ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΜΣ ΚΥΒΕΡΝΟΑΣΦΑΛΕΙΑ
ΚΑΙ ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ

MSc CYBERSECURITY
AND DATA SCIENCE

DEPT OF INFORMATICS
UNIVERSITY OF PIRAEUS

Διαχείριση Μεγάλων Δεδομένων

Big Data Management

Εργαστηριακή Διάλεξη Apache Spark

Γιώργος Αλεξίου
galexiou@athenarc.gr

ATHENA Research Center

Ver. 12/2023

Big Data and Spark

- Data is increasing in volume, velocity, variety.
- The need have faster results from analytics becomes increasingly important.
- Apache Spark is a computing platform designed to be fast and general-purpose, and easy to use

➢ **Speed**
  - In-memory computations
  - Faster than MapReduce for complex applications on disk

➢ **Generality**
  - Covers a wide range of workloads on one system
  - Batch applications (e.g. MapReduce)
  - Iterative algorithms
  - Interactive queries and streaming

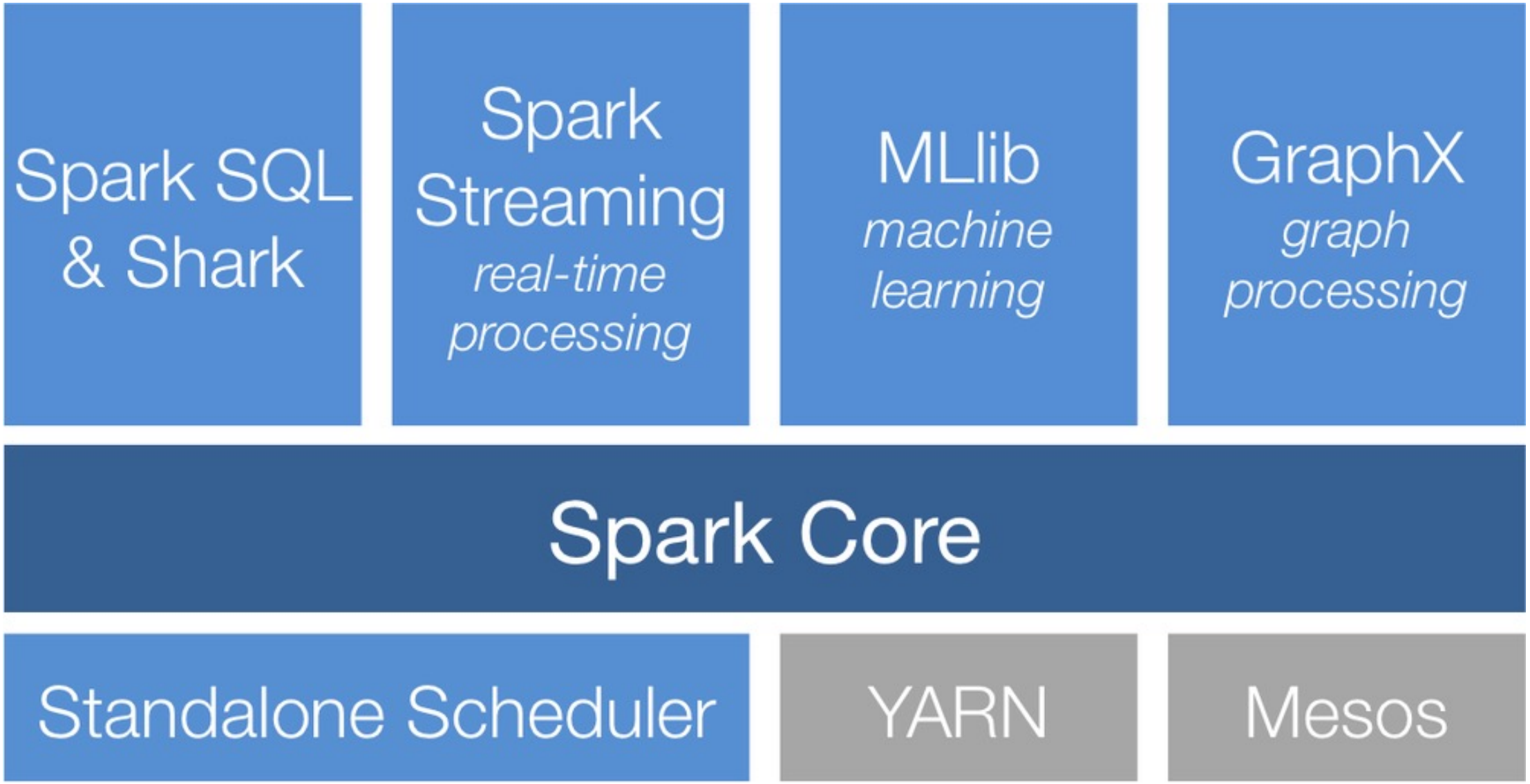
➢ **Ease of use**
  - APIs for Java, Scala and Python
  - Libraries for SQL, Machine Learning, Streaming and Graph processing
  - Runs on Hadoop clusters or as a standalone

Who uses Spark and Why?

- Parallel distributed processing, fault tolerance on commodity hardware, scalability, in-memory processing, high level APIs, etc
- Saves time and money!

- **Data scientists**
  - Analyze and model the data to obtain insight using ad-hoc analysis
  - Transforming the data into re-usable format
  - Statistics, machine learning, SQL
- **Engineers**
  - Develop a data processing system or application
  - Inspect and tune their applications
  - Programming with the Spark API
- **Everyone else**
  - Ease of use
  - Wide variety of functionality
  - Mature and reliable

Spark Unified Stack



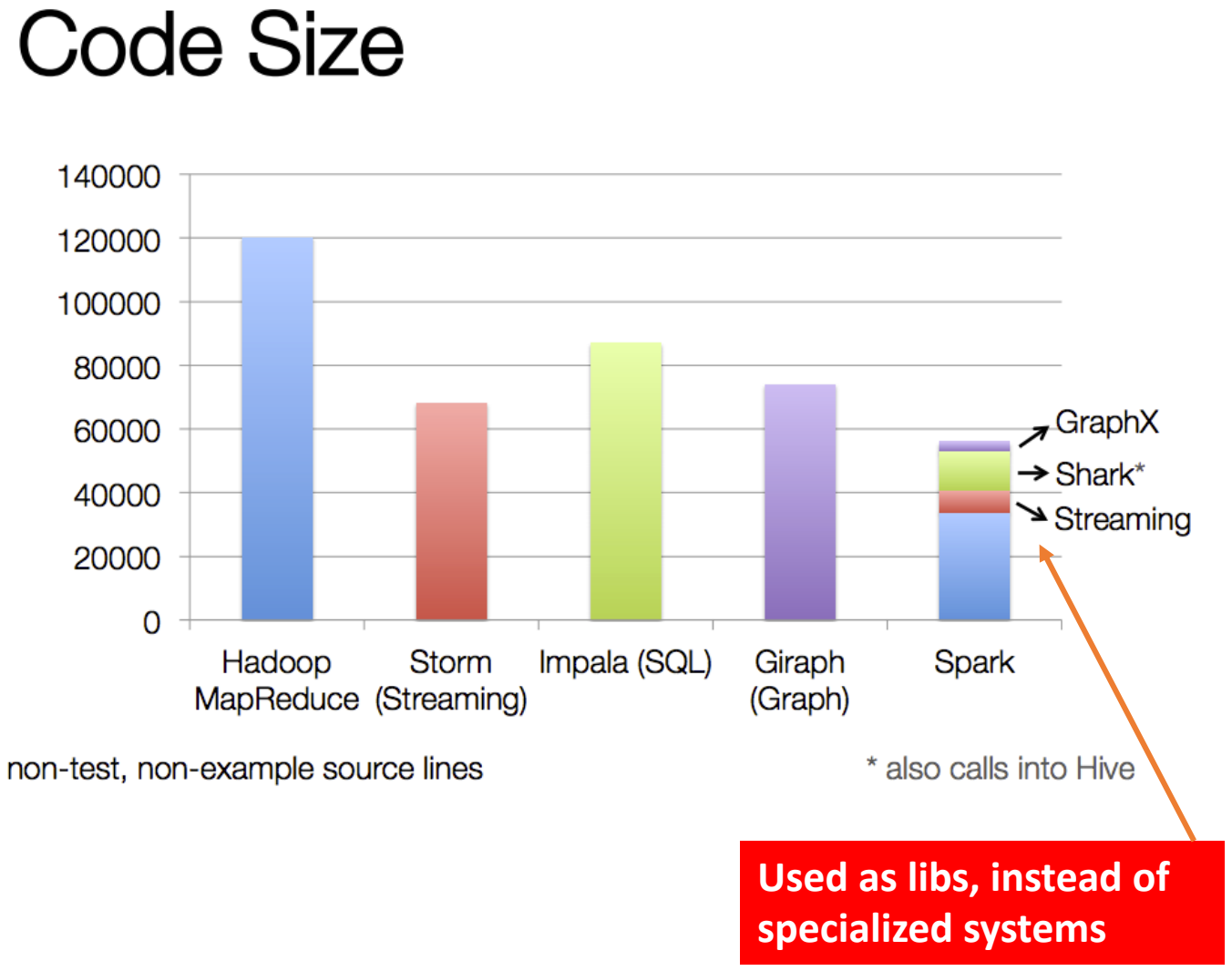
Brief History of Spark

- 2002 - MapReduce @ Google
- 2004 - MapReduce paper
- 2006 - Hadoop @ Yahoo!
- 2008 - Hadoop Summit
- 2010 - Spark paper
- 2014 Apache Spark top-level

- MapReduce stated a general batch processing paradigm

- Two limitations
  1. Difficulty programming in MapReduce
  2. Batch processing did not fit many use cases

- Spawned a lot of specialized systems (Storm, Impala, Giraph, etc.)



Resilient Distributed Datasets (RDD)

- Fault-tolerant collection of elements that can be operated on in parallel.
- Immutable

➢ Three methods for creating RDD:
  - Parallelizing an existing collection
  - Referencing a dataset
  - Transformation from an existing RDD

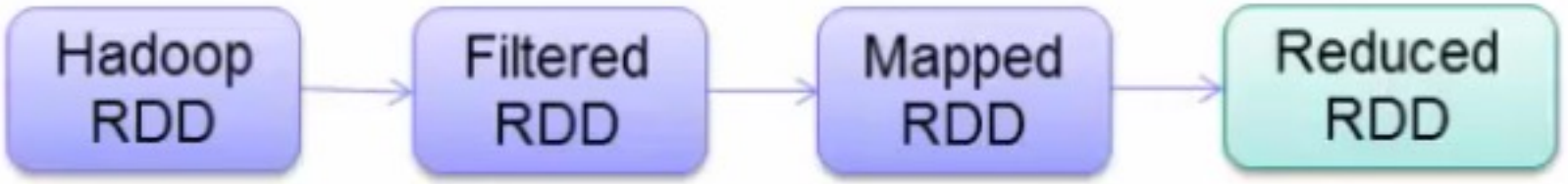
➢ Dataset from any storage supported by Hadoop
  - HDFS
  - Hive
  - HBase
  - Amazon S3
  - etc.

➢ Types of files supported
  - Text files
  - SequenceFiles
  - Hadoop InputFormat

Resilient Distributed Datasets (RDD) (cont'd)

- Two types of RDD operations:
  - **Transformations**
    - Creates a DAG
    - Lazy evaluations
    - No return value
  - **Actions**
    - Performs the transformations and the action that follows
    - Returns a value

- Fault tolerance
- Caching
- Example of RDD flow:



Downloading and Installing Spark

- Runs on both Windows and Unix-like systems (e.g Linux, OSX)

- To run locally on one machine, all you need is to have Java Installed on your system PATH or the JAVA_HOME pointing to a valid Java installation
- Visit this page to download: http://spark.apache.org/downloads.html
  - Select the Hadoop distribution you require under the “Pre-built packages”
  - Place a compiled version of spark on each node on the cluster

- Manually start the cluster by executing:
  - ./sbin/start-master.sh

- Once started, the master will print out a spark://HOST:PORT URL for itself, which you can use to connect workers to it.
  - The default master’s web UI is : http://localhost:8080

- Check out Spark’s website for more information
  - http://spark.apache.org/docs/latest/spark-standalone.html

For the purpose of this lecture

- We are going to use:
  - A docker image of Spark in standalone mode
    - Spark version : 2.4.4

- To get the image:
  - docker pull galexiou/sparklab:latest
  - docker run -d --name sparklab -e SPARK_MODE=master galexiou/sparklab

- Access to shell:
  - docker exec -it sparklab /bin/bash
  - **Then :** cd /
    - spark-shell

Spark Jobs and Shell

- Spark jobs can be written in Scala, Java, or Python
- Spark shells for Scala and Python
- APIs are available for all three
- Must adhere to the appropriate version for each Spark release
- Spark's native language is Scala, so it's natural to write Spark apps using Scala
- This lecture will cover code examples from Scala.

```
b-analytics@master:/home/user$ spark-shell  --deploy-mode client --master yarn
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_67)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc (master = yarn-client, app id = application_1521025523461_0024).
SQL context available as sqlContext.

scala>
```

Brief Overview of Scala

- **Everything is an Object**
  - Primitive types such as numbers or Boolean
  - Functions
- **Numbers are objects**
  - 1+2*3/4 -> (1).+(((2).*(3))./(x)))
  - Where the +, *, / are valid identifiers in Scala
- **Functions are objects**
  - Pass functions as arguments
  - Store them in variables
  - Return them from other functions
- **Function declaration**
  - def functionName ([list of arguments]) : [return type]

Scala - Anonymous Functions

- Functions without a name created for one-time use to pass to another function
- Left side of the right arrow => is where the argument resides (no arguments in the example)
- Right side of the arrow is the body of the function (the println statement)

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies() {
    println("time flies like an arrow...")
  }
  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
  }
}
```

```
object TimerAnonymous {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("time flies like an arrow..."))
  }
}
```

Spark's Scala and Python Shell

- Spark's shell provides a simple way to learn the API
- Powerful tool to analyze data interactively
- The Scala shell runs on the Java VM
  - A good way to use existing Java libraries
- Scala:
  - To launch the Scala shell:
    ./bin/spark-shell
  - To read in a text file:
    scala> val textFile = sc.textFile("README.md")
- Python:
  - To launch the Scala shell:
    ./bin/pyspark
  - To read in a text file:
    >>> textFile = sc.textFile("README.md")

Creating an RDD

- Launch the spark shell:

  Cluster: spark-shell --deploy-mode client --master yarn

  example: spark-shell

- Create some data:

  val data = 1 to 10000

- Parallelize that data (creating the RDD):

  val distData = sc.parallelize(data)

- Perform tranformations or invoke an action on it

  distData.filter(_ %2==1).collect()

  distData.filter(x => x %2==1).collect()

- Alternatively, create an RDD from an external dataset

  val readme = sc.textFile("/labdata/README.md") // local path

RDD Operations - Basics

- Loading a file:

```
val lines = sc.textFile("/labdata/README.md")
```

- Applying transformations:

```
val lineLengths = lines.map( s => s.length)
```

- Invoking actions:

```
val  totalLengths = lineLengths.reduce( (a,b ) => a + b)
```

- MapReduce Example:

```
val wordCounts = lines.flatMap( line => line.split (" "))
.map( word => (word , 1))
.reduceByKey(( a,b) => a + b)

wordCounts.collect()
```

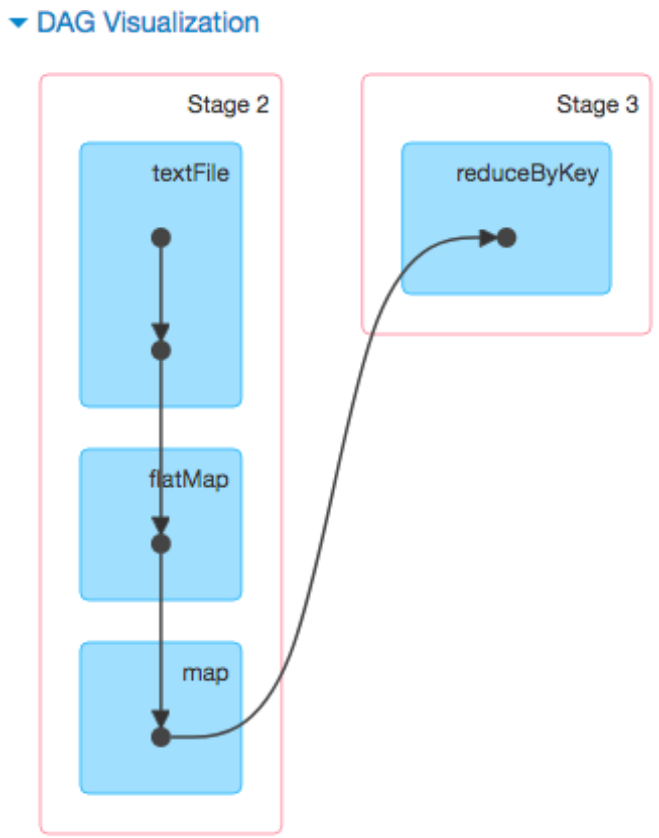
Direct Acyclic Graph (DAG)

- View the DAG:

  wordCounts.toDebugString

- Sample DAG:

```
scala> wordCounts.toDebugString
res8: String =
(2) ShuffledRDD[7] at reduceByKey at <console>:29 []
 +-(2) MapPartitionsRDD[6] at map at <console>:29 []
    |  MapPartitionsRDD[5] at flatMap at <console>:29 []
    |  README.md MapPartitionsRDD[1] at textFile at <console>:27 []
    |  README.md HadoopRDD[0] at textFile at <console>:27 []
```



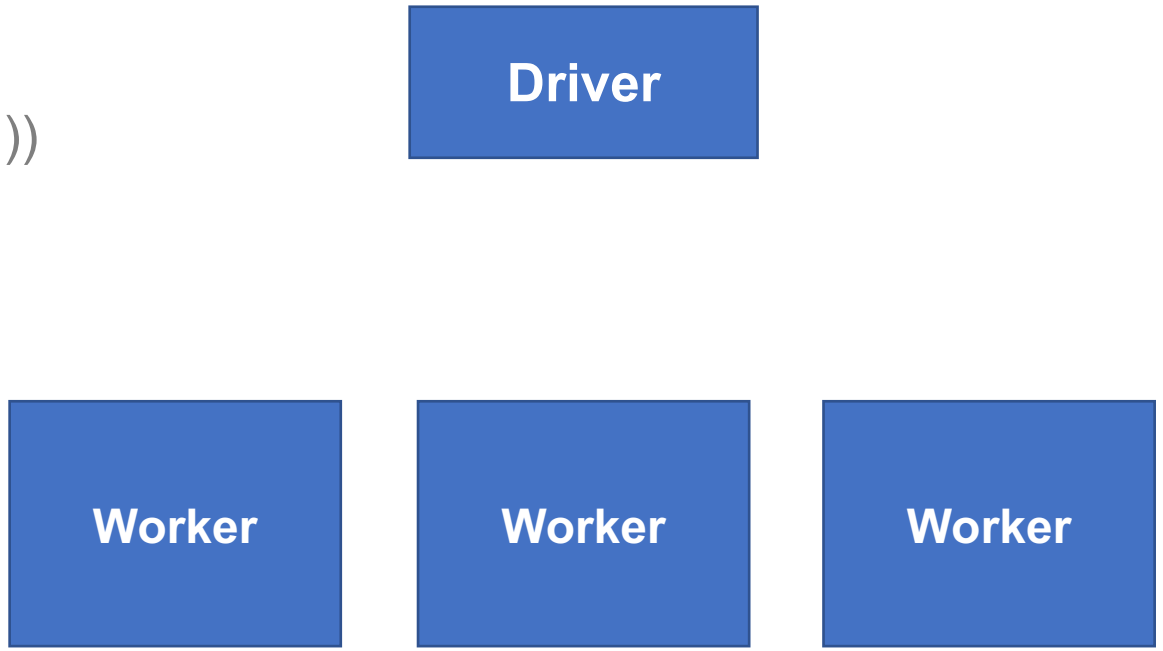
Direct Acyclic Graph (DAG)

- Creating the RDD:
  val logFile= sc.textFile("/labdata/log.txt")

- Tranformations:
  val errors= logFile.filter(_.startsWith("ERROR"))
  val messages = errors.map(_.split("\t")).map(r=>r(1))

- Caching:
  messages.cache()

- Actions:
  messages.filter(_.contains("mysql")).count()
  messages.filter(_.contains("php")).count()



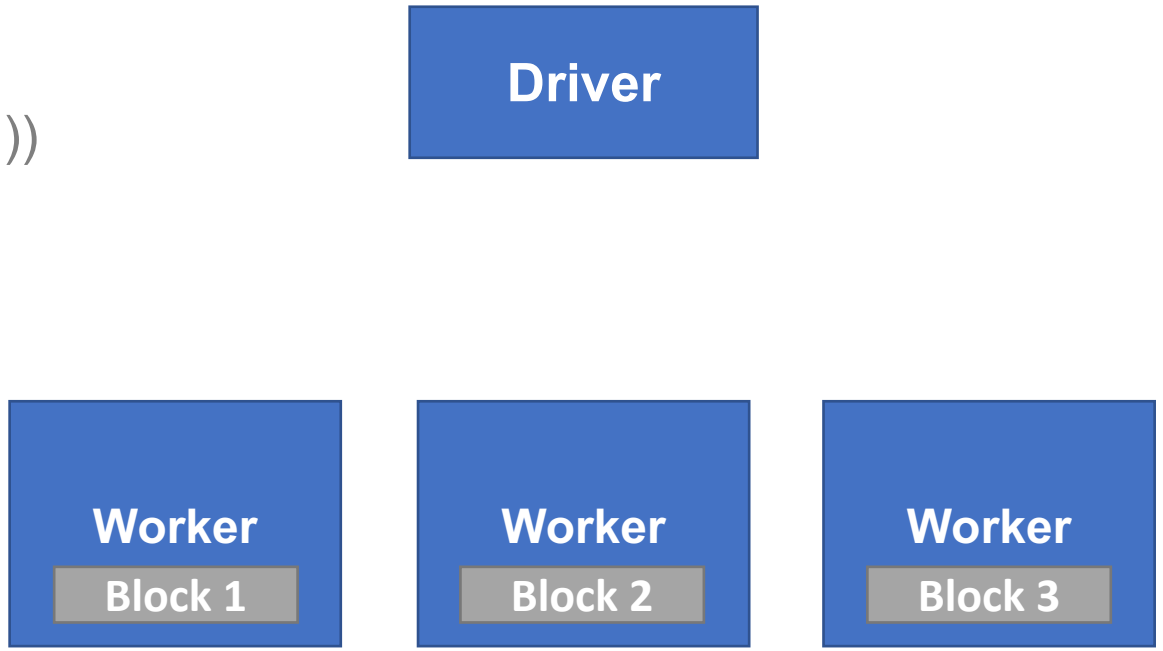
Direct Acyclic Graph (DAG)

- Creating the RDD:
  val logFile= sc.textFile("/labdata/log.txt")

- Tranformations:
  val errors= logFile.filter(_.startsWith("ERROR"))
  val messages = errors.map(_.split("\t")).map(r=>r(1))

- Caching:
  messages.cache()

- Actions:
  messages.filter(_.contains("mysql")).count()
  messages.filter(_.contains("php")).count()



The data is partitioned into different blocks

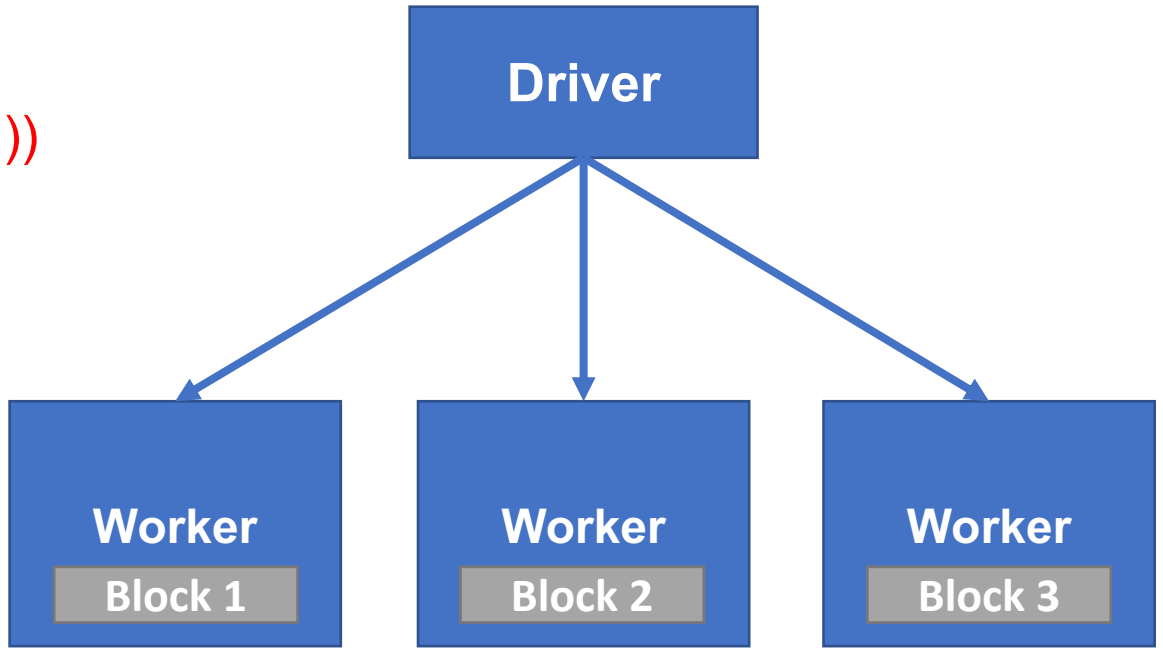
Direct Acyclic Graph (DAG)

- Creating the RDD:
  val logFile= sc.textFile("/labdata/log.txt")

- Tranformations:
  val errors= logFile.filter(_.startsWith("ERROR"))
  val messages = errors.map(_.split("\t")).map(r=>r(1))

- Caching:
  messages.cache()

- Actions:
  messages.filter(_.contains("mysql")).count()
  messages.filter(_.contains("php")).count()



Driver sends the code to be executed on each block

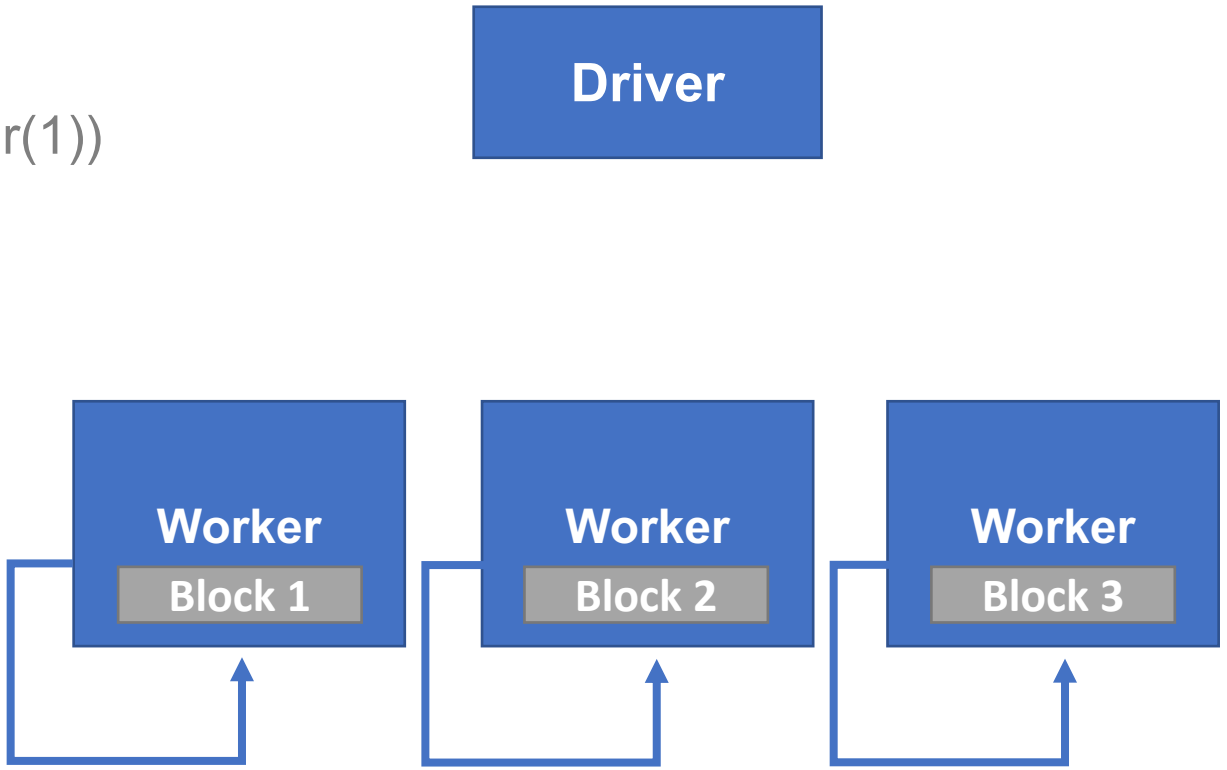
Direct Acyclic Graph (DAG)

- Creating the RDD:
  val logFile= sc.textFile("/labdata/log.txt")

- Tranformations:
  val errors= logFile.filter(_.startsWith("ERROR"))
  val messages = errors.map(_.split("\t")).map(r=>r(1))

- Caching:
  messages.cache()

- Actions:
  messages.filter(_.contains("mysql")).count()
  messages.filter(_.contains("php")).count()



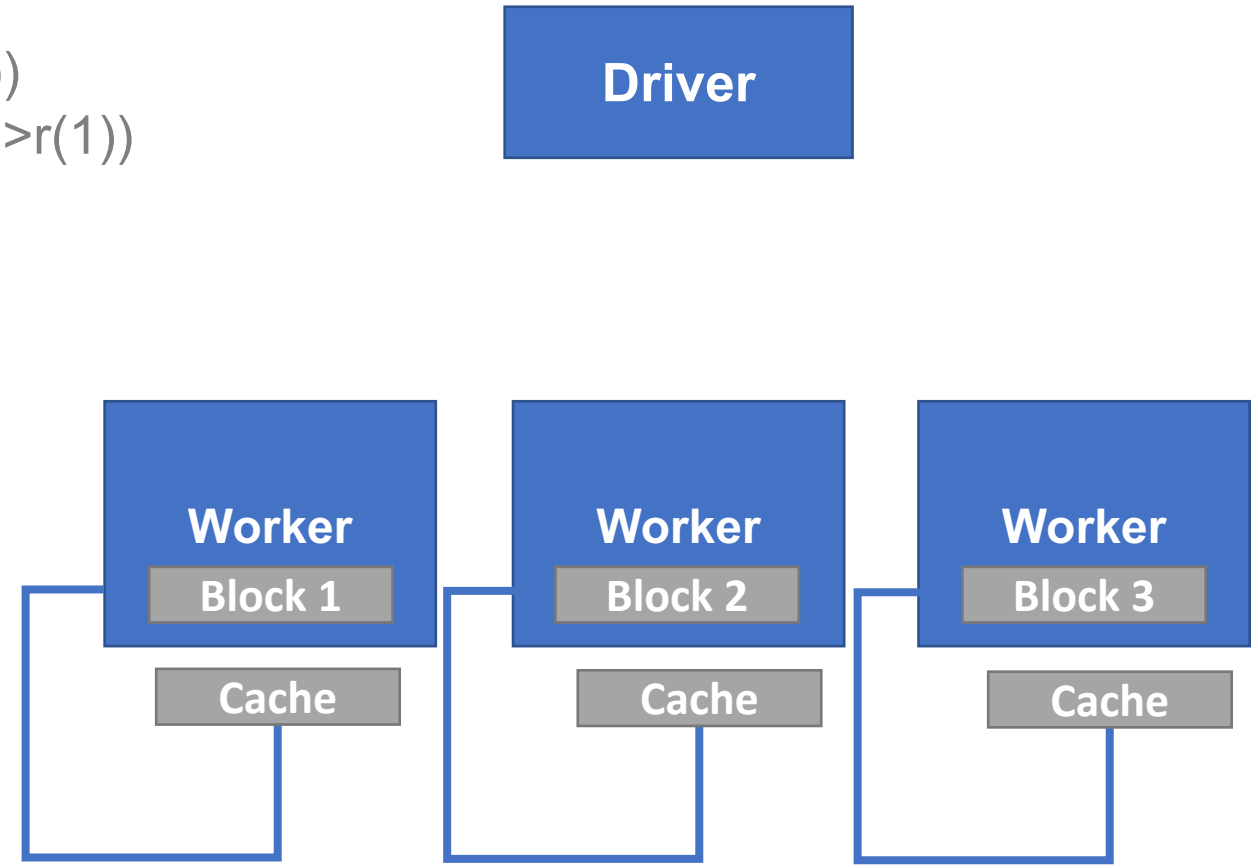
Direct Acyclic Graph (DAG)

- Creating the RDD:
  val logFile= sc.textFile("/labdata/log.txt")

- Tranformations:
  val errors= logFile.filter(_.startsWith("ERROR"))
  val messages = errors.map(_.split("\t")).map(r=>r(1))

- Caching:
  messages.cache()

- Actions:
  messages.filter(_.contains("mysql")).count()
  messages.filter(_.contains("php")).count()



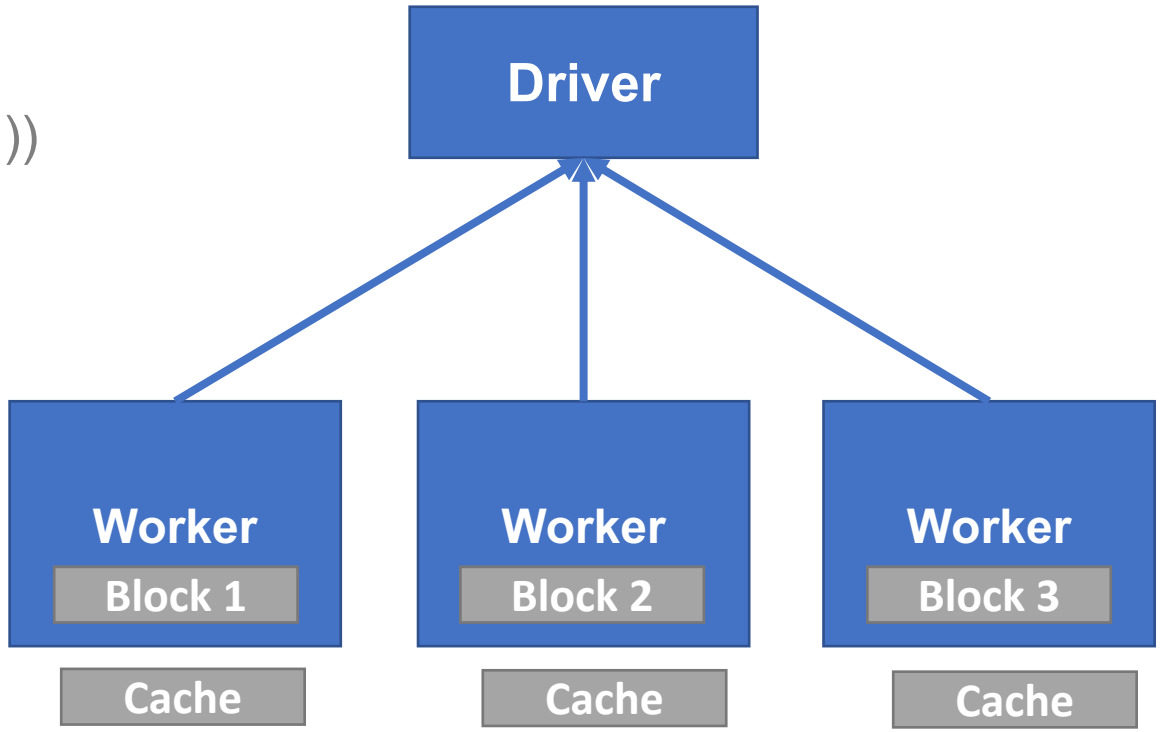
Direct Acyclic Graph (DAG)

- Creating the RDD:
  val logFile= sc.textFile("/labdata/log.txt")

- Tranformations:
  val errors= logFile.filter(_.startsWith("ERROR"))
  val messages = errors.map(_.split("\t")).map(r=>r(1))

- Caching:
  messages.cache()

- Actions:
  messages.filter(_.contains("mysql")).count()
  messages.filter(_.contains("php")).count()



Send the data back to driver

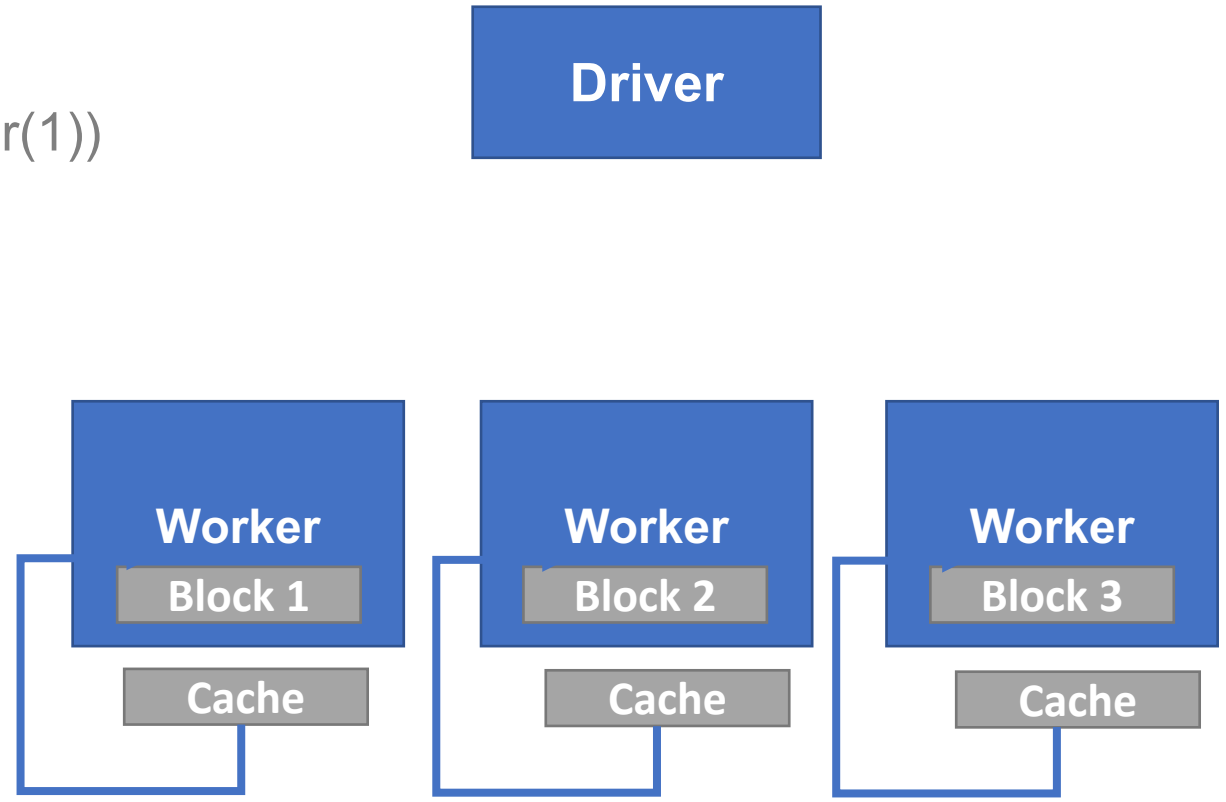
Direct Acyclic Graph (DAG)

- Creating the RDD:
  val logFile= sc.textFile("/labdata/log.txt")

- Tranformations:
  val errors= logFile.filter(_.startsWith("ERROR"))
  val messages = errors.map(_.split("\t")).map(r=>r(1))

- Caching:
  messages.cache()

- Actions:
  messages.filter(_.contains("mysql")).count()
  messages.filter(_.contains("php")).count()



Process from cache

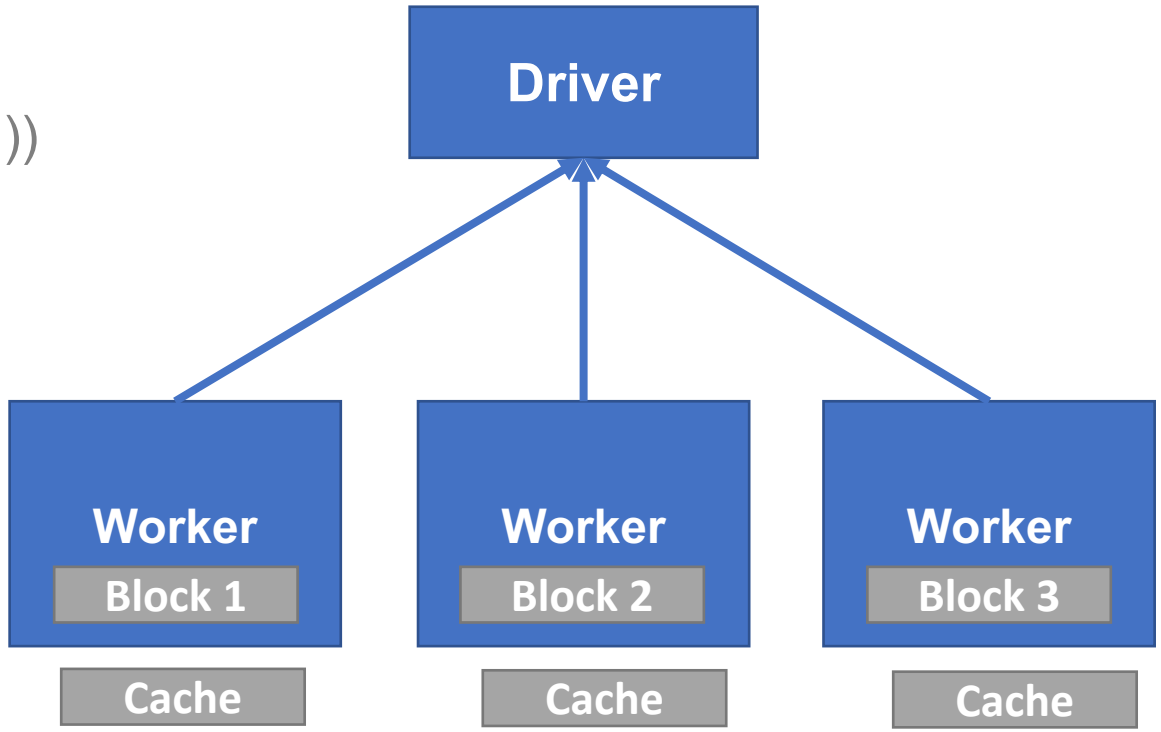
Direct Acyclic Graph (DAG)

- Creating the RDD:
  val logFile= sc.textFile("/labdata/log.txt")

- Tranformations:
  val errors= logFile.filter(_.startsWith("ERROR"))
  val messages = errors.map(_.split("\t")).map(r=>r(1))

- Caching:
  messages.cache()

- Actions:
  messages.filter(_.contains("mysql")).count()
  messages.filter(_.contains("php")).count()



Send the data back to driver

RDD Operations - Transformations

- A subset of the transformations available. Full set can be found on Spark's website
- Transformations are lazy evaluations
- Returns a pointer to the transformed RDD

Transformation	Meaning
map(func)	Return a new distributed dataset formed by passing each element of the source through a function func.
filter(func)	Return a new dataset formed by selecting those elements of the source on which func returns true.
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).
join(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.
reduceByKey(func, [numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V.

RDD Operations - Actions

- Actions return values

Action	Meaning
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(n)	Return an array with the first n elements of the dataset.
foreach(func)	Run a function func on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems.
saveAsTextFile(path)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS

RDD Persistence

- Each node stores any partitions of the cache that it computes in-memory
- Reuses them in other actions on that dataset (or datasets derived from it)
  - Future actions are much faster (often by more than 10x)
- Two methods for RDD persistence
  - persist()
  - cache() -> essentially just persist with *MEMORY_ONLY* storage

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.

RDD Persistence (cont'd)

Storage Level	Meaning
MEMORY_ONLY_SER (Java and Scala)	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

Which Storage Level to Choose?

- If your RDDs fit comfortably with the default storage level *(MEMORY_ONLY)*, leave them that way. This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible.
- If not, try using *MEMORY_ONLY_SER* and selecting a fast serialization library to make the objects much more space-efficient, but still reasonably fast to access.
- Don't spill to disk unless the functions that computed your datasets are expensive, or they filter a large amount of the data. Otherwise, recomputing a partition may be as fast as reading it from disk.
- Use the replicated storage levels if you want fast fault recovery (e.g. if using Spark to serve requests from a web application). All the storage levels provide full fault tolerance by recomputing lost data, but the replicated ones let you continue running tasks on the RDD without waiting to recompute a lost partition.
- In environments with high amounts of memory or multiple applications the experimental *OFF_HEAP* mode has several advantages:
  - It allows multiple executors to share the same pool of memory
  - It significantly reduces garbage collection costs
  - Cached data is not lost if individual executors crash

Shared Variables and Key-Value Pairs

- When a function is passed from the driver to a worker, normally a separate copy of the variables are used.
- Two types of variables:
  - **Broadcast variables**
    - Read-only copy on each machine
    - Distribute broadcast variables using efficient broadcast algorithms
  - **Accumulators**
    - Variables added through an associative operation
    - Implement counters and sums
    - Only the driver can read the accumulators value
    - Numeric types accumulators. Extend for new types

Scala: key-value pairs

