CLOUD COMPUTING SYSTEMS

Lectures 6-7

Nuno Preguiça

(nuno.preguica\_at\_fct.unl.pt)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 1**

OUTLINE

Computing services

1. First generation batch processing: Map-reduce 2. Second generation batch processing: Spark 3. Stream processing

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 2**

OUTLINE

Computing services

**1. First generation batch processing: Map-reduce** • **Programming model**

• Execution model

• Handling faults

2. Second generation batch processing: Spark 3. Stream processing

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 3**

MAPREDUCE

“A new abstraction that allows us to expresses **simple computations** we were trying to perform but **hides the messy details** of parallelization, fault tolerance, data-distribution and load-balancing in a library”

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 4**

MAPREDUCE (2)

“A programming **model** and an associated

**implementation** for

processing **large datasets.”**

“Runs on a large cluster of **commodity machines** … a typical … computation

processes many terabytes of data on **thousands** of

machines.”

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 5**

EXAMPLE APPLICATION

Consider you have a huge text.

**Goal:** find out the words that appear more frequently in a text. **Can be transformed into:**

**Goal 1:** Count the number of times each word appears in the text.

**Goal 2:** Order the words by frequency.

Is this a useless example?

Not really… e.g. analyze web logs to find popular URLs, analyze social media posts to find trending topics, etc.

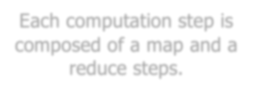
**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 6**

MAPREDUCE: OVERVIEW

Sequentially read a lot of data

**Map phase:**

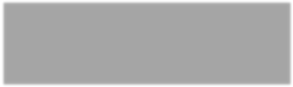
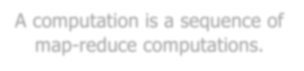
• Extract the important information

Each computation step is composed of a map and a reduce steps.

**Group by key:** Sort and Shuffle the output of the map phase

**Reduce phase:**

• Aggregate, summarize, filter or transform

Write the resultA computation is a sequence of map-reduce computations. 

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 7**

MAPREDUCE: EXAMPLE **Map**: read the input

and produce

key,value pairs

**Sort & shuffle**: performed by the system

**Reduce**: collect values with the same key and produce

(NOVA,1)

(ADA,1)

(ADA,1)

result

NOVA SBE NUNO

NOVA

LUDWIG SBE ANGELO

NOVA ADA CARCAVELOS NOVA MSC SBE

(SBE,1)

(NUNO,1)

(NOVA,1)

(LUDWIG,1)

(SBE,1)

(ANGELO,1)

(NOVA,1)

(ADA,1)

(CARCAVELOS,1)

(NOVA,1)

(MSC,1)

(SBE,1)

**Map**: for each word, output its count.

(ANGELO,1)

(CARCAVELOS,1) (LUDWIG,1)

(MSC,1)

(NOVA,1)

(NOVA,1)

(NOVA,1)

(NOVA,1)

(NUNO,1)

(SBE,1)

(SBE,1)

(SBE,1)

(ANGELO,1)

(CARCAVELOS,1)

(LUDWIG,1)

(MSC,1)

(NOVA,4)

(NUNO,1)

(SBE,3)

**Reduce**: count the frequency per word.

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 8**

WORD COUNT USING MAPREDUCE

map(key, value): // key: document name; value: text of the document for each word w in value:

emit(w, 1)

reduce(key, values): // key: a word; value: an iterator over counts result = 0

for each count v in values:

result += v

emit(key, result)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 9**

MAPREDUCE MODEL

**Input:** a set of key-value pairs

Programmer specifies two methods:

• **Map(k, v)** → <k’, v’>\*

• Takes a key-value pair and outputs a set of key-value pairs • E.g., key is the filename, value is a single line in the file

• Map is called for every (k,v) pair

• **Reduce(k’, <v’>\*)** → <k’, v’’>\*

• All values v’ with same key k’ are reduced together and

processed in v’ order

• Reduce is called for each unique key k’

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 10**

EXAMPLE APPLICATION

Consider you have a huge text.

**Goal:** find out the words that appear more frequently in a text. **Can be transformed into:**

**Goal 1:** Count the number of times each word appears in the text.

**Goal 2:** Order the words by frequency.

Is this a useless example?

Not really… e.g. analyze web logs to find popular URLs, analyze social media posts to find trending topics, etc.

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 11**

GOAL 2: ORDER THE WORDS BY FREQUENCY.

Can we sort the values of the reduce before returning them?

**Reduce**: collect values with the same key and produce

**Reduce**: collect values with the same key and produce

(ADA,1)

result (ADA,1)

result

**NO !!!**

Each reduce will be

processed independently (by a different machine).

Also bad idea because it requires storing potentially large amount of data.

