CLOUD COMPUTING

NOTES FROM THE COURSE SECTION TAUGHT BY

PROF. **PULIAFITO**

A.Y. 2022-23

Luca Arduini

last update: 2024-03-25

Disclaimer

These notes were taken during lectures by professor Puliafito for the "Cloud Computing" course in the Master's Degree program in "Artificial Intelligence and Data Engineering" at the University of Pisa during the academic year 2022-2023. The notes have not been reviewed by any instructor and do not constitute official course material.

Please use these notes as a supplementary resource for your studies and avoid relying solely on them. All images included are copyrighted and have been obtained from the professors' slides.

I hope you find these notes helpful, and I wish you the best of luck in your studies!

WARNING!

These notes are **incomplete**! Specifically, the last section does not include all the topics from the 'Design Pattern' slide deck, as I studied it directly from the slides. I advise those of you reading these notes to do the same, as that section is also part of the exam syllabus and may be asked.

Special thanks

I would like to express my gratitude to my colleague Davide Bruni for sharing his course notes with me, which I have then expanded upon to create this final version of the notes.

Further insights into my notes

Did you find any errors or paragraphs that could have been written more clearly while reading these notes? Feel free to contact me on Telegram, and I'll be happy to hear your suggestions.

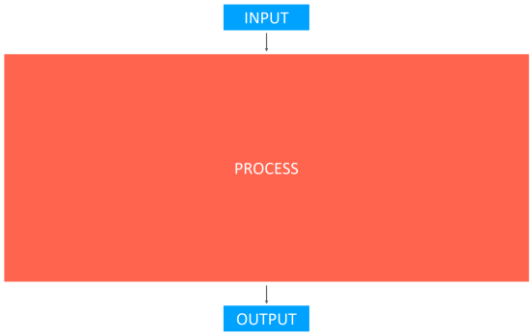
Have you found these notes helpful? Visit my GitHub page for additional notes and explore all the projects I've completed during my academic journey.

Summary

PULIAFITO'S THEORETICAL LECTURE SCHEDULE	
1) 2023-03-27	Big Data
2) 2023-04-18	MapReduce & Hadoop
3) 2023-04-24	HDFS & YARN
4) 2023-05-05	*hadoop installation*
5) 2023-05-08	Design Pattern part 1
6) 2023-05-12	*commands HDFS & java programming*
7) 2023-05-26	Design Pattern part 2

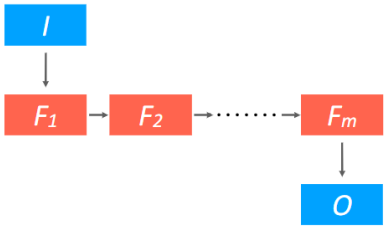
Parallel programming

In a typical application we have a process with data given in input and that returns an output.
Let us suppose that our process is too large to be executed as a monolithic:

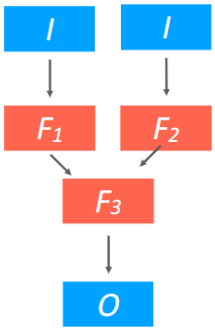


It can be performed by dividing the task in subtask executed:

- Sequentially: O = F(I)



- Parallel with a pipeline: Pipeline → O = F(I) = Fm(Fm-1(…(F1(I)..))

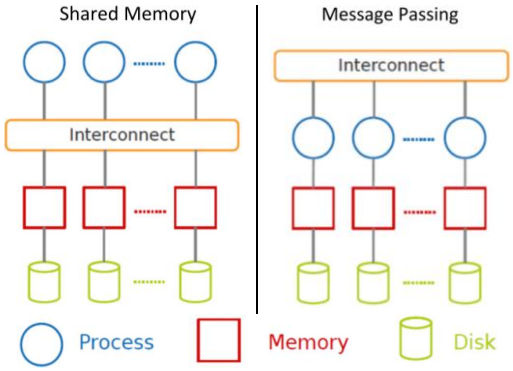


One of the biggest problem is: **how to split the code**? It leads to: How to divide code into parallel tasks? • How to distribute the code? • How to coordinate the execution? • How to load the data? • How to store the data? • What if more tasks than CPUs? • What if a CPU crashes? • What if a CPU is taking too long? • What if the CPUs are different?

Parallel algorithm design

Parallel Architectures:

- **Shared Memory** parallel programming model (Posix Threads, OpenMP, Compiler automatic optimizations, …)
- **Message Passing** programming model (Sockets, Parallel Virtual Machine (obsolete), Message Passing Interface (MPI))



Typical steps:

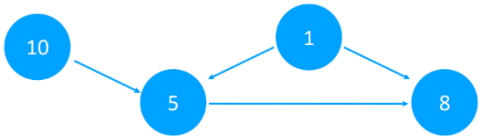
1. Identify what pieces of work can be performed concurrently
2. Partition concurrent work onto independent processors
3. Distribute a program's input, output, and intermediate data
4. Coordinate accesses to shared data: avoid conflicts
5. Ensure proper order of work using synchronization

Some steps can be omitted: for **shared memory** parallel programming model, there is no need to distributed data. Instead for **message passing** parallel programming model, there is no need to coordinate shared data.

We will study **message passing** model patterns. Each pattern has its own: specific structure of parallel tasks, specific techniques for partitioning / allocating data, specific structures of communication.

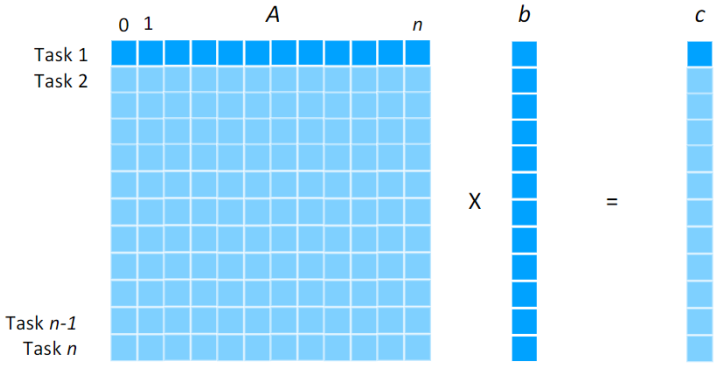
Task Dependency Graph (TDG)

When developing a parallel algorithm the first step is to decompose the problem into task that can be executed concurrently. A given problem may be decomposed into tasks in many different ways.
A **task** represents a unit of work or an activity that needs to be performed.



TDG is a Directed Acyclic Graph (DAG) in which nodes are tasks and directed edges are control dependencies among tasks. The node label is the computation weight of the task.
The Task Dependency Graph provides a clear and structured view of the dependency relationships among tasks and can be used to plan, coordinate, and optimize task execution. It can be particularly useful in complex scenarios where there are many interconnected activities that need to be performed in a specific order or in parallel with respect to specified dependencies.

Example



Example of program developed in concurrent way: **multiplication between a matrix (n x n) and a vector (n x 1)**

The figure highlights the portion of the matrix and vector accessed by task 1.
While tasks share data (the vector b), they do not have any dependencies: the computation of each element of the output vector c is independent of other rows in A. Based on this, a dense matrix-vector product can be decomposed into **n** tasks, where each task performs an equal number of operations.

Task properties

Granularity

The *number of tasks* into which a problem is decomposed determines its *granularity.*
Decomposition into a large number of tasks results in **fine-grained decomposition**, otherwise decomposition into a small number of tasks results in a **coarse grained decomposition**.

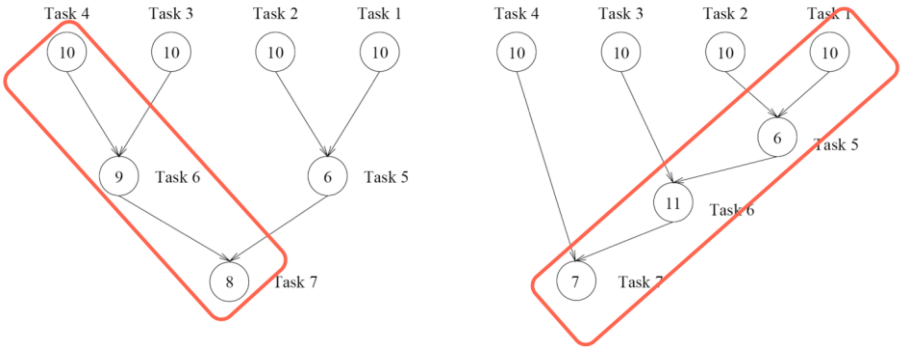
Degree of concurrency

The *number of tasks that can be executed in parallel* is the *degree of concurrency* of a decomposition.

Since the number of tasks that can be executed in parallel may change over program execution, the maximum degree of concurrency is the maximum number of such tasks at any time during execution. So we have to define the maximum degree of concurrency and average degree of composition.

If the average degree of concurrency is similar to the maximum, the parallel system is used very efficiently.
Generally, *the degree of concurrency increases as the decomposition becomes finer in granularity and vice versa.*

Critical path
Directed path in the Task Dependency Graph (TDG) represents a sequence of tasks that must be processed one after the other: the longest path determines the shortest time in which the program can be executed in parallel (it's called critical path).
The length of the path corresponds to the minimum execution time of a parallel program.
Used when all the task has same duration/weight: if not, used the sum of the weight of the tasks.



