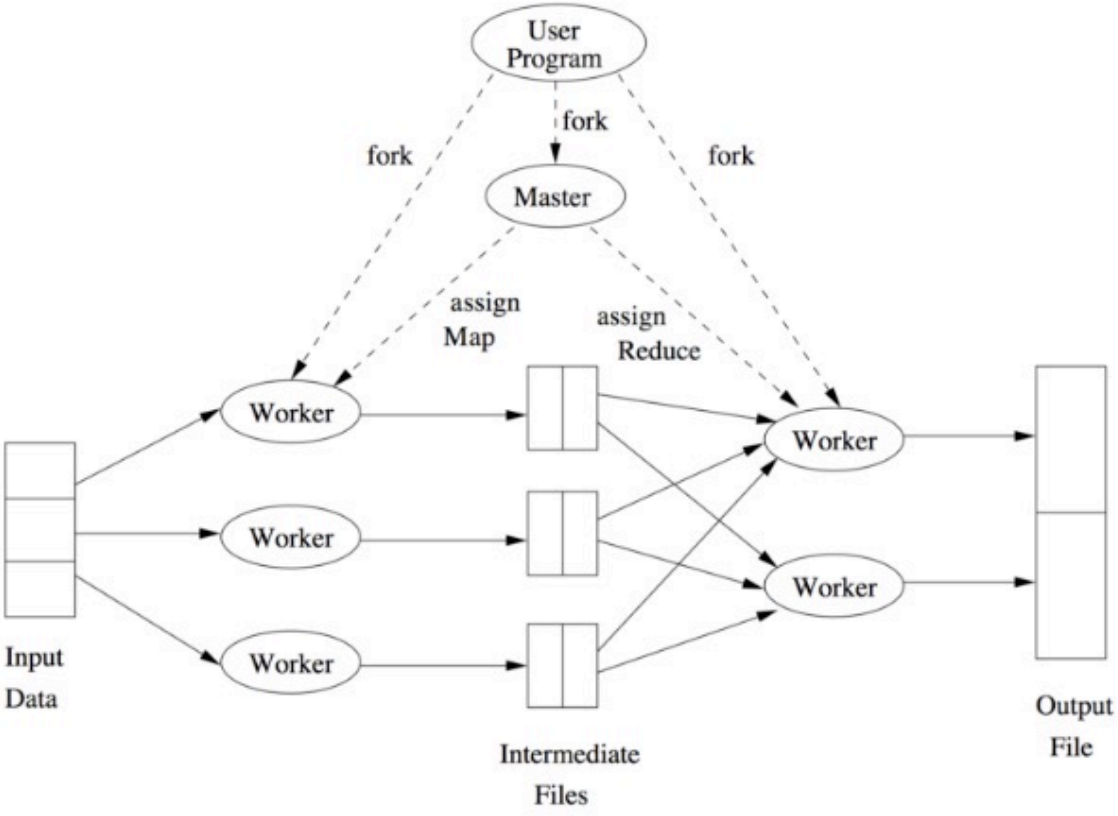
Algorithms and Applications of Data Mining - Introduction to Spark (II)

Yijun Lin

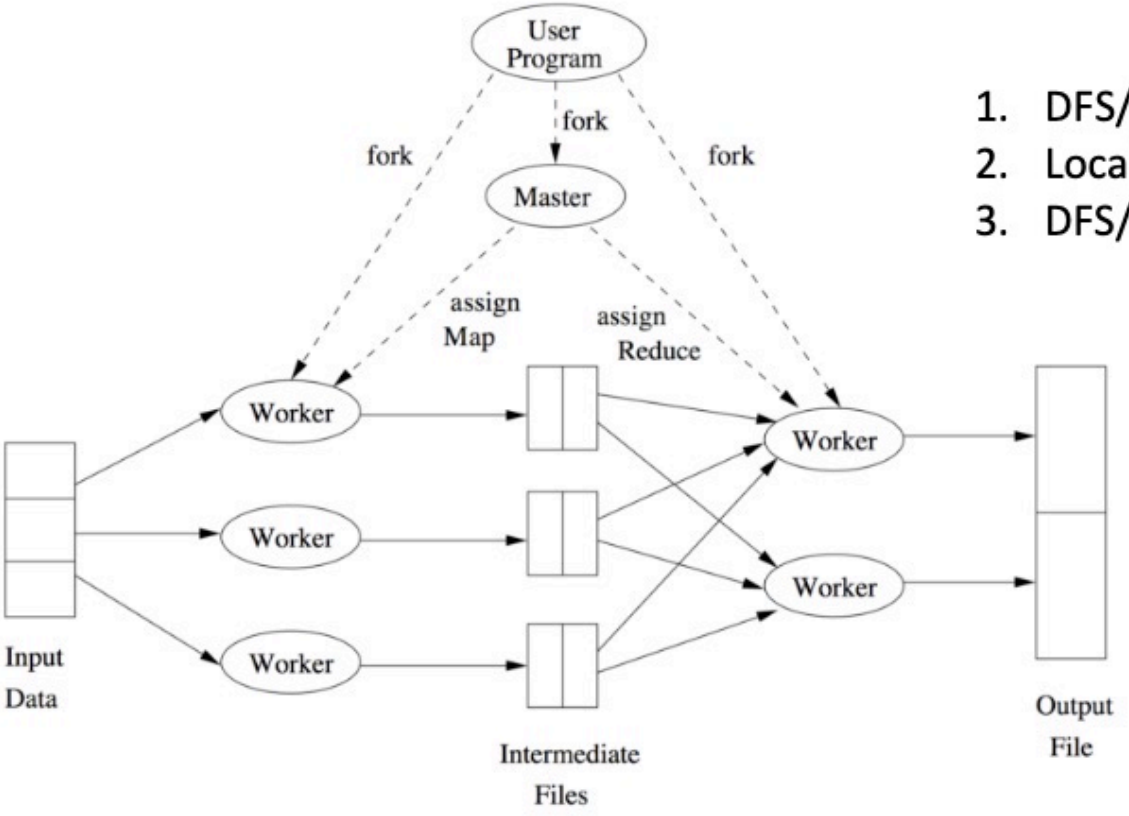
Thanks for source slide and material to: Dr. Heather Miller
https://www.coursera.org/learn/scala-spark-big-data/home/welcome

MapReduce Review



Question: Concerning the architecture, where does the MapReduce program store 1. **input data**, 2. **intermediate files**, 3. **output data**?

