Introduction à Map-Reduce et Hadoop

Vincent Leroy, **Yves Denneulin,**
Francieli Zanon

Ce que nous allons voir ensemble...

- **Traitement par Batch grande échelle**
  - **paradigme du map/reduce**
  - **Hadoop et HDFS**
- Traitement de données en mémoire
  - Spark

- Cours + TP
  - plus orienté utilisation que performances

Sources

- Apache Hadoop
- Yahoo! Developer Network
- Hortonworks
- Cloudera
- Practical Problem Solving with Hadoop and Pig

« Big Data »

- Google, 2008
  - 20 PB/day
  - 180 GB/job (variable)
- Web index
  - 50B pages
  - 15PB
- Large Hadron Collider (LHC) @ CERN : produces 15PB/year
- Criteo : 150 PB

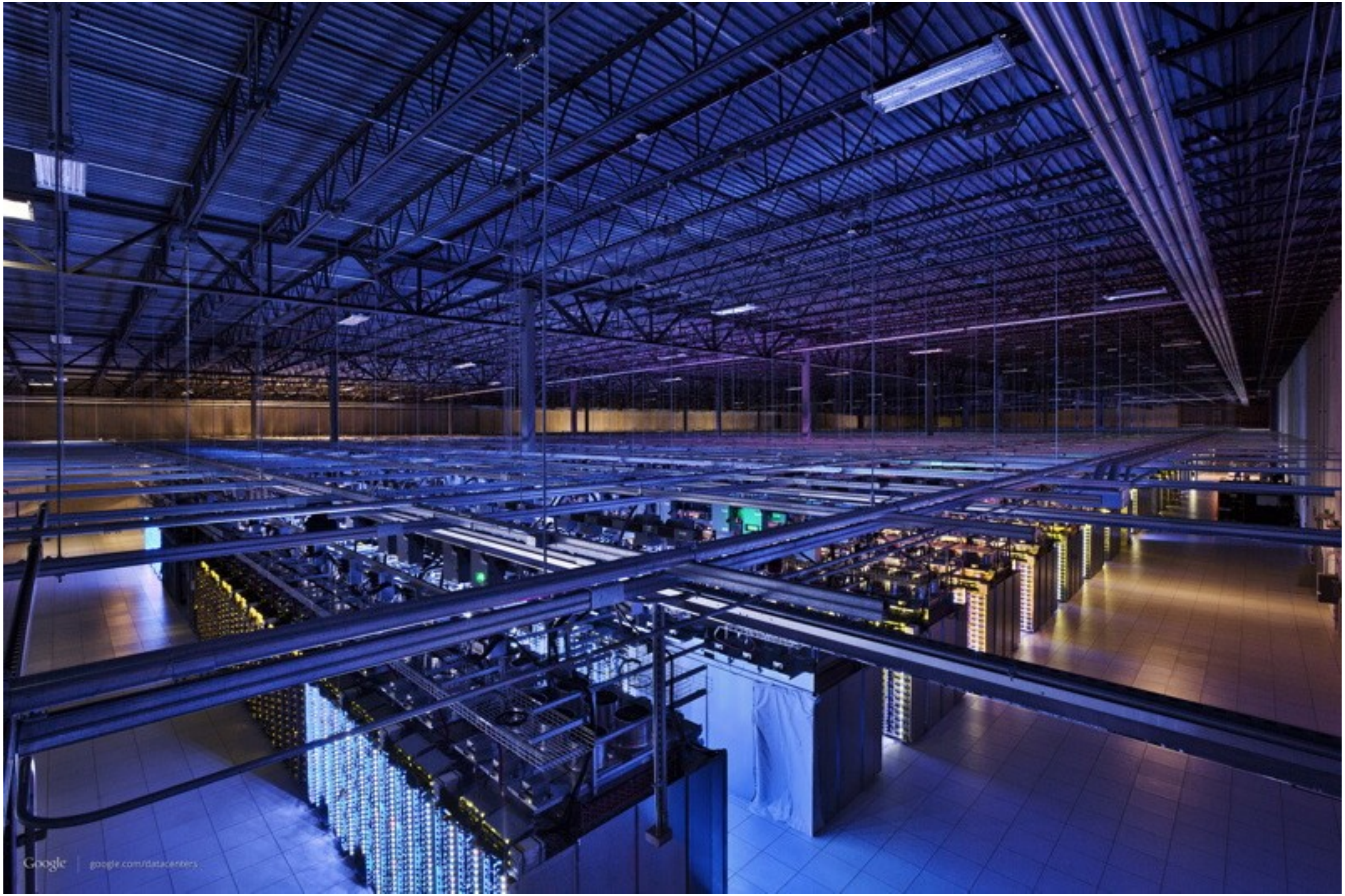
Capacity of a server

- RAM: 256 GB
- Hard drive capacity: 24TB
  - 50PB : ~2100, 150PB : ~6200
- Hard drive throughput: 100MB/s

Solution: Parallelism

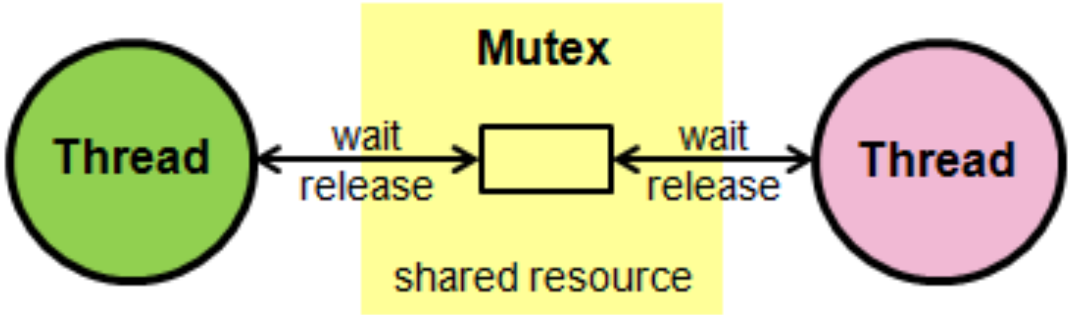
- 1 server
  - 8 disks
  - Read the Web: 230 days
- Hadoop Cluster @ Yahoo
  - 4000 servers
  - 8 disks/server
  - Read the Web in parallel: 1h20

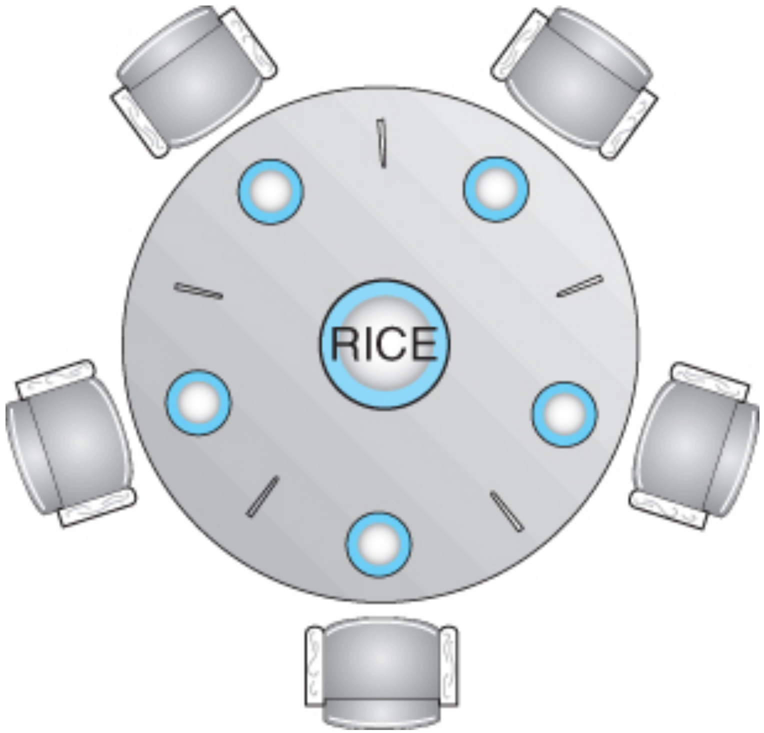
Data center Google



Pitfalls in parallelism

- Synchronization
  - Mutex, semaphores …
- Difficulties
  - Deadlocks
  - Optimization
  - Costly (experts)
  - Not reusable





Fault tolerance

- A server fails every few months
- 1000 servers …
  - MTBF (mean time between failures) < 1 day
- A big job may take several days
  - There will be failures, this is **normal**
  - Computations should finish within a reasonable time
    → You cannot start over in case of failures
- Checkpointing, replication
  - Hard to implement correctly

Big Data Platform

- Let everyone write programs for massive datasets
  - Encapsulate parallelism
    - Programming model
    - Deployment
  - Encapsulate fault tolerance
    - Detect and handle failures

→ Code once (experts), benefit to all

MAP-REDUCE MODEL

What are Map and Reduce?

