GROUPTOWN

Apache Spark

Written in Feb 2020

isep
École d'ingénieurs du numérique

1. Introduction
2. Spark distributed Computing
3. Basics
4. Hands on
5. RDD lineage
6. Cache & Persistence
7. Partitions

Thibaut de Broca
Teacher @ ISEP
thibaut@grooptown.com

*“**Data** is the new **oil**?*

*Better: **Data** is the new **soil**.*”*

—

David McCandles

**soil = terre fertile*

2 février 2021

Databricks (“founder” de Spark) raised 1 Billion

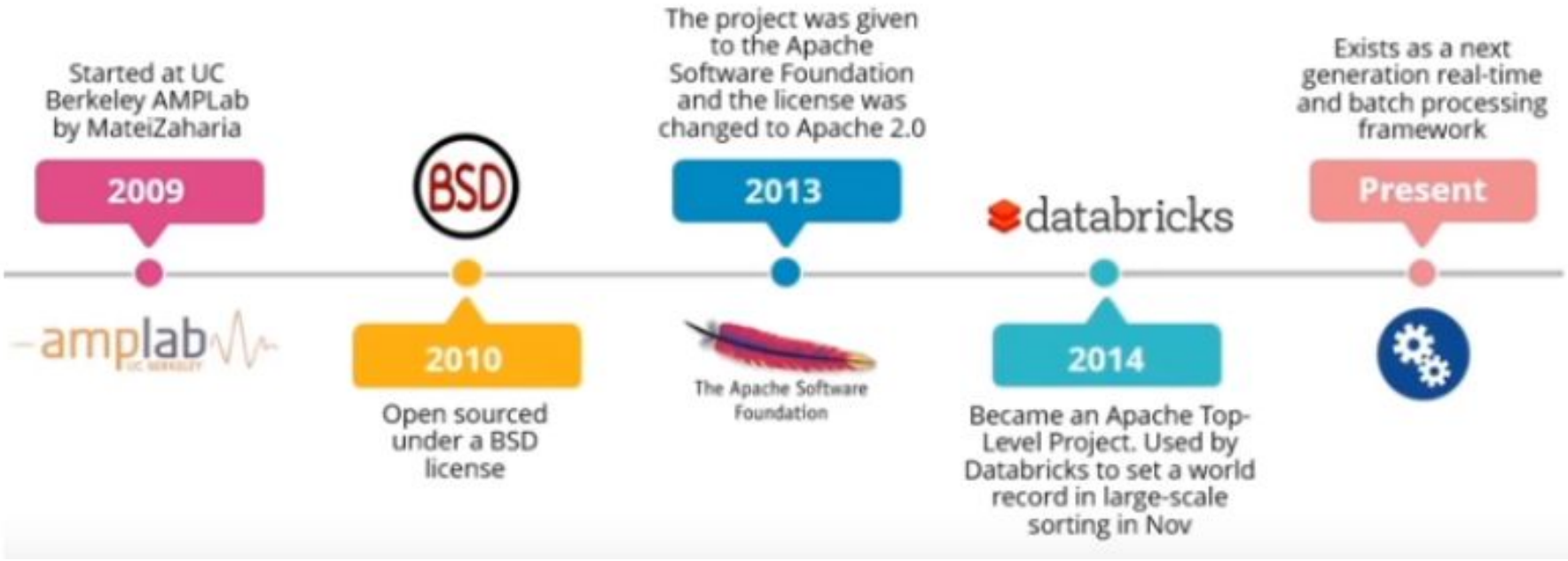
https://www.lebigdata.fr/databricks-leve-1-milliard

Introduction

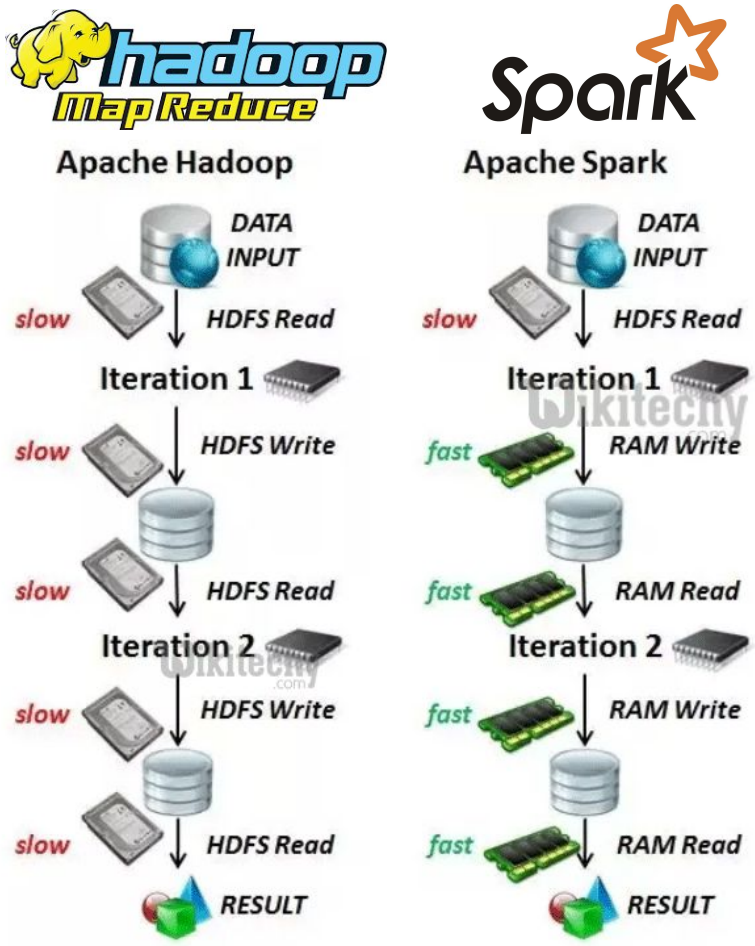
What is Spark ?

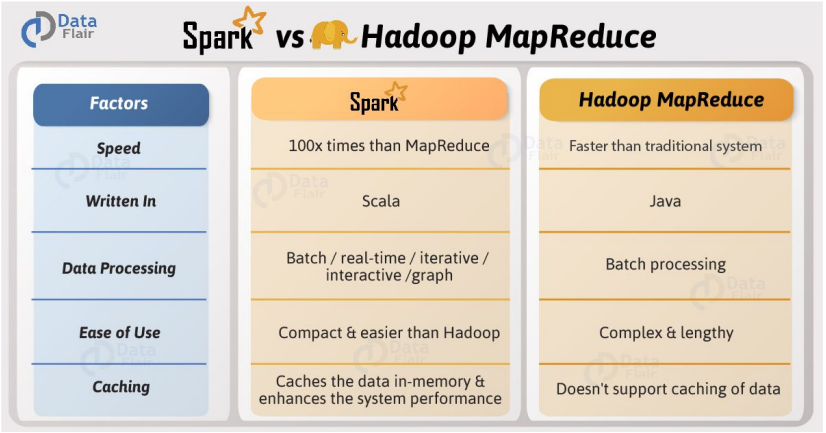
- ★ An open-source solution for processing large volumes of data in memory
- ★ Supports programming languages Scala, Python, R, Java.
- ★ Developed by a dynamic developer community
- ★ Provides an easy to use API (fluent) and supports the execution of SQL queries (Spark SQL)
- ★ Development of complex in memory programs (Spark/JAVA)
- ★ Program tuning (Spark/JAVA)

History

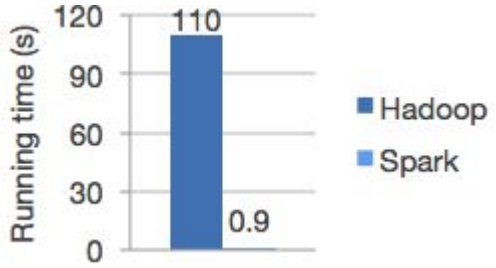

Started at UC Berkeley AMPLab by MateiZaharia
2009
amplab
BSD
2010
Open sourced under a BSD license
The project was given to the Apache Software Foundation and the license was changed to Apache 2.0
2013
The Apache Software Foundation
databricks
2014
Became an Apache Top-Level Project. Used by Databricks to set a world record in large-scale sorting in Nov
Exists as a next generation real-time and batch processing framework
Present

