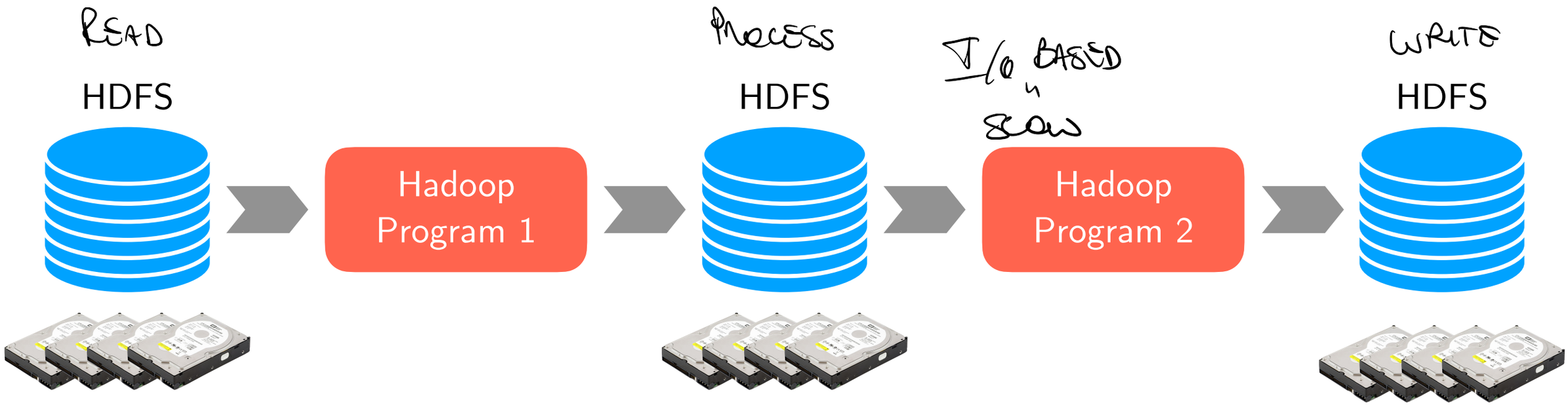
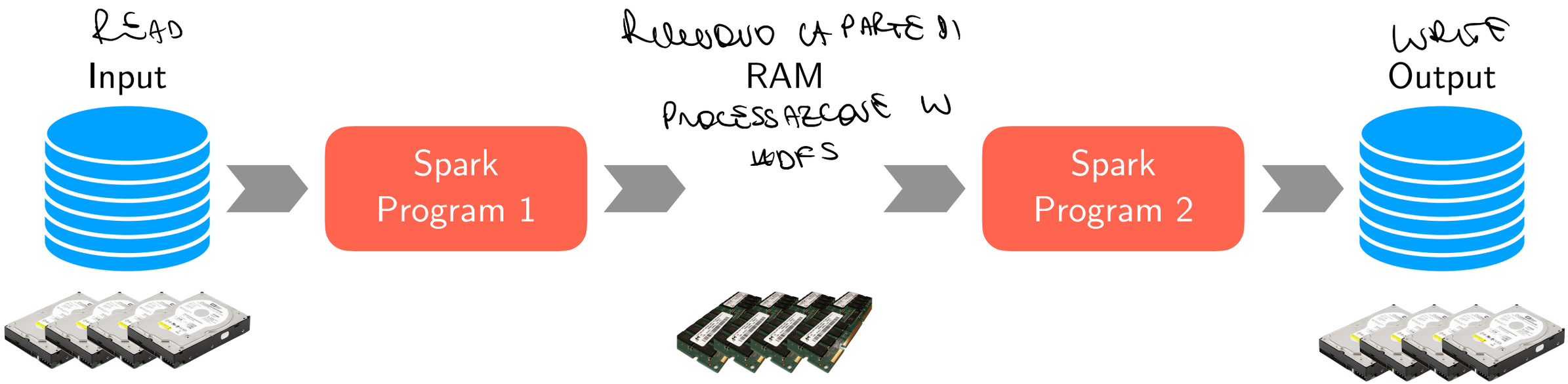
Spark

Hadoop vs Spark





CHE SUCCEDE SE PERDIAMO UNA PARTE DEI DATI?

QUANDO ABBIAMO A CHE FARE CON DATI MOLTO GRANDI CHE NON ENTRANO IN RAM?

Data Flow Models

- **Restrict** the programming interface so that the system can do more **automatically**
- Express jobs as **graphs** of high-level operators COMPUTATION → ORGANIZZATE COME NODI DI UN GRAFO
  - System chooses how to **split** each operator into tasks and **where** to run each task
  - Run parts multiple times for **fault** recovery
- Biggest example: **MapReduce**

Limitations of Map Reduce

- MapReduce is great at **one-pass** computation
- Inefficient for **multi-pass** algorithms
- No efficient primitives for **data sharing**
  - State between steps goes to distributed file system
  - Slow due to replication & disk storage
- Example: **PageRank**
  - **Repeatedly** multiply sparse matrix and vector
  - Requires **repeatedly** hashing together page adjacency lists and rank vector
- While MapReduce is simple, it can require **asymptotically more communication or I/O**
- MapReduce algorithms research doesn't go to waste: still useful to study as an **algorithmic framework**, silly to **use directly**

Spark Stack

TABLE AND DATAFRAME

STREAM OF DATA

REPRESENTATION IN SPARK OF CORE ACCORDING TO MACHINE LEARNING

Spark SQL *structured data*	Spark Streaming *realtime*	MLlib *machine learning*	GraphX *graph processing*

Spark Core

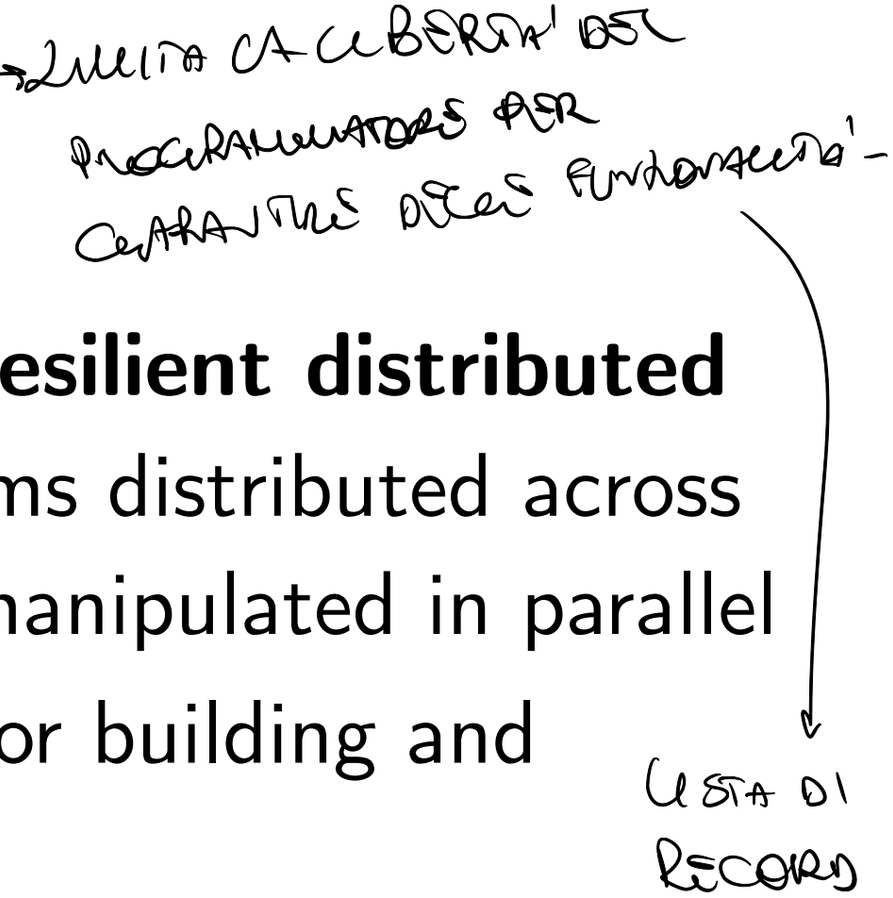
Standalone Scheduler	MESOS	YARN

↳ CLUSTER MANAGEMENT SYSTEM

STESSE FUNZIONALITÀ DI YARN -

Spark Core

- Provides **basic functionalities**, including:
  - task scheduling,
  - memory management,
  - fault recovery,
  - interacting with storage systems
- Provides a data abstraction called **resilient distributed dataset** (**RDD**), a collection of items distributed across many compute nodes that can be manipulated in parallel
  - Spark Core provides many APIs for building and manipulating these collections
- Written in **Scala** but APIs for **Java**, **Python** and **R**



Spark Modules

- **Spark SQL**
  - To work with structured data
  - Allows querying data via SQL
  - Extends the Spark RDD API
- **Spark Streaming**
  - To process live streams of data
  - Extends the Spark RDD API
- **MLlib**
  - Scalable machine learning (ML) Library
  - Many distributed algorithms: feature extraction, classification, regression, clustering, recommendation, ...
- **GraphX**
  - API for manipulating graphs and performing graph-parallel computations
  - Includes also common graph algorithms (e.g., PageRank)
  - Extends the Spark RDD API

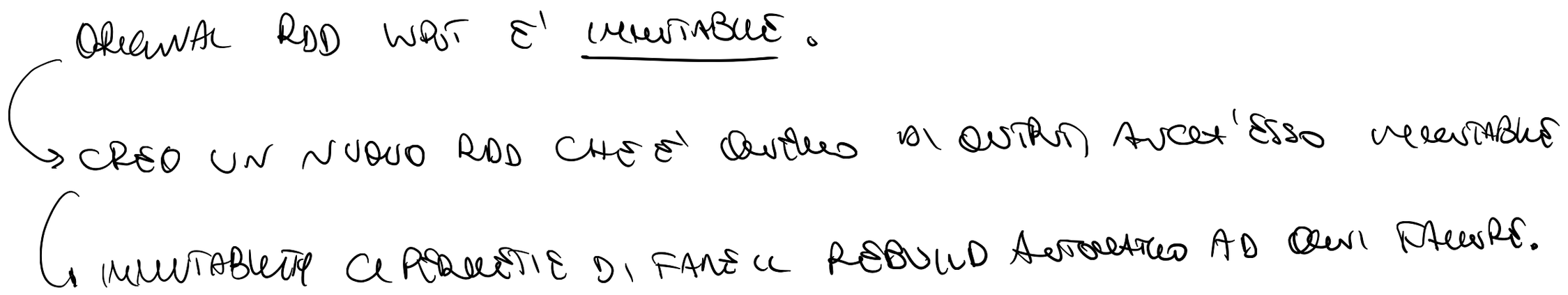
RDD (I)

- A **resilient distributed dataset** (RDD) is a distributed memory abstraction
- Immutable collection of objects spread across the cluster

- An RDD is divided into a number of **partitions**, which are atomic pieces of information
- Partitions of an RDD can be stored on different nodes of a cluster

RDD (II)

