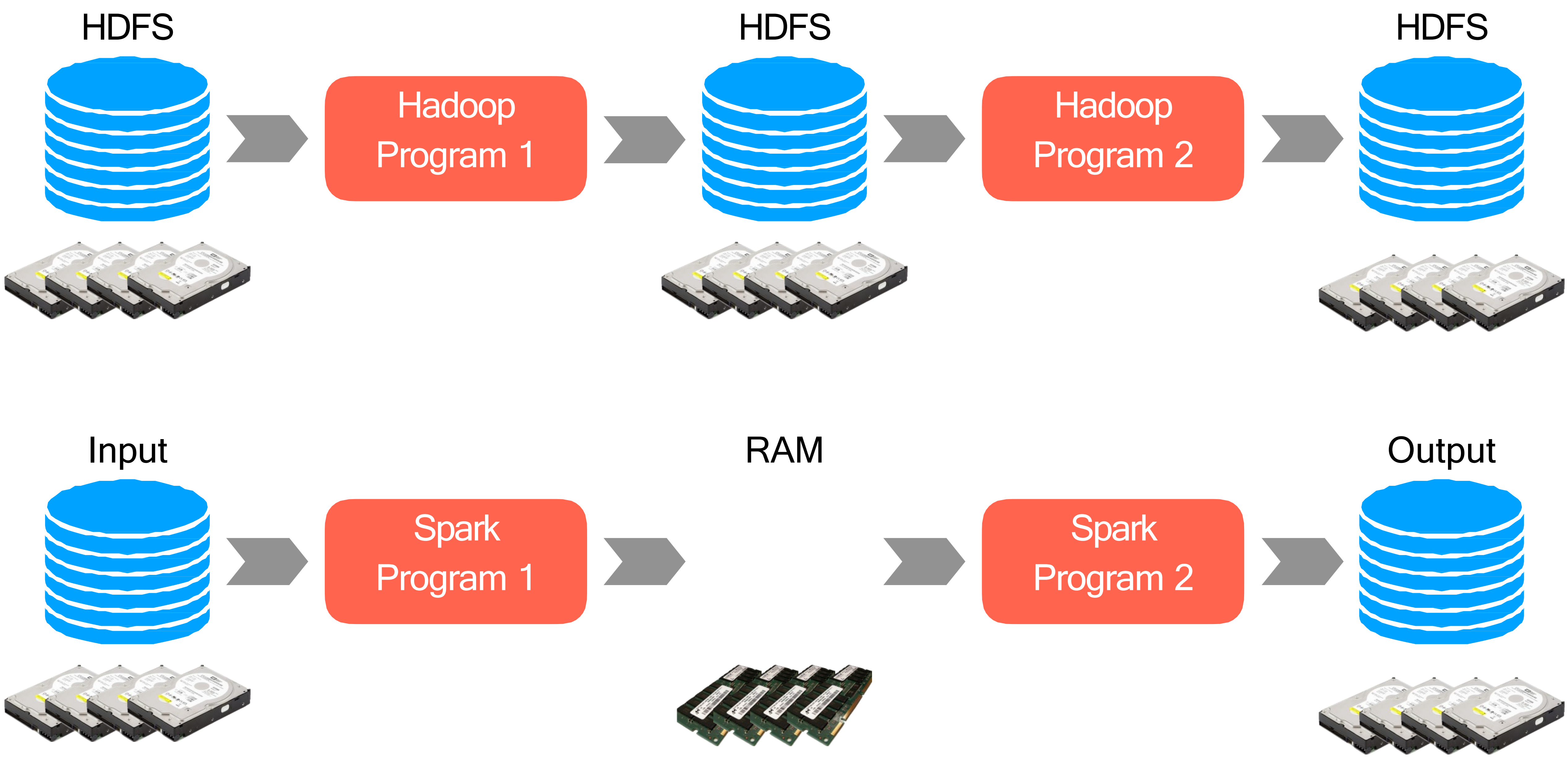
Spark

Hadoop vs Spark



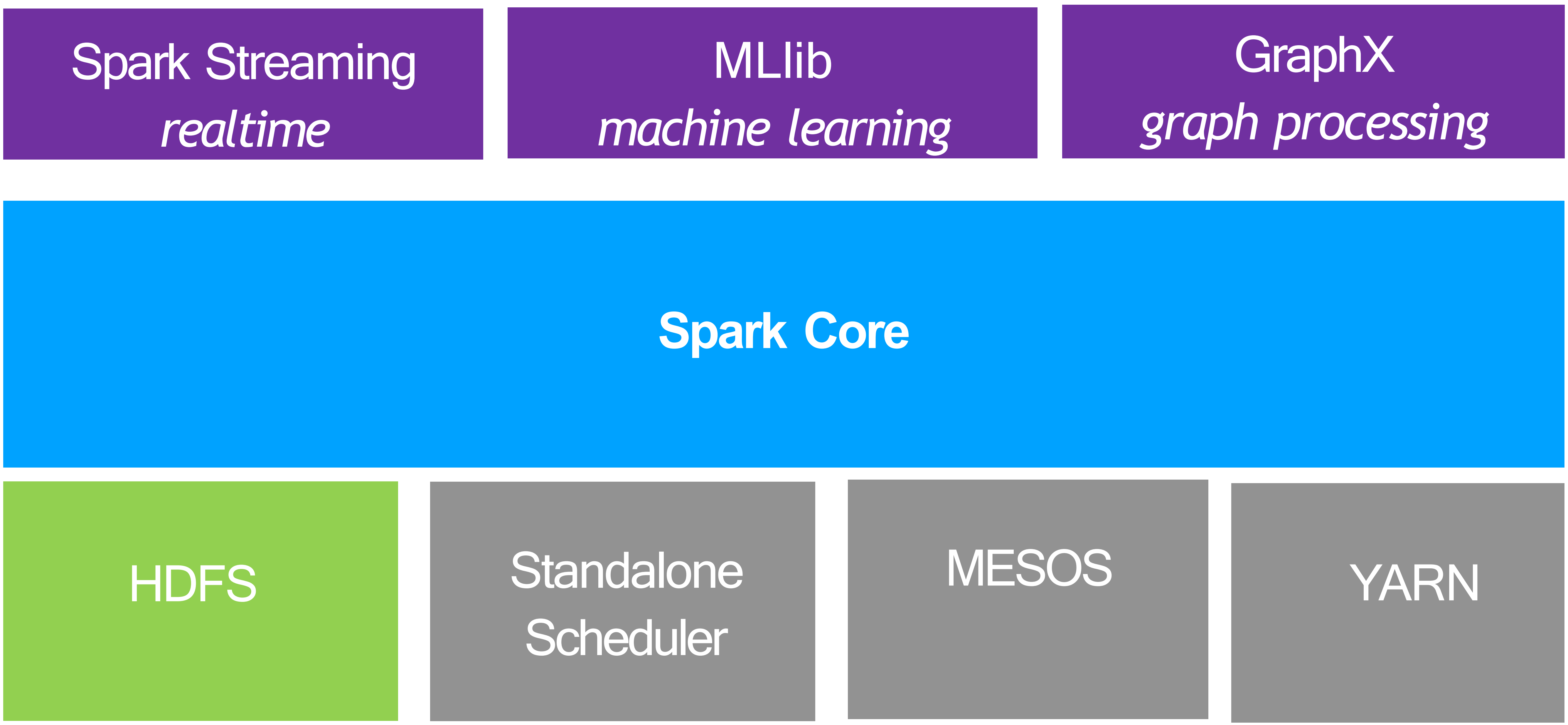
Data Flow Models

- **Restrict** the programming interface
- Express complex computations as **graphs** of high-level operators
  - 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
  - State during shuffle and sort goes through local disk storage
- **Communication and I/O** in MapReduce is the real **bottleneck**

