Algorithms and Applications of Data Mining

Yijun Lin
yijunlin@usc.edu

About This Course

- Spring 2021, Friday, 6-8 PM PST
- Instructor: Yao-Yi Chiang
- TA: Yijun Lin
  - Office Hour Sat. 7-9 PM PST
- Syllabus:

	Topic	Readings and Assignments	Deliverables/Due Dates
Week 1	Introduction to Data Mining	Ch1: Data Mining and	
Week 2	MapReduce	Ch2: Large-Scale File Systems and Map-Reduce	Homework 1 assigned
Week 3	Frequent itemsets and Association rules	Ch6: Frequent itemsets,	Homework 2 assigned
Week 4	Clustering	Ch7: Clustering	Homework 1 due
Week 5	Recommendation Systems: Content-based	Ch9: Recommendation systems	Homework 2 due, Homework 3 assigned
Week 6	Recommendation Systems: Collaborative Filtering	Ch9: Recommendation systems	Homework 3 due

Assignments

- Theoretical and programming questions
  - Real-world datasets

- Homework 1 - basic spark operations
- Homework 2 - mining frequent itemset
- Homework 3 - recommender system

- Optional – clustering

Config Environment

- **Python** is required for all the assignments
- Implementing with Apache Spark Framework
  - python=3.7
  - pyspark=3.0.1
  - git clone https://github.com/linyijun/spark-tutorial.git
- Install miniconda/anaconda
  - conda env create -f spark-env.yml python=3.7
- Install PyCharm

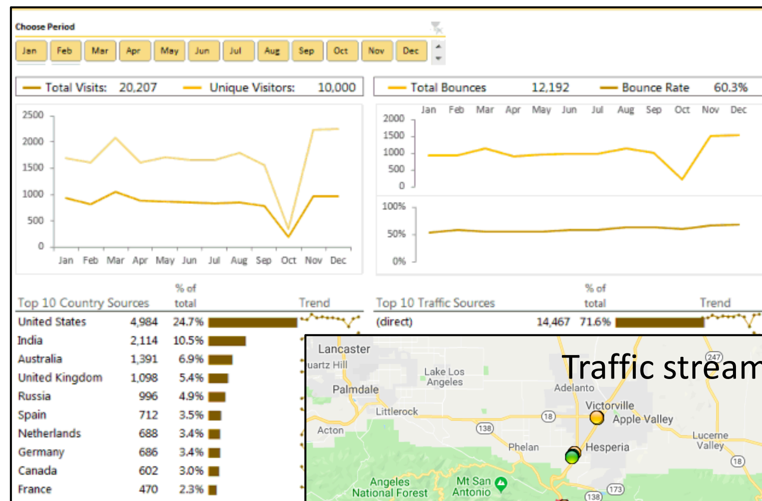
Introduction to Spark

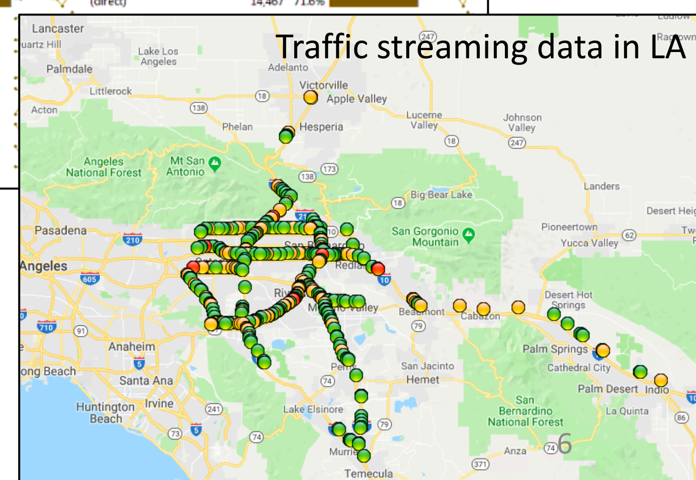
Thanks for source slide and material to: Dr. Heather Miller
https://www.coursera.org/learn/scala-spark-big-data/home/welcome

What is Spark?