Motivation

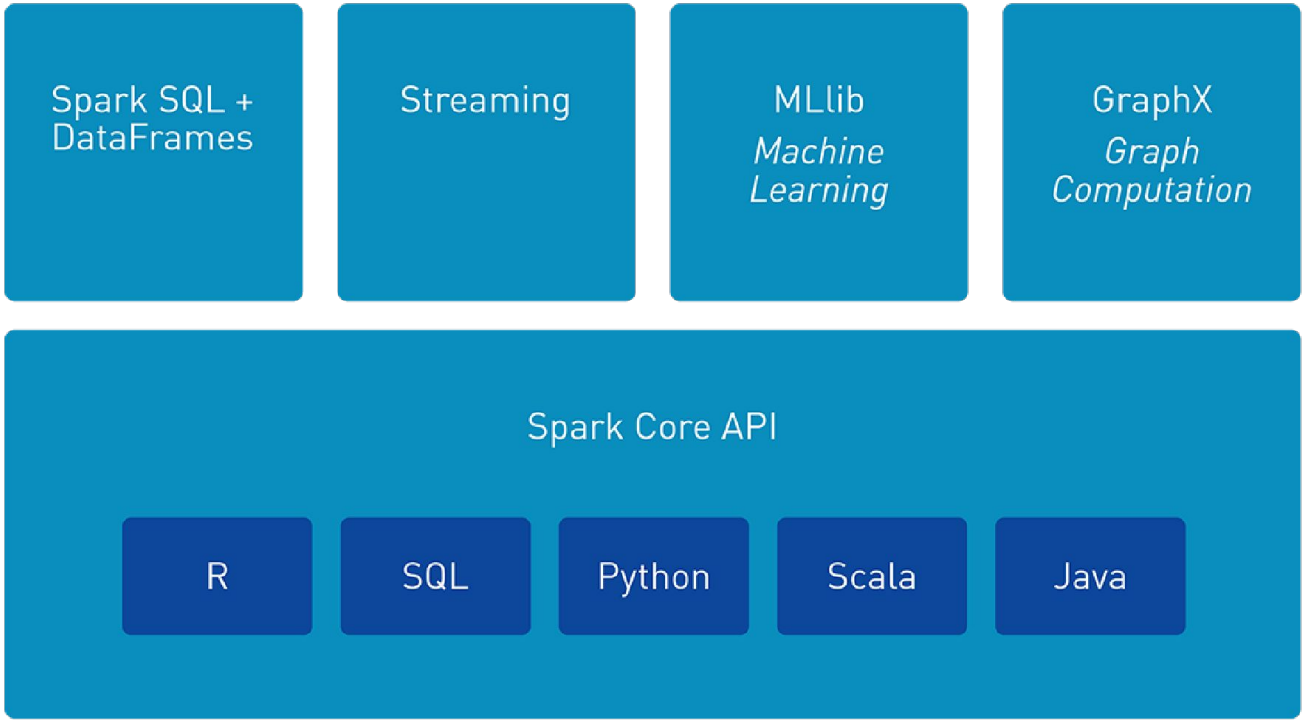




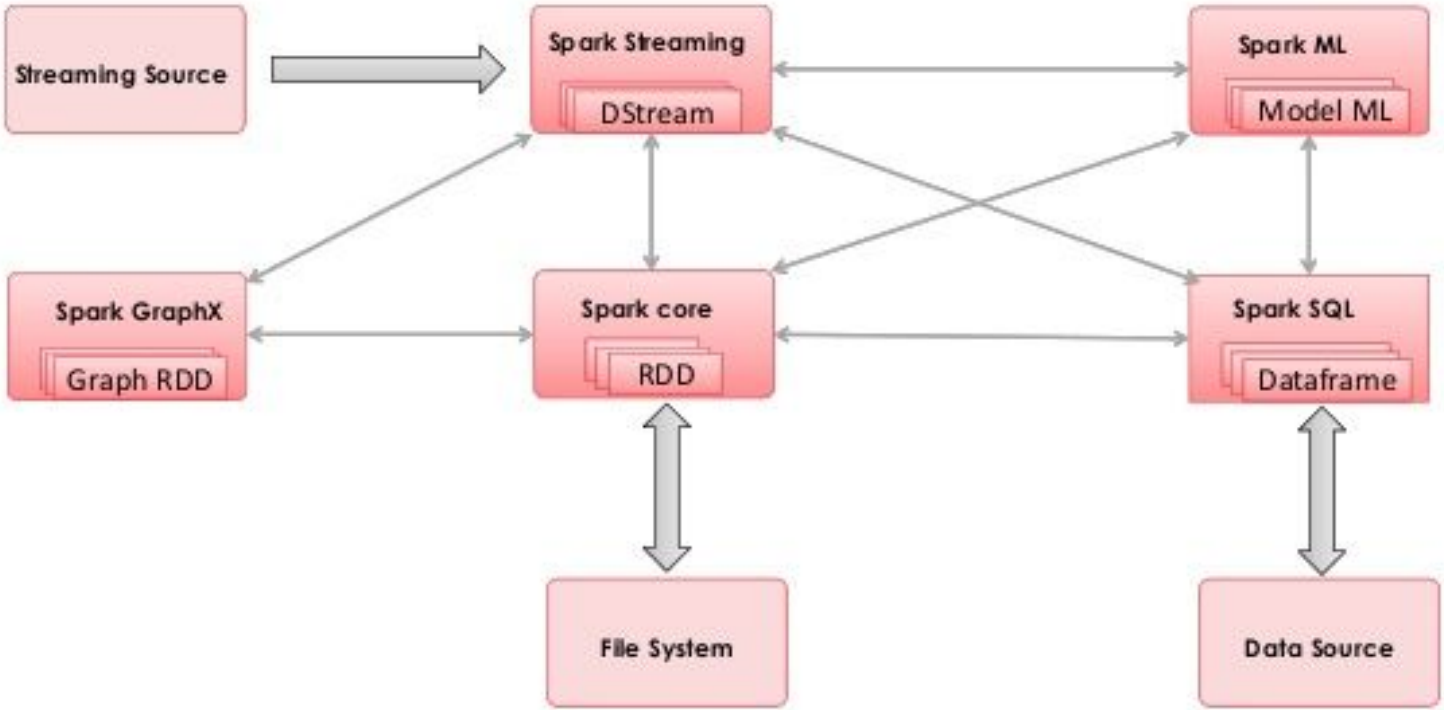
Factors	Spark	Hadoop MapReduce
Speed	100x times than MapReduce	Faster than traditional system
Written In	Scala	Java
Data Processing	Batch / real-time / iterative / interactive /graph	Batch processing
Ease of Use	Compact & easier than Hadoop	Complex & lengthy
Caching	Caches the data in-memory & enhances the system performance	Doesn't support caching of data



Spark Modules



Spark Modules Interactions

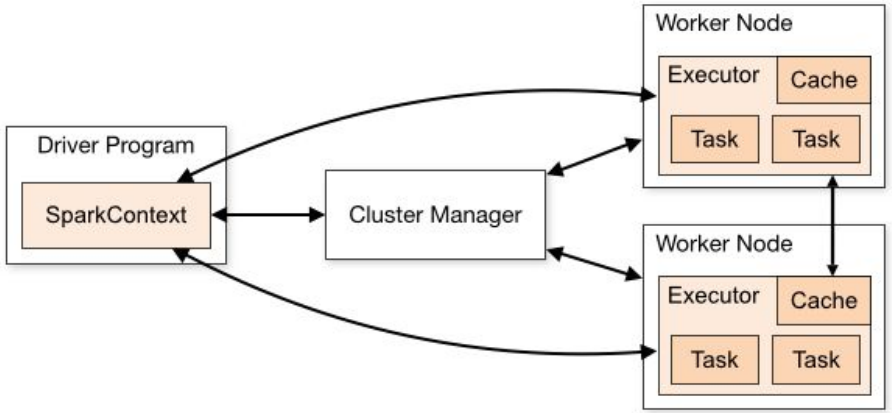


Advantages

- ★ Performance: In memory cluster computing platform
- ★ Code unified for both batch and streaming processing
- ★ Scalability
- ★ Fault tolerance
- ★ Easy to master and use (thanks to abstractions of RDD, Datasets, DStream, …)
- ★ High level Analyse Libraries specific to targeted fields (SparkSql, Mllib, GraphX)
- ★ Supports many languages (Scala, Java, Python)

Spark Distributed Computing

Spark Components



★ **Driver :** Program that starts the main function of a Spark application and manages the execution of parallel operations on executors

★ **Executor :** Launches tasks on a set of data on the driver's demand

Glossary

The following table summarizes terms you'll see used to refer to cluster concepts:

Application User program built on Spark. Consists of a driver program and executors on the cluster.

Application jar A jar containing the user's Spark application. In some cases users will want to create an "uber jar" containing their application along with its dependencies. The user's jar should never include Hadoop or Spark libraries, however, these will be added at runtime.

