PySpark

Apache Spark Framework

Agenda

01 Spark Components

02 Spark Architecture

03 Spark Web UI

04 Introduction to PySpark Shell

05 What are Spark RDDs?

06 RDD Operations

07 RDD Lineage

08 RDD Persistence

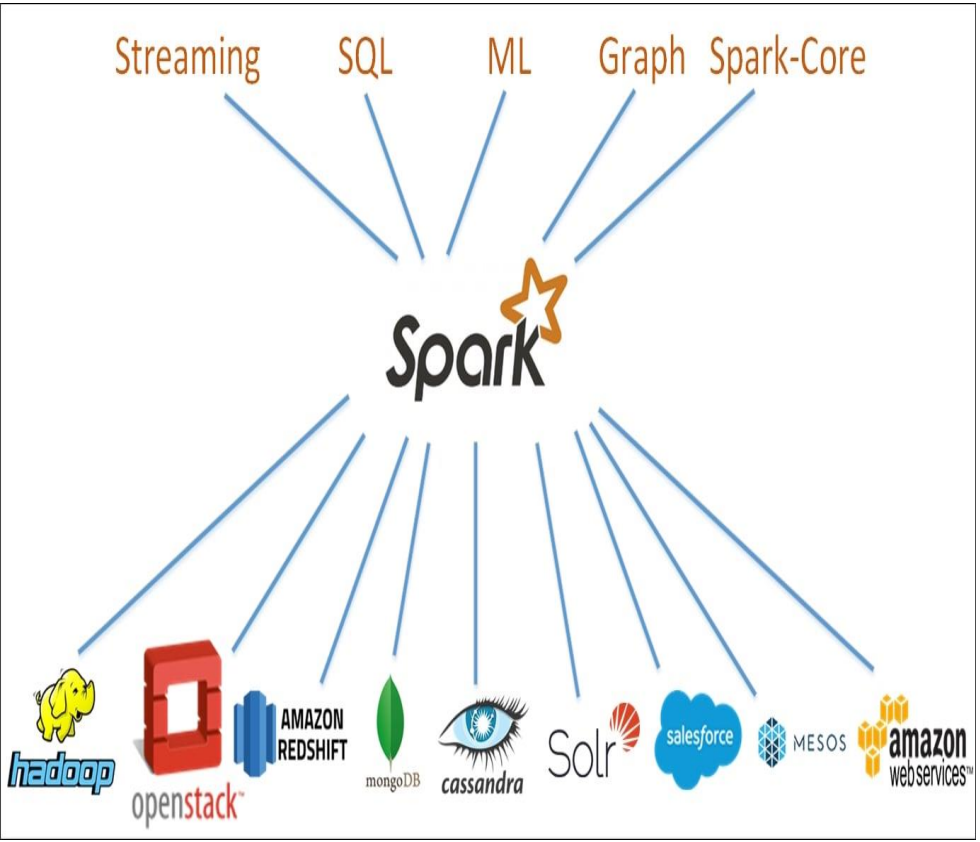
09 RDD Partitioning

10 Passing Functions to Spark

Spark Components

Spark Components

Spark Stack



- Spark Core

  The underlying engine

- Spark SQL

  For SQL and unstructured data processing

- Spark Streaming

  Stream processing of the live data streams

- MLlib

  Machine Learning algorithms

- GraphX

  Graph processing

Spark Core

- The base of the Spark engine
- Contains the basic functionalities of Spark, including task scheduling, memory management, fault recovery, etc.
- The main programming abstraction for Spark Core is RDD which is its basic data abstraction
- Spark Core provides many APIs for processing structured/unstructured data

Spark SQL

- Spark SQL provides integrated APIs to work with structured data
- It has a great set of APIs for people coming from a SQL background as it allows querying data via SQL commands
- It supports various data sources like CSV, JSON, Parquet, Hive, Cassandra, etc.

Spark Streaming

- Spark Streaming enables the processing of live streams of data
- It divides the live input data streams into batches
- For instance, logs being generated from an app server or live tweets from Twitter can be processed in near real time using Spark Streaming APIs