Spark Stack


Spark Streaming
realtime
MLlib
machine learning
GraphX
graph processing
Spark Core
HDFS
Standalone
Scheduler
MESOS
YARN

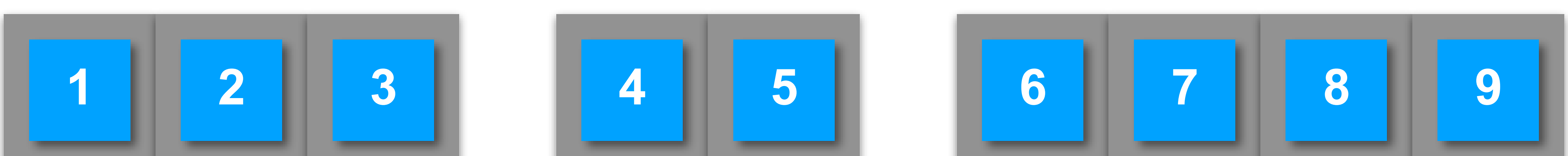
Spark Core

- Provides **basic functionalities**, including:
  - DAG scheduler (manages the execution plan)
  - interaction with HDFS and resource managers (e.g., YARN)
  - memory management (access, caching and shuffling)
  - RDD
- **Resilient distributed dataset** (**RDD**) is a collection of items distributed across many compute nodes that can be manipulated in parallel and are resilient
  - Spark Core provides many APIs for building and manipulating these collections
- Written in **Scala** but APIs for **Java**, **Python** and **R**

RDD (I)

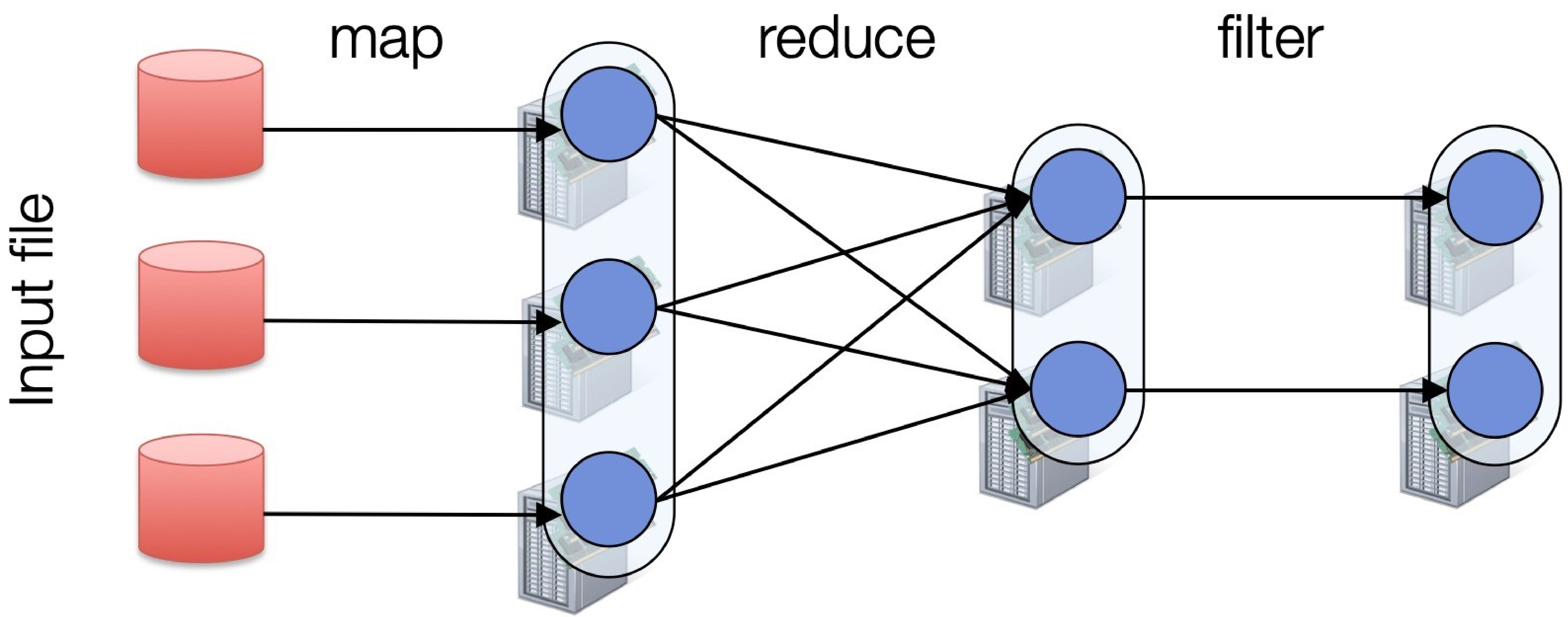
- A **resilient distributed dataset** (RDD) is a distributed memory abstraction
- Immutable collection of objects spread across the cluster

- 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



Lineage

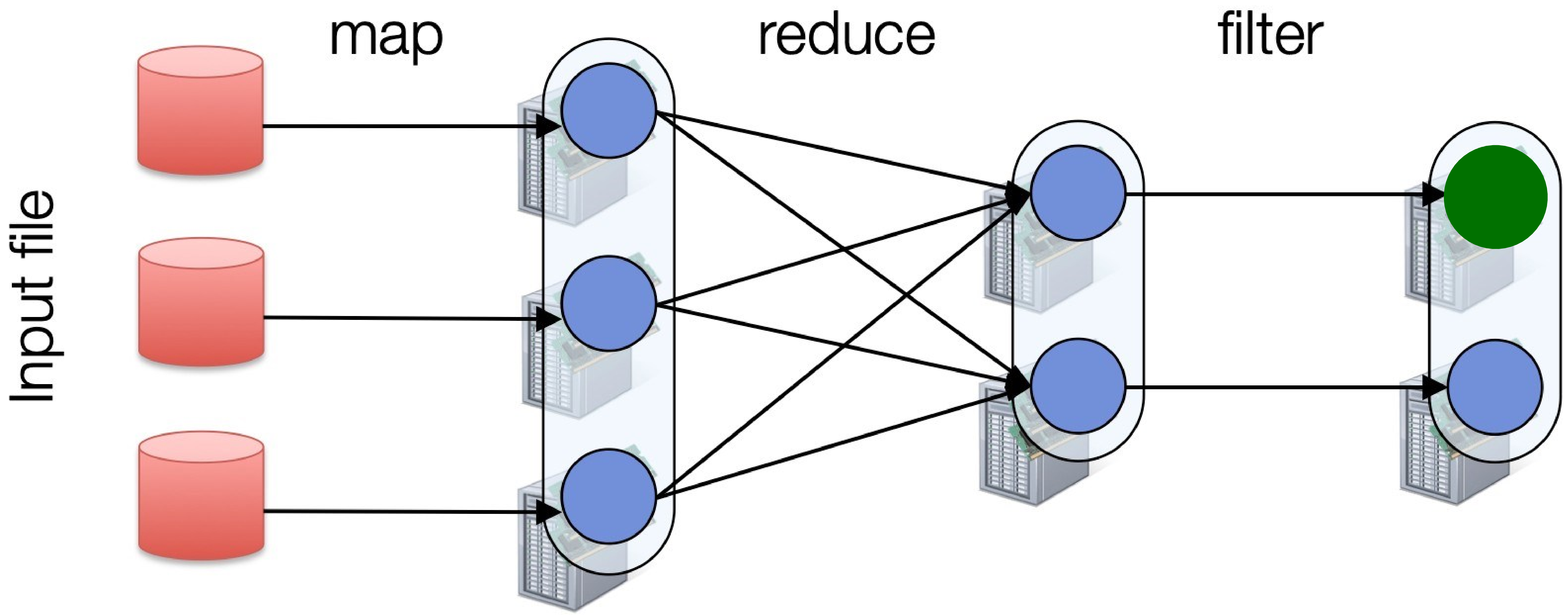
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