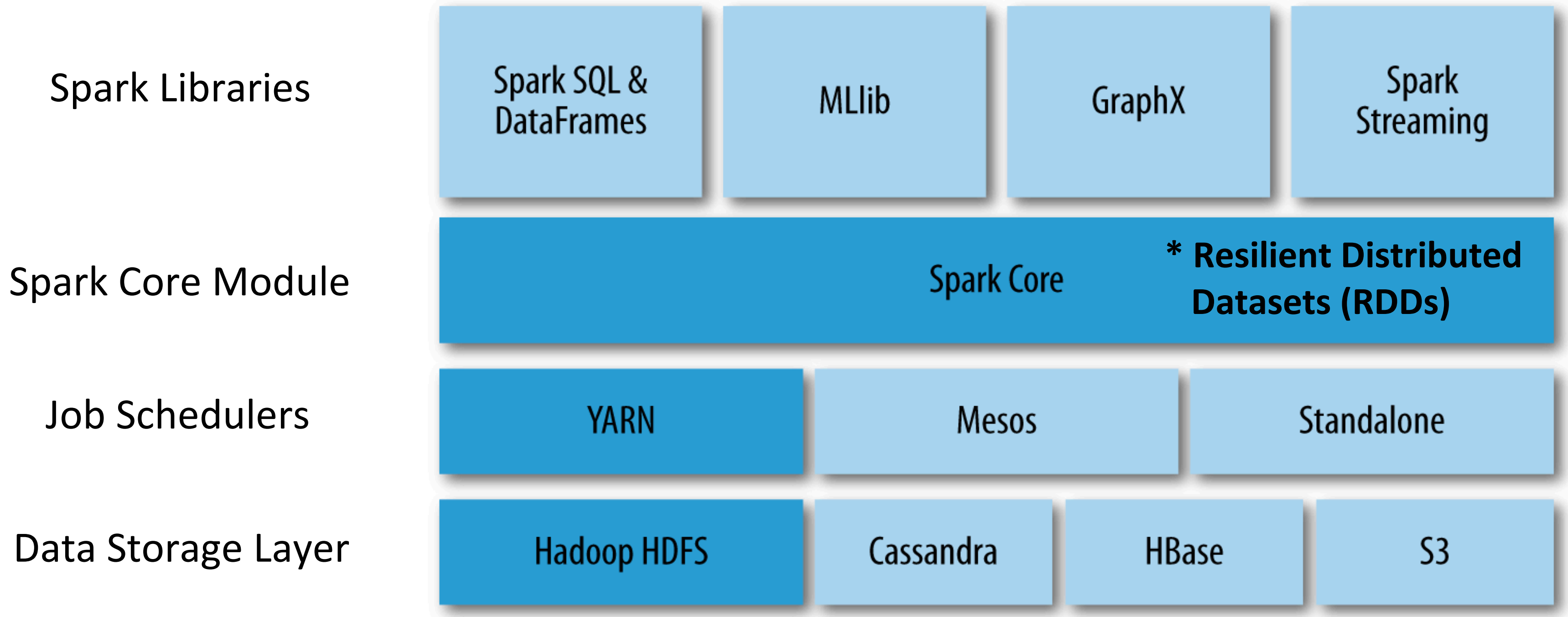
- **Apache Spark** is a unified analytics engine for large-scale data processing

- Application areas
  - Interactive Data Query
  - Real-time Data Analysis
  - Streaming Data Processing





Spark Stack

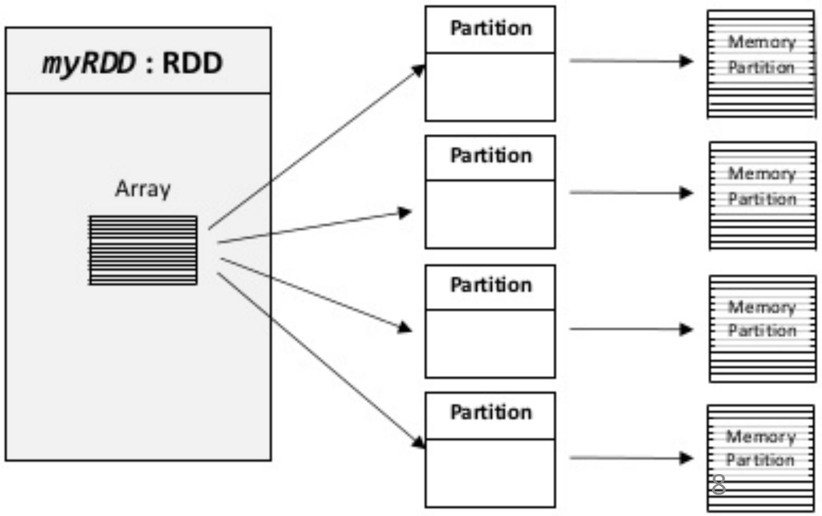


Spark Libraries
Spark Core Module
Job Schedulers
Data Storage Layer
Spark SQL & DataFrames
MLlib
GraphX
Spark Streaming
Spark Core
* Resilient Distributed Datasets (RDDs)
YARN
Mesos
Standalone
Hadoop HDFS
Cassandra
HBase
S3

Resilient Distributed Datasets (RDDs)

- An RDD is 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 seem a lot like Scala collections
  - RDD[T] and List[T]



How to create an RDD

- RDDs can be created in two ways:
  - Creating from a SparkContext object
  - Transforming from an existing RDD

How to create an RDD (Cont.)

- Creating from a **SparkContext** object
  - Can be thought as your handle to the Spark cluster
  - Represents the connection to a Spark cluster

```
if __name__ == '__main__':

    sc_conf = pyspark.SparkConf() \
        .setAppName('task1') \
        .setMaster('local[*]') \
        .set('spark.driver.memory', '8g') \
        .set('spark.executor.memory', '4g')

    sc = pyspark.SparkContext(conf=sc_conf)
    sc.setLogLevel("OFF")
```

How to create an RDD (Cont.)

- Creating from a **SparkContext** object
  - **parallelize:** convert a local Scala collection to an RDD

```
a_list = ['you', 'jump', 'I', 'jump', '']
a_rdd = sc.parallelize(a_list)  # RDD[String]
```

How to create an RDD (Cont.)

- Creating from a **SparkContext** object
  - **parallelize:** convert a local Scala collection to an RDD

```
a_list = ['you', 'jump', 'I', 'jump', '']
a_rdd = sc.parallelize(a_list)  # RDD[String]
```

  - **textFile**: read a file from HDFS or local file system

```
input_file = 'work-count-sample-doc.txt'
text_rdd = sc.textFile(input_file)
```

How to create an RDD (Cont.)