- 2 simple functions inspired from functional programming
  - **Transformation: map**
    $map(f, [x_{1,} \ldots, x_n]) = [f(x_1), \ldots, f(x_n)]$
    Ex: map (*2, [1,2,3]) = [(*2 1),(*2 2),(*2 3)]
    = [2,4,6]
  - **Aggregation: reduce**
    $reduce(f, [x_{1,} \ldots, x_n]) = f(x_1, f(x_2, f(x_{3,} \ldots f(x_{n-1}, x_n)))))$
    Ex: reduce (+,[2,4,6]) = (+2 (+4 6))
    = 12

What are Map and Reduce?

- Generic
  - Take a function as a parameter
- Can be instantiated and combined to solve many different problems
  - map(toUpperCase, [“hello”, “data”]) = [“HELLO”, “DATA”]
  - reduce(max, [87, 12, 91])=91
- The developer provides the functions applied

Data as key/value pairs

- MapReduce does not manipulate atomic pieces of data
  – Everything is a (Key,Value) pair
  – Key and value can be of any type
    - Ex: (Hello, 17)
      – Key = Hello, type text
      – Value = 17 type int
- When initial data is not key/value, interpret it as key/value
  – Input text file becomes [(#line, line_content)…]

Map-Reduce on Key-Value pairs

- Map and Reduce adjusted to Key-Value pairs
  - In map, *f* is applied independently on every key/value pair
    f(key, value) → list(key, value)
  - In reduce, *f* is applied to **all** values associated with the same key
    f(key,list(value)) → list(key,value)
  - The types of keys and values taken as input does not have to be the same as the output

Example: Counting frequency of words

- Input : A file of 2 lines
  - 1, "a b c aa b c"
  - 2, "a bb cc a cc b"
- Output
  - a, 3
  - b, 3
  - c, 2
  - aa, 1
  - bb, 1
  - cc, 2

Word frequency: Mapper

- Map processes a portion (line) of text
  - Split words
  - For each word, count one occurrence
  - Key not used in this example (line number)
- map(Int lineNumber, Text line, Output output){

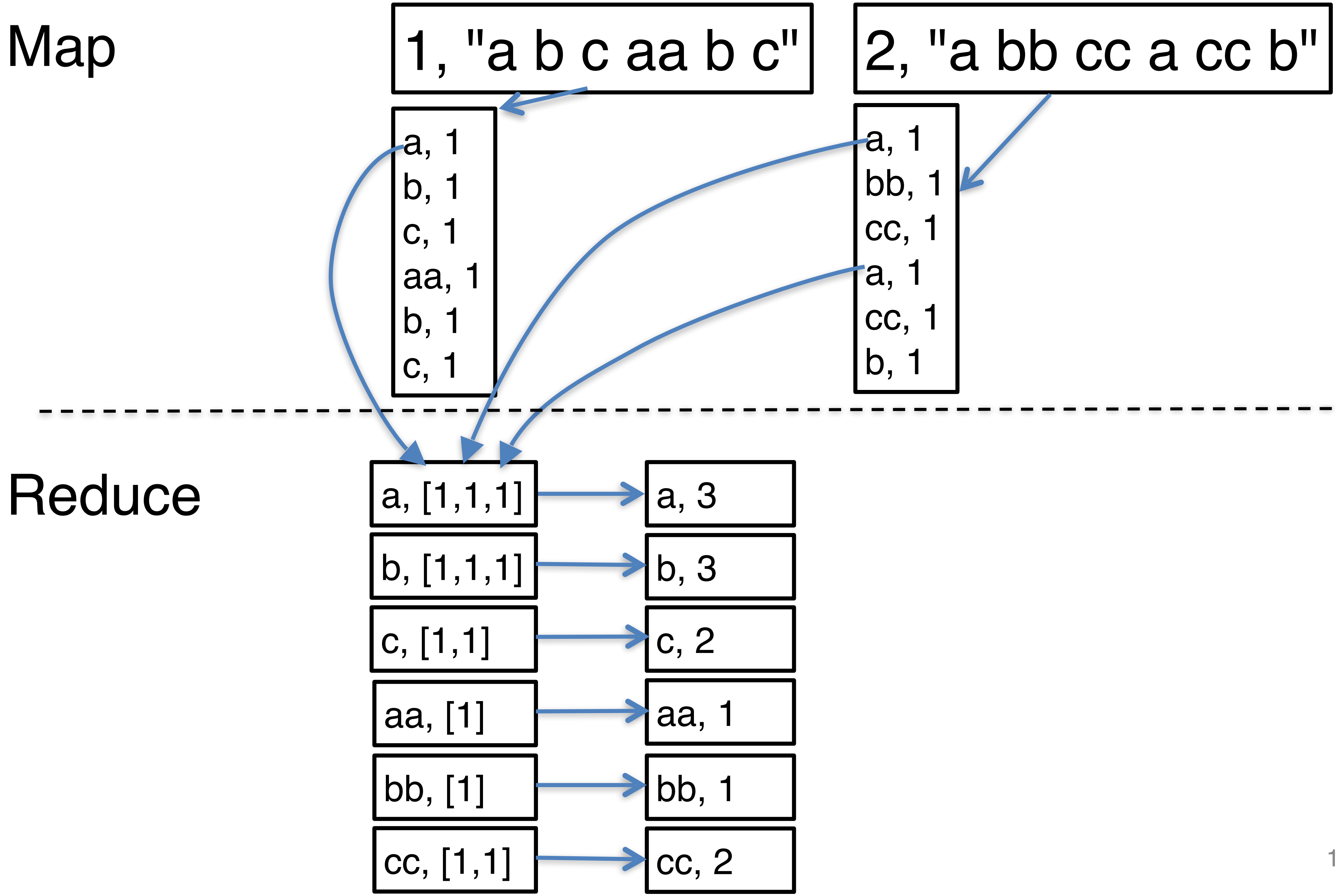
```
foreach word in line.split(space) {
    output.write(word, 1)
}
}
```

Word frequency: Reducer

- For each key, reduce processes all the corresponding values
  - Add number of occurrences
- reduce(String word, List<Int> occurrences, Output output){

```
      int count = 0
      foreach int occ in occurrences {
          count += occ
      }
      output.write(word,count)
}
```

Execution flow



How to build a Web index?

- Initial data: (URL, web_page_content)
- Goal: build inverted index

Grenoble
https://fr.wikipedia.org/wiki/Grenoble
http://www.grenoble.fr/
http://www.grenoble-tourisme.com/
http://wikitravel.org/en/Grenoble

UNIL
http://www.unil.ch/
https://fr.wikipedia.org/wiki/Universit%C3%A9_de_Lausanne
https://twitter.com/unil
http://www.formation-continue-unil-epfl.ch/

How to build a Web index?