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

Lineage

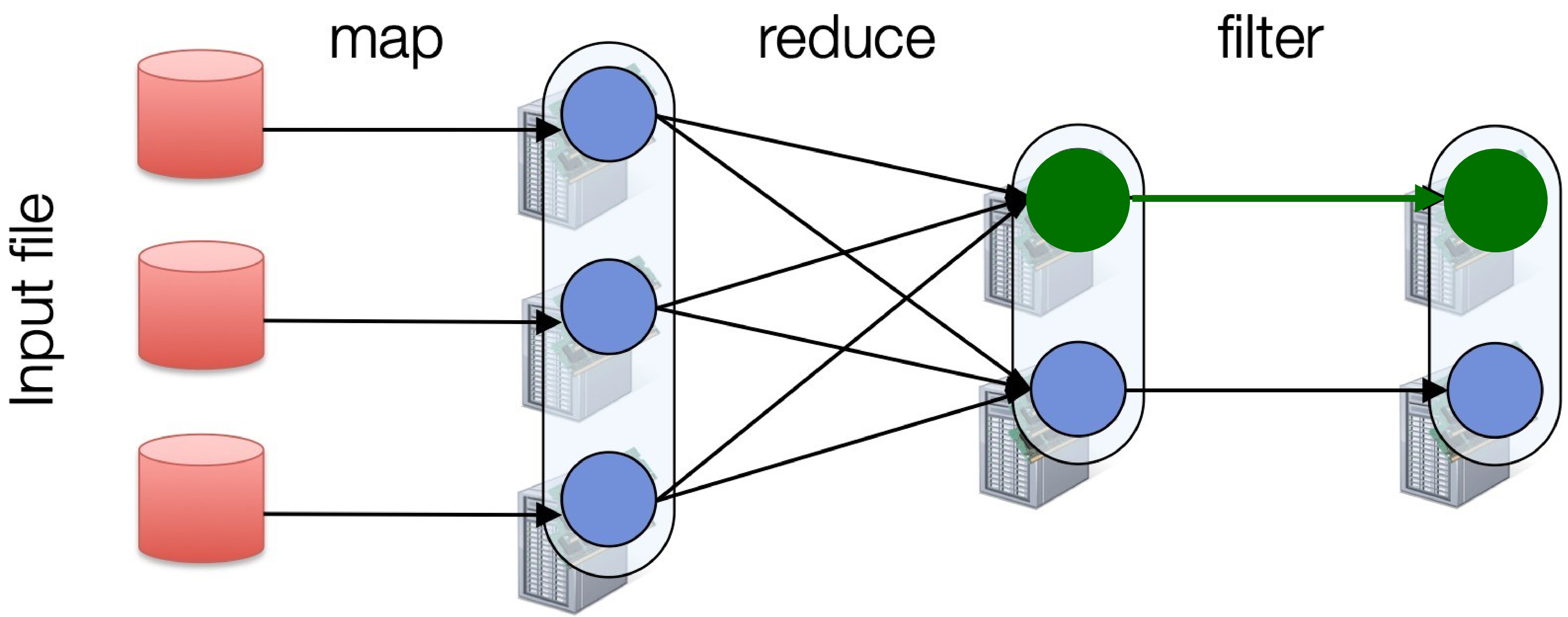
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



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

Lineage

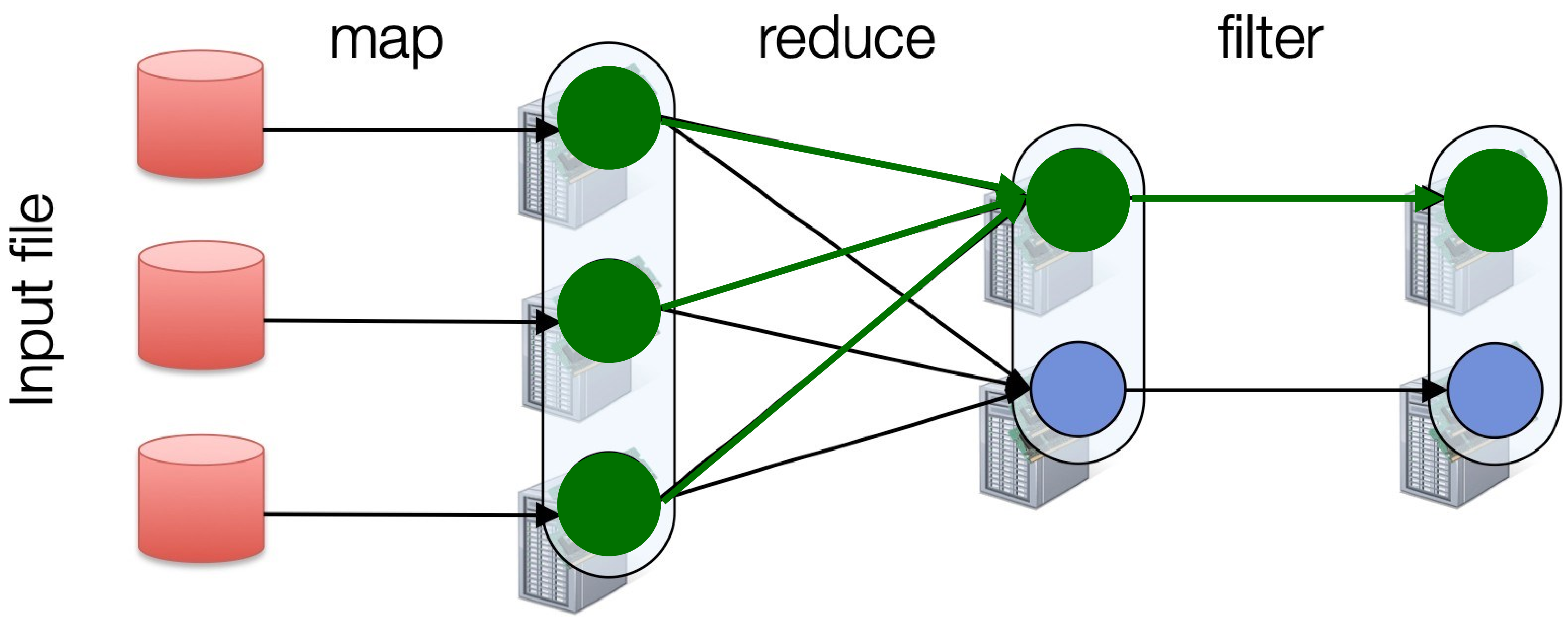
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