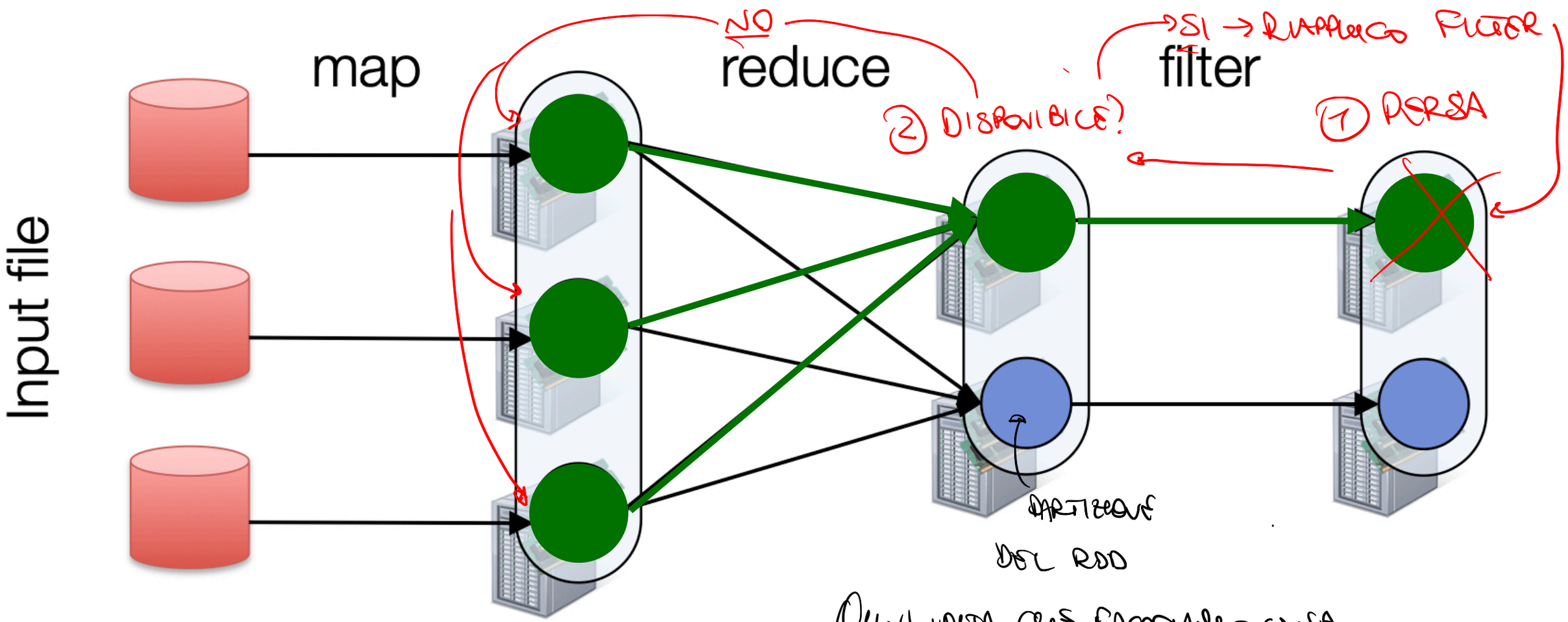
- Collections of objects across a cluster with **user controlled partitioning & storage** (memory, disk, ...)
- Built via **parallel transformations** (`map`, `filter`, ...)
- The world only lets you make RDDs such that **they can be automatically rebuilt on failure**



