



PARTNERSHIP FOR ADVANCED COMPUTING IN EUROPE

# Introduction to Hadoop (part 2)

---

Dr. Giovanna Roda

*Vienna Technical University - IT Services (TU.it)*



## Some advanced topics

- ▶ the YARN resource manager
- ▶ fault tolerance and HDFS Erasure Coding
- ▶ the `mrjob` library
- ▶ HDFS i/o benchmarking
- ▶ MapReduce spilling



## YARN

Hadoop jobs are managed by [YARN](#) (acronym for Yet Another Resource Negotiator), that is responsible for allocating resources and manage job scheduling.

Basic resource types are:

- ▶ memory (`memory-mb`)
- ▶ virtual cores (`vcores`)

Yarn supports an extensible resource model that allows to define any *countable resource*. A countable resource is a resource that is consumed while a container is running, but is released afterwards. Such a resource can be for instance:

- ▶ GPU (`gpu`)



## YARN

Each job submitted to the Yarn is assigned:

- ▶ a *container*, that is an abstract entity which incorporates resources such as memory, cpu, disk, network etc. Container resources are allocated by the Scheduler .
- ▶ an `ApplicationMaster` service assigned by the Application Manager for monitoring the progress of the job, restarting tasks if needed

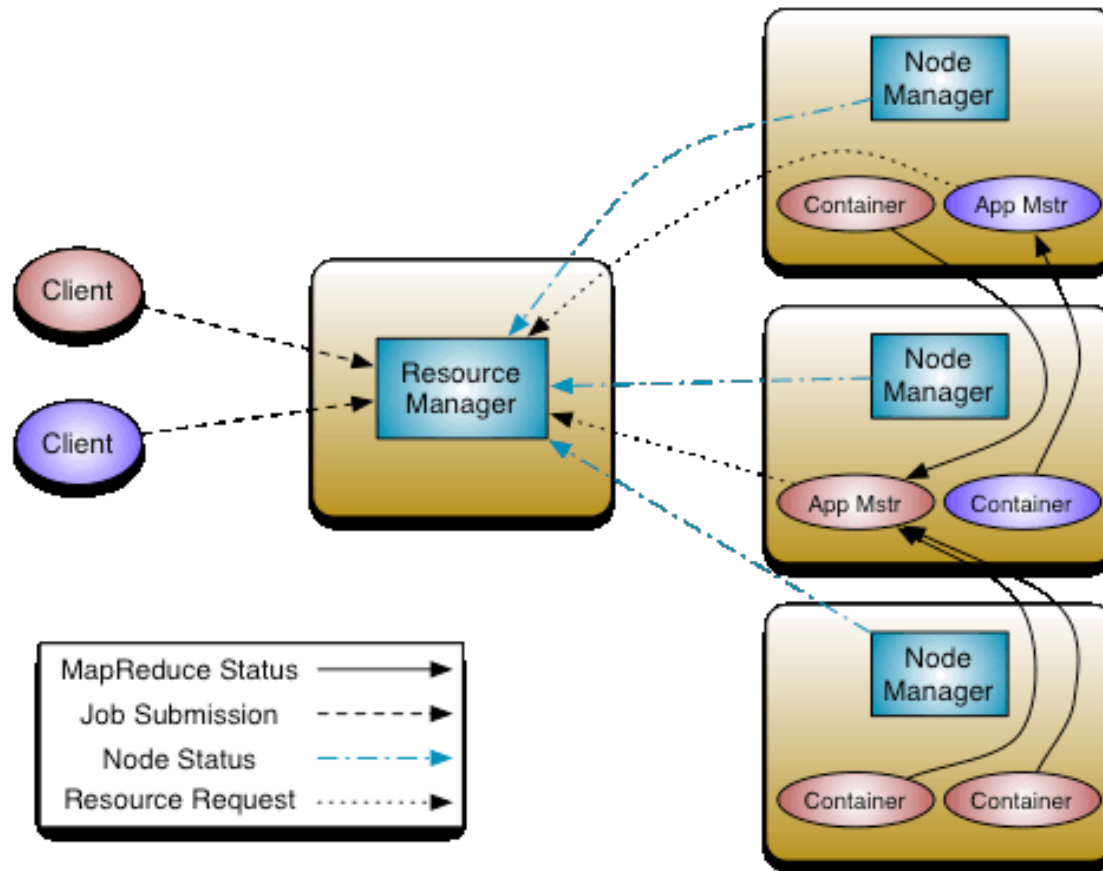


## YARN

The main idea of Yarn is to have two distinct daemons for job monitoring and scheduling, one global and one local for each application:

- ▶ the **Resource Manager** is the global job manager, consisting of:
  - ▶ Scheduler → responsible for allocating of resources across all applications
  - ▶ Applications Manager → accept job submissions, restart Application Masters on failure
- ▶ an **Application Master** for each application, a daemon responsible for negotiating resources, monitoring status of the job, restarting failed tasks.

# YARN architecture



Source: [Apache Software Foundation](http://www.apache.org)



## YARN Schedulers

Yarn supports by default two types of schedulers:

- ▶ Capacity Scheduler

  - share resources providing capacity guarantees → allows scheduler determinism

- ▶ Fair Scheduler

  - all apps get, on average, an equal share of resources over time. Fair sharing allows one app to use the entire cluster if it's the only one running.

  - By default, fairness is based on memory but this can be configured to include also other resources.



## YARN Dynamic Resource Pools

Yarn supports Dynamic Resource Pools (or dynamic queues).

Each job is assigned to a queue (the queue can be determined for instance by the user's Unix primary group).

Queue are assigned a weight and resources are split among queues according to their weights. For instance, if you have one queue with weight 20 and three other queues with weight 10, the first queue will get 40%, and each of the other queues 20% of the available resources because:

$$1 * 20x + 3 * 10x = 100$$



## Testing HDFS I/O throughput with TestDFSIO

TestDFSIO is a tool included in the Hadoop software distribution for measuring read and write performance of the HDFS filesystem.

It's a useful tool for assessing the performance of your Hadoop filesystem, identify performance bottlenecks, support decisions for tuning your HDFS configuration.

TestDFSio uses MapReduce to write and read files on the HDFS filesystem; the reducer is used to collect and summarize test data.



## Testing HDFS I/O throughput with TestDFSIO

Note: you won't be able to run TestDFSIO on a cluster unless you're the cluster's administrator. Of course, it is recommended to use TestDFSIO to run tests when the Hadoop cluster is not in use.

To run test as a non-superuser you need to have read/write access to `/benchmarks/TestDFSIO` on HDFS or else specify another directory (on HDFS) where to write and read data with the option `-D test.build.data=myDir`.



## Testing HDFS I/O throughput with TestDFSIO

TestDFSIO writes and reads file on HDFS spanning exactly one mapper for file.

These are the main options for running stress tests:

- ▶ `-write` to run write tests
- ▶ `-read` to run read tests
- ▶ `-nrFiles` the number of files (set to be equal to the number of mappers)
- ▶ `-fileSize` size of files (followed by B | KB | MB | GB | TB)



## TestDFSIO: run a write test

Run a test with 80 files (remember: this is the number of mappers) each of size 10GB.

```
JARFILE= /opt/pkg/software/Hadoop/2.10.0-GCCcore-8.3.0-  
native/share/hadoop/mapreduce/hadoop-mapreduce-client-jobclient-2.10.0-  
tests.jar
```

```
yarn $JARFILE TestDFSIO -write -nrFiles 80 -fileSize 10GB
```

The jar file needed to run TestDFSIO is `hadoop-mapreduce-client-jobclient*tests.jar` and its location depends on your Hadoop installation.



## TestDFSIO: examine test results

Test results are appended by default to the file `TestDFSIO_results.log`. The log file can be changed with the option `-resFile resultFileName`.

