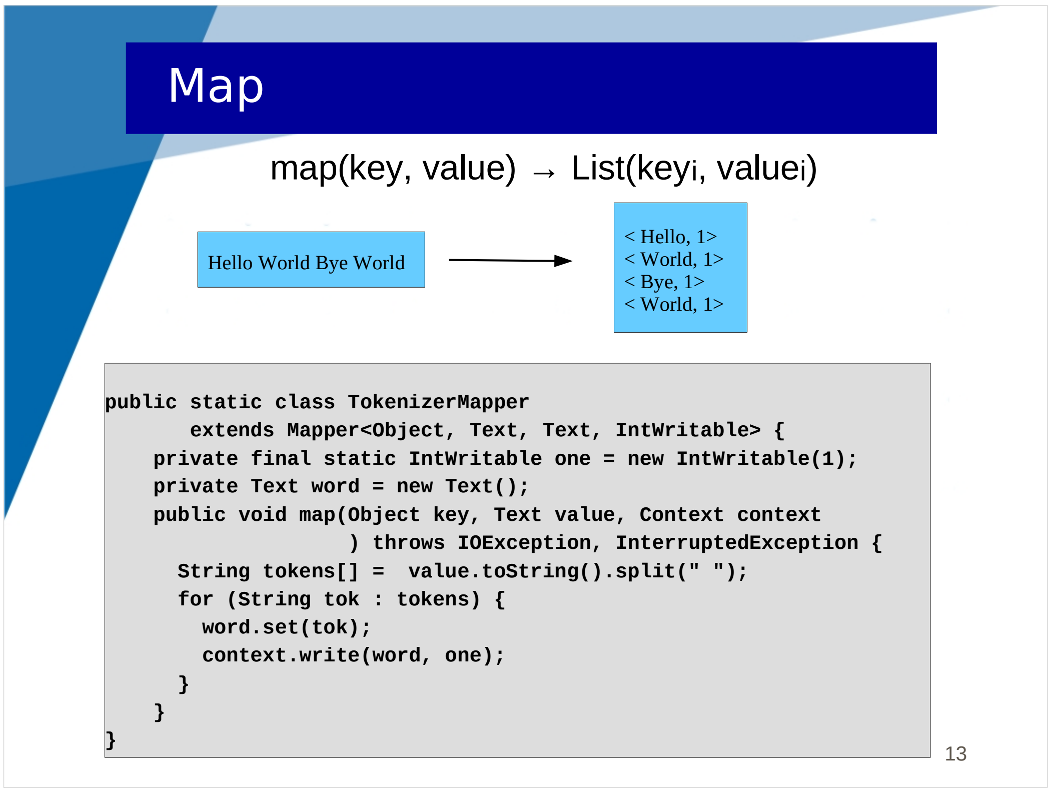
Hello World Bye World Goodbye

The final result is

< Hello, 1>< World, 2>< Bye, 1>< Goodbye, 1>

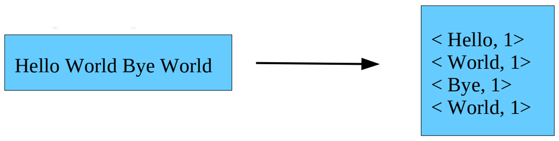
Each block is read line by line and the map() function is called for each line. The map() function splits the line into words (using the blank separator) and generates a KV <word, 1> for each word. This KV indicates 1 occurrence of that word.

In this figure, we assume we have only one reduce, so all the generated KV go to the unique reduce. The reduce invokes the reduce() function for each different K. This reduce() function counts the number of 1 behind a K.



Map

map(key, value) → List(key_i, $value_i$)



```
public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      String tokens[] =  value.toString().split(" ");
      for (String tok : tokens) {
        word.set(tok);
        context.write(word, one);
      }
    }
}
```

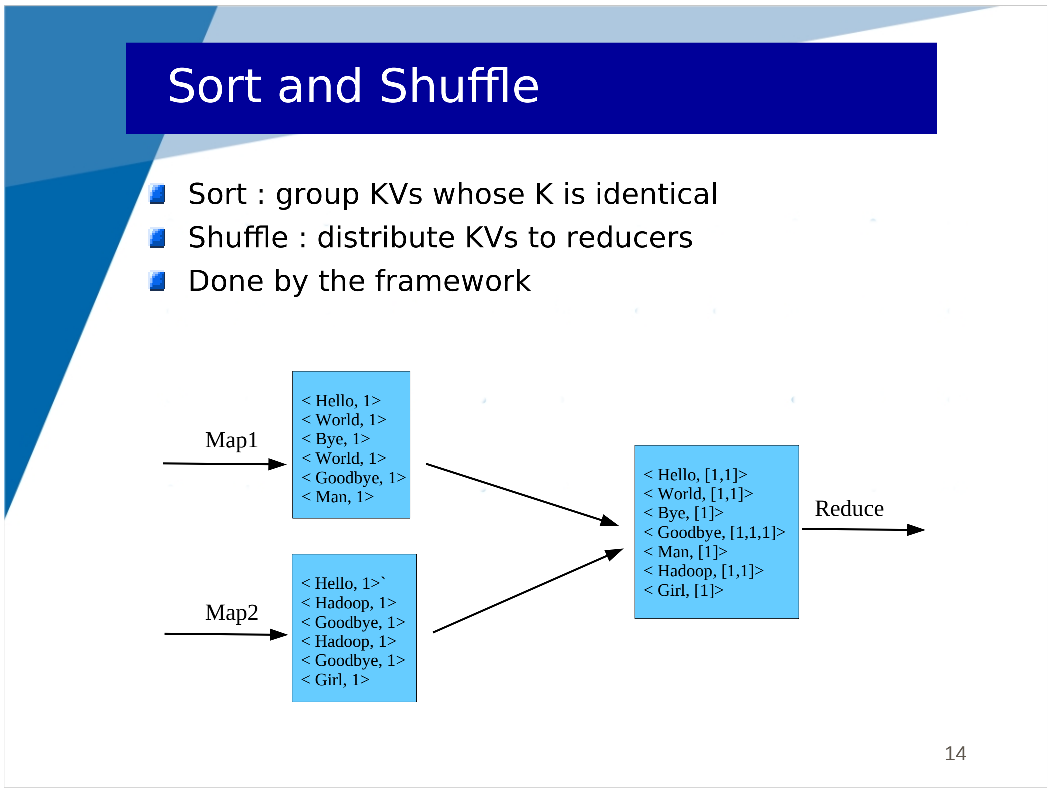
13

This class implements the behavior of the mapper.

The map() method receives as parameter a KV (parameters key and value of the method). In the WordCount example, this KV is <line-number, line>.