(ANGELO,1)

(CARCAVELOS,1) (LUDWIG,1)

(MSC,1)

(NOVA,4)

(NUNO,1)

(SBE,3)

(ANGELO,1)

(CARCAVELOS,1) (LUDWIG,1)

(MSC,1)

(NUNO,1)

(NOVA,4)

(SBE,3)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 12**

MAPREDUCE: EXAMPLE

**Map**: read the input

and produce

key,value pairs

**Sort & shuffle**: performed by the system

(1,ADA)

**Reduce**: collect values with the same key and produce result

(ADA,1)

(ANGELO,1)

(CARCAVELOS,1) (LUDWIG,1)

(MSC,1)

(NOVA,4)

(NUNO,1)

(SBE,3)

(1,ADA)

(1,ANGELO)

(1,CARCAVELOS)

(1,LUDWIG)

(1,MSC)

(4,NOVA)

(1,NUNO)

(3,SBE)

**Map**: reverse order of pair.

(1,ANGELO)

(1,CARCAVELOS) (1,LUDWIG)

(1,MSC)

(1,NUNO)

(3,SBE)

(4,NOVA)

(ADA,1)

(ANGELO,1)

(CARCAVELOS,1)

(LUDWIG,1)

(MSC,1)

(NUNO,1)

(SBE,3)

(NOVA,4)

**Reduce**: reverse order of pair.

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 13**

WORD COUNT SORT USING MAPREDUCE

map(key, value): // key: word; value: word count

emit(value, key)

reduce(key, values): // key: word count; value: word

for each v in values:

emit(v, key)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 14**

MAPREDUCE

**Programmer** responsible for**:**

• **Map** function

• **Reduce** function

**MapReduce system** responsible for**:**

• **Partitioning** the input data

• **Scheduling** the program’s execution across a set of machines • Performing the **sort by key & shuffle** step

• Handling machine **failures**

• Managing required inter-machine **communication**

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 15**

OUTLINE

Computing services

**1. First generation batch processing: Map-reduce** • Programming model

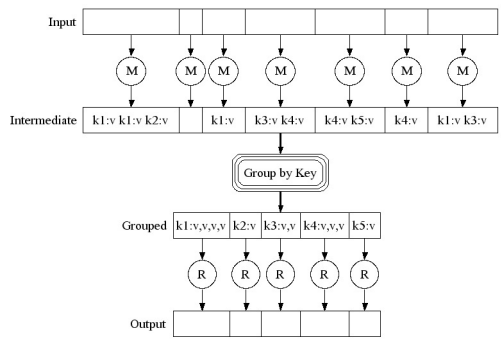
• **Execution model**

• **Handling faults**

2. Second generation batch processing: Spark 3. Stream processing

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 16**

MAPREDUCE: LOGICAL EXECUTION…

**Map**: read the input 

and produce

key,value pairs

**Sort & shuffle**:

performed by the

system

**Reduce**: collect

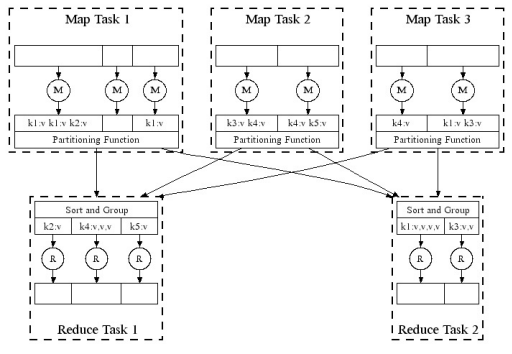
values with the same

key and produce

result

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 17**

MAPREDUCE: DISTRIBUTED EXECUTION…

**Map**: read the input 

and produce

key,value pairs

**Sort & shuffle**:

performed by the

system

**Reduce**: collect

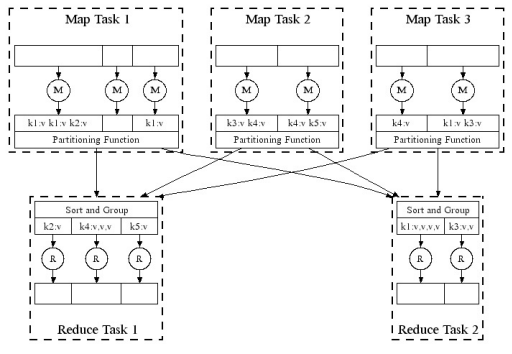
values with the same

key and produce

result

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 18**

MAPREDUCE: DISTRIBUTED EXECUTION…

**Map**: read the input 

and produce

key,value pairs

**Sort & shuffle**:

performed by the

system

**Reduce**: collect

values with the same

key and produce result

Each phase is divided in

**multiple tasks**. Each task

is executed independently

on a **different nodes.**

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 19**

MAPREDUCE ARCHITECTURE

2. Clients send

programs to the

master.

1. Input files stored in a distributed file system, and

divided into splits

3. Master asks

workers to run map tasks: process splits

(1) fork

in parallel, and...

(2)

assign

map

worker

User

Program (1) fork

Master

(1) fork

(2)

assign reduce

4. Master asks workers to run reduce tasks: reducers sort intermediate files before processing values for each key.

split 0 split 1 split 2

(3) read

(4) local write

(5) remote read

worker

(6) write

output file 0

split 3 split 4

worker worker

worker output file 1

Input files

Map phase

Intermediate files

3. ...save

(on local disks)

intermediate results in multiple files by key Figure 1: Execution overview

range

Reduce phase

Output files

**Inverted Index:** The map function parses each docu ment, and emits a sequence of !word, document ID" pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 20**

large clusters of commodity PCs connected together with

switched Ethernet [4]. In our environment:

(1) Machines are typically dual-processor x86 processors

running Linux, with 2-4 GB of memory per machine.

