Big data distributed processing
Spark introduction

INDEX

1. History of Big Data

History of Big Data

Definitions of Big Data:

1. Lots of data! (estimated 44 zettabytes of information in 2020). @me
2. The term that describes a huge amount of data (structured and not structured) that floods the daily businesses. @SAS
3. 3, 4, 5, 7, 10 Big Data V's. @ORACLE @IBM

   Volume, Velocity, Variety, Veracity, Value, Validity, Variability, Venue, ...

History of Big Data

The most accurate definition:

Big Data refers to the systems and technologies needed to obtain, process, analyze, visualize or get value of the data that cannot be done with the previous technologies or systems due to its high volume, traffic and volatility.

BIG DATA & AI LANDSCAPE 2018

V1 – Last updated 6/19/2018

FIRSTMARK
EARLY STAGE VENTURE CAPITAL

History of Big Data

Google and the liberation of their technologies:

2003: The Google File System

- Problems with storage starts to arise.
- *We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs...*

2004: MapReduce: Simplified Data Processing on Large Clusters

- Problems with data processing starts to arise.
- *Over the past* ***five years****, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data...*

History of Big Data

(2003) The Google File System

Definition of three (four) problems to solve:

1. Servers hardware.
2. File sizes.
3. File usage.
4. Flexible applications.

History of Big Data

(2003) The Google File System

1. Servers hardware:

Problem	Solution
Hardware & Software problems Component failures Human errors	Fault tolerance: store multiple copies of the same file in different servers.
High performance servers costs	Horizontal scalability (+1k servers) *Commodity* servers (cheap and small)

(2003) The Google File System

2. File sizes:

Problem	Solution
Multiple GB files (cost of 1GB=10$ in year 2000)	Optimize the reading at the expense of writing.
Block file size: few KBs (1000s of millions of reads per file)	Change the block file size to 64-128 MBs to reduce the amount of reads per file.

(2003) The Google File System

3. File usage:

Problem	Solution
Files are only appended (historic files, audit files, intermediate calculus, ...). Sequential read.	Immutable and incremental files: *read-only*, *append-only*.

History of Big Data

(2004) MapReduce: Simplified Data Processing on Large Clusters

Distributed computing paradigm:

- Distributed computing using distributed file system.
- Functional computing is parallelizable by design.
- The system must provide:
  - Data movement administration.
  - Distributed execution management.
  - Fault tolerance.
- Clusters of *commodity* machines processing TB's of data.

History of Big Data

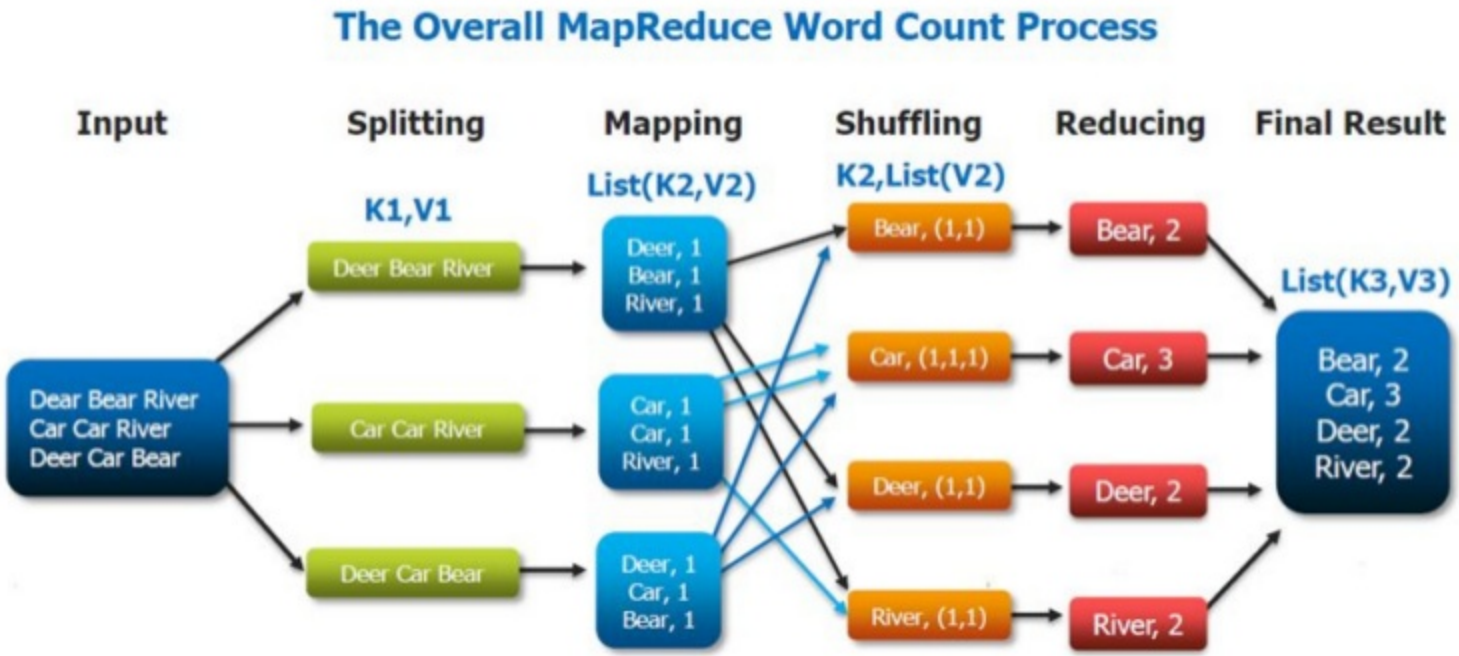
(2004) MapReduce: Simplified Data Processing on Large Clusters

Definition of two simple operations:

- **Map**: a given function is applied for each pair of key-value to generate an intermediate key-value.
- **Reduce**: a combine function groups each key to calculate the aggregation of the multiple values associated to the key.