Driver program The process running the main() function of the application and creating the SparkContext

Cluster manager An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)

Deploy mode Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the cluster.

Worker node Any node that can run application code in the cluster

Executor A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.

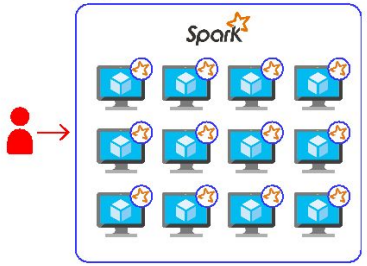
Task A unit of work that will be sent to one executor

Job A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save, collect); you'll see this term used in the driver's logs.

Stage Each job gets divided into smaller sets of tasks called stages that depend on each other (similar to the map and reduce stages in MapReduce); you'll see this term used in the driver's logs.

https://spark.apache.org/docs/latest/cluster-overview.html

Spark Cluster Managers

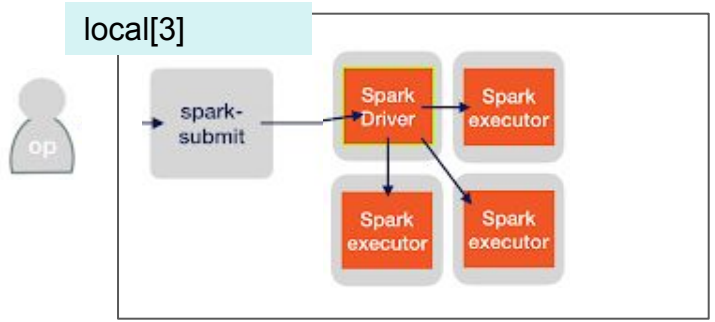


The system currently supports several cluster managers:

- Standalone – a simple cluster manager included with Spark that makes it easy to set up a cluster.
- Apache Mesos – a general cluster manager that can also run Hadoop MapReduce and service applications.
- Hadoop YARN – the resource manager in Hadoop 2.
- Kubernetes – an open-source system for automating deployment, scaling, and management of containerized applications.

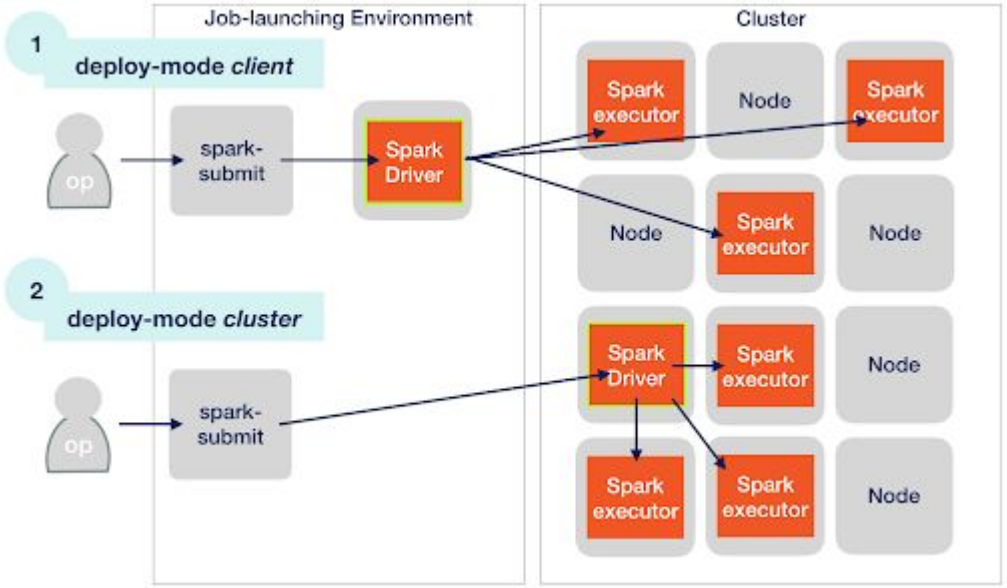
Spark Deploy Mode

In local mode, all run in the same client Machine. Each Executor is a separate Thread (therefore you // your job over CPUs of the machine

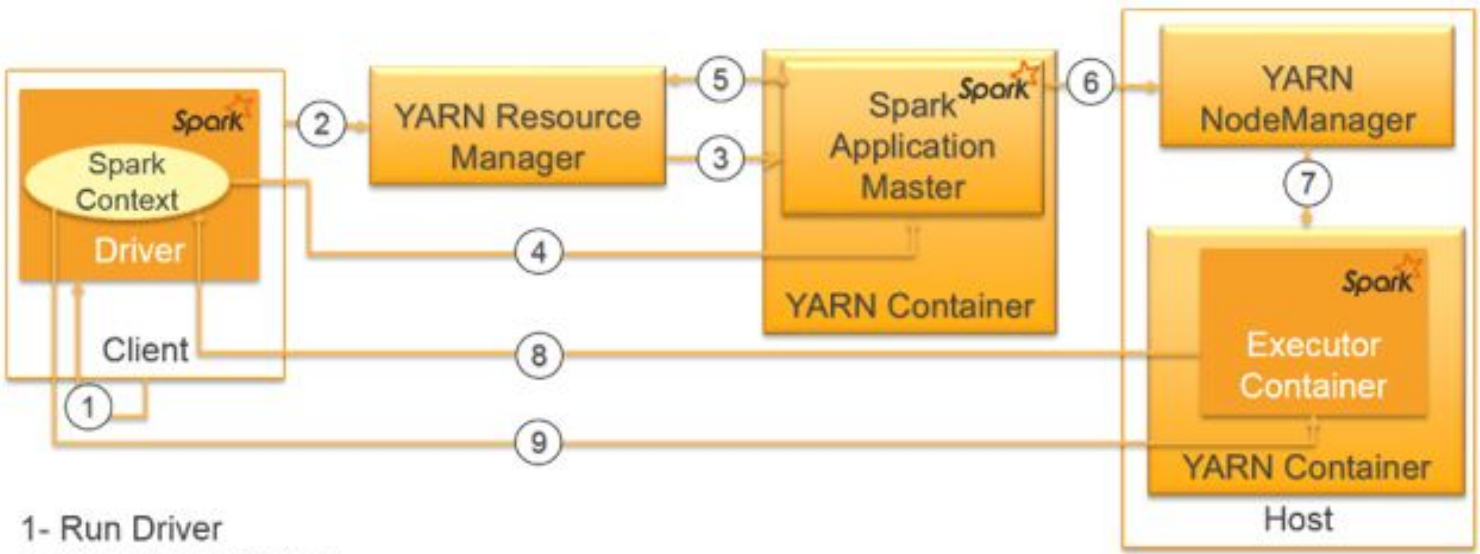


In Master client mode: Your Driver run on your machine. Executors runs on different machines of the cluster.

In Master client mode: Driver and Executors run on different machines of the cluster.



Spark With YARN



1- Run Driver
2- Submit Application
3- Launch Application Master
4,5- Request Resources
6- Launch Containers via YARN NodeManager
7- Launch Spark Executors
8- Register with the Driver
9- Launch Tasks

Spark Basics

Spark Shell

- ★ Spark Shell provides a powerful tool to analyze data interactively (Read/Evaluate/Print/Loop)

- ★ It is available in either Scala (which runs on the Java VM and is thus a good way to use existing Java libraries) or Python

- ★ Download last Spark Version:
  https://spark.apache.org/downloads.html
  Unzip it.
  In Your terminal, run:

```
cd /path/to/your/spark/folder
bin/spark-shell
```

Troubleshooting on Windows

If you had an error on Windows like this: “Illegal character in path at index”. This is because your filesystem have a bad character in your filesystem. If possible, the best is to clean your filesystem name from this character. Otherwise, You have 2 ways to resolve this:

Solution 1: Run in the UNIX shell

In your unix shell follow this link:
https://computingforgeeks.com/how-to-install-apache-spark-on-ubuntu-debian/

Solution 2: Run in Windows via spark master

Launch Powershell
Set SPARK_HOME variable to your Spark Home:

```
$env:SPARK_HOME = '/path/to/your/spark/home’
cd '/path/to/your/spark/home’
./spark-class org.apache.spark.deploy.master.Master
```

Open http://localhost:8080 and copy paste the URL of the Spark Master (it looks like spark://192.168.2.4:7077)
Launch another Powershell and launch these commands (replace X.X.X.X by spark master uri):

```
$env:SPARK_HOME = '/path/to/your/spark/home’
cd '/path/to/your/spark/home’
./spark-shell --master spark://X.X.X.X:7077
```

More here: https://stackoverflow.com/a/69804576/1029722

Read File

★ Every Spark application requires a Spark context, the main entry point to the Spark API

★ The Spark shell provides a preconfigured Spark context called sc

```
scala> sc.textFile("/path/to/sparkFolder/README.md").toDF.show()
```

Spark UI

★ By default, Spark launch Spark UI on your port 4040. If not avaible it will check 4041, then 4042…. The URL is displayed when you launch Spark-shell command line.