MAPREDUCE SYSTEM: MASTER NODE

**Master node** coordinates the

execution:

**Task status:** (idle, in-progress,

User

Program

completed)

(1) fork

(1) fork

(1) fork

**Idle tasks** get scheduled as

Master

workers become available

(2)

assign

(2)

reduce

assign

When a map task completes, it

map

sends the master the location and sizes of its intermediate files, one for each reducer

split 0 split 1 split 2

worker

(3) read

(5) remote read

(4) local write

worker

(6) write

output file 0

Master pushes this info to reducers

split 3 split 4

worker worker

worker output file 1

Input

Map

Master pings workers periodically

files

phase

to detect failures

Intermediate files

(on local disks)

Figure 1: Execution overview

Reduce phase

Output files

**Inverted Index:** The map function parses each docu ment, and emits a sequence of !word, document ID" pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a

large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

(1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 21**

!word, list(document ID)" pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

**Distributed Sort:** The map function extracts the key from each record, and emits a !key, record" pair. The reduce function emits all pairs unchanged. This compu tation depends on the partitioning facilities described in Section 4.1 and the orderin roerties described in Sec

(2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in over all bisection bandwidth.

(3) A cluster consists of hundreds or thousands of ma chines, and therefore machine failures are common.

(4) Storage is provided by inexpensive IDE disks at tached directly to individual machines. A distributed file

MAPREDUCE SYSTEM: WORKER

**Worker node** performs

map or reduce tasks, as

User

Program

requested by the

(1) fork

(1) fork

(1) fork

coordinator.

Master

(2)

assign

(2)

reduce

assign

map

worker

split 0

(6) write

output

split 1 split 2

(3) read

(4) local write

(5) remote read

worker

file 0

split 3 split 4

worker worker

worker output file 1

Input files

Map phase

Intermediate files

(on local disks)

Figure 1: Execution overview

Reduce phase

Output files

**Inverted Index:** The map function parses each docu ment, and emits a sequence of !word, document ID" pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a

large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

(1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 22**

!word, list(document ID)" pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

**Distributed Sort:** The map function extracts the key from each record, and emits a !key, record" pair. The reduce function emits all pairs unchanged. This compu tation depends on the partitioning facilities described in Section 4.1 and the orderin roerties described in Sec

(2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in over all bisection bandwidth.

(3) A cluster consists of hundreds or thousands of ma chines, and therefore machine failures are common.

(4) Storage is provided by inexpensive IDE disks at tached directly to individual machines. A distributed file

MAPREDUCE SYSTEM: HANDLING FAULTS **Map worker failure**

Upon detection of the failure of a worker, map tasks

restarted in different worker

**Reduce worker failure** Reduce task is restarted in

(1) fork

(2)

assign

reduce

split 0

(1) fork

(2)

assign

map

worker

User

Program

(1) fork

Master

(6) write

output

other worker

split 1 split 2

(3) read

(5) remote read

(4) local write

worker

file 0

**Stragglers (slow workers)**

split 3 split 4

worker worker

worker output file 1

If a task is taking too long to complete, it is launched in other worker. First result used.

Input files

Map phase

Intermediate files

(on local disks)

Figure 1: Execution overview

Reduce phase

Output files

**Inverted Index:** The map function parses each docu ment, and emits a sequence of !word, document ID" pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a

large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

(1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 23**

!word, list(document ID)" pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

**Distributed Sort:** The map function extracts the key from each record, and emits a !key, record" pair. The reduce function emits all pairs unchanged. This compu tation depends on the partitioning facilities described in Section 4.1 and the orderin roerties described in Sec

(2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in over all bisection bandwidth.

(3) A cluster consists of hundreds or thousands of ma chines, and therefore machine failures are common.

(4) Storage is provided by inexpensive IDE disks at tached directly to individual machines. A distributed file

MAPREDUCE SYSTEM: HANDLING FAULTS (2) **Master failure**

MapReduce task is aborted and client is notified

(1) fork

(2)

assign

reduce

split 0

(1) fork

(2)

assign

map

worker

User

Program

(1) fork

Master

(6) write

output

split 1 split 2

(3) read

(4) local write

(5) remote read

worker

file 0

split 3 split 4

worker worker

worker output file 1

Input files

Map phase

Intermediate files

(on local disks)

Figure 1: Execution overview

Reduce phase

Output files

**Inverted Index:** The map function parses each docu ment, and emits a sequence of !word, document ID" pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a

large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

(1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 24**

!word, list(document ID)" pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

**Distributed Sort:** The map function extracts the key from each record, and emits a !key, record" pair. The reduce function emits all pairs unchanged. This compu tation depends on the partitioning facilities described in Section 4.1 and the orderin roerties described in Sec

(2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in over all bisection bandwidth.

(3) A cluster consists of hundreds or thousands of ma chines, and therefore machine failures are common.