History of Big Data

(2004) MapReduce: Simplified Data Processing on Large Clusters



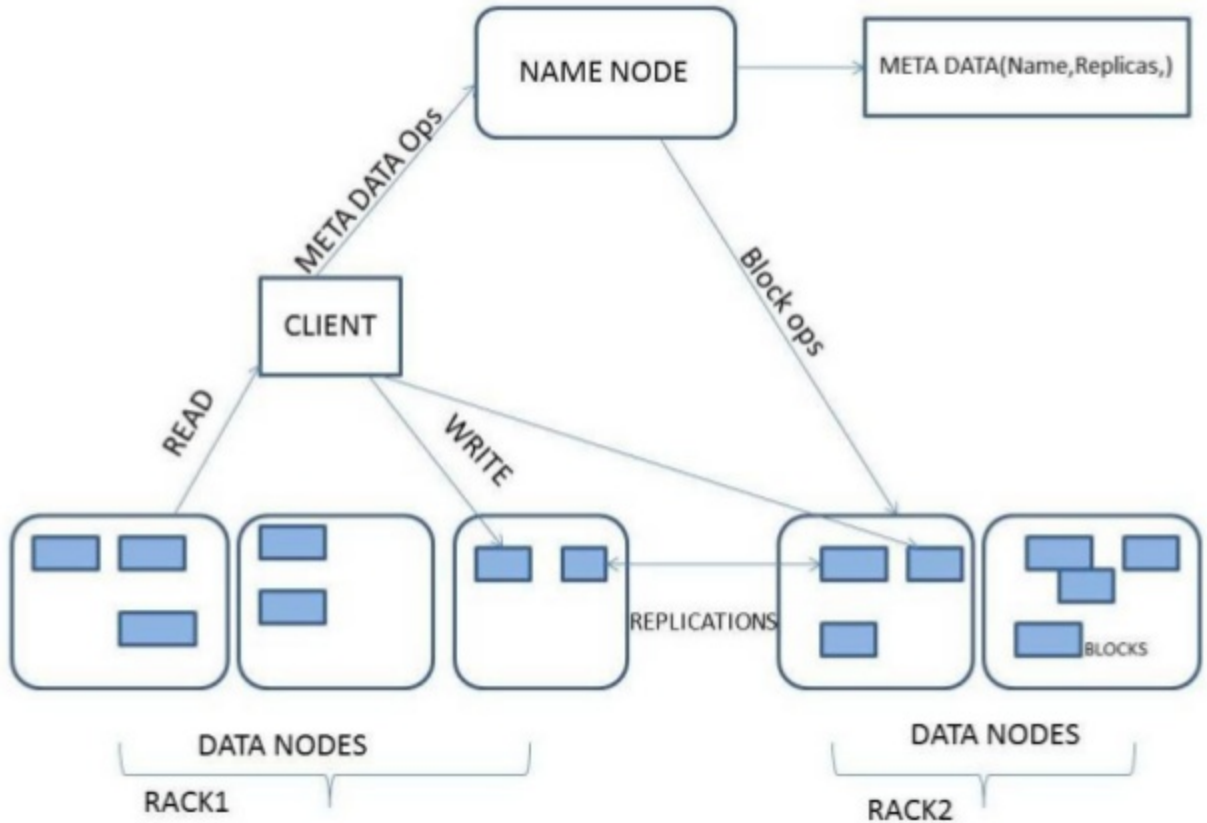
History of Big Data

Hadoop Distributed File System (HDFS)

- ~2003: started with project Nutch: search engine and crawler.
- They had problems processing data with dozens of servers.
- Google papers gave them a new approach.
- 2006: Yahoo! interested on the project and they took the distributed computing part of the software, renaming it as Hadoop.

History of Big Data

Hadoop Distributed File System (HDFS)



2. Apache Spark: Basic concepts

Apache Spark: basic concepts

- Framework for distributed data computing.
- Designed to be executed in large scale *clusters* with lots of data!
- Run faster than MapReduce (memory usage).
- More functions than just Map and Reduce.
- Multiple APIs, multiple programming languages:
  - Core, SQL, Streaming, GraphX, ML, MlLib, Structured Streaming, ...
  - Scala (native), Java, Python, R.
- Runs everywhere:
  - Standalone, YARN, Mesos, Kubernetes, AWS, ...

Apache Spark: basic concepts

- Fault tolerance (RDD).
- Easier resource managing.
- Reusable data: caching.
- Code control and analysis (DAG).
- Generic programming patterns: the same code can run in local mode or 100's of executors.
- Lazy evaluation: transformations and actions.

Apache Spark: basic concepts

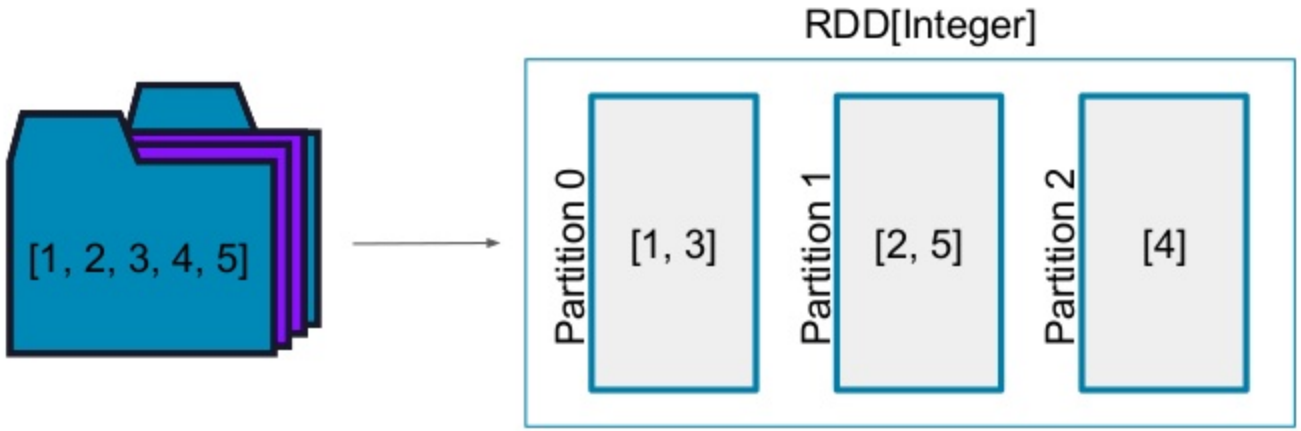
Installation

- GOTO: https://spark.apache.org/downloads.html
- Select version 2.3.2, for Hadoop 2.7 or later.
- Extract it, set SPARK_HOME and add it to PATH.
  - `tar xvf spark-2.3.2-bin-hadoop2.7.tgz`
  - Add the following lines in .bashrc:
    - `export SPARK_HOME=/your/spark/folder`
    - `export PATH=$PATH:$SPARK_HOME/bin`

Apache Spark: basic concepts

RDD

- The basic abstraction: RDD (Resilient Distributed Datasets)



Apache Spark: basic concepts

RDD

- Distributed: splitted in multiple partitions.
- Resilient: fault tolerance. Data can be reprocessed or duplicated in case of failure.
- Immutable typed elements: data cannot be modified.
- Data abstraction that allows to natively parallelize them into different executors.
- Likely to be deprecated soon.