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

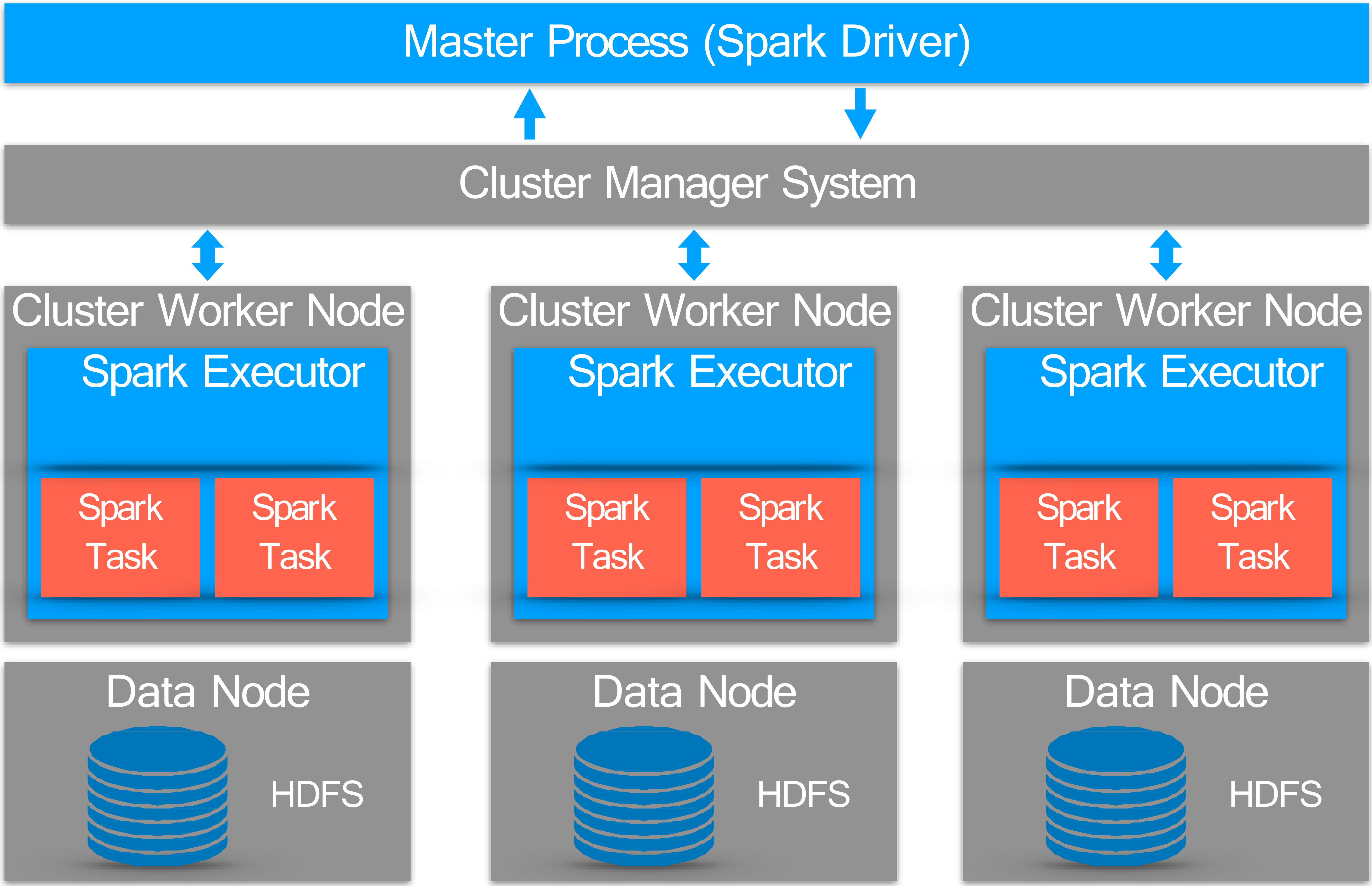
Lineage

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



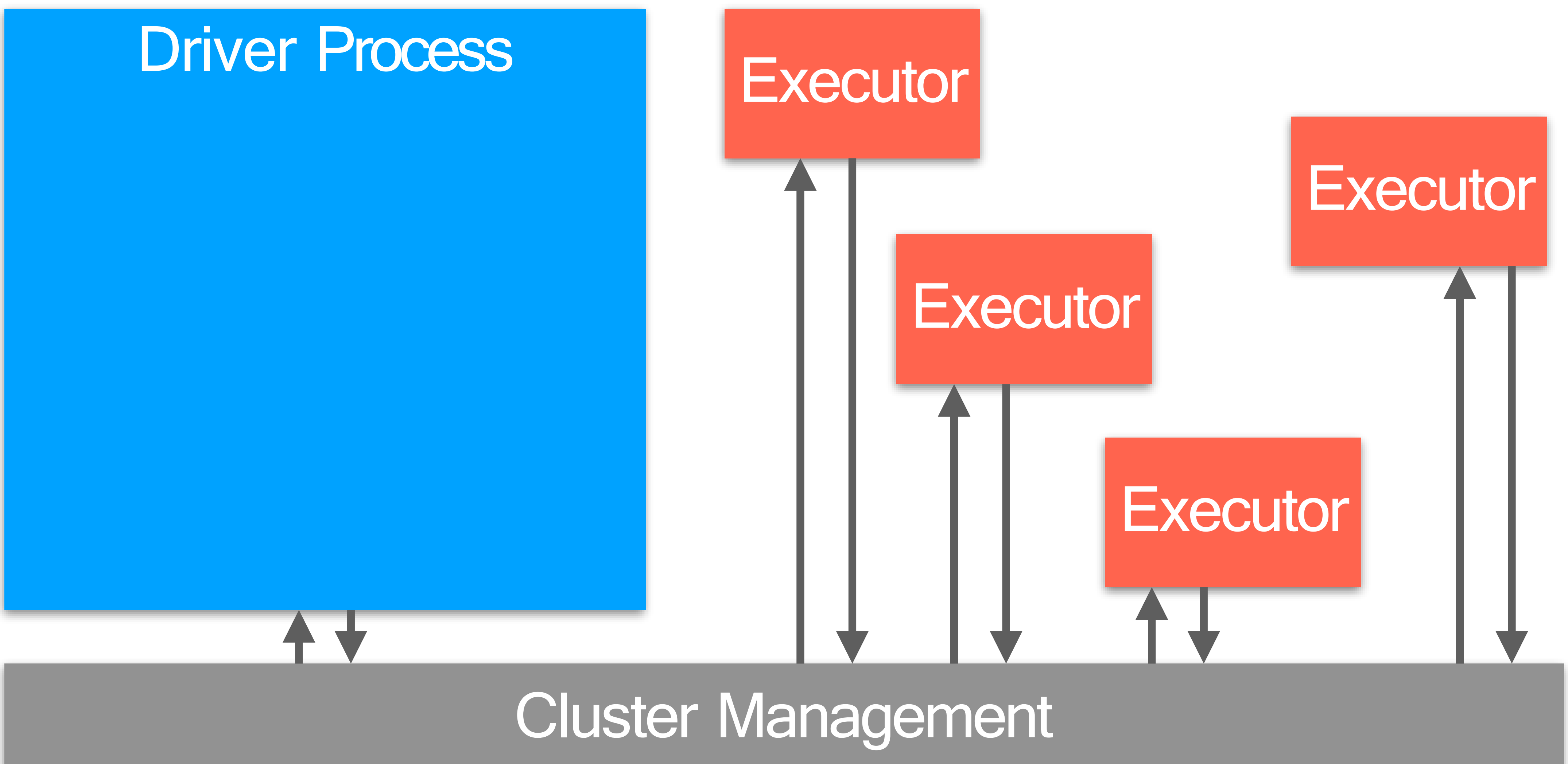
- RDDs track **lineage** info to rebuild lost data