- Transforming from an existing RDD
  - E.g., calling a *map operation* on an existing RDD, it will return a new RDD

```
# call a map operation on an RDD
length_rdd = word_rdd.map(lambda x: len(x))  # RDD[Int]
```

RDD Operations

- Transformations
  - E.g., map, filter, ...

```
# call a map operation on an RDD
length_rdd = word_rdd.map(lambda x: len(x))  # RDD[Int]
```

- Actions
  - E.g., collect, reduce ...

```
a_coll = a_rdd.collect()  # RDD -> collection
print(a_coll)  # ['you', 'jump', 'I', 'jump', '']
```

Transformations VS Actions

- Transformations
  - Return new RDDs as results
  - They are **lazy**, the result RDD is not immediately computed

- Actions
  - Compute a result based on an RDD, and returned
  - They are **eager**, the result is immediately computed

Transformations VS Actions

- Transformations
  - Return new RDDs as results
  - They are **lazy**, the result RDD is not immediately computed

```
# call a map operation on an RDD
length_rdd = word_rdd.map(lambda x: len(x))  # RDD[Int]
```

- Actions
  - Compute a result based on an RDD, and returned
  - They are **eager**, the result is immediately computed

```
a_coll = a_rdd.collect()  # RDD -> collection
print(a_coll)  # ['you', 'jump', 'I', 'jump', '']
```

Transformations VS Actions

- Transformations
  - Return new RDDs as results
  - They are **lazy**, the result RDD is not immediately computed
- Actions
  - Compute a result based on an RDD, and returned
  - They are **eager**, the result is immediately computed

Laziness / Eagerness is how we can limit network communication using the programming model

Example

- Consider the following example:

```
a_list = ['you', 'jump', 'I', 'jump', '']
# create an RDD from a list
a_rdd = sc.parallelize(a_list)  # RDD[String]
# call a map operation RDD
a_len_rdd = a_rdd.map(lambda x: len(x))  # RDD[Int]
```

What has happened on the cluster at this point?

?

Example (Cont.)

- Consider the following example:

```
a_list = ['you', 'jump', 'I', 'jump', '']
# create an RDD from a list
a_rdd = sc.parallelize(a_list)  # RDD[String]
# call a map operation RDD
a_len_rdd = a_rdd.map(lambda x: len(x))  # RDD[Int]
```

What has happened on the cluster at this point?

Nothing. Execution of map (a transformation) is deferred.

Example (Cont.)

- Consider the following example:

```
a_list = ['you', 'jump', 'I', 'jump', '']
# create an RDD from a list
a_rdd = sc.parallelize(a_list)  # RDD[String]
# call a map operation RDD
a_len_rdd = a_rdd.map(lambda x: len(x))  # RDD[Int]
```

What has happened on the cluster at this point?
Nothing. Execution of *map* (a transformation) is deferred.

How to ensure this computation is done on the cluster?

Example (Cont.)

- Consider the following example:

```
a_list = ['you', 'jump', 'I', 'jump', '']
# create an RDD from a list
a_rdd = sc.parallelize(a_list)  # RDD[String]
# call a map operation RDD
a_len_rdd = a_rdd.map(lambda x: len(x))  # RDD[Int]

total_len = a_len_rdd.reduce(lambda a, b: a + b)  # 12
```

add an action, *reduce*

Spark starts the execution when an action is called

Return the total number of characters in the entire RDD of strings

Benefits of Laziness

- Another example:

```
input_file = 'work-count-sample-doc.txt'
text_rdd = sc.textFile(input_file)
word_rdd = text_rdd.flatMap(lambda x: x.split(' ')).take(10)
```

Benefits of Laziness

- Another example:

```
input_file = 'work-count-sample-doc.txt'
text_rdd = sc.textFile(input_file)
word_rdd = text_rdd.flatMap(lambda x: x.split(' ')).take(10)
```

- The execution of *filter* is deferred until the *take* action happens
  – Spark will not compute intermediate RDDs. As soon as 10 elements of the filtered RDD have been computed, *logsWithErrors* is done.

Benefits of Laziness

- Another example:

```
input_file = 'work-count-sample-doc.txt'
text_rdd = sc.textFile(input_file)
word_rdd = text_rdd.flatMap(lambda x: x.split(' ')).take(10)
```

- The execution of *filter* is deferred until the *take* action happens
  - Spark will not compute intermediate RDDs. As soon as 10 elements of the filtered RDD have been computed, *logsWithErrors* is done

- Spark leverages this by analyzing and optimizing the chain of operations before executing it
  - Spark saves time and space to compute elements of the unused result of the *filter operation*

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 Transformations

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.

Common Transformations

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.

Common Actions

count	**count(): Long** Return the number of elements in the RDD.

Common Actions

foreach **foreach(f: A => Unit): Unit**
Apply function to each element in the RDD, and return Unit.

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

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

Cluster Topology

- A simple example with *println*

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

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

What happens?

