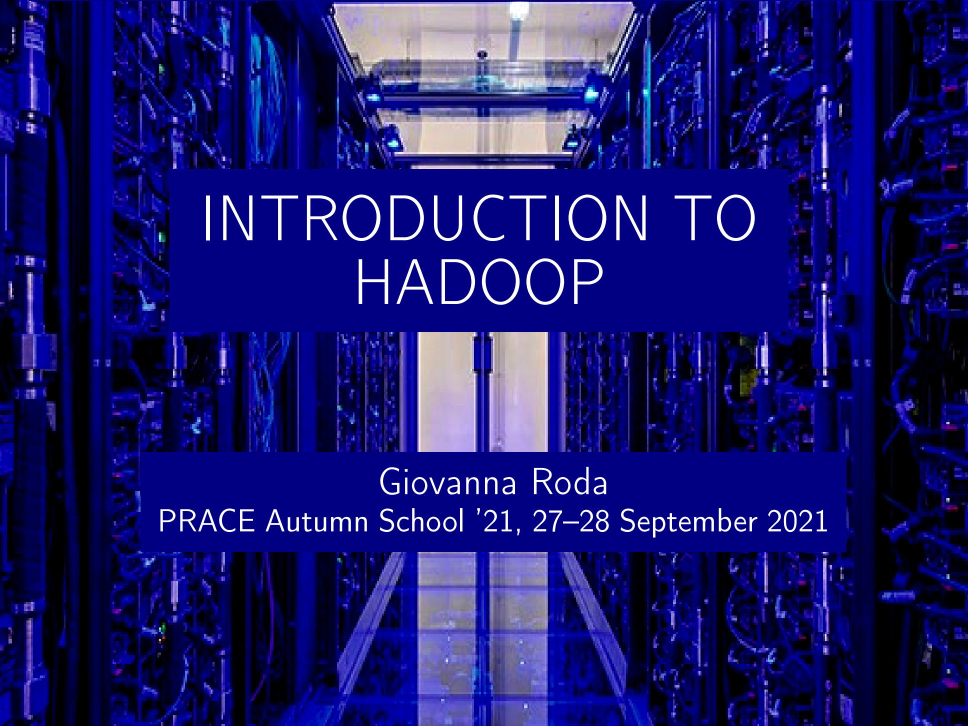


INTRODUCTION TO HADOOP

Giovanna Roda

PRACE Autumn School '21, 27–28 September 2021

Outline

- Schedule
- What is Big Data?
- The Hadoop distributed computing architecture
- MapReduce
- HDFS hands-on exercises
- MapReduce hands-on
- The YARN resource manager
- MRjob
- Benchmarking I/O with testDFSio
- Concluding remarks

Timetable

Monday, Sept. 27th	
13:00–13:30	What is Big Data?
13:30–14:30	The Hadoop distributed computing architecture
14:30–15:00	Coffee break
15:00–17:00	Hands-on exercises: HDFS and MapReduce
Tuesday, Sept. 28th	
9:00–9:30	The YARN resource manager
9:30–10:30	MRjob library
10:30–11:00	testHDFSio

Outline/next

- Schedule
- What is Big Data?
- The Hadoop distributed computing architecture
- MapReduce
- HDFS hands-on exercises
- MapReduce hands-on
- The YARN resource manager
- MRjob
- Benchmarking I/O with testDFSio
- Concluding remarks

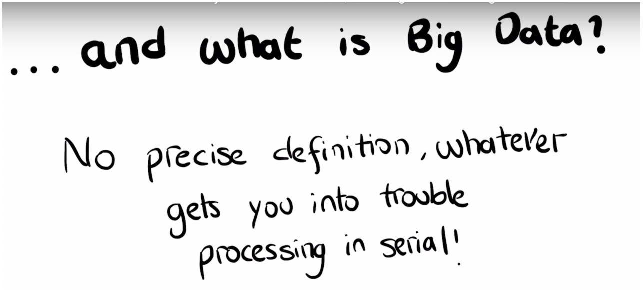
What is Big Data?

What is Big Data?

"**Big Data**" is the catch-all term for massive amounts of data as well as for frameworks and R&D initiatives aimed at working with them efficiently.

Image source: erpinnews.com

A short definition of Big Data



... and what is Big Data?

No precise definition, whatever gets you into trouble processing in serial!

A nice definition from this year's PRACE Summer of HPC presentation "Convergence of HPC and Big Data".

The three V's of Big Data

It is customary to define Big Data in terms of three V's:

- Volume (the sheer volume of data)
- Velocity (rate of flow of the data and processing speed needs)
- Variety (different sources and formats)

The three V's of Big Data

Data arise from disparate sources and come in many sizes and formats. Velocity refers to the speed of data generation as well as to processing speed requirements.

Volume	**Velocity**	**Variety**
MB	batch	table
GB	periodic	database
TB	near-real time	multimedia
PB	real time	unstructured
...	...	...

Reference: metric prefixes

1000000000000000000000000	10^{24}	yotta	Y	septillion
1000000000000000000000	10^{21}	zetta	Z	sextillion
1000000000000000000	10^{18}	exa	E	quintillion
1000000000000000	10^{15}	peta	P	quadrillion
1000000000000	10^{12}	tera	T	trillion
1000000000	10^{9}	giga	G	billion
1000000	10^{6}	mega	M	million
1000	10^{3}	kilo	k	thousand

Note: 1 Gigabyte (GB) is 10^9 bytes. Sometimes GB is also used to denote 1024^3 or 2^30 bytes, which is actually one *gibibyte* (GiB).

Structured vs. unstructured data

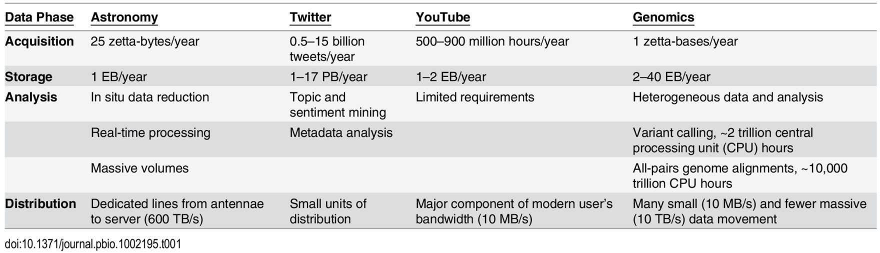
By *structured* data one refers to highly organized data that are usually stored in relational databases or data warehouses. Structured data are easy to search but unflexible in terms of the three "V"s.