Lineage

```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```

IL FATTO CHE CE PARTIZIONI RDD SONO IMMUTABILI PERMETTE DI PROVARE LA PRIMA CHE TROVA PER RECUPERARE LA PERSA –

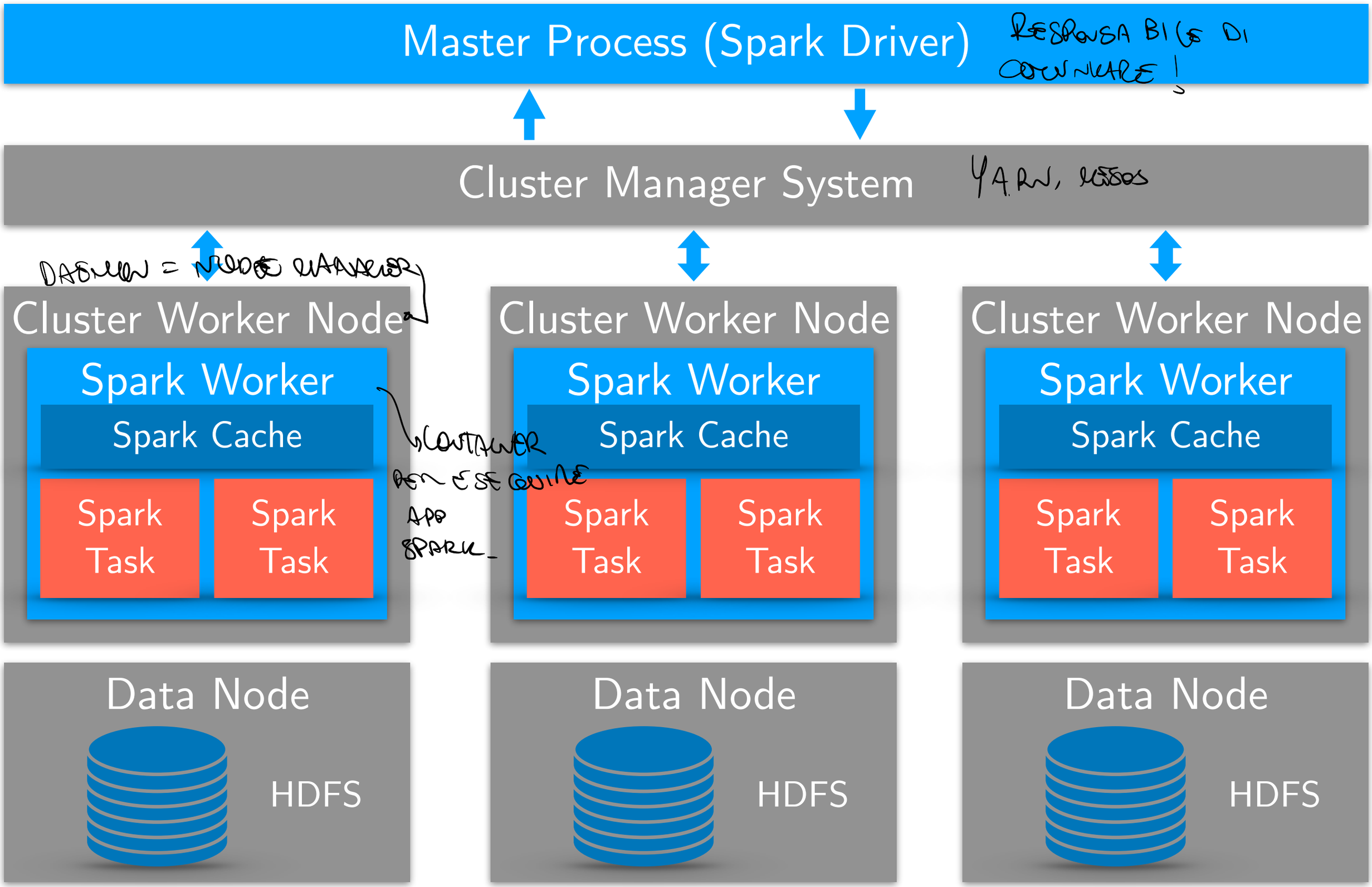


OGNI VOLTA CHE RACCOLGO UNA COLLEZIONE CREO UN NUOVO RDD –

- RDDs track **lineage** info to rebuild lost data

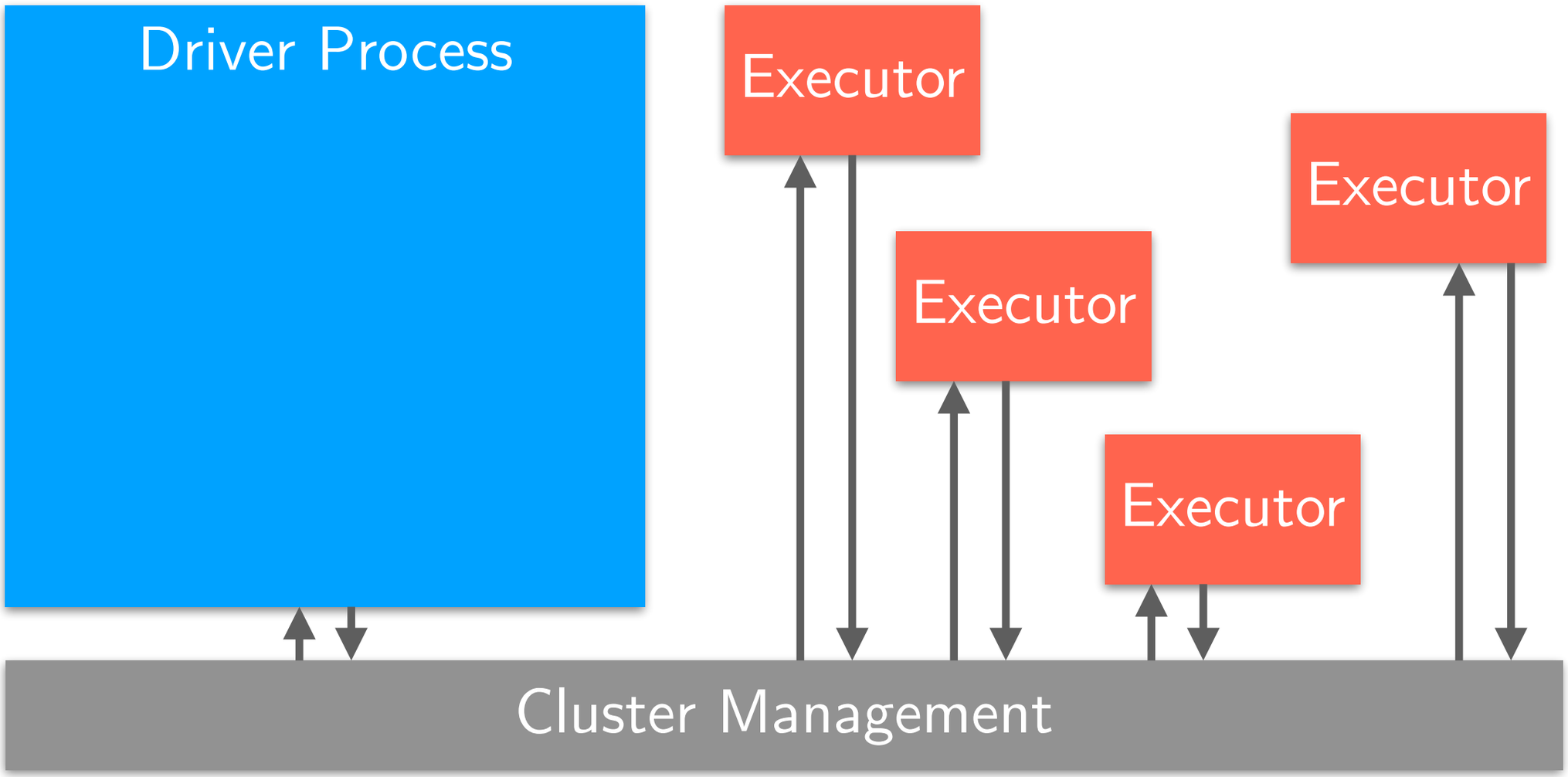
USATA PER RICOSTRUIRE UNA PARTIZIONE

Spark Architecture



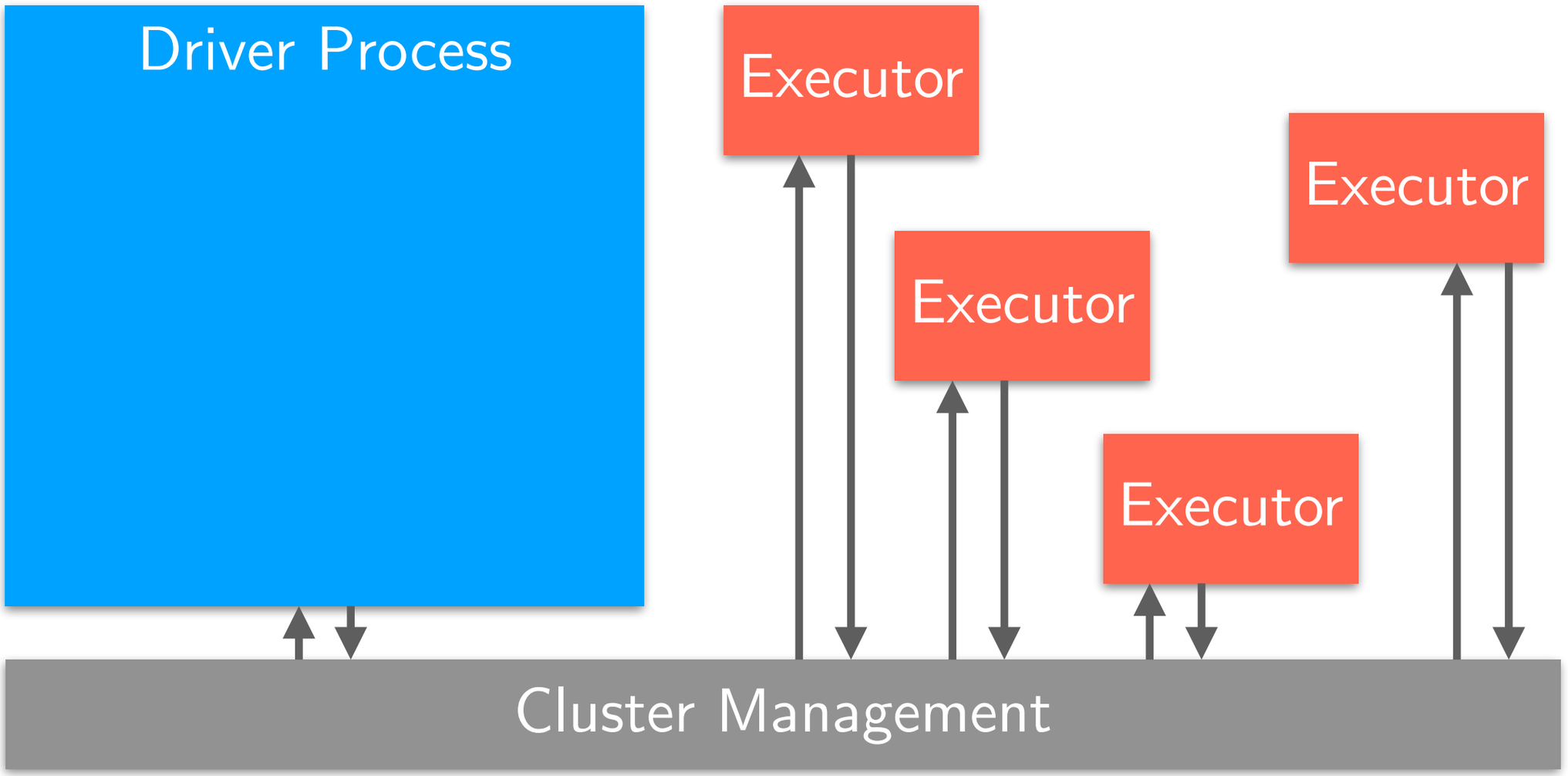
Spark Applications Architecture

- A **Spark application** consists of
  - a **driver** process
  - a **set of executor** processes



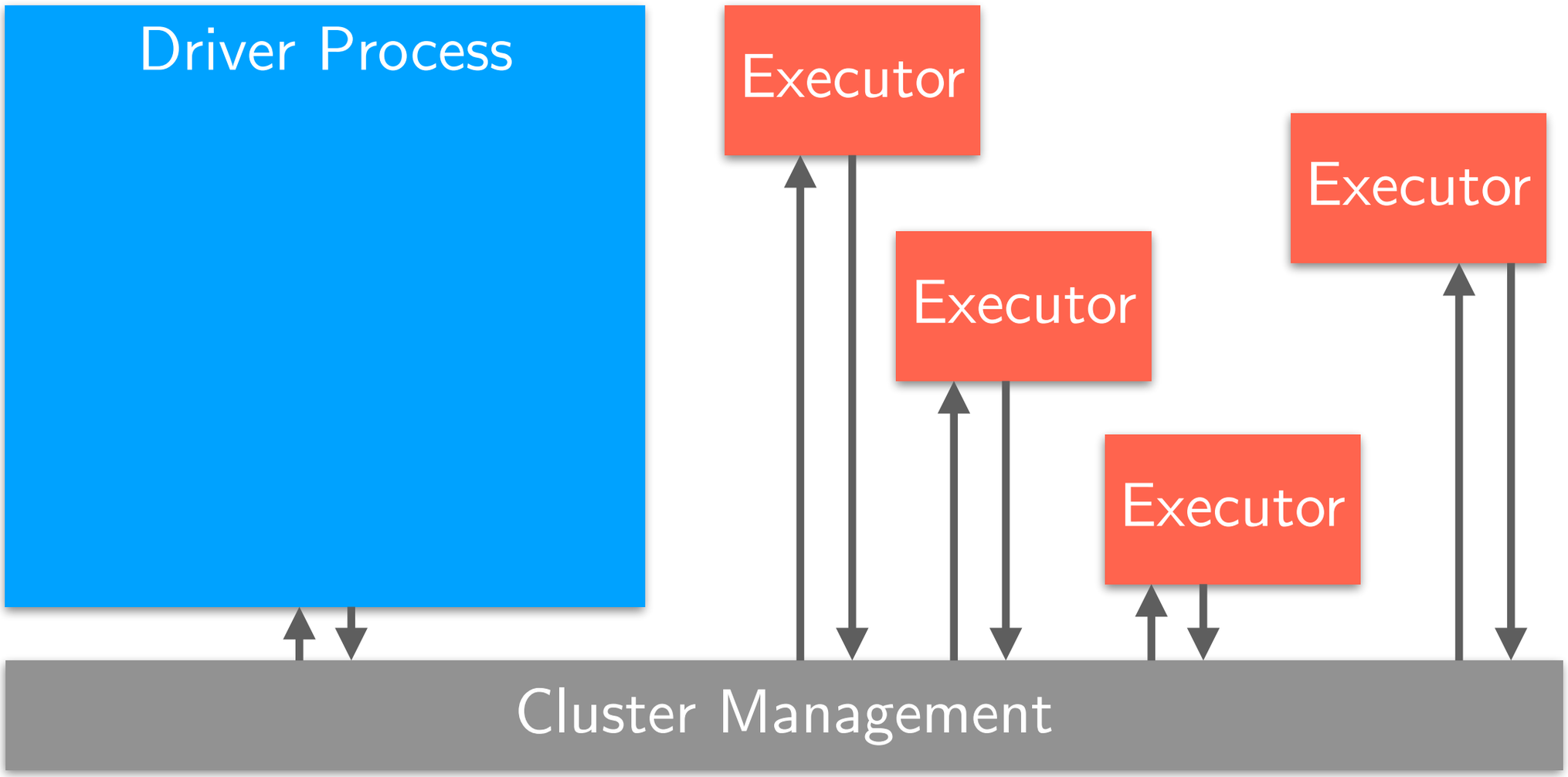
Spark Driver

- The driver process is
  - the heart of a Spark application
  - runs in a node of the cluster
  - runs the `main()` function



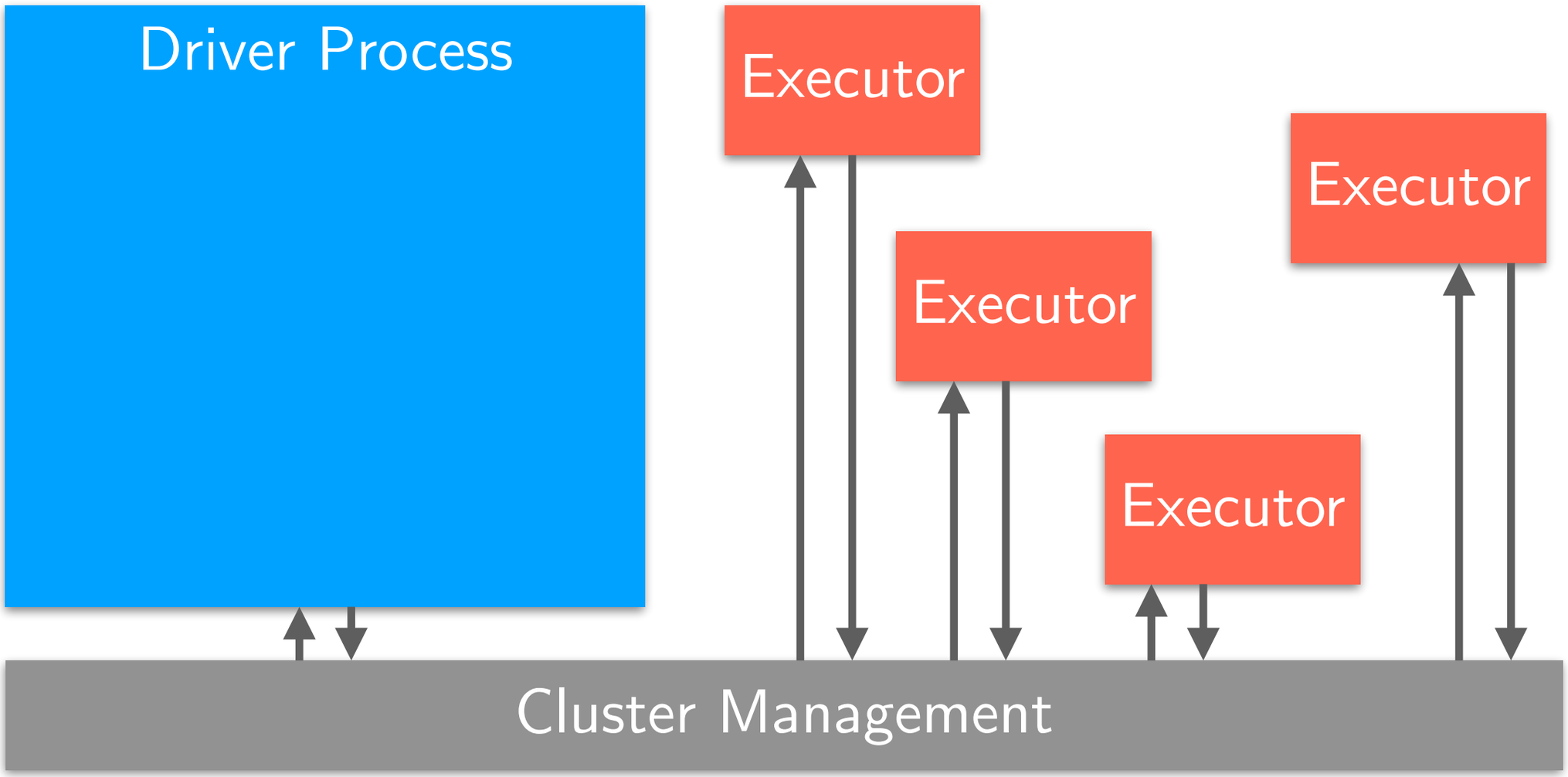
Spark Driver

- Responsible for three things:
  1. **Maintaining information** about the Spark application
  2. **Interacting** with the user
  3. **Analyzing, distributing and scheduling** work across the executors



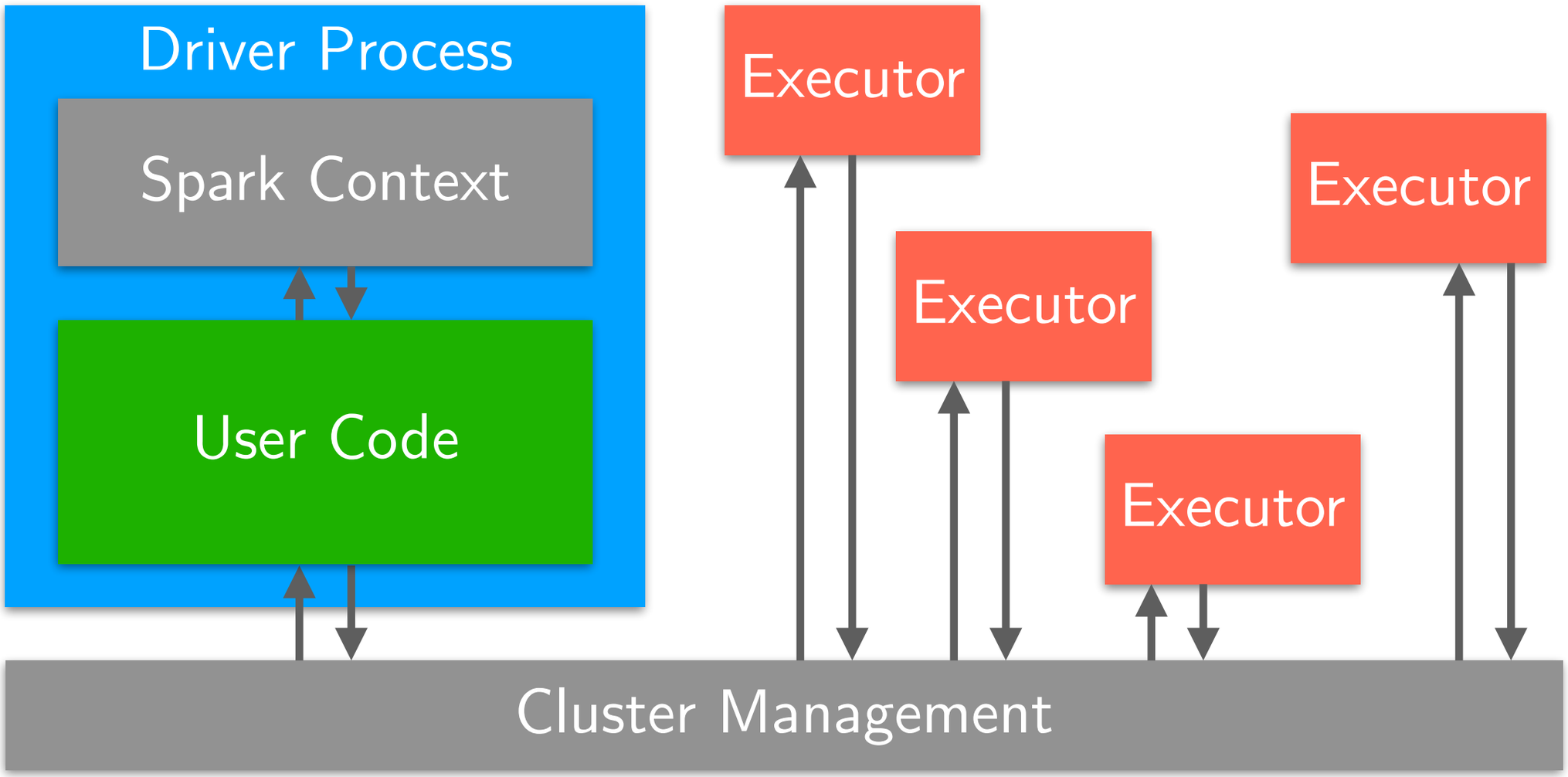
Spark Executors

- Responsible for two things:
  1. **Executing code** assigned to it by the driver
  2. **Reporting the state** of the computation on that executor back to the driver



Spark Context

- The driver process is composed by:
  - A **spark context**
  - A **user code**



Spark Context (I)

- The `SparkContext` object represents a connection with the cluster system.
- In the `pyspark` shell
  - a `SparkContext` is created automatically on start
  - It is accessible through the variable `sc`

- In a Python script including a Spark application you need to create it as soon as necessary