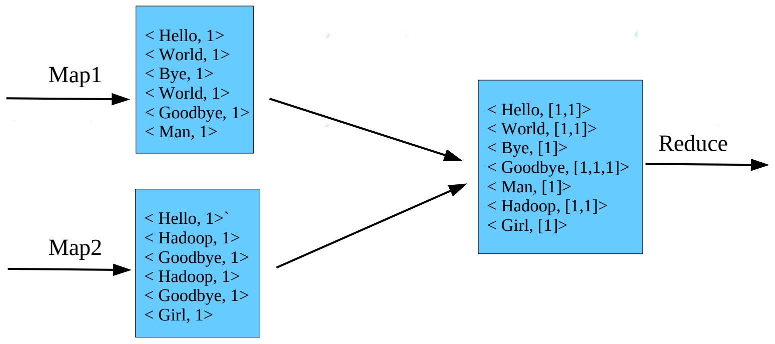
The last parameter of the map() method is (context), a reference to an object allowing to generate KV.

The map() method splits (with a Tokenizer object) the line into words (using the blank separator) and generates a KV <word, 1> for each word.



Sort and Shuffle

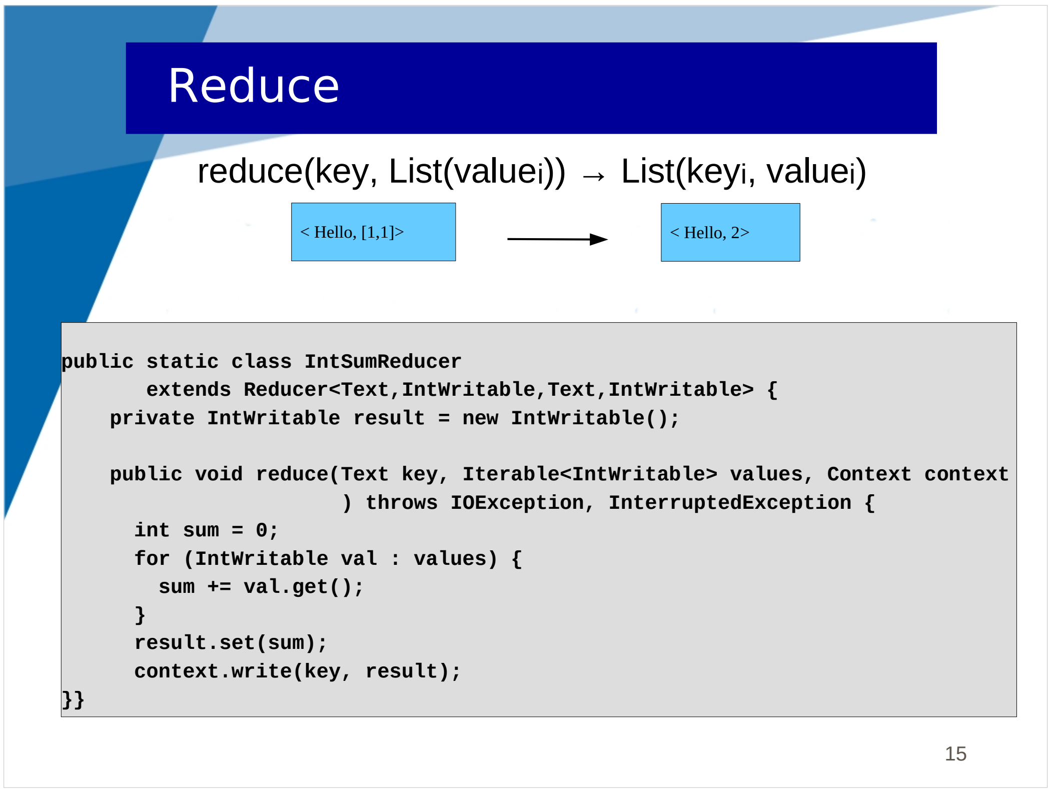
- Sort : group KVs whose K is identical
- Shuffle : distribute KVs to reducers
- Done by the framework



14

All the generated KV are sorted and grouped depending on K (all the KV with the same K are fusionned, giving a <K,{V}> pair), and sent to the reducers.

This is done by the Hadoop framework. It is possible to specialize this mechanism to control the distribution of keys between reducers if we have several reducers.



Reduce

reduce(key, List(value$_i$)) → List(key$_i$, value$_i$)

```
public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
}}
```

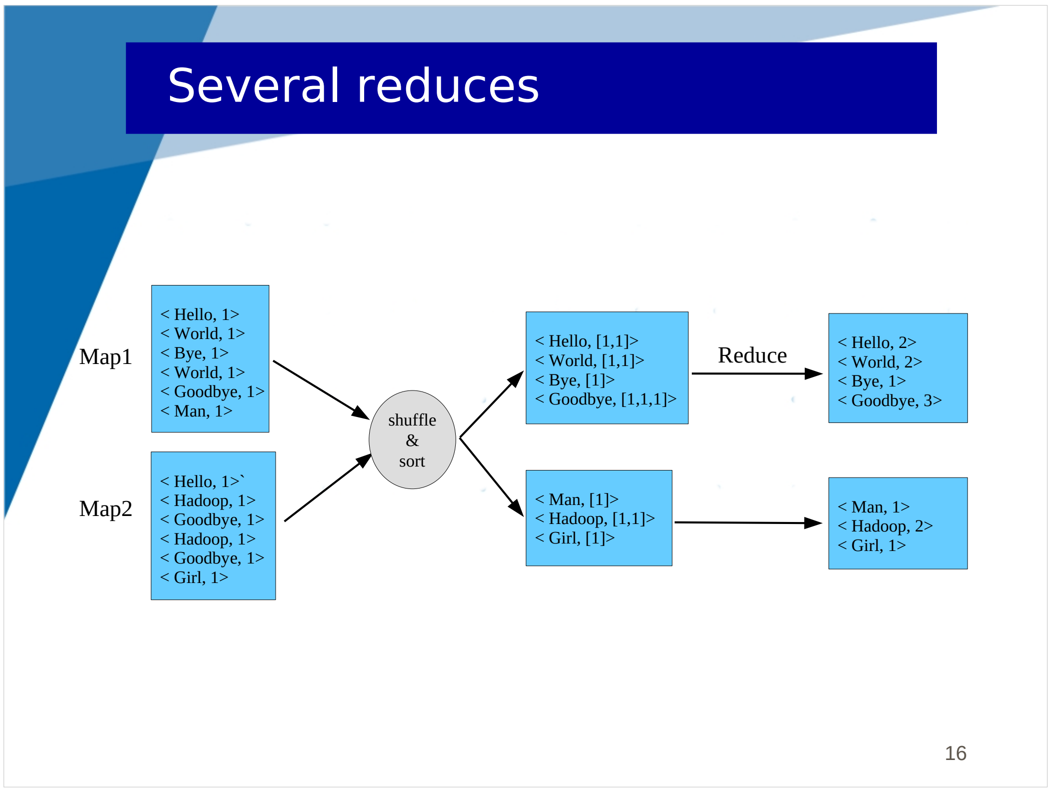
15

This class implements the behavior of the reducer.