Unstructured data come in mixed formats, usually require pre-processing, and are difficult to search. Structured data are usually stored in noSQL databases or in *data lakes* (these are scalable storage spaces for raw data of mixed formats).

Examples of structured/unstructured data

Industry	Structured data	Unstructured data
e-commerce	• products & prices • customer data • transactions	• reviews • phone transcripts • social media mentions
banking	• financial transactions • customer data	• customer communication • regulations & compliance • financial news
healthcare	• patient data • medical billing data	• clinical reports • radiology imagery

Big Data in 2025



Data Phase	Astronomy	Twitter	YouTube	Genomics
Acquisition	25 zetta-bytes/year	0.5–15 billion tweets/year	500–900 million hours/year	1 zetta-bases/year
Storage	1 EB/year	1–17 PB/year	1–2 EB/year	2–40 EB/year
Analysis	In situ data reduction	Topic and sentiment mining	Limited requirements	Heterogeneous data and analysis
	Real-time processing	Metadata analysis		Variant calling, ~2 trillion central processing unit (CPU) hours
	Massive volumes			All-pairs genome alignments, ~10,000 trillion CPU hours
Distribution	Dedicated lines from antennae to server (600 TB/s)	Small units of distribution	Major component of modern user's bandwidth (10 MB/s)	Many small (10 MB/s) and fewer massive (10 TB/s) data movement

doi:10.1371/journal.pbio.1002195.t001

This table[1] shows the projected annual storage and computing needs in four domains (astronomy, social media, genomics)

[1] Stephens ZD et al. "Big Data: Astronomical or Genomical?" In: *PLoS Biol* (2015).

The three V's of Big Data: additional dimensions

Three more "V"s to be pondered:

- Veracity (quality or trustworthiness of data)
- Value (economic value of the data)
- Variability (general variability in any of the aforementioned characteristics)

The challenges of Big Data

Anyone working with large amounts of data will sooner or later be confronted with one or more of these challenges:

- disk and memory space
- processing speed
- hardware faults
- network capacity and speed
- need to optimize resources use

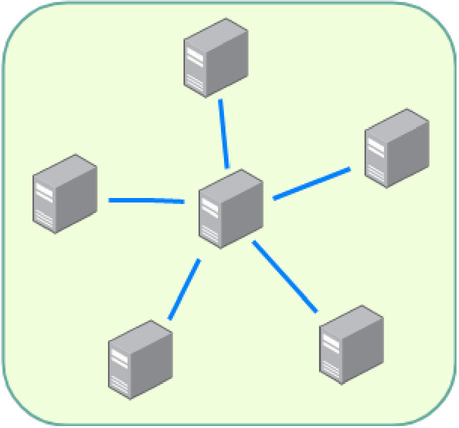
Distributed computing for Big Data

Traditional technologies are inadequate for processing large amounts of data efficiently.

Distributed computation makes it possible to work with Big Data using reasonable amounts of time and resources.

Image: VSC-4 ©Matthias Heisler

What is distributed computing?



A distributed computer system consists of several interconnected *nodes*. Nodes can be physical as well as virtual machines or containers.

When a group of nodes provides services and applications to the client as if it were a single machine, then it is also called a *cluster*.

Main benefits of distributed computing

- **Performance**: supports intensive workloads by spreading tasks across nodes
- **Scalability**: new nodes can be added to increase capacity
- **Fault tolerance**: resilience in case of hardware failures

Outline/next

- Schedule
- What is Big Data?
- The Hadoop distributed computing architecture
- MapReduce
- HDFS hands-on exercises
- MapReduce hands-on
- The YARN resource manager
- MRjob
- Benchmarking I/O with testDFSio
- Concluding remarks

The Hadoop distributed computing architecture

Hadoop for distributed data processing

Hadoop is a framework for running jobs on clusters of computers that provides a good abstraction of the underlying hardware and software.

"*Stripped to its core, the tools that Hadoop provides for building distributed systems—for data storage, data analysis, and coordination—are simple. If there's a common theme, it is about raising the level of abstraction—to create building blocks for programmers who just happen to have lots of data to store, or lots of data to analyze, or lots of machines to coordinate, and who don't have the time, the skill, or the inclination to become distributed systems experts to build the infrastructure to handle it.*[2]"

[2]White T. *Hadoop: The Definitive Guide*. O'Reilly, 1988.

Hadoop: some facts

Hadoop[3] is an open-source project of the Apache Software Foundation. The project was created to facilitate computations involving massive amounts of data.

- its core components are implemented in Java
- initially released in 2006. Last stable version is 3.3.1 from June 2021
- originally inspired by Google's MapReduce[4] and the proprietary GFS (Google File System)

[3] Apache Software Foundation. *Hadoop*. url: https://hadoop.apache.org.

[4] J. Dean and S. Ghemawat. "MapReduce: Simplified data processing on large clusters." In: *Proceedings of Operating Systems Design and Implementation (OSDI)*. 2004. url: https://www.usenix.org/legacy/publications/library/proceedings/osdi04/tech/full_papers/dean/dean.pdf.