```
val pair = ('a', 'b')
pair._1 // will return 'a'
pair._2 // will return 'b'
```

Python: key-value pairs

```
pair = ('a', 'b')
pair[0] # will return 'a'
pair[1] # will return 'b'
```

Java: key-value pairs

```
Tuple2 pair = new Tuple2('a', 'b');
pair._1 // will return 'a'
pair._2 // will return 'b'
```

Programming the Key-Value Pairs

- There are special operations available on RDDs of key-value pairs
  - Grouping or aggregating elements by a key

- Tuple2 objects created by writing (a , b)
  - Must import *org.apache.spark.SparkContext._*

- PairRDDFunctions contains key-value pair operations
  - *reduceByKey((a, b) => a + b)*

- Custom objects as key in key-value pair requires a custom *equals()* method with a matching *hashCode()* method.

- Example:

```
val textFile = sc.textFile(“ …. “ )
val readmeCount = textFile.flatMap(line => line.split(“ “)).map( word => (word, 1))
.reduceByKey( _+_ )
```

Scala - Working with RDD operations

- Create an RDD file from the file README (already in: “/labdata/README.md”). This is created using the spark context *".textFile”*.

- As we know the initial operation is a transformation, so nothing actually happens. We're just telling it that we want to create a readme RDD.

- This was an RDD transformation, thus it returned a pointer to a RDD, which we have named as readme.

  val readme = sc.textFile("/labdata/README.md")

- Let’s perform some RDD actions on this text file. Count the number of items in the RDD using this command:

  readme.count()

- Let’s run another action. Run this command to find the first item in the RDD:

  readme.first()

- Now let’s try a transformation. Use the filter transformation to return a new RDD with a subset of the items in the file. Type in this command:

  val linesWithSpark = readme.filter(line => line.contains("Spark"))
  linesWithSpark.count()

Scala - Working with RDD operations (cont'd)

- You can even chain together transformations and actions. To find out how many lines contains the word "Spark", type in:.

  readme.filter(line => line.contains("Spark")).count()

- You will see that RDD can be used for more complex computations. You will find the line from that readme file with the most words in it.

  readme.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)

- There are two parts to this. The first maps a line to an integer value, the number of words in that line. In the second part reduce is called to find the line with the most words in it. The arguments to map and reduce are Scala function literals (closures). **You can use any language feature or Scala/Java library.**

- Use the Math.max() function to show that you can indeed use a Java library instead. Import in the java.lang.Math library:

  import java.lang.Math

- Now run with the max function:

  readme.map(line => line.split(" ").size).reduce((a, b) => Math.max(a, b))

Scala - Working with RDD operations (cont'd)

- Spark has a MapReduce data flow pattern. We can use this to do a word count on the readme file.

```
val wordCounts = readme.flatMap(line => line.split(" "))
.map(word => (word, 1))
.reduceByKey((a,b) => a + b)
```

- To collect the word counts, use the collect action.

```
wordCounts.collect().foreach(println)
```

- It should be noted that the collect function **brings all of the data into the driver node**. For a small dataset, this is acceptable **but, for a large dataset this can cause an Out Of Memory error.** It is recommended to use collect() for **testing only**. The safer approach is to use the take() function e.g. take(n).foreach(println)

- You can also do:

```
println(wordCounts.collect().mkString("\n"))
```

Scala - Working with RDD operations (cont'd)

- **Analysing a log file**

  val logFile = sc.textFile("/labdata/notebook.log")

- Filter out the lines that contains INFO (or ERROR, if the particular log has it)

  val info = logFile.filter(line => line.contains("INFO"))

- Count the lines:

  info.count()

- Count the lines with Spark in it by combining transformation and action.

  info.filter(line => line.contains("spark")).count()

- Fetch those lines as an array of Strings

  info.filter(line => line.contains("spark")).collect() foreach println

- Remember that we went over the DAG. It is what provides the fault tolerance in Spark. Nodes can re-compute its state by borrowing the DAG from a neighboring node. You can view the graph of an RDD using the *toDebugString* command.

  println(info.toDebugString)

Scala - Working with RDD operations (cont'd)

- **Joining RDDs.**
- Next, you are going to create RDDs for the README and the POM file in the current directory.

```
val readmeFile = sc.textFile("/labdata/README.md")
val pom = sc.textFile("/labdata/pom.xml")
```

- How many Spark keywords are in each file?

```
println(readmeFile.filter(line => line.contains("Spark")).count())
println(pom.filter(line => line.contains("spark")).count())
```

- Now do a WordCount on each RDD so that the results are (K,V) pairs of (word,count)

```
val readmeCount = readmeFile.
        flatMap(line => line.split(" ")).
        map(word => (word, 1)).
        reduceByKey(_ + _)  // .collect() foreach println

val pomCount = pom.
        flatMap(line => line.split(" ")).
        map(word => (word, 1)).
        reduceByKey(_ + _)  // .collect() foreach println
```

Scala - Working with RDD operations (cont'd)

- Now let's join these two RDDs together to get a collective set. The join function combines the two datasets (K,V) and (K,W) together and get (K, (V,W)). Let's join these two counts together and then cache it.

```
val joined = readmeCount.join(pomCount)
joined.cache()
```

- Let's see what's in the joined RDD.

```
joined.collect.foreach(println)
```

- Let's combine the values together to get the total count. The operations in this command tells Spark to combine the values from (K,V) and (K,W) to give us(K, V+W). The ._ notation is a way to access the value on that particular index of the key value pair.