MapReduce Review



Question: Concerning the architecture, where does the MapReduce program store 1. **input data**, 2. **intermediate files**, 3. **output data**?

MapReduce Review

Write **the key-value pair in the Map and Reduce tasks** for joining two tables:

Table 1: Transactions

transaction_id	user_id	date
1	a	d1
2	b	d1
3	c	d2
4	a	d3
5	b	d3

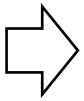
Table 2: Users

user_id	income	age
1	10,000	23
2	100,000	42
3	50,000	34

MapReduce Review

Write **Map tasks** for joining two tables:

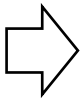
transaction_id	user_id	date
1	a	d1
2	b	d1
3	c	d2
4	a	d3
5	b	d3



relation name

a: “Transactions”, (1, a, d1)
b: “Transactions”, (2, b, d1)
c: “Transactions”, (3, c, d2)
a: “Transactions”, (4, a, d3)
b: “Transactions”, (5, b, d3)

user_id	income	age
a	10,000	23
b	100,000	42
c	50,000	34



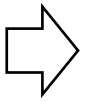
a: “Users”, (a, 10,000, 23)
b: “Users”, (b, 100,000, 42)
c: “Users”, (c, 50,000, 34)

MapReduce Review

Write **Reduce tasks** for joining two tables:

a: “Transactions”, (1, a, d1)
b: “Transactions”, (2, b, d1)
c: “Transactions”, (3, c, d2)
a: “Transactions”, (4, a, d3)
b: “Transactions”, (5, b, d3)

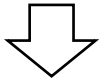
a: “Users”, (a, 10000, 23)
b: “Users”, (b, 100000, 42)
c: “Users”, (c, 50000, 34)



Example: Reduce for key ”a”

a: “Transactions”, (1, a, d1)
a: “Transactions”, (4, a, d3)

a: “Users”, (a, 10,000, 23)

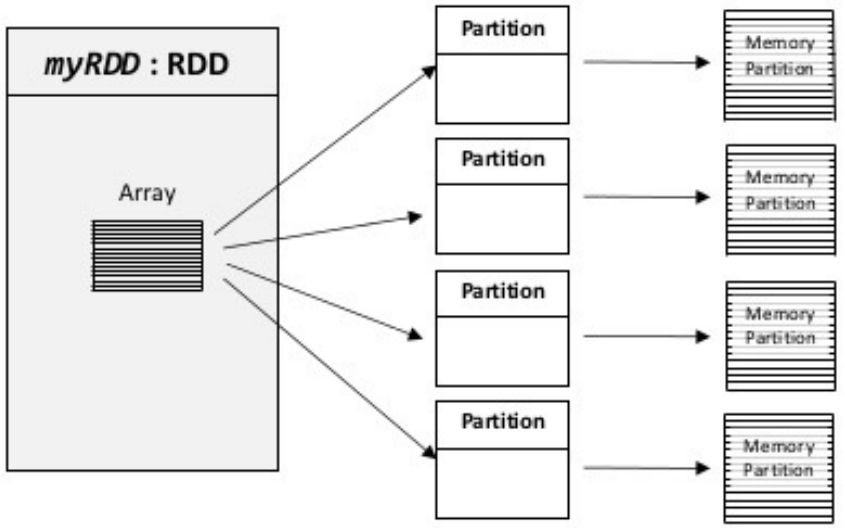


(1, a, d1, 10000, 23)
(4, a, d3, 10000, 23)

How Does Spark Work?

- Resilient Distributed Datasets (RDDs)
- An **immutable**, **in-memory collection** of objects
- Each RDD can be split into multiple partitions, which in turn are computed on different nodes of the cluster

- RDDs are like collections
  - RDD[T] and List[T]



Common Transformations

map — **map[T](f: A=>B): RDD[T]**
Apply function to each element in the RDD and return an RDD of the result.

flatmap — **flatmap[T](f: A=>B): RDD[T]**
Apply function to each element in the RDD and return an RDD of the result, but output is flattened.