Hadoop's features

Hadoop's features addressing the challenges of Big Data:

- scalability
- fault tolerance
- high availability
- distributed cache/data locality
- cost-effectiveness as it does not need high-end hardware
- provides a good abstraction of the underlying hardware
- easy to learn
- data can be queried trough SQL-like endpoints (Hive, Cassandra)

Mini-glossary of Hadoop's distinguishing features

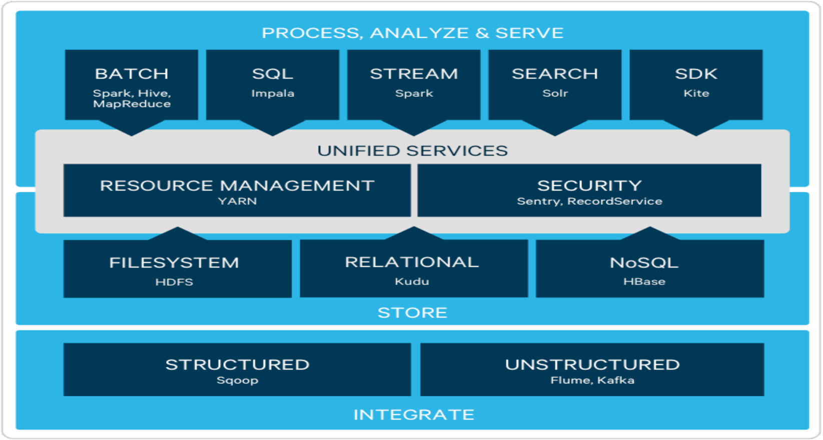
- ***fault tolerance***: the ability to withstand hardware or network failures (also: *resilience*)
- ***high availability***: this refers to the system minimizing downtimes by eliminating single points of failure
- ***data locality***: task are run on the node where data are located, in order to reduce the cost of moving data around

The Hadoop core

The core of Hadoop consists of:

- Hadoop common, the core libraries
- HDFS, the Hadoop Distributed File System
- MapReduce
- the YARN (Yet Another Resource Negotiator) resource manager

The Hadoop ecosystem



There's a whole constellation of open source components for collecting, storing, and processing big data that integrate with Hadoop.

Image source: Cloudera

The Hadoop Distributed File System (HDFS)

HDFS stands for Hadoop Distributed File System and it takes care of partitioning data across a cluster.

In order to prevent data loss and/or task termination due to hardware failures HDFS uses either

- replication (creating multiple copies —usually 3— of the data)
- erasure coding

Data redundancy (obtained through replication or erasure coding) is the basis of Hadoop's fault tolerance.

Replication vs. Erasure Coding

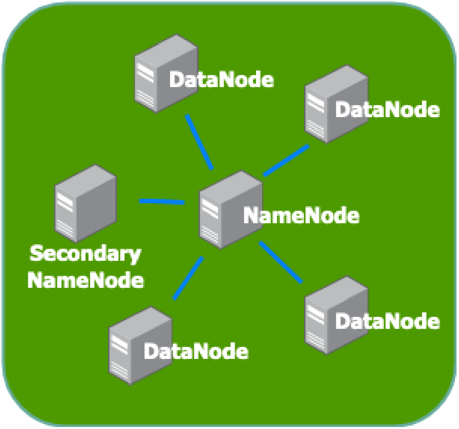
In order to provide protection against failures one introduces:

- data redundancy
- a method to recover the lost data using the redundant data

Replication is the simplest method for coding data by making n copies of the data. n-fold replication guarantees the availability of data for at most $n-1$ failures and it has a storage overhead of 200% (this is equivalent to a storage efficiency of 33%).

Erasure coding provides a better storage efficiency (up to to 71%) but it can be more costly than replication in terms of performance.

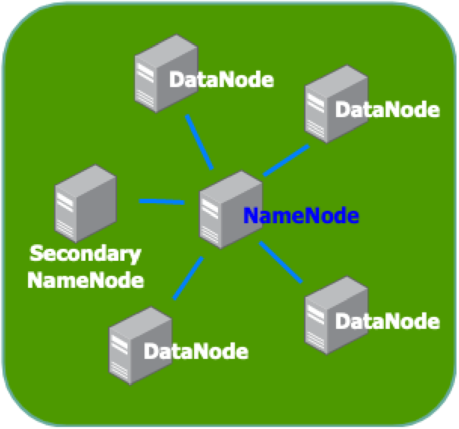
HDFS architecture



A typical Hadoop cluster installation consists of:

- a NameNode
- a secondary NameNode
- multiple DataNodes

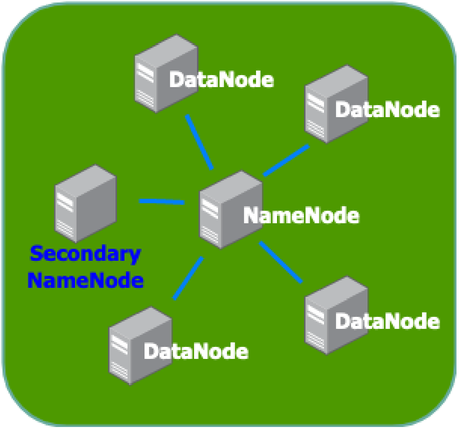
HDFS architecture: NameNode



NameNode

The NameNode is the main point of access of a Hadoop cluster. It is responsible for the bookkeeping of the data partitioned across the DataNodes, manages the whole filesystem metadata, and performs load balancing

HDFS architecture: Secondary NameNode

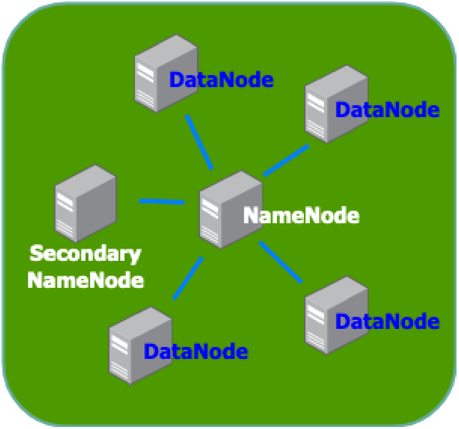


Secondary NameNode

Keeps track of changes in the NameNode performing regular snapshots, thus allowing quick startup.

An additional *standby node* is needed to guarantee high availability (since the NameNode is a single point of failure).