```
# import spark
from pyspark import SparkContext

# initialize a new Spark Context to use for the execution of the script
sc = SparkContext(appName="MY-APP-NAME", master="local[*]")
```

Spark Context (II)

- The `master` argument can assume different values, e.g.:
  - `local`: run in local mode with a single core
  - `local[N]`: run in local mode with `N` cores
  - `local[*]`: run in local mode and use as many cores as the machine has
  - **`yarn`**: connect to a YARN cluster
- Spark provides a single tool for submitting jobs across all cluster managers, called **`spark-submit`**
- We can use the **`--master`** flag to specify the execution mode of the Spark application.

Creating an RDD

- Use the parallelize method on a `SparkContext` object `sc`
- Turns a single node collection into a parallel collection.
- You can also explicitly state the number of partitions.

```
numbers = [1,2,3,4,5]
rdd_numbers = sc.parallelize(numbers)
print(rdd_numbers)

words = "i love spark".split(" ")
rdd_words = sc.parallelize(words, 2)
print(rdd_words)
```

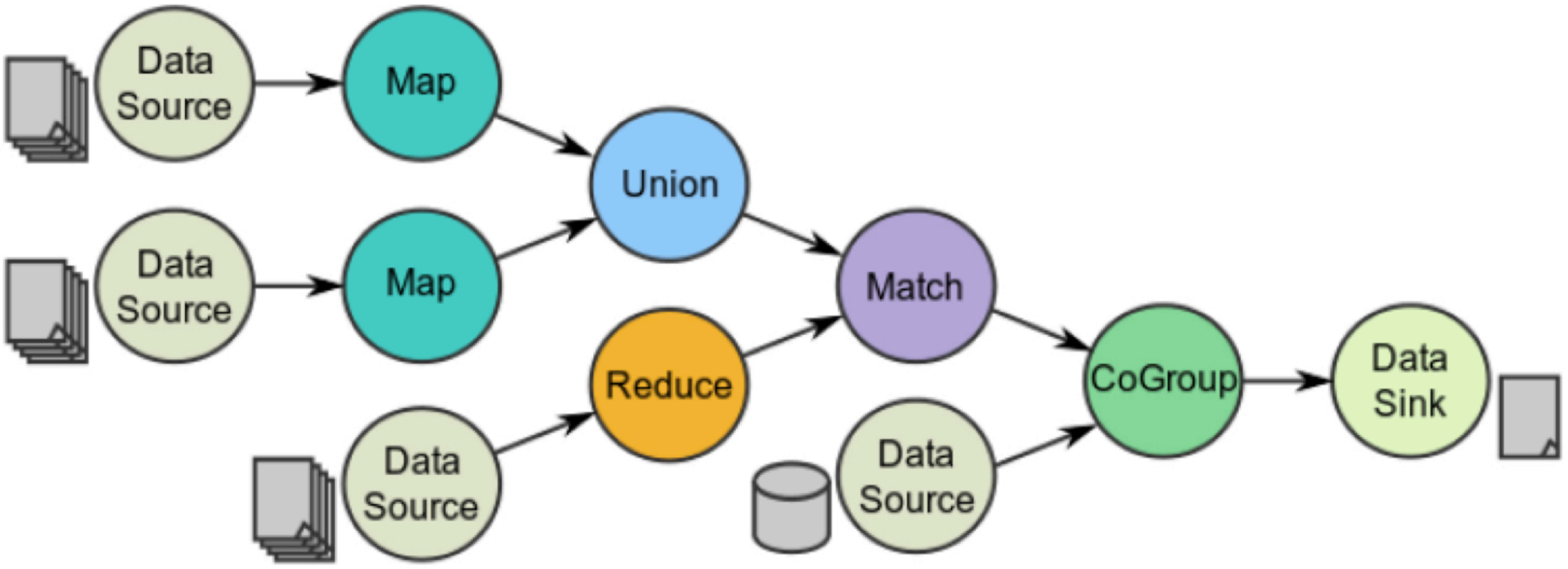
Creating an RDD

- RDDs can be created from external storage
  - Local disk, HDFS, Amazon S3, ...
- Text file RDDs can be created using the `textFile()` method

```
rdd_file = sc.textFile("file.txt")
rdd_hdfs = sc.textFile("hdfs://namenode:9000/path/to/file")
shakespeare_rdd = sc.textFile("hdfs://172.16.0.1/user/hadoop/
shakespeare.txt")
```

Spark Programming Model

- Based on **parallelizable operators**, i.e., **higher-order functions** that execute **user-defined functions** in **parallel**
- **Data flow** is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs
- Job description based on **directed acyclic graph** (DAG)



RDD Operations

- RDDs support **two types** of operations:
  1. **Transformations**: allow us to build the logical plan
  2. **Actions**: allow us to trigger the computation
- **Transformations** create a **new RDD** from an **existing RDD**.
  - **Not compute** their results right away (**lazy**).
  - **Remember** the transformations applied to the base dataset.
  - They are only computed when an action requires a result to be returned to the driver program.
- **Actions** **trigger** the computation.
  - Instruct Spark to **compute a result** from a series of transformations.
  - ? There are **three** kinds of actions:
    - Actions to **view data** in the console
    - Actions to **collect data** to native objects in the respective language
    - Actions to **write to output** data sinks

RDD Actions

Sono questo!

- `collect` returns all the elements of the RDD as an **array** at the driver
- `first` returns the first **value** in the RDD
- `take` returns an **array** with the first ***n*** elements of the RDD
  - Variations on this function: `takeOrdered` and `takeSample`
- `count` returns the **number** of elements in the dataset
- `max` and `min` return the **maximum** and **minimum** values, respectively.
- `reduce` aggregates the elements of the dataset using a given **function**.
  - The given function should be **commutative** and **associative** so that it can be computed correctly in parallel.
- `saveAsTextFile` writes the elements of an RDD as a **text file**.
  - Local filesystem, HDFS or any other Hadoop-supported file system.

RDD Actions Examples

```
numbers = sc.parallelize([1, 2, 2, 2, 1, 1, 4, 3, 3, 5, 5])
numbers.collect() # triggers execution on ALL elements, takes time
# list [1, 2, 2, 2, 1, 1, 4, 3, 3, 5, 5]
numbers.first()
# int 1
numbers.take(4) # triggers execution on 4 elements, good for debug
# list [1, 2, 2, 2]
numbers.takeOrdered(4)
# list [1, 1, 1, 2]
withReplacement = True
numberToTake = 4
randomSeed = 123456
numbers.takeSample(withReplacement, numberToTake, randomSeed)
# list [1, 5, 2, 5]
```

RDD Actions Examples

```
numbers = sc.parallelize([1, 2, 2, 2, 1, 1, 4, 3, 3, 5, 5])
numbers.count()
# int 11
numbers.countByValue()
# defaultdict(int, {1: 3, 2: 3, 4: 1, 3: 2, 5: 2})
numbers.max()
# int 5
numbers.min()
# int 1
numbers.reduce(lambda x, y: x + y)
# int 29
numbers.saveAsTextFile('numbers.txt')
# exit pyspark check contents of file 'numbers.txt'
# ls -ltrh numbers.txt
```

Generic RDD Transformations

- `distinct` removes duplicates from the RDD
- `filter` returns the RDD records that match some **predicate function**

```
numbers = sc.parallelize([1,2,2,2,3,3,4,5,5,5,5])
distinct_numbers = numbers.distinct()
print(distinct_numbers.collect()) # this is an action
[2, 4, 1, 3, 5]
even_numbers = distinct_numbers.filter(lambda x: x % 2 == 0)
print(even_numbers.collect()) # this is an action
[2, 4]
```

- `sample` draws a random sample of the data, with or without replacement

```
data = sc.parallelize(range(20))
sampled_data = data.sample(withReplacement = False, fraction = 0.20)
print(sampled_data.collect()) # this is an action
[5, 13, 14, 16]
```

Generic RDD Transformations

- `map` and `flatMap` apply a given function to each RDD element **independently**
- `map` transforms an **RDD** of **length *n*** into another RDD of **length *n***.
- `flatMap` allows returning **0, 1 or more elements** from map function.

```
data = sc.parallelize(range(10))
squared_data = data.map(lambda x: x * x)
print(squared_data.collect()) # this is an action
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

squared_cubed_data_1 = data.map(lambda x: ( x * x, x * x * x))
print(squared_cubed_data_1.collect()) # this is an action
[(0, 0), (1, 1), (4, 8), (9, 27), (16, 64), (25, 125), (36, 216), (49, 343),
(64, 512), (81, 729)]

squared_cubed_data_2 = data.flatMap(lambda x: ( x * x, x * x * x))
print(squared_cubed_data_2.collect()) # this is an action
[0, 0, 1, 1, 4, 8, 9, 27, 16, 64, 25, 125, 36, 216, 49, 343, 64, 512, 81,
729]
```

Generic RDD Transformations

- `sortBy` sorts an RDD
- `union` performs the **merging** of RDDs
- `intersection` performs the **set intersection** of RDD

```
words = sc.parallelize("nel mezzo del cammin di nostra vita".split(" "))
sorted_words = words.sortBy(lambda w: len(w))
print(sorted_words.collect()) # this is an action
['di', 'nel', 'del', 'vita', 'mezzo', 'cammin', 'nostra']

data1 = sc.parallelize(range(0,7))
data2 = sc.parallelize(range(3,10))
union = data1.union(data2)
print(union.collect()) # this is an action
[0, 1, 2, 3, 4, 5, 6, 3, 4, 5, 6, 7, 8, 9]

intersection = data1.intersection(data2)
print(intersection.collect()) # this is an action
[3, 4, 5, 6]
```

Key-Value RDD Transformations

- In a `(k,v)` pair, `k` is the **key**, and `v` is the **value**
- To create a key-value RDD:
  - `map` over your current RDD to a basic key-value structure.
  - Use the `keyBy` to create a key from the current value.
  - Use the `zip` to zip together two RDD.