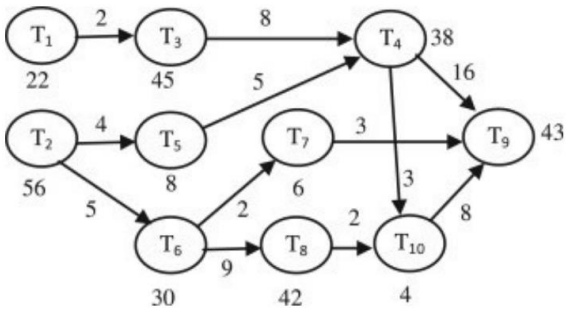
Parallel Performance Limitations

By increasing the number of tasks of the decomposition (going to fine-granularity decomposition) we can reduce the parallel time, but there are limitations depending on the problem.
Inherent bounds on granularity of the computation. i.e.: in matrix-vector multiplication it relies on the number of cells of the matrix (no more than n^2 tasks).
Increasing concurrency we also increase the amount of data that has to be exchanged with other tasks. It results in a communication overhead which increases along with the increase of the parallelism degree.

Task Interaction Graph (TIG)

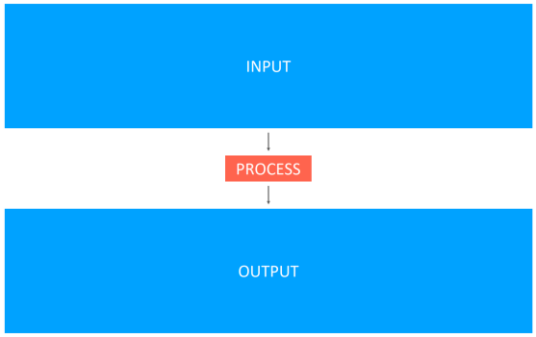
Additionally, there is another graph called Task Interaction Graph (TIG) that allows for visualizing the interactions (data exchange) that occur between tasks.

In this graph, the nodes represent tasks, and the directed edges represent data exchange. The node labels indicate the computational weight of a task, and the edge labels indicate the size of the exchanged data.

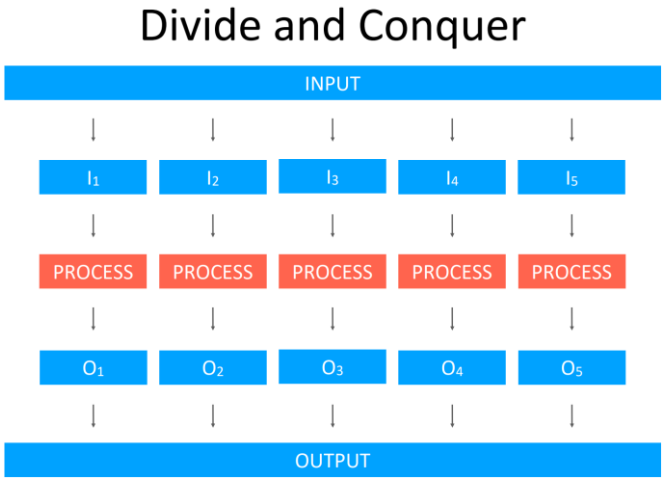


MapReduce

MapReduce is a *programming model*.
What happen if, in a different way respect to the previous chapter, our Input and Output are big?

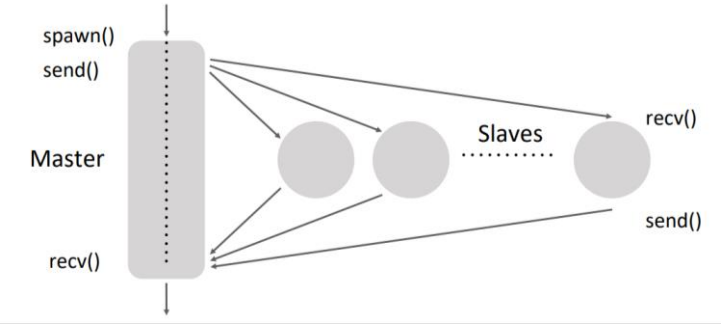


The approach is the following



Process mustn't be extremely simple, it could be easy or not.

So we take the input and split into "Input splits" and then provide each of them as input to the process.
An implementation could be ***Master-Salve implementation:***



Static partitioning of the input data. Each slave is assigned a partition of the input.
The master will reaggregate the final output.

Obviously, ***this kind of implementation doesn't work well if the assign task are unbalanced*** (80% to one slave and then 20% to other ones…)

Typical Big Data Application

1. Iterate over a large number of records
2. Extract something of interest from each
3. *Shuffle and sort* intermediate results
4. Aggregate intermediate results
5. Generate final output

Design ideas:

- Right level of abstraction:
  - Hide implementation details from applications development
  - Write very few lines of code
  - Drawback: the program, everything needs to fit into the abstraction
- Scale "out" (horizontal), not "up":
  - Avoid supercomputer, too costly
  - Use commodity machines, low costs
  - Drawback: many failures
- Move processing to the data:
  - Same code, runs everywhere
  - Reduce data over the network
  - Drawback: code must be portable

Functional Programming

Functional programming is a mathematical approach to solving problems, is **simpler** and **less abstract**, **easy** to **reuse**, to **test** and to handle **concurrency**. MapReduce is based on this type of programming.
Functional programming principles are:

1. **Purity**:
   Will always produce the same output, given the same input parameters (idempotency)
2. **Immutability**:
   a. There are no "variables" in functional programming
   b. All "variables" should be considered as constants
   c. If needed, create a new data structure with the updated state.
   d. Only in few cases, we mutate variables, e.g. short living "local" variables
   e. Advantages
      i. A process does not need to bother about other processes when dealing with a data structure, as nobody can modify it
      ii. History is not wiped out (not erased)
3. **Higher-order functions**:
   The software must function as a composition of a chain of many other functions, where each input is evaluated until the final function completes the evaluation.
   a. Higher-order functions are functions that operate on other functions either by:
      i. Taking them as arguments (e.g. callback function)
      ii. Returning them as output
   b. Higher-order functions:
      i. Make the *code easier to understand and debug*
      ii. Allow to perform operations on functions as you would do on other elements

MapReduce programming model

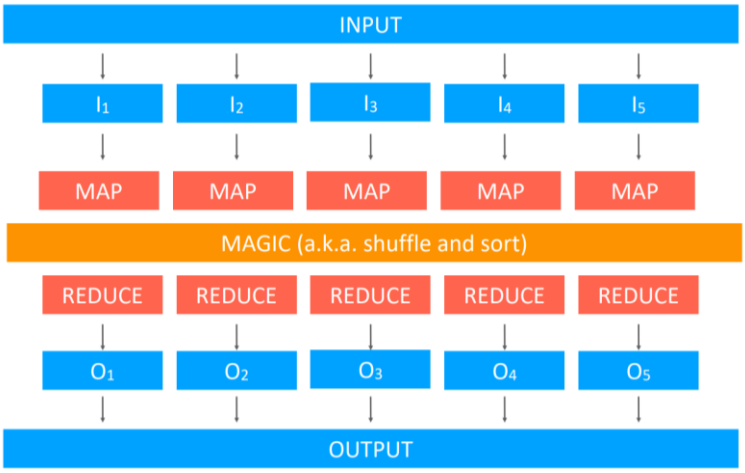
MapReduce is a programming paradigm that simplifies distributed data processing by breaking down complex tasks into smaller, parallelizable operations. It consists of two main phases: the map phase and the reduce phase.
In the **map phase**, input data is divided into smaller chunks and processed independently by multiple map tasks.
In the **reduce phase**, the intermediate results produced by the map tasks are combined, processed, and aggregated to produce the final output.

Especially programmers need to implement two functions:

- **map function**: ***from [key, value]*** *(1)* ***to list of [key, value]*** *(0 or more)*
  Receives as input a key-value pair. Produces as output a list of key-value pairs.
  Called by the Mapper, it's an higher-order function

- **reduce function**: ***from [key, list of values]*** *(1)* ***to list of [key, value]*** *(0 or more)*
  Receives as input a key-list of values pair. Produces as output a list of key-value pairs.
  Called by the Reducer, it's an higher-order function

Both functions are STATELESS.



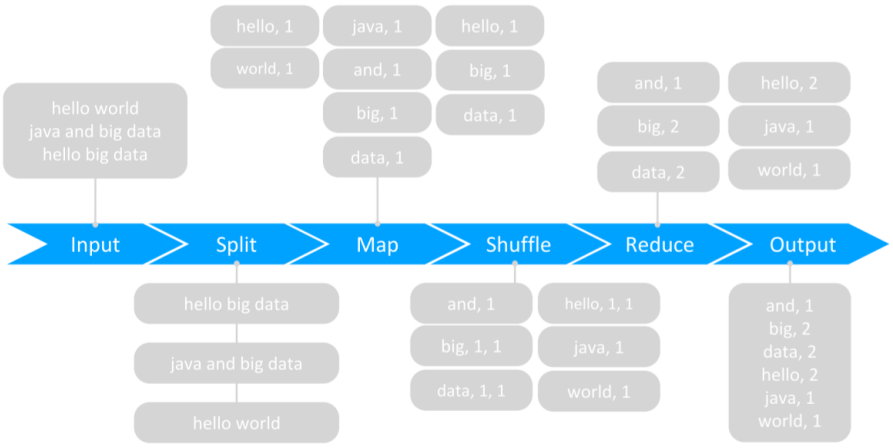
Mappers:

- Mappers ***should run on nodes which hold their (portion of the) data locally***, *to avoid network traffic*
- Multiple mappers run in parallel, each processing (a portion of) the input data
- The mapper reads in the form of [key, value] pairs. These are read from the distributed file system
- ***The mapper may use or completely ignore the input key***: for example, a standard pattern is to read one line of a file at a time: the **key** is the byte offset into the file at which the line starts, the **value** is the contents of the line itself. Typically the key is considered irrelevant.
- Mapper output (if any) must be in the form of [key, value] pairs.

