بسم الله الرحمن الرحیم

تکنولوژی کامپیوتر

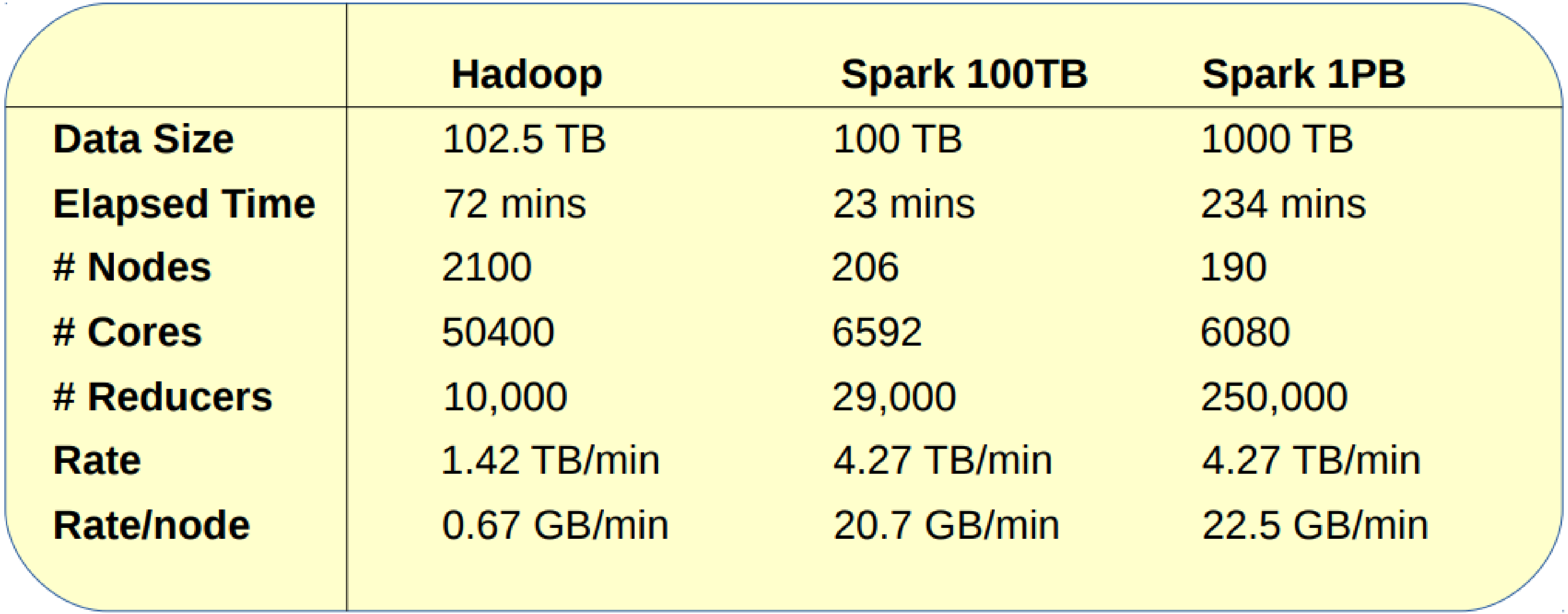
جلسه‌ی بیست و هفتم
اسپارک

جلسه گذشته

جلسه‌ی جدید

SPARK

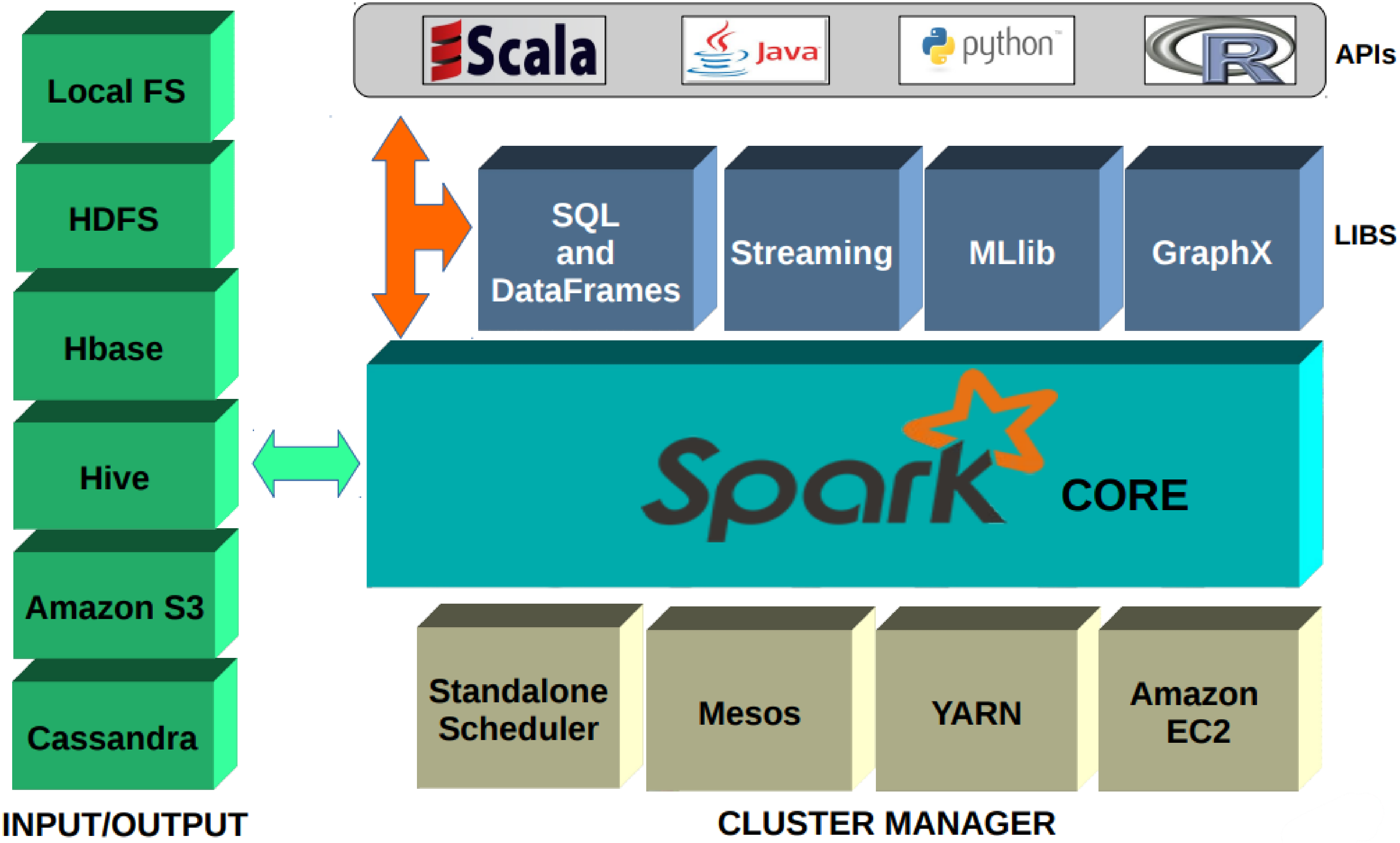
Spark Vs. Hadoop MapReduce



	Hadoop	Spark 100TB	Spark 1PB
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

Use memory!

Spark Basics: architecture



Resilient Distributed Datasets (RDDs)

Data manipulation in Spark is heavily based on RDDs. An RDD is an interface composed of:

- a set of partitions
- a list of dependencies
- a function to compute a partition given its parents
- a partitioner (optional)
- a set of preferred locations per partition (optional)

- Simply stated: an RDD is a distributed collections of items.
- an RDD is a read-only (i.e., immutable) collection of items partitioned across a set of machines that can be rebuilt if a partition is destroyed.

The RDD is the most fundamental concept in Spark since all work in Spark is expressed as:

- creating RDDs
- transforming existing RDDs
- performing actions on RDDs

Creating RDDs

Spark provides two ways to create an RDD:

- **loading** an already existing set of objects
- **parallelizing** a data collection in the driver

Creating RDDs

```
// define the spark context
val sc = new SparkContext(...)

// hdfsRDD is an RDD from an HDFS file
val hdfsRDD = sc.textFile("hdfs://...")

// localRDD is an RDD from a file in the local file system
val localRDD = sc.textFile("localfile.txt")

// define a List of strings
val myList = List("this", "is", "a", "list", "of", "strings")

// define an RDD by parallelizing the List
val listRDD = sc.parallelize(myList)
```