★ Find Your Spark UI Url and go there. By default: http://localhost:4040/jobs/

★ You can see you have already launch 1 job with the “show” action:

Action & Transformations

★ **Resilient Distributed Datasets** (RDD) and other distributed structures like **Datasets** and **DataFrames** support two types of operations that can be either :

- Transformations :
  - Spark Transformation is a function that produces new RDD from the existing RDDs. It takes RDD as input and produces one or more RDD as output
    - dataMulitplied2 = rdd.filter(x => x*2)
- Actions :
  - Actions are RDD operations that produce non-RDD values
  - They materialize a value in a Spark program. In other words, a RDD operation that returns a value of any type but RDD[T] is an action.
    - dataMulitplied2.saveFile(“/path/to/saved/file”)

Create a new RDD

```
# Read File from local filesystem and create an RDD
scala> val data = sc.textFile("README.md")

# Create an RDD through Parallelized Collection
scala>val numbers = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
scala>val numbersDataRDD = sc.parallelize(numbers)

# Create an RDD from existing RDD
scala> val newRDD = numbersDataRDD.map(data => (data * 2))
```

Transformations

★ Transformation only create new RDD:

```
# Create an RDD through Parallelized Collection
scala> val numbers = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
scala> val numbersDataRDD = sc.parallelize(numbers)
scala> val newRDD = numbersDataRDD.map(data => (data * 2))
scala> val newRDD = numbersDataRDD.filter(data => (data > 4))
```

It doesn't launch any job. Go on the UI and verify there no new job launched:
http://localhost:4040/

Actions

★ Actions launch a new Job. It actually launch all Transformations of the rdd and parallelize these as Tasks on the Executors.

```
scala>val numbers = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
scala>val numbersDataRDD = sc.parallelize(numbers)
scala> val newRDD = numbersDataRDD.map(data => (data * 2))
scala> val newRDD = numbersDataRDD.filter(data => (data > 4))
scala> newRDD.collect()
```

Or :

```
scala>val numbers = sc.parallelize((Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)).map(data => (data * 2)).filter(data => (data > 4)).collect()
```

★ Last line launch the Job. You can see it on the UI:

Spark job: Number of Stage ?

★ As we have seen on the UI, the Job was launched with 8 stages :

▾ Completed Jobs (2)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	collect at <console>:26 collect at <console>:26	2020/03/15 15:00:18	51 ms	1/1	8/8

★ Why did it launched 8 Stages ?

By default Spark-shell, launch with master[*]. That means it will // on all Cores of the machine. The machine on which we launched the Jobs had 8 cores, therefore it created the 8 Stages.

Spark Transformations

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).
mapPartitions(func)	Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
union(otherDataset)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(otherDataset)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
groupByKey([numPartitions])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numPartitions argument to set a different number of tasks.
reduceByKey(func, [numPartitions])	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. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

More: https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

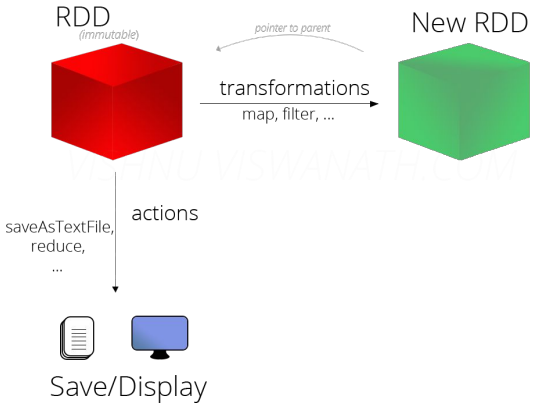
Spark Actions

Action	Meaning
reduce(func)	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
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.
takeSample(withReplacement, num, [seed])	Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(n, [ordering])	Return the first n elements of the RDD using either their natural order or a custom comparator.
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 or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
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. Note: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details.
distinct([numPartitions]))	Return a new dataset that contains the distinct elements of the source dataset.

More: https://spark.apache.org/docs/latest/rdd-programming-guide.html#actions

Spark Basics

RDD



★ **Resilient**, i.e. fault-tolerant with the help of RDD lineage graph and so able to recompute missing or damaged partitions due to node failures

★ **Distributed** with data residing on multiple nodes in a cluster

★ **Dataset** is a collection of partitioned data with primitive values or values of values, e.g. tuples or other objects (that represent records of the data you work with)

Create a new RDD

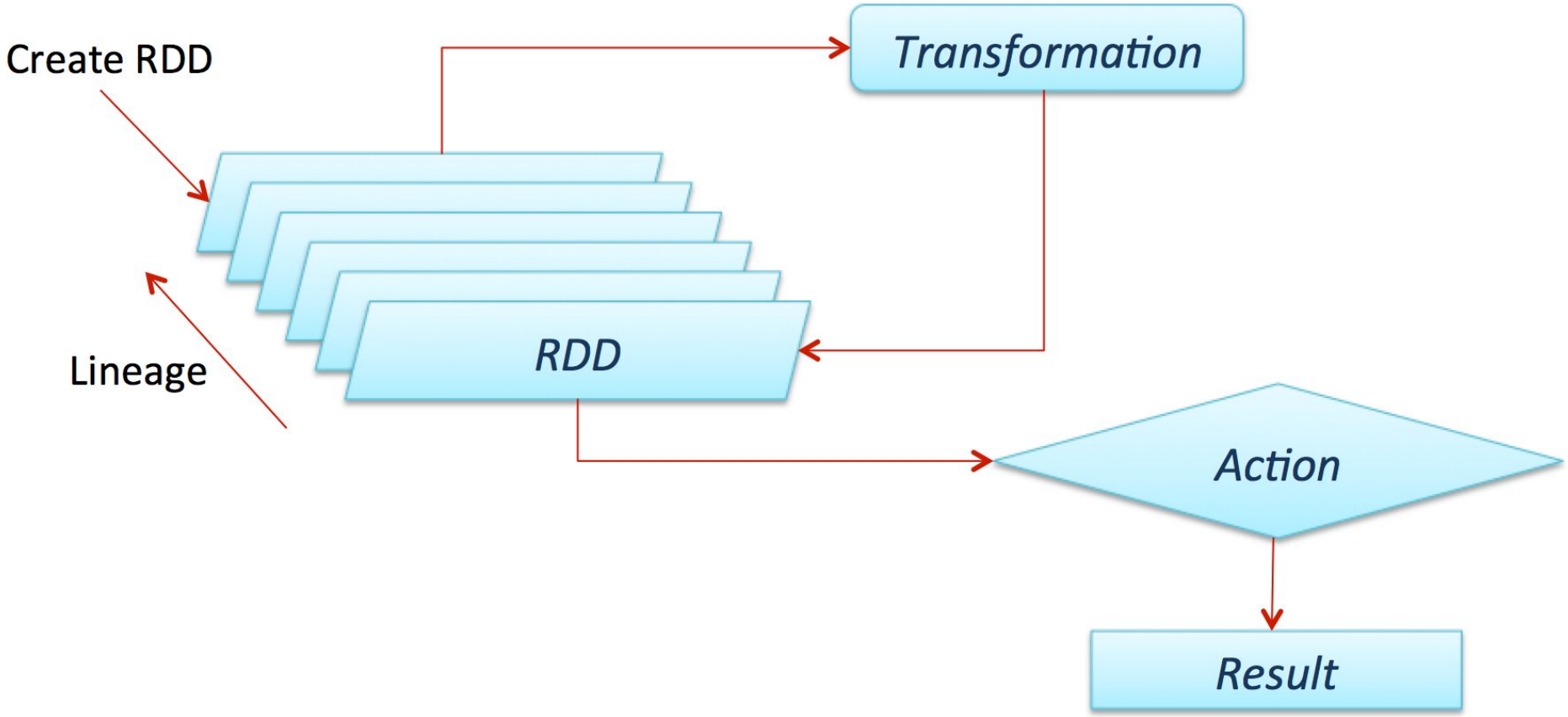
We had these following properties:

- ★ **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible
- ★ **Immutable** or **Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs
- ★ **Lazy evaluated,** i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution

Create a new RDD

- ★ Cacheable, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least prefered due to access speed)
- ★ Parallel, i.e. process data in parallel
- ★ Typed — RDD records have types, e.g. Long in RDD[Long] or (Int, String) in RDD[(Int, String)]
- ★ Partitioned — records are partitioned (split into logical partitions) and distributed across nodes in a cluster
- ★ Location-Stickiness — RDD can define placement preferences to compute partitions (as close to the records as possible)

Create a new RDD



RDD Pair

★ It’s possible to create a pair RDD, which associate a key with a value:

```
scala> val numbers = sc.parallelize(Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
scala> val pairs = numbers.map(x => (x, 1))
```

Hands-On 1

Hands-On 1

★ Write a code that will read all README and count the occurence of each word (number of time each word appear in the file). Display the Top 10 words.

