BIG DATA, MAPREDUCE & HADOOP

LARGE SCALE DISTRIBUTED SYSTEMS

By Jean-Pierre Lozi

A tutorial for the LSDS class

Université Nice Sophia Antipolis

OBJECTIVES OF THIS LAB SESSION

- The LSDS class has been mostly theoretical so far
- The objective of this lab session is to get hands-on experience with Hadoop
- I'll give you a short presentation (<< 1h), after that: exercises
- **This is all just for fun, no grades!**

WHAT IS BIG DATA?

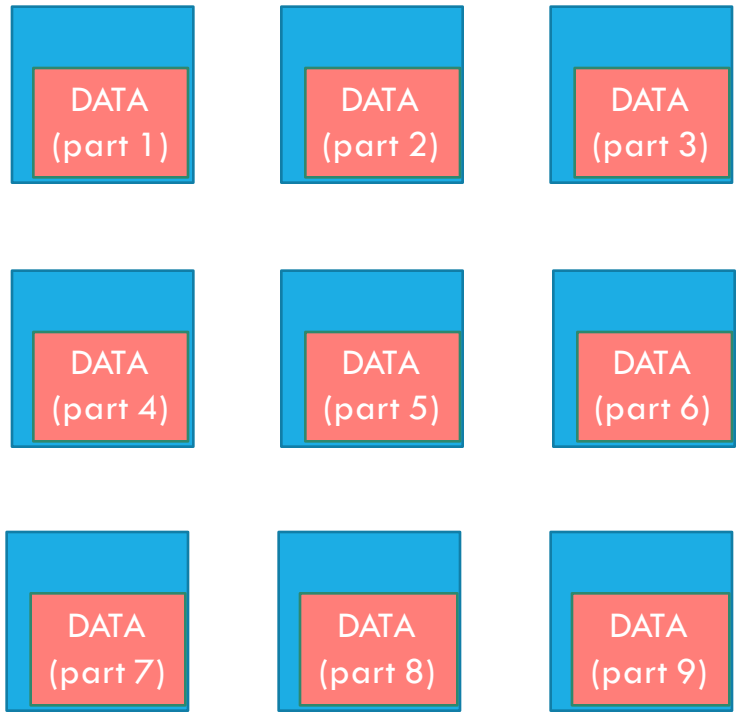
- Companies & organisations gather *lots* of data nowadays
- ***They're able to store it because storage has become very cheap!***
  - The **New York Stock Exchange (NYSE)** generates **1 Terabyte** of data each day
  - **Facebook** stores ~250 billion pictures from users ⇒ **several Petabytes** of data!
  - The **Large Hadron Collider (LHC)** generates **15 million petabytes** of data!
- **Numbers from 2014, so probably even more now !**

WHAT IS BIG DATA?

- **This data comes from various sources.**
  - Re: Facebook, it comes users (as is often the case in the social web),
  - Re: the LHC, it comes from machines
    - Particle accelerator, but it can come from other "machines", such as sensor networks (e.g., monitoring temperatures in your server farms all across the world, taxi companies who want to know where all their cars are at any moment, transactions, log files...)
- **This data is often not very well structured.**
  - Images, text files, comments, health data (prescriptions...), etc.
- **How do you store and process this data?**