(4) Storage is provided by inexpensive IDE disks at tached directly to individual machines. A distributed file

OUTLINE

Computing services

**1. First generation batch processing: Map-reduce** • **Programming model**

• Execution model

• Handling faults

2. Second generation batch processing: Spark 3. Stream processing

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 25**

MAPREDUCE: EXAMPLE **Map**: read the input

and produce

key,value pairs

**Sort & shuffle**: performed by the system

**Reduce**: collect values with the same key and produce

(NOVA,1)

(ADA,1)

(ADA,1)

result

NOVA SBE NUNO

NOVA

LUDWIG SBE ANGELO

NOVA ADA CARCAVELOS NOVA MSC SBE

(SBE,1)

(NUNO,1)

(NOVA,1)

(LUDWIG,1)

(SBE,1)

(ANGELO,1)

(NOVA,1)

(ADA,1)

(CARCAVELOS,1)

(NOVA,1)

(MSC,1)

(SBE,1)

**Map**: for each word, output its count.

(ANGELO,1)

(CARCAVELOS,1) (LUDWIG,1)

(MSC,1)

(NOVA,1)

(NOVA,1)

(NOVA,1)

(NOVA,1)

(NUNO,1)

(SBE,1)

(SBE,1)

(SBE,1)

(ANGELO,1)

(CARCAVELOS,1)

(LUDWIG,1)

(MSC,1)

(NOVA,4)

(NUNO,1)

(SBE,3)

**Reduce**: count the frequency per word.

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 26**

MAPREDUCE: EXAMPLE **Map**: read the input

and produce

key,value pairs

**Sort & shuffle**: performed by the system

**Reduce**: collect values with the same key and produce

(NOVA,1)

(ADA,1)

result

NOVA SBE NUNO

NOVA

LUDWIG SBE ANGELO

NOVA ADA CARCAVELOS NOVA MSC SBE

(SBE,1)

(NUNO,1)

(NOVA,1)

(LUDWIG,1)

(SBE,1)

(ANGELO,1)

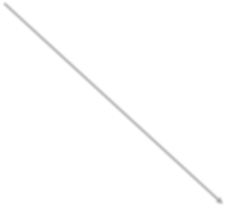
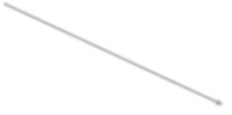
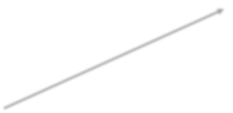
(NOVA,1)

(ADA,1)

(CARCAVELOS,1) (NOVA,1)

(MSC,1)

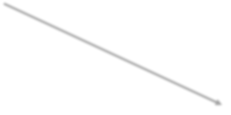
(SBE,1)

(ANGELO,1) (CARCAVELOS,1) 

(LUDWIG,1)

(MSC,1)

(NOVA,4)

(NUNO,1) 

(SBE,3)



**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 27**

MAPREDUCE: EXAMPLE **Map**: read the input

and produce

key,value pairs

**Sort & shuffle**: performed by the system

**Reduce**: collect values with the same key and produce

(NOVA,1)

(ADA,1)

result

NOVA SBE NUNO

NOVA

LUDWIG SBE ANGELO

NOVA ADA CARCAVELOS NOVA MSC SBE

(SBE,1)

(NUNO,1)

(NOVA,1)

(LUDWIG,1)

(SBE,1)

(ANGELO,1)

(NOVA,1)

(ADA,1)

(CARCAVELOS,1) (NOVA,1)

(MSC,1)

(SBE,1)

(ANGELO,1) (CARCAVELOS,1) 

(LUDWIG,1)

(MSC,1)

(NOVA,4)

(NUNO,1) 

(SBE,3)



Mapper 1 sends two tuples

for NOVA !!

**How to improve this?**

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 28**

IMPROVING MAPREDUCE: COMBINER

Combiner allows to pre

aggregate values in the

mapper.

**Combine(k, <v>\*)** → <k, v’>

**MapSort &  shuffle**

**Reduce**

All values v with same key k

are combined and processed

in v order

Combine is called at each

mapper for each unique key

k

Combiner function is usually the same as the reduce function.

NOVA SBE NUNO

NOVA

LUDWIG SBE ANGELO

NOVA ADA CARCAVELOS NOVA MSC SBE

(NOVA,1)

(SBE,1)

(NUNO,1)

(NOVA,1)

(LUDWIG,1)

(SBE,1)

(ANGELO,1)

(NOVA,1)

(ADA,1)

(CARCAVELOS,1) (NOVA,1)

(MSC,1)

(SBE,1)



**r**

**e**

**n**

**ib**

**m**

**o**

**C**

(ADA,1)

(ANGELO,1)

(CARCAVELOS,1) (LUDWIG,1)

(MSC,1)

(NOVA,4)

(NUNO,1)

(SBE,3)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 29**

WHY WAS MAP-REDUCE SO POPULAR?

Distributed computation before MapReduce:

• how to divide the workload among multiple machines? • how to distribute data and program to other machines? • how to schedule tasks?

• what happens if a task fails while running?

• … and … and ...

Distributed computation after MapReduce

• how to write Map function?

• how to write Reduce function?

• systems to efficiently execute map-reduce jobs: Hadoop.**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 30**

APACHE HADOOP 2.0

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 31**

OUTLINE

Computing services

1. First generation batch processing: Map-reduce **2. Second generation batch processing: Spark** 3. Stream processing

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 32**

MAPREDUCE: CHAINING PROGRAMS

MapReduce requires complex computations to be split into successive MapReduce jobs

These complex programs can experience **high latency** due to several factors, including:

• need to **read and write files**

• underlying filesystem **replication** (for writes)

• one job must finish before the next can be started…

Apache Spark tackles these limitations.