- map(URL pageURL, Text pageContent, Output output){

```
    foreach word in pageContent.parse() {
        output.write(word, pageURL)
    }
}
```

How to build a Web index?

- ```
  reduce(Text word, List<URL> webPages,
  Output output){
        postingList = initPostingList()
        foreach url in webPages {
            postingList.add(url)
        }
        output.write(word, postingList)
   }
  ```
  ```

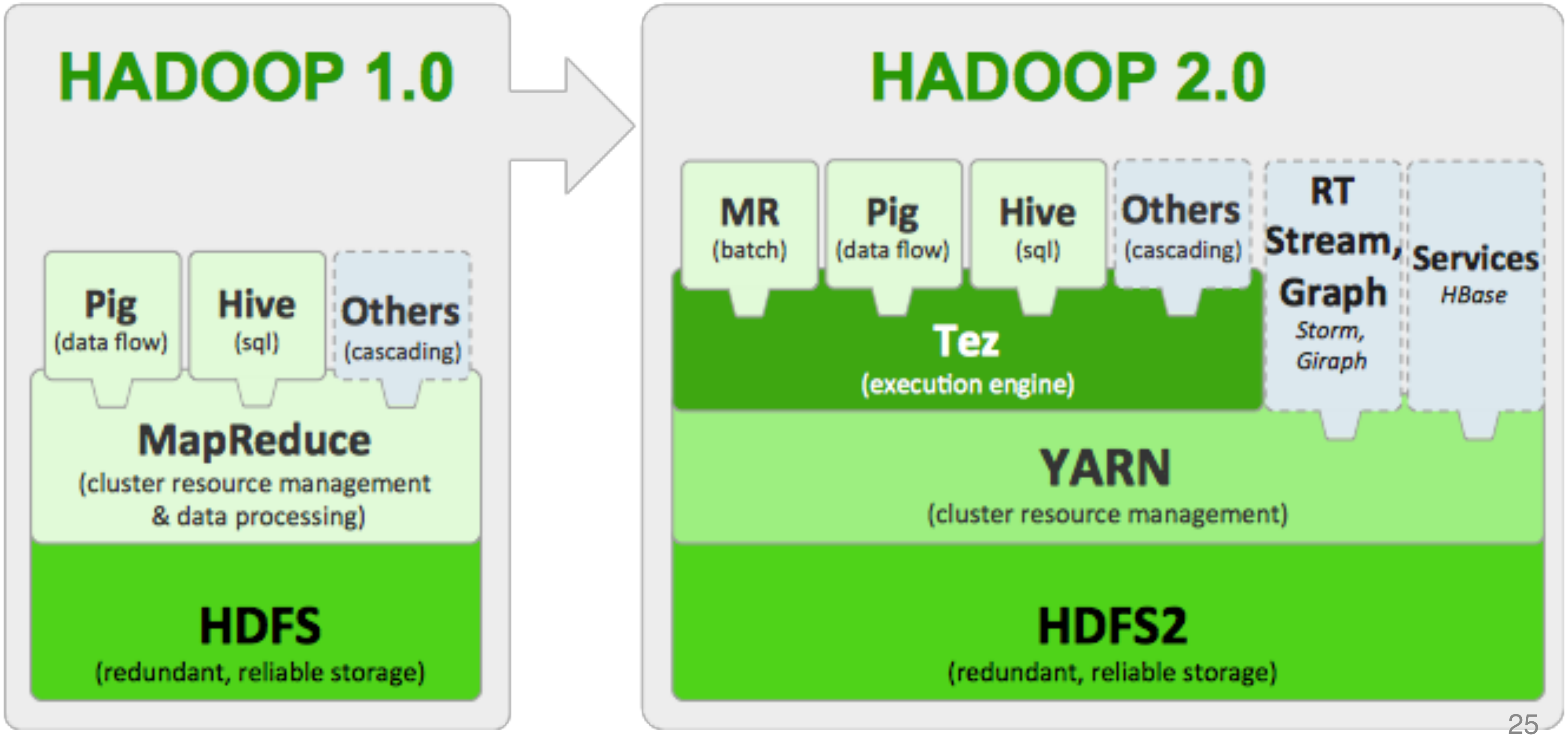
APACHE HADOOP: MAPREDUCE FRAMEWORK

Objective of Hadoop MapReduce

- Provide a simple and generic programming model: map and reduce
- Deploy execution automatically
- Provide fault tolerance
- Scale to thousands of machines
- Performance is important but not the priority
  - What's important is that jobs finish within reasonable time
  - If it's too slow, add servers!
    Kill It With Iron (KIWI principle)

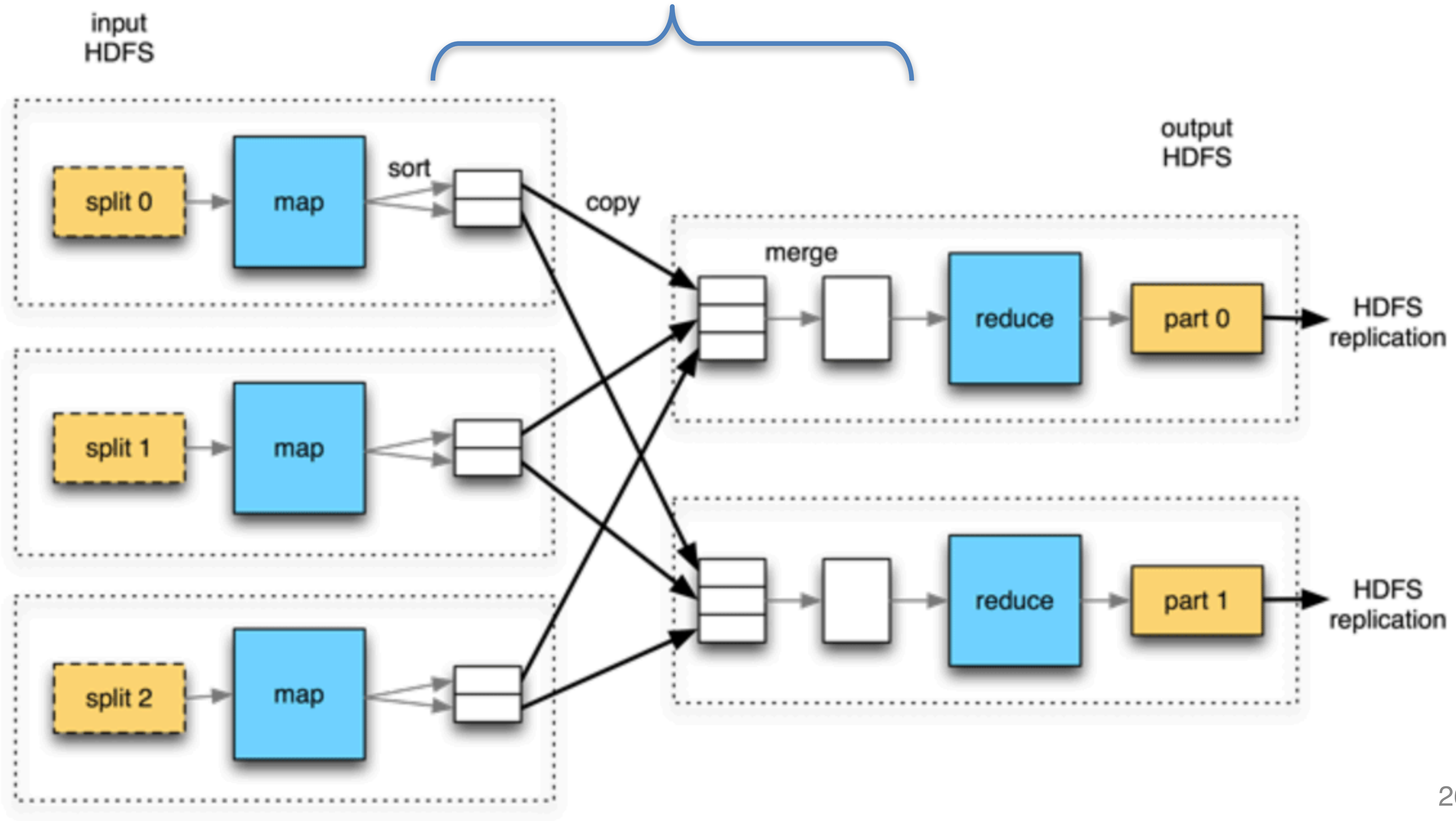
Architecture

- From a monolithic architecture to composable layers



Execution steps

Shuffle &Sort: group by key and transfer to reducer



Shuffle & Sort

- Barrier in the execution
  - All map tasks must complete before starting reduce
- Partitioner to assign keys to servers executing reduce
  - Ex: hash(key) % nbServers
  - Deal with load balancing

Hadoop MapReduce as a developer

- Provide the functions performed by Map and Reduce (Java, C++, Python)
  – Application dependent
- Defines the data types (keys / values)
  – If not standard (Text, IntWritable …)
  – Functions for serialization
- That's all.

Imports