How Spark jobs are Executed

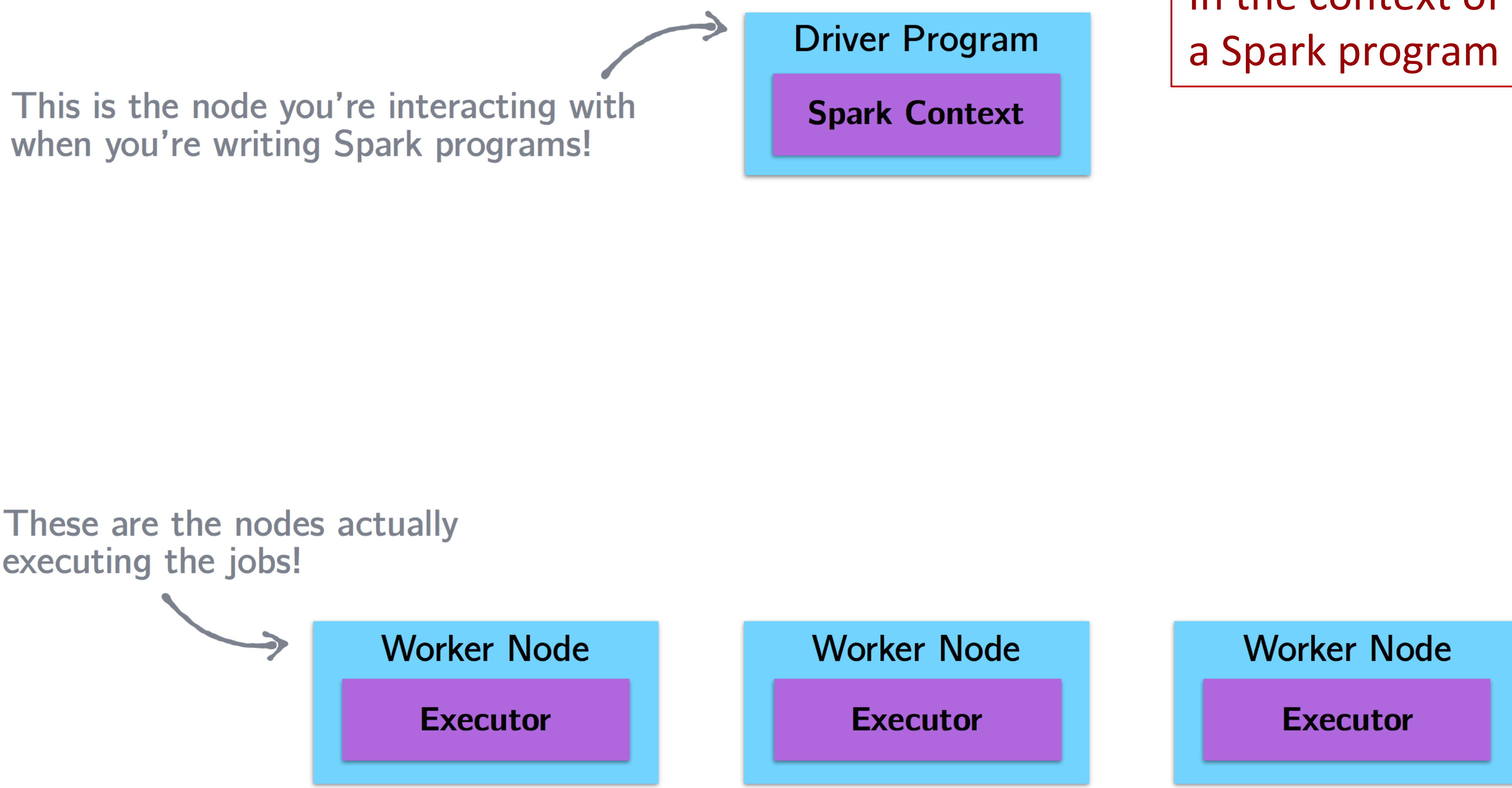
Master-Worker Topology

Master

Workers

How Spark jobs are Executed

In the context of a Spark program



How Spark jobs are Executed

This is the node you're interacting with when you're writing Spark programs!