Reducers*:*

- After the map phase is over, ***all intermediate values for a given intermediate key are combined together into a list***. ***Each of these lists is given to a reducer***
- **There may be a single reducer, or multiple reducers**. *All values associated with a particular intermediate key are guaranteed to go to the same reducer*.
  The intermediate keys, and their value lists, are passed to the reducer in sorted key order: *this step is known as the "**shuffle and sort**"*.
  The reducer outputs zero or more final [key, value] pairs: these are written to the distributed file system.
  In practice, the reducer usually emits a single key/value pair for each input key.

Wordcount Example




```
class MAPPER
method MAP(docid a, doc d)
for all term t in doc d do
EMIT(term t, count 1)

class REDUCER
method REDUCE(term t, counts [c1, c2,...])
sum ← 0
for all count c in counts [c1, c2,...] do
sum ← sum + c
EMIT(term t, count sum)
```

How do I choose which keys go to one reduce task and which to the other? Partitioners.

Partitioners:

Partitioners work *on output of mappers and before reducers*: **decide for each key (with its values) to which reducer task must go**. Partitioner only considers the key and ignores the value**.**

By default, intermediate keys are hashed to reducers: Hash code = (key) % number of reducers

Combiners:

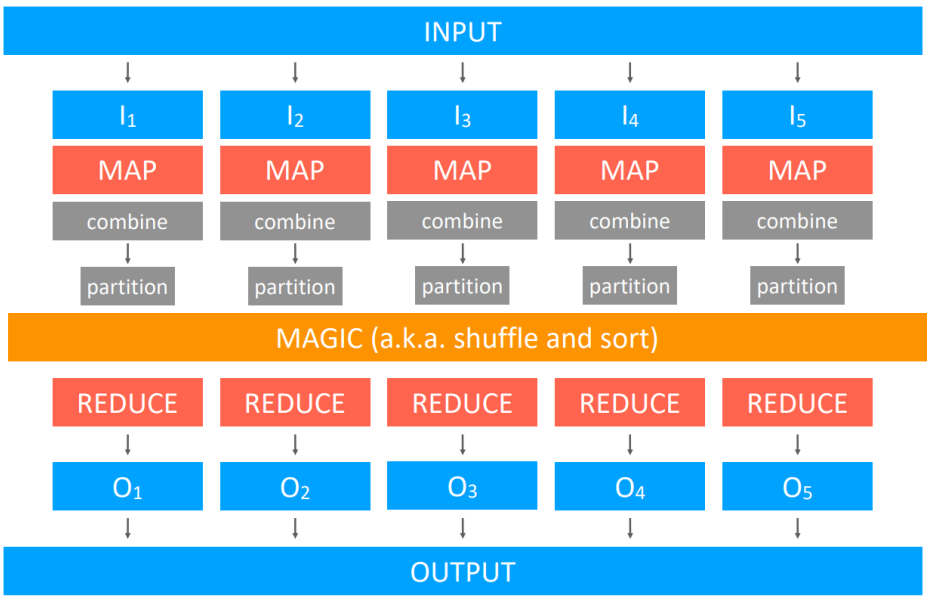
In some cases it can be advantageous to make the map tasks do something more by anticipating the work of the reducers.

Combiners perform local aggregation before the shuffle. ***Combiners are an optimization in Map Reduce.***

Since all the key-value pairs generated by the mappers need to be transmitted across the network, the size of the intermediate data can sometimes exceed that of the original input collection. Therefore, it is beneficial to perform local aggregation on the output of each mapper, which occurs on the same machine where the mapper executes.

Typically, a combiner is a (local) **copy** of the reducer, a «mini-reducer» that takes place on the output of a mapper.

To use combiners, the reduce function must be associative and commutative (like in word count, the sum)



Optimizations in Map Reduce: the most important **bottleneck** is data exchange between the map and the reduce phases, because it is an all-to-all communication and depends on the intermediate data, which depends on the application code and input data.

Knowing how many map function invocations per machine and how many reduce function invocations per machine it is possible to optimize the runtime by looking at the implementation details (→ this means **breaking** functional programming paradigm!)

Hadoop

Hadoop is an open-source framework used for distributed storage and processing of large volumes of data on computer clusters. It is designed to provide a scalable and reliable solution for parallel data processing.

The main components of Hadoop include:

1. **Hadoop Distributed File System (HDFS)**: It is a distributed file system that can store large amounts of data reliably across a cluster of machines. The data is divided into blocks and replicated across multiple nodes within the cluster to ensure fault tolerance.
2. **MapReduce**: It is a programming model and software implementation for parallel processing on large datasets. It consists of two main phases: the mapping phase where data is processed in parallel, and the reducing phase where intermediate results are combined to produce the final output.
3. **YARN (Yet Another Resource Negotiator)**: It is the resource management framework of Hadoop. It handles the scheduling and allocation of computing resources within the cluster efficiently for running applications.

Terminology:

A **MapReduce job** is a unit of work that the client wants to be performed. It consists of:

- Input data
- MapReduce program (map and reduce functions)
- Configuration information

Hadoop runs the MapReduce job by dividing it into ***tasks***. There are two types of tasks: ***map tasks*** and ***reduce tasks***.
Tasks are scheduled using **YARN** (**Yet Another Resource Negotiation**) and run on a nodes in the cluster. If a task fails, it will be automatically rescheduled to run on a different node.

Hadoop divides the input to a MapReduce job into fixed-size pieces called **Input Splits**.
Hadoop creates one map task for each split, which runs the user-defined map function for each record in the split.

Hadoop installation:

Generally Hadoop can run in three modes:

1. ***Local (or standalone) mode*** – this is the default mode
   Hadoop runs as a *single Java process*. Mainly used for debugging.
   There are no daemons (like HDFS and Hadoop itself) used in this mode. Hadoop uses the local file system as a substitute for HDFS file system.
   The jobs will run as if there is 1 mapper and 1 reducer

2. ***Pseudo-distributed mode*** (also known as single-node cluster)
   All the daemons run on a single machine. *A separate JVM for every Hadoop component*.
   This setting mimics the behavior of a cluster: all the daemons run on your machine locally using the HDFS protocol.
   There can be multiple mappers and reducers (1 reducer by default)

3. ***Fully-distributed mode***
   This is how Hadoop runs on a real cluster: *all the daemons run on (a subset of) the cluster's machines using the HDFS protocol*.
   There are multiple mappers and reducers (1 reducer by default)
   This is how we installed it in this course.

Example: word count in Hadoop

Mapper

```
// Mapper as a Generic, speicifies the type for: < input key, input value, output key, output value>
// IntWritable it's a type definied by Hadoop: a Wrapper for a int. It's optimized for serialization and deserialization
public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>
{
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();                          // Text is a writable: wrapper for a String

  public void map(Object key, Text value, Context context)      /* Context: object that allows the Map
                      throws IOException, InterruptedException {  function to write the actual intermediate
    StringTokenizer itr = new StringTokenizer(value.toString());  data somewhere (equivalent of emit of
    while (itr.hasMoreTokens()) {                                  the pseuudo code) */
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}
```

The **Text** class is part of the org.apache.hadoop.io package and implements the **Writable** interface, which allows objects to be efficiently read from or written to Hadoop's format. It represents a sequence of Unicode bytes, which corresponds to a character string.

Iterable object is not equal to a List object: in Iterable the number of elements is not known; the iterable returns a reference to the current object.

For each input split a Mapper will be created, the Mapper will call the map() function on each line of the input split.