HDFS architecture: DataNode



DataNode

Here is where the data is saved and the computations take place (data nodes should actually be called "data and worker nodes")

HDFS architecture: internal data representation

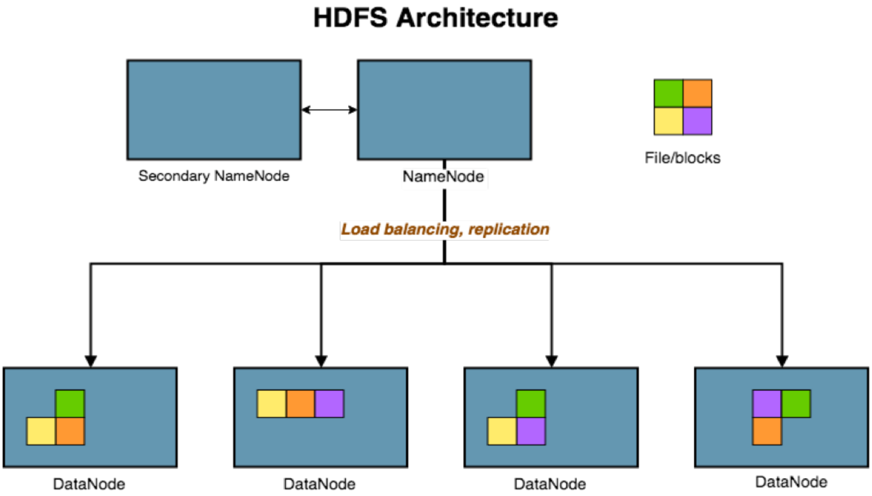
HDFS supports working with very large files.

Internally, data are split into *blocks*. One of the reason for splitting data into blocks is that in this way block objects all have the same size.

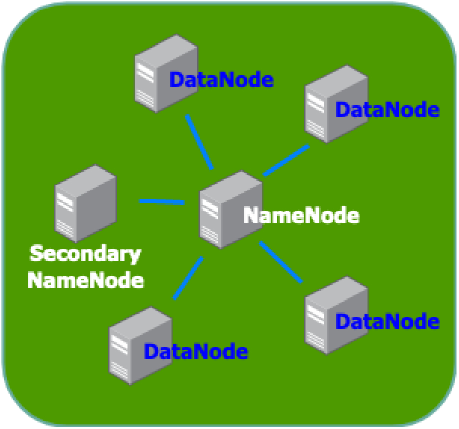
The block size in HDFS can be configured at installation time and it is by default 128MiB (approximately 134MB).

Note: Hadoop sees data as a bunch of records and it processes multiple files the same way it does with a single file. So, if the input is a directory instead of a single file, it will process all files in that directory.

HDFS architecture



DataNode failures



Each DataNode sends a Heartbeat message to the NameNode periodically. Whenever a DataNode becomes unavailable (due to network or hardware failure), the NameNode stops sending requests to that node and creates new replicas of the blocks stored on that node.

The WORM principle of HDFS

The Hadoop Distributed File System relies on a simple design principle for data known as Write Once Read Many (WORM).

"*A file once created, written, and closed need not be changed except for appends and truncates. Appending the content to the end of the files is supported but cannot be updated at arbitrary point. This assumption simplifies data coherency issues and enables high throughput data access.*[5]"

The data immutability paradigm is also discussed in Chapter 2 of "Big Data".[6]

[5] Apache Software Foundation. *Hadoop*. url: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[6] Warren J. and Marz N. *Big Data*. Manning publications, 1988.

Outline/next

- Schedule
- What is Big Data?
- The Hadoop distributed computing architecture
- MapReduce
- HDFS hands-on exercises
- MapReduce hands-on
- The YARN resource manager
- MRjob
- Benchmarking I/O with testDFSio
- Concluding remarks

MapReduce

The origins of MapReduce

The 2004 paper "*MapReduce: Simplified Data Processing on Large Clusters*" by two members of Google's R&D team, Jeffrey Dean and Sanjay Ghemawat, is the seminal article on MapReduce.

The article describes the methods used to split, process, and aggregate the large amount of data for the Google search engine.

The open-source version of MapReduce was later released within the Apache Hadoop project.

MapReduce explained

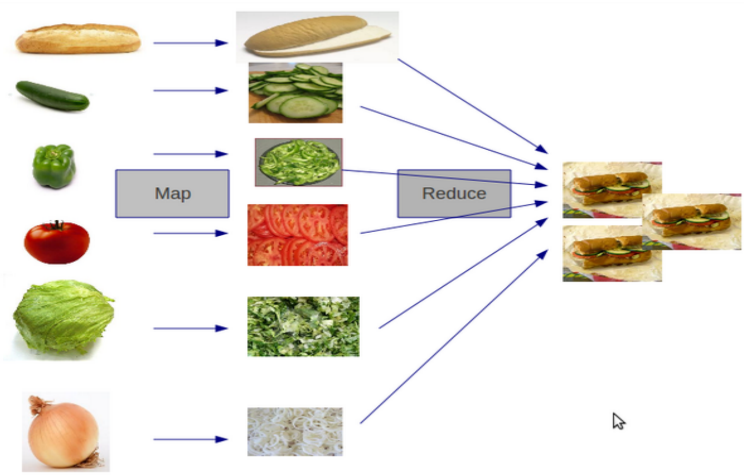


Image source: Stack Overflow

MapReduce explained

The MapReduce paradigm is inspired by the computing model commonly used in functional programming.

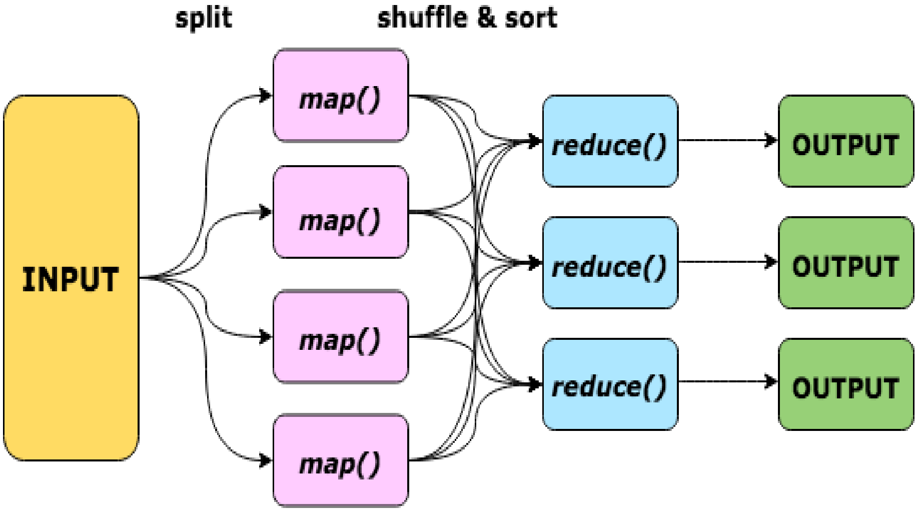
Applying the same function independently to items in a dataset either to transform (*map*) or collate (*reduce*) them into new values, works well in a distributed environment.