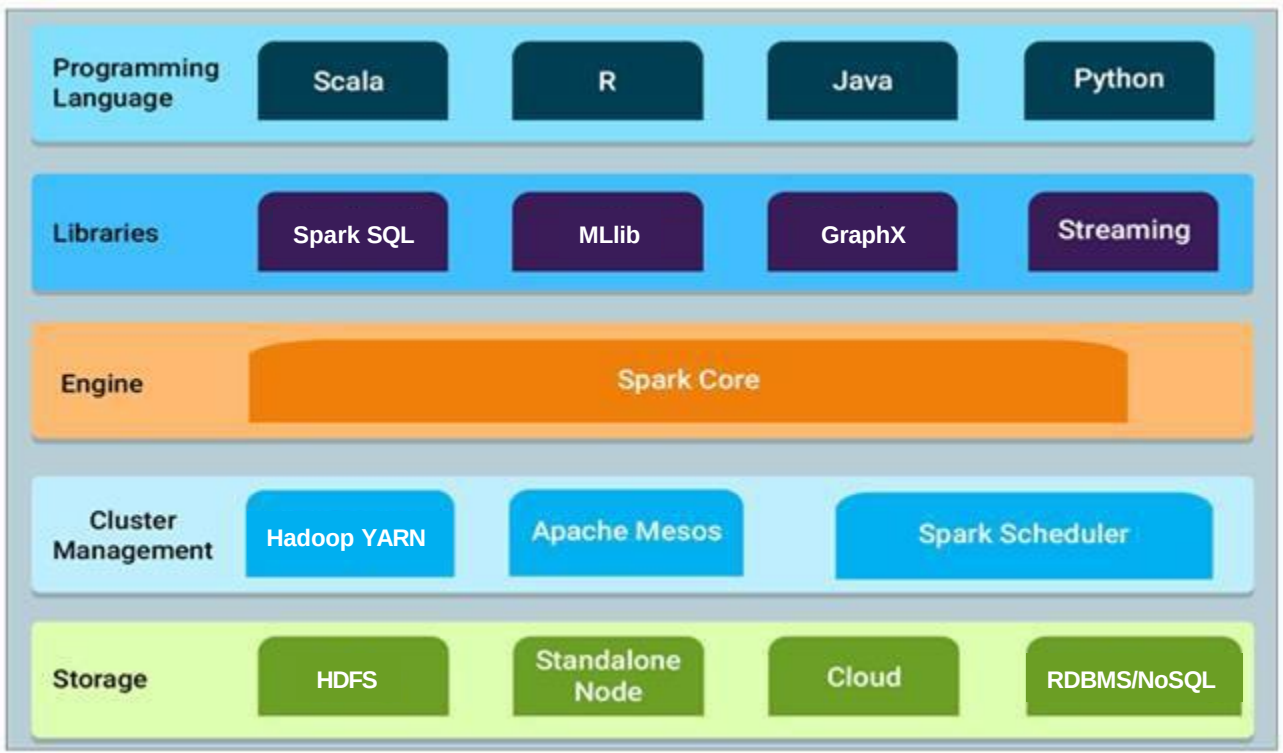
Spark MLlib

- Machine Learning and AI are significantly on hype these days
- Spark MLlib provides in-built libraries to implement Machine Learning algorithms
- These Machine Learning algorithms include classification, regression, clustering, collaborative filtering, etc.
- MLlib also provides various techniques for data preprocessing

Spark GraphX

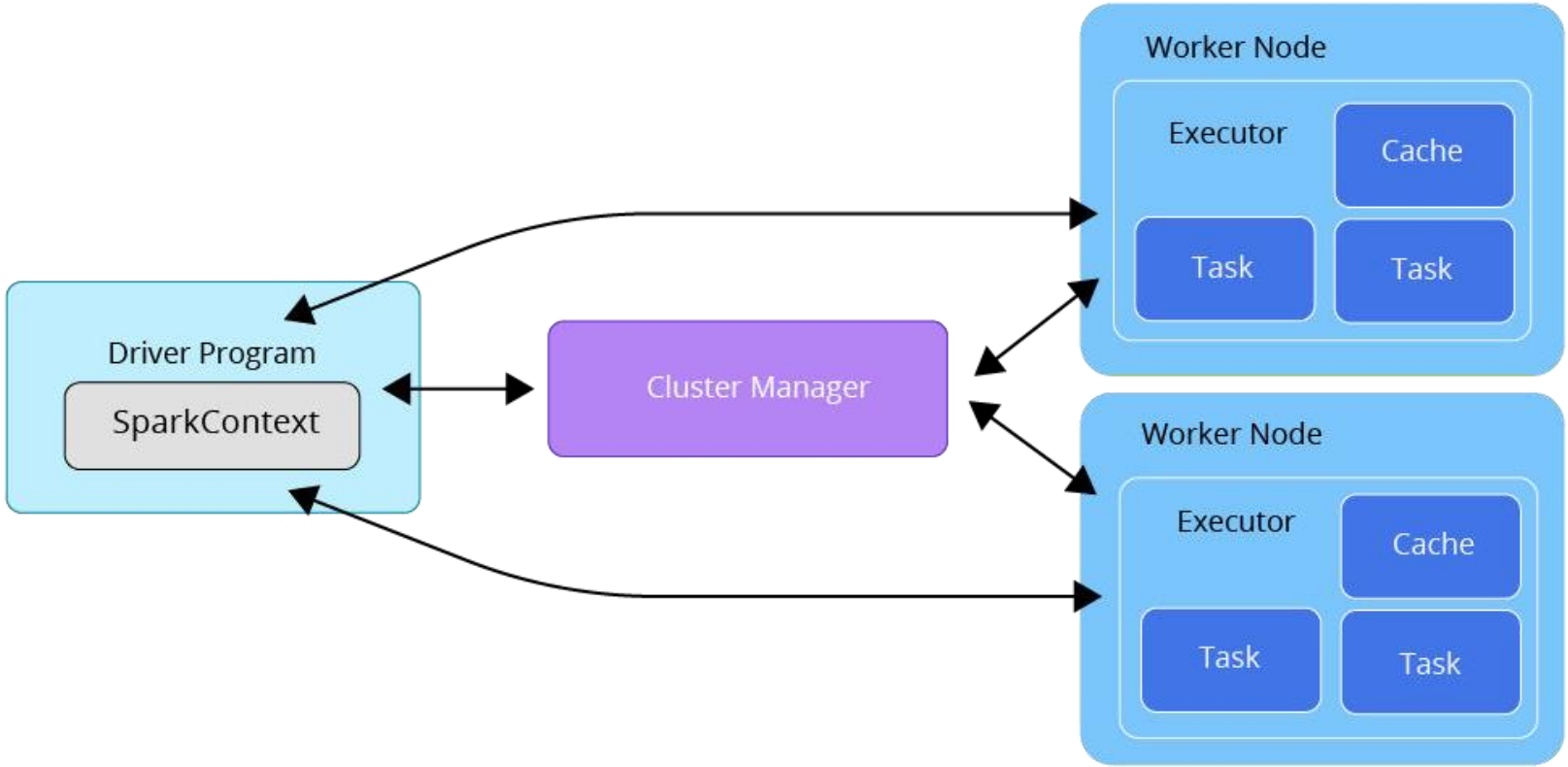
- GraphX is a library for manipulating graphs
- It provides analysis and graph computation for big data
- GraphX comes with a variety of graph algorithms and graph computations
- It can be used for the disaster detection system, page ranking, financial fraud detection, etc.