The reduce() method receives as parameter a K{V} (parameters key and values of the method). In the WordCount example, this K{V} is <word, {1}>.

The last parameter of the reduce() method is (context) a reference to an object allowing to generate KV.

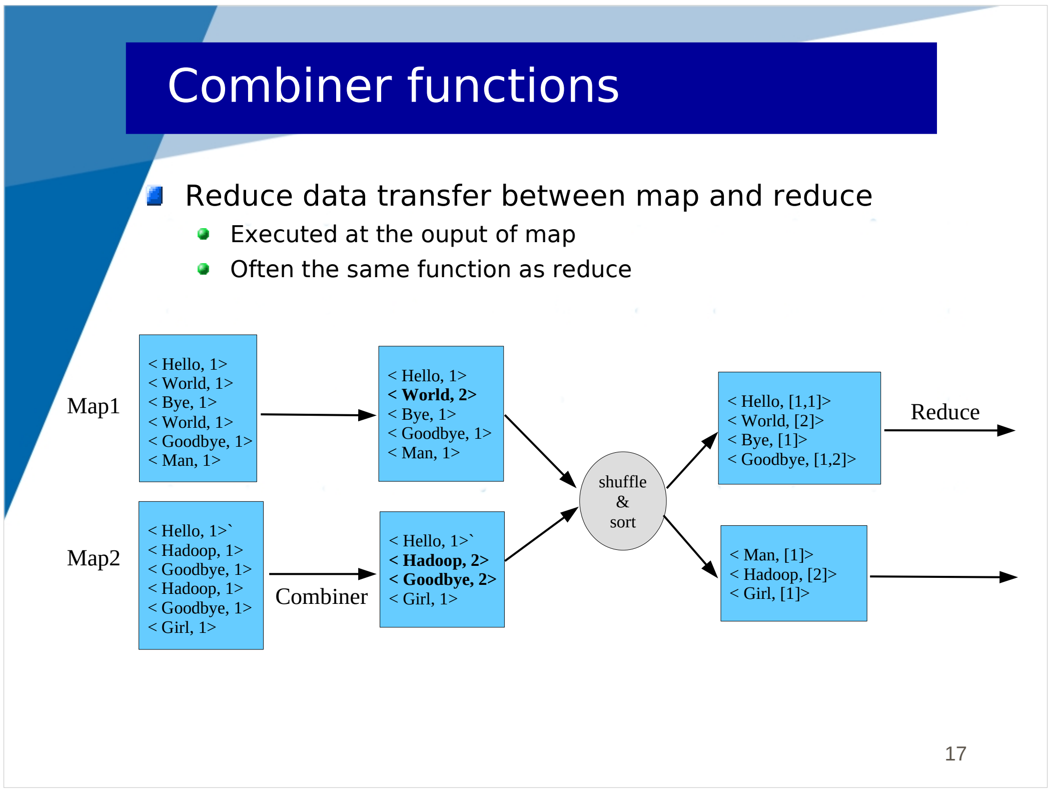
The reduce() method aggregates the values (makes the sum) and generates a KV <word, sum> for the word.



Several reduces

In this figure, we consider several reducers.

The K (words) are distributed between the 2 reduces. For instance, the pairs associated with the Hello word (generated by Map1 and Map2) are both sent to the first reducer. For this word, the reducer receives a pair <Hello, [1,1]> and aggregates the values, generating a pair <Hello, 2>.



Combiner functions

- Reduce data transfer between map and reduce
  - Executed at the ouput of map
  - Often the same function as reduce

17

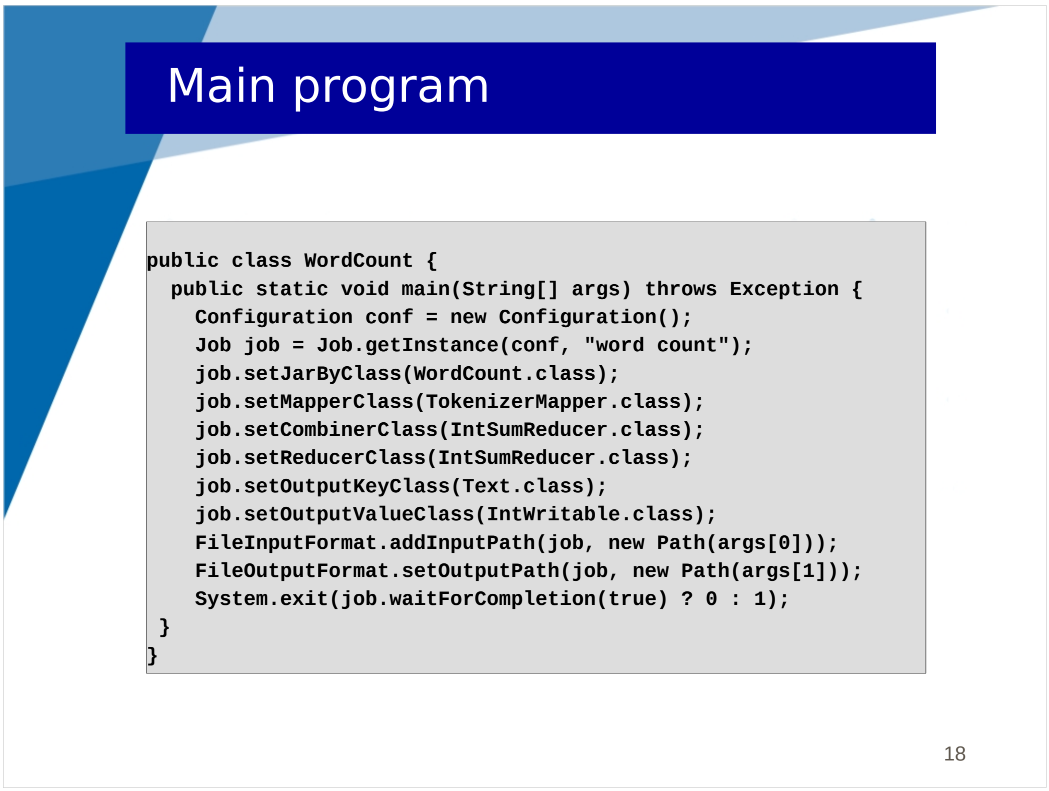
In some applications, a map may generate a lot of pairs which can be aggregated at the exit of the map, before going through the network. This is a way to decrease the network traffic and therefore to optimize performance.

This is possible in Hadoop thanks to a combiner function which is executed at the exit of each map. The KV are sorted and grouped at the exit of the map, generating a set of <K,{V}> which are handled by the combiner function in the same way as the reduce function, except that this is done on the same node as the map (while the reducer is on a different node).

In the figure, the combiner function is able to aggregate pairs with identical words at the exit of Map1 and Map2.

The interface of a combiner function is the same as the interface of a Reduce function.

Notice that in the WordCount application, we can use the implementation of the reduce function as a combiner function. This is not the case for all applications.