Answer on next Slide

Hands-On 1 - Answer

★ Write a code that will read all README and count the occurence of each word (number of time each word appear in the file). Display the Top 10 words.

```
scala> sc.textFile("/path/to/spark/README.md")
.flatMap(line => line.split(" "))
.map(word => (word,1))
.reduceByKey((count1, count2)=> count1 + count2)
.map(item => item.swap)
.sortByKey(false)
.take(10)
```

Hands-On 1 - Explanations

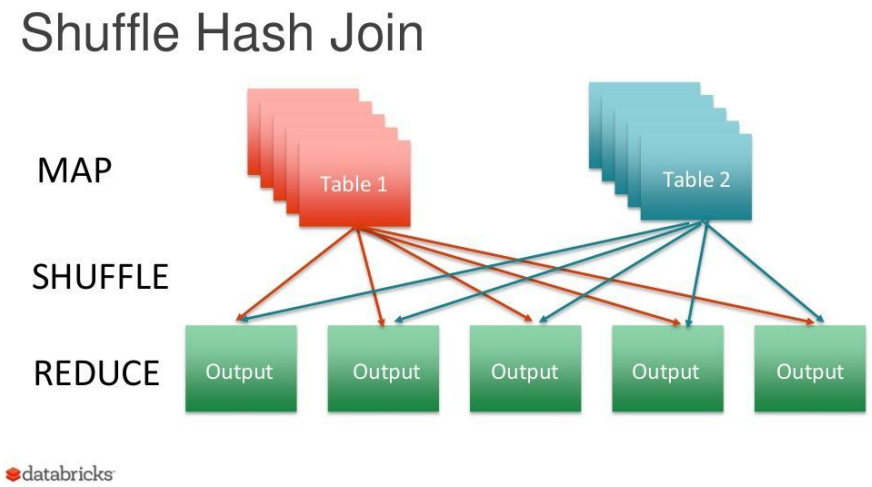
★ Write a code that will read all README and count the occurence of each word (number of time each word appear in the file). Display the Top 10 words.

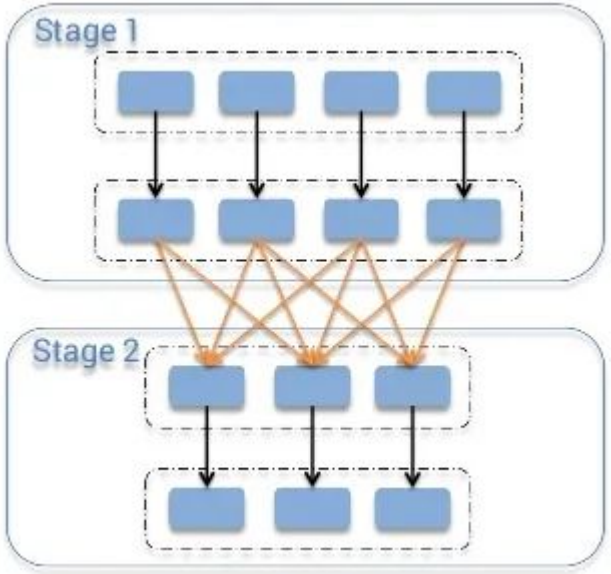
```
scala> sc.textFile("/path/to/spark/README.md")
.flatMap(line => line.split(" ")) #  split each line into words
.map(word => (word,1))  #  create a pair for each word, associate with value 1
.reduceByKey((count1, count2)=> count1 + count2) # compute the count for each word
.map(item => item.swap) # swap key and value (the count become the key)
.sortByKey(false)  # sort by the count
.take(10)   # The ACTION => take the 10 first elements
```

Spark
RDD lineage

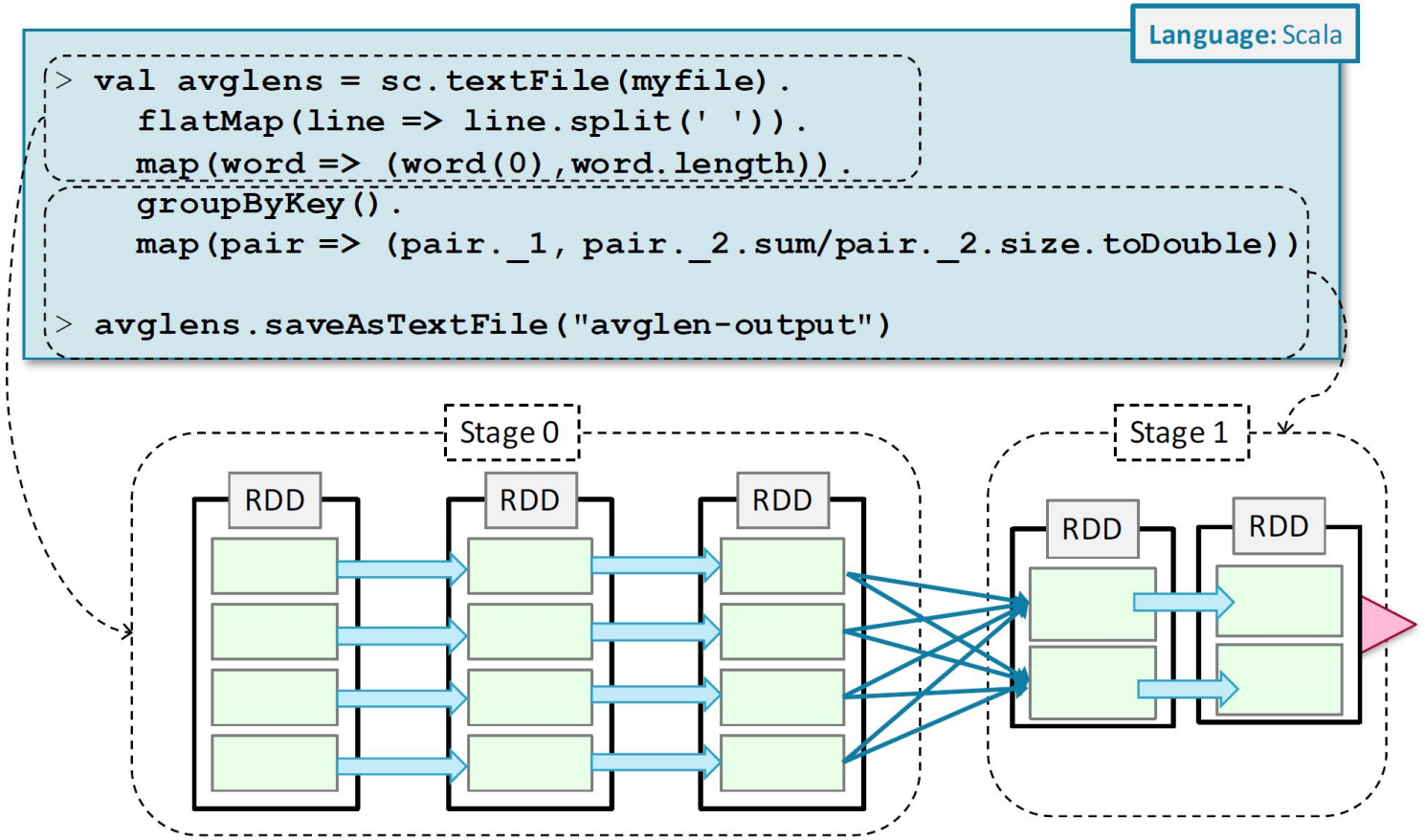
Shuffle

Certain operations within Spark trigger an event known as the shuffle. The shuffle is Spark's mechanism for redistributing data so that it's grouped differently across partitions. This typically involves copying data across executors and machines, making the shuffle a complex and costly operation.