Categories of Spark Components

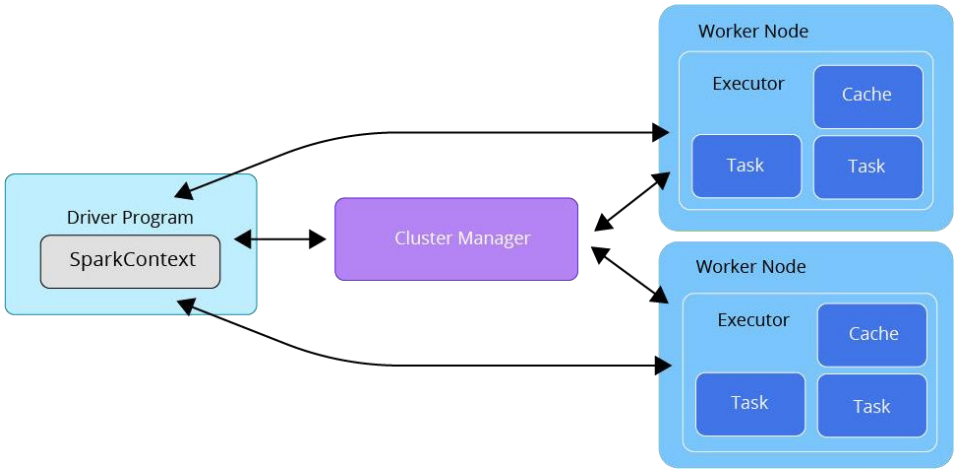


Spark Architecture

Spark Architecture

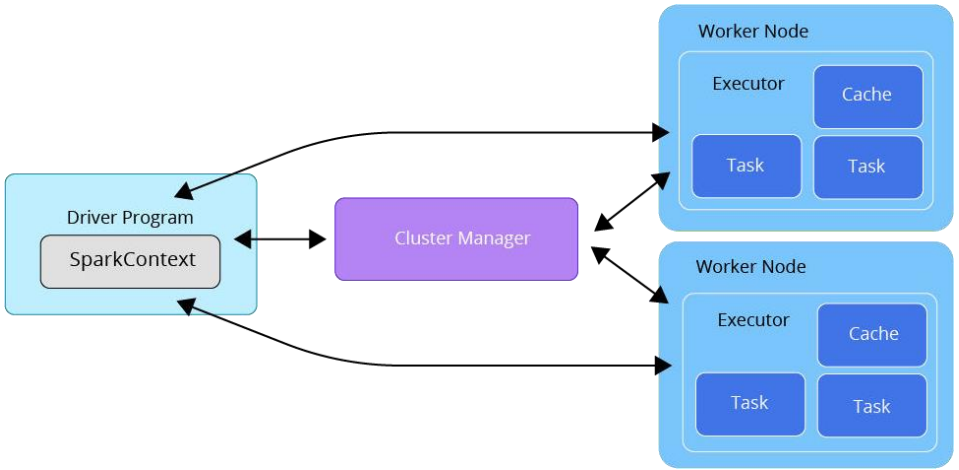


Spark Architecture



- Apache Spark provides a well-defined layered architecture
- All Spark components and layers are loosely coupled
- A driver program runs on the master node of the Spark cluster
- It schedules the job execution and negotiates with the cluster manager
- It also translates RDDs into the execution graphs
- The driver program can split graphs into multiple stages

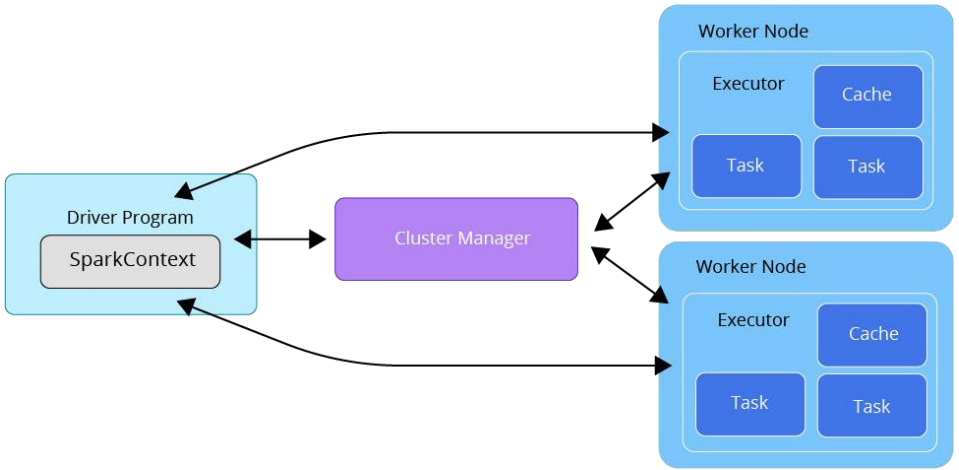
Spark Architecture



The Role of an Executor in Spark

- Executor is a distributed agent responsible for the execution of tasks
- Every Spark application has its own executor process
- An Executor performs all data processing
- It reads data from and writes data to external sources
- It interacts with storage systems