```
import java.io.IOException ;
import java.util.* ;

import org.apache.hadoop.fs.Path ;
import org.apache.hadoop.io.IntWritable ;
import org.apache.hadoop.io.LongWritable ;
import org.apache.hadoop.io.Text ;
import org.apache.hadoop.mapreduce.Mapper ;
import org.apache.hadoop.mapreduce.Reducer ;
import org.apache.hadoop.mapreduce.JobContext ;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat ;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat
;
import org.apache.hadoop.mapreduce.Job ;
```

Do not use the old mapred API!

Mapper

```
 // input key type, input value type, output key type, output
value type
public class WordCountMapper extends Mapper<LongWritable,
Text, Text, IntWritable> {

    @Override
    protected void map(LongWritable key, Text value, Context
context) throws IOException, InterruptedException {
        for (String word : value.toString().split("\\s+")) {
            context.write(new Text(word), new IntWritable(1));
        }
    }
}
```

Reducer

```
// input key type, input value type, output key type, output
value type
public class WordCountReducer extends Reducer<Text,
IntWritable, Text, LongWritable> {

    @Override
    protected void reduce(Text key, Iterable<IntWritable>
values, Context context) throws IOException,
InterruptedException {
        long sum = 0;
        for (IntWritable value : values) {
            sum += value.get();
        }
        context.write(key, new LongWritable(sum));
    }
}
```

Main

```
public class WordCountMain {
    public static void main(String [] args) throws Exception {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf,
args).getRemainingArgs();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCountMain.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(LongWritable.class);
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Terminology

- MapReduce program = job
- Jobs are submitted to the JobTracker
- A job is divided in several tasks
  - A Map is a task
  - A Reduce is a task
- Tasks are monitored by TaskTrackers
  - A slow task is called a straggler

Job execution

- *$ hadoop jar wordcount.jar org.myorg.WordCount inputPath(HDFS) outputPath(HDFS)*
- Check parameters
  - Is there an output directory ?
  - Does it already exist ?
  - Is there an input directory ?
- Compute splits
- The job (MapReduce code), its configuration and splits are copied with a high replication
- Create an object to follow the progress as the tasks are created by the JobTracker
- For each split, create a Map
- Create default number of reducers

Tasktracker

- TaskTracker sends a periodic signal to the JobTracker
  - Show that the node still functions
  - Tell whether the TaskTracker is ready to accept a new task
- A TaskTracker is responsible for a node
  - Fixed number of slots for map tasks
  - Fixed number of slots for reduce tasks
  - Tasks can be from different jobs
- Each task runs on its own JVM
  - Prevents a task crash to crash the TaskTracker as well

Job Progress

- A Map task reports on its progress, i.e. amount of the split processed
- For a reduce task, 3 states
  - copy
  - sort
  - reduce
- Report sent to the TaskTracker
- Every 5 seconds, report forwarded to the JobTracker
- User can see the JobTracker state through Web interface

Progress

Logged in as: dr.who

All Applications

- Cluster
  - About
  - Nodes
  - Applications
    - NEW
    - SUBMITTED
    - ACCEPTED
    - RUNNING
    - FINISHED
    - FAILED
    - KILLED
  - Scheduler
- Tools

Switch Theme

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
1	0	0	1	0	0 KB	8 GB	0 KB	1	0	0	0	0

Show 20 entries Search:

ID	User	Name	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1348382062786_0001	hduser	word count	default	23-Sep-2012 12:06:36	23-Sep-2012 12:07:20	FINISHED	SUCCEEDED		History

Showing 1 to 1 of 1 entries

First Previous 1 Next Last

About Apache Hadoop

End of Job

- Output of each reducer written to a file
- Job tracker notifies the client and writes a report for the job

14/10/28 11:54:25 INFO mapreduce.Job: Job job_1413131666506_0070 completed successfully

Job Counters

Launched map tasks=392

Launched reduce tasks=88

Data-local map tasks=392

[...]

Map-Reduce Framework

Map input records=622976332

Map output records=622952022

Reduce input groups=54858244

Reduce input records=622952022

Reduce output records=546559709

[...]

Hadoop job_200709211549_0003 on localhost

User: hadoop
Job Name: streamjob34453.jar
Job File: /usr/local/hadoop-datastore/hadoop-hadoop/mapred/system/job_200709211549_0003/job.xml
Status: Succeeded
Started at : Fri Sep 21 16:07:10 CEST 2007
Finished at: Fri Sep 21 16:07:26 CEST 2007
Finished in: 16sec

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	3	0	0	3	0	0 / 0
reduce	100.00%	1	0	0	1	0	0 / 0

	Counter	Map	Reduce	Total
Job Counters	Launched map tasks	0	0	3
	Launched reduce tasks	0	0	1
	Data-local map tasks	0	0	3
Map-Reduce Framework	Map input records	77,637	0	77,637
	Map output records	103,909	0	103,909
	Map input bytes	3,659,910	0	3,659,910
	Map output bytes	1,083,767	0	1,083,767
	Reduce input groups	0	85,095	85,095
	Reduce input records	0	103,909	103,909
	Reduce output records	0	85,095	85,095

Change priority from NORMAL to: VERY_HIGH HIGH LOW VERY_LOW

Server failure during a job

- Bug in a task
  - task JVM crashes → TaskTracker JVM notified
  - task removed from its slot
- Task become unresponsive
  - timeout after 10 minutes
  - task removed from its slot
- Each task may be re-run up to N times (default 7) in case of crashes

HDFS : DISTRIBUTED FILE SYSTEM

Random vs Sequential disk access

- Example
  - DB 100M users
  - 100B/user
  - Alter 1% records
- Random access
  - Seek, read, write: 30mS
  - 1M users → 8h20
- Sequential access
  - Read ALL Write ALL
  - 2x 10GB @ 100MB/S → 3 minutes

→ It is often faster to read all and write all sequentially

Distributed File System (HDFS)

- Goal
  - Fault tolerance (redundancy)
  - Performance (parallel access)
- Large files
  - Sequential reads
  - Sequential writes
- “in place” data processing
  - Data is stored on the machines that process it
    - Better usage of machines (no dedicated *filer*)
    - Less network bottlenecks (better performance)

HDFS model

- Data organized in files and directories
  → mimics a standard file system
- Files divided in blocks (default: 64MB) spread on servers
- HDFS reports the data layout to the Map-Reduce framework
  → If possible, process data on the machines where it is already stored

Fault tolerance

- File blocks replicated (default: 3) to tolerate failures
- Placement according to different parameters
  – Power supply
  – Network equipment
  – Diverse servers to increase the probability of having a “close” copy
- Checksum of data to detect corrupter blocks (also available in modern file systems)

Master/Worker architecture

- A *master*, the NameNode
  - Manage the space of file names
  - Manages access rights
  - Supervise operations on files, blocks …
  - Supervise the *health* of the file system (failures, load balance…)
- Many (1000s) workers, the DataNodes
  - Store the data (blocks)
  - Perform read and write operations
  - Perform copies (replication, ordered by the NameNode)