Apache Spark: basic concepts

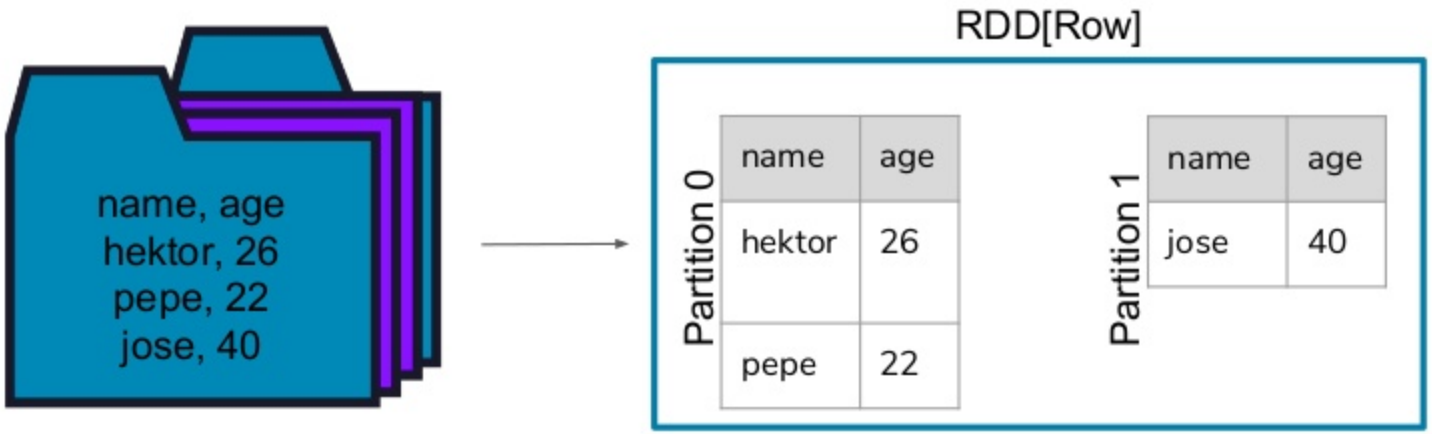
RDD

```
val numbers = Seq(1, 2, 3, 4, 5)
val numbersRDD = spark.sparkContext.parallelize(numbers)
numbersRDD: org.apache.spark.rdd.RDD[Int]
```

Apache Spark: basic concepts

Dataframe

- Distributed collection of Row objects: data organized into columns.
- Data is organized into columns.



Apache Spark: basic concepts

Dataframe

- Catalyst: powers the Dataframe and SQL APIs.
  1. Analyzing a logical plan to resolve references
  2. Logical plan optimization
  3. Physical planning
  4. Code generation to compile parts of the query to Java bytecode.

- Tungsten: provides a physical execution backend which explicitly manages memory and dynamically generates bytecode for expression evaluation.

Apache Spark: basic concepts

Dataframe

```
val rowNumbersRDD = numbersRDD.map(v => Row(v))
val schema = StructType(Seq(
    StructField("id", IntegerType, false)
))
val numbersDF = spark.createDataFrame(rowNumbersRDD, schema)
```

Apache Spark: basic concepts

Dataset

- Extension of the DataFrame API: combines the power of strong typing and lambda functions in RDD and the execution engine of the Dataframe APIs.
- Most used and powerful API.
- Encoders: allows to work with structured and unstructured data.
- The encoders provide type-safe and object-oriented programming.

Apache Spark: basic concepts

Dataset

```
val numbersDS = Seq(1, 2, 3).toDS
[...]
case class Person(name:String, age:Integer)
val personList = Seq(Person("Hektor", 26), Person("Pepe",22))
val personDS = personList.toDS
val personDS = spark.createDataset(personList)
[...]
```

Apache Spark: basic concepts

Spark operation types

- **Transformations:**
  - Operations that creates a new RDD, usually based on a previous one.
  - Does not evaluate the expression until an **action** is called.
  - Spark is able to infer the output type.
  - You can concatenate multiple transformations, before an **action**.

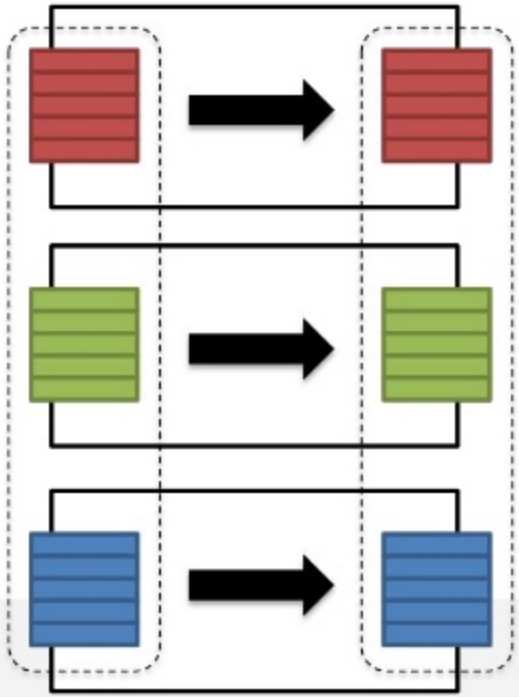
- **Actions:**
  - Operations that evaluates all the transformations defined.
  - Forces the evaluation to save or use the result data.

Apache Spark: basic concepts

Transformation types

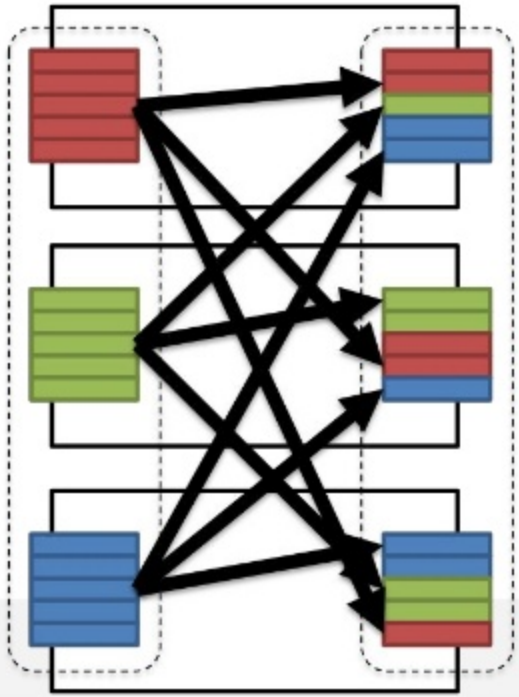
Narrow transformation

- Input and output stays in same partition
- No data movement is needed



Wide transformation

- Input from other partitions are required
- Data shuffling is needed before processing