Spark Architecture



The Role of a Cluster Manager in Spark

- It is an external service responsible for acquiring resources on the Spark cluster and allocating them to a Spark job
- Choosing a cluster manager for any Spark application depends on the goals of the application
- The standalone cluster manager is the easiest one to use when developing a new Spark application

Spark
Deployment Modes

Spark Deployment Modes

- **Spark Standalone:** We can launch a standalone Spark cluster without a third-party cluster manager. This cluster can be used for deploying Spark applications

- **Spark on Apache Mesos:** We can deploy a private cluster using Apache Mesos. When using Mesos, the Mesos master replaces the Spark master as the cluster manager

- **Spark on Hadoop YARN:** We can deploy Spark on top of Hadoop YARN. Spark leveraged with Hadoop can enhance the processing capabilities of Spark

- **Amazon EC2:** We can launch a cluster on Amazon EC2 in about 5 minutes and it accelerates the speed with which Spark works

- **Kubernetes:** We can deploy Spark on top of Kubernetes. Spark can leverage the power of Kubernetes scheduler for faster processing

Running Spark Applications on YARN

- Spark is preconfigured for YARN, and it does not require any additional configuration to run
- YARN controls resource management, scheduling, and security when we run Spark applications on it
- It is possible to run an application in any mode, whether it is cluster mode or client mode

Launching a Spark Application in a Cluster/Client mode:

Cluster Mode

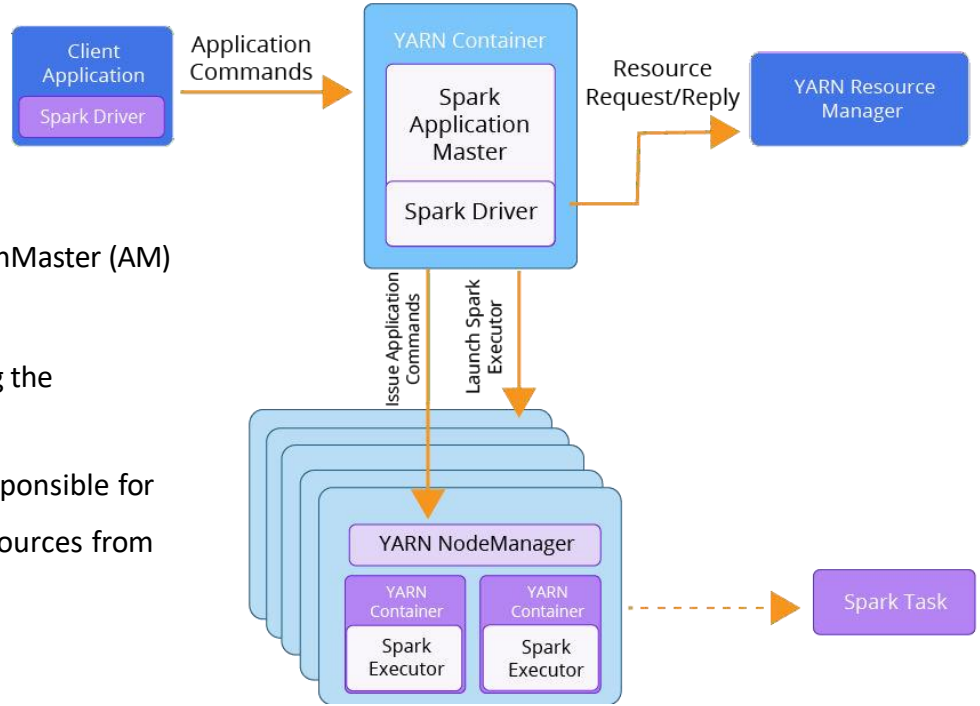
```
$ ./bin/spark-submit --class path.to.your.Class --master yarn
   --deploy-mode cluster [options] <app jar> [app options]
```