```
words = sc.parallelize("nel mezzo del cammin di nostra vita".split(" "))
keywords1 = words.map(lambda w: (w.upper(), 1))
print(keywords1.collect()) # this is an action
[('NEL', 1), ('MEZZO', 1), ('DEL', 1), ('CAMMIN', 1), ('DI', 1), ('NOSTRA', 1), ('VITA',
1)]

keywords2 = words.keyBy(lambda w: w[0].upper())
print(keywords2.collect()) # this is an action
[('N', 'nel'), ('M', 'mezzo'), ('D', 'del'), ('C', 'cammin'), ('D', 'di'), ('N', 'nostra'),
('V', 'vita')]

numbers = sc.parallelize(range(7))
keywords3 = words.zip(numbers)
print(keywords3.collect()) # this is an action
[('nel', 0), ('mezzo', 1), ('del', 2), ('cammin', 3), ('di', 4), ('nostra', 5), ('vita',
```

Key-Value RDD Transformations

- `keys` and `values` extract keys and values from the RDD, respectively
- `lookup` looks up the **list of values** for a particular key in an RDD

```
words = sc.parallelize("nel mezzo del cammin di nostra vita".split(" "))
keywords = words.keyBy(lambda w: w[0])
# [('n', 'nel'), ('m', 'mezzo'), ('d', 'del'), ('c', 'cammin'), ('d', 'di'), ('n',
'nostra'), ('v', 'vita')]
k = keywords.keys()
# ['n', 'm', 'd', 'c', 'd', 'n', 'v']
v = keywords.values()
# ['nel', 'mezzo', 'del', 'cammin', 'di', 'nostra', 'vita']
look = keywords.lookup("n")
print(look)
['nel', 'nostra']
```

Key-Value RDD Transformations

- `reduceByKey` combines values with the same key
  - Takes a **function** as input and uses it to **combine values** of the same key
- `sortByKey` returns an RDD sorted by the key

```
words = sc.parallelize("fare o non fare non esiste provare".split(" "))
wordcount = words.map(lambda w: (w, 1)).reduceByKey(lambda x, y: x + y)
print(wordcount.collect()) # this is an action
[('provare', 1), ('fare', 2), ('non', 2), ('esiste', 1), ('o', 1)]

sorted_wordcount = wordcount.sortByKey()
print(sorted_wordcount.collect()) # this is an action
[('esiste', 1), ('fare', 2), ('non', 2), ('o', 1), ('provare', 1)]
```

Key-Value RDD Transformations

- `join` performs an inner-join on the key
- Other types of join:
  - `fullOuterJoin`
  - `leftOuterJoin, rightOuterJoin`
  - `cartesian`

```
cars = sc.parallelize(["Ferrari", "Porsche", "Mercedes"])
colors = sc.parallelize(["red", "black", "pink"])
joined = cars.cartesian(colors)
print(joined.collect())
[('Ferrari', 'red'), ('Ferrari', 'black'), ('Ferrari', 'pink'), ('Porsche',
'red'), ('Porsche', 'black'), ('Porsche', 'pink'), ('Mercedes', 'red'),
('Mercedes', 'black'), ('Mercedes', 'pink')]

cars = sc.parallelize([(1,"Ferrari"), (1, "Porsche"), (2, "Mercedes")])
colors = sc.parallelize([(1, "red"), (2, "black"), (3, "pink")])
joined = cars.join(colors)
 print(joined.collect())
```

Word Count (I)

1. Load "`comedies.txt`" text file into Spark
2. Transform the lines RDD into a words RDD
3. Transform each `word` into a `(word, 1)` pair
4. Reduce words by key to sum up word occurrences
5. Save results as text file

Word Count (II)

```
text    = sc.textFile("data/comedies.txt")
words   = text.flatMap(lambda x: x.split(" "))
ones    = words.map(lambda w: ( w, 1 ))
counts = ones.reduceByKey(lambda x, y: x + y)
counts.saveAsTextFile("data/comedies_wordcount.txt")
```

Word Count (III)

```
import sys

from pyspark import SparkContext


if __name__ == "__main__":

    master = "local"

    if len(sys.argv) == 2:

        master = sys.argv[1]

    sc = SparkContext(master, "WordCount")

    lines = sc.parallelize(["pandas", "i like pandas"])

    result = lines.flatMap(lambda x: x.split(" ")).countByValue()

    for key, value in result.items():

        print("%s %i" % (key, value))
```

Bigram Count (I)

1. Define a function extracting all bigrams from a string of words.
2. Load "`comedies.txt`" text file into Spark
3. Transform the lines RDD into a bigrams RDD
4. Transform each bigram into a (bigram, 1) pair
5. Reduce bigrams by key to sum up bigram occurrences
6. Save results as text file

Bigram Count (II)

```
def create_bigrams(line):
    pairs = []
    words = line.lower().split()
    for i in range(len(words) -1):
        pairs.append(words[i] + "_" + words[i + 1])
    return pairs

text = sc.textFile("data/comedies.txt")
bigrams = text.flatMap(create_bigrams)
ones = bigrams.map(lambda b: ( b, 1 ))
counts = ones.reduceByKey(lambda x, y: x + y)
counts.saveAsTextFile("data/comedies_bigramscount.txt")
```

RDD Persistence (I)

- By default, each transformed RDD may be **recomputed** each time an action is run on it
- Spark also supports the **persistence** (or **caching**) of RDDs in memory across operations for rapid reuse
  - When RDD is persisted, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it)
  - This allows future actions to be **much faster** (even 100x)
  - To persist RDD, use `persist()` or `cache()` methods on it
  - Spark's cache is **fault-tolerant**: a lost RDD partition is automatically recomputed using the transformations that originally created it
- Key tool for **iterative algorithms** and fast **interactive use**

RDD DI 10 ELEMENTI

RDD Persistence (II)

- Using `persist()` you can specify the storage level for persisting an RDD
- Storage levels for `persist()`: `MEMORY_ONLY`, `MEMORY_AND_DISK`, `DISK_ONLY`, ...
- Calling `cache()` is the same as calling `persist()` with the default storage level (`MEMORY_ONLY`)
- Which storage level is **best**? Few things to consider:
  - Try to **keep in-memory as much as possible**
  - **Serialization** make the objects **much more space-efficient**
    - But select a fast serialization library
  - **Try not to spill to disk** unless the functions that computed your datasets are expensive (e.g., filter a large amount of the data)
  - Use replicated storage levels only if you want **fast fault recovery**

Some issues

- Iterative or single jobs with large global variables → AD ESEMPIO UN DIZIONARIO
  - Sending a large lookup table to workers
  - Sending a large feature vector in a ML algorithm to workers
- Counting events that occur during job execution
  - How many input lines were blank?
  - How many input records were corrupt?
- Problems
  - Closure → FUNZIONI CON IL SUO STACK are (re-)sent with every job
  - Inefficient to send large data to each worker
  - Closures are one-way: from driver to workers

POSSIAMO CACHARE CLOSURE E VARIABILI RICEVUTE?

Distributed Shared Variables

NORMAL PYTHON VARIABLES QUE VENGATE PER ESSERE USATE IN SPARK

- In addition to the Resilient Distributed Dataset (RDD) interface, in Spark there are **two** types of **distributed shared variables**:
  - **broadcast variables**: let you save a value on all the worker nodes and reuse it across many Spark actions without re-sending it to the cluster
  - **accumulators**: let you add together data from all the tasks into a shared result
- These are variables you can use in your user-defined functions (e.g., in a map function on an RDD) that have special properties when running on a cluster

Broadcast Variables

- Efficiently send large, read-only values to all workers
  - Saved at workers for use in one or more Spark operations
  - Like sending a large, read-only lookup table to all workers
- Keep read-only variables cached on workers
  - Ship to each worker only once instead with every task

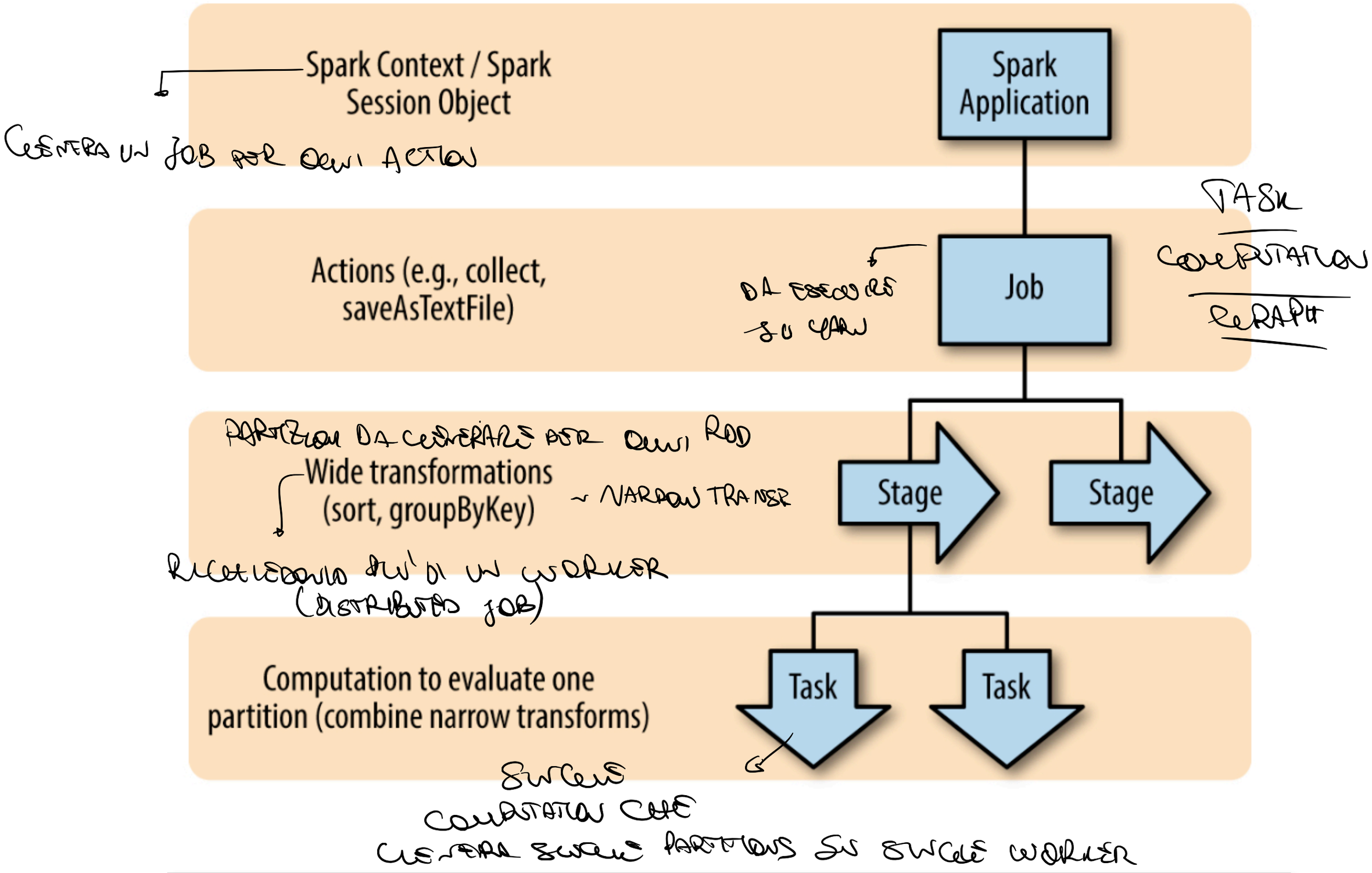
```
# At driver

broadcastVar = sc.broadcast([1, 2, 3])

# At workers (in a closure)

print(broadcastVar.value)

>> [1, 2, 3]
```

Accumulators

- Update a value inside of a variety of transformations and propagating that value to the driver node in an **efficient** and **fault-tolerant** way.
- Accumulators are variables that are "added" to only through an **associative** and **commutative** operation and can therefore be efficiently supported in parallel.
- **Updated only once** that RDD is **actually computed** (lazy evaluation in transformations)
- Tasks at workers see accumulators as **write-only variables**

```
# At driver

accumulator = spark.sparkContext.accumulator(0)

# At workers (in a closure)

accumulator.add(3)

# At driver

print(accumulator.value)