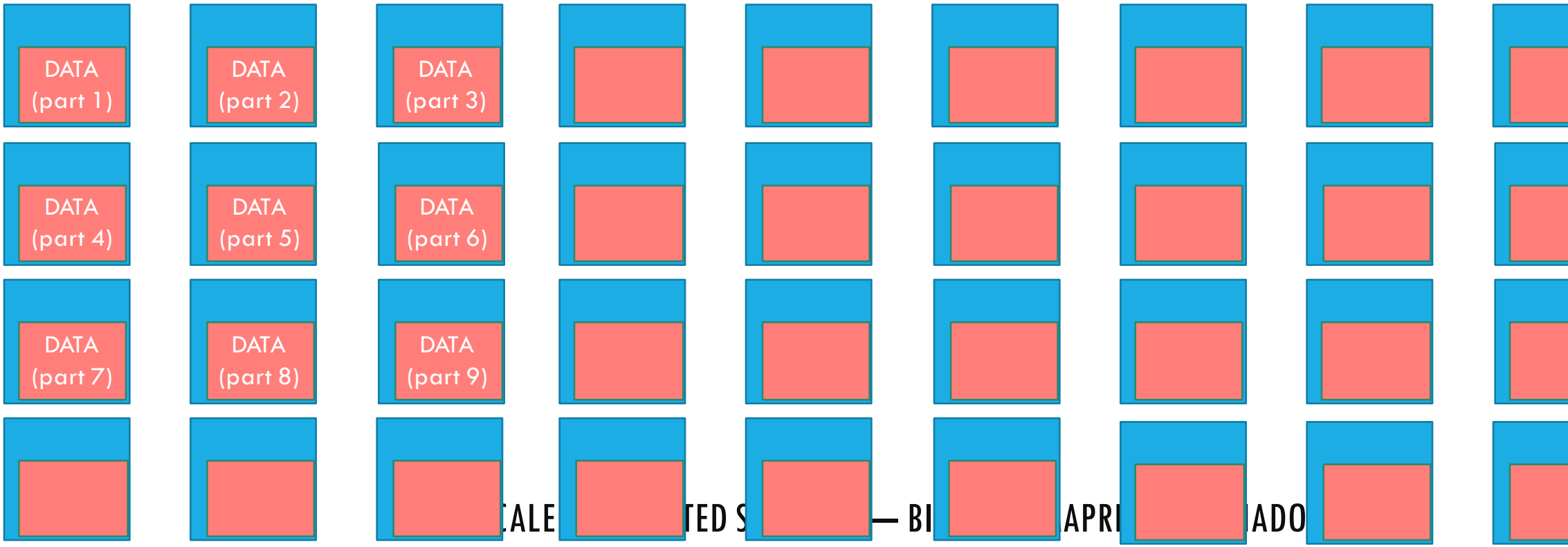
HOW DO YOU STORE THIS DATA?

- You need many machines that will store a small part of the data
  - ***« Oh, that was easy ! »***



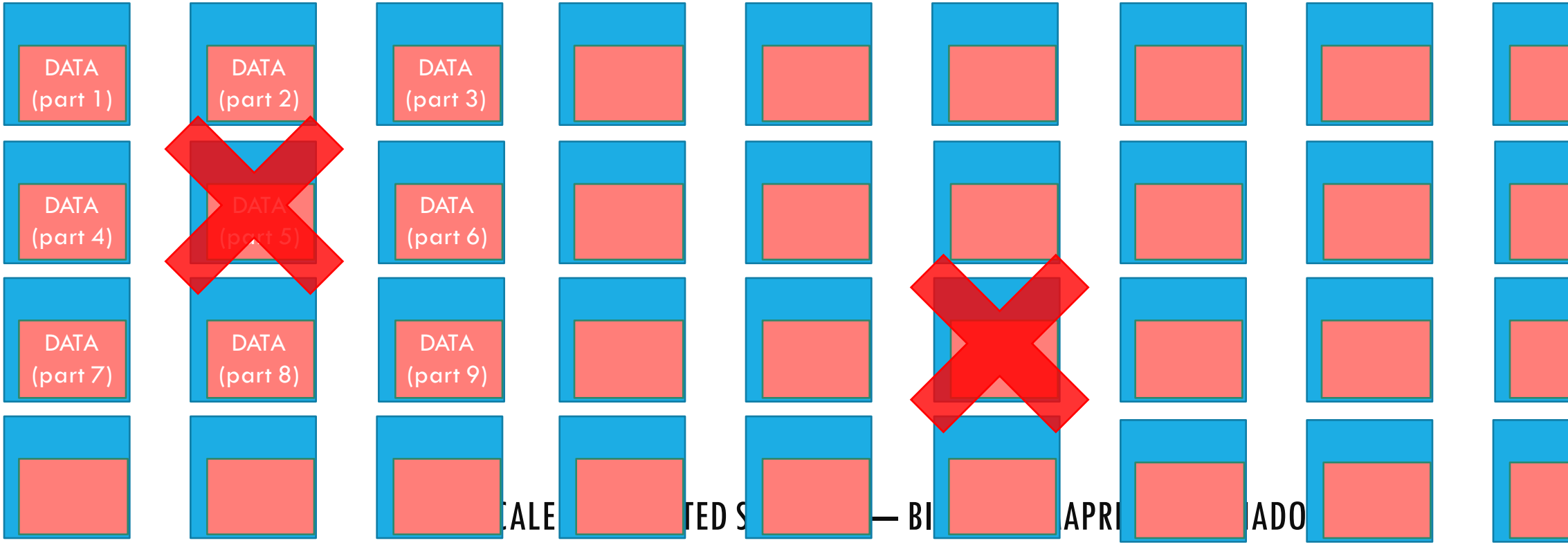
HOW DO YOU STORE THIS DATA?