NameNode

- Stores the metadata of each file and block (*inode*)
  - File name, directory, blocks associated, position of these blocks, number of replicas …
- Keeps all in main memory (RAM)
  - Limiting factor = number of files
  - 60M objects in 16GB

DataNode

- Manage and monitor the state of blocks stored on the host file system (often Linux)
- Directly accessed by the clients
  → data never transit through the NameNode
- Send *heartbeats* to the NameNode to show that the server has not failed
- Report to the NameNode if blocks are corrupted

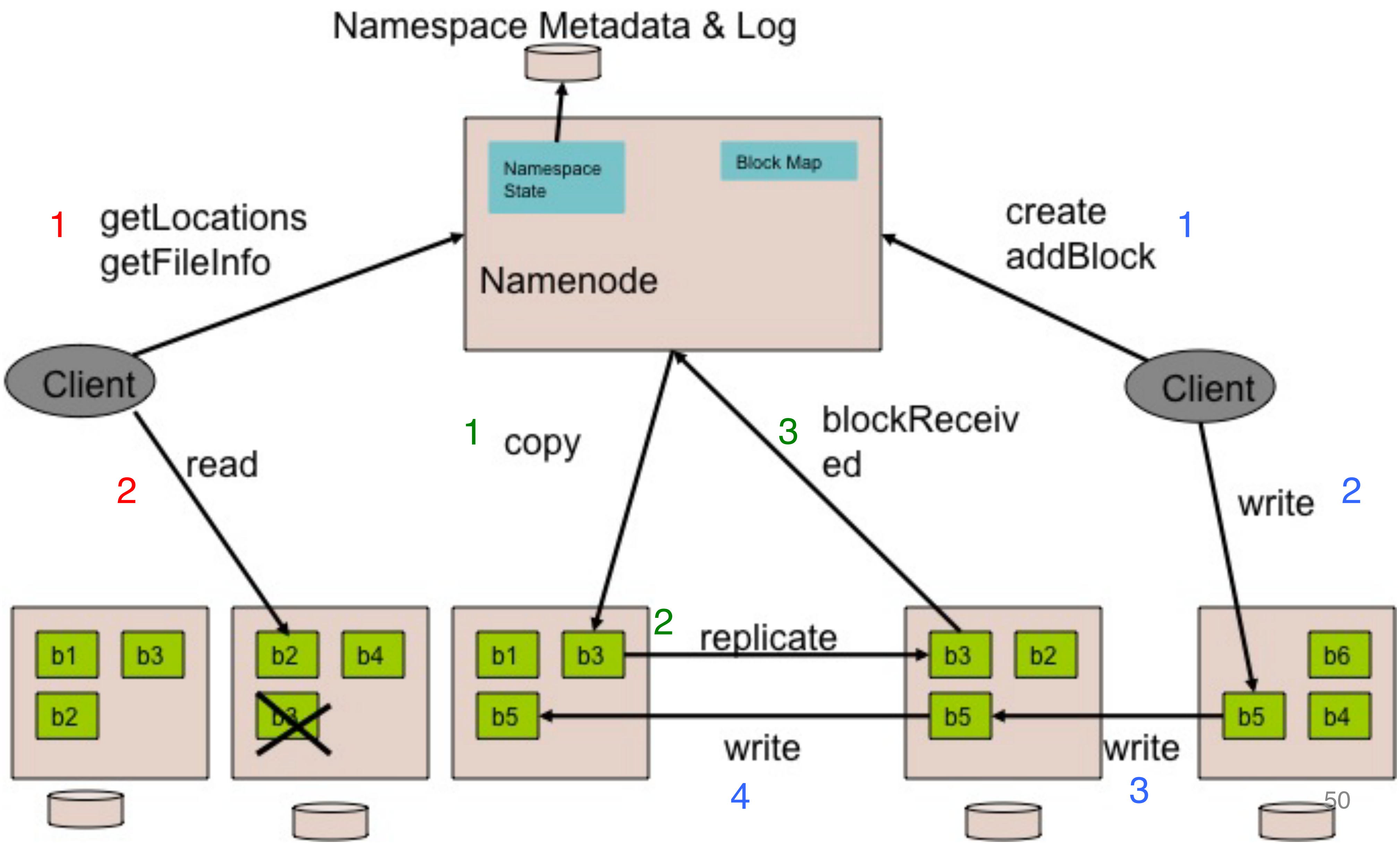
Writing a file

- The client sends a query to the NameNode to create a new file
- The NameNode checks
  - Client authorizations
  - File system conflicts (existing file …)
- NameNode choses DataNodes to store file and replicas
  - DataNodes “pipelined”
- Blocks are allocated on these DataNodes
- Stream of data sent to the first DataNode of the pipeline
- Each DataNode forwards the data received to the next DataNode in the pipeline

Reading a file

- Client sends a request to the NameNode to read a file
- NameNode checks the file exists and builds a list of DataNodes containing the first blocks
- For each block, NameNode sends the address of the DataNodes hosting them
  - List ordered wrt. Proximity to the client
- Client connects to the closest DataNode containing the 1st block of the file
- Block read ends:
  - Close connection to the DataNode
  - New connection to the DataNode containing the next block
- When all blocks are read:
  - Query the NameNode to retrieve the following blocks

HDFS Structure



HDFS commands (directories)

- Create directory dir
  $ hadoop dfs -mkdir /dir
- List HDFS content
  $ hadoop dfs -ls
- Remove directory dir
  $ hadoop dfs -rmdir /dir

HDFS commands (files)

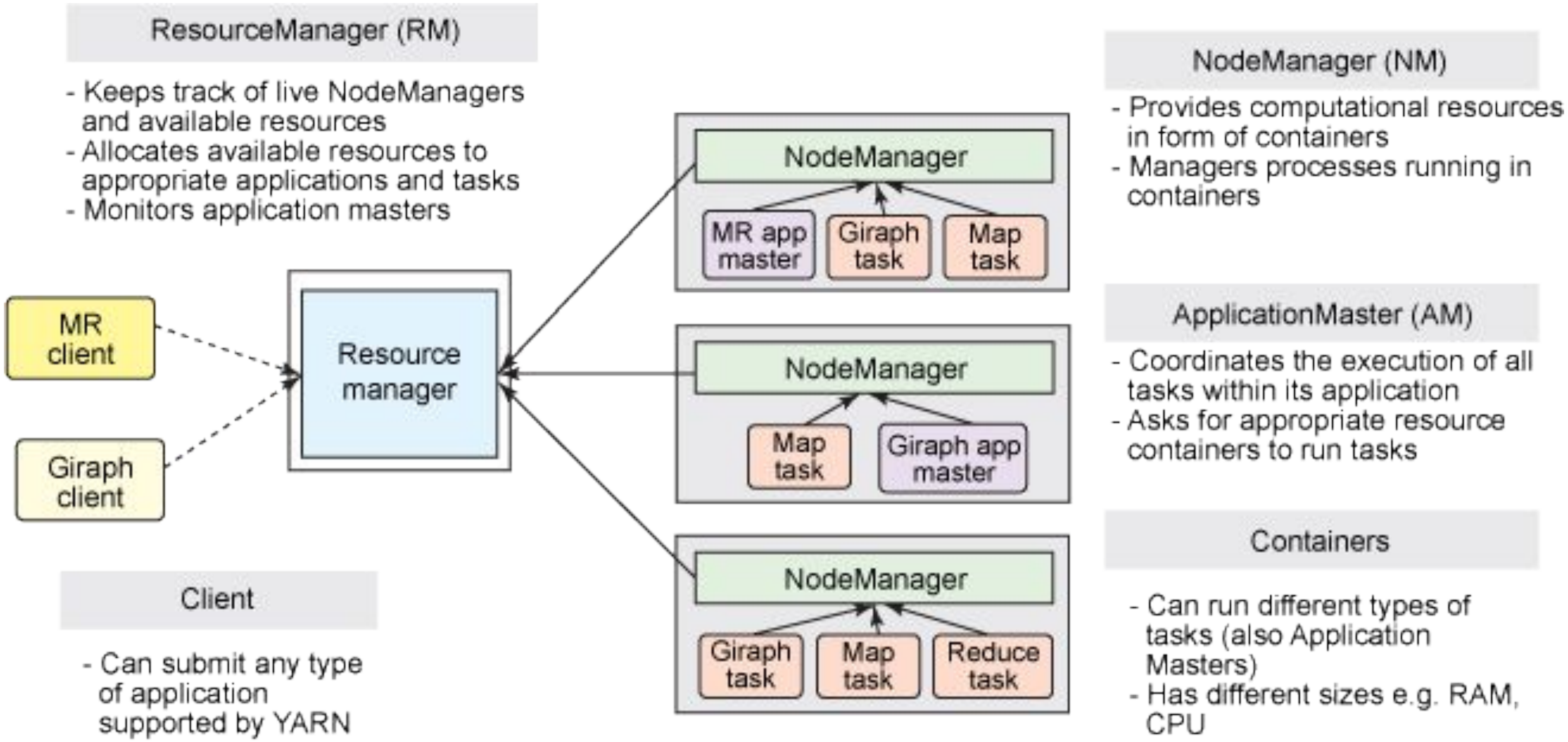
- Copy local file toto.txt to HDFS dir/
  $ hadoop dfs -put toto.txt dir/toto.txt
- Copy HDFS file to local disk
  $ hadoop dfs -get dir/toto.txt ./
- Read file /dir/toto.txt
  $ hadoop dfs -cat /dir/toto.txt
- Remove file /dir/toto.txt
  $ hadoop dfs -rm /dir/toto.txt