Driver Program

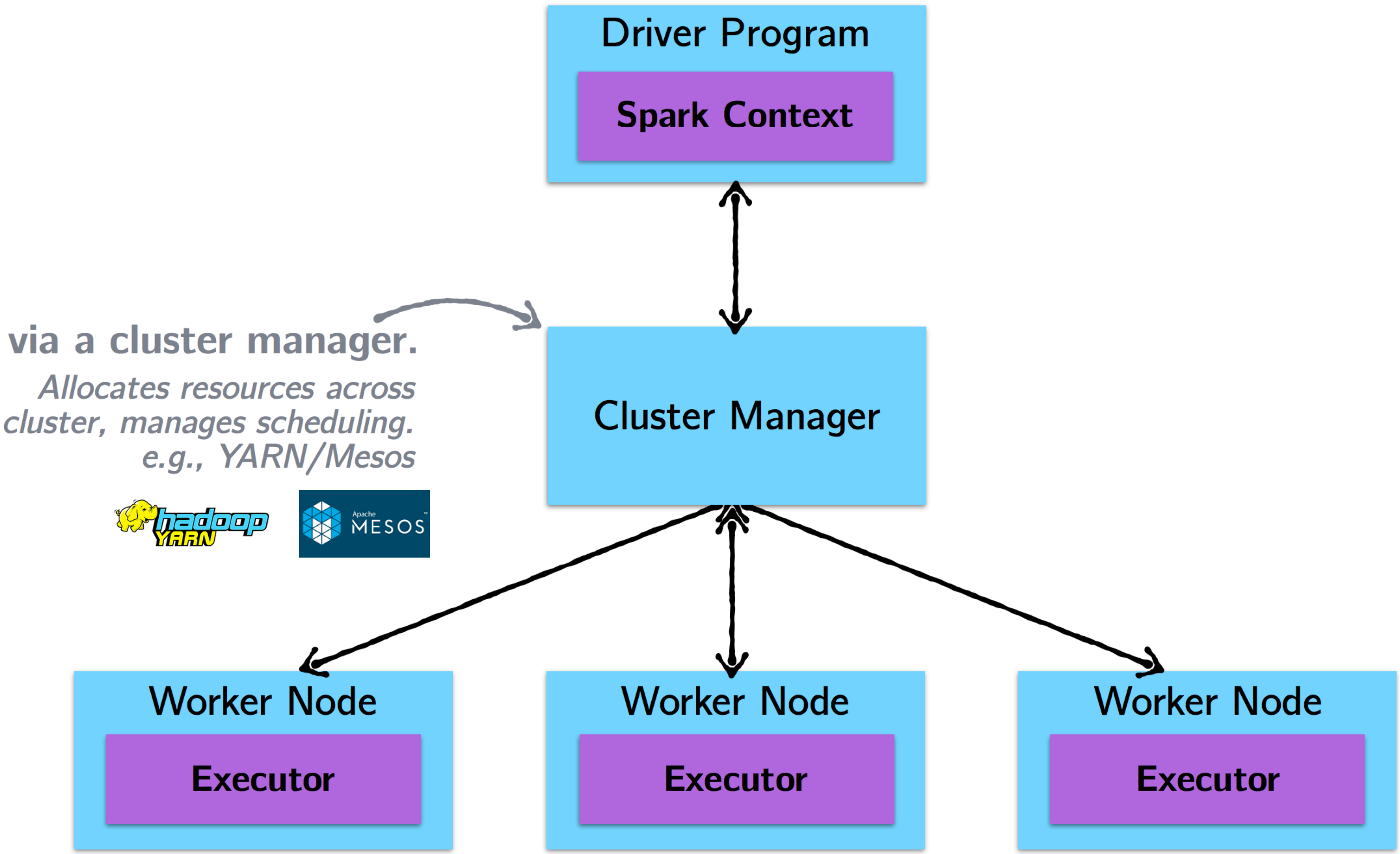
Spark Context

How do they communicate?

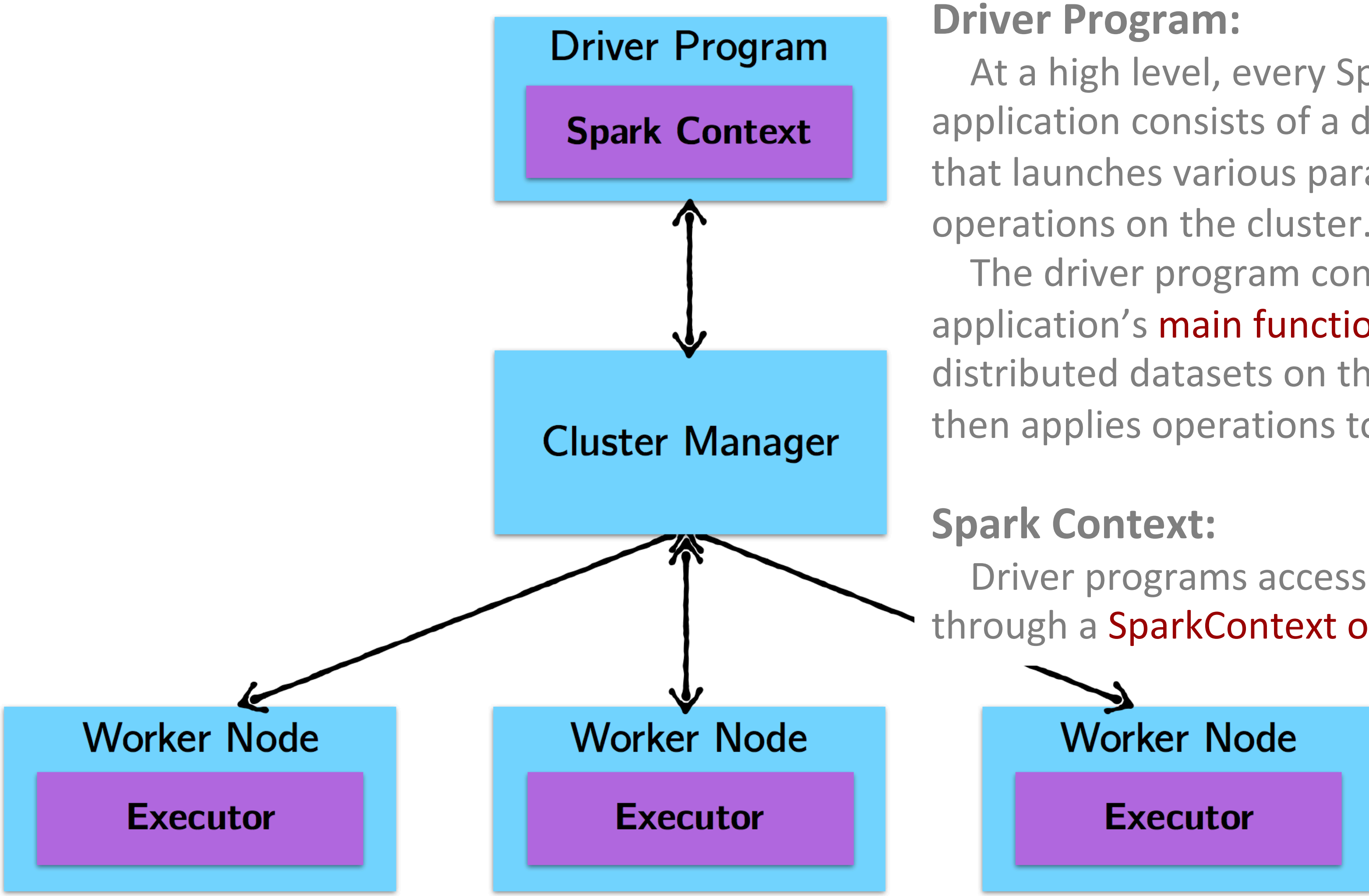
These are the nodes actually executing the jobs!