- But actually, you really need *lots* of machines
  - **E.g., Apple uses their own solar power plant to power their iCloud server farm!**



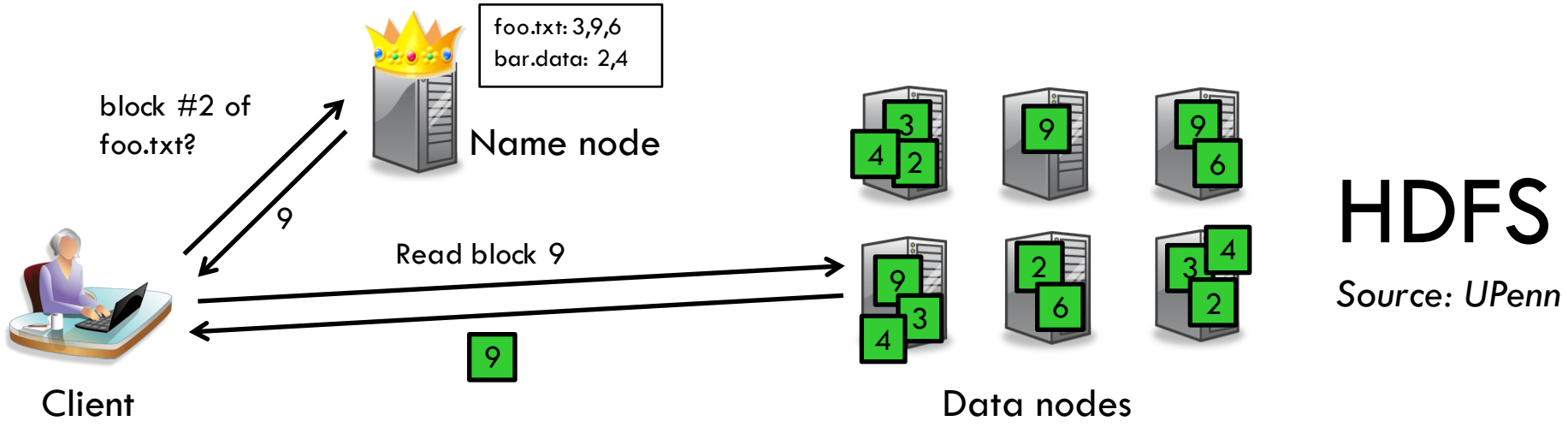
HOW DO YOU STORE THIS DATA?

- **Problem: with that many machines, some are bound to crash!**
  - You can't just naïvely partition the data, or you'll lose some!



HOW DO YOU STORE THIS DATA?

- So you need some replication. You can't handle it manually, of course.
- So you use a Distributed File System (DFS) that does the job for you!
- **In Hadoop, this filesystem is called HDFS (Hadoop Distributed File System)**



HOW DO YOU PROCESS THIS DATA?

- Now we know how to store the data, but how do we process it?
- Historically, we've been using databases for this.
- **It doesn't work anymore!**
- **First, because as we saw earlier : lack of structure!**
  - Images, comments, log files, prescriptions, ...
  - You can put that in a database, with a fixed structure, tables, relations, etc.
- **Second, databases don't scale very well...**
  - Try doubling the number of nodes with a (distributed) database...
  - You won't be twice as fast (far from it)