```
val joinedSum = joined.map(k => (k._1, (k._2)._1 + (k._2)._2))
joinedSum.collect() foreach println
```

- To check if it is correct, print the first five elements from the joined and the joinedSum RDD

```
println("Joined Individial\n")
joined.take(5).foreach(println)

println("\n\nJoined Sum\n")
joinedSum.take(5).foreach(println)
```

Scala - Working with RDD operations (cont'd)

- **Shared variables**

- **Broadcast variables** allow the programmer to keep a read-only variable cached on each worker node rather than shipping a copy of it with tasks.

- They can be used, to give every node a copy of a large input dataset in an efficient manner.

- After the broadcast variable is created, it should be used instead of the value v in any functions run on the cluster so that v is not shipped to the nodes more than once.

- The object v should not be modified after it is broadcast in order to ensure that all nodes get the same value of the broadcast variable (e.g. if the variable is shipped to a new node later).

- Let's create a broadcast variable:

  val broadcastVar = sc.broadcast(Array(1,2,3))

- To get the value, type in:

  broadcastVar.value

Scala - Working with RDD operations (cont'd)

- **Shared variables**

- **Accumulators** are variables that can only be added through an associative operation.

- It is used to implement counters and sum efficiently in parallel.

- Spark natively supports numeric type accumulators and standard mutable collections.

- Programmers can extend these for new types.

- **Only the driver** can read the values of the accumulators. The workers can only invoke it to increment the value.

- Create the accumulator variable. Type in:

  val accum = sc.accumulator(0)

- Next parallelize an array of four integers and run it through a loop to add each integer value to the accumulator variable. Type in:

  sc.parallelize(Array(1,2,3,4)).foreach(x => accum += x)
  accum.value

Scala - Working with RDD operations (cont'd)

- **Key-Value Pairs**

- You have already seen a bit about key-value pairs in the Joining RDD section. Here is a brief example of how to create a key-value pair and access its values. Remember that certain operations such as map and reduce only works on key-value pairs.

- Create a key-value pair of two characters. Type in:

  val pair = ('a', 'b')

- To access the value of the first index using the ._1 method and ._2 method for the 2nd.

  pair._1
  pair._2

Scala - Example Application

- A subset of a data for taxi trips will be used, that will determine the top 10 medallion numbers based on the number of trips.

  val taxi = sc.textFile("/labdata/nyctaxi.txt")

- To view the five rows of content, invoke the take function. Type in:

  taxi.take(5).foreach(println)

- To parse out the values, including the medallion numbers, you need to first create a new RDD by splitting the lines of the RDD using the comma as the delimiter. Type in:

  val taxiParse = taxi.map(line=>line.split(","))

- Now create the key-value pairs where the key is the medallion number and the value is 1. We use this model to later sum up all the keys to find out the number of trips a particular taxi took and in particular, will be able to see which taxi took the most trips. Map each of the medallions to the value of one. Type in:

  val taxiMedKey = taxiParse.map(vals=>(vals(6), 1))

- **vals(6)** corresponds to the column where the medallion key is located

Scala - Example Application (cont'd)

- Next use the reduceByKey function to count the number of occurrence for each key.

```
val taxiMedCounts = taxiMedKey.reduceByKey((v1,v2)=>v1+v2)

taxiMedCounts.take(5).foreach(println)
```

- Finally, the values are swapped so they can be ordered in descending order and the results are presented correctly.

```
for (pair <-taxiMedCounts.map(_.swap).top(10)) println("Taxi Medallion %s had %s
Trips".format(pair._2, pair._1))
```

- While each step above was processed one line at a time, you can just as well process everything on one line:

```
val taxiMedCountsOneLine
= taxi.map(line=>line.split(',')).map(vals=>(vals(6),1)).reduceByKey(_ + _)
```

Scala - Example Application (cont'd)

- Let's cache the taxiMedCountsOneLine to see the difference caching makes. Run it with the logs set to INFO and you can see the output of the time it takes to execute each line.

- First, let's cache the RDD

  taxiMedCountsOneLine.cache()

- Next, you have to invoke an action for it to actually cache the RDD. Note the time it takes here

  taxiMedCountsOneLine.count()

- Run it again to see the difference.

  taxiMedCountsOneLine.count()

- The bigger the dataset, the more noticeable the difference will be. In a sample file such as ours, the difference may be negligible.

APPLICATION PROGRAMMING

Spark Context

- The main entry point for Spark Functionality
- Represents the connection to a Spark cluster
- Create RDDs, accumulators and broadcast variables on that cluster
- In the spark shell, the SparkConetxt, sc, is automatically initialized for you to use
- In a Spark program, import some classes and implicit conversions into your program:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
```

Linking with Spark - Scala

- Spark applications requires dependencies.
- Must have a compatible Scala version to write applications
  e.g Spark 1.6.0 uses Scala 2.10

- To write a Spark application, you need to add a Maven dependency on Spark.
  - Spark is available through Maven Central at.

    groupId = org.apache.spark
    artifactId = spark-core_2.10
    version = 1.6.0

- To access a HDFS Cluster, you need to add a dependency on Hadoop-client for your version of HDFS

    groupId = org.apache.hadoop
    artifactId = Hadoop-client
    version = <version>

Linking with Spark - Java

- Spark 1.6.0 works with Java 7 and higher.
  - Java 8 supports lambda expressions

- Add a dependency on Spark.
  - Available through Maven Central at.

    groupId = org.apache.spark
    artifactId = spark-core_2.10
    version = 1.6.0

- If you wish to access a HDFS Cluster, you must add a dependency as well:

    groupId = org.apache.hadoop
    artifactId = Hadoop-client
    version = <version>

- Import some spark classes

```
import org.apache.spark.api.java.JavaSparkContext
import org.apache.spark.api.java.JavaRDD
import org.apache.spark.SparkConf
```

Initializing Spark - Scala

- Build a SparkConf object that contains information about your application

  val conf = new SparkConf().setAppName("appname").setMaster(master)

- The appName parameter -> Name for your application to show on the cluster UI

- The master parameter -> is a Spark or YARN cluster URL (or a special "local" string to run in local mode)
  - In testing, you can pass "local" to run Spark
  - local[16] will allocate 16 cores
  - In production mode, do not hardcode master in the program. Launch with spark-submit and provide it there.

- Then, you will need to create the SparkContext object

  new SparkContext(conf)

Initializing Spark - Java

- Build a SparkConf object that contains information about your application

  SparkConf conf = new SparkConf().setAppName(appname).setMaster(master);

- The appName parameter -> Name for your application to show on the cluster UI

- The master parameter -> is a Spark or YARN cluster URL (or a special “local” string to run in local mode)
  - In testing, you can pass “local” to run Spark
  - local[16] will allocate 16 cores
  - In production mode, do not hardcode master in the program. Launch with spark-submit and provide it there.

- Then, you will need to create the SparkContext object

  JavaSparkContext sc = new JavaSparkContext(conf);

Passing functions to Spark

- Spark's API relies on heavily passing functions in the driver program to run on the cluster

- Three methods
  - **Anonymous function syntax**

    (x : Int) => x + 1
  - **Static methods in a global singleton object**

    object Myfunctions {

    def func1 (s : String) : String = { … }

    }

    myRdd.map(Myfunctions.func1)
  - **Passing by reference**
    - to avoid sending the entire object consider copying the function to a local variable
    - Example:

      val field = "hello"
    - *Avoid*

      def doStuff(rdd : RDD[String]) : RDD[String] = {rdd.map(x => field + x)}
    - Consider

      def doStuff(rdd : RDD[String]) : RDD[String] = {

      val field_ = this.field

      rdd.map(x=> field_ +x) }

Programming the Business Logic

- Spark's API available in Scala, Java or Python
- Create the RDD from an external dataset or from existing RDD
- Transformations and actions to process the data
- Use RDD persistence to improve performance
- Use broadcast variables or accumulators for specific use cases

```scala
package org.apache.spark.examples

import org.apache.spark._


object HdfsTest {

  /** Usage: HdfsTest [file] */
  def main(args: Array[String]) {
    if (args.length < 1) {
      System.err.println("Usage: HdfsTest <file>")
      System.exit(1)
    }
    val sparkConf = new SparkConf().setAppName("HdfsTest")
    val sc = new SparkContext(sparkConf)
    val file = sc.textFile(args(0))
    val mapped = file.map(s => s.length).cache()
    for (iter <- 1 to 10) {
      val start = System.currentTimeMillis()
      for (x <- mapped) { x + 2 }
      val end = System.currentTimeMillis()
      println("Iteration " + iter + " took " + (end-start) + " ms")
    }
    sc.stop()
  }
}
```

Create Spark Standalone applications - Scala

```scala
/* SimpleApp.scala */
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "YOUR_SPARK_HOME/README.md" // Should be some file on your system
    val conf = new SparkConf().setAppName("Simple Application")
    val sc = new SparkContext(conf)
    val logData = sc.textFile(logFile, 2).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println("Lines with a: %s, Lines with b: %s".format(numAs, numBs))
  }
}
```

Import statements

SparkConf and SparkContext

Transformations + Actions

Create Spark Standalone applications - Java

```java
/* SimpleApp.java */
import org.apache.spark.api.java.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.function.Function;

public class SimpleApp {
  public static void main(String[] args) {
    String logFile = "YOUR_SPARK_HOME/README.md"; // Should be some file on your system
    SparkConf conf = new SparkConf().setAppName("Simple Application");
    JavaSparkContext sc = new JavaSparkContext(conf);
    JavaRDD<String> logData = sc.textFile(logFile).cache();

    long numAs = logData.filter(new Function<String, Boolean>() {
      public Boolean call(String s) { return s.contains("a"); }
    }).count();

    long numBs = logData.filter(new Function<String, Boolean>() {
      public Boolean call(String s) { return s.contains("b"); }
    }).count();

    System.out.println("Lines with a: " + numAs + ", lines with b: " + numBs);
  }
}
```

Import statements

JavaSparkContext

Transformations + Actions

Run Standalone Applications

- Define the dependencies - can use any system build (Ant, sbt, Maven)

- Example
  - Scala -> simple.sbt
  - Java -> pom.xml
  - Python -> --py-files argument (not needed for SimpleApp.py)

- Create the typical directory structure with the files

Scala using SBT :	Java using Maven:
./simple.sbt ./src ./src/main ./src/main/scala ./src/main/scala/SimpleApp.scala	./pom.xml ./src ./src/main ./src/main/java ./src/main/java/SimpleApp.java

- Create a JAR package containing the application's code
  - Scala: sbt
  - Java: mvn
  - Python: submit-spark

  - Use spark-submit to run the program

Submit Applications to the Cluster

- Package application into a JAR (Scala/Java) or set of .py or .zip (Python)

- Use spark-submit under the $SPARK_HOME/bin directory

```
spark-submit \
--class <main-class> \
--master <master-url> \
--deploy-mode <deploy-mode> \
-- conf <key>=<value> \
… # other-options …
<application-jar> \
[application-arguments]
```

- spark-submit --help will show you the other options

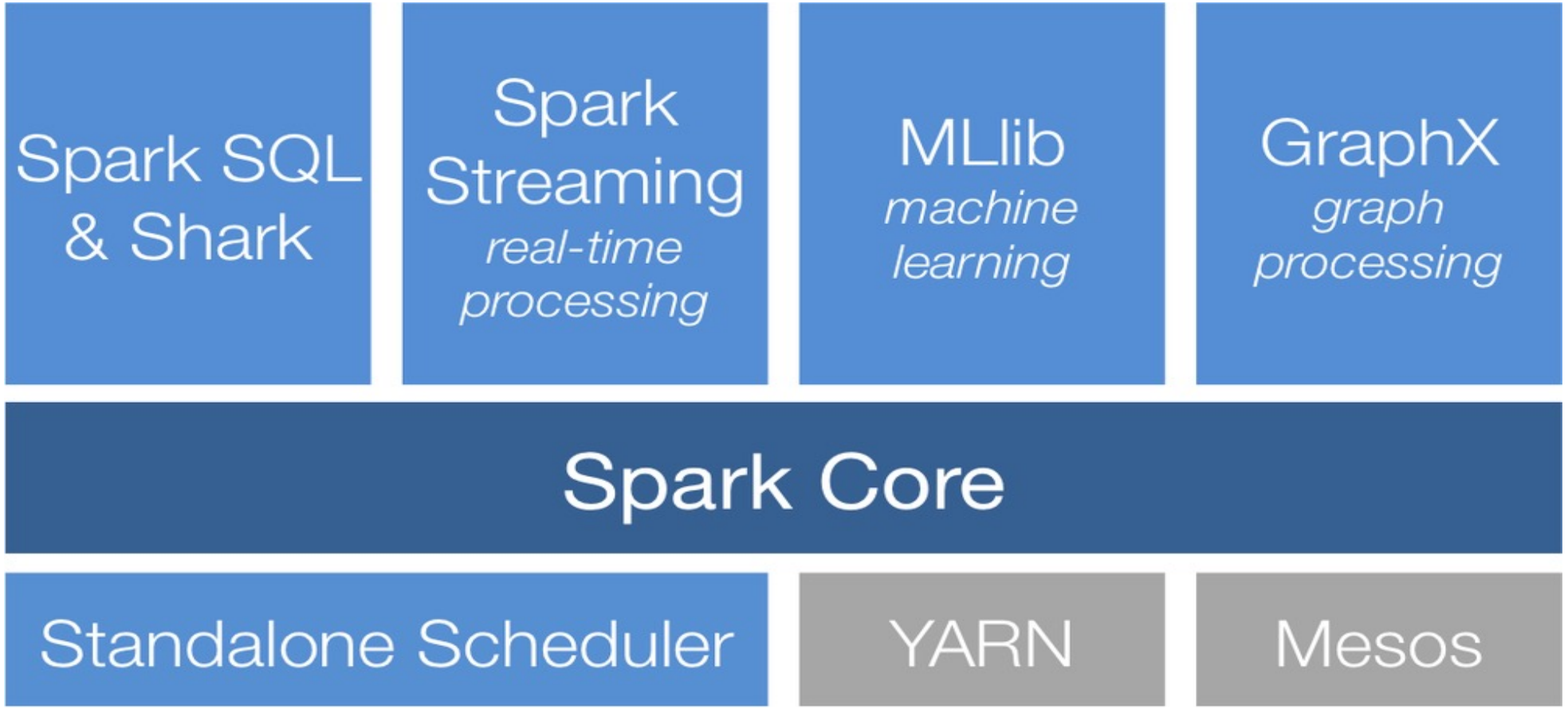
- Example:

```
spark-submit \
--class org.apache.spark.examples.sparkPi \
--master local[8] \
/path/to/examples.jar \
100
```

Spark Libraries

Spark Libraries

- Extension of the core Spark API
- Improvements made to the core are passed to these libraries
- Little overhead to use with the Spark Core



Spark SQL

- **Allows relational queries expressed in**
  - SQL
  - HiveQL
  - Scala

- **SchemaRDD**
  - Row objects
  - Schema
  - Created from:
    - Existing RDD
    - Parquet file
    - JSON dataset
    - HiveQL agianst Apache Hive

- **Supports Scala, Java and Python**

Spark SQL - Getting Started

- SQLContext
  - Created from a SparkContext
  - Scala:

    ```
    val sc : SparkContext //  An existing SparkContext
    val sqlContext = new org.apache.spark.sql.SQLContext(sc)
    ```

  - Java:

    ```
    JavaSparkContext sc = …; //  An existing JavaSparkContext
    JavaSQLContext sqlContext = new org.apache.spark.sql.api.java.JavaSQLContext(sc);
    ```

  - Python:

    ```
    from pyspark.sql import SQLContext sqlContext = SQLContext(sc)
    ```

- Import a library to convert an RDD to a SchemaRDD
  - **Scala only**: *import sqlContext.createSchemaRDD*

- SchemaRDD data sources:
  - Inferring the schema using reflection
  - Programmatic interface

Spark SQL - Getting Started (cont'd)

- SQLContext
  - Scala for spark > version 2

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder().appName("Spark SQL").getOrCreate() //already in spark-shell

val sqlContext = spark.sqlContext
```

- For implicit conversions like converting RDDs to DataFrames:

```
import spark.implicits._
```

Spark SQL - Inferring Schema Using Reflection

- The case class in Scala defines the schema of the table

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

- The arguments of the case class become the names of the columns

- Create the RDD of the *Person* object

  val people = sc.textFile("labdata/people.txt").map(_.split(","))
  .map(p => Person(p(0),p(1).trim.toInt)).toDF()

- Register the RDD as a table

  people.createOrReplaceTempView("people")

- Run SQL statements using the sql method provided by the SQLContext

  val teenagers = sqlContext.sql("SELECT name, age FROM people WHERE age >13 AND age<=18")

- The results of the queries are SchemaRDD. Normal RDD operations work on them

  teenagers.map(t => "Name: "+t(0)+" Age: "+t(1)).collect().foreach(println)

Spark SQL - Programmatic Interface

- Use when you cannot define the case classes ahead of time
- Create the RDD:

```
val people = sc.textFile("/labdata/people.txt")
```

- Three steps to create the SchemaRDD:
  1. Create an RDD of ***Rows*** from the original RDD

```
val schemaString = "name age"
```

  2. Import SQL types

```
import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType}
Or…
import org.apache.spark.sql.types._
```

  3. Create the schema represented by a *StructType* matching the structure of the ***Rows*** in the RDD from step 1.