Worker Node	Worker Node	Worker Node
Executor	**Executor**	**Executor**

How Spark jobs are Executed



How Spark jobs are Executed



Driver Program:

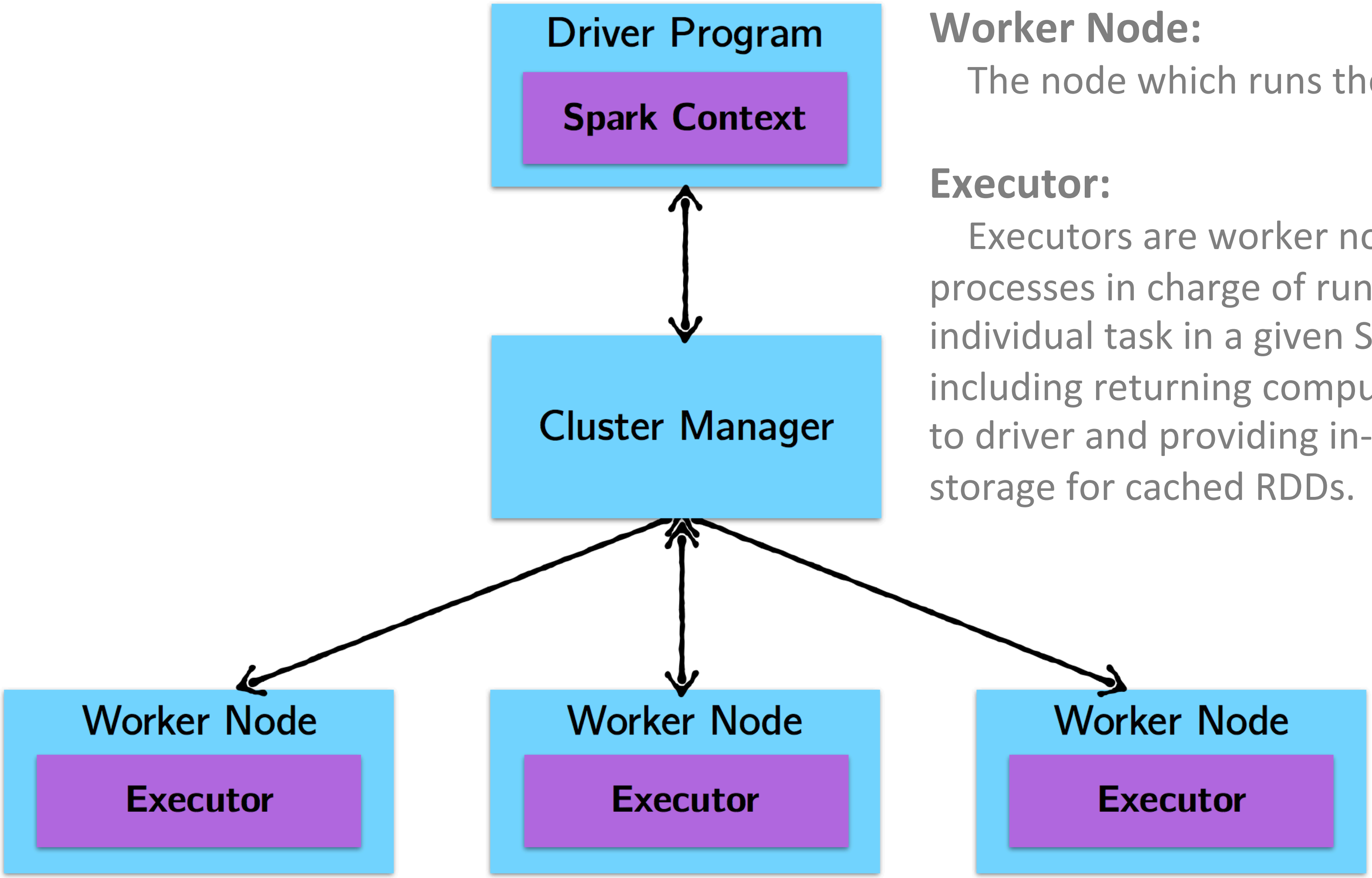
At a high level, every Spark application consists of a driver program that launches various parallel operations on the cluster.