>> 3
```

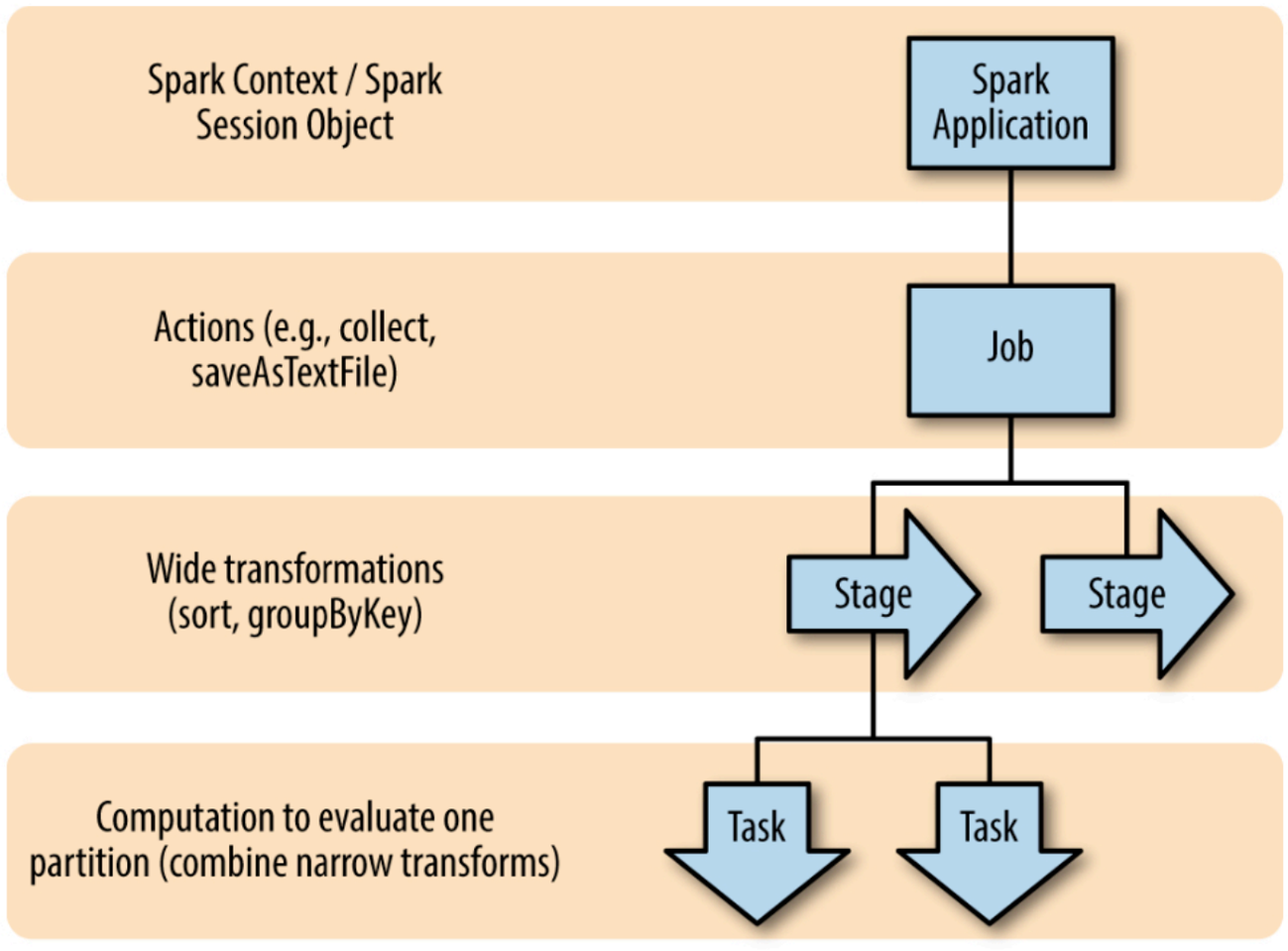
Anatomy of a Spark Job



Source: Karau et al., "High Performance Spark", O'Reilly, 2017

Spark application

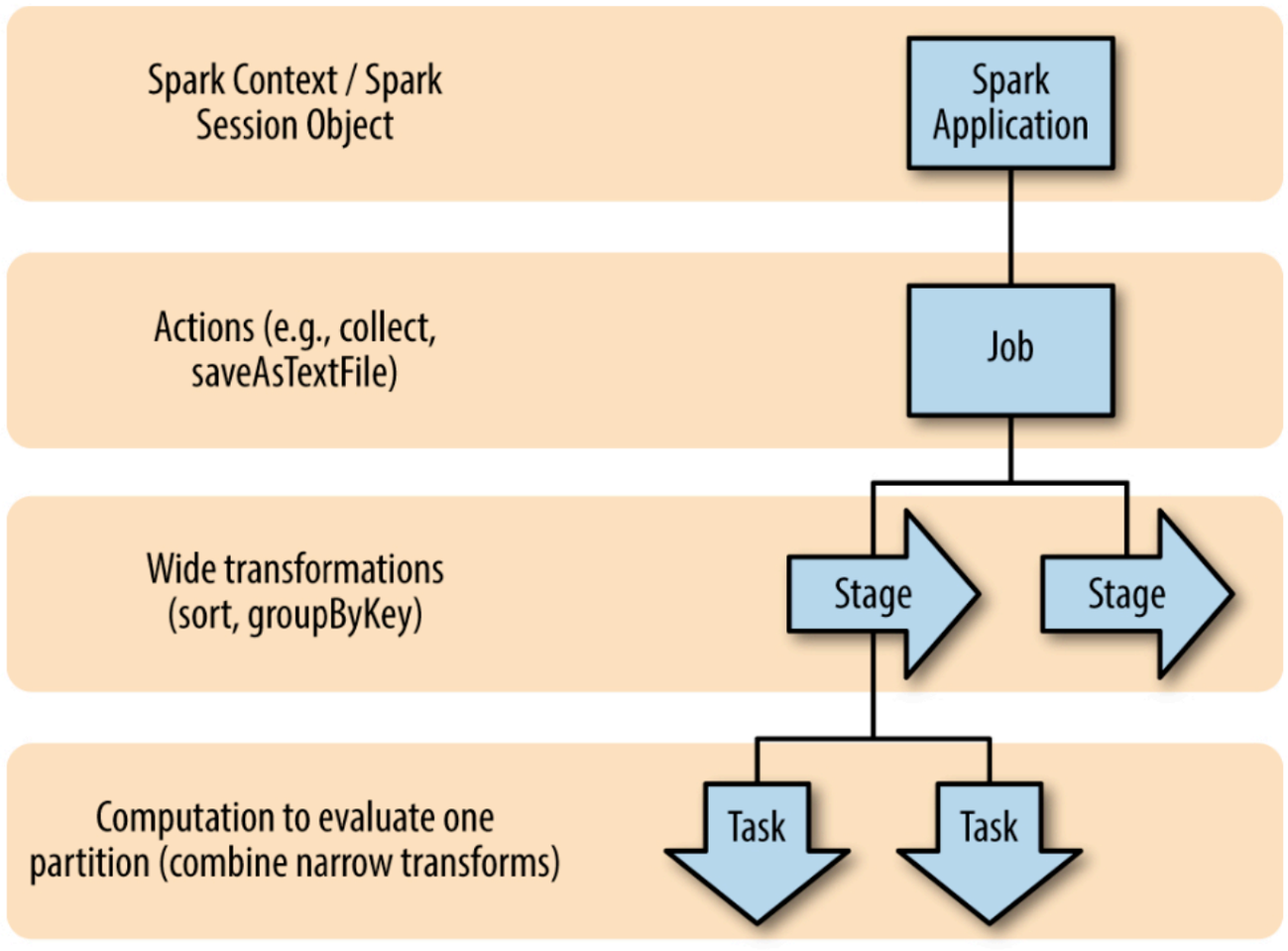
- A **Spark application** doesn't "do anything" until the driver program calls an action (**lazy evaluation**)



Source: Karau et al., "High Performance Spark", O'Reilly, 2017

Spark Job

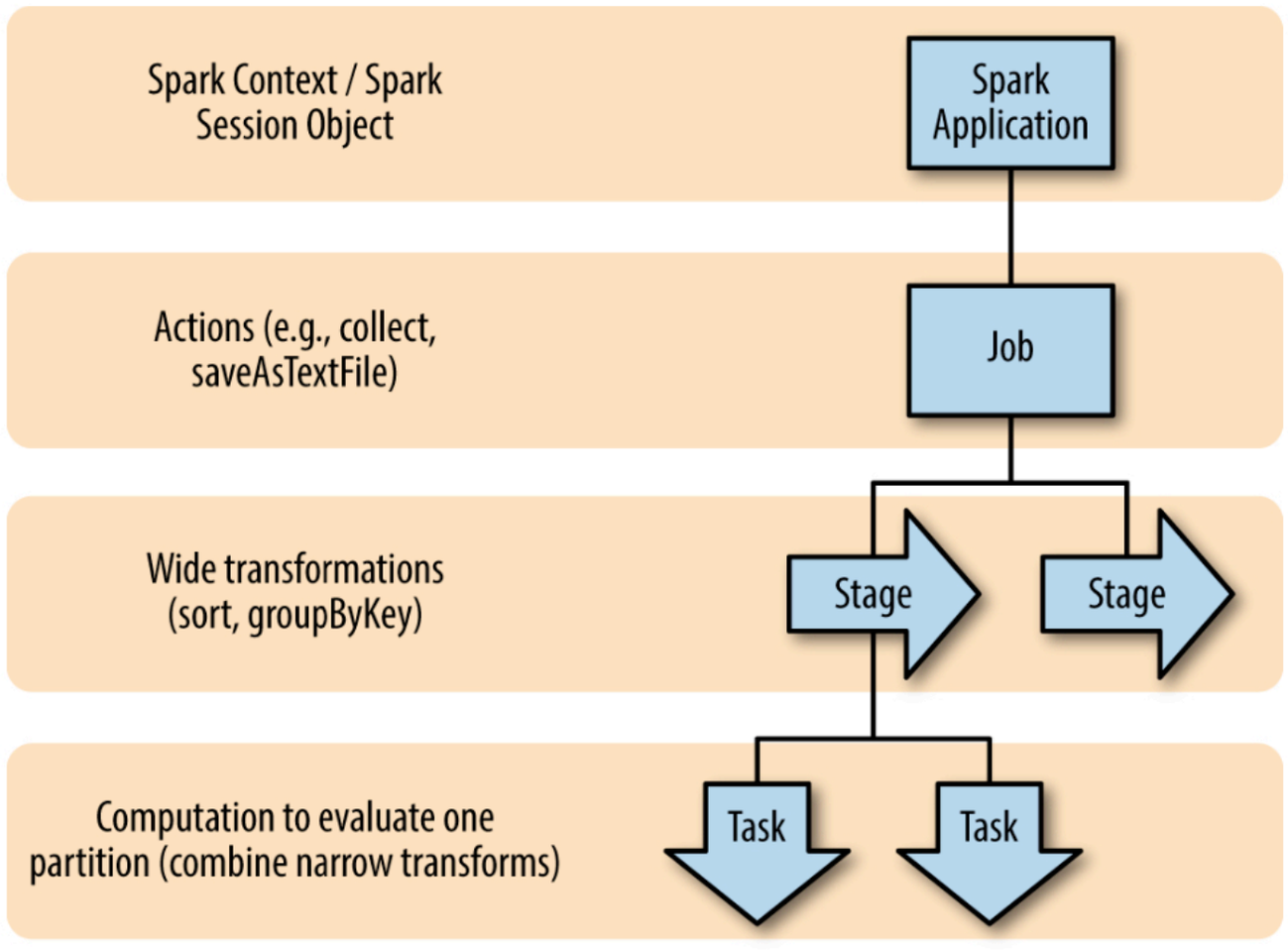
- A **Spark job** is the highest element of Spark's execution hierarchy
  - Each Spark job corresponds to **one action**
  - Each **action** is called by the **driver** program of a Spark application



Source: Karau et al., "High Performance Spark", O'Reilly, 2017

Spark Stage

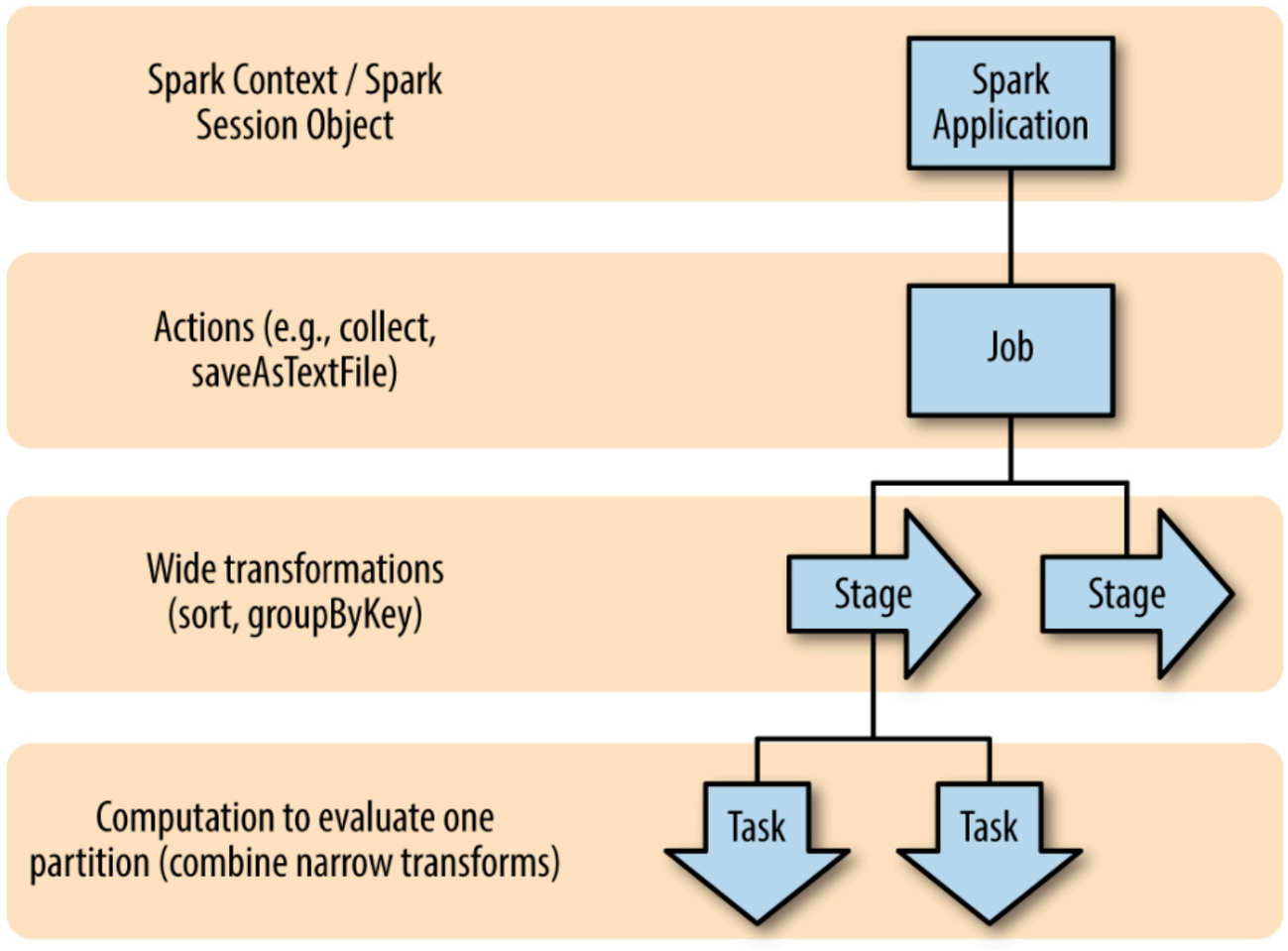
- Each **job** breaks down into a **series of stages**
  - Stages in Spark represent **groups of tasks** that can be executed **together**
  - **Wide transformations** define the breakdown of jobs into stages



Source: Karau et al., "High Performance Spark", O'Reilly, 2017

Spark Task

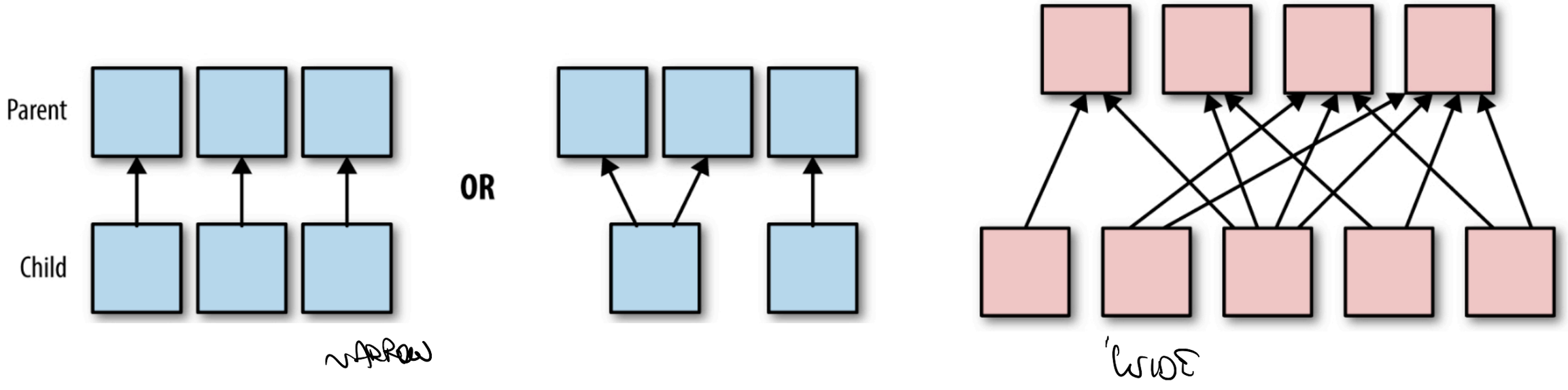
- A **stage** consists of **tasks**, which are the **smallest execution unit**
  - Each task represents **one local computation**
  - All of the tasks in one stage execute the **same code** on a **different piece of the data**



Source: Karau et al., "High Performance Spark", O'Reilly, 2017

Wide and Narrow Transformations

- Transformations fall into **two categories**: transformations with narrow dependencies and transformations with wide dependencies
- **Narrow transformations**
  - Dependencies can be determined at **design time**, irrespective of the values of the records in the parent partitions, and if **each parent has at most one child partition**
  - Can be executed on an arbitrary subset of the data without any information about the other partitions.
- **Wide transformations (shuffle)**
  - Cannot be executed on arbitrary rows and instead **require the data to be partitioned in a particular way**, e.g., according the value of their key



Developing Spark Applications

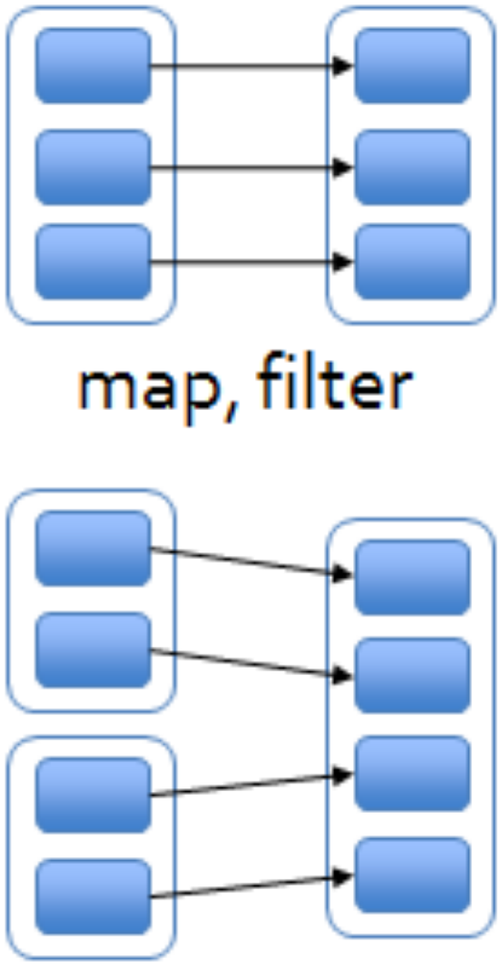
Life of a Spark Application

(DATA DOPO)

1. **Create** some input RDDs from external data or parallelize a collection in your driver program.
2. **Lazily transform** them to define new RDDs using transformations like `filter()` or `map()`
3. Ask Spark to `cache()` any intermediate RDDs that will need to be **reused**.
4. **Launch actions** such as `count()` and `collect()` to kick off a parallel computation, which is then optimized and executed by Spark.

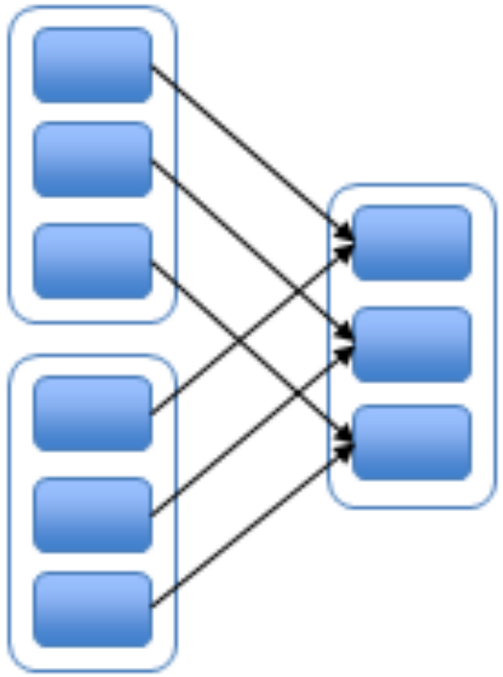
Communication Costs

"Narrow" deps:



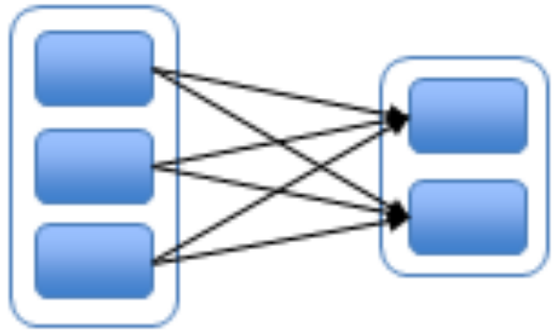
map, filter

union



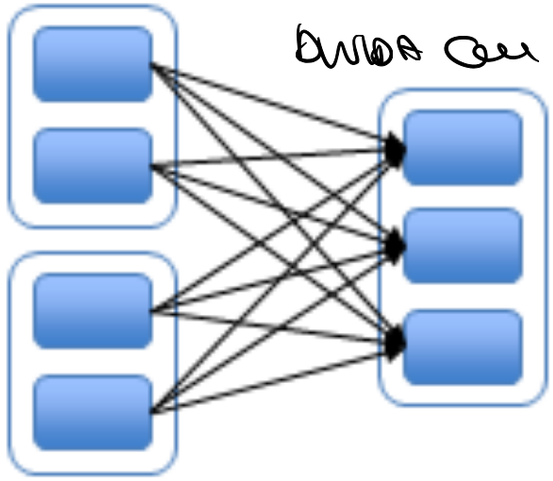
join with inputs co-partitioned

"Wide" (shuffle) deps:



groupByKey

ARRIVO BLOCCO DI QUALCOSA CHE DEVO CAMBIARE



join with inputs not co-partitioned

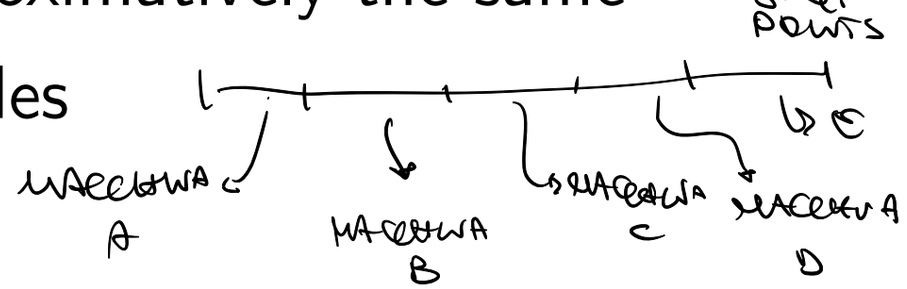
COME FACCIAMO A FARLO LO SHUFFLE DI NUOVO EFFICIENTE?

Shuffling

- **Shuffling** requires to solve the **sorting problem**, defined as follows:
  - There is **some ordering** in the data that we want to sort
  - The data **starts** out **distributed** on p machines
  - **All-to-all** communication is unavoidable
  - The best we can hope for is that the data are **sent** over the network **only once**
  - The **sorted** data set is stored in a **distributed** manner

Distributed Sorting

- Each machine sends a **uniform sample** of its data to the sorting driver
- The driver uses the **samples** from each machine **weighted** by the number of data to compute an **approximate distribution** of the data to be sorted
- This distribution is used to compute p **split points** where the number of data between two consecutive points is approximatively the same