Reducer

```
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)      //Take a list of values
                      throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();          //This is the implementation, in this case is the sum but it can be whatever
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

Driver code: *in the following lines, config and setting for map reduce jobs are specified.*

```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
                                                                    // Parsing of CLI args
  if (otherArgs.length < 2){
   System.err.println("Usage: wordcount <in> [<in>...] <out>");
   System.exit(2);
  }
  Job job = Job.getInstance(conf, "word count");        // Configuration obj and name of map reduce job
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);                    // Type of the output key
  job.setOutputValueClass(IntWritable.class);
  for (int i = 0; i < otherArgs.length - 1; ++i)
    FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
  FileOutputFormat.setOutputPath(job, new Path(otherArgs[otherArgs.length - 1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);     // true: verbose or not mode
}
```

/* When you provide your driver code, it can be in a differnt jar file from the mapper and the reducer class: you can explicitally tell where mapper and reducer are.
You say: they are in the jar that contains this class. */

// usually match with MapperOutputKeyClass and valueClass, if not use the method setMapperOutputKeyClass, etc

/* input arguments are file path: 1 input path or more, and only one output path.
Input: it can be a file, a dir or a file pattern */

With the functions `setOutputValueClass()` and `setOutputKeyClass()` we declare the types of the output variables of the reducer.
Why don't we have the key and value type for the mapper? Because, by default, those types match with these type of the output of the reducer. If they don't match we need to provide these information using the `setMapOutputValueClass()` and `setMapOutputKeyClass()`.

Once the configurations are finished, you can run a MapReduce job with a single method call: `submit()` on a Job object (you can also call `waitForCompletion()`, which submits the job if it hasn't been submitted already, then waits for it to finish).

One or more input file but only one output file.

Hadoop Types

- *map function*: from [key (K1), value (V1)] pair to list of [key (K2), value (V2)] pairs
- *combiner function*: from [key (K2), list of values (V2)] pair to list of [key (K2), values (V2)] pairs
- *reduce function*: from [key (K2), list of values (V2)] pair to list of [key (K3), values (V3)] pairs
- *partitioner function*: from [key (K2), value (V2)] pair to integer

- If a combiner function is used, then it has the same form as the reduce function, except its output types are the intermediate key and value types (K2 and V2)
- Often the combiner and reduce functions are the same, in which case K3 is the same as K2, and V3 is the same as V2
- The partition function operates on the intermediate key and value types (K2 and V2) and returns the partition index.

Hadoop Job Configuration

Property	Job setter method	Input types		Intermediate types		Output types	
		K1	V1	K2	V2	K3	V3
Properties for configuring types:							
mapreduce.job.inputformat.class	setInputFormatClass()	•	•				
mapreduce.map.output.key.class	setMapOutputKeyClass()			•			
mapreduce.map.output.value.class	setMapOutputValueClass()				•		
mapreduce.job.output.key.class	setOutputKeyClass()					•	
mapreduce.job.output.value.class	setOutputValueClass()						•
Properties that must be consistent with the types:							
mapreduce.job.map.class	setMapperClass()	•	•	•	•		
mapreduce.job.combine.class	setCombinerClass()			•	•		
mapreduce.job.partitioner.class	setPartitionerClass()			•	•		
mapreduce.job.output.key.comparator.class	setSortComparatorClass()			•			
mapreduce.job.output.group.comparator.class	setGroupingComparatorClass()			•			
mapreduce.job.reduce.class	setReducerClass()			•	•	•	•
mapreduce.job.outputformat.class	setOutputFormatClass()					•	•

Input types are set by the input format: for instance, a **TextInputFormat** (which is the default) generates keys of type LongWritable and values of type Text.
The other types are set explicitly by calling the methods on the Job. If not set explicitly, the intermediate types default to the (final) output types, which default to LongWritable and Text.
For instance, if K2 (mapper output) and K3 (reducer input) are the same, you don't need to call `setMapOutputKeyClass()`. Similarly, if V2 and V3 are the same, you only need to use `setOutputValueClass()`.

Pay attention: It is possible to configure an Hadoop job with incompatible types, because the configuration isn't checked at compile time.

Default configuration

The only configuration that we set is an input path and an output path.

- The default input format is TextInputFormat
  - The *input key is of type LongWritable*, and the *value is of type Text*
- The default mapper is the Mapper class, which writes the input key and value unchanged to the output
  - The *output key is of type LongWritable*, and the *output value is of type Text*.
- The default reducer is the Reducer class, which simply writes all its input to its output.
  - The *output key is of type LongWritable*, and the *output value is of type Text*

- The default partitioner is ***HashPartitioner**, which hashes an intermediate key to determine which **partition** the key belongs to*.
  - Each partition is processed by a reduce task.
    So the number of partitions is equal to the number of reduce tasks for the job

- By default there is a single reducer, and therefore a single partition. The programmer can choose how many reducers he wants.
- We did not set the number of map tasks: the number is equal to the number of splits that the input is turned into. *It depends on the size of the input and the file's block size (if the file is in HDFS).*

Serialization

Serialization is the process of turning objects into a byte stream, deserialization is the reverse process of turning a byte stream back into a series of objects.
Serialization is used in two quite distinct areas of distributed data processing:

- ***for interprocess communication,*** implemented using remote procedure calls (RPCs).

- ***for persistent storage***

Hadoop uses its own serialization format called **Writables**: very compact and fast, but not so easy to extend or use from languages other than Java.
There are other serialization frameworks supported in Hadoop.

Writable interfaces *(IntWritable implements it)*

```
package org.apache.hadoop.io;

import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;

public interface Writable
{
  void write(DataOutput out) throws IOException;

  void readFields(DataInput in) throws IOException;
}
```

```
package org.apache.hadoop.io;