Spark Architecture



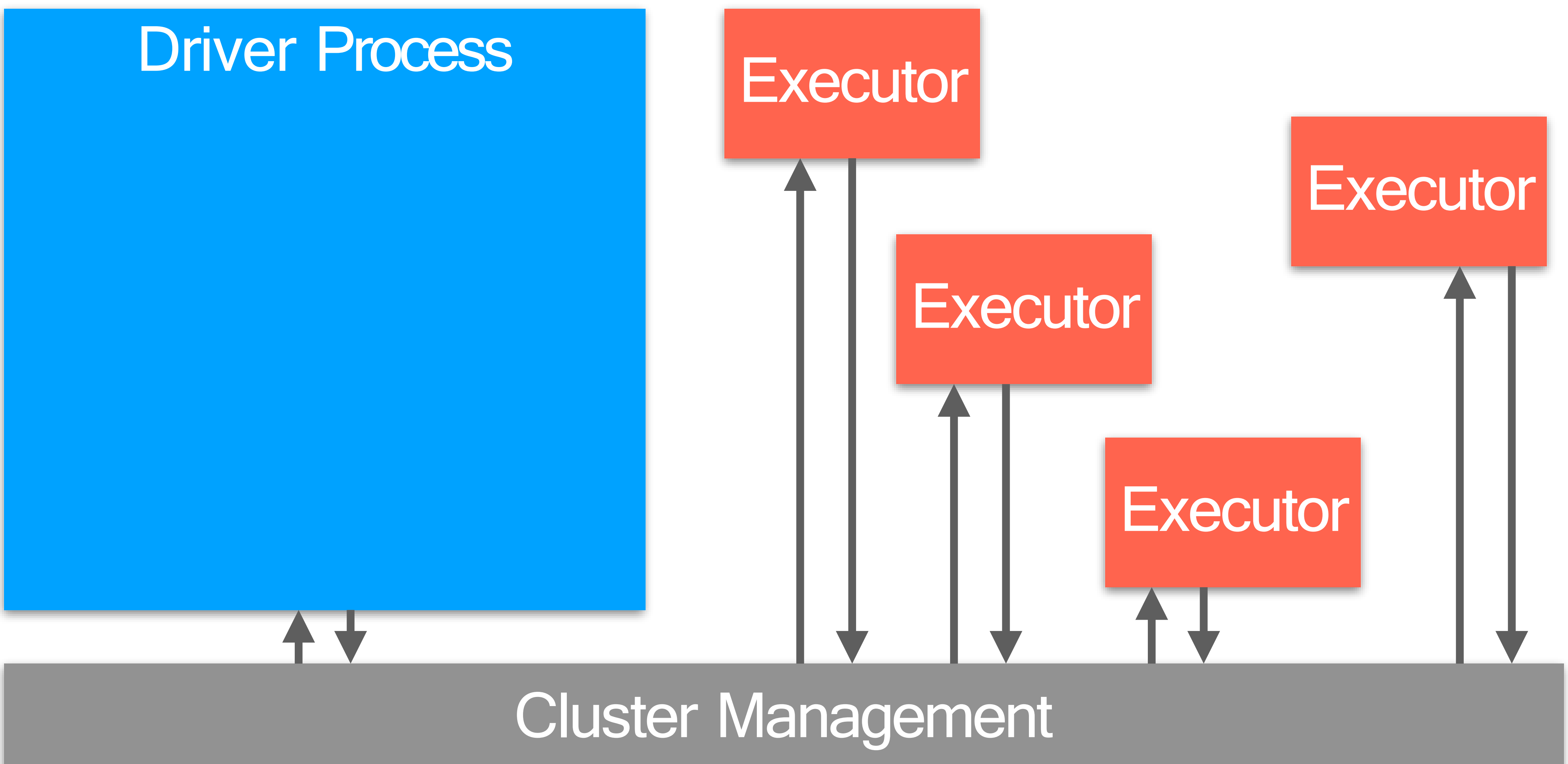
Spark Applications Architecture

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



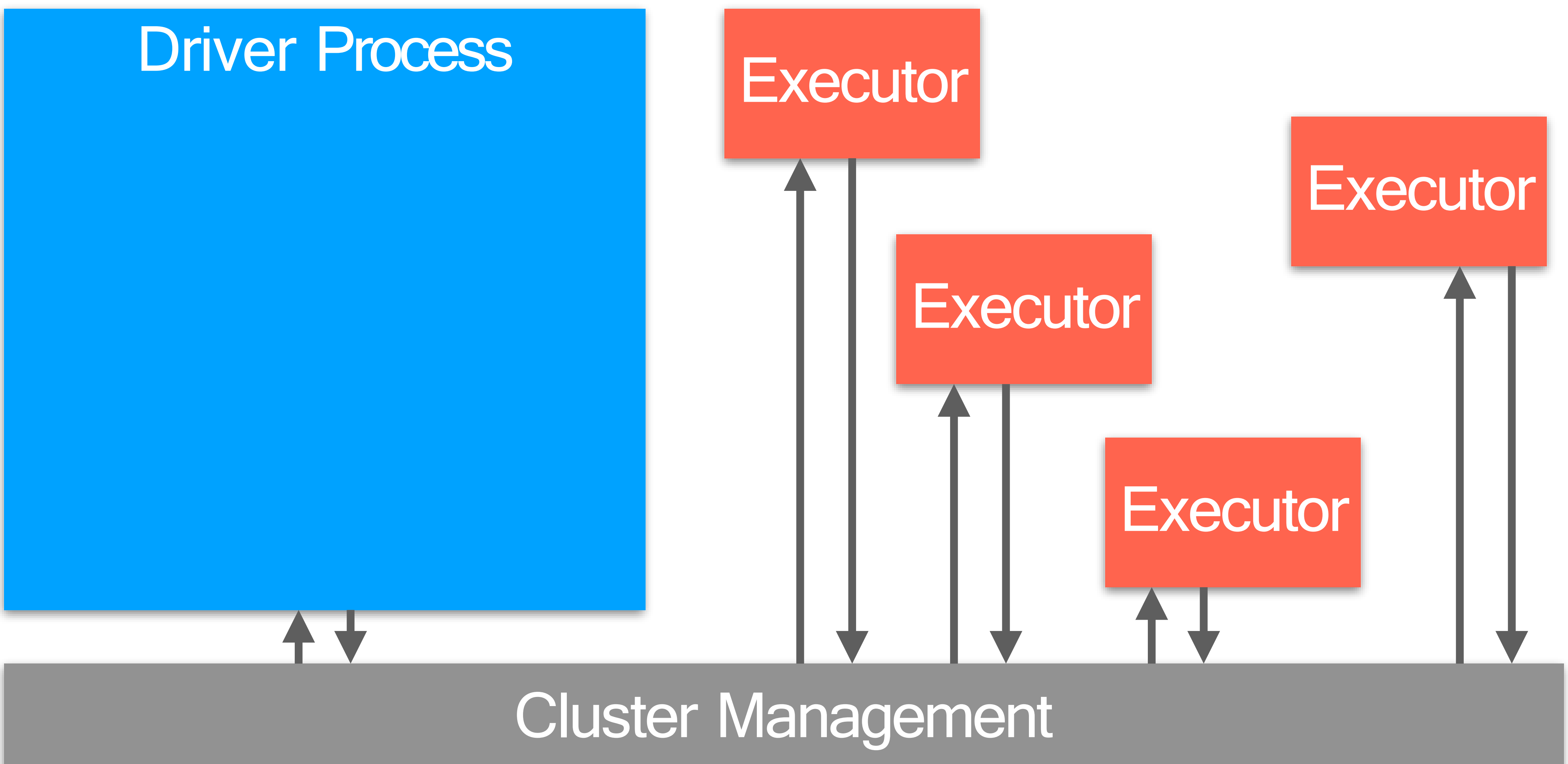
Spark Driver

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



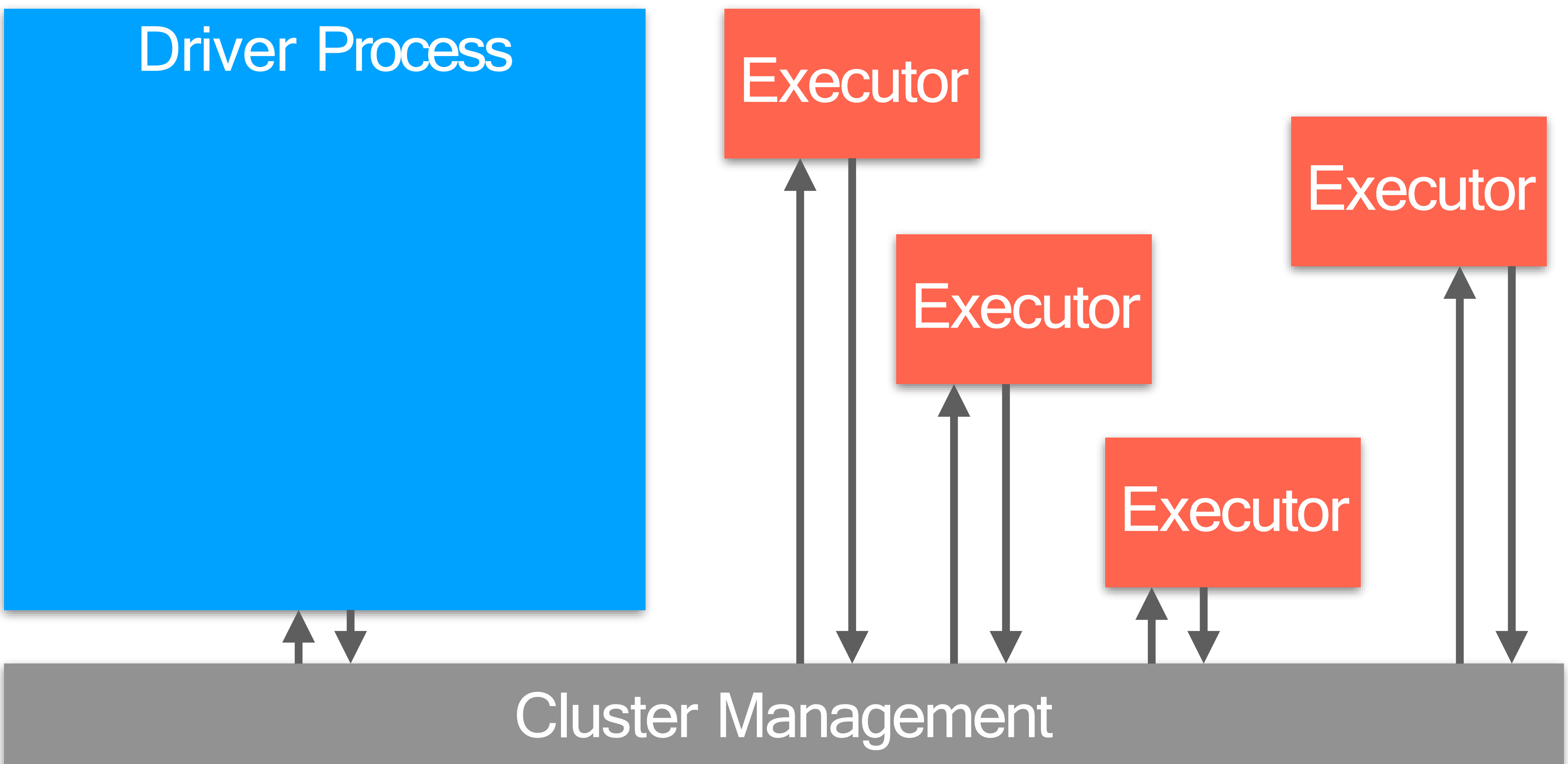
Spark Driver

- 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



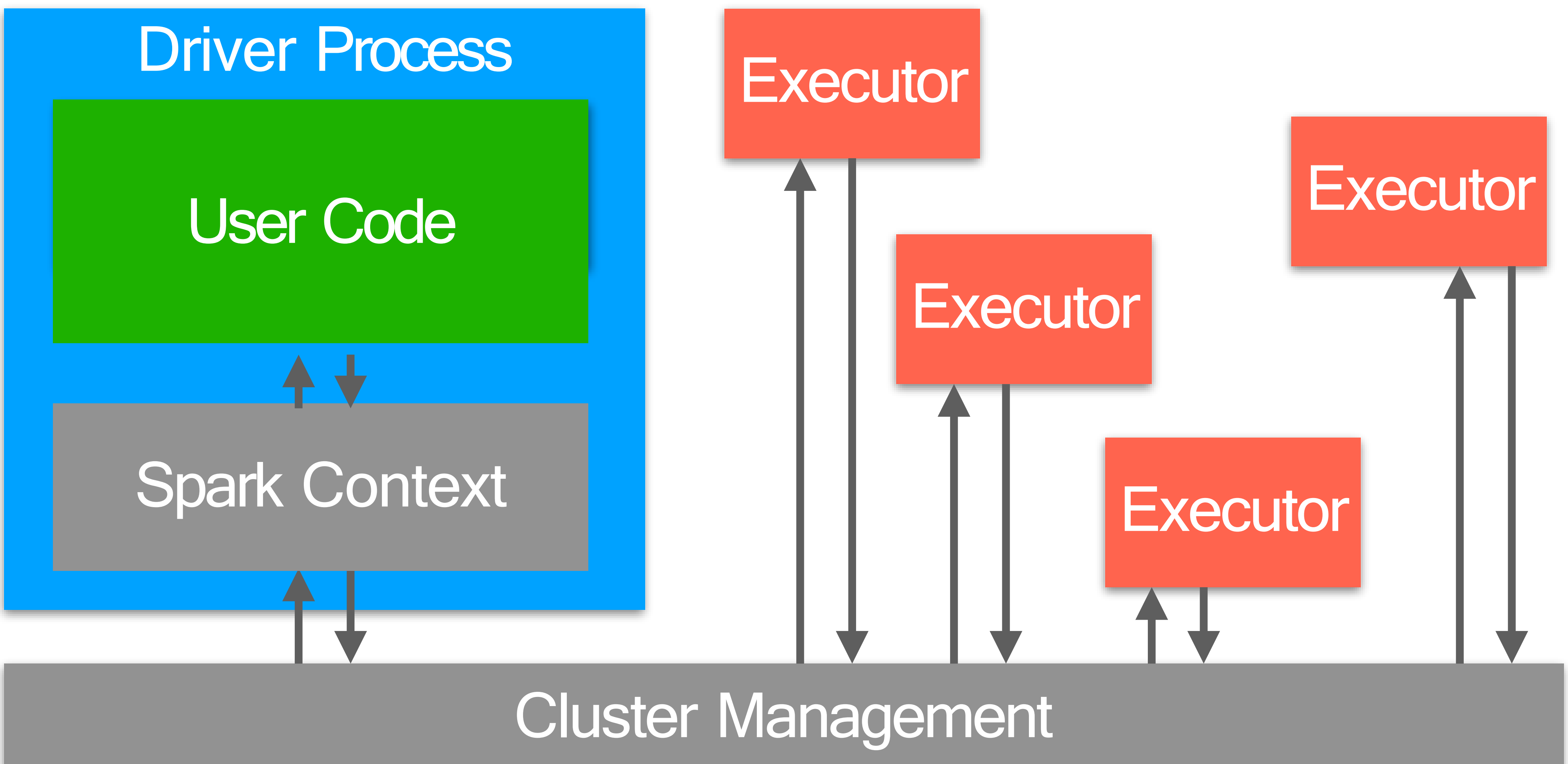
Spark Executors

- 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 SparkConf, SparkContext
# initialize SparkConf
conf = SparkConf()
# set the application name
conf.setAppName("My Spark App")
# initialize a new SparkContext with Spark configuration
sc = SparkContext(conf=conf)
```

spark-submit and master flag

- You submit Spark jobs using **spark-submit**
- Use the **--master** flag to the spark-submit command to specify the execution mode for the Spark application (driver + executors)
- The **master** flag can assume different values, e.g.:
  - **local**: run in local mode as a single JVM process, using 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 (this is conceptually similar to **"uber"** mode in Hadoop)
  - **yarn**: use the whole cluster through YARN

spark-submit and deploy-mode flag

- Use the **--deploy-mode** flag to the spark-submit command to specify how the Spark Driver should run
- The **deploy-mode** flag can assume different values, e.g.:
  - **client**: the Spark Driver runs as part of the spark-submit process. The **default** and mainly used for testing. If the user terminates the spark-submit command or the machine fails, the Driver and the whole Spark application fails.
  - **cluster**: the Spark Driver runs as a separate process, within the Application Master container if YARN is used.

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 **suggest** 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)
```