Tasks and Stages

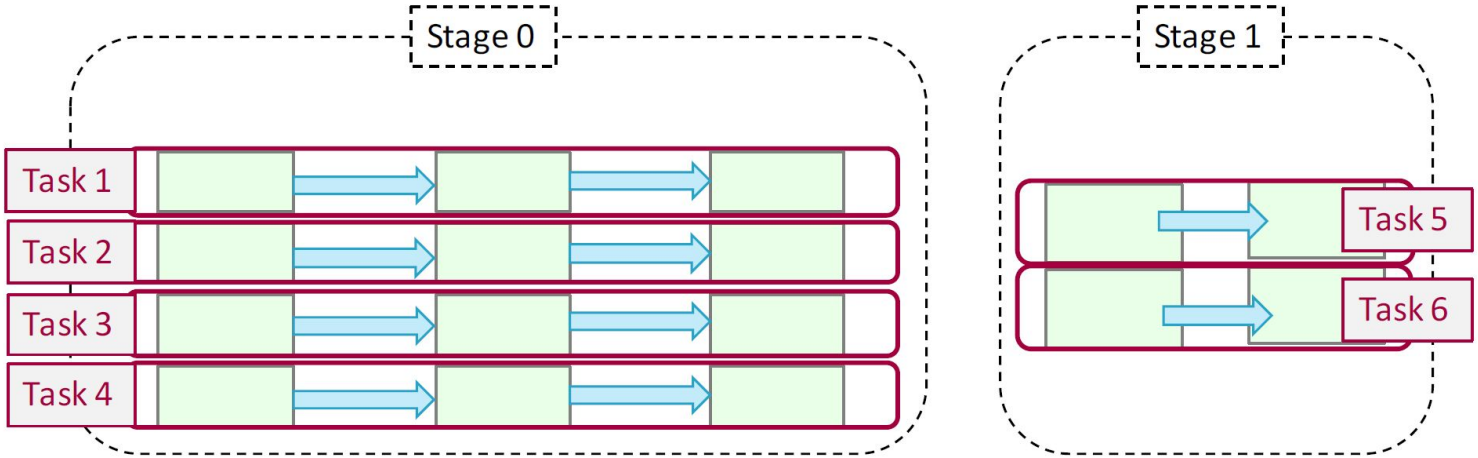


Tasks and Stages

Language: Scala

```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split(' ')).
    map(word => (word(0),word.length)).
    groupByKey().
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

> avglens.saveAsTextFile("avglen-output")
```

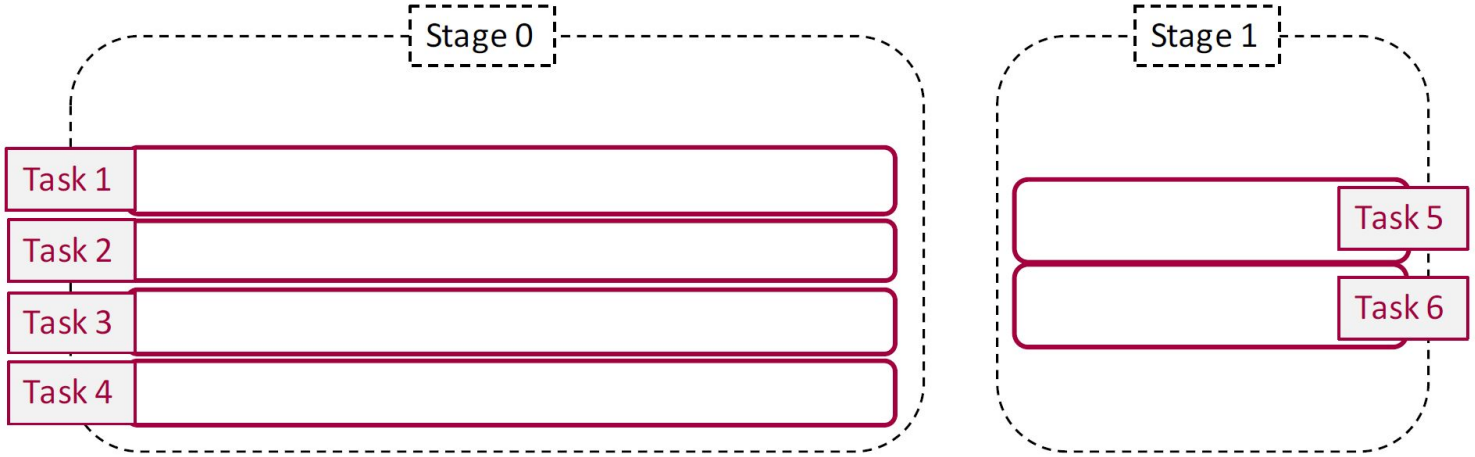


Tasks and Stages

Language: Scala

```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split(' ')).
    map(word => (word(0),word.length)).
    groupByKey().
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

> avglens.saveAsTextFile("avglen-output")
```

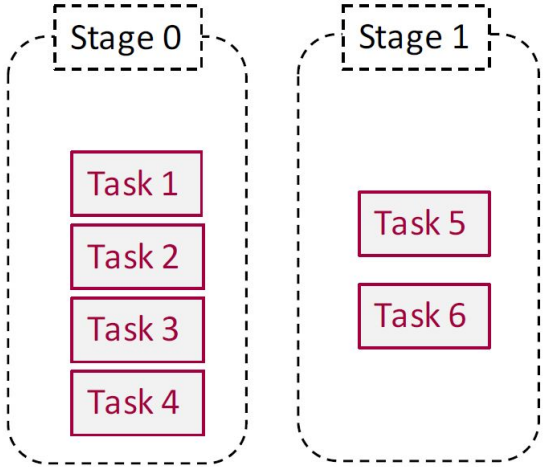


Tasks and Stages

Language: Scala

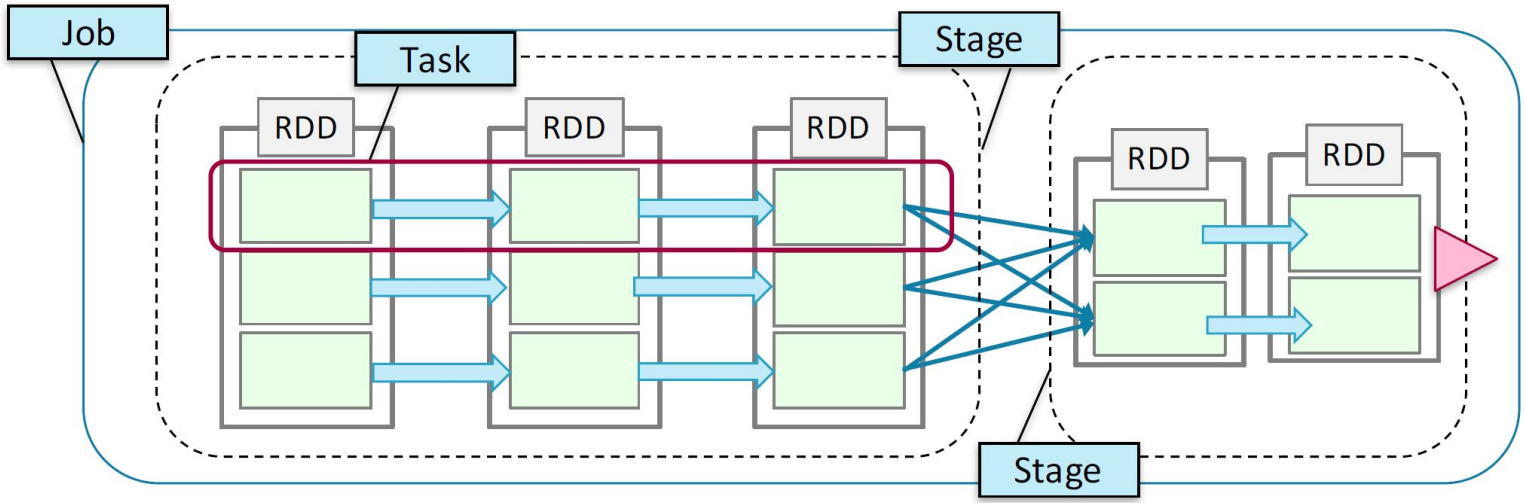
```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split(' ')).
    map(word => (word(0),word.length)).
    groupByKey().
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

> avglens.saveAsTextFile("avglen-output")
```