The phases of MapReduce

The phases of a MapReduce job:

- **split**: data is partitioned across several computer nodes
- **map**: apply a map function to each chunk of data
- **sort & shuffle**: the output of the mappers is sorted and distributed to the reducers
- **reduce**: finally, a reduce function is applied to the data and an output is produced

The phases of MapReduce



The phases of MapReduce

We have seen that a MapReduce job consists of four phases:

- **split**
- **map**
- **sort & shuffle**
- **reduce**

While splitting, sorting and shuffling are done by the framework, the map and reduce functions are defined by the user.

It is also possible for the user to interact with the splitting, sorting and shuffling phases and change their default behavior, for instance by managing the amount of splitting or defining the sorting comparator. This will be illustrated in the hands-on exercises.

MapReduce: some notes

Notes

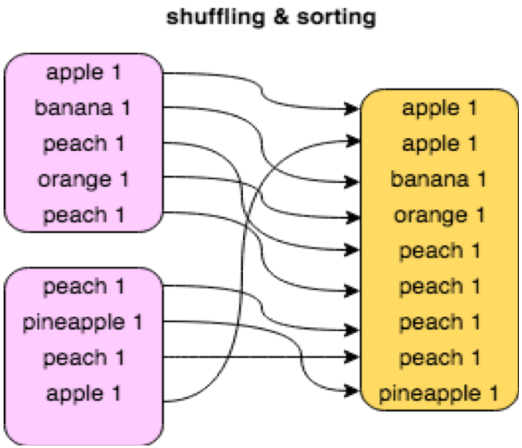
- the same map (and reduce) function is applied to all the chunks in the data
- the map and reduce computations can be carried out in parallel because they're completely independent from one another.
- the split is not the same as the internal partitioning into blocks

MapReduce: shuffling and sorting

The shuffling and sorting phase is often the the most costly in a MapReduce job.

The mapper takes as input unsorted data and emits key-value pairs. The purpose of sorting is to provide data that is already grouped by key to the reducer. This way reducers can start working as soon as a group (identified by a key) is filled.

MapReduce: shuffling and sorting



Outline/next

- Schedule
- What is Big Data?
- The Hadoop distributed computing architecture
- MapReduce
- HDFS hands-on exercises
- MapReduce hands-on
- The YARN resource manager
- MRjob
- Benchmarking I/O with testDFSio
- Concluding remarks

HDFS hands-on exercises

Where to find commands listing

For this part of the training you will need to activate the Hadoop module using the command:

```
module load Hadoop/2.6.0-cdh5.8.0-native
```

All commands in this section can be found in the file:

```
HDFS_commands.txt
```

Basic HDFS filesystem commands

One can regard HDFS as a regular file system, in fact many HDFS shell commands are inherited from the corresponding bash commands.

To run a command on an Hadoop filesystem use the prefix `hdfs dfs`, for instance use:

```
hdfs dfs -mkdir myDir
```

to create a new directory `myDir` on HDFS.

Note: One can use interchangeably `hadoop` or `hdfs dfs` when working on a HDFS file system. The command `hadoop` is more generic because it can be used not only on HDFS but also on other file systems that Hadoop supports (such as Local FS, WebHDFS, S3 FS, and others).

Basic HDFS filesystem commands

Basic HDFS filesystem commands that also exist in bash

`hdfs dfs -mkdir`	create a directory
`hdfs dfs -ls`	list files
`hdfs dfs -cp`	copy files
`hdfs dfs -cat`	print files
`hdfs dfs -tail`	output last part of a file
`hdfs dfs -rm`	remove files

Basic HDFS filesystem commands

Here's three basic commands that are specific to HDFS.

`hdfs dfs -put`	Copy single src, or multiple srcs from local file system to the destination file system
`hdfs dfs -get`	Copy files to the local file system
`hdfs dfs -usage`	get help on hadoop fs

Basic HDFS filesystem commands

To get more help on a specific `hdfs` command use: `hdfs -help <command>`

```
$ hdfs dfs -help tail
# -tail [-f] <file> :
#   Show the last 1KB of the file.

#   -f  Shows appended data as the file grows.
```

Some things to try

```
# create a new directory called "input" on HDFS
hdfs dfs -mkdir input
# copy local file wiki_1k_lines to input on HDFS
hdfs dfs -put wiki_1k_lines input/
# list contents of directory ("-h" = human)
hdfs dfs -ls -h input
# disk usage
hdfs dfs -du -h input
# get help on "du" command
hdfs dfs -help du
# remove directory
hdfs dfs -rm -r input
```

Some things to try

What is the size of the file `wiki_1k_lines`? What is its disk usage?

```
# show the size of wiki_1k_lines on the regular filesystem
ls -lh wiki_1k_lines
# show the size of wiki_1k_lines on HDFS
hdfs dfs -put wiki_1k_lines
hdfs dfs -ls -h wiki_1k_lines

# disk usage of wiki_1k_lines on the regular filesystem
du -h wiki_1k_lines
# disk usage of wiki_1k_lines on HDFS
hdfs dfs -du -h wiki_1k_lines
```

Disk usage on HDFS

The command `hdfs dfs -help du` will tell you that the output is of the form:

`size disk space consumed filename.`

You'll notice that the space on disk is larger than the file size (38.6MB versus 19.3MB):

```
hdfs dfs -du -h wiki_1k_lines
# 19.3 M  38.6 M  wiki_1k_lines
```

This is due to replication. You can check the replication factor using:

```
hdfs dfs -stat 'Block size: %o Blocks: %b Replication: %r'
   input/wiki_1k_lines
# Block size: 134217728 Blocks: 20250760 Replication: 2
```

Disk usage on HDFS

From the previous output:

```
Block size: 134217728 Blocks: 20250760 Replication: 2
```

we can see that the HDFS filesystem currently supports a replication factor of 2.

Note that the Hadoop block size is defined in terms of *mebibytes*, in fact 134217728 bytes corresponds to 128MiB and 134MB. One MiB is larger than a MB since one MiB is $1024^2 = 2^{20}$ bytes, while one MB is 10^6 bytes.