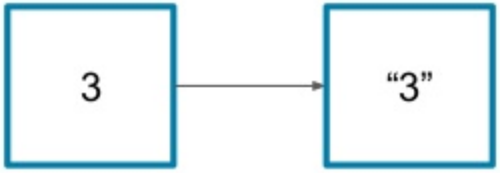


Apache Spark: basic concepts

Narrow transformations

- **Map** **map[U] (f: (T)=>U)**
  Applies the given function to all single elements. Can modify the values or return a different type.

```
val myF = (v:Int) => v + 1
val newRDD : RDD[Int] = numbersRDD.map(myF)
val strRDD : RDD[String] = numbersRDD.map(v => v.toString)
val toLong = numbersRDD.map(_.toLong)
```



Apache Spark: basic concepts

Narrow transformations

- **flatMap** **flatMap[U](f: (T) => TransversableOnce[U])**
  Applies the given function to all single elements and then flattens. Can modify the values, return a different type, return multiple values or none.

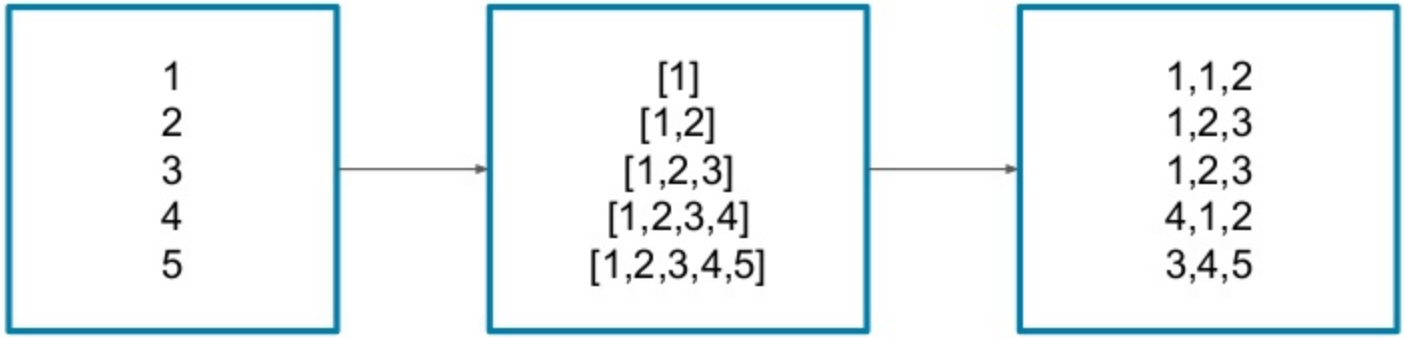
```
val myF = (v:Int) => Seq(v + 1)
val newRDD = numbersRDD.flatMap(myF)
val strRDD = numbersRDD.flatMap(v => Seq(v.toString))
val toLong = numbersRDD.flatMap(_.toLong::Nil)
```

Apache Spark: basic concepts

Narrow transformations

- **flatMap** **flatMap[U](f: (T) => *TransversableOnce*[U])**

```
numbersRDD.flatMap(v=> 1 to v)
```



Apache Spark: basic concepts

Narrow transformations

- **mapPartitions** **mapPartitions[U](f: (Iterator[T]) ⇒ Iterator[U])**
  Similar to map, but applying the function to the whole partition.

```
numbersRDD.mapPartitions(v => v.map(_ + 1))
```

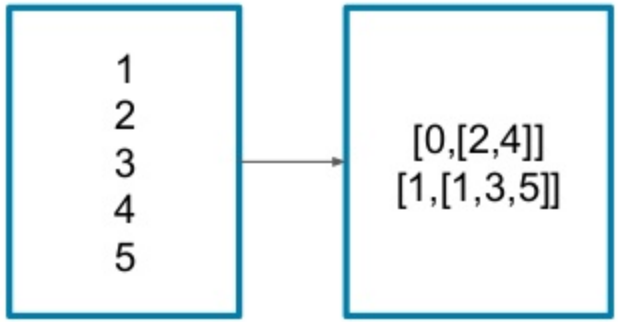
- **filter** **filter(f: (T) ⇒ Boolean)**
  Obtains a new RDD with those elements succeeded the predicate

```
val myF = (v:Int) => v%2==0
val evenNumbers = numbersRDD.filter(myF)
```

Wide transformations

- **groupBy** **groupBy[(K, Iterable[T])](f: (T) ⇒ K, p: Partitioner)**
  Obtains a new RDD grouped by key.

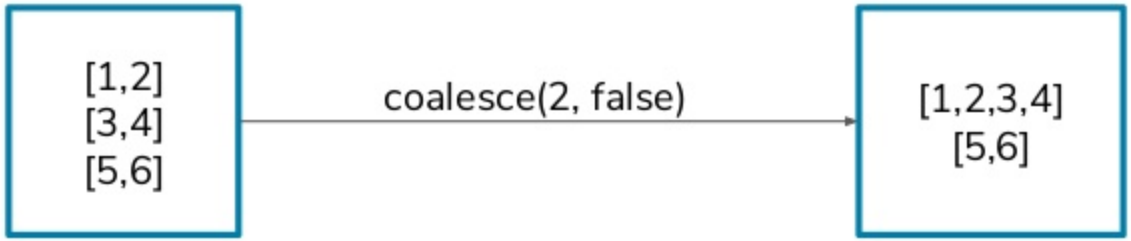
```
val myF = (v:Int) => v % 2
val groupedRDD = numbersRDD.groupBy(myF)
```



Wide transformations

- **repartition(n)**
  Rearranges the RDD to match the new number of partitions with equal size of partitions.