Main program

```
public class WordCount {
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

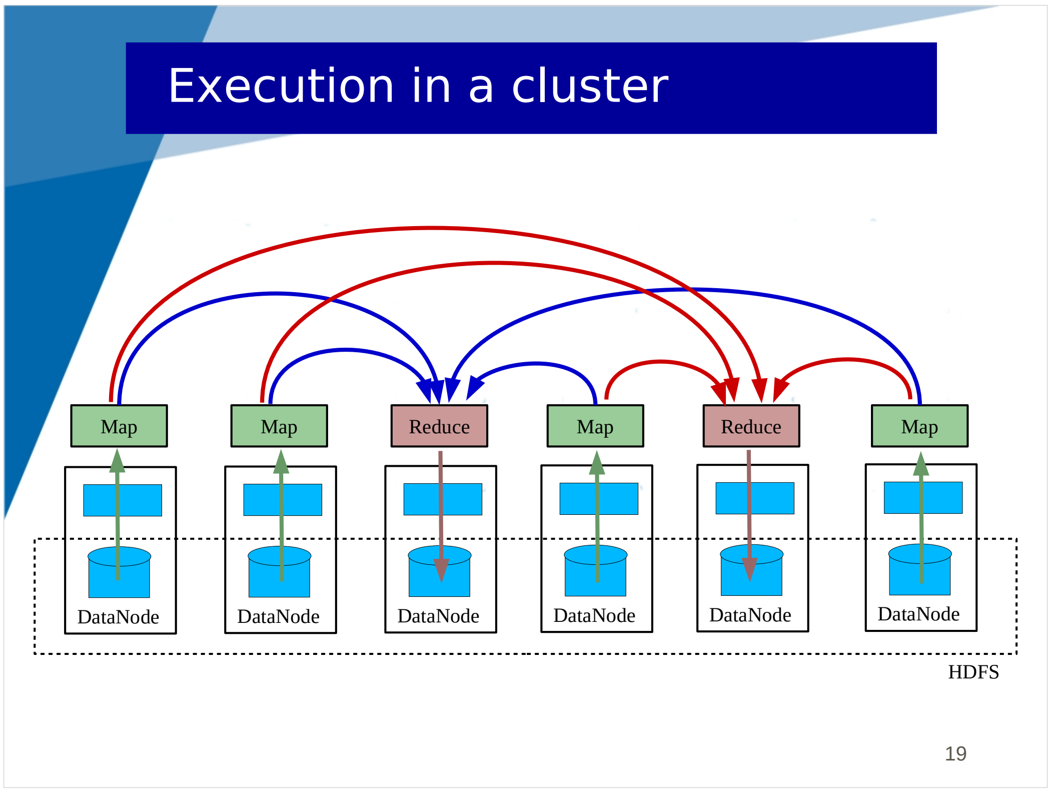
18

Here is the main class of the WordCount application.

It creates a new job which is initialized with :

- the main class
- the mapper class
- the combiner class (the same as the reducer class)
- the reducer class
- the classes for the output (for K and V)
- the path where input data will be found
- the path where output data will be stored

These paths refer to files on the local file system if we run Hadoop standalone (i.e. locally to test a program), or to files in HDFS if we run Hadoop in cluster mode.



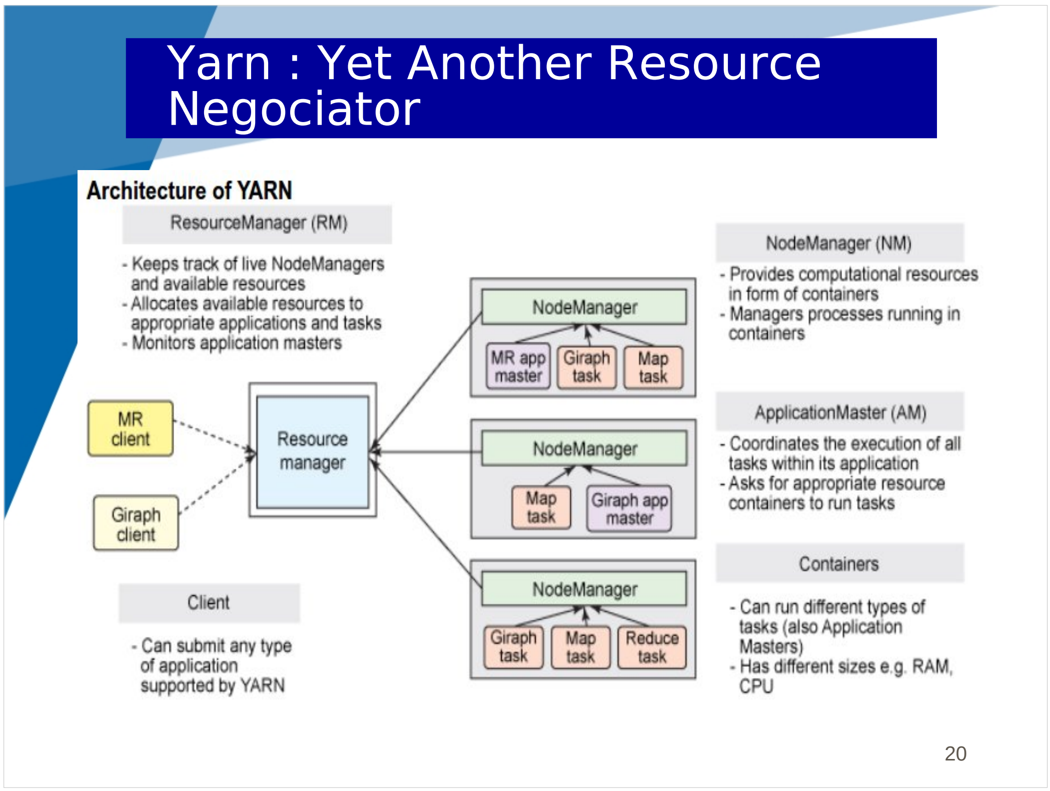
Execution in a cluster

19

This figure illustrates the execution in a cluster.

HDFS run on all the node with a DataNode daemon.

When running an application, several maps are executed on nodes where blocks are stored. Maps are generating KV which are transmitted to one reduce according to K (blue or red). Each reduce generates a result block which is stored locally in HDFS.



Yarn : Yet Another Resource Negociator

Architecture of YARN

ResourceManager (RM)

- Keeps track of live NodeManagers and available resources
- Allocates available resources to appropriate applications and tasks
- Monitors application masters

NodeManager (NM)

- Provides computational resources in form of containers
- Managers processes running in containers

ApplicationMaster (AM)

- Coordinates the execution of all tasks within its application
- Asks for appropriate resource containers to run tasks

Containers

- Can run different types of tasks (also Application Masters)
- Has different sizes e.g. RAM, CPU

Client

- Can submit any type of application supported by YARN

YARN is a resource manager used to deploy Hadoop applications, but also other types of application, in a cluster.

Yarn is composed of :

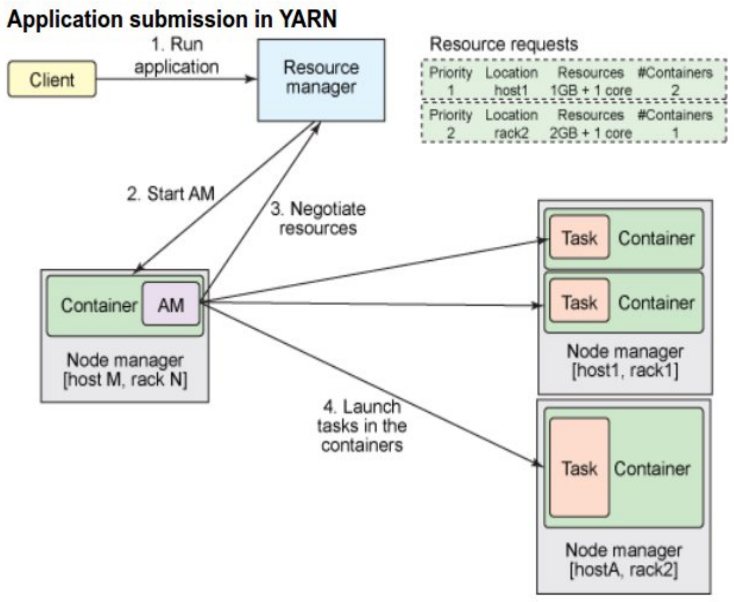