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 33**

APACHE SPARK

Apache Spark provides in-memory, fault-tolerant distributed processing.

Key ideas:

• Spark programs comprise **multiple chained data**

**transformations**, using a high-level functional programming model;

• Spark defines a distributed collection data-structure : **Resilient Distributed Dataset** (RDD).

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 34**

DATA MODEL AND APIS

RDDs are immutable data

• logically a RDD is an **immutable collection of data tuples**;

• **physically distributed** (partitioned) across many nodes;

• upon a failure (or cascade of failures), RDDs can be **recreated** automatically and efficiently **from the dependencies**.

Spark Dataframes

• DataFrames are distributed collections of data that is grouped into named columns.

• DataFrames can be seen as RDDs with a schema that names the fields of the underlying tuples.

Spark SQL

• SQL for specifying computations

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 35**

**SPARKSQL ARCHITECTURE**

Programs using SQL/DataFrames are **translated** into Spark programs.

Programs are **optimized** to execute efficiently.

JDBC Console User Programs (Java, Scala, Python) 

Based on the techniques used in database systems.

Spark SQL

DataFrame API

Libraries for advanced analytics algorithms such as **graph processing** and **machine learning**.

Catalyst Optimizer

Spark 

Resilient Distributed Datasets

Figure 1: Interfaces to Spark SQL, and interaction with Spark.

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 36** 3.1 DataFrame API

The main abstraction in Spark SQL’s API is a DataFrame, a dis tributed collection of rows with a homogeneous schema. A DataFrais equivalent to a table in a relational database, and can also be

FIRST EXAMPLE

**SparkSession.builder. …**

• A SparkSession represents the entry point to submit programs to a Spark cluster. • master("local") : defines where the master Spark node is located – local means running on local mode, i.e., not connected to a cluster.

from pyspark.sql import SparkSession

spark = SparkSession.builder \

.master("local") \

.appName("Simple test") \

.getOrCreate()

try:

df = spark.read.text("doc.txt")

df.printSchema()

df.show()

finally:

spark.stop()

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 37**

FIRST EXAMPLE (2)

**spark.stop()**

• Shutdown the underlying SparkContext.

• You should stop a SparkContext in the end, as only a single SparkContext may exist – we are doing this in a finally clause to guarantee this.

from pyspark.sql import SparkSession

spark = SparkSession.builder \

.master("local") \

.appName("Simple test") \

.getOrCreate()

try:

df = spark.read.text("doc.txt")

df.printSchema()

df.show()

finally:

spark.stop()

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 38**

FIRST EXAMPLE: CREATING DATAFRAME FROM TEXT FILE

**dataframe = spark.read.text(filename)**

• Creates a Dataframe from a text file. The Dataframe includes a single column named “value”, and each line is a row of the DataFrame.

from pyspark.sql import SparkSession

spark = SparkSession.builder \

.master("local") \

.appName("Simple test") \

.getOrCreate()

try:

df = spark.read.text("doc.txt")

df.printSchema()

df.show()

finally:

spark.stop()

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 39**

FIRST EXAMPLE: DISTRIBUTED EXECUTION

CPU + GPU

df = spark.read.text("doc.txt")

**File** “doc.txt”

NOVA SBE

NUNO

NOVA

LUDWIG SBE

ANGELO

NOVA ADA

CARCAVELOS

NOVA MSC

SBE

disk

[“NOVA SBE”]

[“NUNO”]

[“NOVA”]

CPU + GPU

memory disk

IO, net, …

memory disk

IO, net, …

[“LUDWIG SBE”] [“ANGELO”]

[“NOVA ADA”] CPU + GPU

memory disk

IO, net, …

[“CARCAVELOS”] [“NOVA MSC”] [“SBE”]

CPU + GPU

memory disk

IO, net, …

**df DataFrame** is distributed across 3 machines **Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 40**

FIRST EXAMPLE: CREATING DATAFRAME FROM TEXT FILE

**dataframe.show()**

• Displays the contents of the DataFrame.

• To show the values of a DataFrame, it is necessary to collect them – remember that a DataFrme might be distributed over multiple machines, and your program is running in a single machine.

from pyspark.sql import SparkSession

spark = SparkSession.builder \

.master("local") \

.appName("Simple test") \

.getOrCreate()

try:

df = spark.read.text("doc.txt")

df.printSchema()

df.show()

finally:

spark.stop()

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 41**

FIRST EXAMPLE: DISTRIBUTED EXECUTION CPU + GPU

Value of

variable **res** [“NOVA SBE”, “NUNO”,

“NOVA”,

“LUDWIG SBE”, “ANGELO”,

“NOVA ADA”, “CARCAVELOS”, “NOVA MSC” “SBE”]

df.show()

[“NOVA SBE”]

[“NUNO”]

[“NOVA”]

CPU + GPU

memory disk

IO, net, …

memory disk

IO, net, …

[“LUDWIG SBE”] [“ANGELO”]

[“NOVA ADA”] CPU + GPU

memory disk

IO, net, …

[“CARCAVELOS”] [“NOVA MSC”] [“SBE”]

CPU + GPU

memory disk

IO, net, …

**df DataFrame** is distributed across 3 machines**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 42**

PROGRAMMING MODEL

Spark Dataframe programs describe the flow of

transformations that creates a DataFrame from another, usually in several steps.