- The split points are **broadcasted** to the nodes
- Each machine does a **local sort** on its data
- Using the split points, each machine builds an **index** of which data goes to which machine
- Machine i asks machine j for its portion of data
  - **All-to-all** communication occurs in this step
- Each machine does a ***p*-way merge** of p different sorted data sets that is has received from each machine



Uniform Sampling (I)

- A stream σ is a sequence of elements $(s_1, ..., s_m)$ where each $s_i \in \{1, ..., n\}$. Typically, the length of the stream is **unknown** and n is large but **n << m**.
- **Uniform Sampling**: given a stream σ, for each $i \in \{1,..., n\}$, define f_i as the number of occurrences of value i in the stream. We can define a probability distribution P on $\{1, ..., n\}$ by $p_i = f_i/m$. The goal is to output a single element sampled in accordance to the distribution P.
- For example, consider the stream σ = (1, 3, 4, 5, 5, 2, 1, 1, 1, 7). In this case $p_1 = 4/10$, $p_2 = 1/10$, $p_3 = 1/10$, $p_4 = 1/10$, $p_5 = 2/10$, $p_7 = 1/10$. The goal would be to output a random variable in accordance to this distribution.

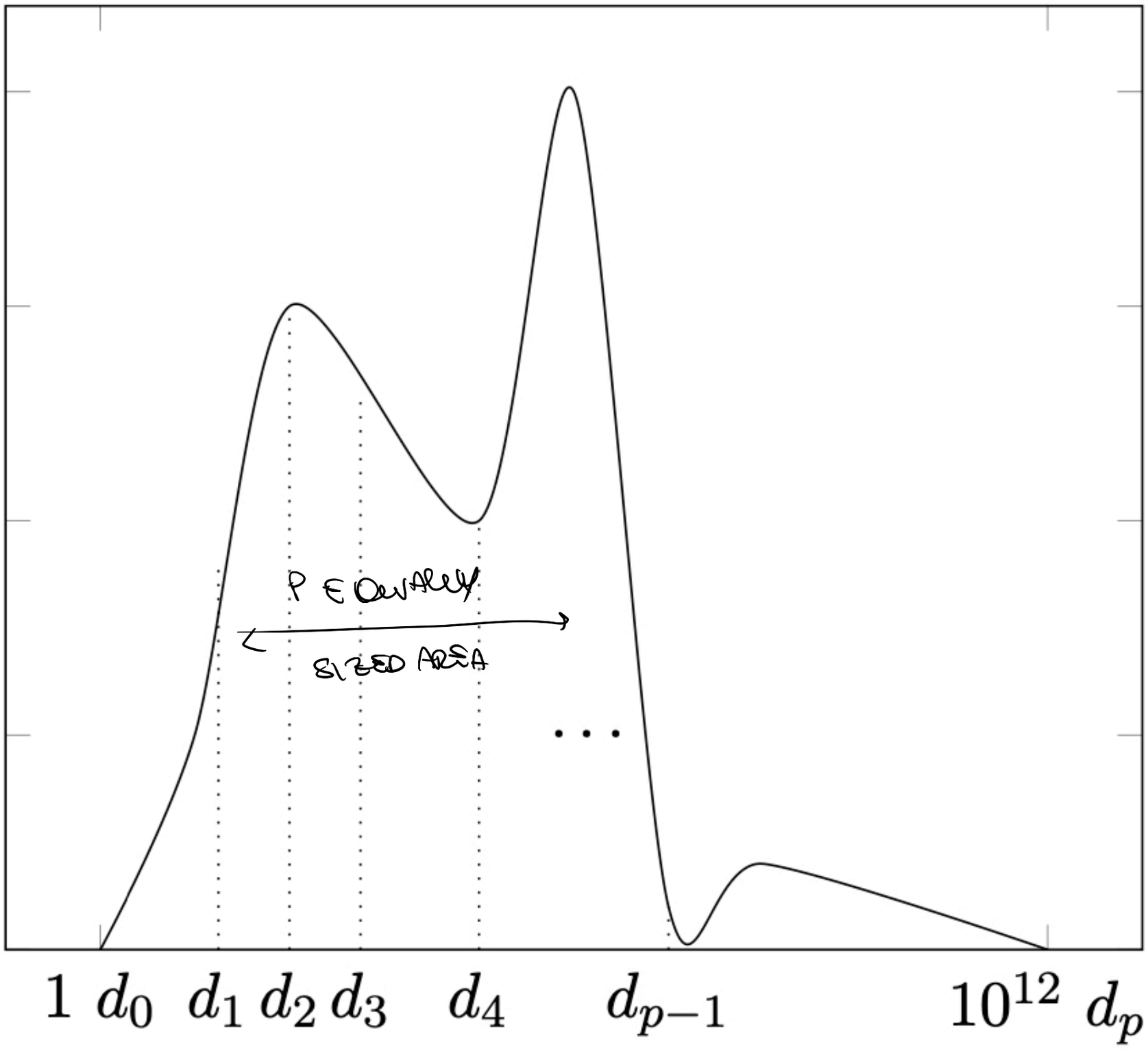
Uniform Sampling (II)

- **Naive Sampling Algorithm**:
  - *Initialization*: for each value i = 1, ..., n, initialize a counter $f_i \leftarrow 0$. Initialize $m \leftarrow 0$.
  - *Streaming*: for each s_j, if $s_j = i$, set $f_i \leftarrow f_i + 1$. Increment $m \leftarrow m + 1$.
  - *Output*: calculate $p_i = f_i/m$. Choose an i according to $P = (p_1, ..., p_n)$.
- Storing the entire stream would take **O(m log n)** space
- In this case we store n counters which range between 0 and m so require **O(n log m)** space in total.
- As **n** << **m**, this is better but not the best we can do.

Uniform Sampling (III)

- **Uniform Sampling Algorithm**:
  - *Initialize*: set x ← null, k ← 0.
  - *Streaming*: for each s_j, increment k ← k + 1. With probability 1/k, set x ← s_j. Otherwise, do nothing.
  - *Output*: return x.
- In this case we store 1 counter for x and 1 counter for j, so **O(log n + log m)** space in total.

Computing Split Points



Sorting (shuffle)

	Hadoop World Record	Spark 100 TB *	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400	6592	6080
# Reducers	10,000	29,000	250,000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min
Sort Benchmark Daytona Rules	Yes	Yes	No
Environment	dedicated data center	EC2 (i2.8xlarge)	EC2 (i2.8xlarge)

MLlib Algorithms

- **classification**: logistic regression, linear SVM, naïve Bayes, least squares, classification tree
- **regression**: generalized linear models (GLMs), regression tree
- **collaborative filtering**: alternating least squares (ALS), non-negative matrix factorization (NMF)
- **clustering**: k-means||
- **decomposition**: SVD, PCA
- **optimization**: stochastic gradient descent, L-BFGS

Gradient Descent (I)

- We are going to implement **gradient descent** on Spark
- **Separable** objective functions of the form

$$\min_w \sum_{i=1}^{n} f_i(w)$$

- $w \in \mathbb{R}^d$ is the vector of **minimizing parameters**
- $f_i(w)$ is the **loss function** applied to a training point i
  - Can be linear or non-linear
  - Can be least squares or neural network

Gradient Descent (II)

- We assume that the **number of training points** n is large, e.g., trillions
- We assume that the **number of parameters** d can fit on a single machine's RAM, e.g., millions
- Start at a random vector x_0
- Then

$$x_{k+1} \leftarrow x_k - \alpha \sum_{i=1}^{n} \nabla F_i(x_k)$$

- Where $\nabla F_i(\cdot)$ is the gradient **vector**

Gradient Descent (III)

- A training point is stored on a single machine
- Our data points are arranged in a matrix, divided among machines row-by-row
- The matrix is encoded in text
- Input data are not moved among machines