                                        // Why Comparable? Need to compare for the sort phase
public interface WritableComparable<T> extends Writable, Comparable<T>
{
}
```

Writable Wrappers

Java primitive	Writable implementation	Serialized size (bytes)
boolean	BooleanWritable	1
byte	ByteWritable	1
short	ShortWritable	2
int	IntWritable	4
	VIntWritable	1–5
float	FloatWritable	4
long	LongWritable	8
	VLongWritable	1–9
double	DoubleWritable	8

Input

An input split is a portion of the input that is processed by a single map task.

Each split is divided into records, and the *map task processes each record – a key-value pair – in turn.*

Splits and records are logical: they are *references*, not required to be files, although commonly they are. In a database context, a split might correspond to a range of rows from a table, and a record to a row in that range.

Input splits are represented by the class InputSplit.

An InputSplit has a length in bytes and a set of storage locations (i.e., hostname strings).

A split doesn't contain the input data, it's just a reference to the data. The storage locations are used by Hadoop to place map tasks as close to the split's data as possible.

The size is used to order the splits so that the largest get processed first, to minimize the job runtime.

As a MapReduce application writer, you do not need to deal with InputSplits directly, as these are created by an InputFormat interface implementation.

It is enough for us to know that that splits are then sent to the application master, which uses their storage locations to schedule map tasks that will process them on the cluster (we will see it in the next lecture).

Setup and cleanup methods

It is common to want your Mapper or Reducer to execute some code before the map() or reduce() method is called for the first time.

Initialize data structures, read data from an external file, set parameters: the ***setup()*** method is run before the map() or reduce() method is called for the first time.

Similarly, you may wish to ***perform some action(s) after all the records have been processed*** *by your Mapper or Reducer: th*e ***cleanup()*** method is called before the Mapper or Reducer terminates.

HDFS (Hadoop File System)

Hadoop File System (HDFS) is a distributed file system that is part of the Apache Hadoop framework. It is designed to store and manage large amounts of data across multiple machines in a distributed computing environment. HDFS provides fault tolerance, high throughput, and scalability, making it suitable for processing big data.

Feature we want from a Distributed File System:

- manages storage across a network of machines in a cluster.
- Designed to run on clusters of commodity hardware
    - Does not require expensive, highly-reliable hardware
    - Commonly available, low-cost (commodity) hardware
- **Highly fault-tolerant**: Failures are the norm rather than the exception. Every day we can have several failures.
  Replication: if some node fails, we don't lose data

Organization of a DFS:

- Files (we talk about huge files, TBs) are divided into **chunks** (or *blocks*), typically **64/128 megabytes** in size.
  Blocks are replicated at different compute nodes (usually 3+)
- Nodes holding copies of one block are located on different racks.
- Block size and the degree of replication can be decided by the user
- A special node ***(the master node) stores, for each file, the positions of its blocks***
  The master node itself is replicated

Usually if the dimension of the file is not multiple of the dimension of the chunk, the remain part of a file doesn't occupy the entire block, but just the size remained.
For example: 64MB blocks, file of 70MB → 1 Block and 6MB (in single disk filesystem should be 2 blocks)
In order to reduce the overhead of each seeks, you prefer to have large blocks.

Namenodes & Datanodes

In HDFS we have a master-slave architecture

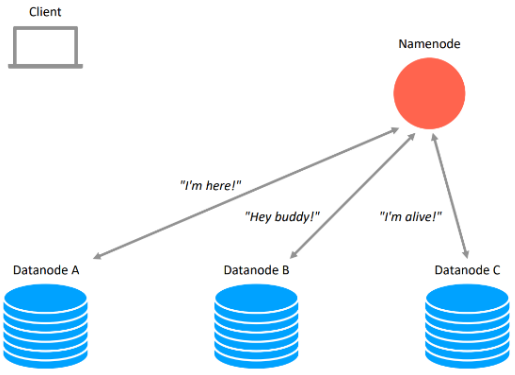
- Master: **Namenode**
- Slaves: **datanodes**

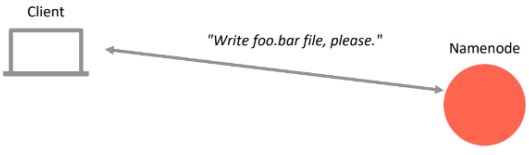
Namenode (single) is a service that manages the file system namespace and regulates access to files by clients:

- Maintains filesystem tree
- Files metadata
- *File-to-block mapping*, each block to which file belongs
- *Location of blocks*, each block on which datanode is stored
- Access permissions

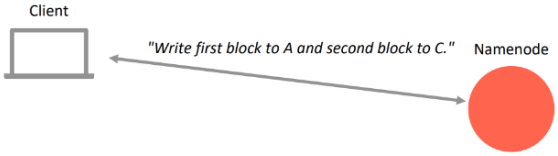
Datanodes (more than one) are the actual managers of data:

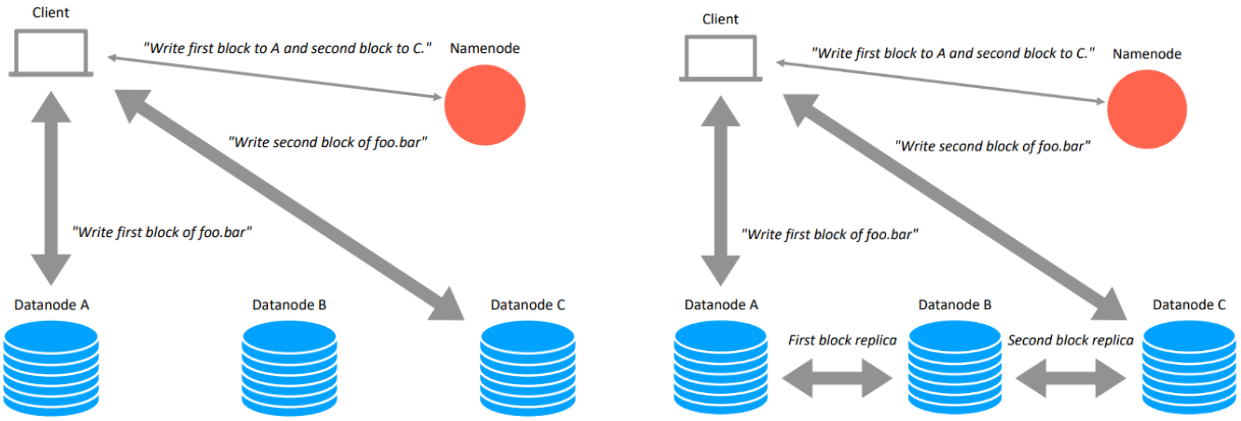
- A file is split into one or more blocks, and blocks are stored in Datanodes.
- The Datanodes manage storage attached to the nodes that they run on.
- Datanodes serve ***read/write requests***, perform block creation, deletion, and replication upon instruction from Namenode
- Each Datanode sends a heartbeat message periodically to the Namenode.





The Namenode responds with:

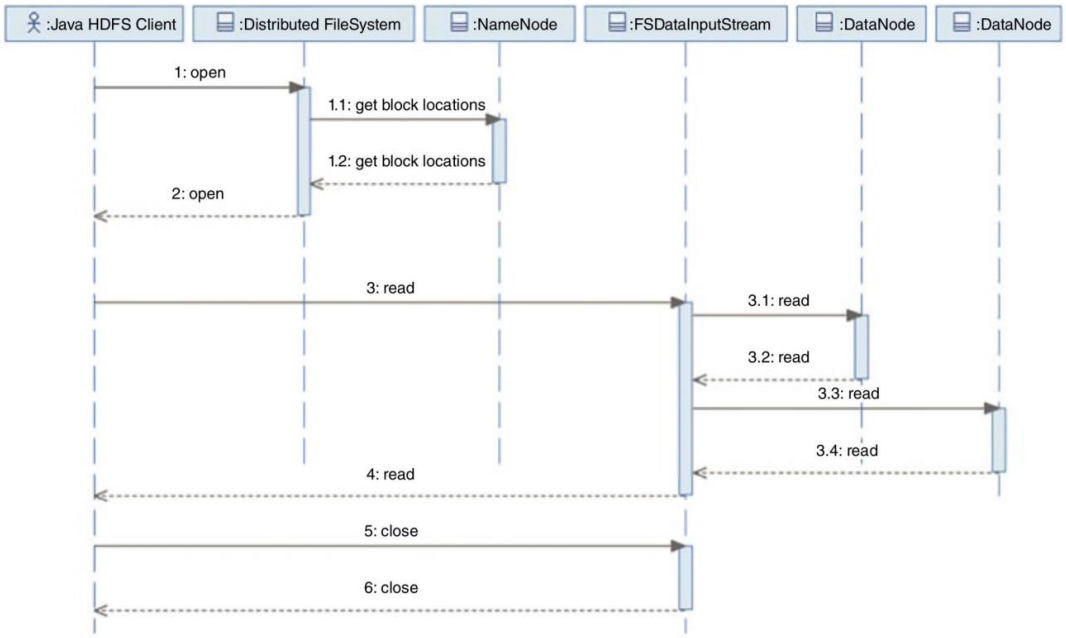




Note that any data transfer in read or write happens directly between client and datanode, doesn't happen through the namenode. The namenode is only to allocate resources, manage permissions ecc…

Note: the replication of data is managed by the Datanodes.

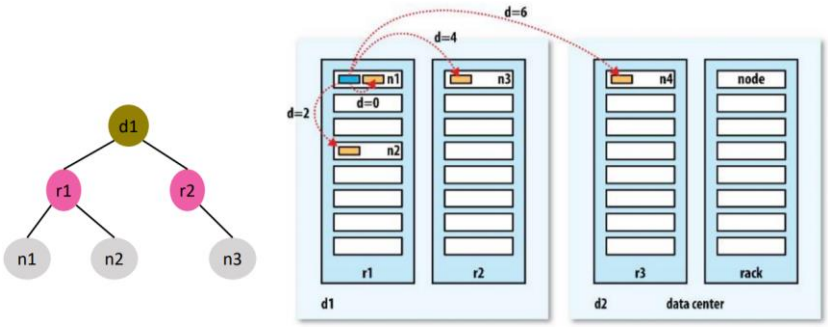
Anatomy of a read



The client uses the Java class `DistributedFileSystem` that allows the Java client to interact with Namenode. The client asks to `open` a file to read it, this interface interacts with the Namenode to get the block locations. The Namenode knows which block compose the file and were these are stored, so for each for block it indicates the locations were the block is available. It does not provide one possible location, but it gives all the possible locations on which each block is replicated. In this list the Datanodes are sorted according to their proximity to the client (according to the topology of the cluster's network). So the first node in the list will be the closest node to the client.

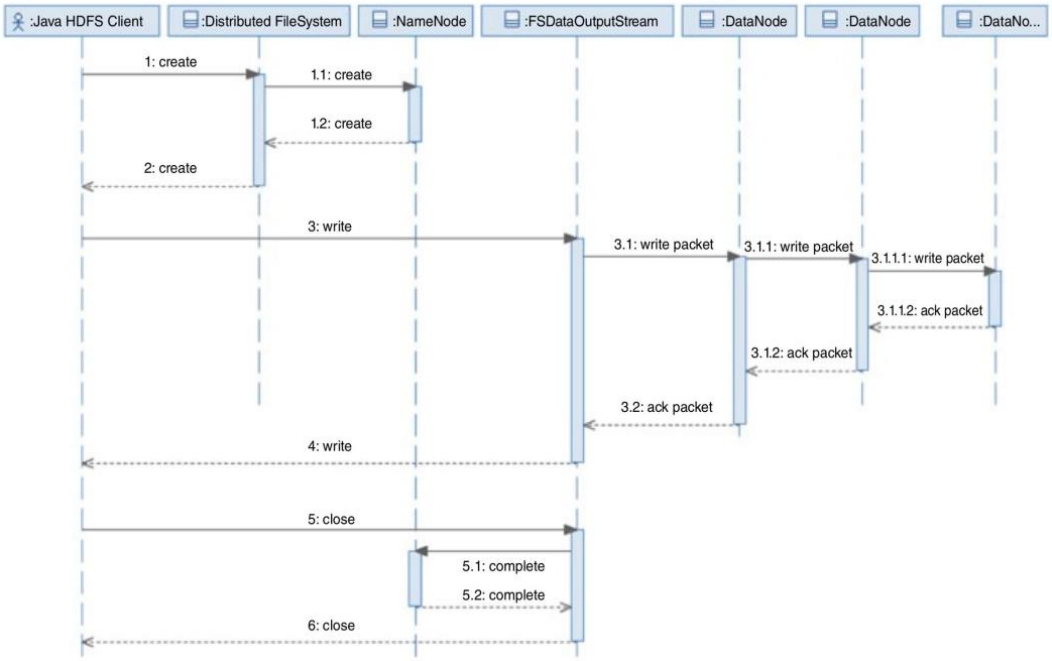
So the `DistributedFileSystem` returns an `FSDataInputStream` which is an object used to read data managing the Datanode I/O. The client then calls `read()` on the stream (step 3).
The stream, which has stored the datanode addresses for the blocks, connects to the first (closest) datanode for the first block in the file. When the end of the block is reached, `FSDataInputStream` will close the connection to the datanode, then find the best datanode for the next block.
This happens transparently to the client, which from its point of view is just reading a unique continuous stream.

Network topology
The network is represented as a tree and *the **distance** between two nodes is the sum of their distances to their closest common ancestor.*



d(r1,n1) = 1; d(r1,n2) = 1 → d(n1,n2) = 2
d(n1,n3) = 4

Anatomy of a write

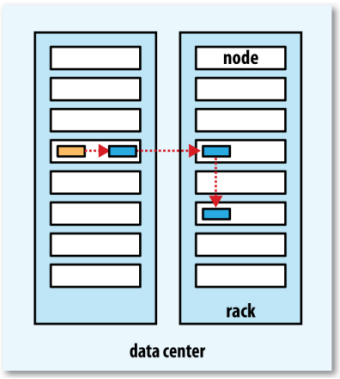


The client asks to the Namenode to create a file, the Namenode computes the file-to-block mapping and for each block it tells where each block must be stored through data nodes (only one address per block). This information block by block is returned to `DistributedFileSystem` which creates an `FSDataOutputStream` object.

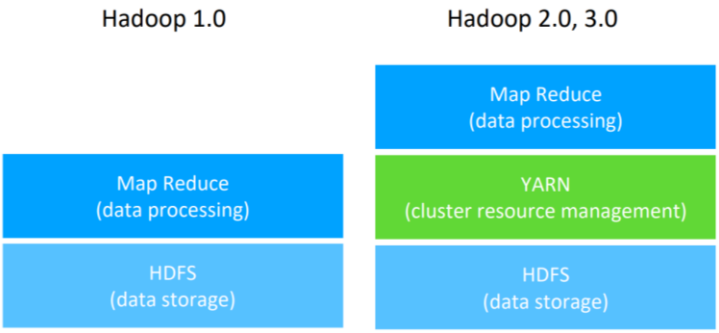
`FSDataOutputStream` : write each block in the closest datanode, then is up to datanodes to perform replication. First data nodes store the packet, the packet forward to the next data node and so on.
The `FSDataOutputStream` will advertise the NameNode, which know that every thing is stored divided in block store in that data nodes.

Replica Placement
How does the Namenode choose which Datanodes to store the replicas on?

- *First replica on the same node as the client* (For clients running outside the cluster, a node is chosen at random)
- Second replica on a different rack from the first, chosen at random
- *Third replica on the same rack as the second, but on a different node chosen at random*
- Further replicas are placed on random nodes in the cluster (The system always tries to avoid placing too many replicas on the same rack/node)



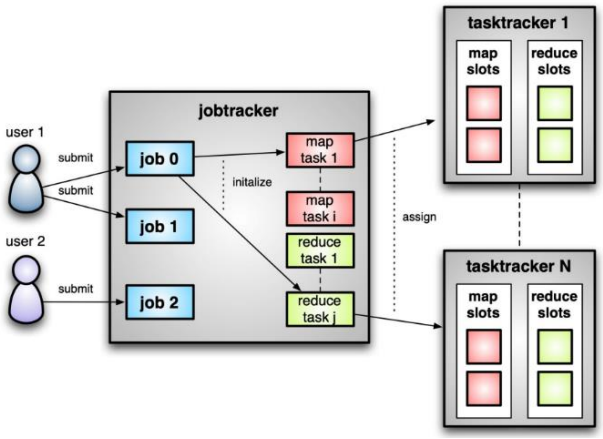
Hadoop distributed resource management



In Hadoop 1.0 there were only 2 systems: MapReduce execution Framework and HDFS. The first one had to handle also resources, but it was too much.

In **Hadoop 1.0** we had:

- ***Job***: Unit of work that the client wants to be performed (like an entire MapReduce process)
- ***Task***: *Unit of work that Hadoop schedules and runs run on nodes* in the cluster (map tasks & reduce tasks)
- ***Slot***: Processing element for tasks
- **Job Tracker**
  - Accepts jobs submitted by users and creates tasks
  - Assigns map and reduce tasks to Task Trackers
  - Monitors tasks and Task Trackers status, keeping a record of progress for each job and re-executing tasks upon failure



- **Task Tracker**
  - Runs map and reduce tasks upon instruction from the Job Tracker
  - *Manages storage and transmission of intermediate output*
  - Sends progress reports to the Job Tracker

Problems of this architecture:

- **Job tracker is performing too many things**: assign tasks and monitor them should be performed by another component.
- ***Slots***: each task tracker was configured with a pre-configured number of slots, each slot represents a set of compute resources (a cert amount of RAM and CPU). All the slots have the same amount of resources:
  - ***Preconfigured: you cannot say I need more or less for that slot***
  - ***Each slot has a tag: map/reduce slot; each slot can be used only for a type of task***

Limitations:

- ***Scalability***
  - Job Tracker performs resource allocation and monitoring for all the jobs
  - No more than 4,000 nodes and 40,000 concurrent tasks (whereas with YARN, it goes up to 10,000 nodes and 100,000 tasks)
- ***Availability***
  - ***Job tracker is a single point of failure, any failure kills all queued and running jobs***
    (we also have a in single point of failure in YARN but here the job tracker does too many things)
  - Replicating the state of this component to achieve availability can be complex (due to the large amount of data that compose it)
- **Resource Utilization**
  - Due to the predefined number of map and reduce slots for each Task Tracker, utilization issues occur, e.g., a reduce task has to wait because only map slots are available in the cluster
  - Furthermore, a slot can be too big (waste of resources) or too small (which may cause a failure) for a particular task

YARN