Client Mode

```
$ ./bin/spark-shell --master yarn --deploy-mode client
```

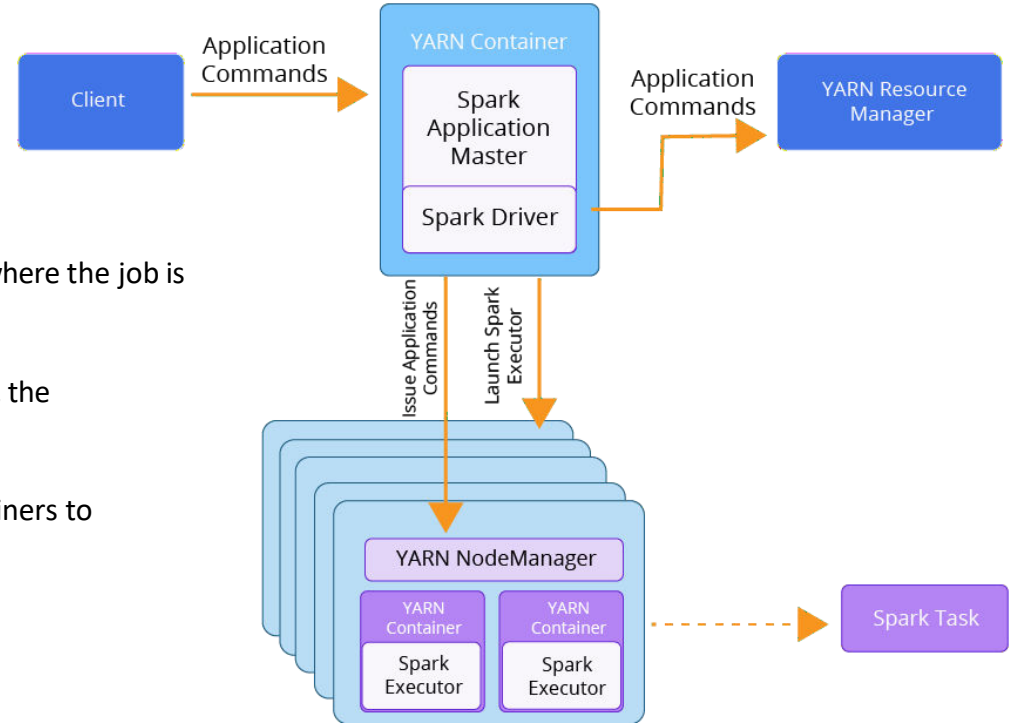
Cluster Deployment Mode

- The Spark driver runs inside an ApplicationMaster (AM) process, which is managed by YARN
- Thus, the client can go away after initiating the application
- A single process in a YARN container is responsible for driving the application and requesting resources from YARN



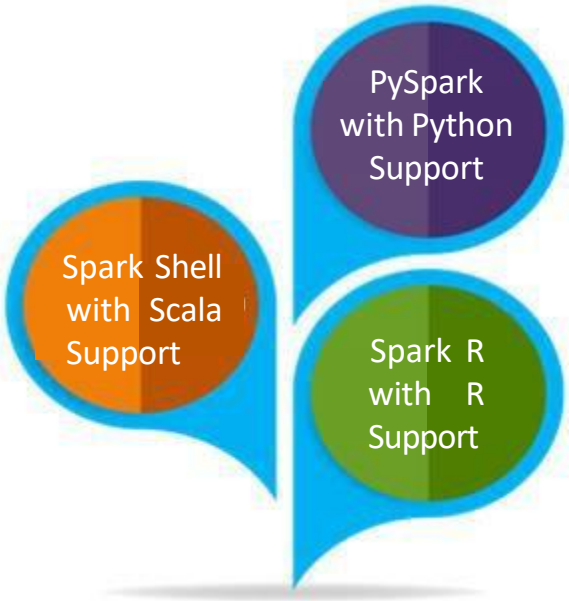
Client Deployment Mode

- In this mode, the driver runs on the host where the job is submitted
- To request executor containers from YARN, the ApplicationMaster is used
- The client communicates with those containers to schedule the work once they start



Spark Shell

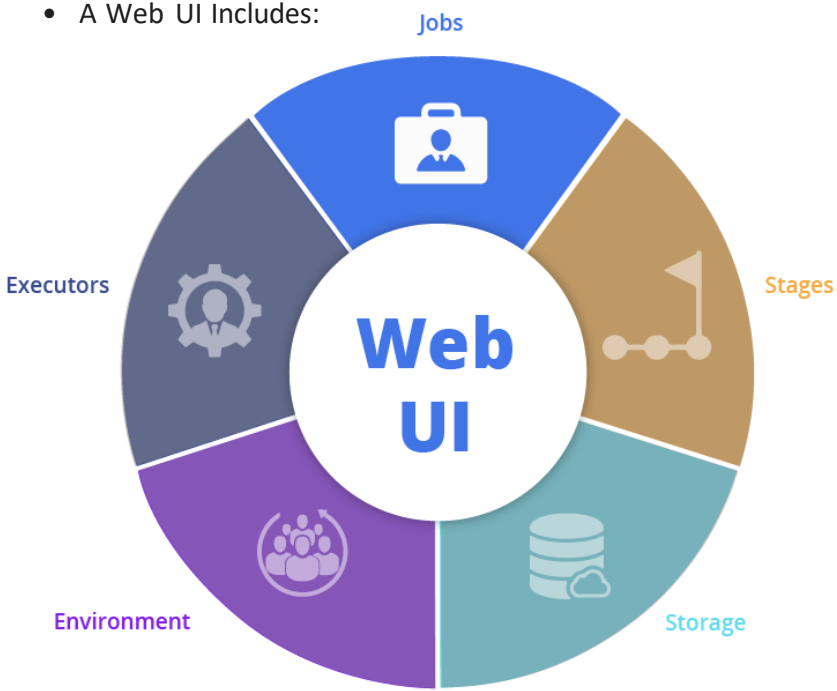
- Spark shell provides a simple way to learn the API, as well as a powerful tool to analyze data interactively
- Spark comes with three types of interactive shells