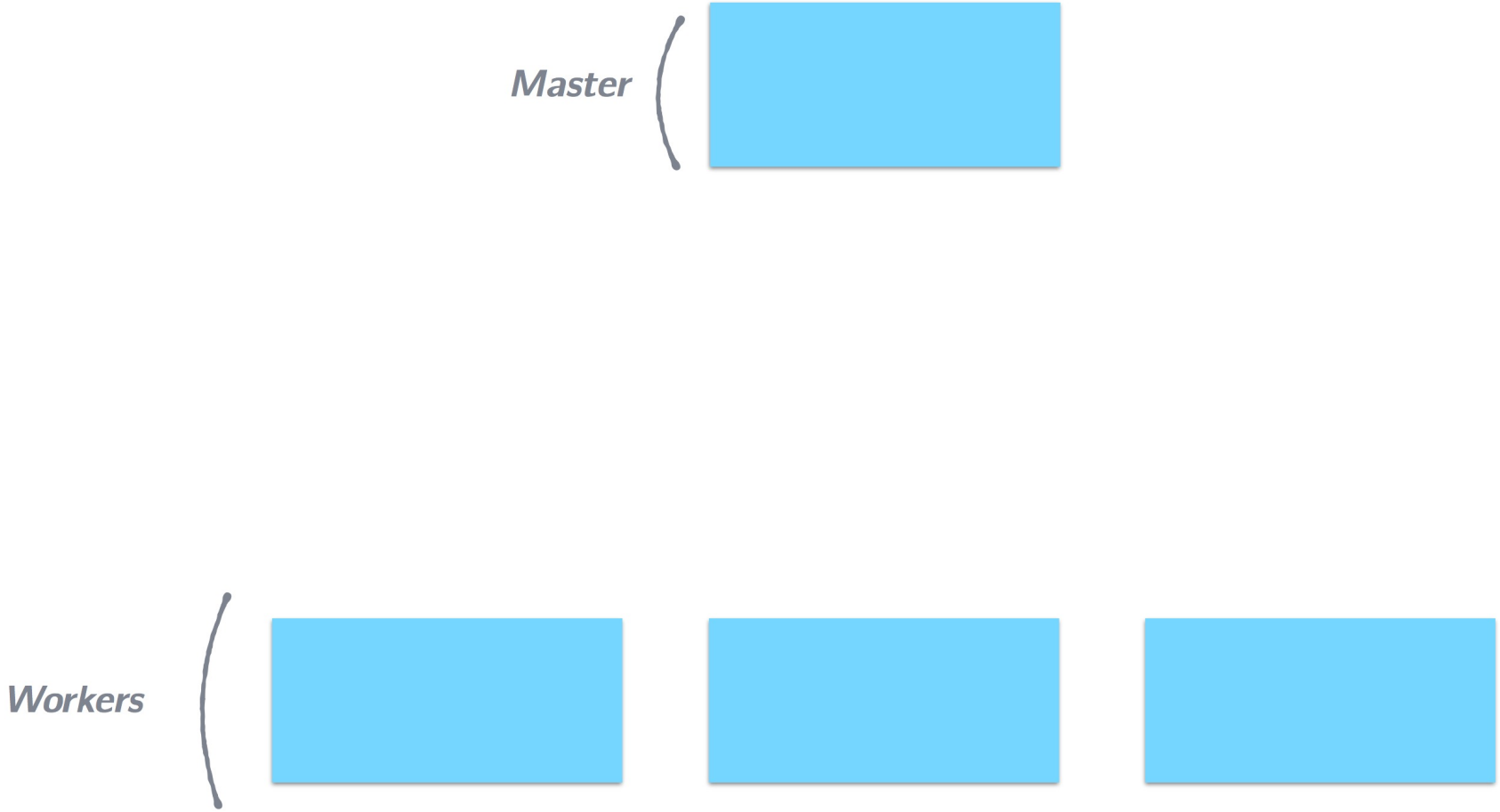
filter — **filter[T](pred: A=>Boolean): RDD[T]**
Apply predicate function, pred, to each element in the RDD and return an RDD of elements that passed the condition.

distinct — **distinct():RDD[T]**
Return an RDD with duplicates removed

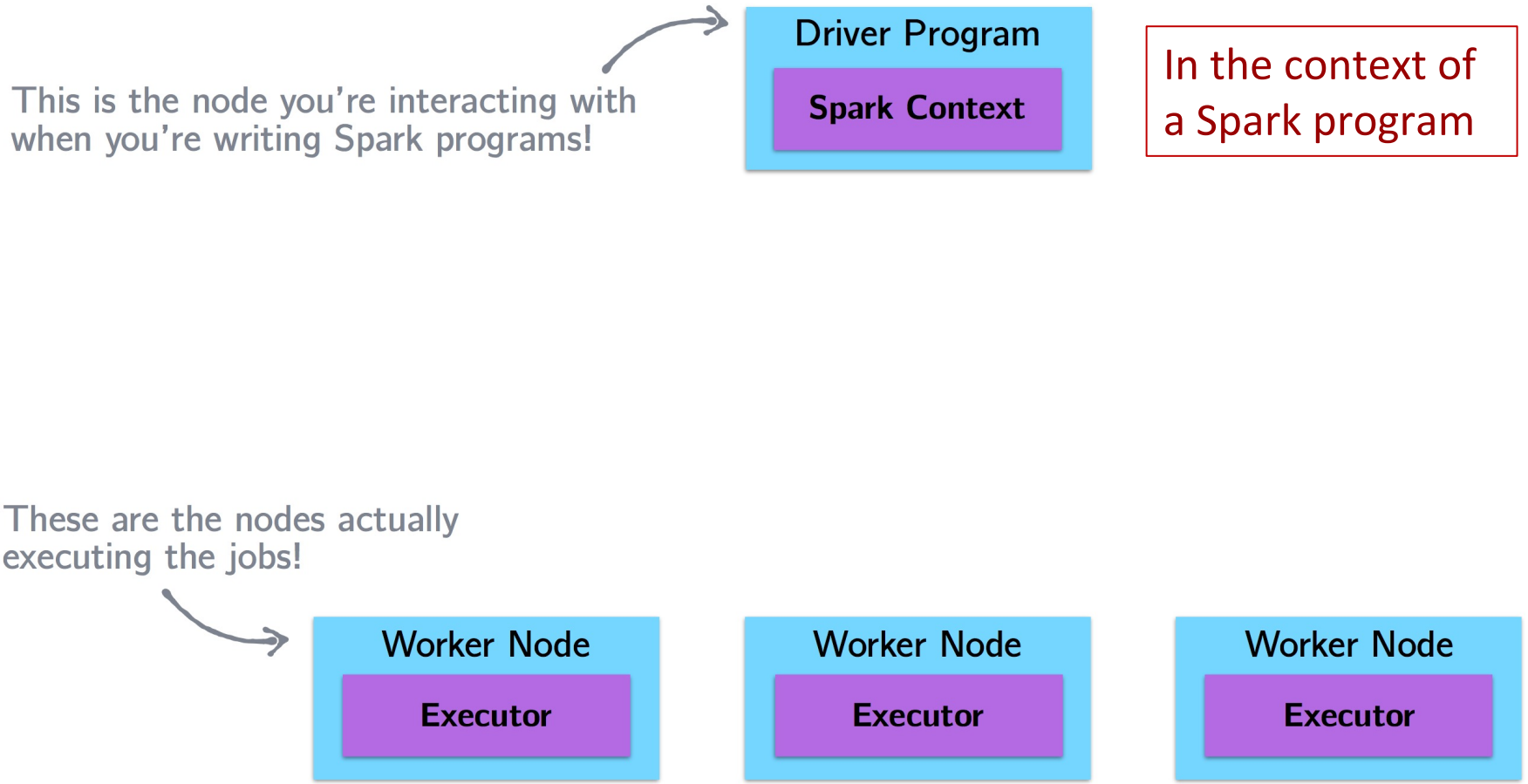
Common Actions

collect	**collect: Array[T]** Return all elements from RDD.
count	**count(): Long** Return the number of elements in the RDD.
take	**take(num: Int): Array[T]** Return the first <num> elements of the RDD.
reduce	**reduce(op: (A, A) => A): A** Combine the elements in the RDD together using op function and return result.
foreach	**foreach(f: A => Unit): Unit** Apply function to each element in the RDD and return Unit.