RDD Operations

There are **transformations** on RDDs that allow us to create new RDDs: `map, filter, groupBy, reduceByKey, partitionBy, sortByKey, join, etc`

Also, there are **actions** applied in the RDDs: `reduce, collect, take, count, saveAsTextFile, etc`

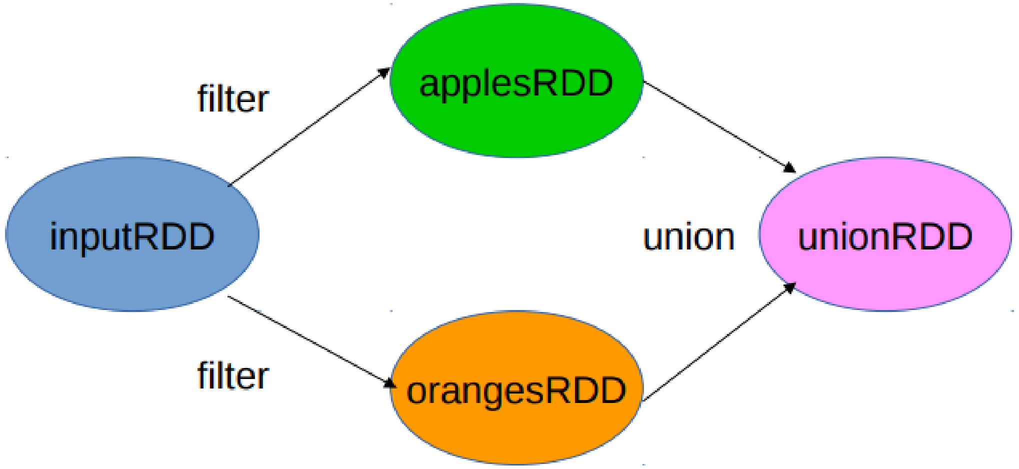
Note: computation takes place only in actions and not on transformations! (This is a form of **lazy evaluation**. More on this soon.)

RDD Operations: transformations

```
val inputRDD = sc.textFile("myfile.txt")

// lines containing the word “apple”
val applesRDD = inputRDD.filter(x => x.contains("apple"))

// lines containing the word “orange”
val orangesRDD = inputRDD.filter(x => x.contains("orange"))

// perform the union
val aoRDD = applesRDD.union(orangesRDD)
```

RDD Operations: transformations

Graphically speaking:



RDD Operations: actions

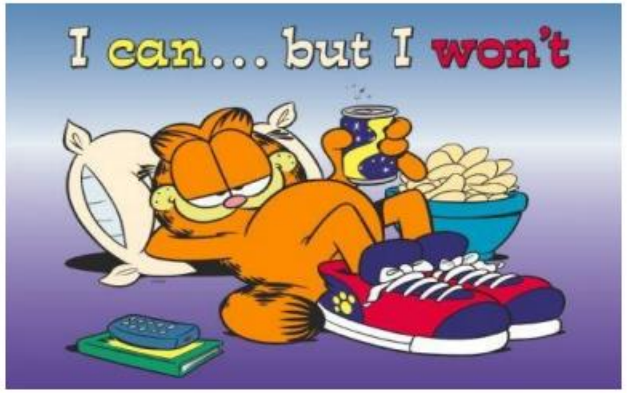
An action denotes that **something must be done**

We use the action `count()` to find the number of lines in unionRDD containing apples or oranges (or both) and then we print the 5 first lines using the action `take()`

```
val numLines = unionRDD.count()
unionRDD.take(5).foreach(println)
```

Lazy Evaluation

The benefits of being lazy



1. more optimization alternatives are possible if we see the **big picture**
2. we can avoid unnecessary computations

Ex:

Assume that from the unionRDD we need only the first 5 lines.

If we are eager, we need to compute the union of the two RDDs, materialize the result and then select the first 5 lines.

If we are lazy, there is no need to even compute the whole union of the two RDDs, since when we find the first 5 lines we may stop.

Lazy Evaluation

At any point we can **force the execution** of transformation by applying a simple action such as **`count()`**. This may be needed for debugging and testing.