```
def parsePoint(row):

  tokens = row.split(' ')

  return np.array(tokens[:-1], dtype=float), float(tokens[-1])

w = np.zeros(d)

points = sc.textFile(input).map(parsePoint).cache()
```

Gradient Descent (III)

- To compute a gradient we need to perform a computation on each machine and a summation across machines

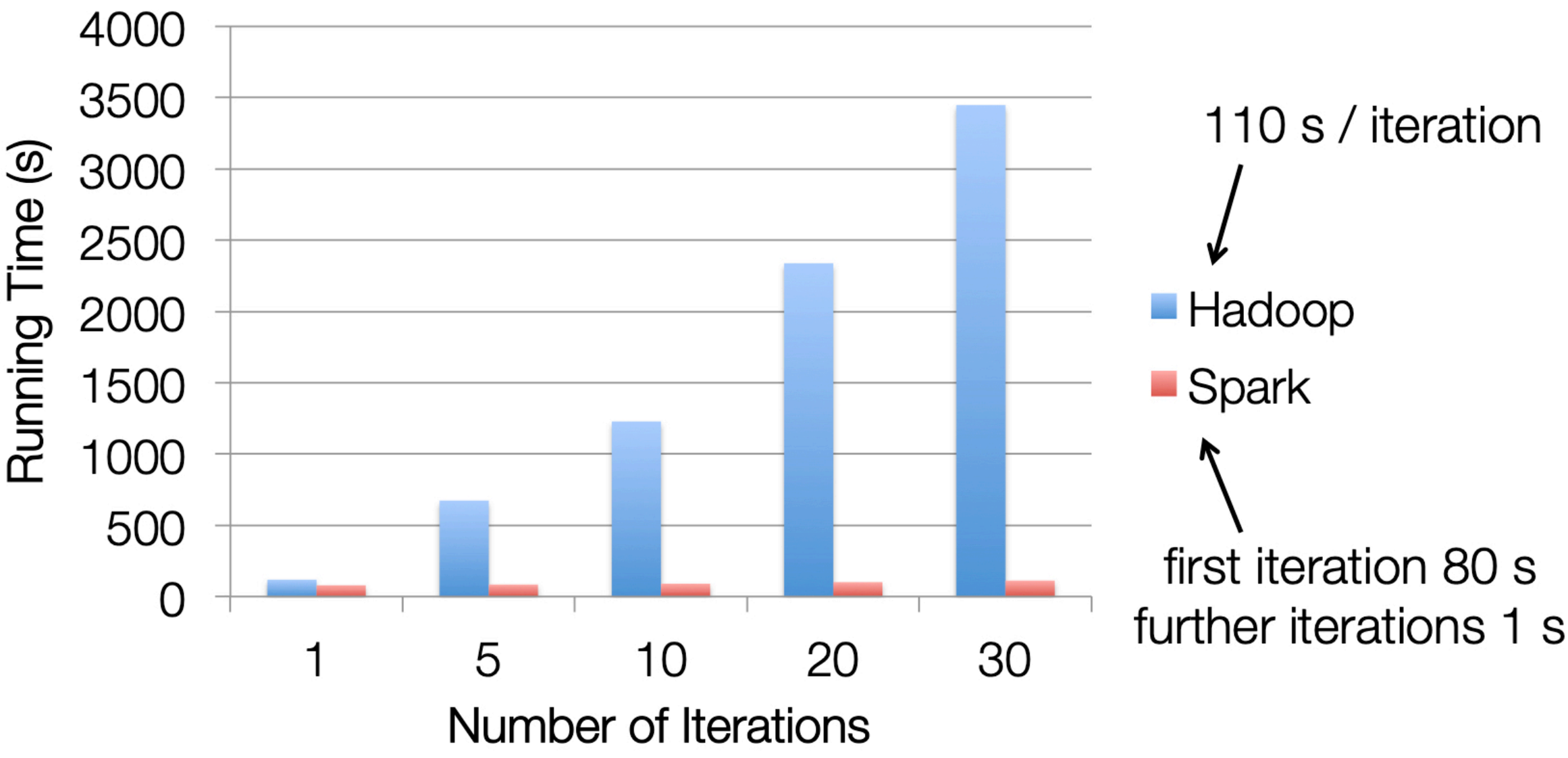
```
def compute_gradient(p, w):
    ...

gradient = points.map(lambda p: compute_gradient(p, w))
                 .reduce(lambda x, y: x + y)
```

- On a single machine we can have multiple CPUs, so we can avoid to store w for each CPU using `broadcast`

Gradient Descent (IV)

```
points = sc.textFile(input).map(parsePoint).cache()

w = np.zeros(d)

w_br = sc.broadcast(w)

for i in range(num_iterations):

    gradient = points.map(lambda p: compute_gradient(p, w_br.value))

                     .reduce(lambda x, y: x + y)

  w -= alpha * gradient

  w_br = sc.broadcast(w)
```

Results with logistic regression



100 GB of data on 50 m1.xlarge EC2 machines