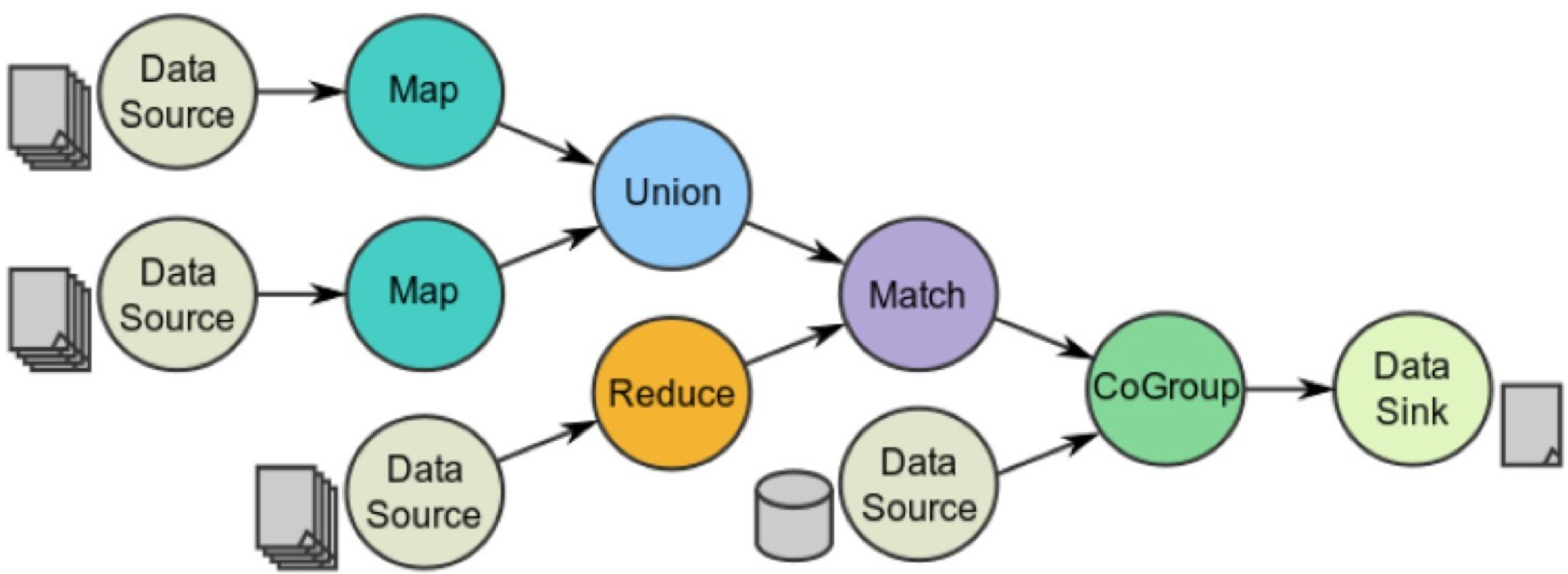
Creating an RDD

- RDDs can be created from external storage
  - Local disk, HDFS, Amazon S3, ...
  - Again, it is possible to **suggest** the number of partitions for the RDD
  - Usually, if the file is taken from HDFS, Spark creates as many partitions as the number of blocks in HDFS
- Text file RDDs can be created using the `textFile()` method

```
rdd = sc.textFile("/user/hadoop/file.txt")

rdd =  sc.textFile("file.txt’’, 2)

rdd =  sc.textFile(“/user/hadoop/*il*.txt’’)

rdd =  sc.textFile(“/user/hadoop/’’)
```

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**).
  - 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

- `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 one or more part-xxxxx **text files** (one for each partition). It wants a directory path as argument.
  - 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()
# dict {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")
# then, you may check contents of files 'numbers/part-xxxxx'
```

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]
```

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 RDDs **of the same length**.

```
words = sc.parallelize("nel mezzo del cammin di nostra vita".split(" "))
keywords1 = words.map(lambda w: (w, 1)) print(keywords1.collect())
[('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', 6)]
```

Key-Value RDD Transformations

- `keys` and `values` extract keys and values from the RDD, respectively

```
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']
```

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)]
```

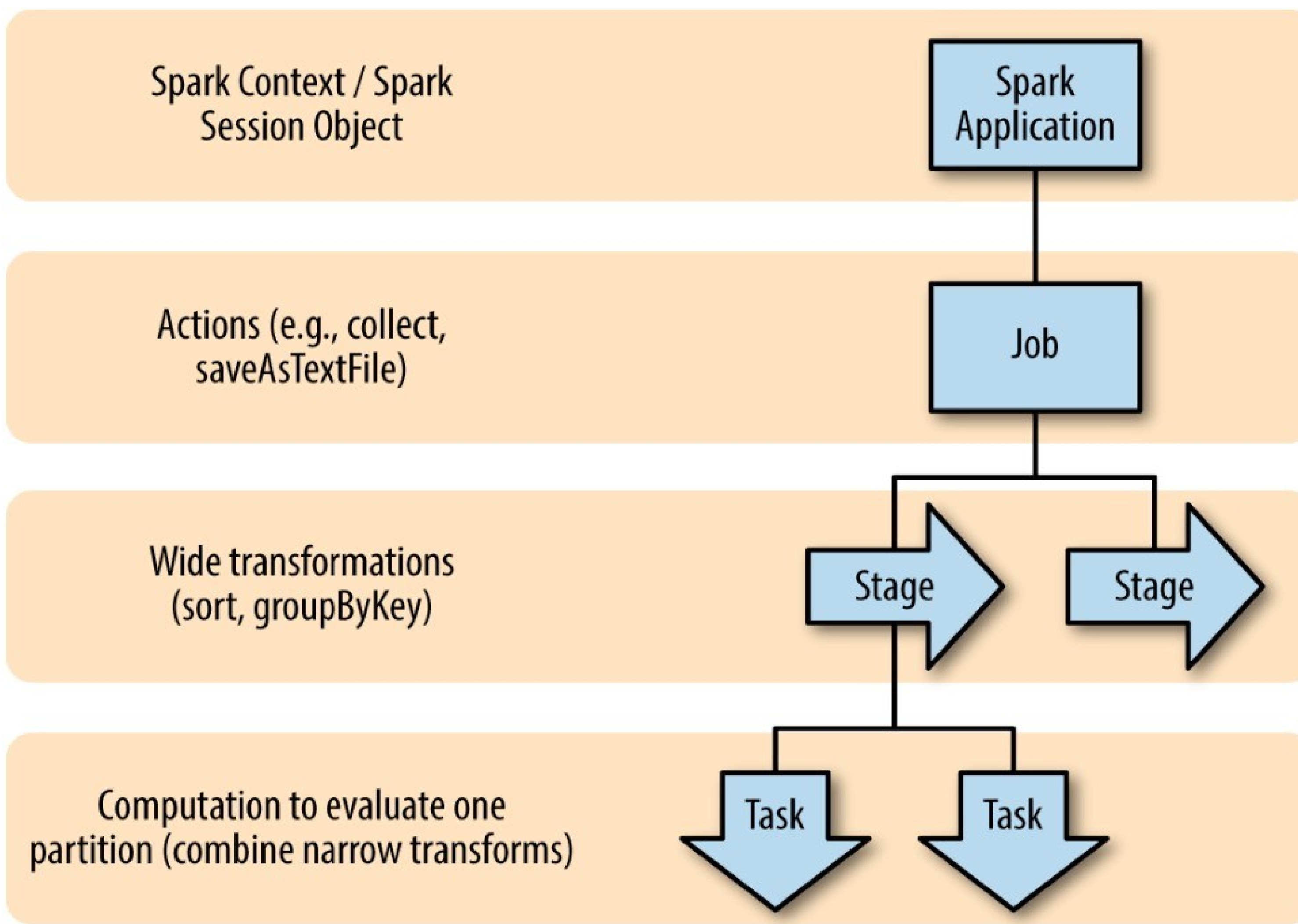
Key-Value RDD Transformations

- 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())
[(1, ('Ferrari', 'red')), (1, ('Porsche', 'red')), (2, ('Mercedes', 'black'))]
```