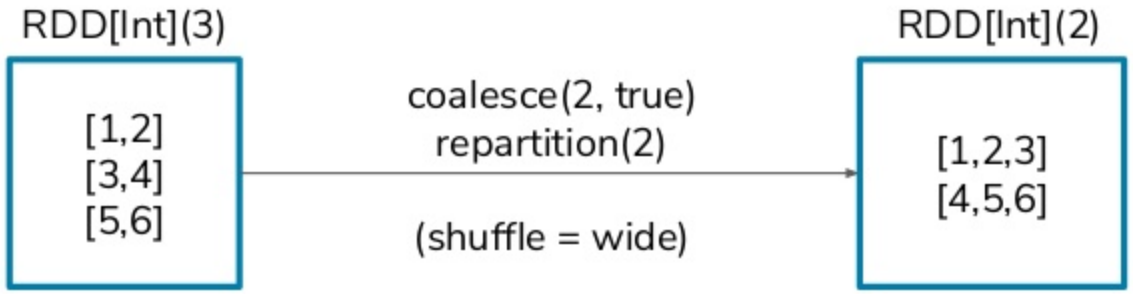
- **coalesce(n, shuffle : Boolean = false)**
  Rearranges the RDD to match the new number of partitions with equal size of partitions. If shuffle is false, you can only reduce the number of partitions and the transformation will be **narrow**.



Apache Spark: basic concepts

Wide transformations

- **coalesce(n, shuffle : Boolean = false)**



Key-value transformations

The RDD automatically cast to PairRDD when a (K,V) is detected.
New transformations are available:

```
val strings = "tomato apple apple pear tomato"

val stringsRDD = spark.sparkContext.parallelize(strings.split(" "))

val pairs = stringsRDD.map(v => (v,1))

val countPairs = pairs.reduceByKey(_ + _)
```

- **mapValues, flatMapValues, sortByKey, countByKey, foldByKey, ...**

Apache Spark: basic concepts

Key-value transformations

- **foldByKey** **foldByKey(zero: V)(f: (V, V) ⇒ V): RDD[(K, V)]**
  Applies the function for each partition using the zero value as first value.

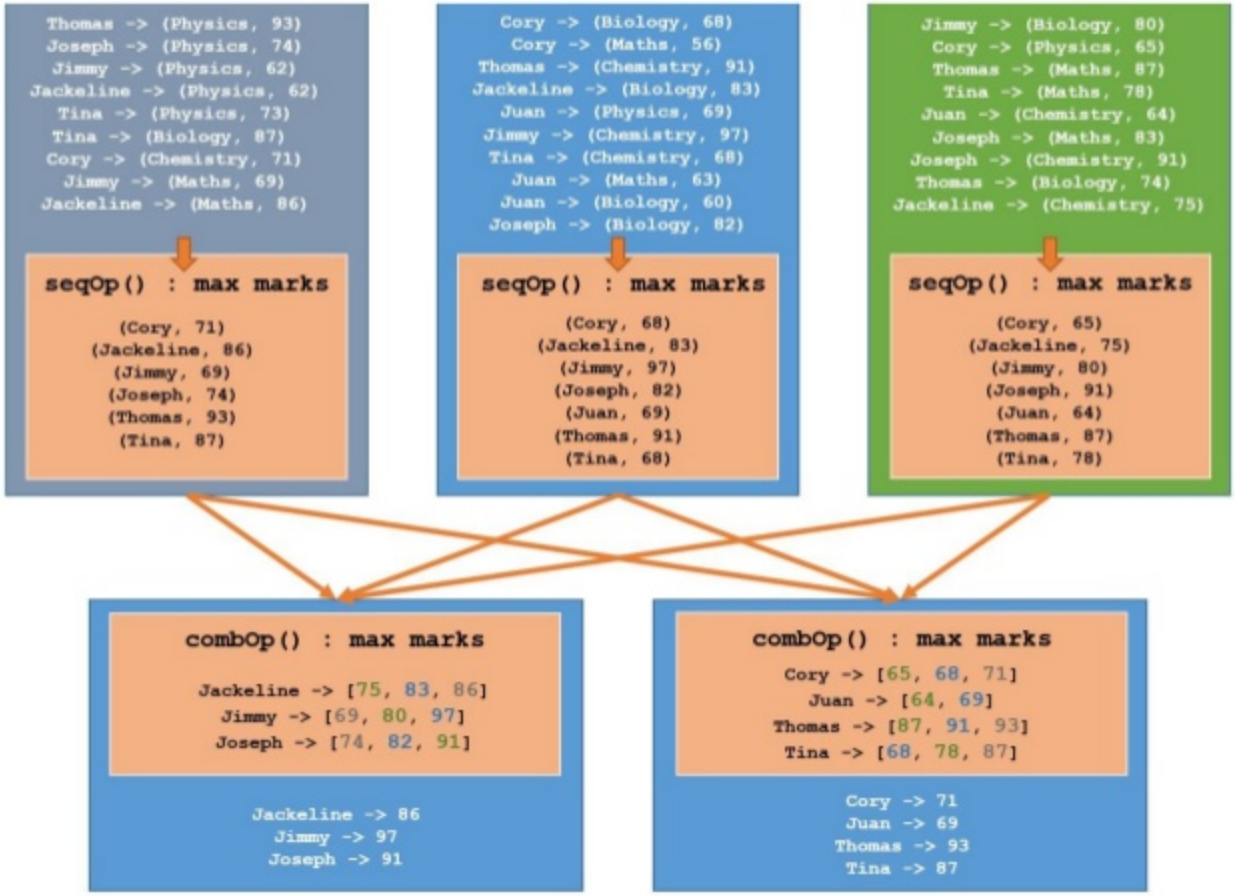
```
val sumFunc = (v1:Int, v2:Int) => v1 + v2

val countPairs = pairs.foldByKey(0)(_ + _) // or sumFunc
```

- **aggregateByKey[U](zeroValue: U)(seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U)**
  Like foldByKey, but combOp can be a different operation.

* *These functions have their RDD counterparts (without ByKey)*

Apache Spark: basic concepts

foldByKey and aggregateByKey



Apache Spark: basic concepts

Actions

Executes the current function and all the previous ones defined in the DAG.

Spark Driver data collection
These functions get the data from executors into the driver.

- **collect:** the driver obtains all the information. This is a really dangerous operation that could kill the driver if you handle huge amounts of data.
- **take(n), takeOrdered(n), first:** take the first n results to the driver.
- **count:** counts the number of elements.

Actions

Data usage or movement

- **foreach(f: [T] => Unit):** applies the function to each element. Cannot modify the values (use map instead).
  Use this function for side effects instead of map.

```
rdd.foreach { v =>
    doSomethingWithIt(v) // good behaviour
    v + 1   // this won't modify the value
}

rdd.map { v =>
    doSomethingWithIt(v) // bad behaviour
    v + 1   // this will modify the value
}
```

Actions

Data usage or movement

- **saveAs***
  Saves the RDD into multiple sinks, like Hadoop or FileSystem.