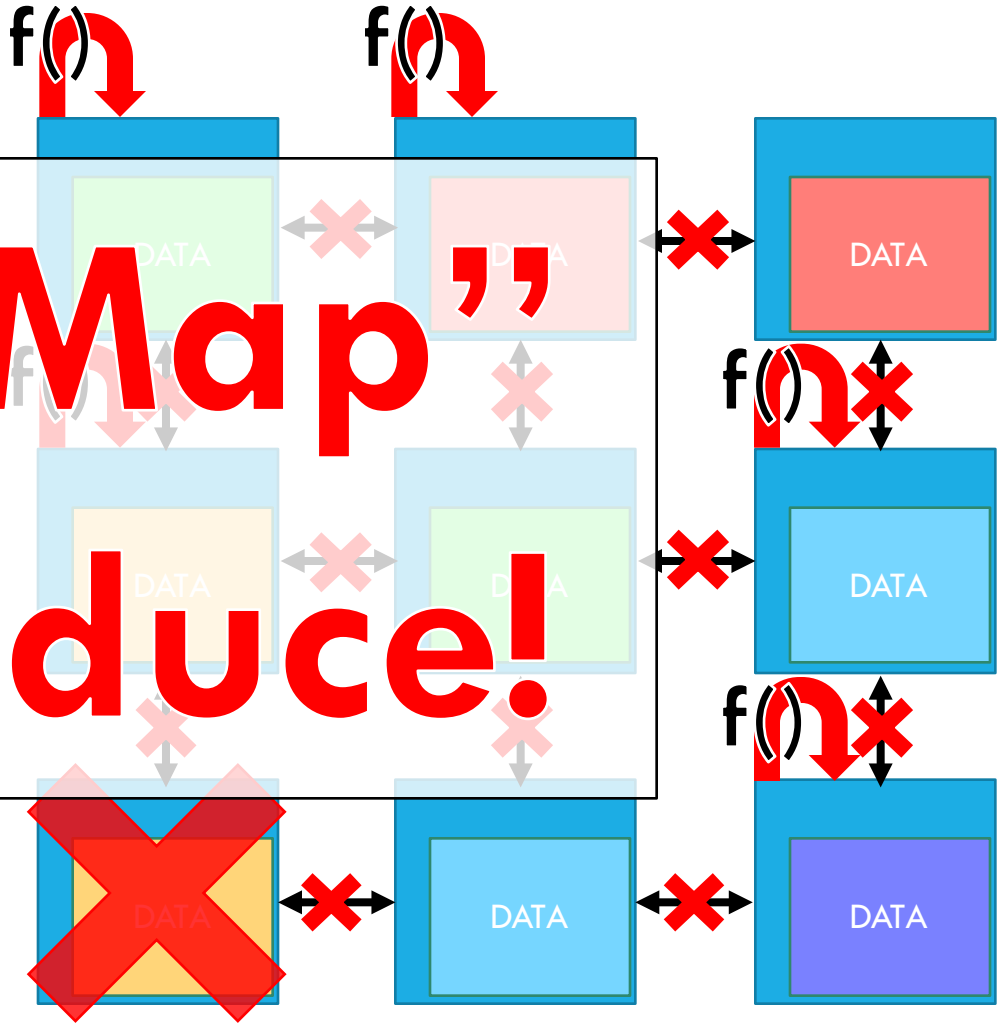
HOW DO YOU PROCESS THIS DATA?

- So what's the alternative?
- **Google invented MapReduce to make the indexer for their search engine scale!**
- **Idea of MapReduce :**
  - You write a function that you're going to run as a batch process on all of your data
  - And you want to get one result (which can be large)
- **MapReduce is really good at doing this efficiently!**
- **Different use case from databases** that are better at accessing small bits of your data all the time frequently, instead of all of your data once in a while in a batch process!

HOW DO YOU PROCESS THIS DATA?

- How does MapReduce manage to be so efficient at what it does?
- **A very old idea: execute things on nodes as much as possible and to avoid transfers between nodes as much as possible !**
- **MapReduce first runs a function f() on all data**
- Of course, if a node crashes during computation, you're only going to run the function on the nodes only (just ensure it's run on all of the data)
- And if a node is dead, you'll make sure you run the function on another node that has the same data
- **All of this is done automatically!**

f() is the "Map" of MapReduce!