- a ResourceManager daemon. It's the front-end of Yarn. It is run on one node of the cluster and receives job submissions.

- NodeManager daemons. A NodeManager is run on each node of the cluster. It manages containers (nothing to do with Docker) for the execution of tasks (map or reduce or others). It keeps track of running tasks and available resources (CPU, memory).

The ResourceManager interacts with NodeManagers in order to keep track of available resources globally and to deploy tasks on NodeManagers.

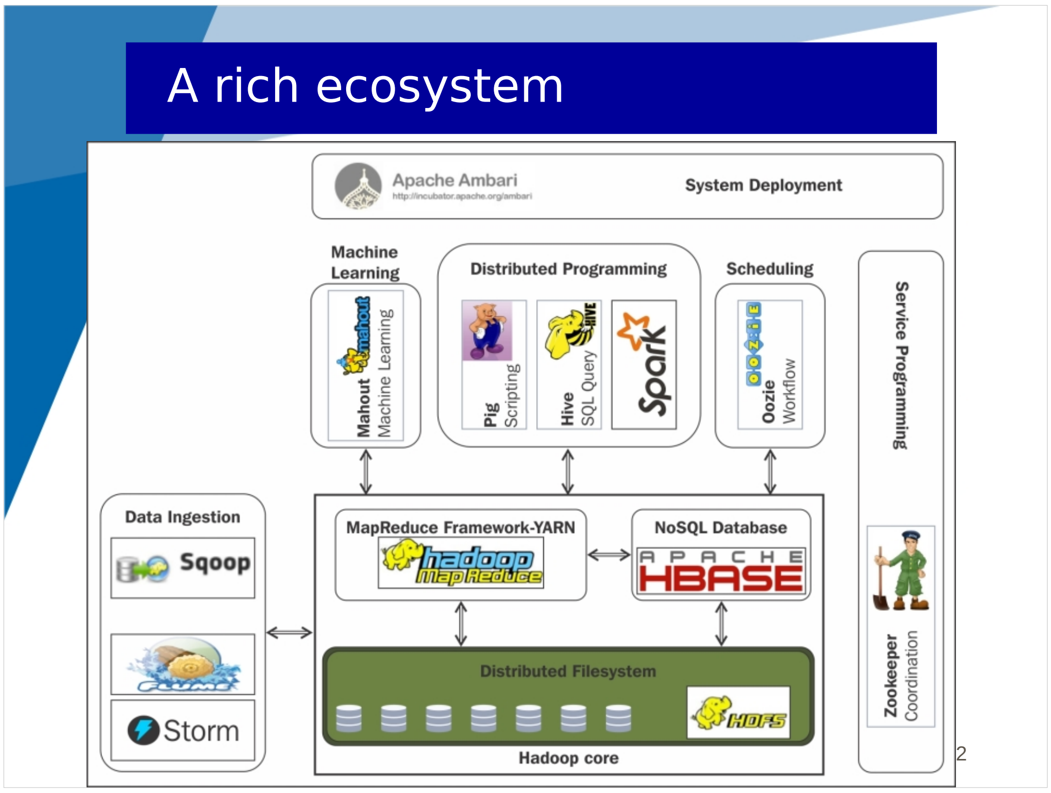
When an application is launched, an ApplicationManager is created, which coordinates the execution of the different tasks within the application.

Yarn : Yet Another Resource Negociator


Application submission in YARN
Client
1. Run application
Resource manager
Resource requests
Priority Location Resources #Containers
1 host1 1GB + 1 core 2
Priority Location Resources #Containers
2 rack2 2GB + 1 core 1
2. Start AM
3. Negotiate resources
Container
AM
Node manager [host M, rack N]
Task
Container
Task
Container
Node manager [host1, rack1]
4. Launch tasks in the containers
Task
Container
Node manager [hostA, rack2]

21

When a Hadoop map-reduce application is launched (at the ResourceManager), an ApplicationManager is created and launched in a container. This ApplicationManager knows that it has to create and launch a given number of map and reduce tasks. It asks the ResourcesManager for new containers for these tasks.



A rich ecosystem

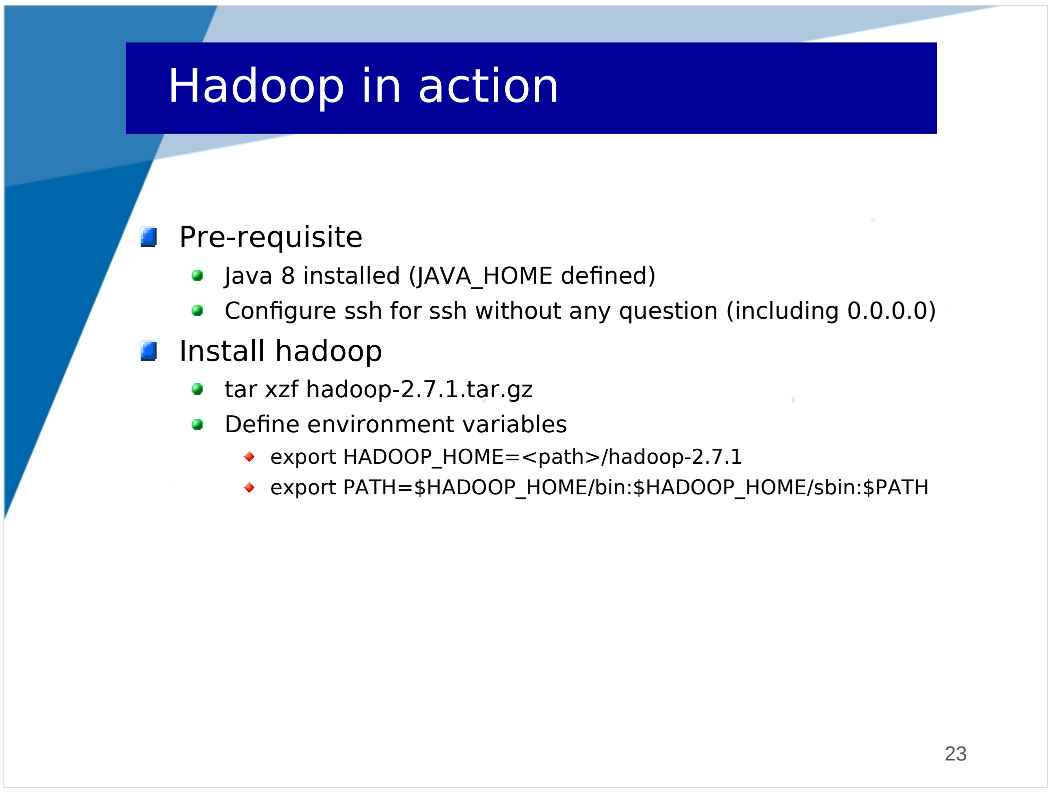
Hadoop is only a part of a rich ecosystem.

Hadoop is composed of HDFS (figure - bottom) and Hadoop and Yarn (figure – center).

Spark (figure – top) proposes another engine and programming model for big data applications.

Storm (figure- left) introduces a framework for processing streams of data in real-time.

The 2 previous services (Spark and Storm) will be presented in next lectures.