The main measurements returned by the HDFSio test are:

- ▶ *throughput* in mb/sec
- ▶ average *IO rate* in mb/sec
- ▶ standard deviation of the IO rate
- ▶ test execution time



## TestDFSIO: examine test results

Main units of measure:

- ▶ *throughput* or data transfer rate measures the amount of data read or written (expressed in Megabytes per second -- MB/s) to the filesystem.
- ▶ *IO rate* also abbreviated as IOPS measures IO operations per second, which means the amount of read or write operations that could be done in one seconds time.

Throughput and IO rate are connected:

$$\text{Average IO size} \times \text{IOPS} = \text{Throughput in MB/s}$$



## TestDFSIO: examine test results

Sample output of a test write run:

```
----- TestDFSIO ----- : write
      Date & time: Sun Sep 13 16:14:39 CEST 2020
      Number of files: 80
      Total MBytes processed: 819200
      Throughput mb/sec: 30.64
      Average IO rate mb/sec: 34.44
      IO rate std deviation: 17.45
      Test exec time sec: 429.73
```



## TestDFSIO: examine test results

### Concurrent versus overall throughput

The resulting throughput is the average throughput among all map tasks. To get an approximate overall throughput on the cluster one should divide the total MBytes by the test execution time in seconds.

In our test the overall write throughput on the cluster is:

$$819200 / 429.73 = 1906\text{MB}/\text{sec}$$



## TestDFSIO: some things to try

- ▶ run a read test using the same command used for the write test but using the option `-read` in place of `-write`.
- ▶ use the option `-D dfs.replication=1` to measure the effect of replication on I/O performance

```
yarn $JARFILE TestDFSIO -D dfs.replication=1 \  
-write -nrFiles 80 -fileSize 10GB
```



## TestDFSIO: remove data after completing tests

When done with testing, remove the temporary files with:

```
yarn $JARFILE TestDFSIO -clean
```

If you used a special `myDir` output directory use

```
yarn $JARFILE TestDFSIO -clean -D test.build.data=myDir
```

Further reading: [TestDFSIO documentation](#) , source code [TestDFSIO.java](#).



## The `mrjob` library

A typical data processing pipeline consists of multiple steps that needs to run in a sequence.

`mrjob` is a Python library that offers a convenient framework which allows you to write multi-step MapReduce jobs in pure Python, test them on your machine, and run them on a cluster.



## The `mrjob` library: how to define a MapReduce job

A job is defined by a class that inherits from `MRJob`. This class contains methods that define the *steps* of your job.

A “step” consists of a mapper, a combiner, and a reducer.

Step-by-step tutorials and documentation can be found in: <https://mrjob.readthedocs.io/>



## The `mrjob` library: a sample MapReduce job

```
from mrjob.job import MRJob

class MRWordFrequencyCount(MRJob):
    def mapper(self, _, line):
        yield "chars", len(line)
        yield "words", len(line.split())
        yield "lines", 1

    def reducer(self, key, values):
        yield key, sum(values)

if __name__ == '__main__':
    MRWordFrequencyCount.run()
```



## The `mrjob` library: a sample MapReduce job

Call the script `mr_wordcount.py` and run it with:

```
python mr_wordcount.py file.txt
```

This command will return the frequencies of characters, words, and lines in `file.txt`.

To get aggregated frequency counts for multiple files use:

```
python mr_wordcount.py file1.txt file2.txt ...
```



## The `mrjob` library: a sample MapReduce pipeline

To define a succession of MapReduce jobs define steps as a list of steps:

```
def steps(self):  
    return [  
        MRStep(mapper=self.mapper_get_words,  
               combiner=self.combiner_count_words,  
               reducer=self.reducer_count_words),  
        MRStep(reducer=self.reducer_find_max_word)  
    ]
```



## The `mrjob` library: run offline and on a cluster

Note that up to now we ran everything offline, on our local machine. To run the script on a cluster use the option `-r Hadoop`. This will set Hadoop as the target runner.

Alternatively:

- ▶ `-r emr` for AWS clusters
- ▶ `-r dataproc` for Google cloud



PARTNERSHIP FOR ADVANCED COMPUTING IN EUROPE

**THANK YOU FOR YOUR ATTENTION**

[www.prace-ri.eu](http://www.prace-ri.eu)