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

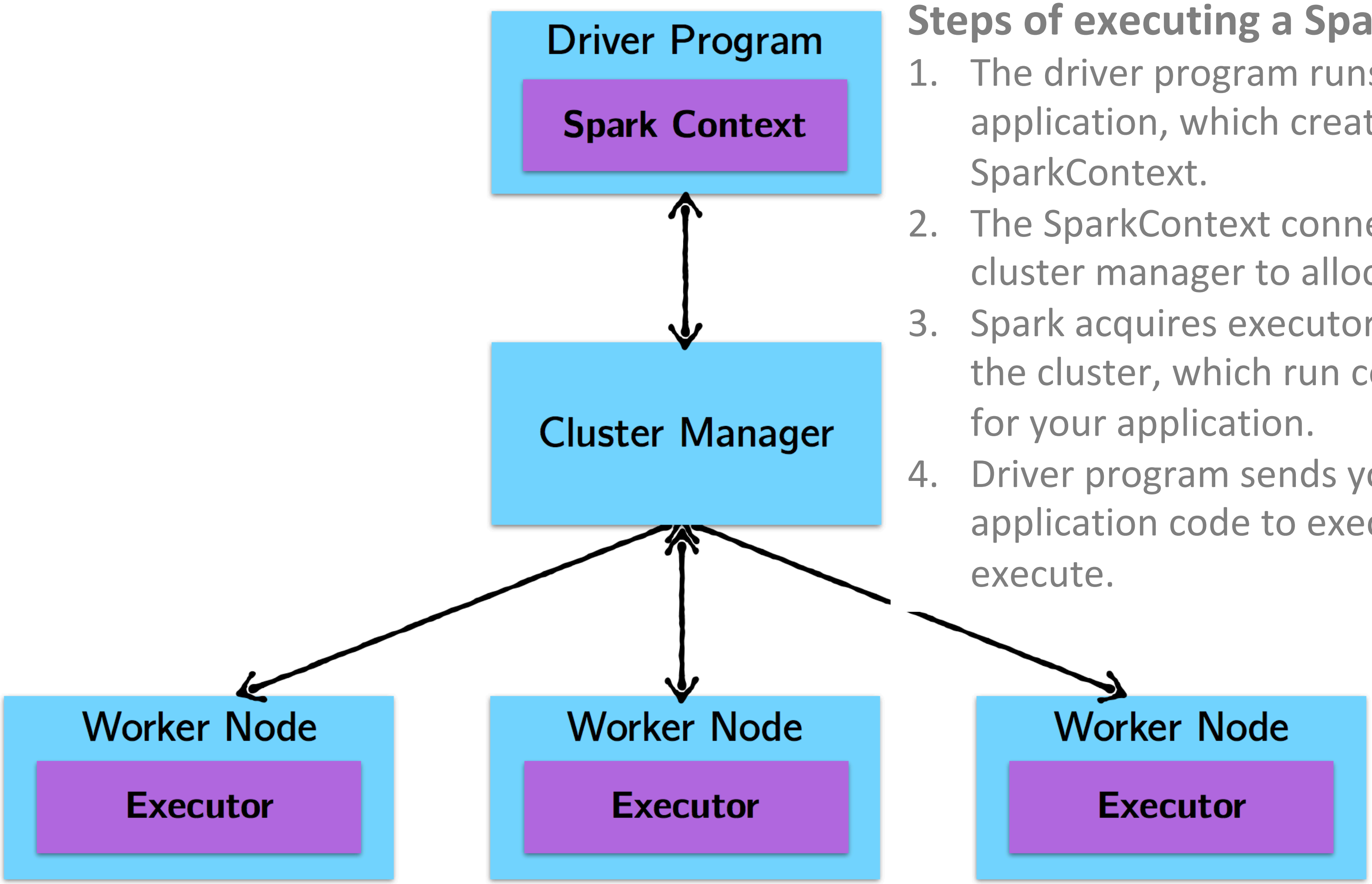
How Spark jobs are Executed



Worker Node:
The node which runs the application

Executor:
Executors are worker nodes' processes in charge of running individual task in a 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, which creates a 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 your 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*

```scala
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.

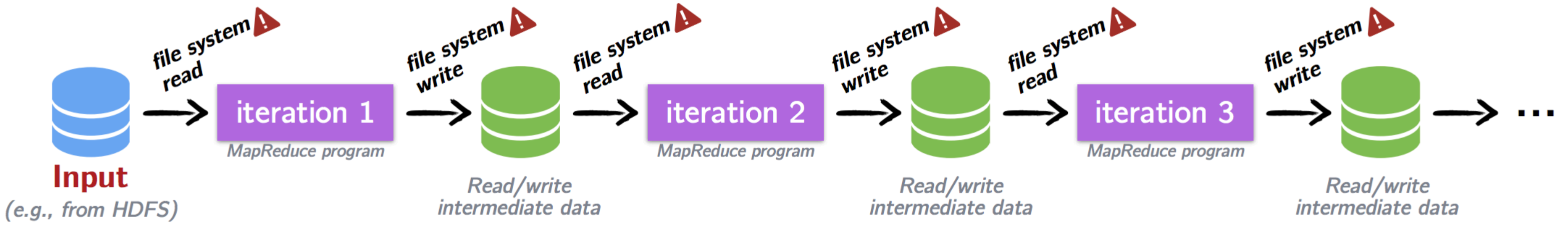
Why Spark is Good for Data Sci

- In-memory computation
- RDD operations
  - Transformations: Lazy, deferred
  - Actions: Eager, kick off staged transformations
- Why Spark is good for data science?
  - Machine learning algorithms

Iteration

- Most data science problems **involve iterations**

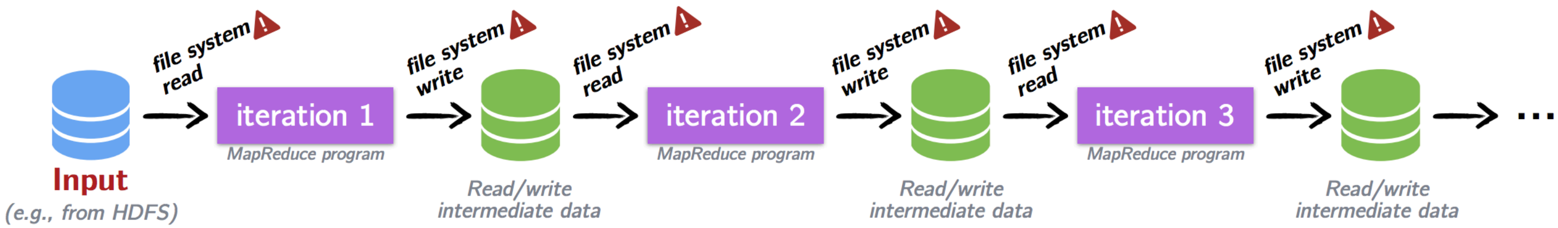
Iteration in Hadoop:



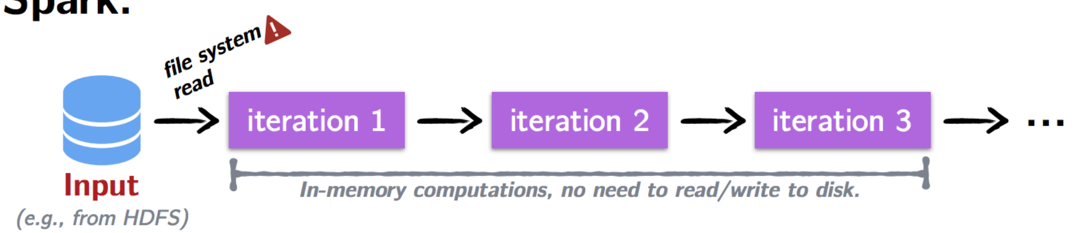
Iteration

- Most data science problems **involve iteration**

Iteration in Hadoop:

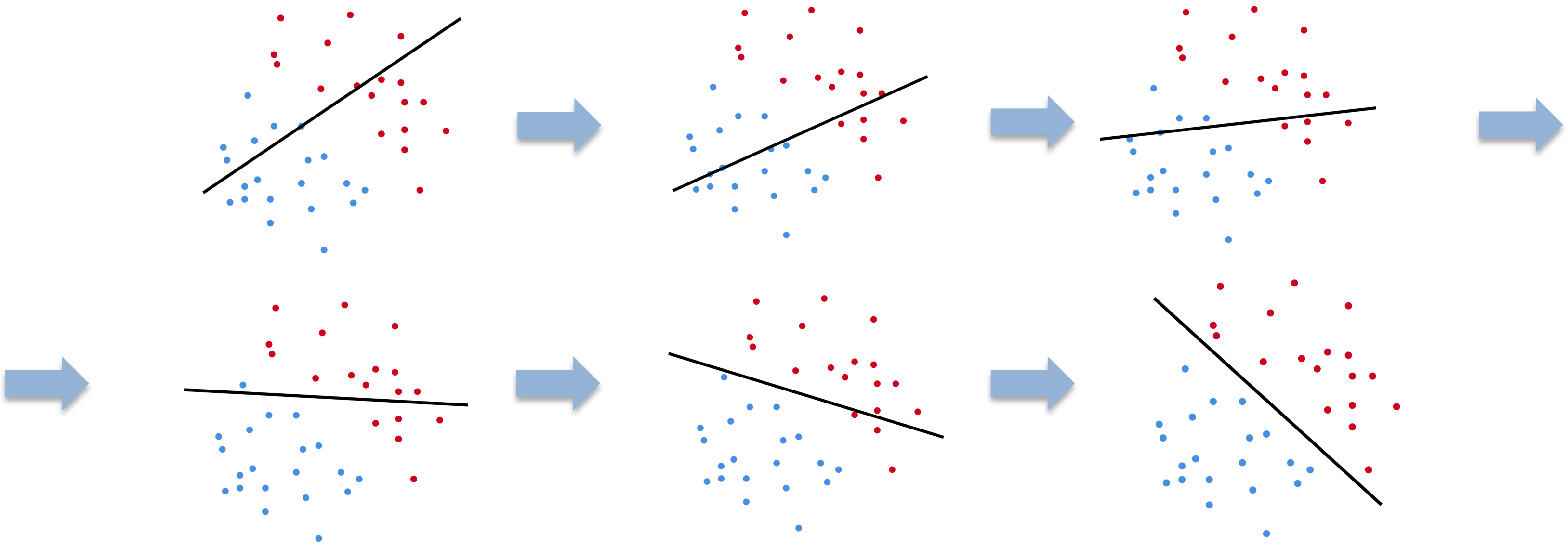


Iteration in Spark:



Iteration
Example: Logistic Regression

- Logistic regression is **an iterative algorithm** typically used for classification. Like most classification algorithms, it **updates weights iteratively** base 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** base 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.

Why Spark is Good for Data Sci

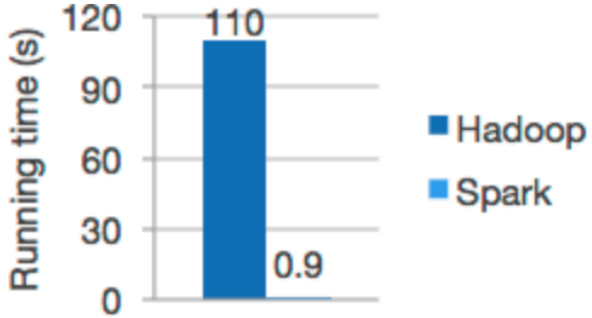
- **The lazy semantics** of RDD transformation operations help improve the performance.