Spark programs, encode the dependencies among the various DataFrames (and underlying RDDs):

• this is known as the **lineage graph**

4

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 43**

3

SECOND EXAMPLE

Count the number of occurrences of each word and print those that appear more than once.

df2 = df.select(explode(split(col("value"), " ")).alias("word")) result = df2.groupBy(df2.word) \

.count() \

.where(col("count") > 1)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 44**

SECOND EXAMPLE: EXPLODE + SPLIT

**split( column, delimeter)**

• Divides the value of the column by delimiter, creating an array of values

**explode( column).alias(name)**

• Flattens the array, making each value an independent row, with name the result column.

df2 = df.select(explode(split(col("value"), " ")).alias("word")) result = df2.groupBy(df2.word) \

.count() \

.where(col("count") > 1)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 45**

SECOND EXAMPLE: FLATMAP

(“NOVA SBE”) (“NUNO”)

(“NOVA”)

(“LUDWIG SBE”) (“ANGELO”)

(“NOVA ADA”)

(“CARCAVELOS”) (“NOVA MSC”) (“SBE”)

df2 = df.select(explode(split(col("value"), " ")).alias("word"))

(“NOVA”) (“SBE”)

(“NUNO”) (“NOVA”)

(“LUDWIG”) (“SBE”)

(“ANGELO”) (“NOVA”) (“ADA”)

(“CARCAVELOS”) (“NOVA”)

(“MSC”)

(“SBE”)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 46**

SECOND EXAMPLE: GROUPBY

**groupBy( column)**

• Groups the rows using the value of the given column

df2 = df.select(explode(split(col("value"), " ")).alias("word")) result = df2.groupBy(df2.word) \

.count() \

.where(col("count") > 1)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 47**

SECOND EXAMPLE: GROUPBY().COUNT()

**groupBy( column).count()**

• Counts the number of rows in the group, adding a column with name “column”.

df2 = df.select(explode(split(col("value"), " ")).alias("word")) result = df2.groupBy(df2.word) \

.count() \

.where(col("count") > 1)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 48**

SECOND EXAMPLE: REDUCEBYKEY

(“NOVA”) (“SBE”)

(“NUNO”) (“NOVA”)

(“LUDWIG”) (“SBE”)

(“ANGELO”) (“NOVA”) (“ADA”)

(“CARCAVELOS”) (“NOVA”)

(“MSC”)

(“SBE”)

result = df2.groupBy(df2.word) \ .count() \

(“NOVA”,3) (“ANGELO”,1) (“NUNO”,1)

(“LUDWIG”,1) (“ADA”,1)

(“SBE”,4)

(“CARCAVELOS”,1) (“MSC”,1)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 49**

SECOND EXAMPLE: WHERE

**where ( condition)**

• Returns a DataFrame with the rows that satisfy the given condition.

df2 = df.select(explode(split(col("value"), " ")).alias("word")) result = df2.groupBy(df2.word) \

.count() \

.where(col("count") > 1)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 50**

SECOND EXAMPLE: FILTER

(“NOVA”,3) (“ANGELO”,1) (“NUNO”,1)

(“LUDWIG”,1) (“ADA”,1)

(“SBE”,4)

(“CARCAVELOS”,1) (“MSC”,1)

.where(col("count") > 1)

(“NOVA”,3) (“SBE”,4)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 51**

PROGRAMMING AND EXECUTION MODEL

DataFrame programs are converted into RDD programs, which involve:

• **Transformations**: RDD -> RDD

• **Actions**: RDD -> Result (directly available to the client application)

Execution consists in applying the transformations in all the partitions of an RDD in parallel

• Performance is best when a RDD partition result does not require data from input RDD partitions located in different nodes (i.e., avoids shuffles)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 52**

FROM DATAFRAME TO RDDS

df2 = df.select(explode(split(col("value"), " ")).alias("word")) result = df2.groupBy(df2.word) \

.count() \

.where(col("count") > 1)

freq = doc.flatMap (lambda s: s.split(' ‘)

.map(lambda s: (s,1))

.reduceByKey (lambda v1,v2: v1+v2)

.filter(lambda t: t[1] > 1)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 53**

FROM DATAFRAME TO RDDS (2)

df2 = df.select(explode(split(col("value"), " ")).alias("word")) result = df2.groupBy(df2.word) \

.count() \

.where(col("count") > 1)

freq = doc.flatMap (lambda s: s.split(' ‘)

.map(lambda s: (s,1))

.reduceByKey (lambda v1,v2: v1+v2)

.filter(lambda t: t[1] > 1)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 54**

FROM DATAFRAME TO RDDS (2)

df2 = df.select(explode(split(col("value"), " ")).alias("word")) result = df2.groupBy(df2.word) \

.count() \

.where(col("count") > 1)

freq = doc.flatMap (lambda s: s.split(' ‘)

.map(lambda s: (s,1))

.reduceByKey (lambda v1,v2: v1+v2)

.filter(lambda t: t[1] > 1)

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 55**

SECOND EXAMPLE: COMPLETE EXECUTION

freq = doc.flatMap (lambda s: s.split(' ‘)