YARN stands for Yet Another Resource Negotiator. It is the resource management framework in Apache Hadoop that handles resource allocation and job scheduling in a Hadoop cluster. YARN is responsible for managing and coordinating the compute resources (CPU, memory, etc.) across the cluster and providing a framework for running distributed applications.

YARN components

YARN provides its core services via two types of long-running daemons:

- a **Resource Manager** (one per cluster) to manage the use of resources across the cluster. It allocates resources to applications and tracks resource utilization across the cluster. The Resource Manager receives resource requests from applications and schedules tasks to run on the available cluster resources.
- **Node Managers** (one per node in the cluster) is responsible for managing resources on a particular node. The Node Manager reports the available resources to the Resource Manager and manages the execution of application tasks on the node.

It has two other key components:

- **Application Master**: For each application running on the cluster, YARN launches an Application Master. The Application Master is responsible for negotiating resources with the Resource Manager and coordinating the execution of tasks within the application. It monitors the progress of the application, handles failures, and requests additional resources as needed.
- **Containers**: (equivalent of a slot in Hadoop 1.0) YARN manages resources in the form of containers. A container represents a set of computer resources (CPU, memory, etc.) allocated on a node where an application task can be executed. The Resource Manager allocates containers to applications based on their resource requirements. No more tags indicating map or reduce tasks.

The **difference with Hadoop 1.0** is that the resource manager is not doing the same things the job tracker was doing. Resource manager does only half of the work: allocation of resources, so it's not doing progress report monitoring that will be performed by the **application master** introduced by YARN.

A resource request for a set of containers can express:

- the amount of computer resources required for each container (memory and CPU)
- locality constraints for the containers in that request (e.g., allocate container on a node where there is a replica of the HDFS block). Mainly for Map tasks, in order to move the process to the data and not the opposite.

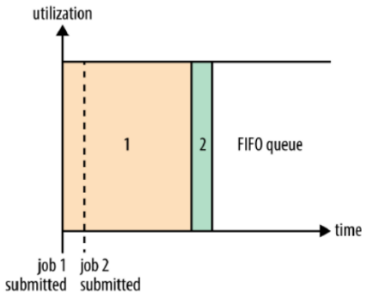
If the locality constraint cannot be met: *no allocation is made*, or *the constraint can be loosened* (e.g., on another node in the same rack).

YARN scheduling

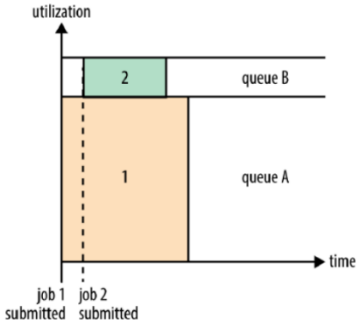
We have to think we are not talking about a single user submitting a single job in a cluster, but typically what happens is that you have several users using the cluster, each submitting MapReduce jobs. So we need some kind of scheduling to decide how to allocate resources among the different jobs of different users.
Cluster resources are limited and scheduling (i.e., allocating resources to applications) is a complex problem and there is no one "best" policy.

YARN provides tree different schedulers that allocate resources to applications based on different policies:

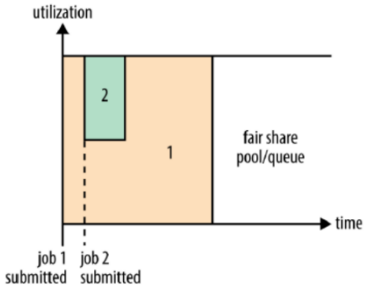
- **FIFO scheduler**
  Places applications (MapReduce job) in a *single queue and runs them in the order of submission* (first in, first out).
  Requests for the first application in the queue are allocated first; once its requests have been satisfied, the next application in the queue is served.
  Simple to understand and no configuration needed.
  Not suitable for shared clusters. ***Large applications will use all the resources in a cluster, so other applications have to wait their turn.***



- **Capacity scheduler**
  The Capacity Scheduler has *separate queues.*
  Each queue dedicated to an organization or to an user accessing the cluster.
  Each queue configured to use a given fraction of the cluster capacity: within a queue, applications are scheduled using FIFO scheduling.
  A large job finishes later than when using FIFO Scheduler



- **Fair scheduler**
  There is no need to reserve a set amount of capacity: it will dynamically balance resources among all running jobs.
  E.g., when the first job starts, it gets all cluster resources (it is the only one running). When the second job starts, it gets half of the cluster resources, and so on.
  Fair scheduling is per-user: if user A submits two jobs and user B only one, then each A's job gets 25% while B's job gets 50% of resources



Delay scheduling

All YARN schedulers try to honor locality requests. On a busy cluster, however, there is a good chance that a required node is saturated by other containers at the time of a request.

The obvious course of action is to immediately loosen the locality requirement and allocate a container on the same rack. However, it has been observed in practice that waiting a short time (no more than a few seconds) can dramatically increase the chances of being allocated a container on the requested node, and therefore increase the efficiency of the cluster.
This feature is called Delay scheduling and is supported by both the Capacity Scheduler and the Fair Scheduler.

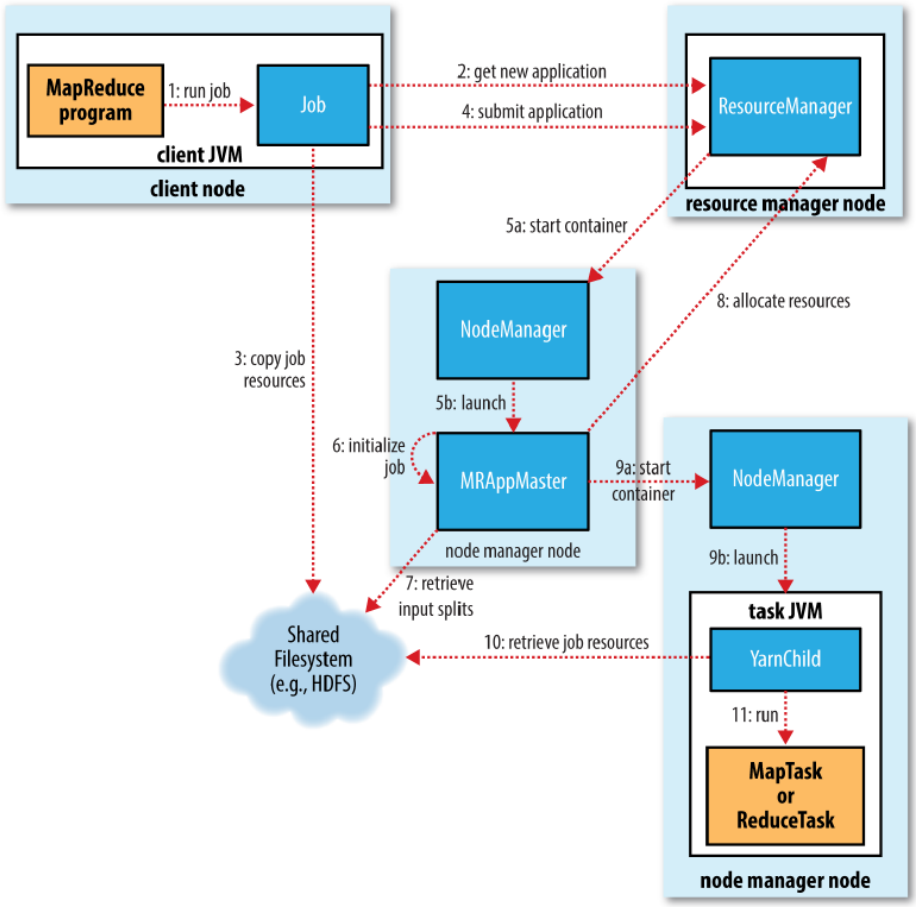
Every node manager in a YARN cluster periodically sends a **heartbeat** request to the resource manager: Heartbeats carry information about the node manager's running containers and the resources available for new containers, so each heartbeat is a potential scheduling opportunity for an application to run a container.

When using delay scheduling, the scheduler doesn't simply use the first scheduling opportunity it receives, but waits for up to a given maximum number of scheduling opportunities to occur before loosening the locality constraint.
So we are able to satisfy most of the locality constraints just by waiting few seconds and using this opportunity delay scheduling.

Anatomy of a MapReduce application run

There are five independent entities:

- The ***client***, which submits the MapReduce job.
- The ***YARN resource manager***, which coordinates the allocation of compute resources on the cluster.
- The ***YARN node managers***, which launch and monitor the compute containers on machines in the cluster.
- The ***MapReduce application master***, which coordinates the tasks running the MapReduce job.
  The **application master** and the ***MapReduce tasks*** run in containers that are scheduled by the resource manager and managed by the node managers.
- HDFS, which is used for sharing job files between the other entities.



1. The `submit()` (or `waitForCompletation()`) method on the object `job` creates an internal `JobSubmitter` instance and calls `submitJobInternal()` on it. The `JobSubmitter` instance implements the step from 2 to 4

2. Asks the Resource Manager for a new **application ID**, used for the MapReduce job ID

3. Computes the input splits for the job and interact with HDFS to store a copy of the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the shared filesystem in a directory named after the job ID

4. Submits the job by calling `submitApplication()` on the Resource Manager.

5. When the Resource Manager receives a call to its `submitApplication()` method, it pass the request to the YARN scheduler. The scheduler allocates a container, and the Resource Manager then launches the application master's process there**,** under the Node Manager's management.

   Remember that respect to the job tracker in hadoop 1.0 now we are splitting the concerns of resource allocation and progress report monitoring. Resource manager performs only resource allocation, we interact with it every time we which need to allocate resources. The application master instead is the component introduced in Hadoop 2.0 which has the purpose of actually perform monitoring progress report of each task of the job. This component is one per job. Is not like in job tracker which was one component for all the jobs, now we have one application master per job so his job is easier

6. The application master initializes the job by creating several objects to track the progress of the tasks.

7. Next, it retrieves the input splits computed in the client from the shared filesystem.
   It then creates a map task object for each split, as well as a number of reduce task objects determined by the `mapreduce.job.reduces` property.

   Note: The application master must decide how to run the tasks that make up the MapReduce job. If the job is small, the application master may choose to run the tasks in the same JVM as itself. This happens when it judges

that the overhead of allocating and running tasks in new containers outweighs the gain to be had in running them in parallel, compared to running them sequentially on one node. Such a job is said to be uberized, or **run as an uber task**.
What qualifies as a small job? By default, a small job is one that has less than 10 mappers, only one reducer, and an input size that is less than the size of one HDFS block.
Uber tasks must be enabled explicitly, for an individual job or across the cluster.

> The Resource Manager, with the help of HDFS, determines the location of the input data blocks and schedules the tasks to run on nodes where the data is already present.
> Remember, in fact, that it is the Resource Manager who has to choose on which nodes to execute a job, and it is always the Resource Manager that performs the scheduling.

8. If the job does not qualify for running as an uber task, then the application master requests containers for all the map and reduce tasks in the job from the Resource Manager

9. Once a task has been assigned resources for a container on a particular node by the Resource Manager's scheduler, the application master starts the container by contacting the node manager. The task is executed by a Java application whose main class is `YarnChild`.

10. Before it can run the task, it localizes and retrieve from HFDS the resources that the task needs, including the job configuration and JAR file, and any files.

11. The YarnChild runs in a dedicated JVM, so that any bugs in the user-defined map and reduce functions (or even in YarnChild) don't affect the node manager by causing it to crash or hang, for example.

YARN fault tolerance

We now analyze the different possible of failure, from least to most serious.

a) Task failure

- The user code in the map or reduce task throws a ***runtime exception***:
  - the task JVM reports the error back to its parent application master before it exits.
  - The application master marks the task attempt as failed