Hadoop in action

- Pre-requisite
  - Java 8 installed (JAVA_HOME defined)
  - Configure ssh for ssh without any question (including 0.0.0.0)
- Install hadoop
  - tar xzf hadoop-2.7.1.tar.gz
  - Define environment variables
    - export HADOOP_HOME=<path>/hadoop-2.7.1
    - export PATH=$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$PATH

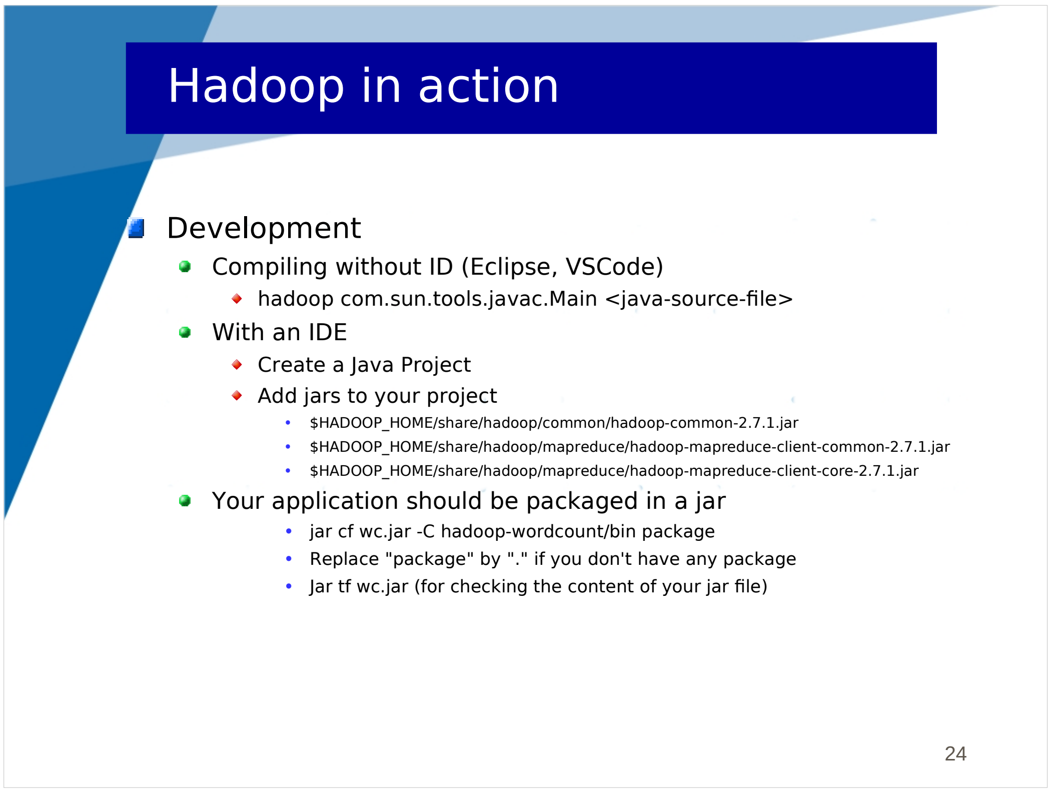
23

The following slides describe the main instructions for using Hadoop.

You should have Java installed and you should configure ssh to be able to log on any node of the cluster without any question (known_hosts, password).

Installing Hadoop is simply expanding an archive. Notice that in a cluster, the binaries should be accessible at the same path on any node. This is obvious with proper NFS mounts.

You have to define environment variables (store that in your bashrc).



Hadoop in action

- Development
  - Compiling without ID (Eclipse, VSCode)
    - hadoop com.sun.tools.javac.Main <java-source-file>
  - With an IDE
    - Create a Java Project
    - Add jars to your project
      - $HADOOP_HOME/share/hadoop/common/hadoop-common-2.7.1.jar
      - $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-common-2.7.1.jar
      - $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.7.1.jar
  - Your application should be packaged in a jar
      - jar cf wc.jar -C hadoop-wordcount/bin package
      - Replace "package" by "." if you don't have any package
      - Jar tf wc.jar (for checking the content of your jar file)

24

As usually, I prefer using an IDE for edition of code only and not for running programs (I run programs with shell commands only).

The first command is a shortcut for compiling without IDE. You can also simply compile with Java if you provide the necessary jars that are given below.

If you use Eclipse or VSCode, you just need to create a Java project and add the given jars to your project.