Spark's RDD API is not commonly used nowadays, and it should be only used when you cannot use Datasets API.

```
dataset.rdd().makeRDDThings.toDS()
```

Apache Spark: basic concepts

Persistence and caching

One of the most important things in Spark. This allows us to reuse intermediate results to optimize its usage.

- **cache()**:
  Persist this RDD with the default storage level (MEMORY_ONLY).

- **persist**(newLevel: StorageLevel)
  Set this RDD's storage level to persist its values across operations after the first time it is computed.

Apache Spark: basic concepts

Persistence and caching

Storage Levels:

- MEMORY_ONLY
- MEMORY_AND_DISK
- MEMORY_ONLY_SER
- MEMORY_AND_DISK_SER
- DISK_ONLY
- All of the above with _2
- OFF_HEAP

Painful RDD example

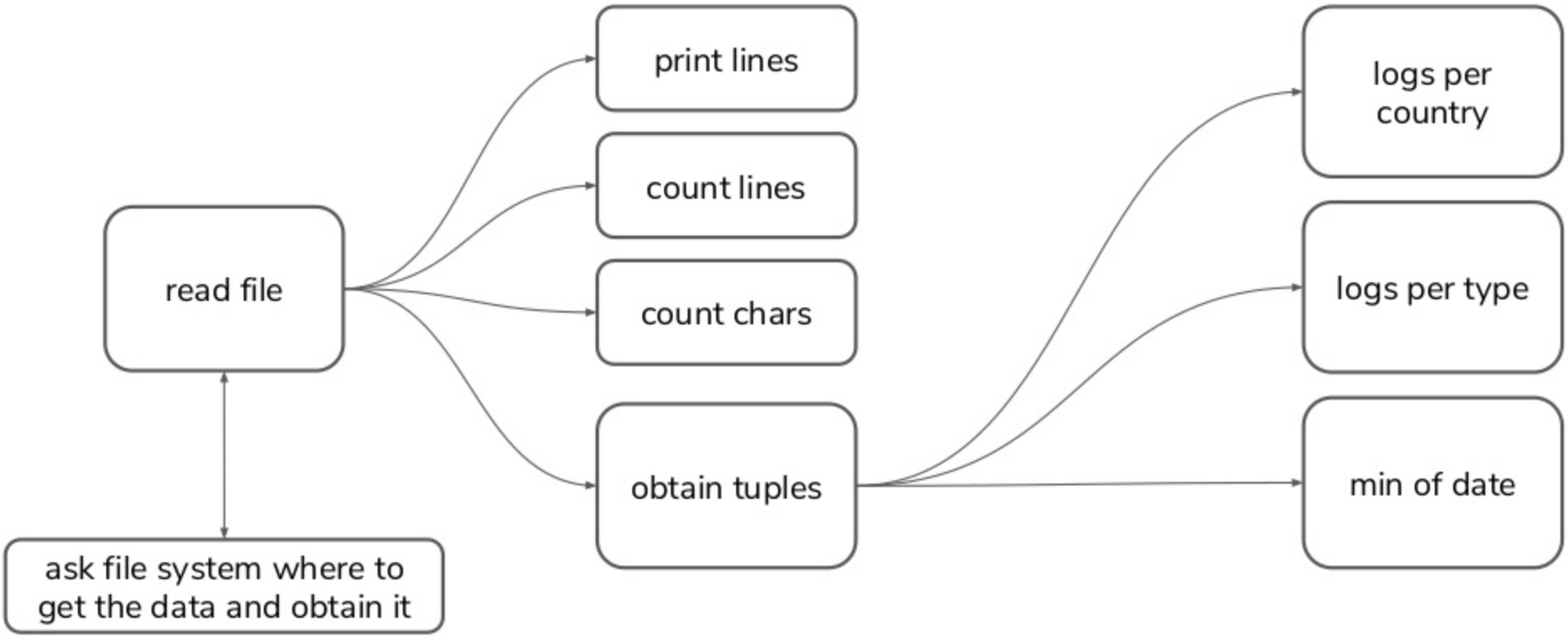
1. Read log file
2. Print total number of lines, chars and show first few lines.
3. Separate the line into a tuple and persist it.
4. Calculate the number of logs per country. Show top 3.
5. Calculate the number of logs per type.
6. Check when the errors starts happening.
7. Do the same with Datasets