Quelques compléments

Yarn

- The cluster resource manager: dynamically allocates resources to jobs
- Multiple engines (in addition to MapReduce) run in parallel on the cluster
- Hierarchical architecture for scalability

Yarn architecture



Source: https://www.ibm.com/developerworks/library/bd-yarn-intro/index.html

That is costly!

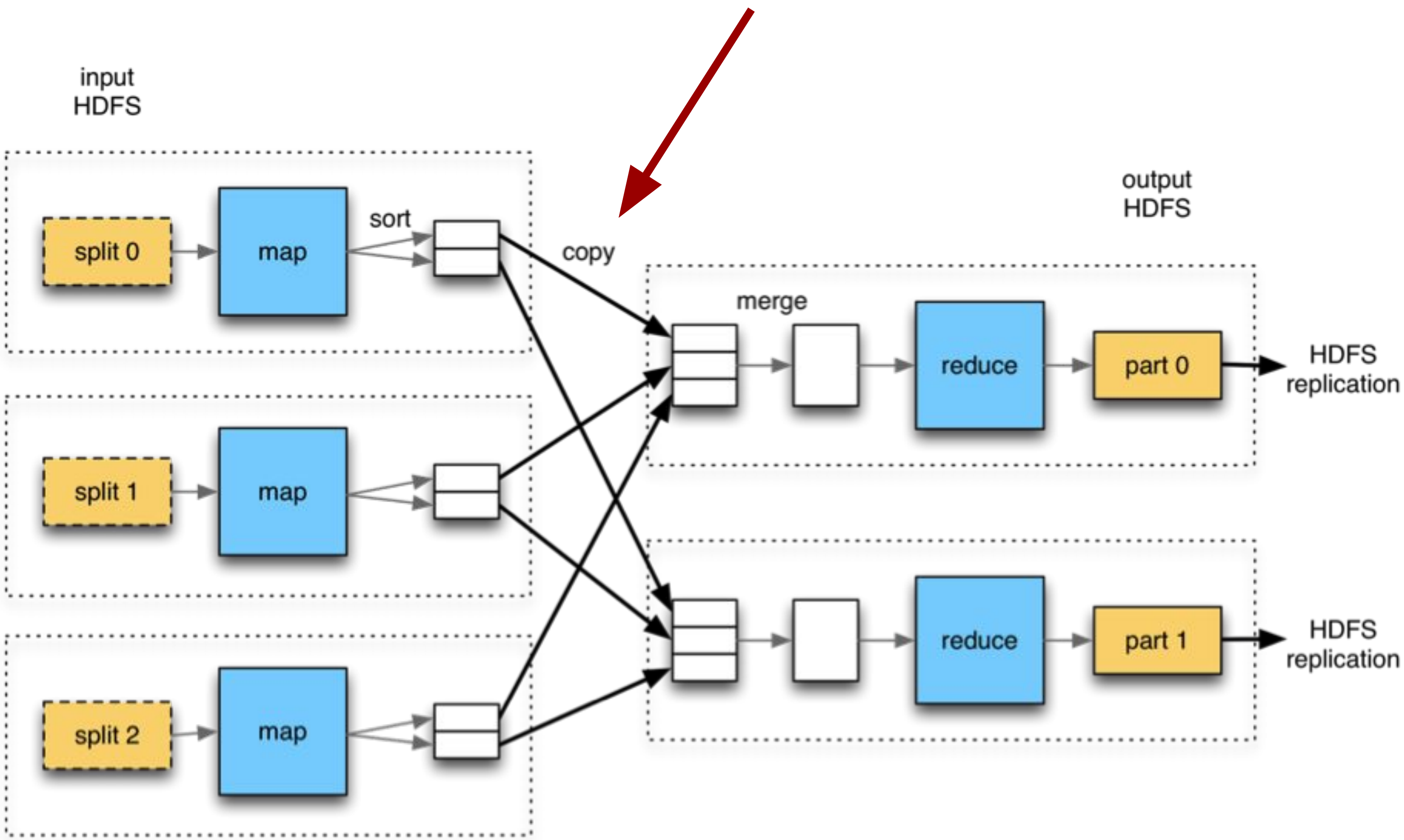
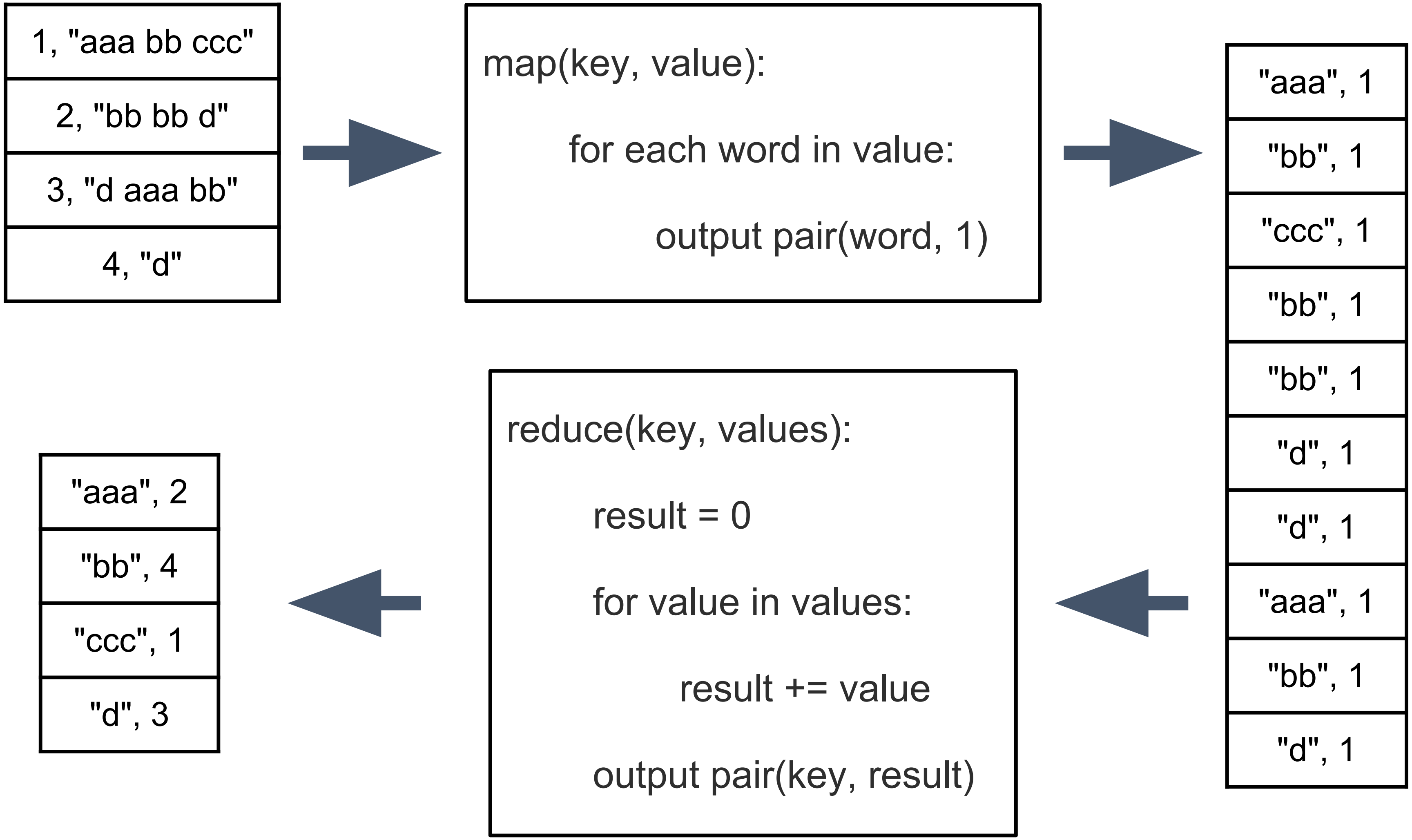
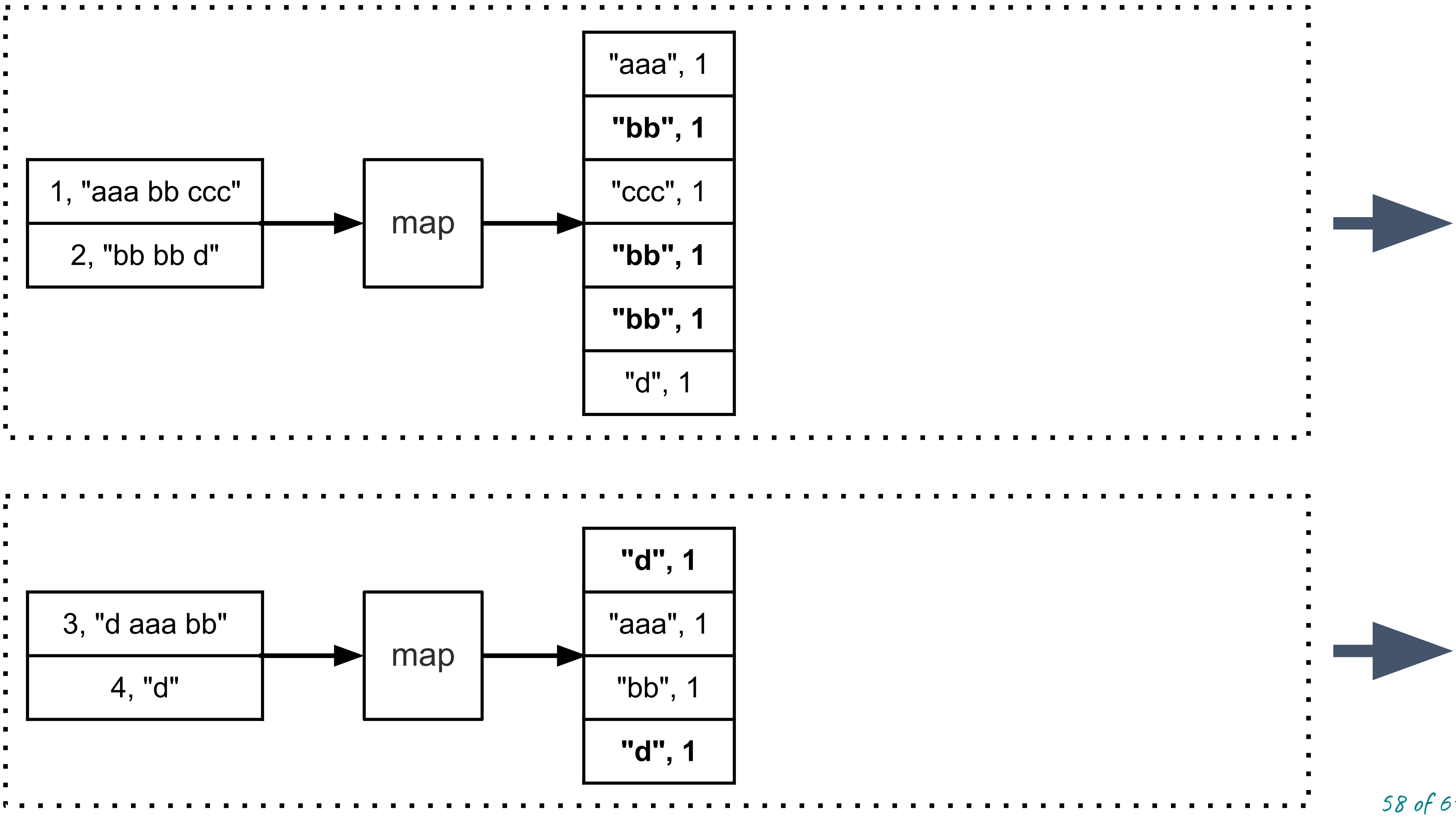