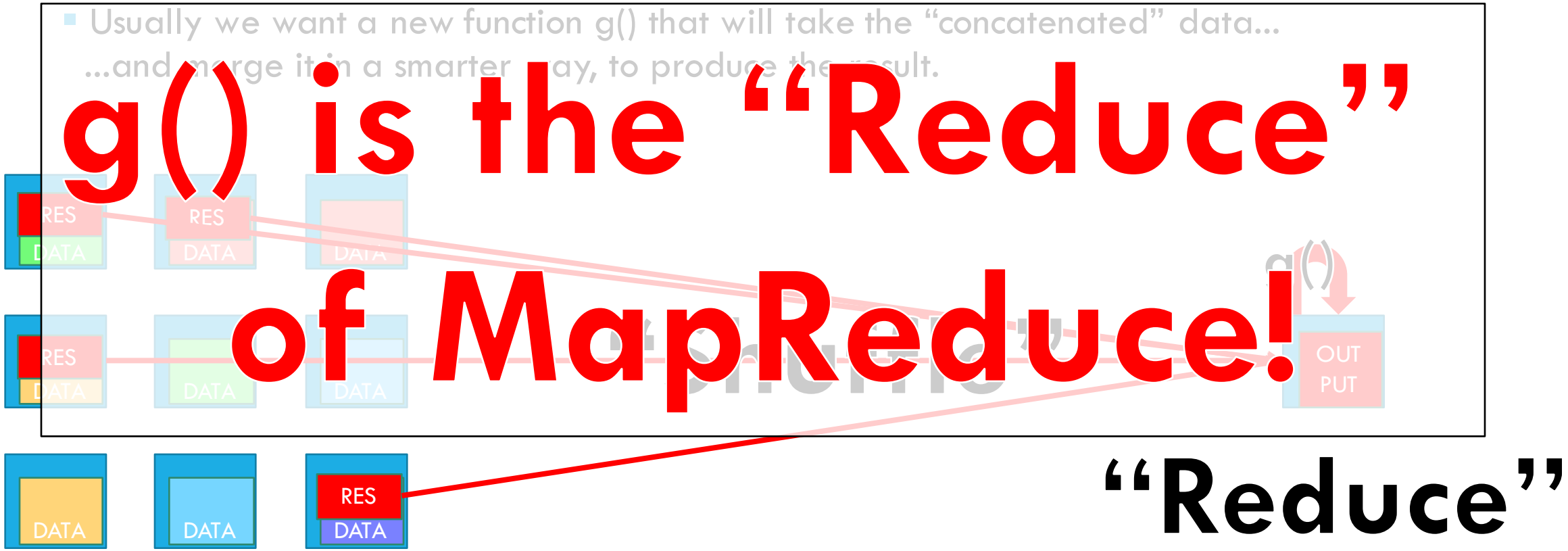


HOW DO YOU PROCESS THIS DATA?

- After the Map phase, you have partial results located on all nodes…
- **So you want to gather and aggregate all of these results into global results!**
- **Intermediary phase:** the Shuffle phase brings all data to one machine (could be one of the previous ones)

HOW DO YOU PROCESS THIS DATA?

- The Shuffle phase naïvely "concatenates" the results together
- Usually we want a new function g() that will take the "concatenated" data...
  ...and merge it in a smarter way, to produce the result.



g() is the "Reduce" of MapReduce!

"Reduce"

HOW DO YOU PROCESS THIS DATA?