Exercise

Complete the following exercise

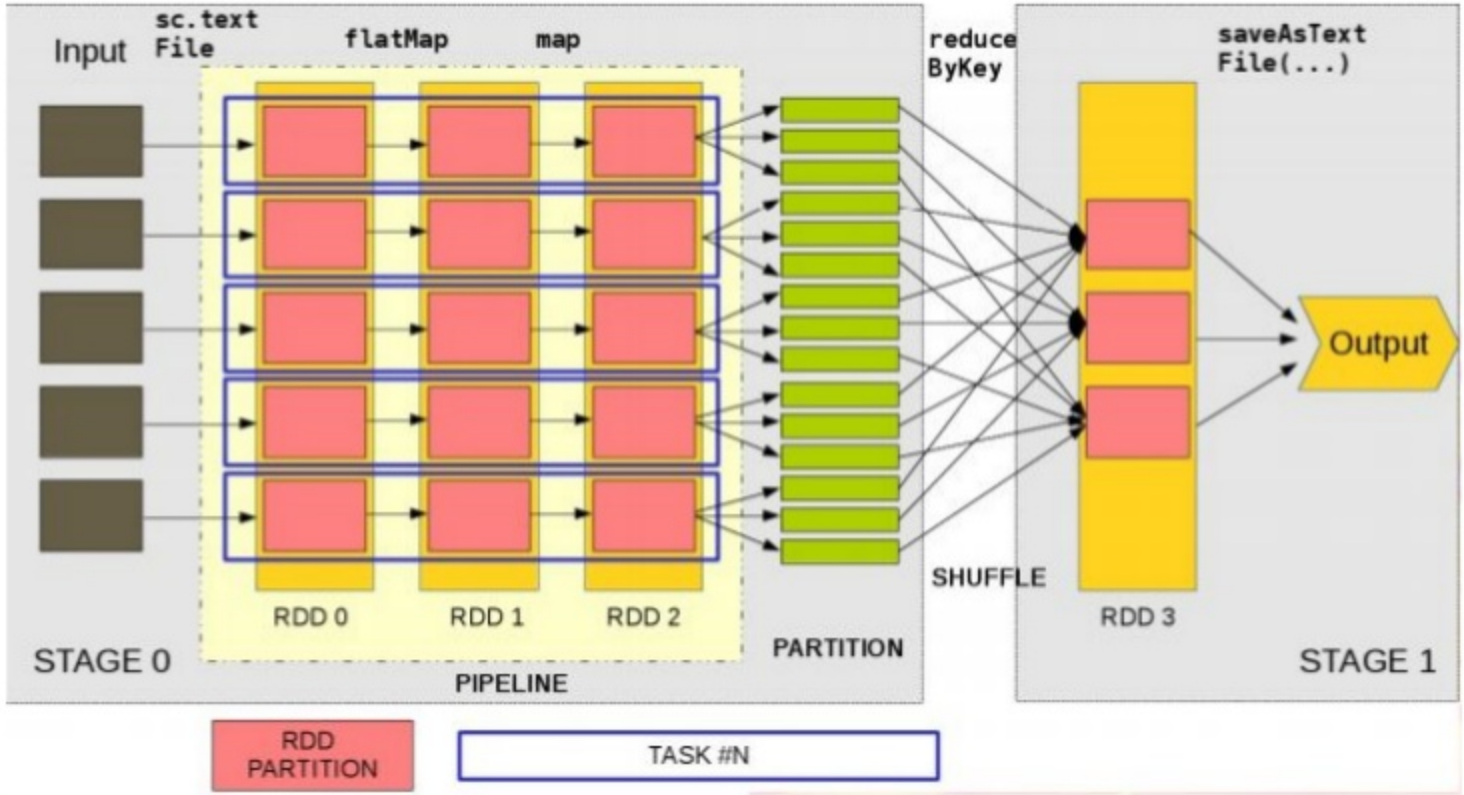
Apache Spark: basic concepts

Persistence and caching



Apache Spark: basic concepts

Wordcount example



3. Spark SQL

Datasets

Cleaner API than the RDD:

- Data inference: strong-type safe in compile time!!!
- Business intelligence easier than ever!
- Throw some SQL if you are stuck!!

```
spark.read.text("./myfile.txt").createOrReplaceTempView("data")

val data = spark.sql("SELECT * FROM data")

data.show()
```

Data usage or movement (Dataset)

- **read:** reads the data from the selected datasource.
  If the data is partitioned, Spark will have its job done!

```
spark.read.options(MapWithOptions).csv
                                  parquet
                                  json
                                  text
                                  jdbc
                                  orc
                                  [...]
```

Data usage or movement (Dataset)

- **write:** saves the data into the selected datasource.
  Each partition is saved separately, unless you coalesce or repartition first.
  The output will create parts inside the designed folder, and they can be read natively with spark reading the parent folder.

```
ds.write.options(
    Map("header"->"true", delimiter"->"|")
).csv("file:/tmp/test")
```

```
/tmp/test » ls -l
total 16
-rw-r--r-- 1 hjacynycz hjacynycz 15 dic 11 17:06 part-00000-52183aaf-d38c-4733-ad9d-37d6b39dc1e3-c000.csv
-rw-r--r-- 1 hjacynycz hjacynycz 14 dic 11 17:06 part-00001-52183aaf-d38c-4733-ad9d-37d6b39dc1e3-c000.csv
-rw-r--r-- 1 hjacynycz hjacynycz 14 dic 11 17:06 part-00002-52183aaf-d38c-4733-ad9d-37d6b39dc1e3-c000.csv
-rw-r--r-- 1 hjacynycz hjacynycz 22 dic 11 17:06 part-00003-52183aaf-d38c-4733-ad9d-37d6b39dc1e3-c000.csv
-rw-r--r-- 1 hjacynycz hjacynycz  0 dic 11 17:06 _SUCCESS
```

Dataset transformations

- **select("col1", "col2":*):** obtain the columns selected from the dataset.
- **drop("col1", "col2":*):** drop the columns selected from the dataset.
- **filter(expr):** obtains the rows that satisfies the expression.
- **union(ds):** combines with other dataset with similar schema.
- **dropDuplicates("col1", "col2":*):** drop the duplicated rows considering the columns given.
- **except(ds):** obtains the columns not seen in the other dataset.
- **join(ds, joinExprs, joinType):** the good ol' join.
- **groupBy("cols"):** groups the dataset with the given columns. You can now use aggregation functions with **agg(f1,f2,f3,...)**

[...]