Basic RDD Transformations

Assume that our RDD contains the list`{1,2,3}`.

`map()`	`rdd.map(x => x + 2)`	`{3,4,5}`
`flatMap()`	`rdd.flatMap(x => List(x-1,x,x+1))`	`{0,1,2,1,2,3,2,3,4}`
`filter()`	`rdd.filter(x => x>1)`	`{2,3}`
`distinct()`	`rdd.distinct()`	`{1,2,3}`
`sample()`	`rdd.sample(false,0.2)`	`non-predictable`

Two-RDD Transformations

These transformations require two RDDs

`union()`	`rdd.union(another)`
`intersection()`	`rdd.intersection(another)`
`subtract()`	`rdd.substract(another)`
`cartesian()`	`rdd.cartesian(another)`

Some Actions

collect()	rdd.collect()	{1,2,3}
count()	rdd.count()	3
countByValue()	rdd.countByValue()	{(1,1),(2,1),(3,1)}
take()	rdd.take(2)	{1,2}
top()	rdd.top(2)	{3,2}
reduce()	rdd.reduce((x,y) => x+y)	6
foreach()	rdd.foreach(func)	

DAG (DIRECTED ACYCLIC GRAPHS)

RDDs and DAGs

A set of RDDs corresponds is transformed to a Directed Acyclic Graph (DAG)

Input: RDD and partitions to compute

Output: output from actions on those partitions

Roles:

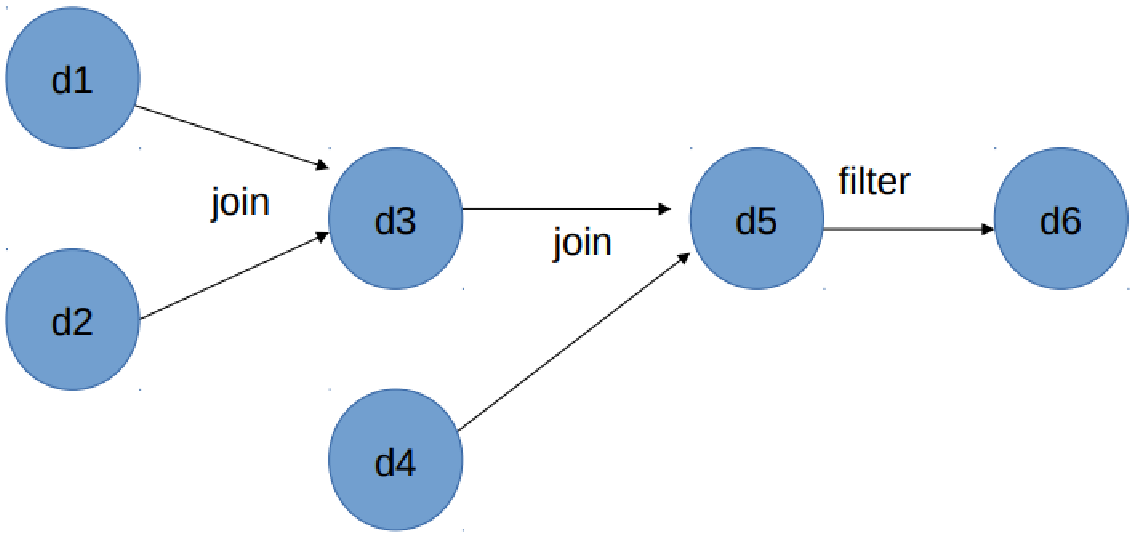
> Build stages of tasks

> Submit them to lower level scheduler (e.g. YARN, Mesos, Standalone) as ready

> Lower level scheduler will schedule data based on locality

> Resubmit failed stages if outputs are lost

DAG Scheduling



Distributed Execution in Spark

Outline of the whole process:

1. The user submits a job with **`spark-submit`**.

2. **`spark-submit`** launches the driver program and invokes the **`main()`** method specified by the user.