Assuming that your project is called hadoop-wordcount, you can create an archive (jar) of the compiled application with the command "jar cf".

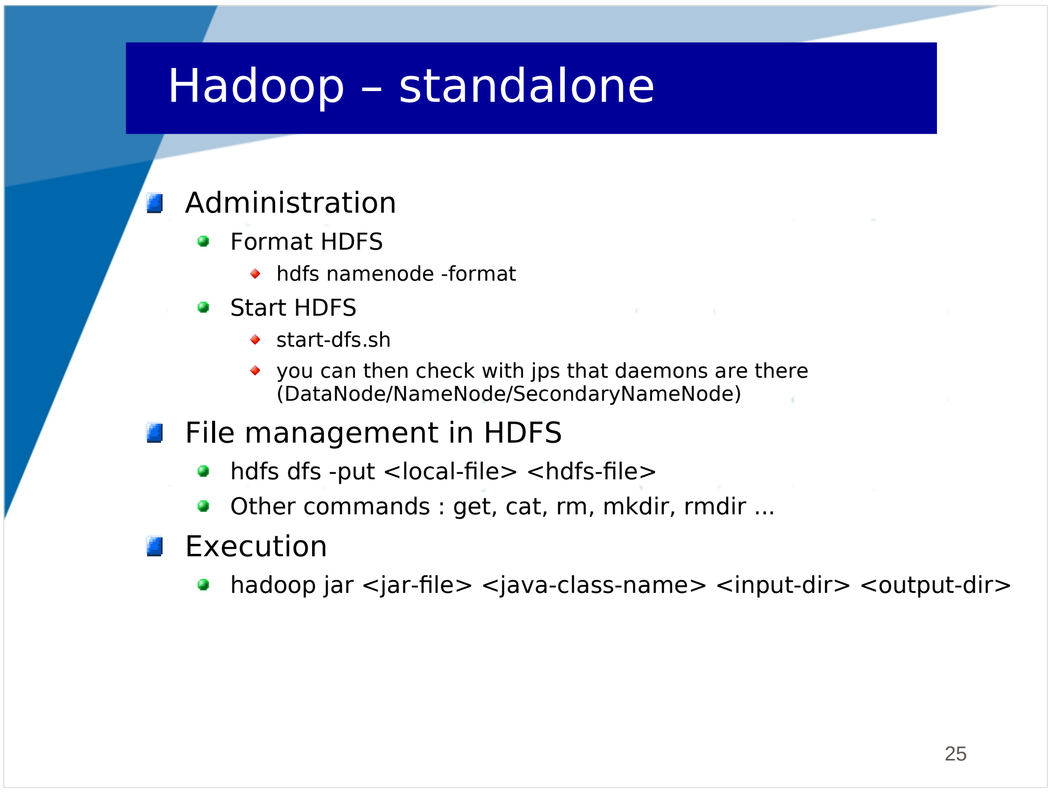
"-C" means changing to that directory

"package" is the name of the package embedding your classes

This archive is what you give to Hadoop to launch an application.

Replace "package" by "." if you don't have any package in your project.

It's interesting to check the content of the archive with "jar tf".



Hadoop – standalone

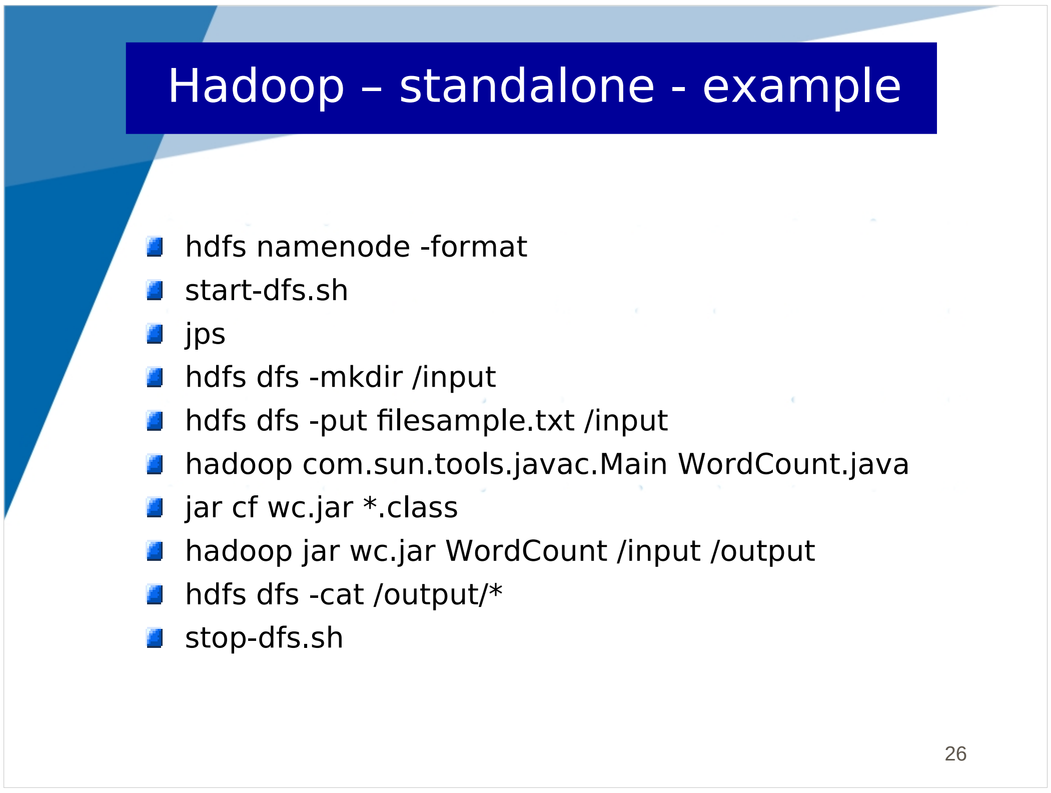
- Administration
  - Format HDFS
    - hdfs namenode -format
  - Start HDFS
    - start-dfs.sh
    - you can then check with jps that daemons are there (DataNode/NameNode/SecondaryNameNode)
- File management in HDFS
  - hdfs dfs -put <local-file> <hdfs-file>
  - Other commands : get, cat, rm, mkdir, rmdir ...
- Execution
  - hadoop jar <jar-file> <java-class-name> <input-dir> <output-dir>

25

Here are the instructions for testing Hadoop locally (local machine).

You have instructions for :

- formatting HDFS
- starting HDFS
- managing files in HDFS
- running an hadoop application



Hadoop – standalone - example

- hdfs namenode -format
- start-dfs.sh
- jps
- hdfs dfs -mkdir /input
- hdfs dfs -put filesample.txt /input
- hadoop com.sun.tools.javac.Main WordCount.java
- jar cf wc.jar *.class
- hadoop jar wc.jar WordCount /input /output
- hdfs dfs -cat /output/*
- stop-dfs.sh

26

This is a complete example where :

- we start HDFS

- create a /input directory in HDFS

- copy filesample.txt in this directory of HDFS

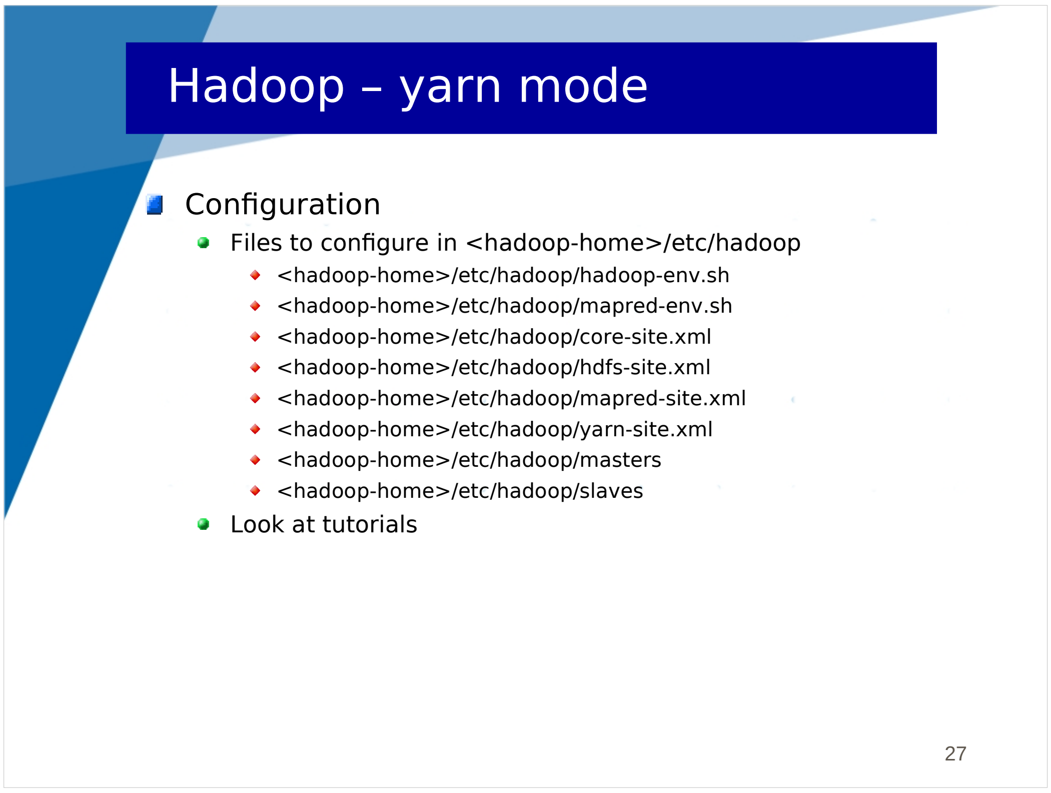
- compile WordCount.java

- create the archive of the compiled application

- launch the application. Data to be processed are taken in the /input directory in HDFS. The results are to be stored in a /output directory that should not exist and will be created.

- show everything (the results) that was stored in /output