Exercise

Complete the following exercise

4. Spark deployment

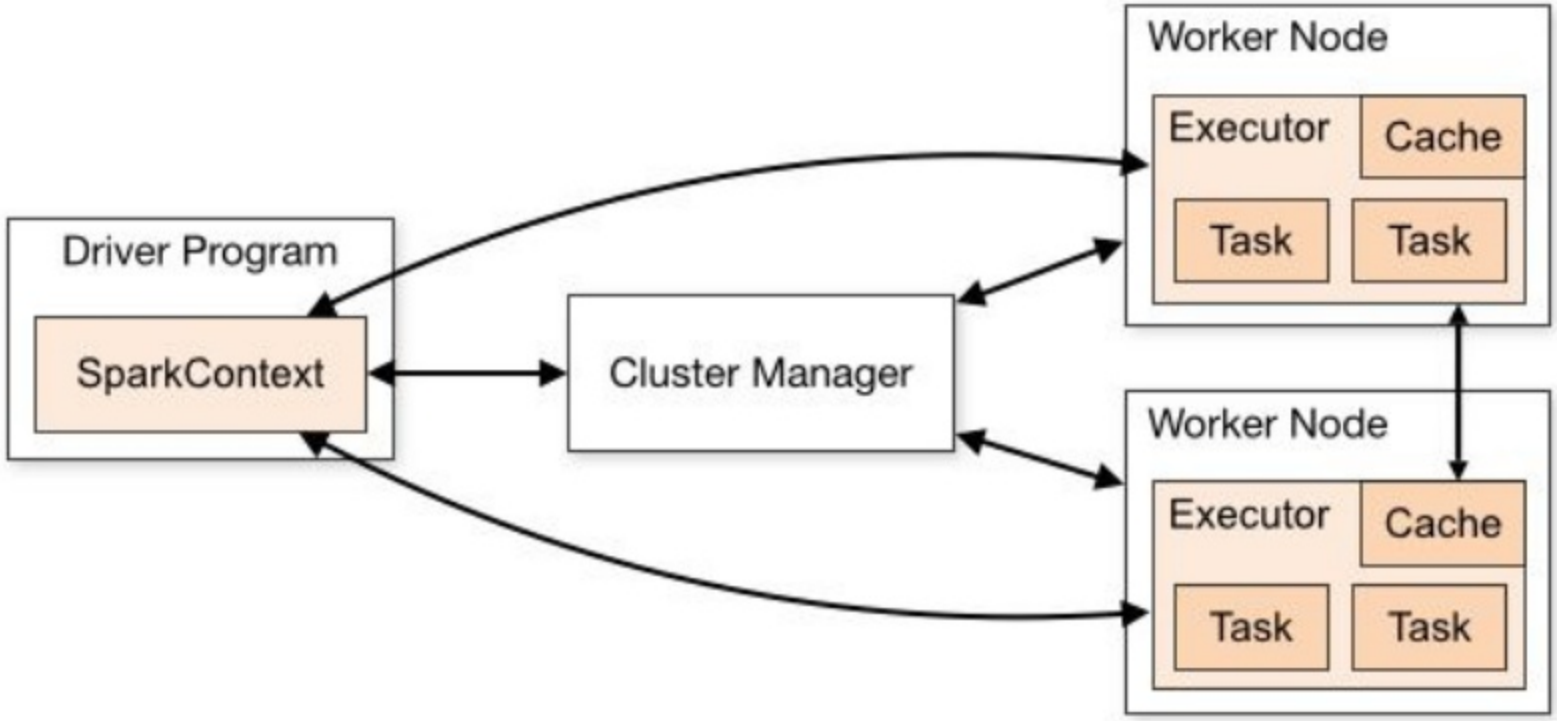
Spark deployment

Components

- **Driver**: in charge of managing the job sent to the executors.
- **Executors**: in charge of running the tasks from the job.
- **Cluster Manager**: provides connectivity and resources to the driver needs.
  - Standalone
  - Mesos
  - Yarn
  - Kubernetes (k8s)
- **SparkContext / SparkSession**: must be created to run Spark jobs. Configures the context where the jobs will be run (number and location of executors, resources, ...). Only one SparkSession can exist per JVM.

Spark deployment

Components



Spark deployment

Components

Application lifecycle

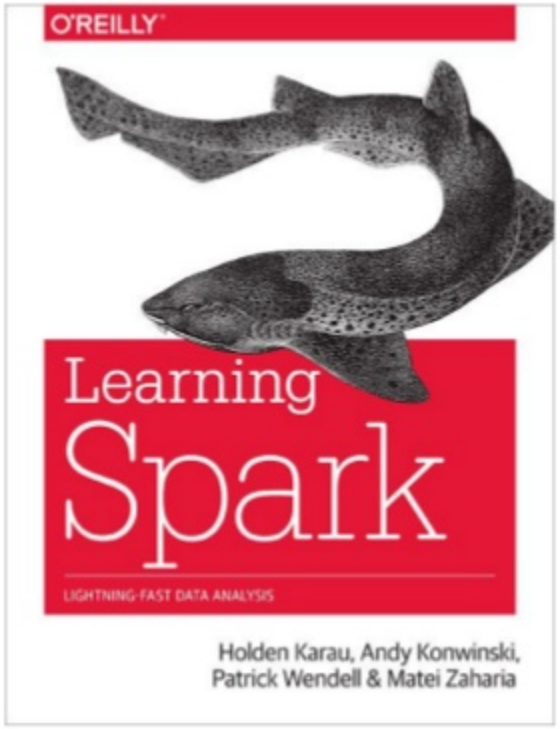
- The Spark job is packaged into a JAR, usually an “uber JAR” w/o Spark or Hadoop dependencies (they are added in runtime).
- To launch the job in the Cluster Manager varies between them. Usually, you have an endpoint where you can submit your jobs.
  `./bin/spark-submit --master spark://host:port [...]`
- This JAR is received from the Cluster Manager, who proceeds to launch a Driver program containing the JAR.
- The driver starts the job, and ask the Cluster Manager for the resources needed for the executors.
- The driver connects to the executors, sends the JAR and the executors starts receiving tasks until the job is finished.

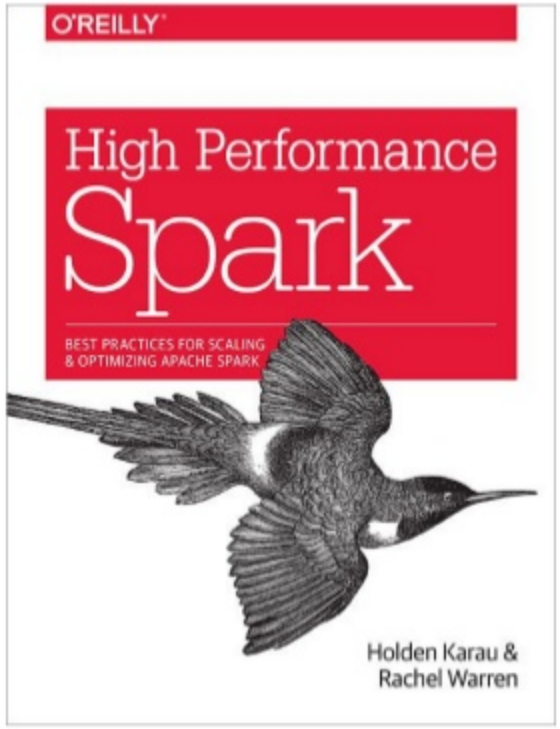
Components

Job components

- **Task:** a unit of work that will be sent to one executor.
- **Job:** a parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save, collect); you'll see this term used in the driver's logs.
- **Stage**: each job gets divided into smaller sets of tasks called stages that depend on each other (similar to the map and reduce stages in MapReduce).

Books...


O'REILLY
Learning
Spark
LIGHTNING-FAST DATA ANALYSIS
Holden Karau, Andy Konwinski,
Patrick Wendell & Matei Zaharia


O'REILLY
High Performance
Spark
BEST PRACTICES FOR SCALING
& OPTIMIZING APACHE SPARK
Holden Karau &
Rachel Warren

Nu
-ma

no-university
big data & AI

Numa es un programa de inmersión en Big Data e Inteligencia Artificial desarrollado bajo el paradigma “learning by doing”

- Personas recién graduadas, poca o nula experiencia
- Tiene una duración de 6 meses - ¡supéralos y conviértete en un Stratian!
- Formación en Big Data & IA
- Experiencia real en los equipos

numa@stratio.com

stratio.com/numa

WE ARE HIRING

people@stratio.com

@StratioBD

Thanks!

hjacynycz@stratio.com