Outline/next

- Schedule
- What is Big Data?
- The Hadoop distributed computing architecture
- MapReduce
- HDFS hands-on exercises
- MapReduce hands-on
- The YARN resource manager
- MRjob
- Benchmarking I/O with testDFSio
- Concluding remarks

MapReduce hands-on

Where to find commands listing

For this part of the training you will need to activate the Hadoop module using the command:

```
module load Hadoop/2.6.0-cdh5.8.0-native
```

All commands in this section can be found in the file:

```
MapReduce_commands.txt
```

MapReduce streaming

The mapreduce streaming library allows to use any executable as mappers and reducers.

- read the input from stdin (line by line)
- emit the output to stdout

The documentation for streaming can be found in:
https://hadoop.apache.org/docs/stable/hadoop-streaming/HadoopStreaming.html

Locate the streaming library

First of all, we need to locate the streaming library on the system.

```
# find out where Hadoop is installed (variable $HADOOP_HOME)
echo $HADOOP_HOME
#/opt/apps/software/Hadoop/2.6.0-cdh5.8.0-native/share/
   hadoop/mapreduce

# find the streaming library
find /opt/apps/software/Hadoop/2.6.0-cdh5.8.0-native -name "
   hadoop-streaming*jar"
# . . .
#/opt/apps/software/Hadoop/2.6.0-cdh5.8.0-native/share/
   hadoop/tools/lib/hadoop-streaming-2.6.0-cdh5.8.0.jar

# save library in the variable $STREAMING
export STREAMING=/opt/apps/software/Hadoop/2.6.0-cdh5.8.0-
   native/share/hadoop/tools/lib/hadoop-streaming-2.6.0-
   cdh5.8.0.jar
```

Check input and output

We're going to use the file `wiki_1k_lines` (later you can experiment with a larger, for instance `wiki_1k_lines`.

```
# check that the output directory does not exist
hdfs dfs -rm -r output

# copy the file to HDFS
hdfs dfs -put wiki_1k_lines
```

Note: If you use a directory or file name that doesn't start with a slash ('/') then the directory or file is meant to be in your home directory (both in bash and on HDFS). A path that starts with a slash is called an *absolute path name*.

Run a simple MapReduce job

Using the streaming library, we can run the simplest MapReduce job.

```
# launch MapReduce job
hadoop jar $STREAMING \
        -input  wiki_1k_lines  \
        -output  output   \
        -mapper /bin/cat   \
        -reducer  '/bin/wc -l'
```

This job uses as a mapper the `cat` command, that does nothing else than echoimg the input. The reducer `wc -l` counts the lines in the given input. Note how we didn't need to write any code for the mapper and reducer

because the executables (`cat` and `wc`) are already there as par of any standard Linux distribution.

Run a simple MapReduce job

```
# launch MapReduce job
hadoop jar $STREAMING \
        -input  wiki_1k_lines  \
        -output  output   \
        -mapper /bin/cat   \
        -reducer  '/bin/wc -l'
```

If the job was successful, the output directory on HDFS (we called it `output`) should contain an empty file called `_SUCCESS`.

The file `part-*` contains the output of our job.

```
# check if job was successful (output should contain a file
   named _SUCCESS)
hdfs dfs -ls output
# check result
hdfs dfs -cat output/part-00000
```

Run a simple MapReduce job

Launch a MapReduce job with 4 mappers

```
hdfs dfs -rm -r output

# launch MapReduce job
hadoop jar $STREAMING \
        -D mapreduce.job.maps=4 \
        -input  wiki_1k_lines  \
        -output  output    \
        -mapper /bin/cat    \
        -reducer  '/bin/wc -l'

# check if job was successful (output should contain a file
    named _SUCCESS)
hdfs dfs -ls output
# check result
hdfs dfs -cat output/part-00000
```

Run a simple MapReduce job

Note how it is necessary to delete the `output` directory on HDFS (`hdfs dfs -rm -r output`) because according to the WORM principle, Hadoop will not delete or overwrite existing data!

The option `-D mapreduce.job.maps=4` right after the `jar` directive (in this example `-D mapreduce.job.maps=4`) allows to change MapReduce properties at runtime.

The list of all MapReduce options can be found in: mapred-default.xml

Note: this is the link to the last stable version, there might be some slight changes with respect to the version that is currently installed on the cluster.

Wordcount

We are now going to run a wordcount job using Python executables as mapper and reducer.

The mapper will be called `mapper.py` and the reducer `reducer.py`. Since these executables are not known to Hadoop, it is necessary to add them with the options

```
-files mapper.py -files reducer.py
```

Note: it is possible to have several mappers and reducers in one Mapreduce job, the output of each function is sent as input to the next one.

Define the mapper

```python
#!/bin/python3
import sys
for line in sys.stdin:
    words = line.strip().split()
    for word in words:
        print("{}\t{}".format(word,1))
```

Listing 1: mapper.py

Define the reducer

```
#!/bin/python3
import sys
current_word, current_count = None, 0
for line in sys.stdin:
    word, count = line.strip().split('\t', 1)
    try:
        count = int(count)
    except ValueError:
        continue
    if current_word == word:
        current_count += count
    else:
        if current_word:
            print("{}\t{}".format(current_word,
    current_count))
        current_count = count
        current_word = word
if current_word == word:
    print("{}\t{}".format(current_word, current_count))
```

Listing 2: reducer.py

Run the job

```
# upload file to HDFS
hdfs dfs -put data/wiki_1k_lines
# remove output directory
hdfs dfs -rm -r output

hadoop jar $STREAMING \
      -files mapper.py \
      -files reducer.py \
      -mapper mapper.py \
      -reducer reducer.py \
      -input wiki_1k_lines \
      -output output
```

Check results.

```
# check if job was successful (output should contain a file
   named _SUCCESS)
hdfs dfs -ls output
# check result
hdfs dfs -cat output/part-00000|head
```

Sorting the output after the job

The reducer just writes the list of words and their frequency in the order given by the mapper.