```
val schema = StructType( schemaString.split(" ")
.map(fieldname => StructField( fieldname, StringType, true)))
```

Spark SQL - Programmatic Interface (cont'd)

4. Apply the schema to the RDD of ***Rows*** using the *applySchema* method.

```
val rowRDD = people.map(_.split(",")).map( p => Row(p(0),p(1).trim))
val peopleDataFrame = sqlContext.createDataFrame(rowRDD, schema)
```

- Register the RDD as a table

```
peopleDataFrame.registerTempTable("people")
```

- Run SQL statements using the sql method provided by the SQLContext

```
val  results = sqlContext.sql(" SELECT name FROM people")
results.map( t => "Name: "+t(0)).collect().foreach(println)
```

Spark SQL - Hands on!

- Spark SQL provides the ability to write relational queries to be run on Spark. There is the abstraction SchemaRDD which is to create an RDD in which you can run SQL, HiveQL, and Scala.

- In this example, we will use SQL to find out the average weather and precipitation for a given time period in New York. The purpose is to demonstrate how to use the Spark SQL libraries on Spark.

- Let's take a look at the nycweather.csv data:

```
"2013-01-01",1,0
"2013-01-02",-2,0
"2013-01-03",-2,0
"2013-01-04",1,0
"2013-01-05",3,0
"2013-01-06",4,0
"2013-01-07",5,0
"2013-01-08",6,0
"2013-01-09",7,0
"2013-01-10",7,0
```

There are three columns in the dataset, the date, the mean temperature in Celsius, and the precipitation for the day. Since we already know the schema, **we will infer the schema using reflection.**

Spark SQL - Hands on! (cont'd)

- Import a library for creating a SchemaRDD.

  import sqlContext.implicits._

- Create a case class in Scala that defines the schema of the table

  case class Weather(date: String, temp: Int, precipitation: Double)

- Create the RDD of the Weather object. You first load in the file, and then you map it by splitting it up by the commas and then another mapping to get it into the Weather class.

  val weather = sc.textFile("/labdata/nycweather.csv").map(_.split(",")). map(w => Weather(w(0), w(1).trim.toInt, w(2).trim.toDouble)).toDF()

- Next you need to register the RDD as a table. Type in:

  weather.createOrReplaceTempView("weather")

Spark SQL - Hands on! (cont'd)

- At this point, you are ready to create and run some queries on the RDD. You want to get a list of the hottest dates with some precipitation. Type in:

  *val hottest_with_precip = sqlContext.sql("SELECT * FROM weather WHERE precipitation > 0.0 ORDER BY temp DESC")*

  hottest_with_precip.collect()

- Print the top hottest days with some precipitation out to the console:

  hottest_with_precip.map(x => ("Date: " + x(0), "Temp : " + x(1), "Precip: " + x(2))).take(10).foreach(println)

Spark Streaming

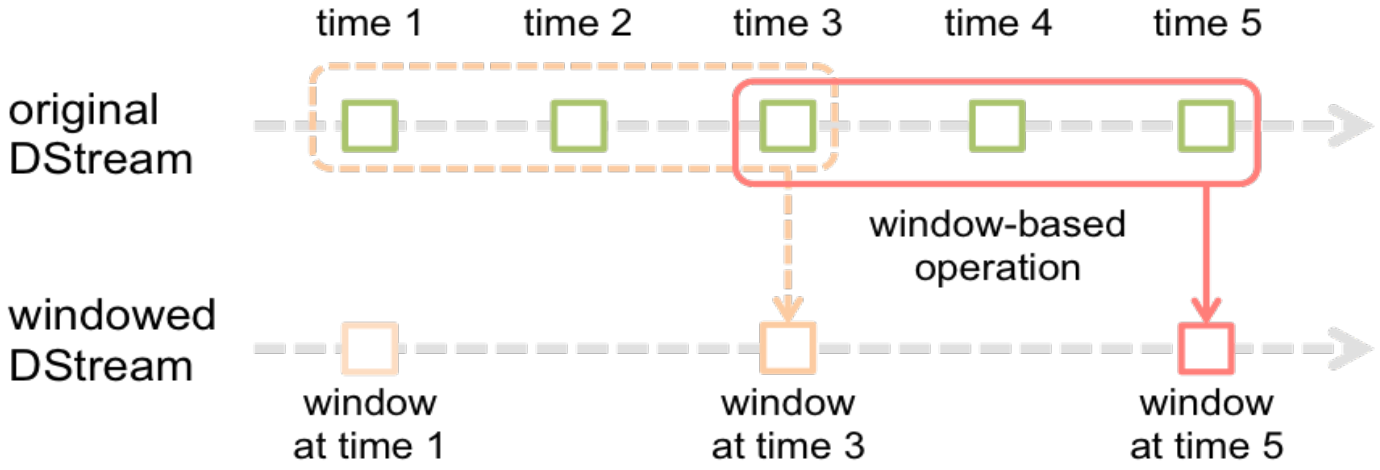
- Scalable, high-throughput, fault tolerant stream processing of live data streams
- Receives live input data and divides into small batches which are processed and returned as batches
- Dstream - sequence of RDD
- Currently supports Scala and Java
- Limited Python support from Spark 1.2

- **Receives Data from:**
  - Kafka
  - Flume
  - HDFS / S3
  - Kinesis
  - Twitter
- **Pushes Data out to:**
  - HDFS
  - Databases
  - Dashboards

Spark Streaming - Internals

- The input stream (Dstream) goes into Spark Streaming
- Breaks up into batches
- Feeds into the Spark engine for processing
- Generate the final results in streams of batches

- Sliding window operations:
  - Windowed computations
    - Window length
    - Sliding interval
    - *reduceByKeyAndWindow*



Spark Streaming - Getting Started

- Scenario: Count the number of words coming in from the TCP socket.

- Import the Spark Streaming classes and some implicit conversion

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
```

- Create the StreamingContext object

```
val ssc = new StreamingContext(sc, Seconds(1))
```

- Create a DStream

```
val lines = ssc.socketTextStream("localhost",7777)
```

- Split the lines into words

```
val words = lines.flatMap(_.split(" "))
```

- Count the words

```
val wordCounts = words.map(word => (word, 1)).reduceByKey(_+_)
wordCounts.print()
```

Spark Streaming - Getting Started (cont'd)

- No real processing happens until you tell it

  ssc.start() // Start the computation
  ssc.awaitTermination() // wait for the computation to terminate

- The entire code and application can be found in the NetworkWordCount example @github
  https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/streaming/NetworkWordCount.scala

Spark Streaming - Hands on!

- In this Example, taxi trip data will be streamed using a socket connection and then analyzed to provide a summary of number of passengers by taxi vendor. This will be implemented in the Spark shell using Scala.

- There are two relevant files for this section. The first one is the *nyctaxi100.csv* which will serve as the source of the stream. The other file is a python file, *taxistreams.py*, which will feed the csv file through a socket connection to simulate a stream.

- Turn off logging so that you can see the output of the application and Import the required libraries:

```
import org.apache.log4j.Logger
import org.apache.log4j.Level
Logger.getLogger("org").setLevel(Level.OFF)
Logger.getLogger("akka").setLevel(Level.OFF)

import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
```

Spark Streaming - Hands on! (cont'd)

- Create the StreamingContext by using the existing SparkContext (sc). It will be using a 1 second batch interval, which means the stream is divided to 1 sec batches and each batch becomes a RDD

```
val ssc = new StreamingContext(sc, Seconds(1))
```

- Create the socket stream that connects to the localhost socket 7777. This matches the port that the Python script is listening on. Each batch from the Stream be a lines RDD.

```
val lines = ssc.socketTextStream("localhost", 7777)
```

- Next, put in the business logic to split up the lines on each comma and mapping pass(15), which is the vendor, and pass(7), which is the passenger count. Then this is reduced by key resulting in a summary of number of passengers by vendor.

```
val pass = lines.map(_.split(","))
    .map(pass=>(pass(15), pass(7).toInt))
    .reduceByKey(_+_)
```

- Print out to console

```
pass.print()
```

Spark Streaming - Hands on! (cont'd)

- The next two line starts the stream.

  ssc.start()
  ssc.awaitTermination()

- It will take a few cycles for the connection to be recognized, and then the data is sent. In this case, 2 rows per second of taxi trip data is receive in a 1 second batch interval.

Spark MLlib

- Mllib for machine learning library
  - Common algorithms and utilities
    - Classification
    - Regression
    - Clustering
    - Collaborative filtering
    - Dimensionality reduction

- We will have a practical example on next slide

Spark MLlib - Hands on!

- Spark will be used to acquire the K-Means clustering for drop-off latitudes and longitudes of taxis for 3 clusters.

- The sample data contains a subset of taxi trips with *hack license, medallion, pickup date/time, drop off date/time, pickup/drop off latitude/longitude, passenger count, trip distance, trip time* and other information. As such, this may give a good indication of where to best to hail a cab.

- This is only a subset of the file that you used in a previous example. If you ran this example on the full dataset, it would take a long time as we are only running on a test environment with limited resources.

.

- Import the needed packages for K-Means algorithm and Vector packages:

```
import org.apache.spark.mllib.clustering.KMeans
import org.apache.spark.mllib.linalg.Vectors
```

- Create an RDD

```
val taxiFile = sc.textFile("/labdata/nyctaxisub.csv")
```

- Determine the number of rows in taxiFile.

```
taxiFile.count()
```

Spark MLlib - Hands on! (cont'd)

- Cleanse the data.

```
val taxiData=taxiFile.filter(_.contains("2013"))
.filter(_.split(",")(3)!="" )      //dropoff_latitude
 .filter(_.split(",")(4)!="")      //dropoff_longitude
```

The **first filter limits** the rows to those that occurred in the year 2013. This will also **remove any header in the file**. The third and fourth columns contain the drop off latitude and longitude. **The transformation will throw exceptions if these values are empty.**

- Do another count to see what was removed.

```
taxiData.count()
```

In this case, if we had used the full set of data, it would have filtered out a great many more lines.

- To fence the area roughly to New York City use this command:

```
val taxiFence=taxiData
                .filter(_.split(",")(3).toDouble>40.70)
                .filter(_.split(",")(3).toDouble<40.86)
                .filter(_.split(",")(4).toDouble>(-74.02))
                .filter(_.split(",")(4).toDouble<(-73.93))
```

Spark MLlib - Hands on! (cont'd)

- Determine how many are left in taxiFence:

```
taxiFence.count()
```

- Create Vectors with the latitudes and longitudes that will be used as input to the K-Means algorithm.

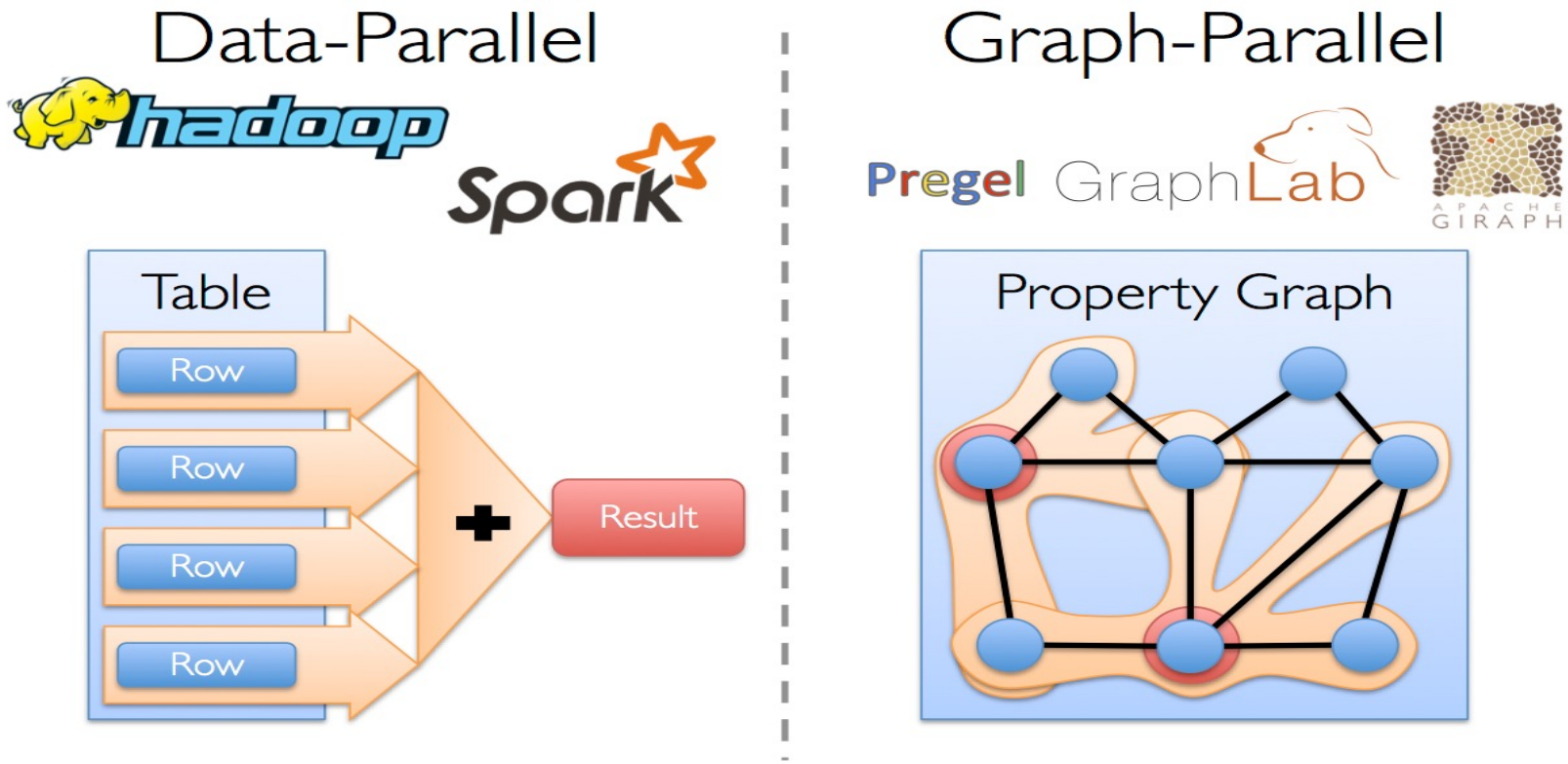
```
val taxi=taxiFence.map{ line=> Vectors.dense( line.split(',').slice(3,5).map(_ .toDouble))}

taxi.cache()
val iterationCount=10
val clusterCount=3

val model=KMeans.train(taxi,clusterCount,iterationCount)
val clusterCenters=model.clusterCenters.map(_.toArray)

clusterCenters.foreach(lines=>println(lines(0),lines(1)))
```

Now we know the map co-ordinates. Try yourself the printed co-ordinates in google maps and you won't be surprised by the results!

Spark GraphX

- GraphX for graph processing
  - Graphs and graph parallel computation
  - Social networks and language modeling

- We will have a practical exercise on finding attributes associated with top users.



Spark GraphX - Hands on!

- *users.txt* is a set of users and followers is the relationship between the users. Take a look at the contents of these two files.