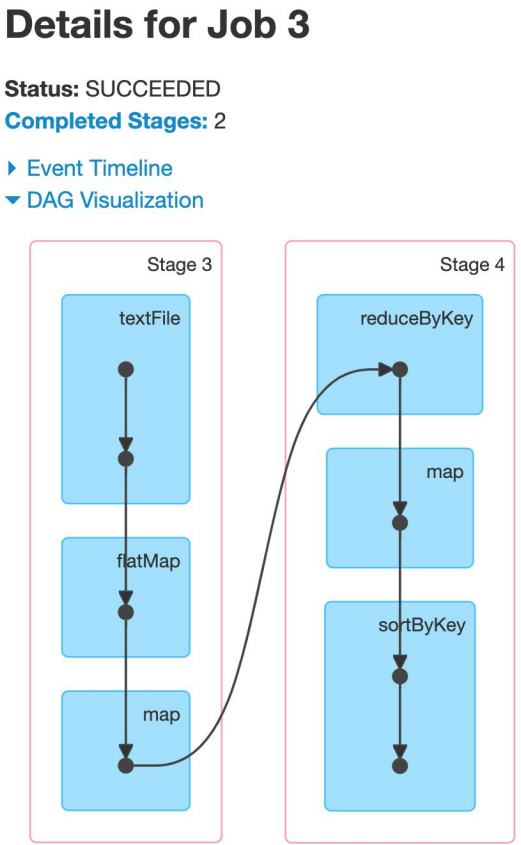
Tasks and Stages

- **Job**—a set of tasks executed as a result of an *action*
- **Stage**—a set of tasks in a job that can be executed in parallel
- **Task**—an individual unit of work sent to one executor
- **Application**—the set of jobs managed by a single driver



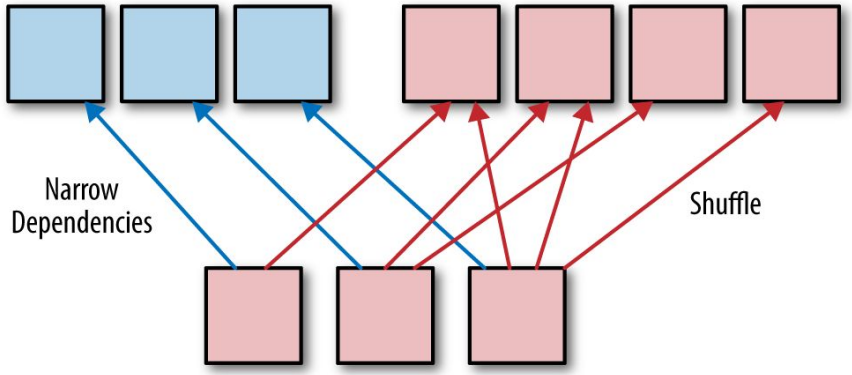
Tasks and Stages

- Go back in your spark, UI, you should see something like this:
- As you can see, only reduceByKey has generated a new Stage. This is because reduceByKey require Shuffle between Executors:



Narrow & Wide Depencies

rddA has a known partitioner
rddB does not

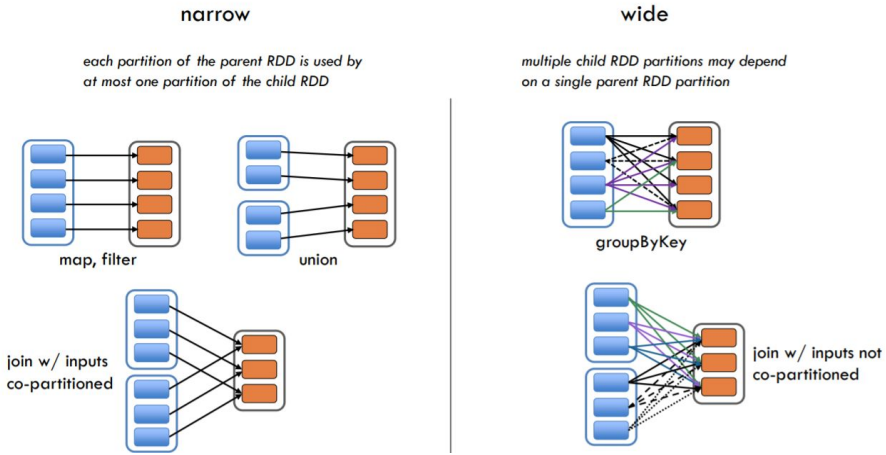


Narrow dependency doesn't imply that there is no network traffic.

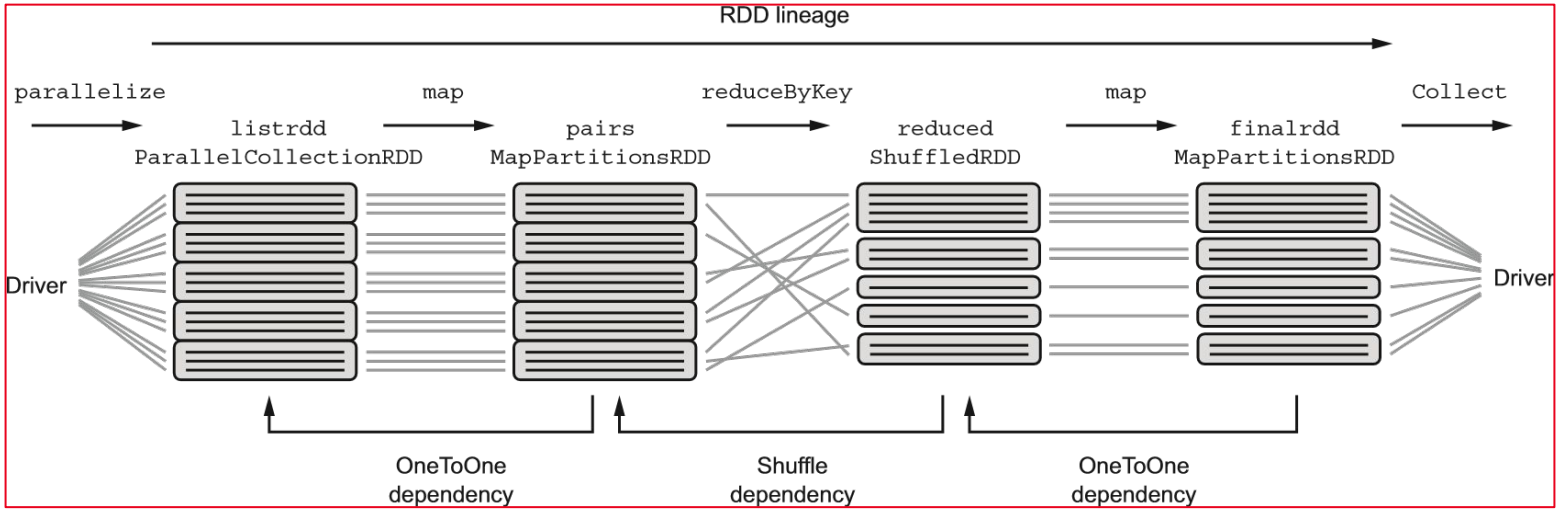
The distinction between narrow and wide is more subtle:

- ★ With wide dependency each child partition depends on each partition of its parents. It is many-to-many relationship.
- ★ With narrow dependency each child partition depends on at most one partition from each parent. It can be either one-to-one or many-to-one relationship.

If network traffic is required depends on other factors than transformation alone. For example co-partitioned RDDs can be joined without network traffic if shuffle happened during the same action (in this case there is both co-partitioning and co-location) or with network traffic otherwise.



Create a new RDD



Spark
Cache & Persistence

Cache & Persistence

What is the problem here ?

```
scala> val linesFiltered = sc.textFile("/path/spark/README.md").filter(l -> l.contains('bob'))
scala> linesFiltered.count()
scala> linesFiltered.saveAsTextFile("lines-with-bob.txt")
```

There is 2 actions, so 2 jobs will be triggered and the transformations before (filter), will be executed two times, which would be a waste of CPU.