The output of the reducer is sorted by key (the word) because that's the ordering that the reducer becomes from the mapper. If we're interested in sorting the data by frequency, we can use the Unix `sort` command with the options `k2`, `n`, `r` meaning respectively "by field 2", "numeric", "reverse".

```
hdfs dfs -cat output/part-00000|sort -k2nr|head
```

The output should be something like:

```
the 193778
of  117170
and 89966
in  69186
. . .
```

Sorting with MapReduce

To sort by frequency using the mapreduce framework, we can employ a simple trick: create a mapper that interchanges words with their frequency values. Since by construction mappers sort their output by key, we get the desired sorting as a side-effect.

Create a script `swap_keyval.py`

```
#!/bin/python3
import sys
for line in sys.stdin:
    word, count = line.strip().split('\t')
    if int(count)>100:
        print("{}\t{}".format(count, word))
```

Listing 3: swap_keyval.py

Sorting with MapReduce

Run the new MapReduce job using output as input and writing results to a new directory output2.

```
# write the output to the directory output2
hdfs dfs -rm -r output2

hadoop jar $STREAMING \
      -files swap_keyval.py \
      -input output \
      -output output2 \
      -mapper swap_keyval.py
```

Looking at the output, one can see that it is sorted by frequency but alphabetically.

```
hdfs dfs -cat output2/part-00000|head
# 10021 his
# 1005  per
# 101 merely
# . . .
```

Using comparator classes for sorting

In general, we can determine how mappers are going to sort their output by configuring the comparator directive to use the special class `KeyFieldBasedComparator`:

```
-D mapreduce.job.output.key.comparator.class=\
    org.apache.hadoop.mapred.lib.KeyFieldBasedComparator
```

This class has some options similar to the Unix sort (`-n` to sort numerically, `-r` for reverse sorting, `-k pos1[,pos2]` for specifying fields to sort by). See documentation: KeyFieldBasedComparator.html

Using comparator classes for sorting

```
hdfs dfs -rm -r output2

comparator_class=org.apache.hadoop.mapred.lib.
   KeyFieldBasedComparator

hadoop jar $STREAMING \
     -D mapreduce.job.output.key.comparator.class=
   $comparator_class \
     -D mapreduce.partition.keycomparator.options=-nr \
     -files swap_keyval.py \
     -input output \
     -output output2 \
     -mapper swap_keyval.py
```

Using comparator classes for sorting

Now MapReduce has performed the desired sorting on the data.

```
hdfs dfs -cat output2/part-00000|head
193778  the
117170  of
89966 and
69186 in
. . .
```

Modify the Wordcount example

Try to modify the wordcount example:

- using executables in other programming languages
- adding a mapper that filters certain words
- using larger files

Run the MapReduce examples

The MapReduce distribution comes with some standard examples including source code.

To get a list of all available examples use:

```
hadoop jar \
  $HADOOP_HOME/hadoop-mapreduce-examples-2.6.0-cdh5.8.0.jar
```

Run the Wordcount example:

```
hadoop jar \
  $HADOOP_HOME/hadoop-mapreduce-examples-2.6.0-cdh5.8.0.jar
    wordcount wiki_1k_lines output3
```

Outline/next

- Schedule
- What is Big Data?
- The Hadoop distributed computing architecture
- MapReduce
- HDFS hands-on exercises
- MapReduce hands-on
- The YARN resource manager
- MRjob
- Benchmarking I/O with testDFSio
- Concluding remarks

The YARN resource manager

YARN: Yet Another Resource Negotiator

Hadoop jobs are usually managed by YARN (acronym for Yet Another Resource Negotiator), that is responsible for allocating resources and managing 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 architecture

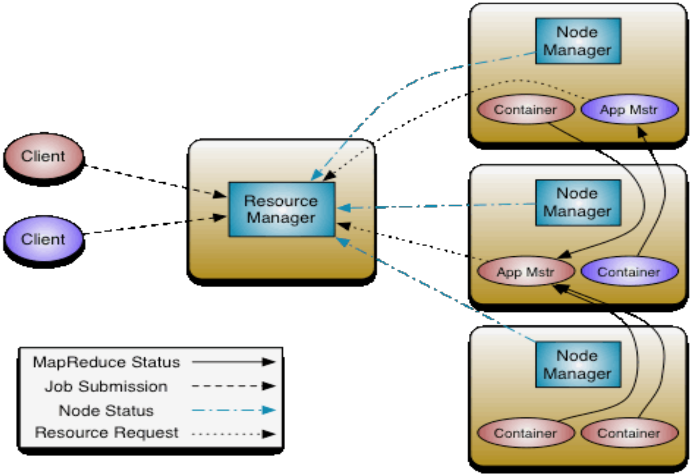


Image source: Apache Software Foundation

YARN architecture

Each job submitted to the Yarn is assigned:

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

YARN architecture

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**: allocates resources across all applications
  - **Applications Manager**: accepts job submissions, restart Application Masters on failure
- an **Application Master** is the local application manager, responsible for negotiating resources, monitoring status of the job, restarting failed tasks

Dynamic resource pools

Sharing computing resources fairly can be a big issue in multi-user environments.

YARN supports *dynamic resource pools* for scheduling applications.

A resource pool is a given configuration of resources to which a group of users is granted access. Whenever a group is not active, the resources are *preempted* and granted to other groups.

Groups are assigned a priority and resources are shared among groups according to these priority values.

Additionally, resource configurations can be scheduled for specific intervals of time.

YARN on SLURM

When running YARN on top of SLURM, it is not clear how to take advantage of the flexibility of YARN's dynamic resource pools to optimize resource utilization.

How to leverage the bses characteristics of job schedulers from both Big Data and HPC architectures in order to decrease latency is a subject of active study.

Outline/next

- Schedule
- What is Big Data?
- The Hadoop distributed computing architecture
- MapReduce
- HDFS hands-on exercises
- MapReduce hands-on
- The YARN resource manager
- MRjob
- Benchmarking I/O with testDFSio
- Concluding remarks

MRjob

MRjob

What is MRjob? It's a wrapper for MapReduce that allows to write MapReduce jobs in pure Python.

The library can be used for testing MapReduce as well as Spark jobs without the need of a Hadoop cluster.