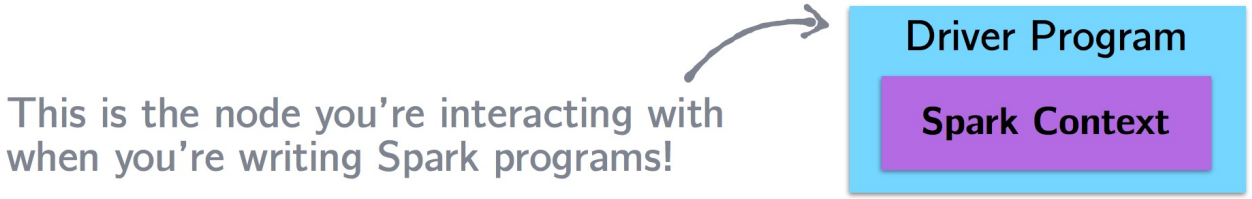
How Spark jobs are Executed



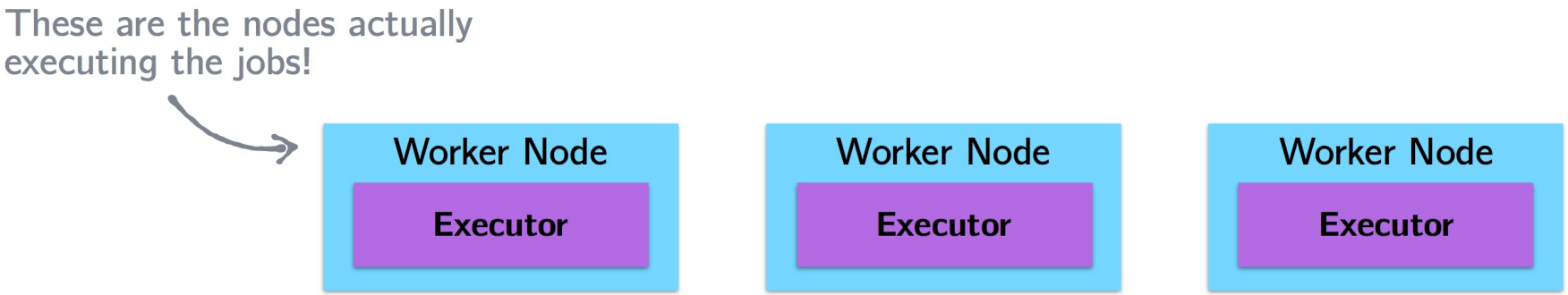
How Spark jobs are Executed



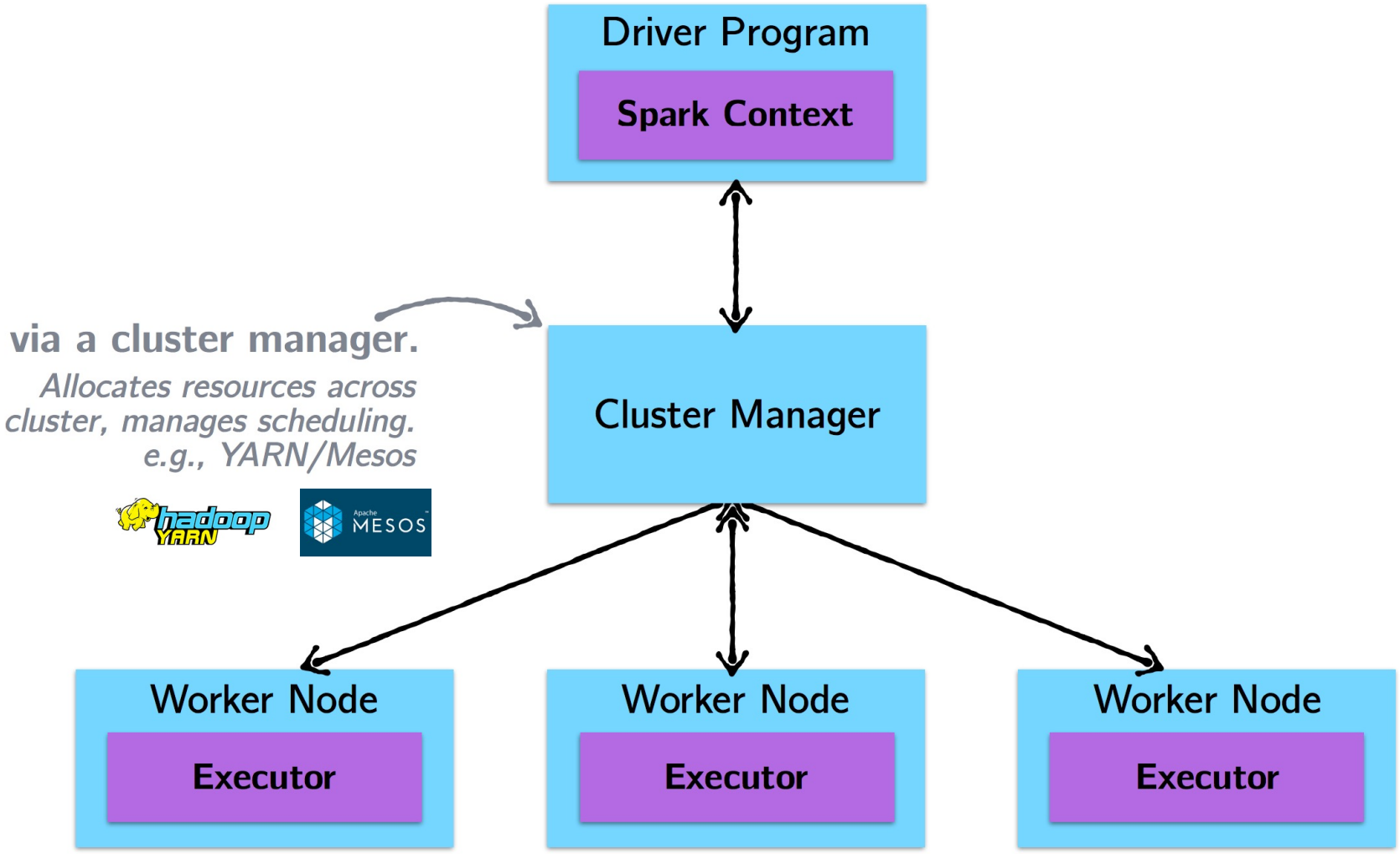