Users:
1,BarackObama,Barack Obama
2,ladygaga,Goddess of Love
3,jeresig,John Resig

Followers:
2 1
4 1
1 2

- Import the GraphX package:

```
import org.apache.spark.graphx._
```

- Create the users RDD and parse into tuples of user id and attribute list:

```
val users = (sc.textFile("/labdata/users.txt").map(line => line.split(",")).map(parts =>
(parts.head.toLong, parts.tail)))

users.take(5)
```

- Parse the edge data, which is already in *userId -> userId* format:

```
val followerGraph = GraphLoader.edgeListFile(sc, ("/labdata/followers.txt"))
```

Spark GraphX - Hands on! (cont'd)

- Attach the user attributes:

```
val graph = followerGraph.outerJoinVertices(users) {
    case (uid, deg, Some(attrList)) => attrList
    case (uid, deg, None) => Array.empty[String]
}
```

- Restrict the graph to users with usernames and names:

```
val subgraph = graph.subgraph(vpred = (vid, attr) => attr.size == 2)
```

- Compute the PageRank:

```
val pagerankGraph = subgraph.pageRank(0.001)
```

- Get the attributes of the top pagerank users:

```
val userInfoWithPageRank = subgraph.outerJoinVertices(pagerankGraph.vertices) {
    case (uid, attrList, Some(pr)) => (pr, attrList.toList)
    case (uid, attrList, None) => (0.0, attrList.toList)
}
```

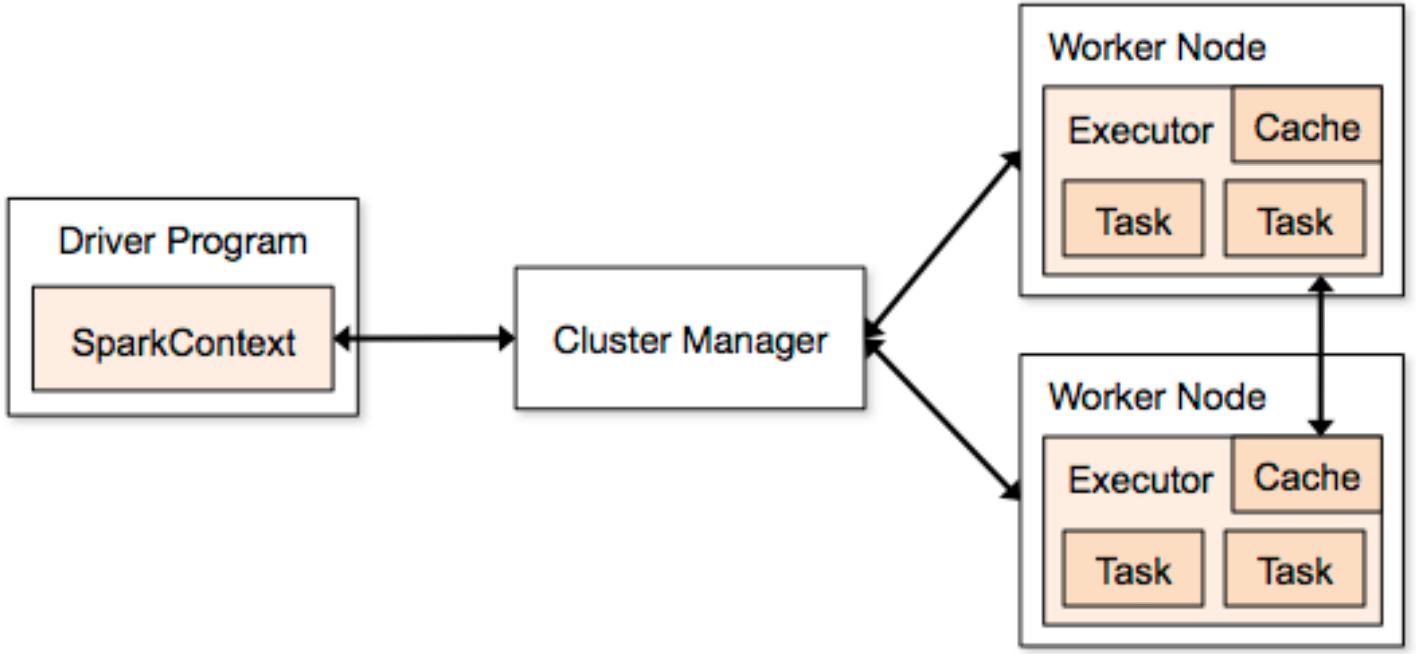
- Print the line out:

```
println(userInfoWithPageRank.vertices.top(5)(Ordering.by(_._2._1)).mkString("\n"))
```

Spark - Configuration / Monitoring / Tuning

Brief History of Spark

- Components
  - Driver
  - Cluster Master
  - Executors



- Cluster Manager
  - Standalone
  - Hadoop YARN
  - Apache Mesos

Spark Configuration

- **Three locations for configuration**
  - Spark properties
  - Environment variables
    conf/spark-env.sh
  - Logging
    log4j.properties

- **Override default configuration directory (*SPARK_HOME/Conf*)**
  - *SPARK_CONF_DIR*
    - spark-defaults.conf
    - spark-env.sh
    - log4j.properties
    - etc

- **Spark shell verbose**
  - To view ERRORs only, changed the INFO variable to ERROR in the log4j.properties
    $SPARK_HOME/conf/log4j.properties

Spark Configuration - Spark Properties

- Set application properties via the SparkConf object

```
val conf = new SparkConf().setMaster("local").setAppName("name")
.set("spark.executor.memory", "1G")

val sc = new SparkContext(conf)
```

- Dynamically setting Spark properties
  - Create a SparkContext with an empty conf

```
val sc = new SparkContext( new SparkConf() )
```

  - Supply the configuration values during runtime

```
spark-submit --name "My app" --master yarn --deploy-mode client  --conf
"spark.executor.extraJavaOptions=-XX:+PrintGCDetails -XX:+PrintGCTimeStamps" myApp.jar
```

  - conf/spark-defaults.conf

- Application web UI

```
http://<driver>:4040 // In YARN cluster -> http://<master>:8088
```

Spark Monitoring - Web UI / History Server

- Port 4040 // in our case: **18088**
- Shows current application

- **Contains**
  - A list of scheduler stages and tasks
  - A summary of RDD sizes and memory usage
  - Environmental information
  - Information about the running executors

- **Configure the history server to set**
  - Memory allocated
  - JVM options
  - Public address for the server
  - Various properties

Spark Tuning

- **Data serialization**
  - Java serialization
  - Kryo serialization

    conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

- **Memory tuning**
  - Amount of memory used by the objects
    - Avoid Java features that add overhead
    - Go with arrays or primitive types
    - Avoid nested structures when possible
  - Cost of accessing those objects
    - Serialized RDD storage
  - Overhead of garbage collection
    - Analyze the garbage collection
    - SPARK_JAVA_OPTS

      -verbose:gc -XX:+PrintGCDetails -XX:PrintGCTimeStamps to your SPARK_JAVA_OPTS

Spark Tuning - Other Considerations

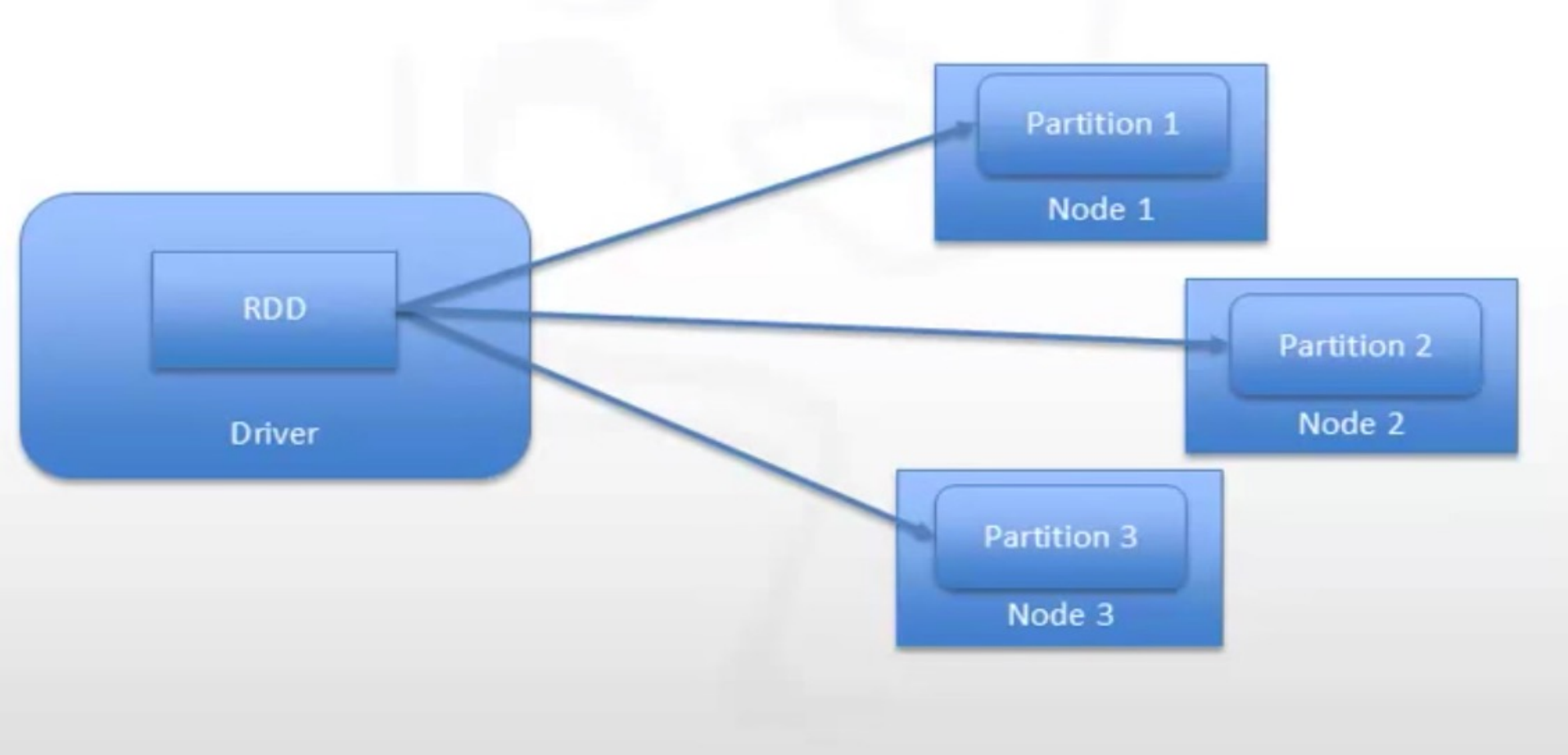
- **Level of parallelism**
  - Automatically set according to the file size
  - Optional parameters such as SparkContext.textFile
  - 2-3 tasks per CPU core in the cluster
  - *spark.default.parallelism*

- **Memory usage of reduce tasks**
  - OutOfMemoryError can be resolved by increasing the level of parallelismCost of accessing those objects
- **Broadcasting large variable**
- **Serialized size of each tasks are located on the master**
  - Tasks > 20KB worth optimizing

RDD Architecture

Review of RDD

- **Resilient Distributed Dataset** Collection of Elements **partitioned** across the nodes of the cluster that can be operated on in **parallel**.
- Maintains a list of dependencies (previous RDDs in its graph)



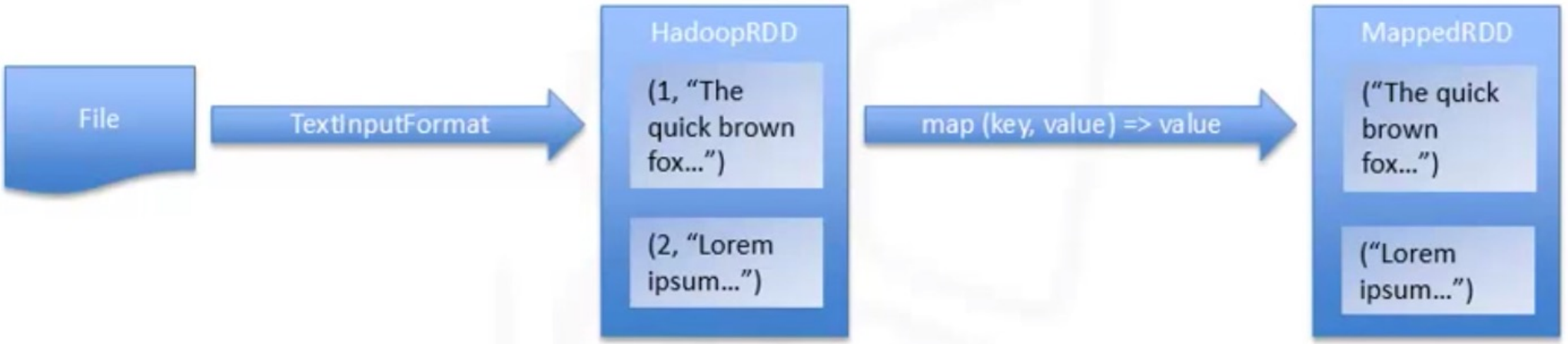
How an RDD is Generated

- Uses Hadoop InputFormat APIs to input data
  - support for many data stores
  - Not just HDFS
    - Local file systems
    - Cloud storage(Google, AWS, Azure, etc)
    - Hbase, Cassandra, MongoDB
    - Custom InputFormats

- InputFormat specifies splits and locality
  - Spark partitions correlate to HDFS splits
  - Observes data locality when possible
  - Optionally specify minimum partitioning

- Partitions Impact Parallelism
  - A task executes against a partition

Example: RDD from a Text File



TextFile API

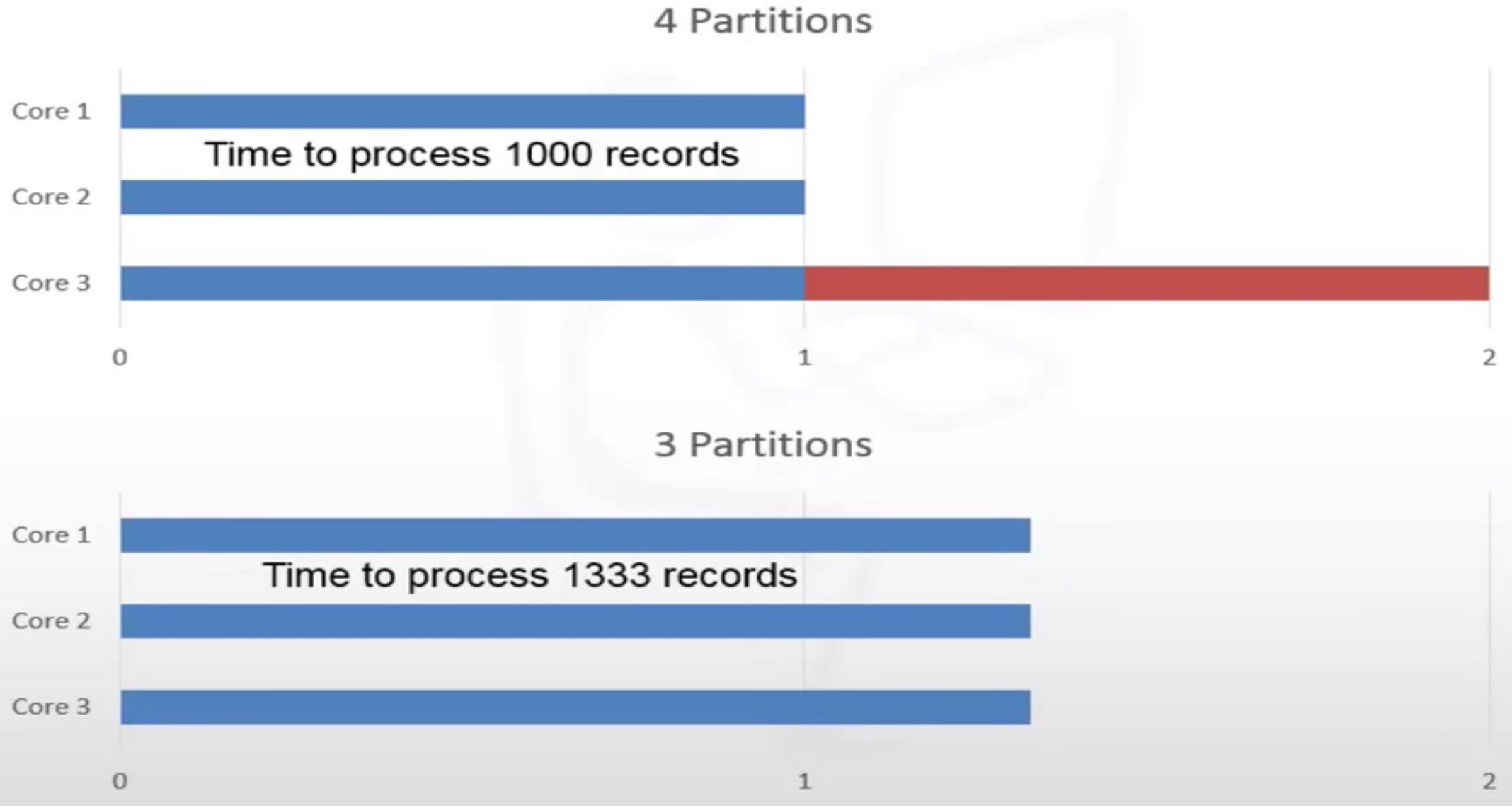
Partitions are inherited from Parent RDD

```
scala> val input = sc.textFile("/user/b-analytics/README.md")
input: org.apache.spark.rdd.RDD[String] = /user/b-analytics/README.md MapPartitionsRDD[1] at textFile at <console>:27

scala> input.toDebugString
res0: String =
(2) /user/b-analytics/README.md MapPartitionsRDD[1] at textFile at <console>:27 []
 |  /user/b-analytics/README.md HadoopRDD[0] at textFile at <console>:27 []

scala> input.partitions
res1: Array[org.apache.spark.Partition] = Array(org.apache.spark.rdd.HadoopPartition@691, org.apache.spark.rdd.HadoopPartition@692)
```

Partitioning Considerations

- Evenly distributed data
- Preferred locations
  - On systems like Hadoop and Cassandra, partitions should align with cores
- Number of CPU cores
- How long it takes to run a task
- Avoid **Shuffling**
  - Movement of data between nodes
  - Very Expensive
  - OOM errors
- It all depends on your data and the type of operations that you will perform