  - The ***application master frees up the container and will try to reschedule the task in another node***
- ***Sudden exit of the task JVM***
  - *the node manager informs the application master*
  - The application master marks the task attempt as failed
  - The application master frees up the container
- ***Hanging tasks***
  - The application master notices that it hasn't received a progress update for a while
  - *The task JVM process will be killed automatically after this period (typically 10 minutes)*
  - *The application master marks the task as failed*

When the application master is notified of a task attempt that has failed, it will reschedule execution of the task. The application master will try to avoid rescheduling the task on a node manager where it has previously failed

b) Application Master Failure

- Application master periodically sends ***heartbeats*** to the resource manager.
- *The resource manager will detect the failure and start a new instance of the application master running in a new container* (managed by a node manager)
- *The client needs to go back to the resource manager to ask for the new application master's address*, this is handled by Hadoop in the background and is not the responsibility of the programmer (this process is transparent to the user)
- ***If a MapReduce application master fails twice it will not be tried again and the job will fail***

c) Node manager failure

- Node managers periodically send ***heartbeats*** to the resource manager
- *The resource manager will notice a node manager that has stopped sending heartbeats if it hasn't received one for 10 minutes*
- *The resource manager will remove it (temporarily) from its pool of nodes to schedule containers on*
- ***Any task or application master running on the failed node manager will be considered failed***
- ***Node managers may be blacklisted if the number of failures for the application is high (even if node manager itself has not failed)***
- Blacklisting is done by the application master (for the application, "I don't want that my application run on that node, maybe there's incompatibility")

d) Resource Manager failure

- Failure of the resource manager is serious: without it neither jobs nor task containers can be launched
- ***In the default configuration, the resource manager is a single point of failure***
- ***To achieve high availability, it is necessary to run a pair of resource managers in an active-standby configuration***
- Information about all the running applications is stored in a highly available state store
- The standby resource manager can recover the core state of the failed active resource manager
- There's a leader election mechanism to ensure that there is only a single active resource manager at one time

Namenode (HDFS) and Resource Manager (YARN) runs on the same machine?

In a typical Hadoop cluster setup, the NameNode and the Resource Manager can run on separate machines, although they can also run on the same machine depending on the cluster configuration.
By running the NameNode and the Resource Manager on separate machines, you can achieve better fault tolerance and scalability in the Hadoop cluster. However, in smaller or development setups, it is possible to run both components on the same machine for simplicity or resource constraints.

Speculative Execution

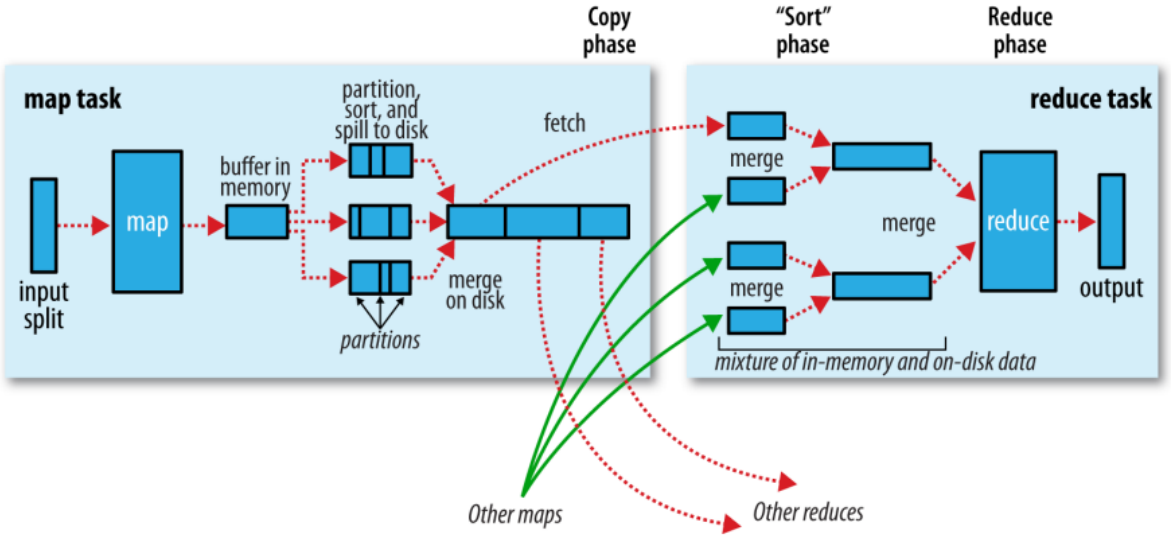
The MapReduce model aims to divide jobs into tasks and execute them in parallel, thereby reducing the overall job execution time compared to sequential execution. However, there can be instances where a particular task runs significantly slower, leading to increased job execution time.

Tasks can experience slowness due to different reasons, such as software misconfiguration or, more commonly, hardware degradation. Hadoop doesn't try to diagnose and fix slow-running tasks; instead, it tries to detect when a task is running slower than expected and only for those very slow task (if any), the application master will launches another equivalent task as a backup. Whichever one (i.e., original and speculative) finishes first, "wins". The other is killed.
This approach is called "speculative execution" of tasks.

Of course there is a cost: a few percent more resource usage. However, the benefits of this approach are significant, as it allows us to significantly accelerate the execution of tasks. The improved overall job execution time outweighs the additional resource usage, making it worthwhile to adopt this strategy.

Shuffle and Sort

MapReduce makes the guarantee that the input to every reducer is sorted by key.
The shuffle phase is the heart of MapReduce and is where the "magic" happens: it is responsible for transferring the map outputs to the reducers as inputs. In this section, we look at how the shuffle works.



The Map Side

Inside the map task, the map function processes the input split record by record (one input split per map task) and when an output key-value pair is emitted with the `context.write()` operation, these output key-value pairs are stored in a circular memory buffer (we have a buffer for each map task).
As the data in the buffer reaches a threshold, typically 80%, a separate thread defined inside the map task called the "spilling thread" is triggered. This thread takes the key-value pairs from the buffer and writes them to files called "spill files". Map outputs will continue to be written to the buffer while the spill takes place, but if the buffer fills up during this time, the map will block until the spill is complete.

Before it writes to disk, the thread first divides the data into partitions corresponding to the reducers that they will ultimately be sent to. Within each partition, the background thread performs an in-memory sort by key, and if there is a combiner function, it is run on the output of the sort. Running the combiner function makes for a more compact map output, so there is less data to write to local disk and to transfer to the reducer.
At the end of this process, you have all your output data that has been partitioned, sorted and combined. It is now ready to be sent to the reducers!

Each time the memory buffer reaches the spill threshold, a new spill file is created, so after the map task has written its last output record, there could be several spill files. The combiner, which was previously invoked on each individual spill file, is now rerun on the set of spilling files before the output file is written if there are at least three spill files.
Before the task is finished, the spill files are merged into a single partitioned and sorted output file.

Please note that during this merge phase, the operations do not involve HDFS. These operations occur in the local memory of the machine where the task is being executed.
The output file's partitions are made available to the reducers over HTTP.

The Reduce Side

Map output file is sitting on the local disk of the machine that ran the map task, but now it is needed by the machine that is about to run the reduce task for the partition. Moreover, the reduce task needs the map output for its particular partition from several map tasks across the cluster. The map tasks may finish at different times, so the reduce task starts copying their outputs as soon as each completes. This is known as the **copy phase** of the reduce task. The reduce task has a small number of copier threads so that it can fetch map outputs in parallel.

How do reducers know which machines to fetch map output from?

As map tasks complete successfully, they notify their application master using the heartbeat mechanism. Therefore, for a given job, the application master knows the mapping between map outputs and hosts. A thread in the reducer periodically asks the master for map output hosts until it has retrieved them all.

Hosts do not delete map outputs from disk as soon as the first reducer has retrieved them, as the reducer may subsequently fail. In- stead, they wait until they are told to delete them by the application master, which is after the job has completed.

When all the map outputs have been copied, the reduce task moves into the **sort phase** (which should properly be called the merge phase, as the sorting was carried out on the map side), which merges the map outputs, maintaining their sort ordering.

During the reduce phase, the reduce function is invoked for each key in the sorted output. The output of this phase is written directly to the output filesystem, typically HDFS.

Design Pattern

MapReduce is a powerful framework known for its simplicity and ease of use. The programmer's role primarily involves preparing input data, configuring the necessary information, and implementing the mapper and reducer functions. Additional functions like setup, cleanup, combiner, and partitioner can also be incorporated as needed.

One of the advantages of MapReduce's simplicity is that it establishes clear limitations on what can and cannot be done within the framework. This helps in maintaining a structured approach to problem-solving. However, it also poses a challenge as programmers need to fit their algorithms into a specific set of components that must interact in a predefined manner.

To leverage the power of MapReduce, programmers need to reshape their algorithms to align with the MapReduce programming model. This typically involves breaking down the algorithm into a sequence of one or more MapReduce jobs. It is crucial to understand how to effectively utilize the framework to ensure optimal performance and efficiency.

It is important to note that even skilled programmers can sometimes create inefficient MapReduce algorithms. However, there are established **MapReduce design patterns** that offer principles and guidelines for building better software. These patterns are not limited to a specific domain but provide a general approach to problem-solving within the MapReduce framework.

- In-mapper Combining
- Pairs and stripes
- Patterns for relational algebra operators
- Patterns for matrix-vector and matrix-matrix multiplication

In-mapper Combining

The goal is to minimize the data transferred within the cluster network, especially the output of the map tasks. This is because transferring data over a network is a expensive operation, so it is preferable to reduce the size of this data locally. One solution we have already seen is to use a combiner. Now let's explore "in-mapper combining."

Pseudo-code of stateful in-mapper combining (the true in-mapper combining)

in the stateful approach we use the same associative array for all the executions of the map() function