.map(lambda s: (s,1))

.reduceByKey (lambda v1,v2: v1+v2)

.filter(lambda t: t[1] > 1)

flatMap

V1 V2 V3 flatMap

map

V4

V5

V6

map

reduceByKey

V7

V8 V9

reduceByKey

filter

V10 V11 V12 filter

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 56**

APACHE SPARK: (SCALA) API EXCERPT

| Transformations | *map*(*f* : T *)* U) : RDD[T] *)* RDD[U]  *filter*(*f* : T *)* Bool) : RDD[T] *)* RDD[T]  *flatMap*(*f* : T *)* Seq[U]) : RDD[T] *)* RDD[U]  *sample*(*fraction* : Float) : RDD[T] *)* RDD[T] (Deterministic sampling)  *groupByKey*() : RDD[(K, V)] *)* RDD[(K, Seq[V])]  *reduceByKey*(*f* : (V*,*V) *)* V) : RDD[(K, V)] *)* RDD[(K, V)]  *union*() : (RDD[T]*,*RDD[T]) *)* RDD[T]  *join*() : (RDD[(K, V)]*,*RDD[(K, W)]) *)* RDD[(K, (V, W))]  *cogroup*() : (RDD[(K, V)]*,*RDD[(K, W)]) *)* RDD[(K, (Seq[V], Seq[W]))]  *crossProduct*() : (RDD[T]*,*RDD[U]) *)* RDD[(T, U)]  *mapValues*(*f* : V *)* W) : RDD[(K, V)] *)* RDD[(K, W)] (Preserves partitioning) *sort*(*c* : Comparator[K]) : RDD[(K, V)] *)* RDD[(K, V)]  *partitionBy*(*p* : Partitioner[K]) : RDD[(K, V)] *)* RDD[(K, V)] |
| --- | --- |
| Actions | *count*() : RDD[T] *)* Long  *collect*() : RDD[T] *)* Seq[T]  *reduce*(*f* : (T*,*T) *)* T) : RDD[T] *)* T  *lookup*(*k* : K) : RDD[(K, V)] *)* Seq[V] (On hash/range partitioned RDDs)  *save*(*path* : String) : Outputs RDD to a storage system, *e.g.,* HDFS |

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

that searches for a hyperplane *w* that best separates two 

ranks0 input file*map*

sets of points (*e.g.,* spam and non-spam emails). The al gorithm uses gradient descent: it starts *w* at a random value, and on each iteration, it sums a function of *w* over the data to move *w* in a direction that improves it.

links

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 57** *join* 

contribs0

*reduce + map*

**ranks1

contribs

PROGRAMMING MODEL: DEPENDENCIES

Narrow Dependencies: Wide Dependencies:

**Wide-Depend~~enci~~es** are

produced when ~~an R~~DD

Sta

partition depen~~ds on~~

map, filter

multiple partitions stored on different nodes

groupBy, join

Expensive due to high cost of

groupByKey

C:

network bandwidth union

join with inputs co-partitioned

join with inputs not co-partitioned

St

Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

Figure 5: with solid in black if t

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 58** G, we build

*map* to the parent’s records in its *iterator* method. *union*: Callin *union* on two RDDs returns an RDD

row transfoutput RD

PROGRAMMING MODEL: DEPENDENCIES (2)

**Narrow-dependencies** are produced when a RDD partition depends on data

Narrow Dependencies: Wide Depen

that is co-located (in the same node).

Filter (where), map

Fast as executed in the same

map, filter

join with inputs

groupB

machine.union

co-partitioned

join with inco-partit

Figure 4: Examples of narrow and wide dependenbox is an RDD, with partitions shown as shaded rect

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 59** *map* to the parent’s records in its *iterator* meth

*union*: Calling *union* on two RDDs returnswhose partitions are the union of those of th

FAULT-TOLERANCE

Sparks deals with node failures by **recomputing lost**

**partitions**, using lineage

V1 V2 V3 flatMap

information.

V4

map

V5

V6

Optimized by persisting

intermediate RDDs.

V7

V8 V9

reduceByKey

V10 V11 V12 filter

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 60**

FAULT-TOLERANCE

Sparks deals with node failures by **recomputing lost**

**partitions**, using lineage

V1 V2 V3 flatMap

information.

V4

map

V5

V6

Optimized by persisting

intermediate RDDs.

V7

V8 V9

reduceByKey

V10 V11 V12 filter

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 61**

FAULT-TOLERANCE

Sparks deals with node failures by **recomputing lost**

**partitions**, using lineage

V1 V2 V3 flatMap

information.

V4

map

V5

V6

Optimized by persisting

intermediate RDDs. In the example, if V9 is

V7

V8 V9

reduceByKey

persisted, lots of

recomputation would be saved.