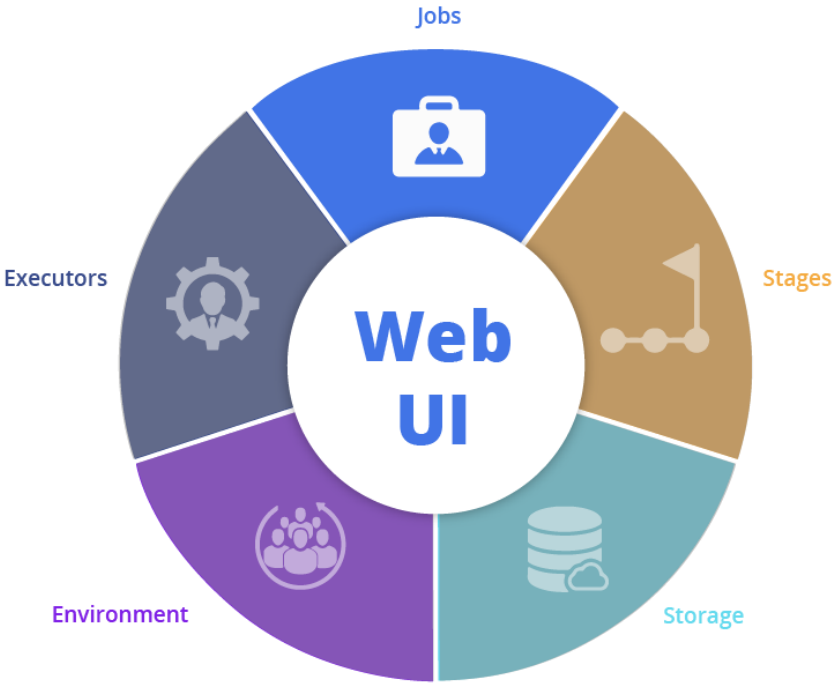
Spark Web UI

Spark Web UI

- Every SparkContext launches a web UI
- A Web UI Includes:



Spark Web UI: Jobs

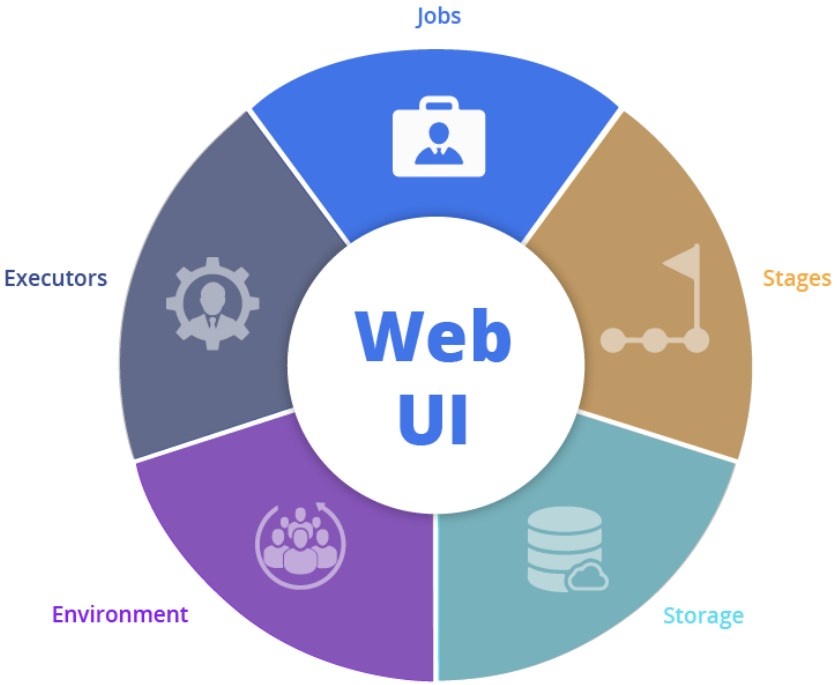


- The **Jobs** tab in a Web UI shows the status of all Spark jobs in a Spark application (i.e., a SparkContext)
- The Jobs tab consists of two pages:
  - All Jobs
  - Details for Job
- The tab displays stages per state, such as, active, pending, completed, skipped, or failed

Spark Web UI: Stages

- The **Stages** tab is created exclusively when a SparkUI is initialized
- The Stages tab in the Web UI shows the current state of all stages
- The Stages tab includes:
  - AllStagesPage
  - StagePage
  - PoolPage

Spark Web UI: Storage



- The Storage tab displays the information about RDDs
- The Summary page in the Storage tab shows the storage levels and partitions of all RDDs
- It also shows the sizes and the executors used for all partitions in an RDD
- The Storage tab includes the following pages:
    - StoragePage
    - RDDPage

Spark Web UI: Environment

- The Environment tab displays the values of different environment and configuration variables
- It shows Java, Spark, and system properties
- The Environment tab includes the following when created
  - Parent SparkUI
  - AppStatusStore

Spark Web UI: Executors

- The Executors tab displays the summary information about the executors that were created for the application
- It displays memory and disk usage and task and shuffle information
- The Storage Memory column in the Executors tab shows the amount of memory used and reserved for caching data

Introduction to **PySpark Shell**

PySpark Shell

- PySpark shell exposes the Spark programming model to Python
- PySpark shell links the Python API to the Spark Core and initializes the SparkContext
- Majority of Data Scientists and Analytics Experts today use Python for its rich library set
- Thus, integrating Python with Spark is a boon
- PySpark requires Python to be available on the system PATH and uses it to run programs

PySpark Shell

Working on the interactive PySpark shell:

```
Using Span‹’s defaul: log4j profile: ong;apa:he/span‹,'log4j-defaul:s.pnopen:ies
Se::ing defaul: log level :o ”v:CRM”.
 o adjus- logging level use s:.se-LogLevel‹nel,Level,. Fon Span‹R, use se-LogLevel{nel Level,.
Kel:ome -o



            *, , / / /  *, ',   version 1.4.3


Using P;zhon version 3.7.4 ‹/zags/v3.7.4:e093s9112e, Jul  8 2019 19:29:22}
Spar<Session available as ’spare’.
>>> PySpar< = ”Apache Spare tizh P;zhon”
>>> PySpar<
’Mpa he Spam uizh Python’
```

Submitting a **PySpark Job**

Submitting a PySpark Job

PySpark

- Many times our code depends on other projects or source codes, So we need to package them alongside our application to distribute the code to a Spark cluster
- For this, we create an assembly jar containing our code and its dependencies
- Both SBT and Maven have assembly plugins
- When creating assembly jars, list Spark and Hadoop as provided dependencies
- Once we have an assembled jar, we can call the bin/spark-submit script while passing our jar
- The spark-submit script is used to launch applications on the cluster