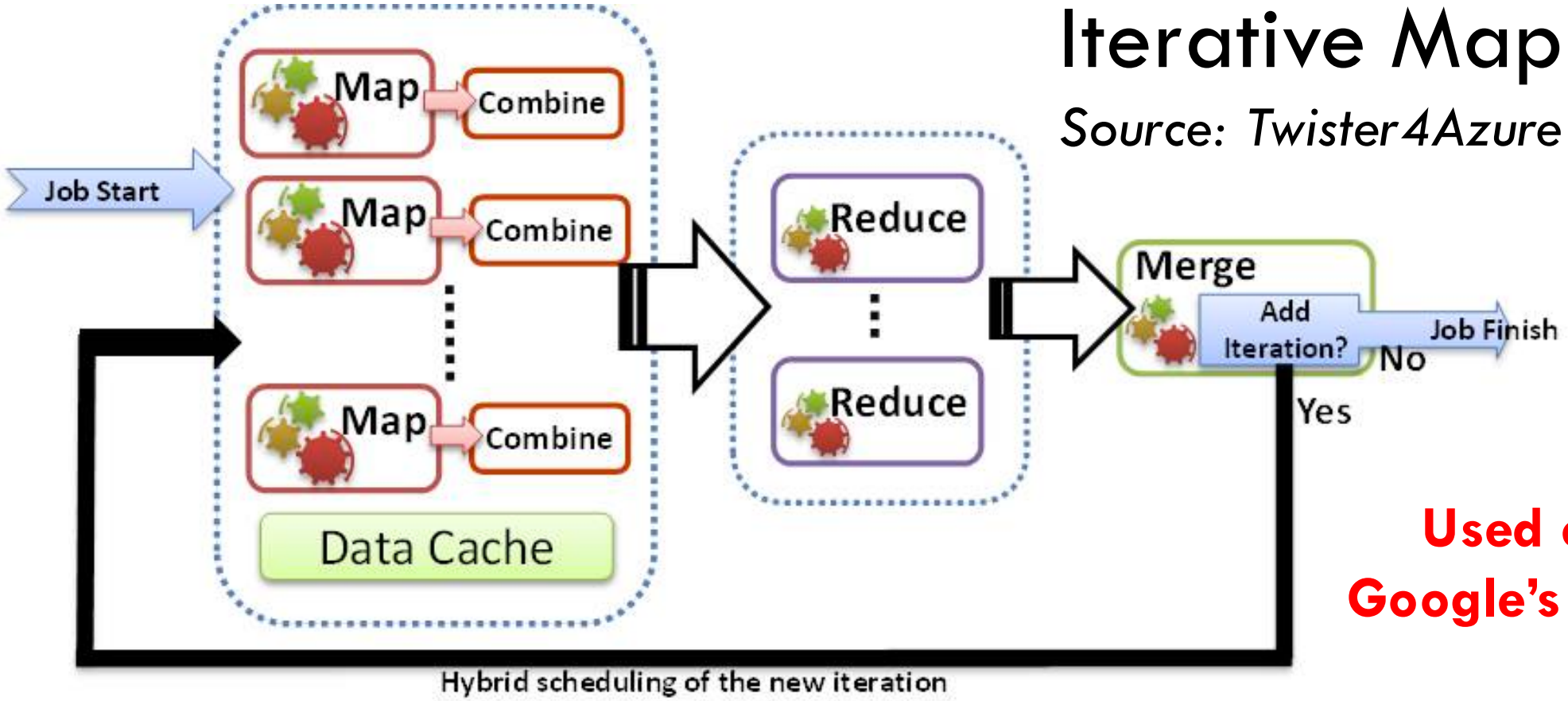
- MapReduce can seem a bit restrictive:
  You have to express all of your algorithms with two functions, Map and Reduce.
- And actually, it is: you can't express everything with MapReduce.
- **But in practice, you will see that many operations that are executed on large amounts of data can be expressed following this paradigm!**
- And if you're able to, you can very easily implement an algorithm that scales well…
- **…without having to worry about how you distribute, replicate, or transfer the data!**

HOW DO YOU PROCESS THIS DATA?

In practice, it can get more complicated than this, among other things :

- **You can alter the Shuffle phase with a Combiner** that will prepare the data after the Map phase locally before it's sent to the Reducer (useful for reducing the amount of data transferred)
- **You can use several Reducers:** each will produce part of the data, results stored on the HDFS, so the results will just look like a bunch of files, which sometimes can be just what you want (just merge them or something)
- ...but pretty often, when you have several Reducers, you'll want to combine the data again, so what do you do?
- **You can run a Map phase on the output of your reducers again!**
  ...and you can do this over and over again (iterative MapReduce)

HOW DO YOU PROCESS THIS DATA?



Used e.g. for Google's PageRank

WAIT BUT THAT'S MAPREDUCE, WHAT'S HADOOP?

- MapReduce was invented and is used by Google. Hadoop is a free, open-source implementation of MapReduce. A bit of history...