- stop HDFS



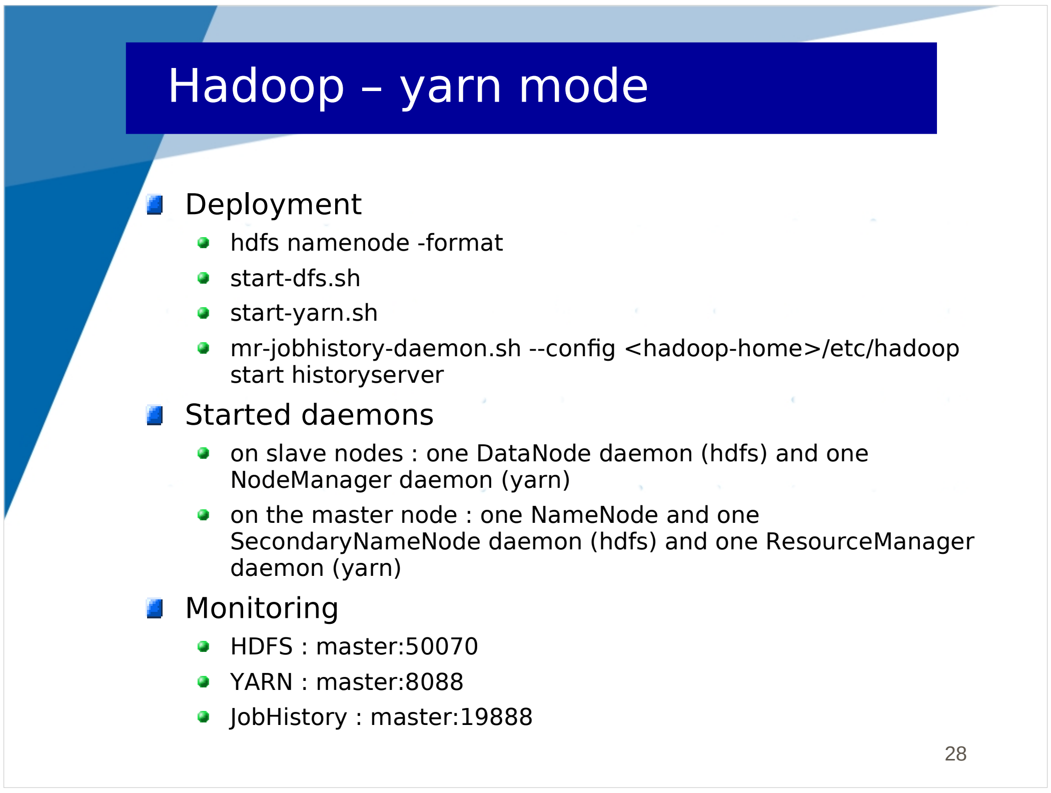
Hadoop – yarn mode

- Configuration
  - Files to configure in <hadoop-home>/etc/hadoop
    - <hadoop-home>/etc/hadoop/hadoop-env.sh
    - <hadoop-home>/etc/hadoop/mapred-env.sh
    - <hadoop-home>/etc/hadoop/core-site.xml
    - <hadoop-home>/etc/hadoop/hdfs-site.xml
    - <hadoop-home>/etc/hadoop/mapred-site.xml
    - <hadoop-home>/etc/hadoop/yarn-site.xml
    - <hadoop-home>/etc/hadoop/masters
    - <hadoop-home>/etc/hadoop/slaves
  - Look at tutorials

27

To execute Hadoop in cluster mode, you have to configure a set of files (not much things to edit in each file).

I don't detail these configuration file here.



Hadoop – yarn mode

- Deployment
  - hdfs namenode -format
  - start-dfs.sh
  - start-yarn.sh
  - mr-jobhistory-daemon.sh --config <hadoop-home>/etc/hadoop start historyserver
- Started daemons
  - on slave nodes : one DataNode daemon (hdfs) and one NodeManager daemon (yarn)
  - on the master node : one NameNode and one SecondaryNameNode daemon (hdfs) and one ResourceManager daemon (yarn)
- Monitoring
  - HDFS : master:50070
  - YARN : master:8088
  - JobHistory : master:19888

28

In the cluster mode, the deployment is about the same, except that you have to run a start-yarn.sh command which starts the ResourceManager and NodeManagers.

Optionally, you can start a jobhistory server which logs all the jobs that were run in the cluster.

Once deployed, you should observe the following daemons :

- the master node is the node which run the unique daemons : NameNode, SecondaryNameNode and ResourceManager

- the slave nodes are the nodes used for parallel computations. They each run a DataNode and a NodeManager.

Each manager provides a web console for observing what's happening.

You can observed

- block distribution in HDFS at http://<masternode>:9870

- job executions at http://<masternode>:8088

- job history at http://<masternode>:19888