Figure from https://www.supinfo.com/articles/single/2807-introduction-to-the-mapreduce-life-cycle

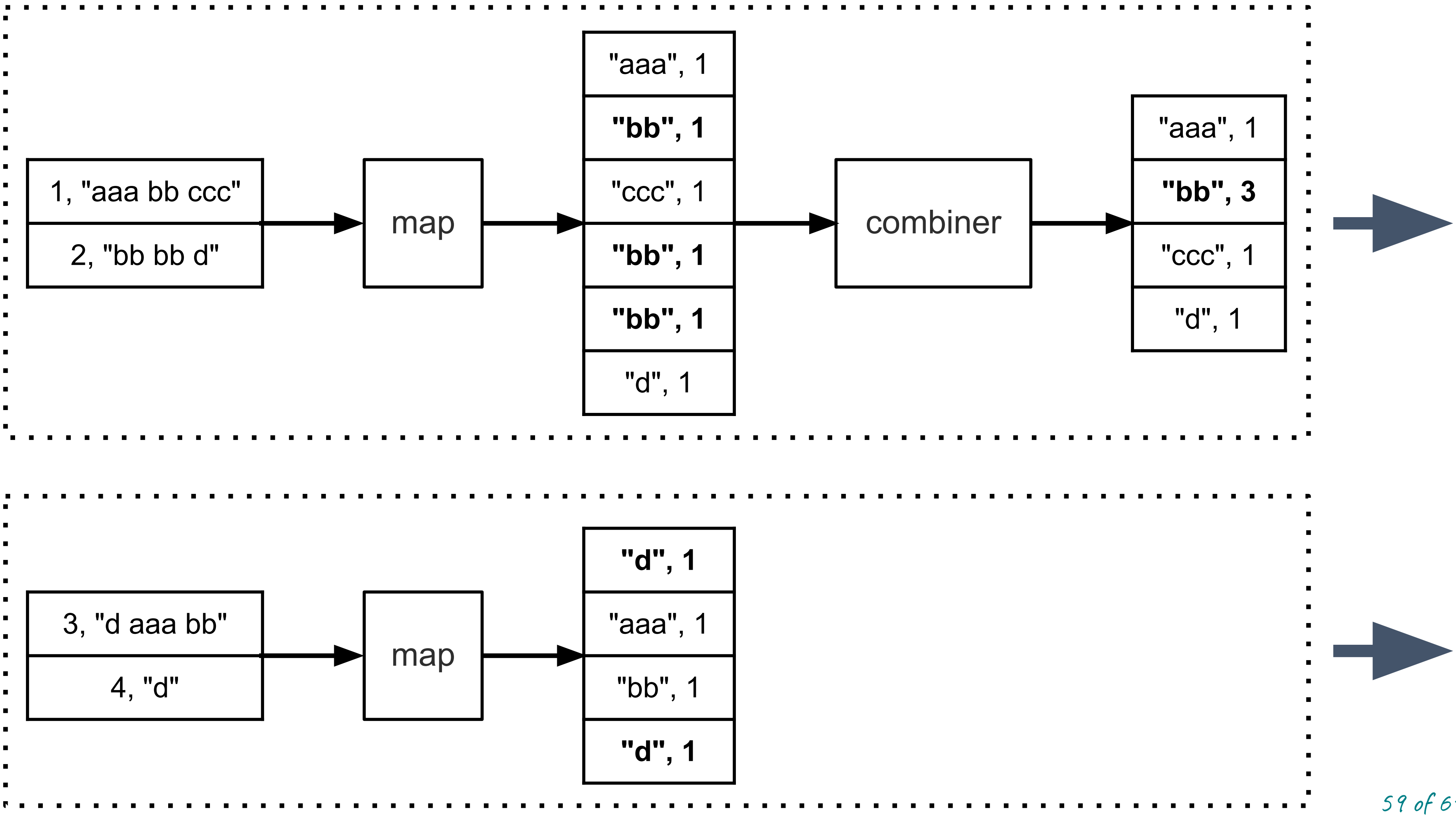
55 of 61

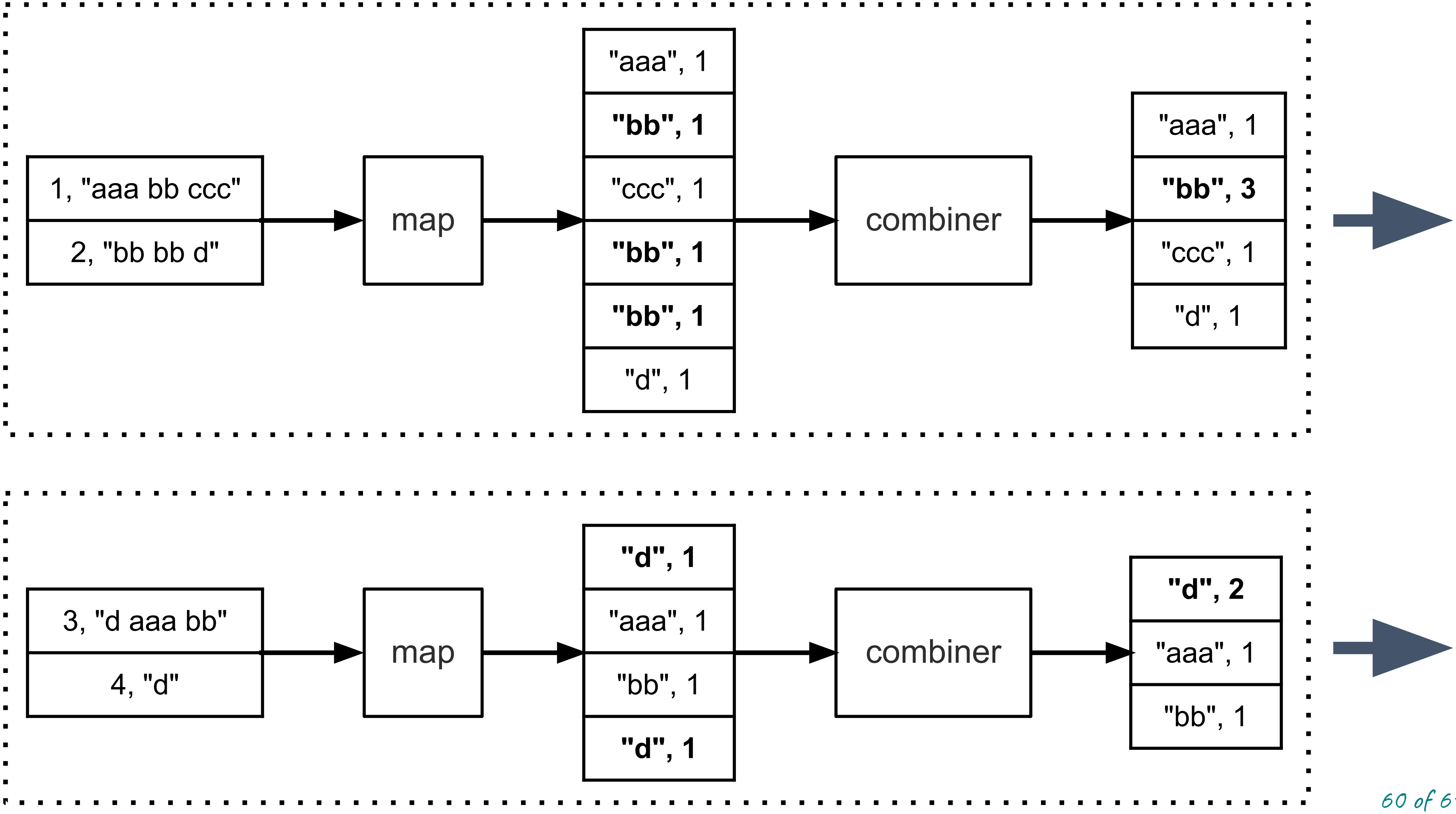

1, "aaa bb ccc"
2, "bb bb d"
3, "d aaa bb"
4, "d"
map(key, value):
for each word in value:
output pair(word, 1)
"aaa", 1
"bb", 1
"ccc", 1
"bb", 1
"bb", 1
"d", 1
"d", 1
"aaa", 1
"bb", 1
"d", 1
reduce(key, values):
result = 0
for value in values:
result += value
output pair(key, result)
"aaa", 2
"bb", 4
"ccc", 1
"d", 3

56 of 61


1, "aaa bb ccc"
2, "bb bb d"
map
"aaa", 1
"bb", 1
"ccc", 1
"bb", 1
"bb", 1
"d", 1
3, "d aaa bb"
4, "d"
map
"d", 1
"aaa", 1
"bb", 1
"d", 1

58 of 61


1, "aaa bb ccc"
2, "bb bb d"
map
"aaa", 1
"bb", 1
"ccc", 1
"bb", 1
"bb", 1
"d", 1
combiner
"aaa", 1
"bb", 3
"ccc", 1
"d", 1
3, "d aaa bb"
4, "d"
map
"d", 1
"aaa", 1
"bb", 1
"d", 1



1, "aaa bb ccc"
2, "bb bb d"
map
"aaa", 1
"bb", 1
"ccc", 1
"bb", 1
"bb", 1
"d", 1
combiner
"aaa", 1
"bb", 3
"ccc", 1
"d", 1
3, "d aaa bb"
4, "d"
map
"d", 1
"aaa", 1
"bb", 1
"d", 1
combiner
"d", 2
"aaa", 1
"bb", 1

60 of 61

Combiner

- Same API as reduce (key, List<value>)
  - Not the same contract!
    For one key, you get SOME values
- Often the same aggregation as reduce
  - E.g. WordCount
- Different when using global properties
  - E.g. Keep words present at least 5 times

Le TP

- Prise en main de hadoop et map/reduce
  - Java ou Python
- Deux parties
  - écriture et exécution de wordcount
  - analyse de données Flickr

- Exécution sur un cluster Hadoop maison
  - accessible depuis le réseau Ensimag

- Python :
  - 2 programmes à écrire map.py reduce.py
  - entrée mapper -> ligne de fichiers, sortie mapper : clé \t valeur
  - entrée reducer : liste de clé \t valeur triée par valeur de clé