V10 V11 V12 filter

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 62**

EXERCISES

Consider the information about products, stored in file "shopdata.csv", with the following format (where elements are separated by a tab):

```store product price```, where elements are separated by a tab.

6Ave Express LLC 13.3 MacBook Air (Mid 2017, Silver) 892.49 Amazon.com 13.3 MacBook Air (Mid 2017, Silver) 979 Best Buy 13.3 MacBook Air (Mid 2017, Silver) 899.99 bhphotovideo.com 13.3 MacBook Air (Mid 2017, Silver) 799

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 63**

LOAD CSV FILE

**dataframe = spark.read.csv(filename)**

• Creates a Dataframe from a CSV file.

• Option “header” specifies if the first line is the header of the table. • Option “inferSchema” instructs Spark to infer data type for each column.

df = spark.read.option("header", True) \

.option("inferSchema",True) \

.csv("shopdata.csv")

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 64**

REGISTER DATAFRAME AS SQL VIEW

**dataframe.createOrReplaceTempView( table\_name)**

Registers a DataFrame as a SQL view / table. The table is available for the SparkSession.

After registering the table, it is possible to issue SQL statements.

df = spark.read.option("header", True) \

.option("inferSchema",True) \

.csv("shopdata.csv")

df.createOrReplaceTempView("products")

result = spark.sql("SELECT \* FROM products")

result.show()

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 65**

EXECUTING SQL OPERATIONS

**dataframe = spark.sql( SQL statement)**

Execute SQL statement. The result is a DataFrame.

df = spark.read.option("header", True) \

.option("inferSchema",True) \

.csv("shopdata.csv")

df.createOrReplaceTempView("products")

result = spark.sql("SELECT \* FROM products")

result.show()

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 66**

EXECUTING SQL OPERATIONS

**dataframe = spark.sql( SQL statement)**

Execute SQL statement. The result is a DataFrame.

df = spark.read.option("header", True) \

.option("inferSchema",True) \

.csv("shopdata.csv")

df.createOrReplaceTempView("products")

result = spark.sql("SELECT \* FROM products")

result.show()

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 67**

EXECUTING SQL OPERATIONS

**dataframe = spark.sql( SQL statement)** 

Execute SQL statement. The result is a DataFrame.

df = spark.read.option("header", True) \

.option("inferSchema",True) \

.csv("shopdata.csv")

df.createOrReplaceTempView("products")

result = spark.sql("SELECT \* FROM products")

result.show()

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 68**

SIMPLE STATISTICS (1)

Let’s assume data is registered under view name products.

Find the **minimum** price for each product.

result = spark.sql("""SELECT product, min(price) AS min\_price FROM products GROUP BY product""")

result.show()

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 69**

SIMPLE STATISTICS (2)

Find the **average** price for each product.

result = spark.sql("""SELECT product, mean(price) AS avg\_price FROM products GROUP BY product""")

result.show()

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 70**

SIMPLE STATISTICS (3)

Find the **minimum** price **and shop** for each product.

result = spark.sql("""SELECT m.product, p.shop, m.min\_price FROM (SELECT product, min(price) AS min\_price FROM products GROUP BY product) m JOIN products p ON m.product = p.product AND m.min\_price = p.price ORDER BY m.product""")

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 71**

OUTLINE

Computing services

1. First generation batch processing: Map-reduce 2. Second generation batch processing: Spark **3. Stream processing**

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 83**

BIG DATA / BATCH PROCESSING

All data known at the time of processing

Goal: Execute computation over data and produce result

Problem: what if new data arrives continuously, and new results should be computed continuously?

**Source data** 

Batch Processing System 

(e.g. Hadoop, Spark)

**Results data**

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 84**

EXAMPLES OF BIG STREAMING DATA



Producing information on

traffic based on information

collected from users’

mobile phones

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 85**

STREAMING PROCESSING: REQUIREMENTS

Need to process data as it arrive (or at most with a very small delay)

Need to be able to process data from multiple sources Need to tolerate faults

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 86**

TWO PROCESSING MODELS (1)

Continuous

• Each tuple processed as it arrives

• Processing system may keep state for executing window computation and incremental computation

Stream Processing

System 

(e.g. Storm)

**Results**

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 87**

TWO PROCESSING MODELS (2)

Mini-batches

• Tuples received for each X ms grouped in a mini-batch • Process mini-batches

• Processing system may keep state for executing window computation and incremental computation

Stream Processing

System 

(e.g. Spark Streaming)

**Results**

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 88**

WINDOWING

When doing stream processing, it is often interesting to compute results based on data from a given interval, but compute results more frequently than the time interval — for example, process data of last 3 minutes, but produce results every minutes.

System for stream processing support the definition of sliding time windows.

E.g. In SparkStreaming, s.window(“3s”) would output results comprising the records in intervals: [0,3), [1,4), [2,5), …



0 1 2 3 4 5 6 7 8 …

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 89**

SYSTEMS FOR STREAM PROCESSING

Continuous processing

• Apache Storm

• Open sourced by Twitter

• API: proprietary, SQL-like

• Apache Flink

• API: proprietary, table-based (similar to DataFrames), SQL-like

Mini-batch processing

• Spark streaming

• API: proprietary, table-based, SQL-like

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 94**

SPARK STREAMING

**NOTE: slides with Spark Streaming intro are just for those wanting to know a little more on this topic.**

Spark Streaming is an extension of the core Spark API to enable scalable, high-throughput, fault-tolerant stream processing of live data streams.

Matei Zaharia, et. al. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In Proc. SOSP’13.

http://people.csail.mit.edu/matei/papers/2013/sosp\_spark\_streaming.pdf http://spark.apache.org/streaming/

**Cloud Computing System 21/22 – Nuno Preguiça – DI/FCT/NOVA / 95**

95