Submitting a PySpark Job

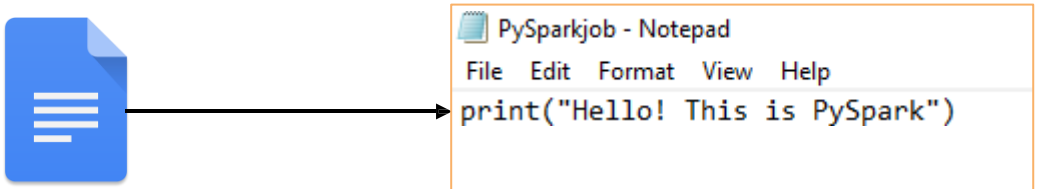
- Spark is preconfigured for YARN, and it does not require any additional configuration to run
- YARN controls resource management, scheduling, and security when we run Spark applications on it
- It is possible to run an application in any mode, whether it is cluster mode or client mode

Launching a Spark application in the Cluster/Client mode:

```
$ ./bin/spark-submit \
 --class org.apache.spark.examples.SparkPi \
 --master yarn \
 --deploy-mode cluster \
 --driver-memory 4g \
 --executor-memory 2g \
 --executor-cores 1 \
 --queue thequeue \
 examples/jars/spark-examples*.jar \
 10
```

Submitting a PySpark Job

The following steps will help us build our first application in Spark using **Python**:



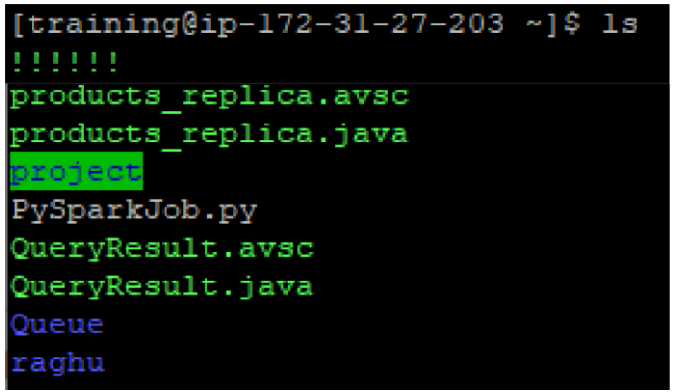
Create a document named **PySparkJob.py** on the local drive

PySparkJob.py

The Source Code

Submitting a PySpark Job

We can upload the PySparkJob.py file on our CloudLab's local storage (if we are not using aVM) and perform the following steps for CloudLab (or VM)



Submitting a PySpark Job

Submitting a Spark job using the YARN cluster:

```
$spark-submit –master yarn –deploy
client PySparkJob.py
```

```
[training@ip-172-31-27-203 ~]$ spark-submit --master yarn --deploy-mode client P
ySparkJob.py
2019-09-17 08:12:32 WARN  NativeCodeLoader:62 - Unable to load native-hadoop lib
rary for your platform... using builtin-java classes where applicable
Hello! This is PySpark
2019-09-17 08:12:33 INFO  ShutdownHookManager:54 - Shutdown hook called
2019-09-17 08:12:33 INFO  ShutdownHookManager:54 - Deleting directory /tmp/spark
-bc139572-7ec6-40ab-9880-18ab3b9fa271
[training@ip-172-31-27-203 ~]$
```