We could save the value temporarly with caching (WARNING: Caching time could cost more than executing 2 times the transformation / CPU time).

```
scala> val linesCached = sc.textFile("/path/spark/README.md").filter(l -> l.contains('bob')).cache()
scala> linesCached.count()
scala> linesCached.saveAsTextFile("lines-with-bob.txt")
```

Cache & Persistence

★ **Cache** is a shortcut for **persit(level : StorageLevel)** method. With StorageLevel you can describe « How and Where »

★ Describes « How and Where » RDD is persisted :

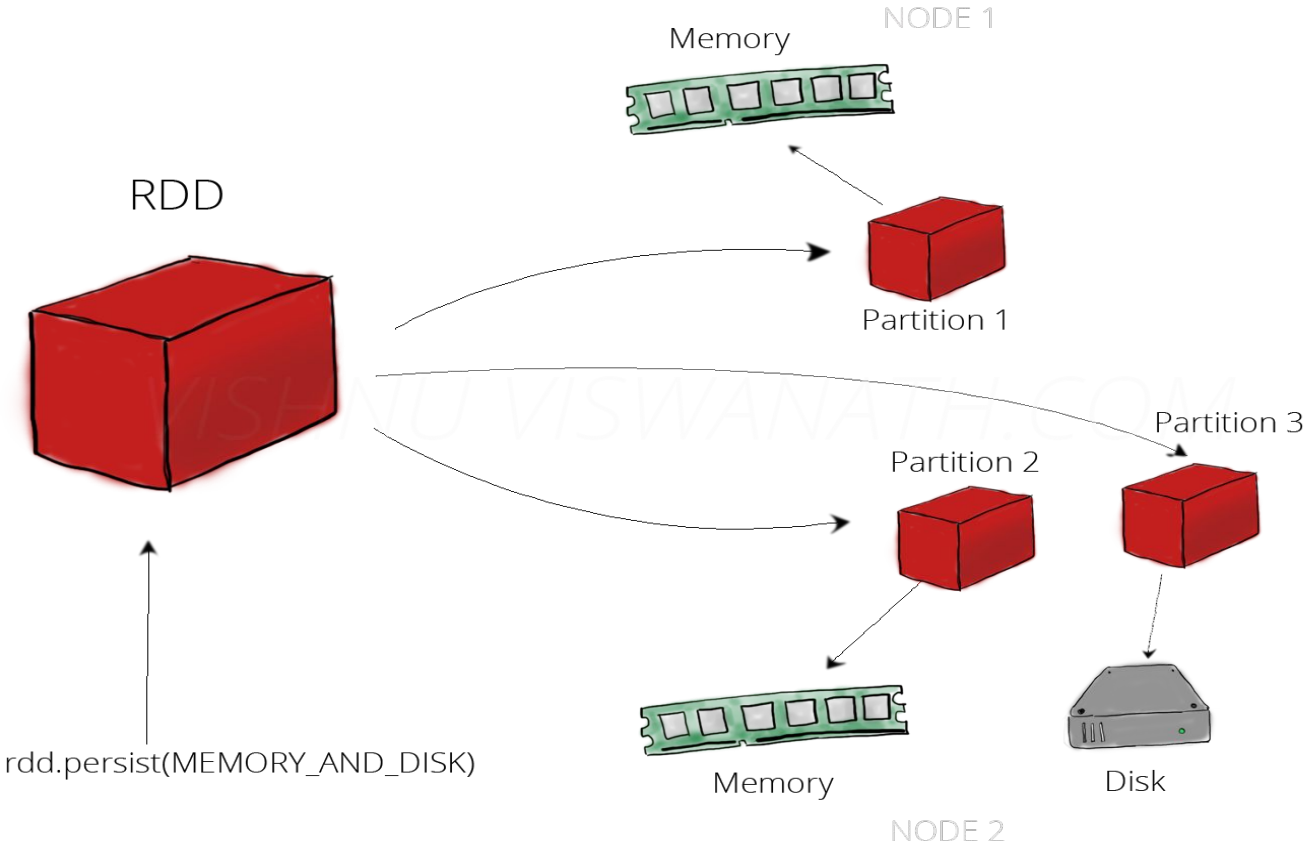
- Use Memory *and/or* Disk *and/or* OffHeap ? *(OffHead is experimental, and need to be enabled)*
- How much Replicas ? *(1 > replication > 40)*
- Should RDD be stored in deserialized format ? *(Java Objects or using configured Spark Serialization (like Kryo))*

Persistence allows to replicate results, so they can be used by another workers without re-computation

Cache & Persistence

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.
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.

Cache & Persistence



Partitions

Partition

- ★ A partition (aka *split*) is a logical chunk of a large distributed data set.
- ★ Spark manages data using partitions to help parallelize distributed data processing with minimal network traffic for sending data between executors.
- ★ Spark tries to read data into an RDD from the nodes that are close to it (*data locality* notion)
- ★ Features:
  - ○ size
  - ○ number
  - ○ partitioning scheme
  - ○ node distribution
  - ○ repartitioning

Partition

- ★ By default, a partition is created for each HDFS partition, which by default is 128MB
- ★ RDDs get partitioned automatically without programmer intervention
- ★ The number and size of partitions can be set according to the need of your application
- ★ Use *getPartitions*: Array[Partition] method on a RDD to know the set of partitions in this RDD

Partitions

Try these commands:

```
scala> sc.textFile("/Users/thibautdebroca/Downloads/spark-3.0.0-preview2-bin-hadoop2.7/README.md").partitions.size
res17: Int = 2

scala> sc.textFile("/Users/thibautdebroca/Downloads/spark-3.0.0-preview2-bin-hadoop2.7/README.md", 10).partitions.size
res21: Int = 10
```

In the second one, you see, we have manually specified the number of partitions in our RDD.
Test now:

```
scala> sc.textFile("/Users/thibautdebroca/Downloads/spark-3.0.0-preview2-bin-hadoop2.7/README.md", 15).filter(x =>
x.contains("the")).collect()
```

On the UI, you should see this:

Indeed the number of Tasks is based on the number of partitions, 1 task per partition

Partitions - Cons of using too few partitions ?

- ★ **Less concurrency** : you are not using advantages of parallelism. There could be worker nodes which are sitting idle.

- ★ **Improper resource utilization on data skewing :** your data might be skewed on one partition and hence your one worker might be doing more than other workers and hence resource issues might come at that worker.

Custom Partitioner

It’s possible to create your own partitioner:

- ★ **Hash Partitioner** : A Partitioner that implements hash-based partitioning using Java's Object.hashCode. https://stackoverflow.com/questions/31424396/how-does-hashpartitioner-work
- ★ **Range Partitioner:** Range partition algorithm divides the dataset into multiple partitions of consecutive and not overlapping ranges of values.

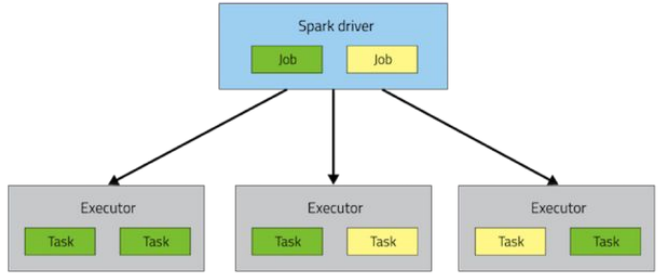
  https://www.waitingforcode.com/apache-spark-sql/range-partitioning-apache-spark-sql/read
- ★ **Custom Partitioner**: You can extend Partitioner class to create your own Partitioner.

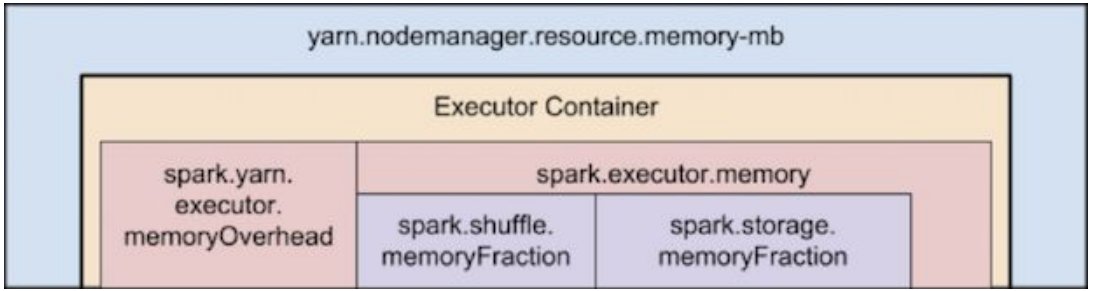
  https://labs.criteo.com/2018/06/spark-custom-partitioner/

Spark Job Resources

Job Resources

Configuration	Description	Default Value
spark.executor.instances (--num-executors)	The number of executors	2
spark.executor.cores (--executor-cores)	Number of CPU cores used by each executor	1
spark.executor.memory (--executor-memory)	Java heap size of each executor	512m
spark.yarn.executor.memoryOverhead	The amount of off-heap memory (in megabytes) to be allocated per executor	executorMemory * 0.07, with minimum of 384





Hands-on: Spark SQL

Hands-On

The goal here is to discover Spark SQL and use it to query some data.
Here is a great tutorial for that:
https://openclassrooms.com/fr/courses/4297166-realisez-des-calculs-distribues-sur-des-donnees-massives/4308676-mettez-spark-au-service-des-data-scientists