Here's a quick-start tutorial:
https://mrjob.readthedocs.io/en/latest/index.html

A MRjob wordcount

```
from mrjob.job import MRJob

class MRWordFrequencyCount(MRJob):
    """
    A class to represent a Word Frequency Count mapreduce
    job
    """
    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()
```

Listing 4: word_count.py

A MRjob wordcount

Install `mrjob` in a virtual environment:

```
# install mrjob
python3 -m venv mypython
mypython/bin/pip install ipython mrjob
```

Run a basic mrjob wordcount

```
mypython/bin/python3 word_count.py data/wiki_1k_lines
```

Outline/next

- Schedule
- What is Big Data?
- The Hadoop distributed computing architecture
- MapReduce
- HDFS hands-on exercises
- MapReduce hands-on
- The YARN resource manager
- MRjob
- Benchmarking I/O with testDFSio
- Concluding remarks

Benchmarking I/O with testDFSio

TestDFSio

TestDFSio is a tool for measuring the performance of read and write operations on HDFS and can be used to measure performance, benchmark, or stress-test a Hadoop cluster.

TestDFSio uses MapReduce to write files to the HDFS filesystem spanning one mapper for file; the reducer is used to collect and summarize test data.

Find library location

Find the location of the `testDFSio` library, save it in the variable `testDFSio`:

```
find /opt/apps/software/Hadoop/2.6.0-cdh5.8.0-native \
     -name "hadoop-mapreduce-client-jobclient*tests.jar"
# /opt/apps/software/Hadoop/2.6.0-cdh5.8.0-native/share/
    hadoop/mapreduce/hadoop-mapreduce-client-jobclient
    -2.6.0-cdh5.8.0-tests.jar

export testDFSiojar=/opt/apps/software/Hadoop/2.6.0-cdh5
    .8.0-native/share/hadoop/mapreduce/hadoop-mapreduce-
    client-jobclient-2.6.0-cdh5.8.0-tests.jar
```

Options

Main options:

- `-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 generates exactly 1 map task per file, so it is a 1:1 mapping from files to map tasks.

Specify custom i/o directory

To avoid permission problems (you need to have read/write access to /benchmarks/TestDFSIO on HDFS).

By default TestHDFSio uses the HDFS directory /benchmarks to read and write, therefore it is recommended to run the tests as hdfs.

In case you want to run the tests as a user who has no write permissions on HDFS root folder /, you can specify an alternative directory with the option `-D` assigning a new value to `test.build.data`.

Running a write test

We are going to run a test with `nrFiles` files, each of size `fileSize`, using a custom output directory.

```
export myDir=/user/${USER}/benchmarks
export nrFiles=2
export fileSize=10MB
cd ~
hadoop jar $testDFSiojar TestDFSIO -D test.build.data=$myDir
    -write -nrFiles $nrFiles -fileSize $fileSize
```

Running a read test

We are going to run a test with `nrFiles` files, each of size `fileSize`, using a custom output directory.

```
export myDir=/user/${USER}/benchmarks
export nrFiles=2
export fileSize=10MB
cd ~
hadoop jar $testDFSiojar TestDFSIO -D test.build.data=$myDir
    -read -nrFiles $nrFiles -fileSize $fileSize
```

A sample test output

```
19/07/25 15:44:26 INFO fs.TestDFSIO: ----- TestDFSIO ----- :
    write
19/07/25 15:44:26 INFO fs.TestDFSIO:            Date & time:
    Thu Jul 25 15:44:26 CEST 2019
19/07/25 15:44:26 INFO fs.TestDFSIO:        Number of files:
    20
19/07/25 15:44:26 INFO fs.TestDFSIO: Total MBytes processed:
    204800.0
19/07/25 15:44:26 INFO fs.TestDFSIO:      Throughput mb/sec:
    39.38465325447428
19/07/25 15:44:26 INFO fs.TestDFSIO: Average IO rate mb/sec:
    39.59946060180664
19/07/25 15:44:26 INFO fs.TestDFSIO:  IO rate std deviation:
    3.0182194679812717
19/07/25 15:44:26 INFO fs.TestDFSIO:     Test exec time sec:
    292.66
```

How to interpret the results

The main measurements produced by the HDFSio test are:

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

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

Advanced test configuration

In addition to the default sequential file access, the mapper class for reading data can be configured to perform various types of random reads:

- `random read` always chooses a random position to read from (skipSize $= 0$)
- `backward read` reads file in reverse order (skipSize < 0)
- `skip-read` skips skipSize bytes after every read (skipSize > 0)

The `-compression` option allows to specify a *codec* for the input and output of data.

What is a codec

Codec is a portmanteau of coder and decoder and it designates any hardware or software device that is used to encode—most commonly also reducing the original size—and decode information. Hadoop provides classes that encapsulate compression and decompression algorithms.

These are all currently available Hadoop compression codecs:

Compression format	Hadoop CompressionCodec
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec
LZO	com.hadoop.compression.lzo.LzopCodec
LZ4	org.apache.hadoop.io.compress.Lz4Codec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

Concurrent versus overall throughput

Throughput or data transfer rate measures the amount of data read or written (expressed in Megabytes per second—MB/s) to the filesystem.

Throughput is one of the main performance measures used by disk manufacturers as knowing how fast data can be moved around in a disk is an important important factor to look at.

The listed throughput shows the average throughput among all the map tasks. To get an approximate overall throughput on the cluster you can divide the total MBytes by the test execution time in seconds.

Clean up

When done with the tests, clean up the temporary files generated by testDFSio.

```
export myDir=/user/${USER}/benchmarks

hadoop jar $testDFSiojar TestDFSIO \
     -D test.build.data=$myDir -clean
```

Outline/next

- Schedule
- What is Big Data?
- The Hadoop distributed computing architecture
- MapReduce
- HDFS hands-on exercises
- MapReduce hands-on
- The YARN resource manager
- MRjob
- Benchmarking I/O with testDFSio
- Concluding remarks

Concluding remarks

Big Data on VSC course

As part of the Vienna Scientific cluster training program, we offer a course "Big Data on VSC".

The first two editions ran in January and March of this year and the next edition will take place next spring.

Our Hadoop expertise comes from managing a Big Data cluster LBD (Little Big Data*) at the Vienna University of technology. The cluster is running since 2017 and is used for teaching and research.

(*) https://lbd.zserv.tuwien.ac.at/

Thanks

Thanks to:

- Janez Povh and Leon Kos for inviting me once again to hold this training on Hadoop.
- Dieter Kvasnicka my colleague and co-trainer for Big Data on VSC for his constant support