- **Early 2000s: Doug Cutting develops two open-source search projects:**
  - Lucene: Search indexer, used e.g., by Wikipedia
  - Nutch: A spider/crawler (with Mike Carafella)

- **Nutch:**
  - Aims to become a web-scale, crawler-based search
  - Written by a few part-time developers
  - Distributed, "by necessity" (too much data)
  - Able to parse100MB of web pages on 4 nodes, but can't scale to the whole web...

Source: UPenn

WAIT BUT THAT'S MAPREDUCE, WHAT'S HADOOP?

- **2003/2004: Google File System (GFS) and MapReduce papers published**
  - SOSP 2003: Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: "The Google File System"
  - OSDI 2004: Jeffrey Dean and Sanjay Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters"
  - Directly addressed Nutch's scaling issues
- **Following this, GFS & MapReduce added to Nutch**
  - Two part-time developers over two years (2004-2006)… With 20 nodes.
  - Much easier to program and run, scales to several 100M web pages… …but still far from web scale

Source: UPenn

WAIT BUT THAT'S MAPREDUCE, WHAT'S HADOOP?

- **2006: Yahoo hires Doug Cutting**
  - Provides engineers, clusters, users… Big boost for the project, tens of M$
  - Not without a price: slightly different focus (e.g. security) than the rest of the project, delays results…
- **Following this, Hadoop project splits out of Nutch!**
  - HDFS corresponds to Google's GFS
  - **Finally hits web scale in 2008!**

Source: UPenn

WAIT BUT THAT'S MAPREDUCE, WHAT'S HADOOP?

- **Cutting is now at Cloudera…**
  - Originally a startup, started by three top engineers from Google, Facebook, Yahoo, and a former executive from Oracle
  - Has its own version of Hadoop; software remains free, but company sells support and consulting services
  - Was elected chairman of Apache Software Foundation
  - **Now Hadoop maintained by the Apache Foundation!**

Source: UPenn

HADOOP IN PRACTICE

- **Map and Reduce functions operate on (key, value) pairs**
- E.g., the Map function takes (key, value) pairs, produces (hopefully less!) pairs that are sent as the input of the Reduce function…
- **The Shuffle phase concatenates the values that have the same key.**

  For instance, if your Map phase outputs three pairs :

  `("foo", 3), ("bar", 4)` and `("foo", 5)`

  The Reduce phase will receive :

  `("foo", [3, 5]), ("bar", 4)`
- **The Reduce function takes these pairs and produces (key, value) pairs again… Which means that your output will always be a list of (key, value) pairs ! (It may need further processing.)**

HADOOP IN PRACTICE

- **Let's start with an example: we have files that contain meteorological data**
- **These files contain records, each record is one line, containing:**
  - The code of a weather station on five digits
  - The year when the temperature was recorded
  - The average temperature for that year times ten on four digits (we'll suppose they're all positive to simplify things, multiplication to avoid floats). Only one data point per year here so it's not really Big Data, but this is just a toy example, we could have a lot more records, one per hour for instance.
  - Many more fields such as the wind speed, humidity, etc.
- **An example of a record:** `123451950016393627 43...`

HADOOP IN PRACTICE

- The input data will look like this:

```
123451950016393627 43
121231950013418927 69
121111950013112719 87
120941949022312121 22
120931949016512091 82
...
```

- The data can be stored in many files, one per weather station, one per year... Etc.
- **We'll use Hadoop to calculate the maximum average temperature for each year!**

HADOOP IN PRACTICE

```
12345195001639362743
12123195001341892769
12111195001311271987
12094194902231212122
12093194901651209182
...
```

- **What will the input of the Map function be?**
  - Each line produces a (key, value) pair
  - We can ignore the key (usually the character offset), the value is the contents of the line:

```
(0, 12345195001639362743)
(20, 12123195001341892769)
(40, 12111195001311271987)
(60, 12094194902231212122)
(80, 12093194901651209182)
...
```

HADOOP IN PRACTICE

```
1234519500163936274 3
1212319500134189276 9
1211119500131127198 7
1209419490223121212 2
1209319490165120918 2
...
```

- **What will the Map function do?**
  - It will discard the key, parse the values, and return (key, value) pairs where the key is the year and the value is the average temperature. The output will be:

```
(1950, 0163)
(1950, 0134)
(1950, 0131)
(1949, 0223)
(1949, 0165)
...
```

  - So basically our Map function will be a Java function that takes two parameters, the key (a number) and the value (a string), it will parse the string using the standard API, and produce the (key, value) pair as the output...