Partitioning Example



Factors that Affect Partitioning

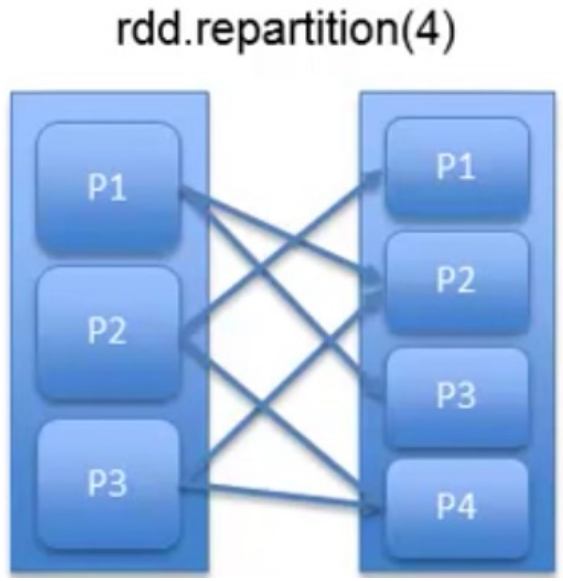
- Default partitioning is defined by input format
  - E.g. on Hadoop splits by HDFS cores

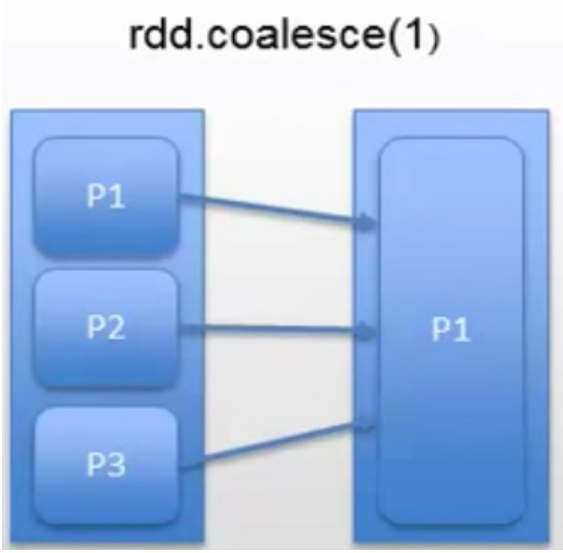
- **Repartition:** increase partitions
  - Rebalancing partitions after filter
  - Increase parallelism

.repartition(numPartitions : Int)

- **Coalesce:** decrease partitions WITHOUT shuffle
  - Consolidate before outputting to HDFS/external

coalesce(numPartitionsL Int, shuffle: Boolean = false)

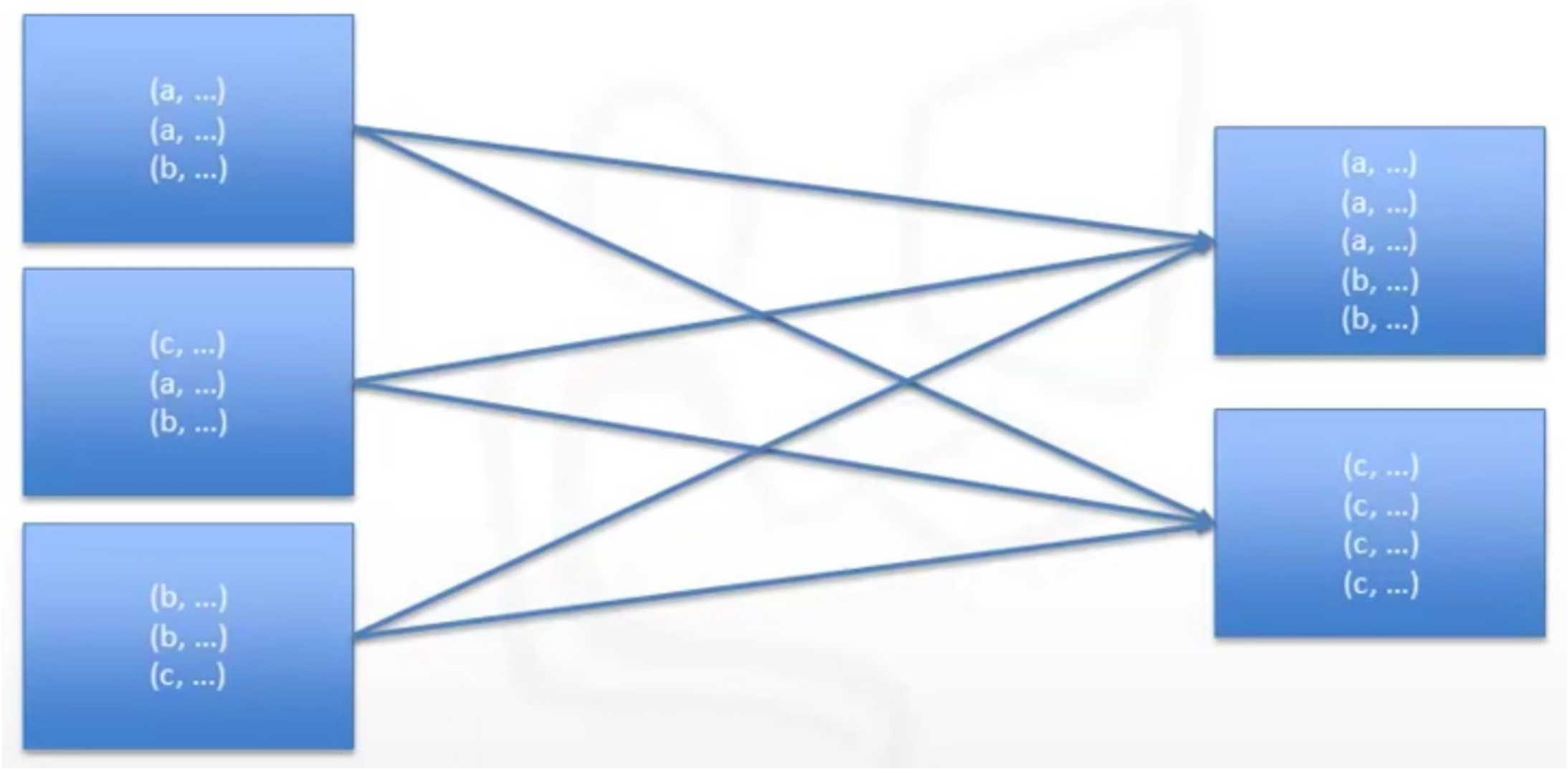




Partitioners for Key/Value RDDs

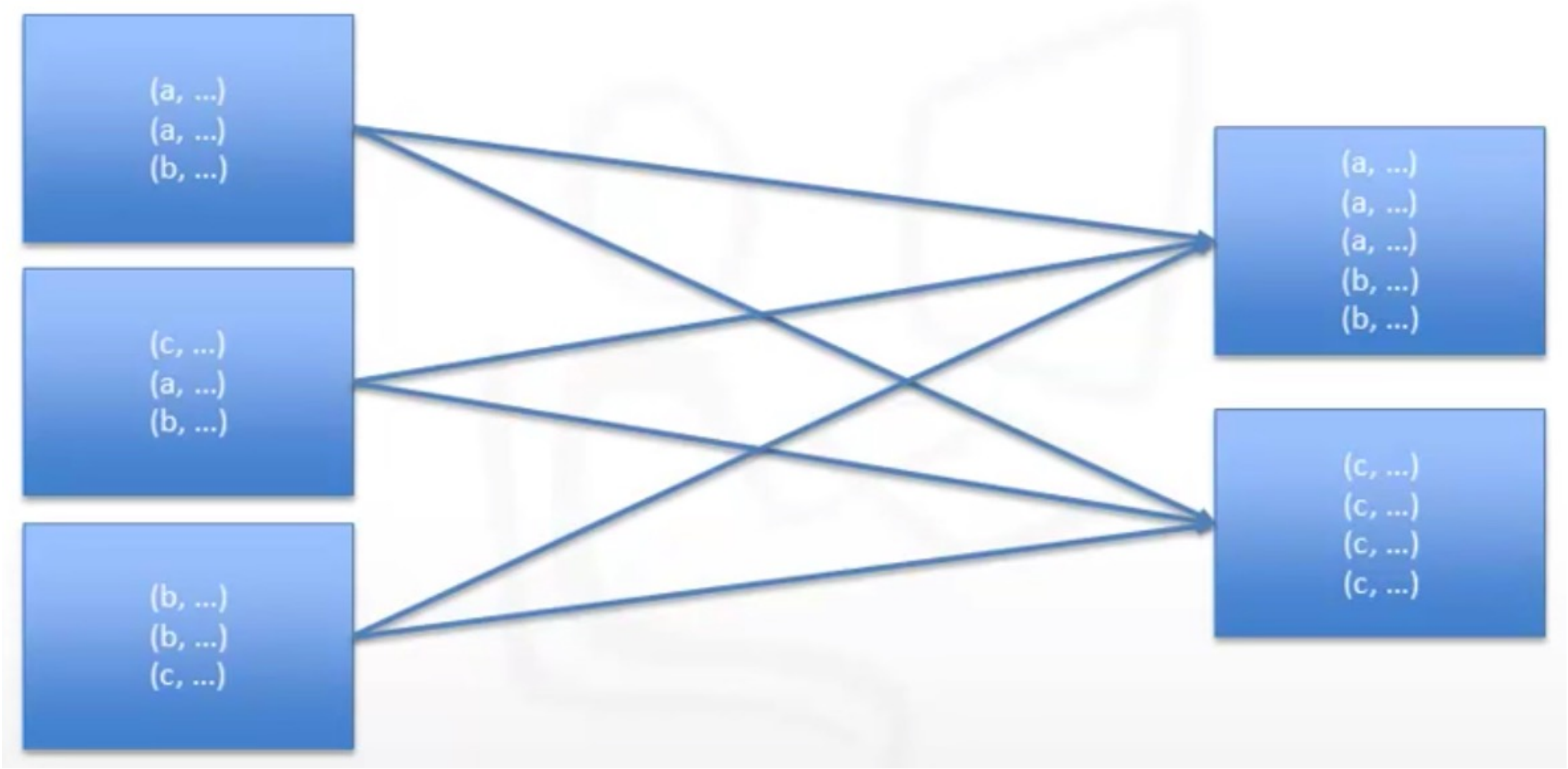
- Adding a key to an RDD does not change the partitioning
- Pairs with same key may not be in same partition (Co-located)
- co-location beneficial for many operations
- We can partition records by their keys
- **RangePartitioner**
- **HashPartitioner**
  - Ensures all pairs with the same key are co-located
  - Partition = key % numPartitions
- partitionBy causes a shuffle

Partitioners for Key/Value RDDs



rdd.partitionBy(new HashPartitioner(2))

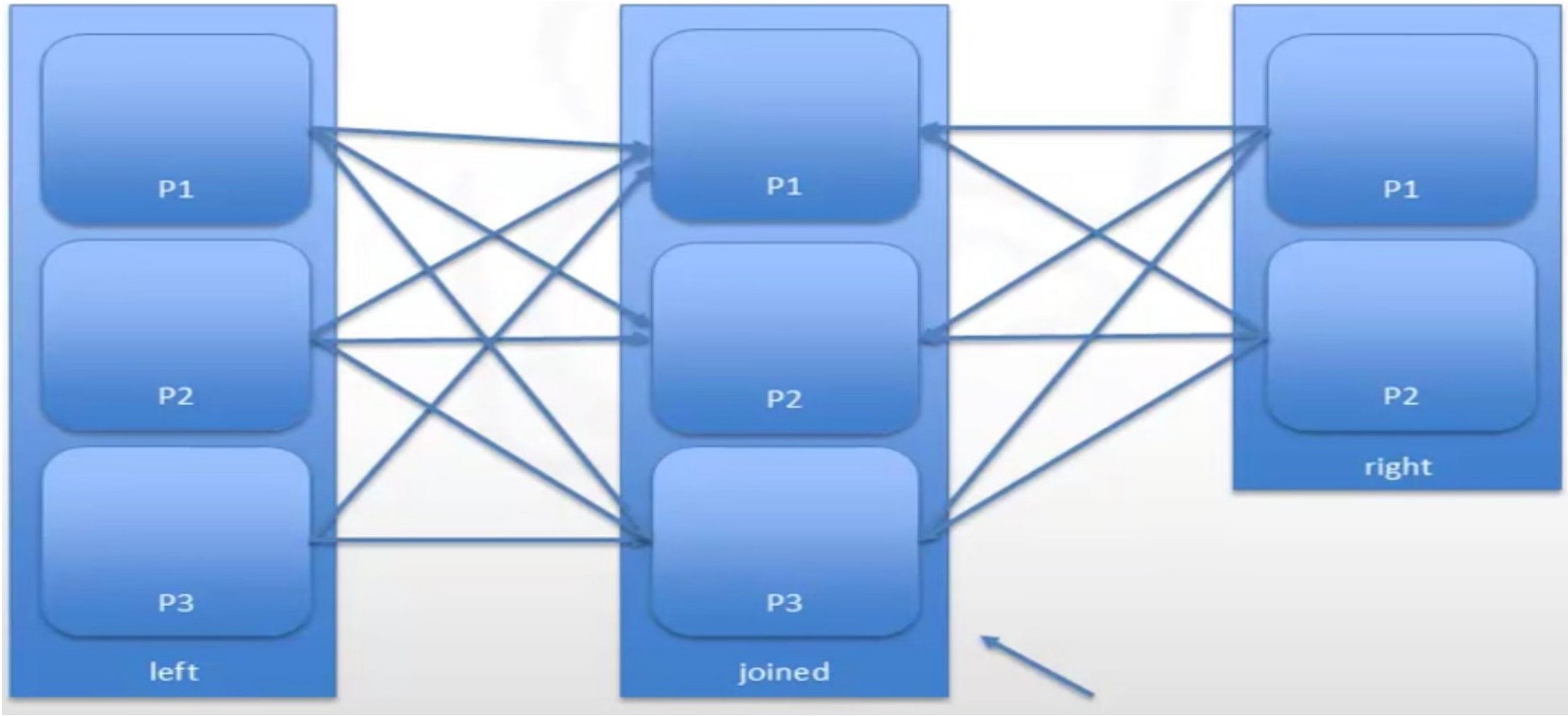
Partitioners for Key/Value RDDs



rdd.partitionBy(new HashPartitioner(2))