Anatomy of a Spark Job

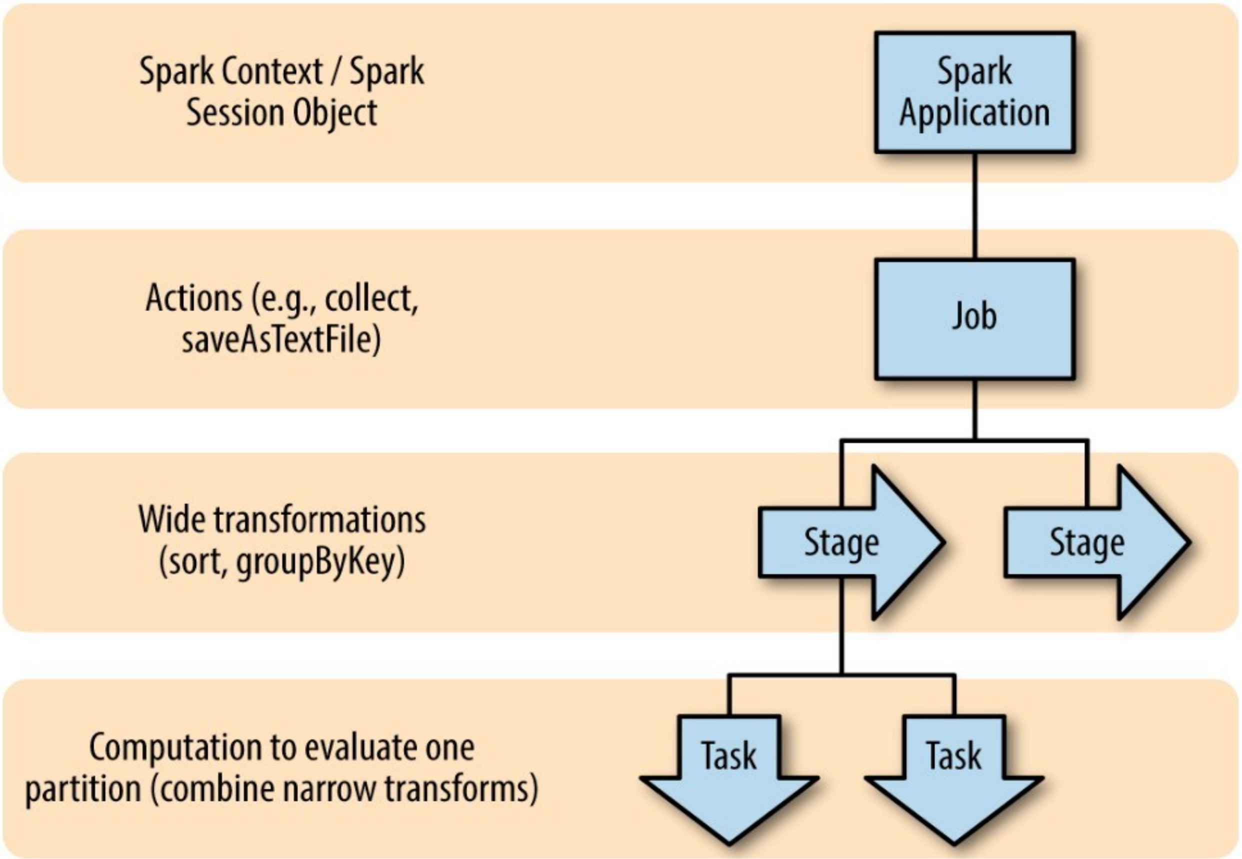
- A **Spark application** doesn't "do anything" until the driver program calls an action (**lazy evaluation**)
- Each **action** is called by the driver program of a Spark application
- Each **Spark job** corresponds to **one action**



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

Spark Stage

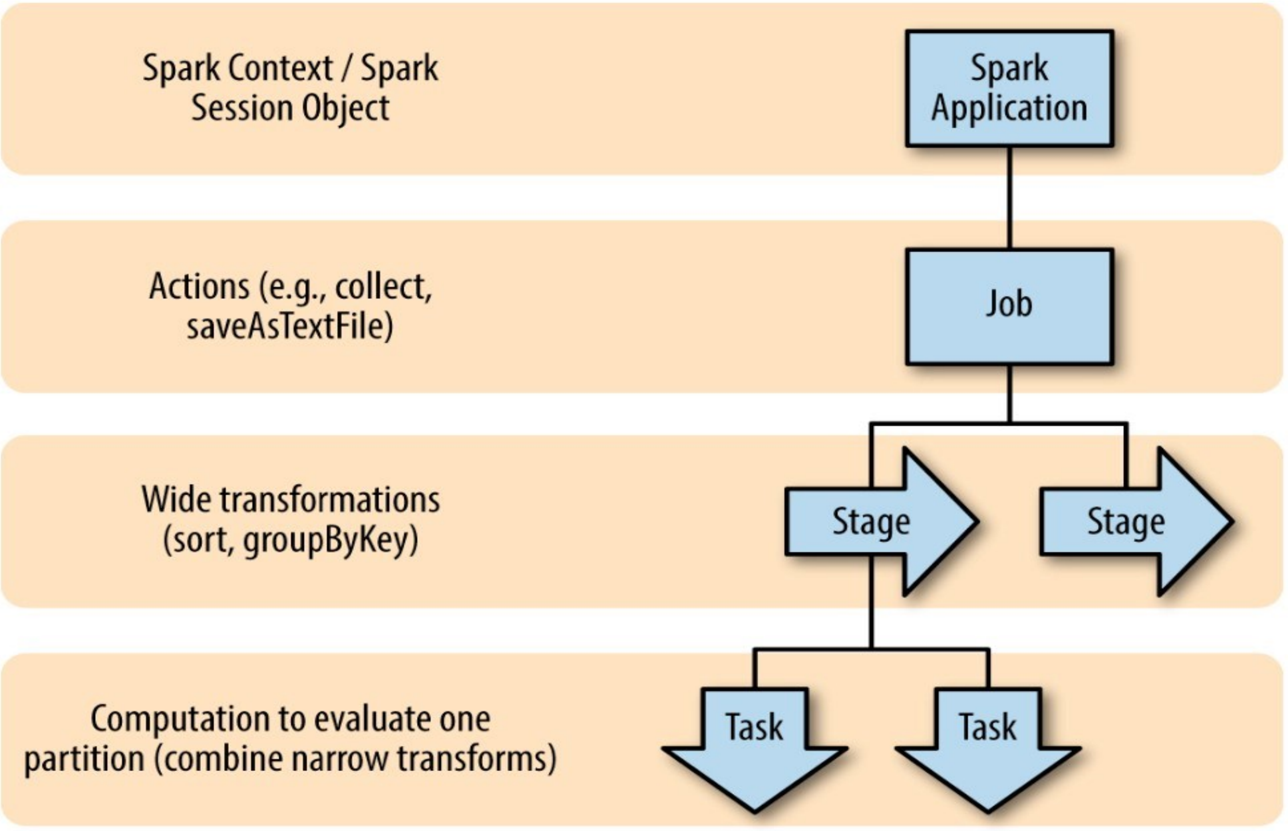
- Each **job** breaks down into a **series of stages**
  - A stage in Spark represents a **group of tasks** that can be executed **together**
  - **Wide transformations** define the breakdown of jobs into stages
  - Wide transformations are those requiring data shuffling, namely more RDD partitions as input (e.g., reduceByKey())
  - **Narrow transformations** instead work on a single RDD partition (e.g., map())



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** but on **different partitions of the data**
  - Basically, a task is the execution of all the **narrow transformations** of that stage over a specific piece of data



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

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**

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**)