HADOOP IN PRACTICE

```
12345195001639362743
12123195001341892769
12111195001311271987
12094194902231212122
12093194901651209182
...
```

- **What will the Shuffle phase do?**
  - As we've seen earlier, it will concatenate the values for each key. Plus, keys are sorted:

```
(1949, [0223, 0165])
(1950, [0163, 0134, 0131])
...
```

  - We don't have to implement this phase, it's done automatically...

HADOOP IN PRACTICE

```
1234519500163936274 3
1212319500134189276 9
1211119500131127198 7
1209419490223121212 2
1209319490165120918 2
...
```

- **What will the Reduce phase do?**
  - It's just going to calculate the maximum of each list. The input was:

```
(1949, [0223, 0165])
(1950, [0163, 0134, 0131])
...
```

  - The output will be:

```
(1949, 0165)
(1950, 0163)
...
```

  - **And that's it, we have the result we want!** All we have to do is to implement two very simple functions in Java, Map and Reduce, and everything else, distribution, replication, load-balancing (of keys), fault tolerance (tasks rescheduled to machines that work), etc., will be handled by Hadoop!

HADOOP IN PRACTICE

```
12345195001639362743
12123195001341892769
12111195001311271987
12094194902231212122
12093194901651209182
...
```

- **One last thing before we start writing the code…**
- **Hadoop uses its own serialization because** Java serialization known to be inefficient
- **Result: a special set of data types**
  - All implement the "Writable" interface
  - Most common types shown here…
    …more specialized types exist (SortedMapWritable, ObjectWritable…)

Name	Description	JDK equivalent
IntWritable	32-bit integers	Integer
LongWritable	64-bit integers	Long
DoubleWritable	Floating-point numbers	Double
Text	Strings	String

Source: UPenn

HADOOP IN PRACTICE

```
1234519500163936274 3
1212319500134189276 9
1211119500131127198 7
1209419490223121212 2
1209319490165120918 2
...
```

- **First thing to do when you write a MapReduce program :** find the input types of the Map and Reduce functions.
- Map takes pairs like (0, 1234519500163936274 3), and produces pairs like (1950, 0163). Input types = (LongWritable, Text), output types = (Text, IntWritable), for instance. (We could use an IntWritable for the year too, but we never use its numerical properties.)
- **Consequently, the Map class will extend:**

```
Mapper<LongWritable, Text, Text, IntWritable>
```

- **And contain this function:**

```
public void map(LongWritable key, Text value, Context context)
```

HADOOP IN PRACTICE

```
12345195001639362743
12123195001341892769
12111195001311271987
12094194902231212122
12093194901651209182
...
```

- **First thing to do when you write a MapReduce program :** find the input types of the Map and Reduce functions.
- Reduce takes pairs like (1949, [0223, 0165]), and produces pairs like (1949, 0165). Input types = (Text, IntWritable), output types = (Text, IntWritable), for instance.
- **Consequently, the Reduce class will extend:**
  `Reducer<Text, IntWritable, Text, IntWritable>`
- **And contain this function:**
  `reduce(Text key, Iterable<IntWritable> values, Context context)`

HADOOP IN PRACTICE

```
1234519500163936274 3
1212319500134189276 9
1211119500131127198 7
1209419490223121212 2
1209319490165120918 2
...
```

```
class MaxTemperature {
    static class MaxTemperatureMapper
            extends Mapper<LongWritable, Text, Text, IntWritable> {

        public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {
            String line = value.toString();
            String year = line.substring(5, 9);
            int temp = Integer.parseInt(line.subString(9, 13));
            context.write(new Text(year), new IntWritable(temp));
        }
    }
    ...
    // Now write the reducer.
    ...
    // And create the main() function that creates the job and launches it.
}
```

YOUR TURN!

- That's enough information to get you started!
- You can now start working on the exercises you will find here: http://i3s.unice.fr/~jplozi/hadooplab_lsds/hadooplab_lsds.pdf
- You will probably need more information than just what we saw in these slides... **...you're expected to use Google and to figure things out on your own!**

Good luck !