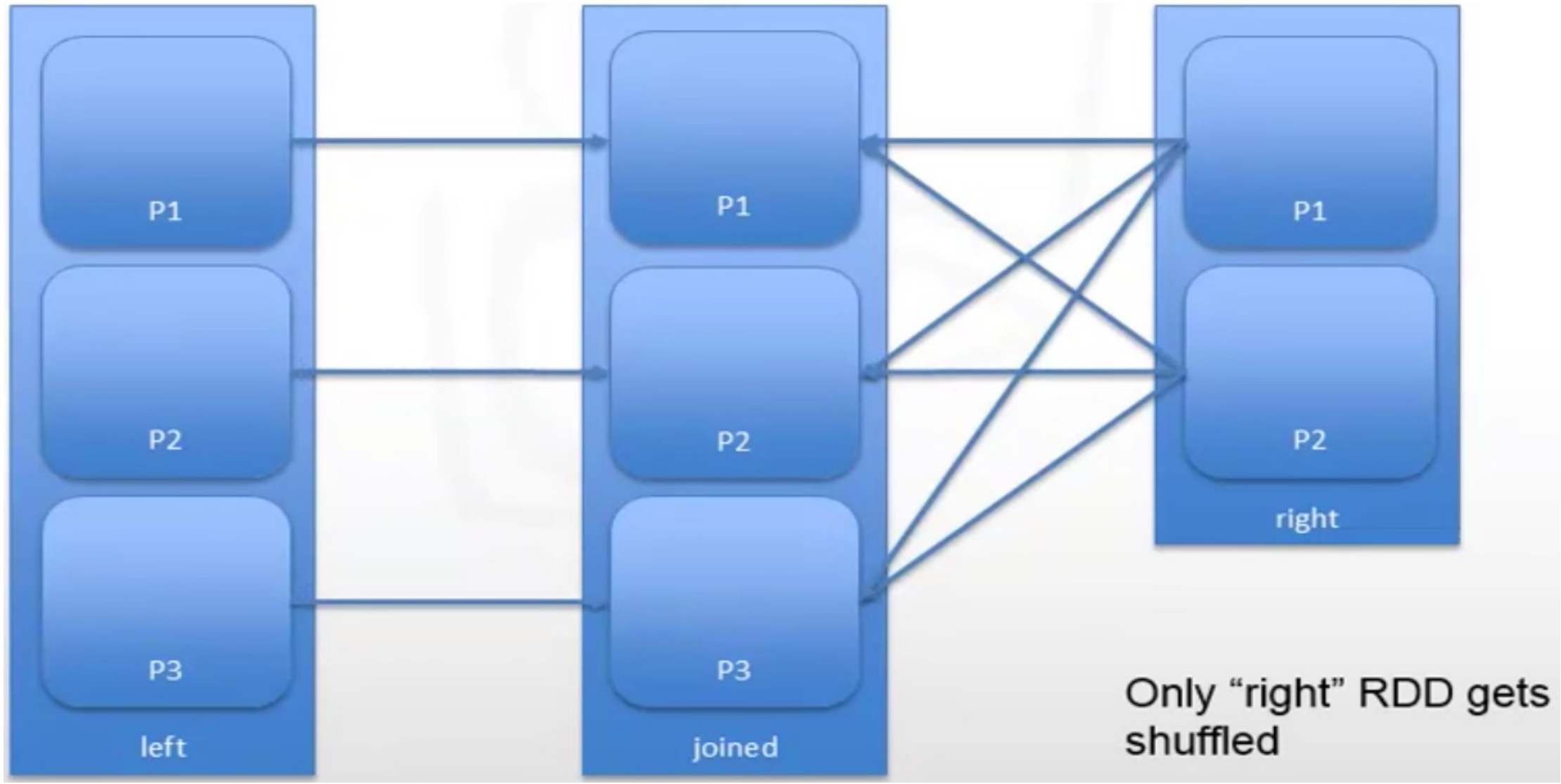
Partitioners for Key/Value RDDs

Both RDDs are shuffled



HashPartitioner with 3 partitions

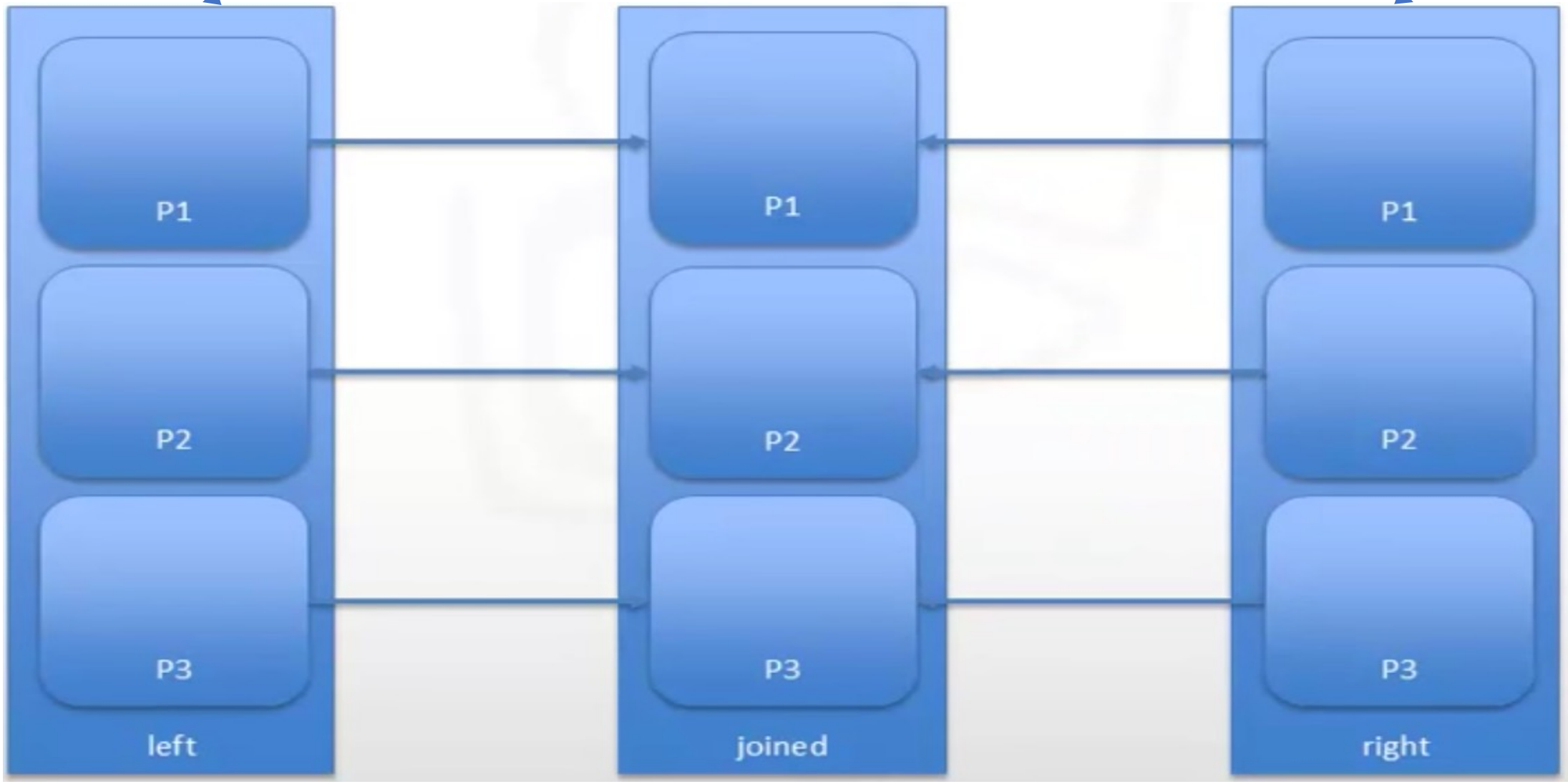
Optimizing Joins with a Partitioner



Already Partitioned

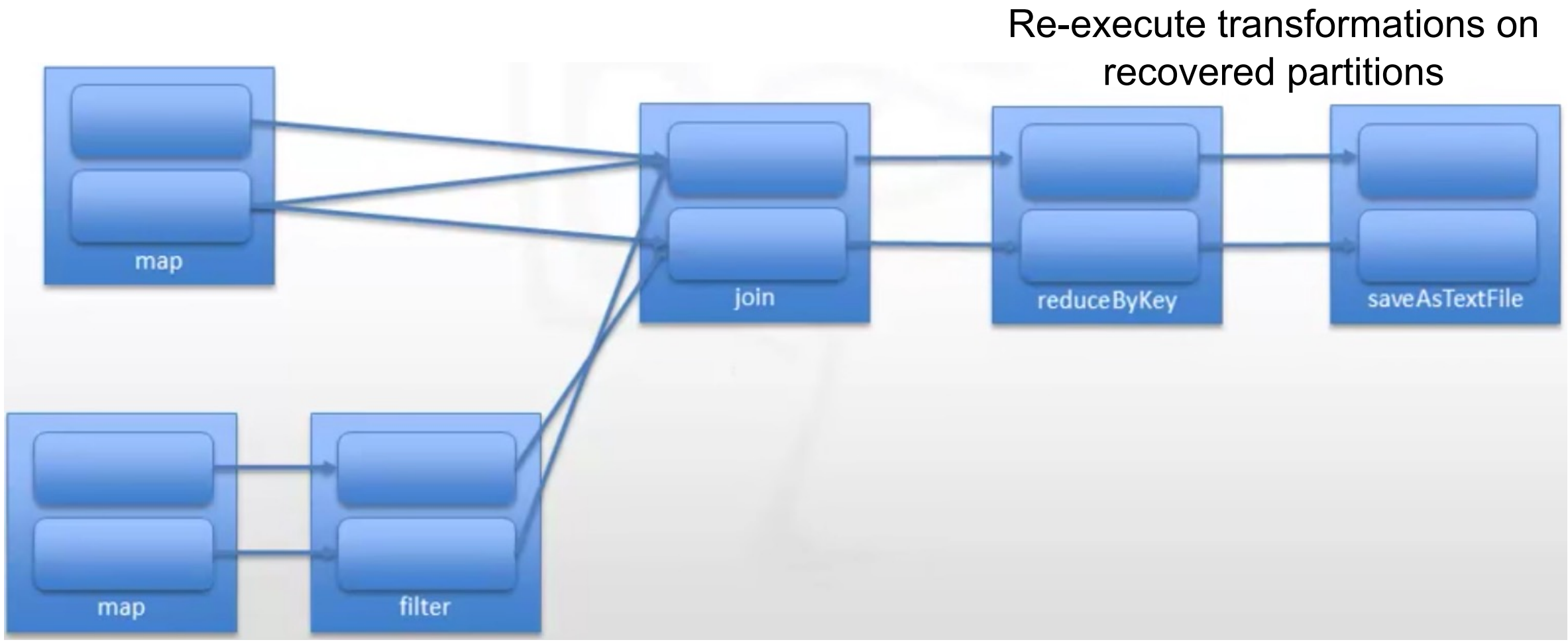
Optimizing Joins by Co-Partitioning

Similar keys, same partitioner,
same # of partitions



Resilience

- RDD lineage makes it resilient
- Re-compute results from root / persisted RDD



Executing Jobs

- **Job:** sequence of transformations initiated by an action
  - collect, reduce, first, take, foreach etc
- Job -> Stages -> Tasks
- DAG Scheduler analyzes RDD to generate stages and tasks
  - **Stage boundaries** are defined by shuffle dependencies (AKA wide dependency)
- Tasks are serialized and distributed to executors

Tasks

- **Task:** operation defined in driver for an RDD

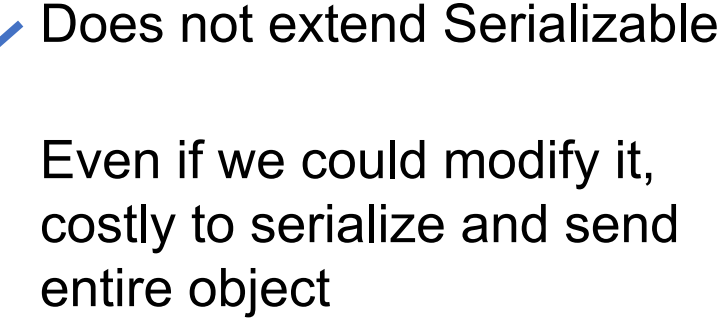
- Closure (AKA lambda) is serialized and deployed to each executor
  - Must be **serializable**
  - Including any references

- Be careful of large tasks
  - Serialization cost
  - Network I/O cost

Troubleshooting “Task Not Serializable”

```
Class MyHelper {
        def doSomething(input : String) : String = input
}

val helper = new MyHelper()

val output = input1.map( helper.doSomething(_) )
```

Troubleshooting “Task Not Serializable”




```
Class MyHelper extends Serializable {
        val inner = new ExternalClass()
        def doSomething(input : String) : String = inner.doStuff(input)
}

val helper = new MyHelper()

val output = input1.map( helper.doSomething(_) )
```

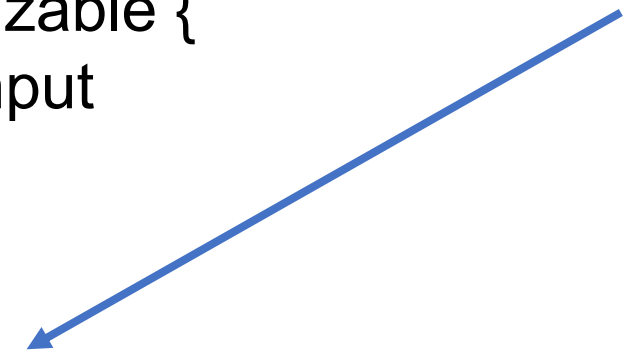
Troubleshooting “Task Not Serializable”

Instantiated locally in each task

```
Class MyHelper extends Serializable {
    @transient lazy val inner = new ExternalClass()
    def doSomething(input : String) : String = inner.doStuff(input)
}

val helper = new MyHelper()

val output = input1.map( helper.doSomething(_) )
```

Be Careful what you Serialize

```
  Class MyOtherHelper extends Serializable {
        def doSomething(input : Int) = input
}

object MySparkApp {
        val helper1 = new MyOtherHelper

        def main (args: Array[String]) {
                val sc = new SparkContext(new SparkConf())
                val input = sc.parallelize(1 to 1000)

                val output = input1.map( helper1.doSomething(_) )
                output.collect()
  }
}
```



Member variable of MySparkApp

Spark tries to serialize MySparkApp

Serializing would send entire app to every worker

Be Careful what you Serialize

```
  Class MyOtherHelper extends Serializable {
        def doSomething(input : Int) = input
}

object MySparkApp {
        val helper1 = new MyOtherHelper

        def main (args: Array[String]) {
                val sc = new SparkContext(new SparkConf())
                val input = sc.parallelize(1 to 1000)

                val localHelper = helper1
                val output = input1.map( localHelper.doSomething(_) )
                output.collect()
  }
}
```

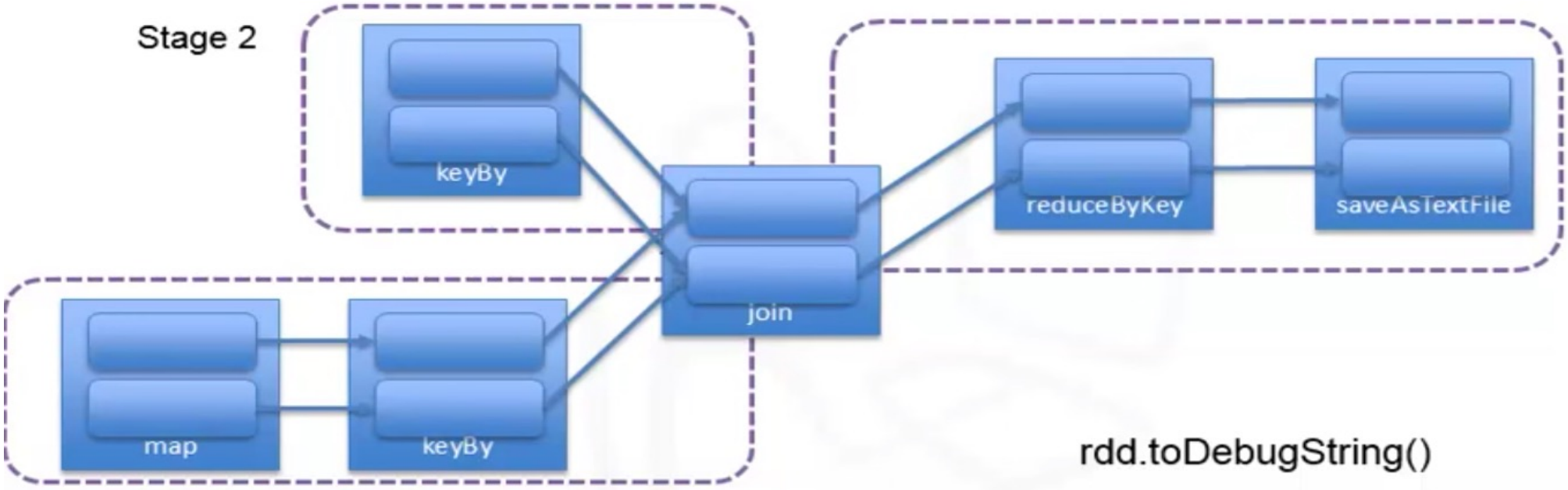
Member variable of MySparkApp

Spark tries to serialize MySparkApp

Serializing would send entire app to every worker

Solution: local reference to helper1

Identifying Stages



rdd.toDebugString()

```
(2) MapPartitionsRDD[316] at reduceByKey at <console>:84 []
 |  FlatMappedValuesRDD[311] at join at <console>:82 []
 |  MappedValuesRDD[310] at join at <console>:82 []
 |  CoGroupedRDD[309] at join at <console>:82 []
 +-(2) MappedRDD[307] at keyBy at <console>:75 []
 |  |  MappedRDD[306] at map at <console>:74 []
 |  |  FilteredRDD[305] at filter at <console>:73 []
 |  |  /stations MappedRDD[304] at textFile at <console>:68 []
 |  |  /stations HadoopRDD[303] at textFile at <console>:68 []
 +-(2) MappedRDD[308] at keyBy at <console>:76 []
    |  MappedRDD[302] at map at <console>:74 []
    |  FilteredRDD[301] at filter at <console>:73 []
    |  /trips MappedRDD[300] at textFile at <console>:68 []
    |  /trips HadoopRDD[299] at textFile at <console>:68 []
```

Handling Stragglers and Failures

- Speculative execution
  - Slow running tasks in stage are re-launched
  - spark.speculation = true (default = false)

- “Slow” is defined by spark speculation multiplier (default=1.5)
  - How many times slower a task is than the median

- Task failures due to memory issues, hardware, network, etc.
  - spark.task.maxFailures (default=4)

- May want to fail fast vs. running long task multiple times

- Be aware of side-effects
  - e.g. output to external system
  - if task fails and re-runs it might output same data again
  - Need to write custom code to be idempotent

Using Fair Scheduler Pools

```xml
<?xml version="1.0"?>
<allocations>
  <pool name="default">
    <schedulingMode>FAIR</schedulingMode>
    <weight>1</weight>
    <minShare>2</minShare>
  </pool>
  <pool name="user1">
    <schedulingMode>FIFO</schedulingMode>
    <weight>2</weight>
    <minShare>4</minShare>
  </pool>
  <pool name="user2">
    <schedulingMode>FIFO</schedulingMode>
    <weight>2</weight>
    <minShare>4</minShare>
  </pool>
</allocations>
```

```
conf.set("spark.scheduler.allocation.file", "/path/to/schedulerpool.xml")

sc.setLocalProperty("spark.scheduler.pool", "user1")

dataset.foreachPartition(part => {

})

sc.setLocalProperty("spark.scheduler.pool", null) // sets back to default
```

Concurrent Jobs

- Default First-In-First-Out (FIFO) Scheduler
  - Fine for single-user apps
  - Each Job gets as many resources it needs
  - Jobs from multiple threads can run in parallel if resources allow
  - But one large task will back things up

- Fair Scheduler is more appropriate for multi-user
  - spark.scheduler.mode = FAIR (default FIFO)
  - Resources allocated equally across jobs
  - Can define pools with weights / minimum share of resources
    - Pools can have theor own scheduler (default FIFO)
    - Useful for priority queues
    - e.g. Zeppelin gives each user a pool