How Spark jobs are Executed



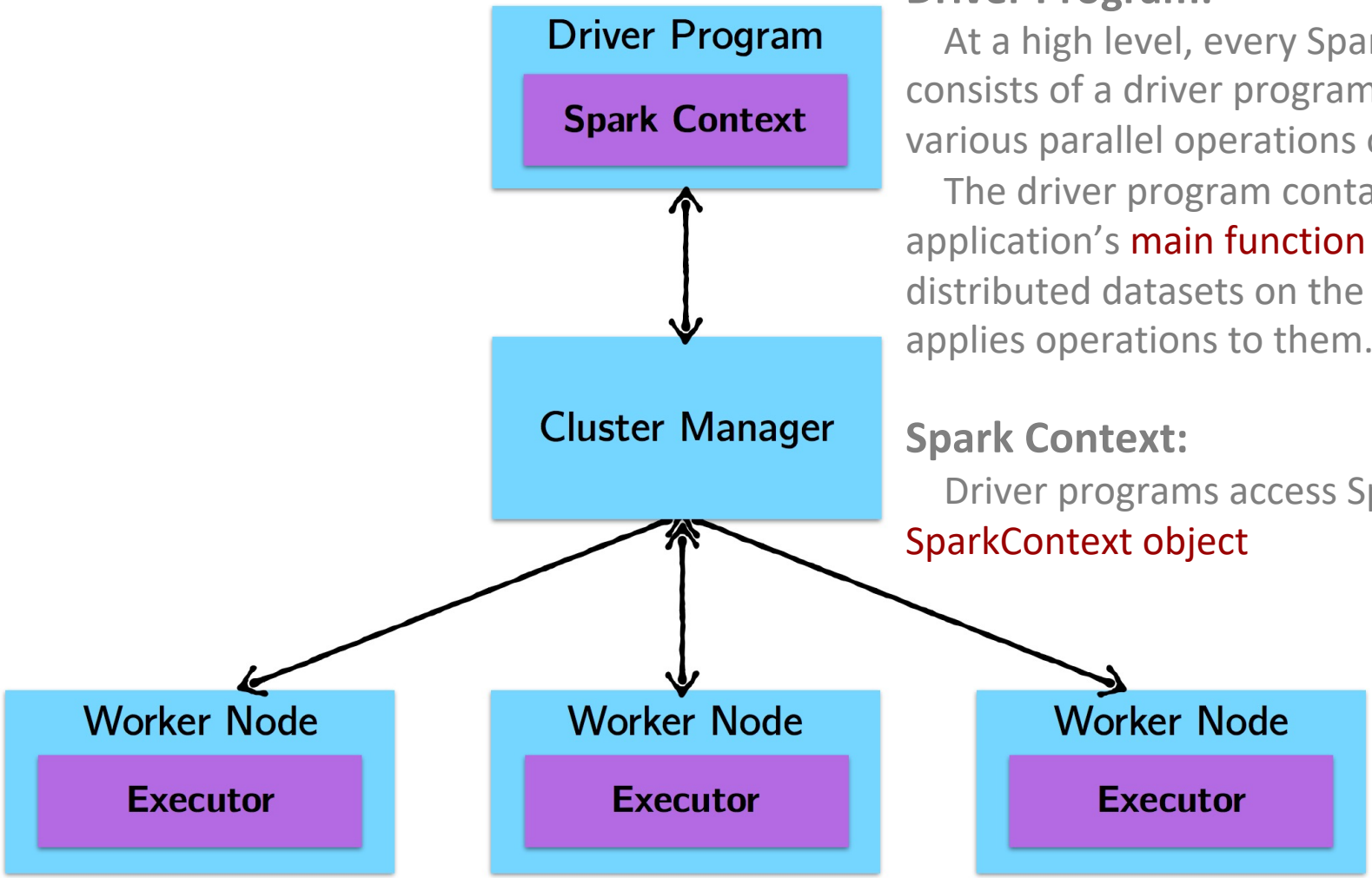
How do they communicate?



How Spark jobs are Executed



How Spark jobs are Executed



Driver Program:

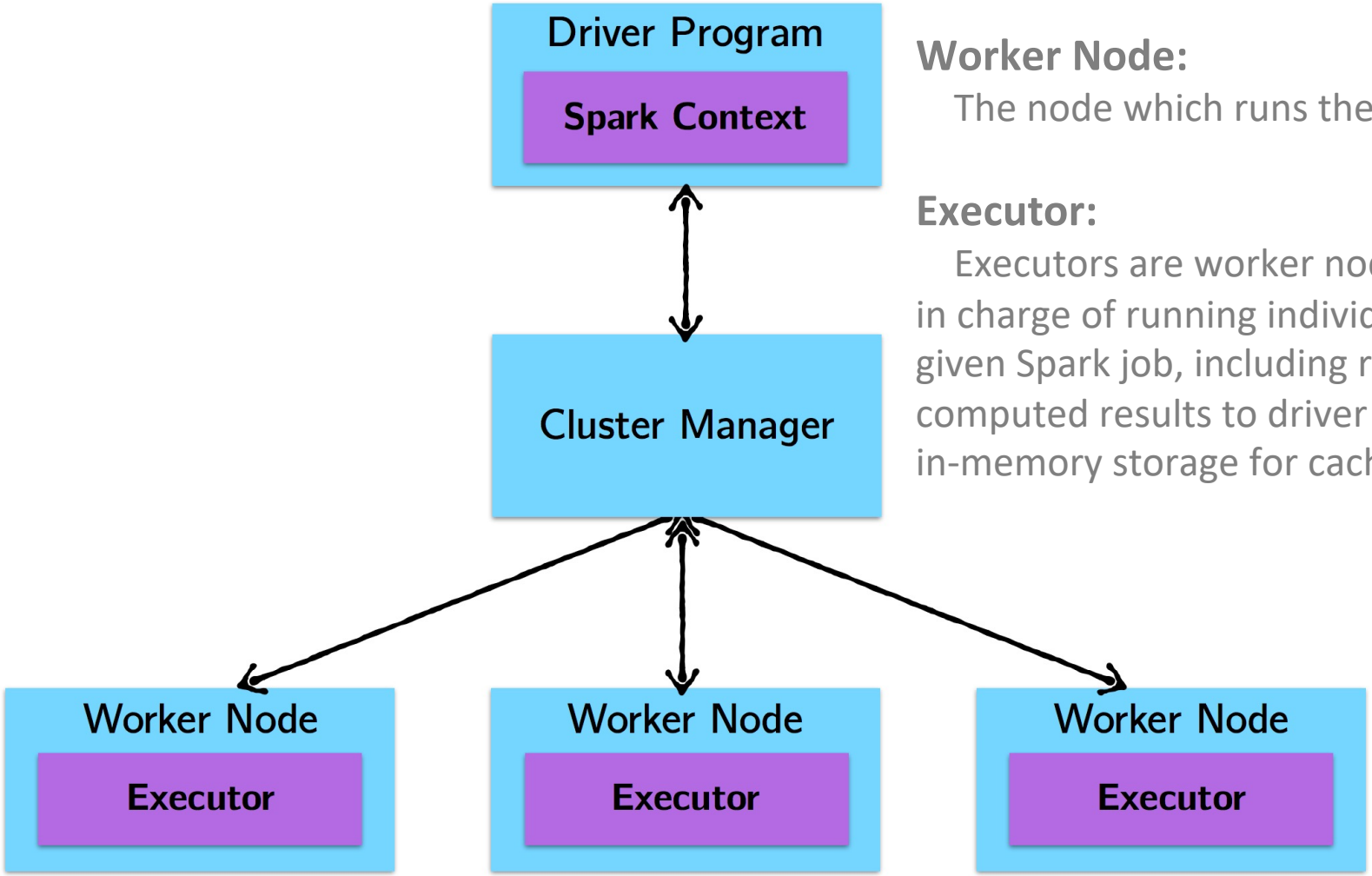
At a high level, every Spark application consists of a driver program that launches various parallel operations on the cluster.

The driver program contains your application's main function and defines distributed datasets on the cluster, then applies operations to them.

Spark Context:

Driver programs access Spark through a SparkContext object

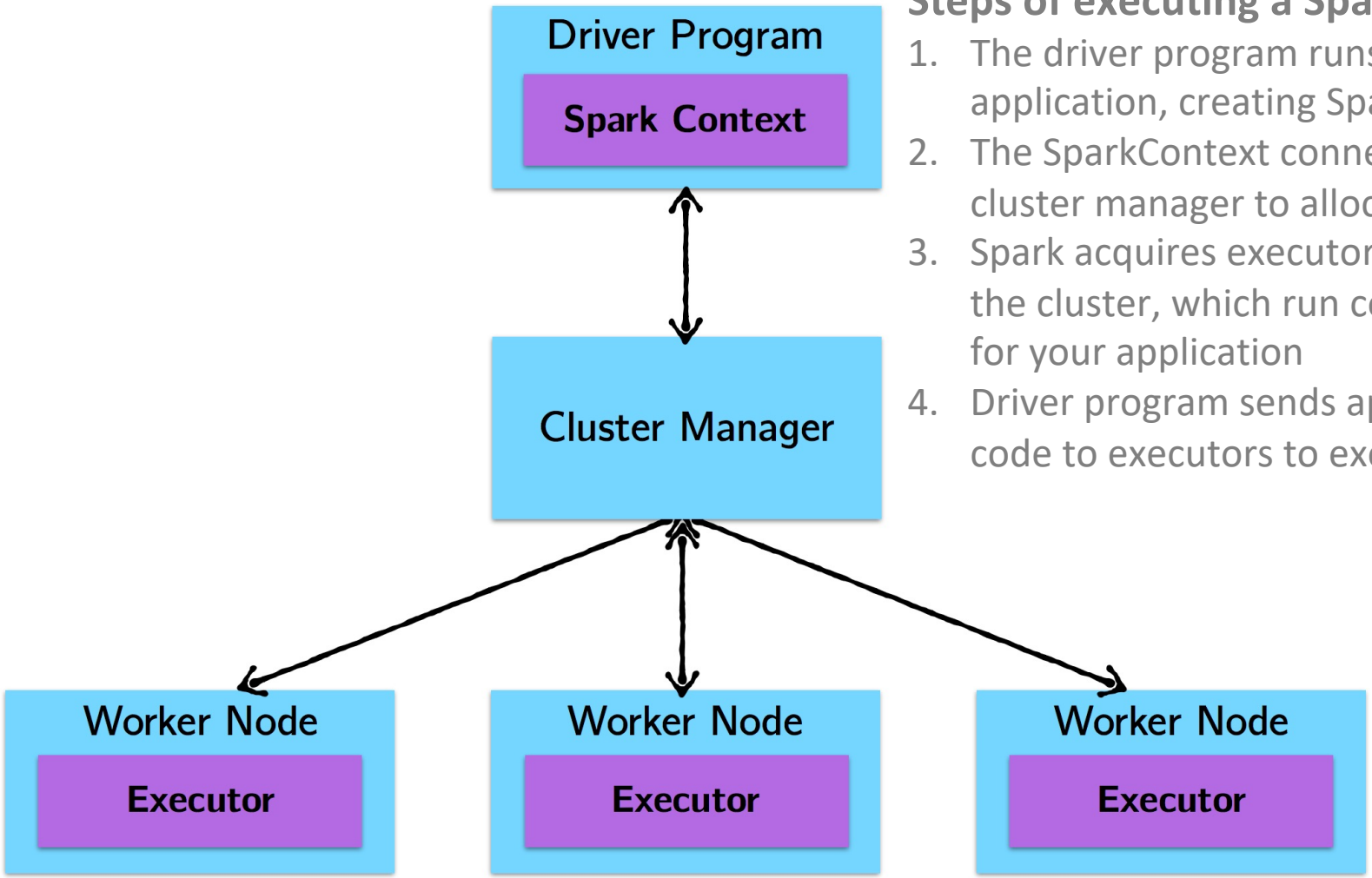
How Spark jobs are Executed



Worker Node:
The node which runs the application

Executor:
Executors are worker nodes' processes in charge of running individual task in given Spark job, including returning computed results to driver and providing in-memory storage for cached RDDs

How Spark jobs are Executed



Steps of executing a Spark program:

1. The driver program runs the Spark application, creating SparkContext
2. The SparkContext connects to a cluster manager to allocate resources
3. Spark acquires executors on nodes in the cluster, which run computations for your application
4. Driver program sends application code to executors to execute

Cluster Topology

- A simple example with *println*

```
case class Person(name: String, age: Int)

val people: RDD[Person]
people.foreach(println)
```

What happens?

Cluster Toplogy

- A simple example with *println*

```
case class Person(name: String, age: Int)

val people: RDD[Person]
people.foreach(println)
```

On the driver: Nothing.

Why? Recall that foreach is **an action**, with **return type Unit**. Therefore, it will be eagerly executed on the executors. Thus, any calls to *println* are happening on the worker nodes and are not visible in the drive node.

Cluster Topology

- Another simple example with *take*

```
case class Person(name: String, age: Int)

val people: RDD[Person]
people.foreach(println)

val first10 = people.take(10)
```

Where will *first10* end up?

Cluster Topology

- Another simple example with *take*

```
case class Person(name: String, age: Int)

val people: RDD[Person]
people.foreach(println)

val first10 = people.take(10)
```

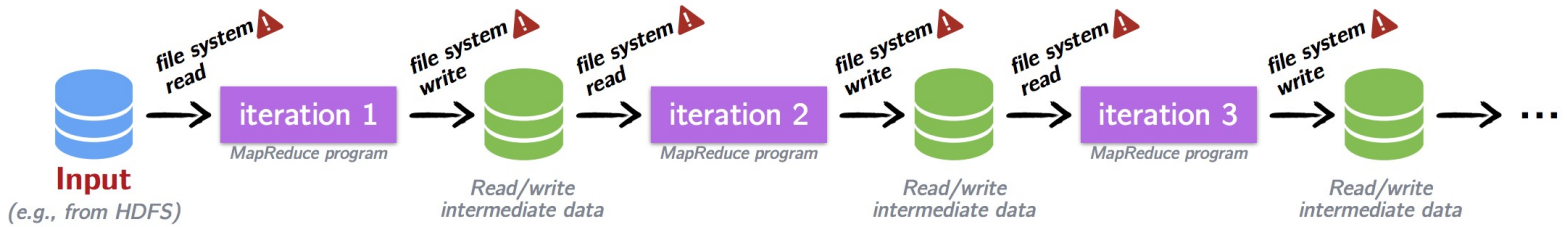
Where will *first10* end up? The driver program.

In general, executing an action involves communication between worker nodes and the node running the driver program.

Iteration

- Most data science problems **involve iterations**

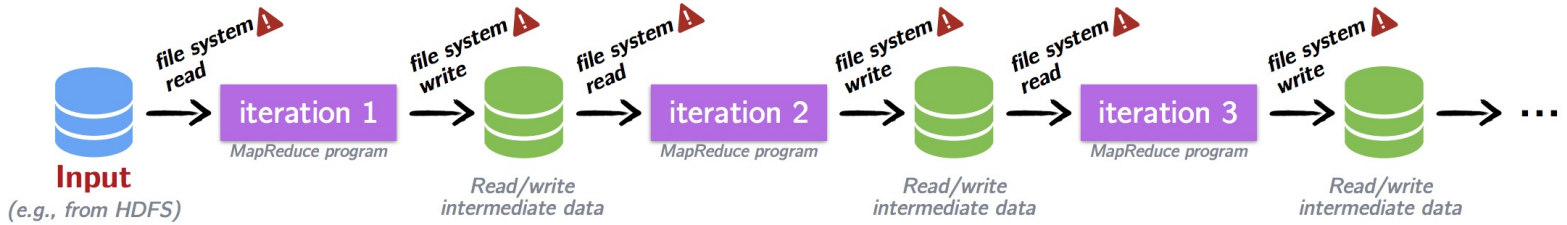
Iteration in Hadoop:



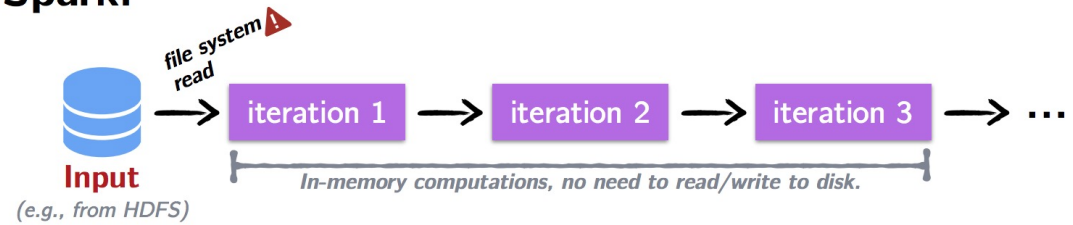
Iteration

- Most data science problems **involve iteration**

Iteration in Hadoop:

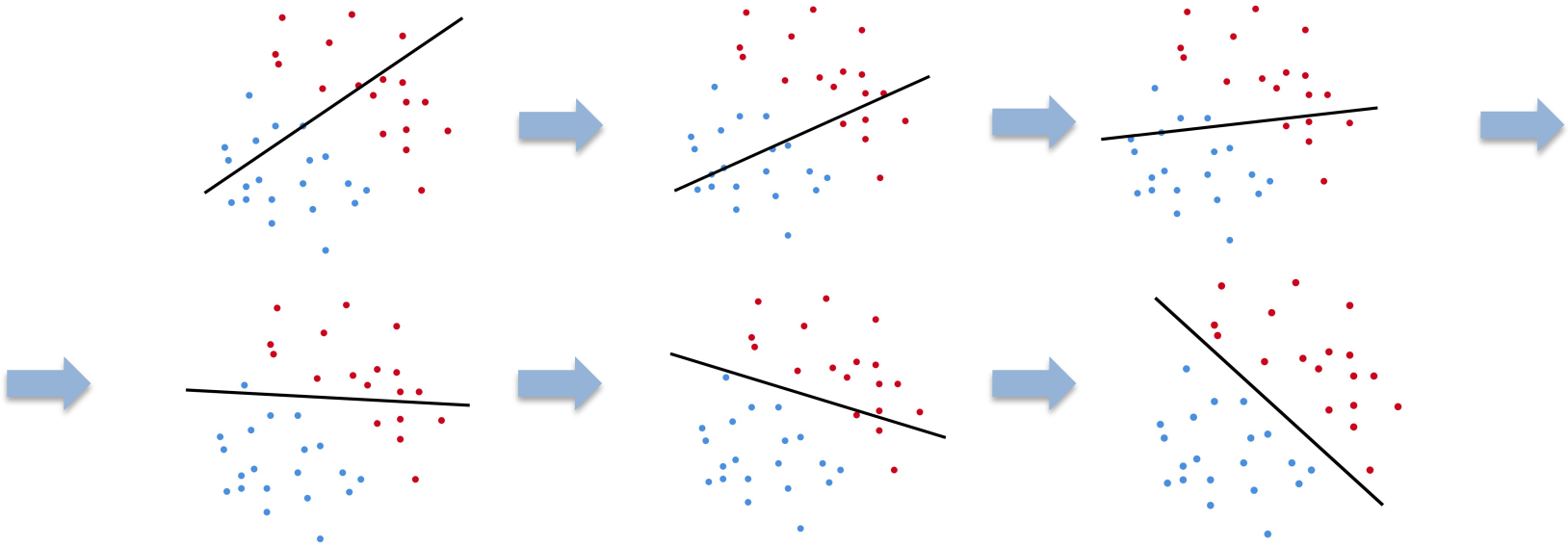


Iteration in Spark:



Iteration
Example: Logistic Regression

- Logistic regression is **an iterative algorithm** typically used for classification. Like most classification algorithms, it **updates weights iteratively** based on the training data.



Iteration
Example: Logistic Regression

- Logistic regression is **an iterative algorithm** typically used for classification. Like most classification algorithms, it **updates weights iteratively** based on the training data.

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

Iteration
Example: Logistic Regression

- Logistic regression sample code:

```
val points = sc.textFile(...).map(parsePoint) // case class Point(x: Double, y: Double)
var w = Vector.zero(d)
for(i <- 1 to numIterations) {
  val gradient = points.map {p =>
    g(p) // Apply the function of logistic regression
  }.reduce(_+_)
  w -= alpha * gradient
}
```

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

Iteration
Example: Logistic Regression

- Logistic regression sample code:

```
val points = sc.textFile(...).map(parsePoint) // case class Point(x: Double, y: Double)
var w = Vector.zero(d)
for(i <- 1 to numIterations) {
  val gradient = points.map {p =>
    g(p) // Apply the function of logistic regression
  }.reduce(_+_)
  w -= alpha * gradient
}
```

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

What is the weakness for this code?

Iteration
Example: Logistic Regression

- Logistic regression sample code:

```
val points = sc.textFile(...).map(parsePoint) // case class Point(x: Double, y: Double)
var w = Vector.zero(d)
for(i <- 1 to numIterations) {
  val gradient = points.map {p =>
    g(p) // Apply the function of logistic regression
  }.reduce(_+_)
  w -= alpha * gradient
}
```

Spark starts the execution when the action *reduce* is applied

Iteration
Example: Logistic Regression

- Logistic regression sample code:

```
val points = sc.textFile(...).map(parsePoint) // case class Point(x: Double, y: Double)
var w = Vector.zero(d)
for(i <- 1 to numIterations) {
  val gradient = points.map {p =>
    g(p) // Apply the function of logistic regression
  }.reduce(_+_)
  w -= alpha * gradient
}
```

***points* is being re-evaluated upon every iteration!**
Unnecessary!

Caching and Persistence

- By default, RDDs are recomputed each time you run an action on them. This can be expensive (time-consuming) if you need to use a dataset more than once.

Spark allows you to control what is cached in memory
use *persist()* or *cache()*

cache() : using the default storage level
persist(): can pass the storage level as a parameter,
e.g., “MEMORY_ONLY”, “MEMORY_AND_DISK”

Iteration
Example: Logistic Regression

- Logistic regression sample code:

```
val points = sc.textFile(...).map(parsePoint).persist() // case class Point(x: Double, y: Double)
var w = Vector.zero(d)
for(i <- 1 to numIterations) {
  val gradient = points.map {p =>
    g(p) // Apply the function of logistic regression
  }.reduce(_+_)
  w -= alpha * gradient
}
```

***points* is evaluated once and is cached in memory.**
It can be re-used on each iteration.