3. The **driver program** contacts the **cluster manager** to ask for resources to launch **executors**.

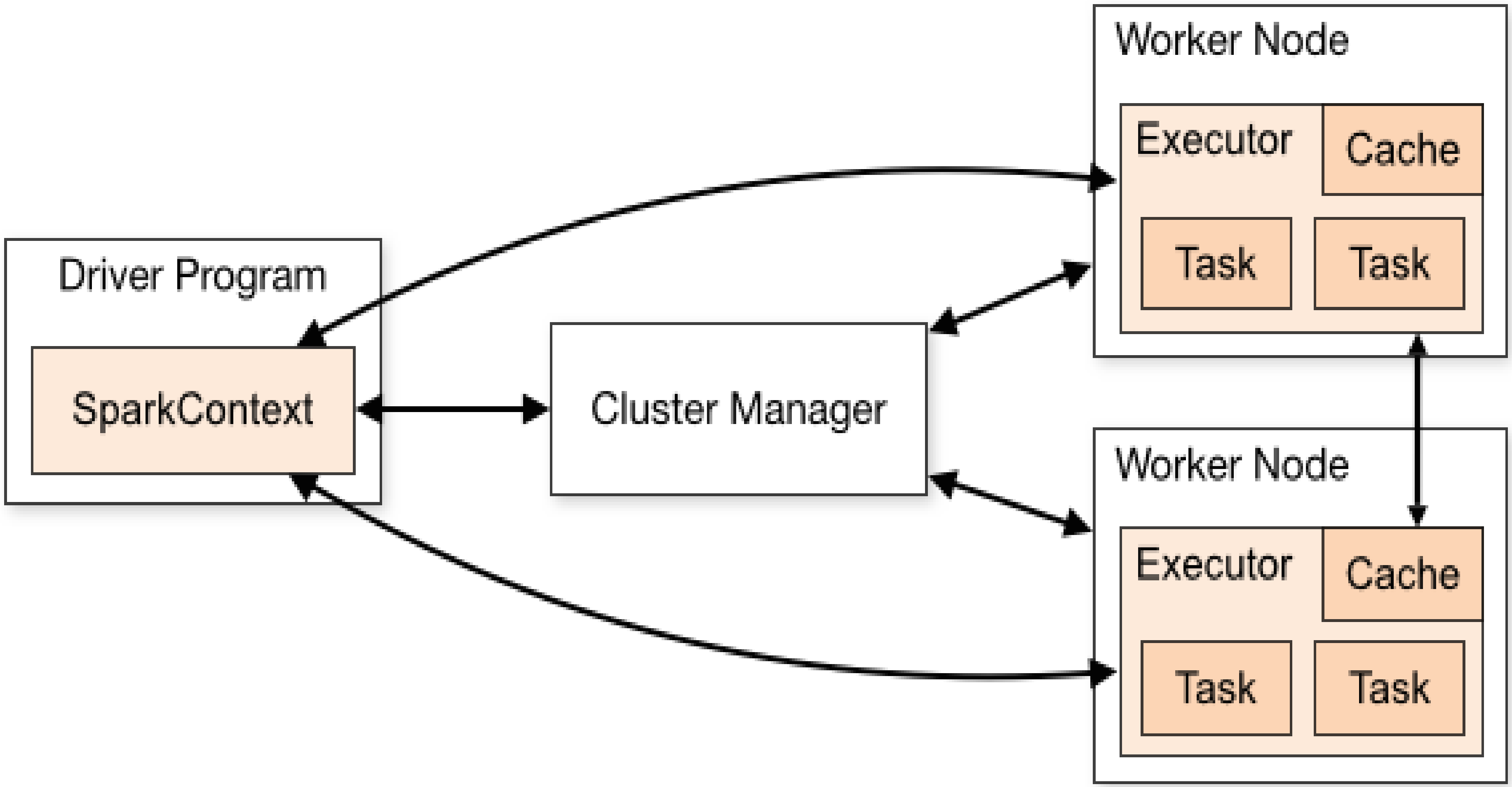
4. The **cluster manager** launches **executors** on behalf of the **driver program**.

5. The **driver process** runs through the user application. Based on the RDD actions and transformations in the program, the **driver** sends work to **executors** in the form of **tasks**.

6. **Tasks** are run on **executor processes** to compute and save results.

7. If the **driver's** **`main()`** method exits or it calls **`SparkContext.stop()`** , it will terminate the **executors** and release resources from the **cluster manager**.

Under the Hood



Shuffle in spark

Spark dataset / dataframe / sql

Spark Structured Streaming

کار کردن با HDFS