- One of the most common performance bottlenecks for newcomers to Spark arises from unknowingly re-evaluating several transformations when caching could be used.

Spark vs. Hadoop

- **Spark is Faster**
  - When the output of an operation needs to be fed into another operation, Spark passes the data directly without writing to persistent storage
  - Better for some iterative algorithms

  e.g. machine learning algorithms



Logistic regression in Hadoop and Spark

Spark runs programs up to 100x faster than Hadoop MapReduce in memory. [1]

[1] https://spark.apache.org/

Other advantages

- **Easy to use**
  - Write applications quickly in Java, Scala, Python
- **Runs Everywhere**
  - Spark runs on Hadoop, standalone, or in the cloud
  - It can access diverse data sources including HDFS, Cassandra, HBase, and S3
- **Generality**
  - Combine SQL, streaming, and complex analytics

[1] https://spark.apache.org/

Word Count

word_count.py × text.txt ×

```
from pyspark import SparkContext
import os

[redacted]

sc = SparkContext('local[*]', 'wordCount')

input_file_path = './text.txt'
textRDD = sc.textFile(input_file_path)

counts = textRDD.flatMap(lambda line: line.split(' ')) \
    .map(lambda word: (word, 1)).reduceByKey(lambda a, b: a+b).collect()

for each_word in counts:
    print(each_word)
```

If you want to learn more…

- Official documentation
  - http://spark.apache.org/docs/latest/

- Online course
  - Coursera: Big Data Analysis with Scala and Spark

- Books
  - *Learning Spark, O' Reilly*
  - Advanced Analytics with Spark: Patterns for Learning from Data at Scale, *O' Reilly*
  - *Machine Learning with Spark, Packt*