```
1: class Mapper
2:     method Initialize                    sarà la nostra setup() function
3:         H ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1
7:     method Close                         sarà la nostra cleaup() function
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

Which is the difference between using standard combiner and this in-mapper combining? The ultimate result, which is the output of the mapper, will be the same, but the second approach offers some advantages.

1. When using a standard combiner, you provide the combiner logic as an optimization suggestion to the execution, but it is up to Hadoop to decide whether to execute the combiner or not based on various factors. However, with in-mapper combining, Hadoop is compelled to execute the combiner because it is integrated into the Mapper logic.
2. Standard combiners operate at the end of the map task, after the records have been processed. In contrast, with in-mapper combining, you are already providing aggregated data in the circular buffer. This means that less memory is utilized within the buffer, resulting in more efficient memory usage.

Disadvantages:

- Potential ordering-dependent bugs: preserving state across records may cause algorithmic behaviour to depend on the order in which input records are encountered
- Memory scalability bottleneck: the associative array may no longer fit in memory.

Solved by: Either emitting intermediate data after processing every n (how big?) key-value pairs rather than all of them, or monitoring memory footprint and flush intermediate data to disk once memory usage has exceeded a certain (how big?) threshold

▲ ATTENTION ▲

The final part of the 'Design Pattern' slide deck is missing from these notes as I studied it directly from the slides. I advise those of you reading these notes to do the same, as that section is also part of the exam syllabus and may be asked.