PySpark Job Using Jupyter Notebooks

The First PySpark Job

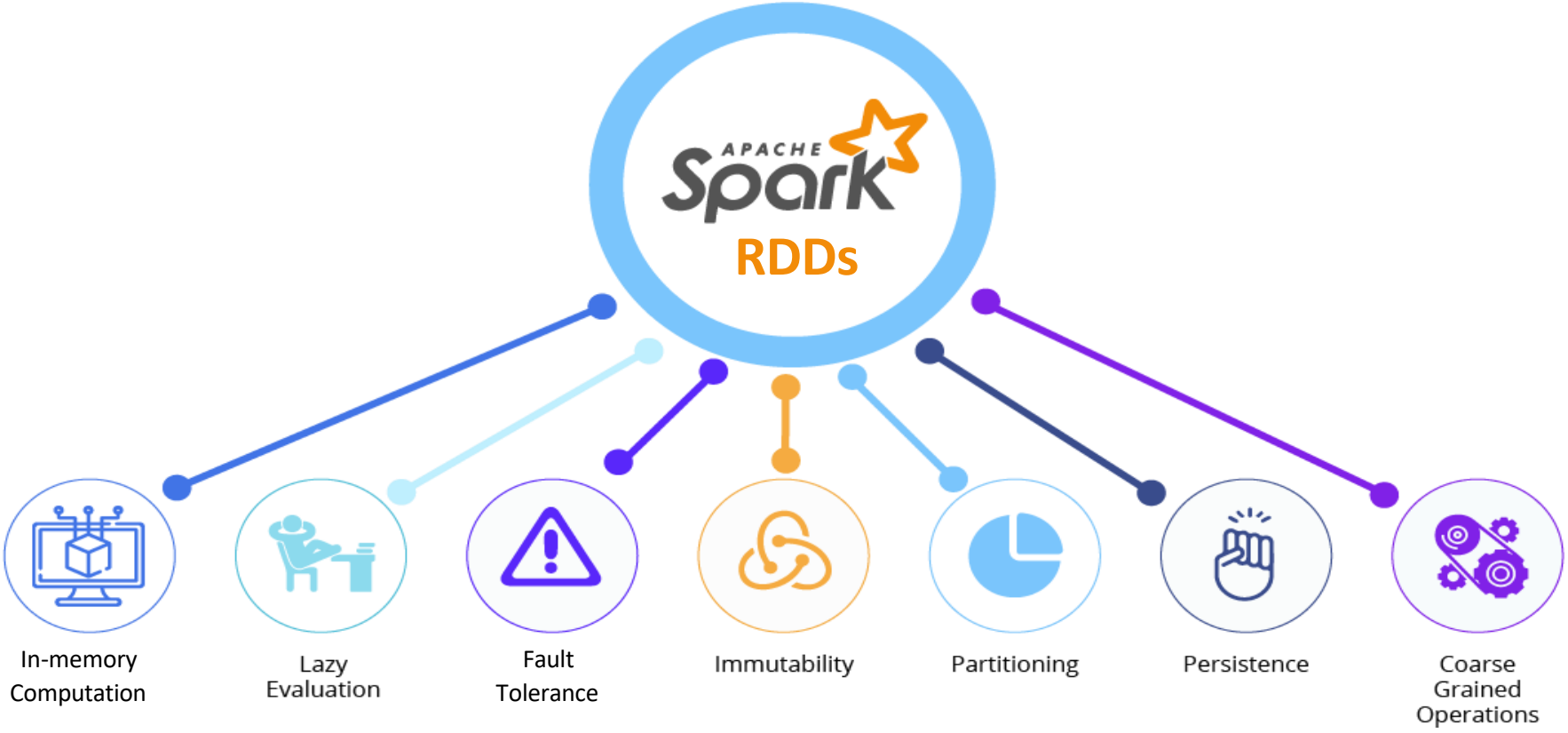
- Spark can easily run a Spark standalone job
- This example runs a minimal Spark script
- It includes:
  - Importing PySpark
  - Initializing a SparkContext
  - Performing a distributed calculation on a Spark cluster in the standalone mode
  - Defining a method ‘mod’ to perform an action in Spark Job
  - Using an RDD operation to perform a transformation

```
from pyspark import SparkConf
from pyspark import SparkContext


conf = SparkConf()
conf.setMaster('local')
conf.setAppName('spark-basic')
sc = SparkContext(conf=conf)


def mod(x):
    import numpy as np
    return (x, np.mod(x, 2))


rdd = sc.parallelize(range(1000)).map(mod).take(10)
print(rdd)
```

```
(1, 1), (2, 0), (3, 1), (4, 0), (5, 1), (6, 0), (7, 1), (8, 0), (9, 1)]
```

What are **Spark RDDs**?

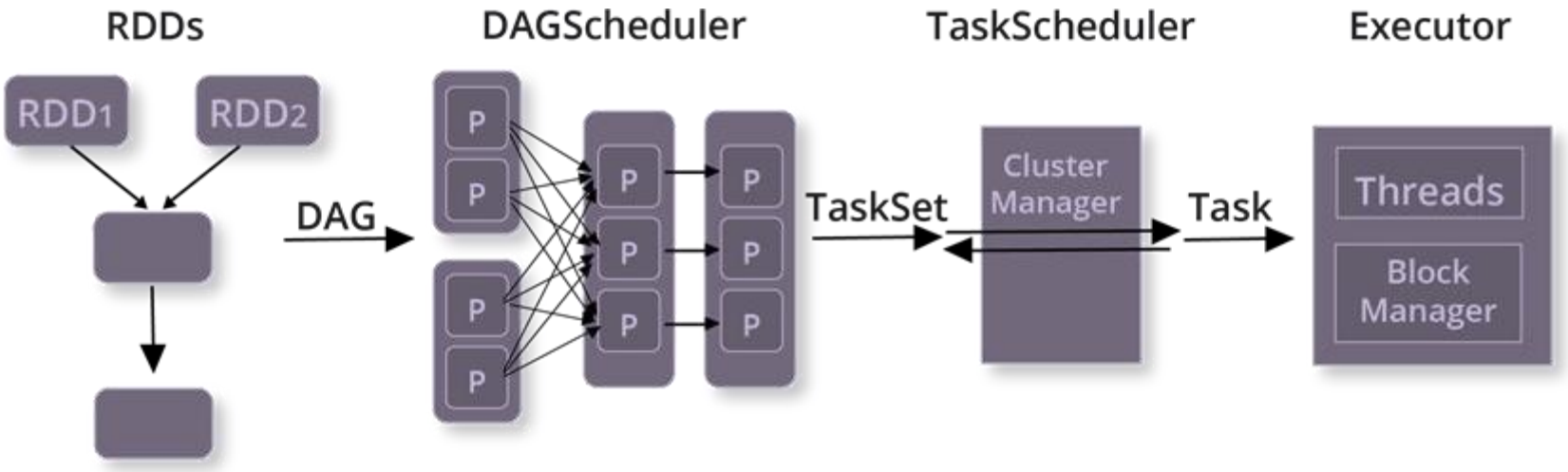
Spark RDDs


APACHE Spark
RDDs
In-memory Computation
Lazy Evaluation
Fault Tolerance
Immutability
Partitioning
Persistence
Coarse Grained Operations

Spark RDDs

- RDDs are the main logical data unit in Spark
- They are a distributed collection of objects, stored in memory or on disks of different machines of a cluster
- A single RDD can be divided into multiple logical partitions so that these partitions can be stored and processed on different machines of a cluster
- RDDs in Spark can be cached and used again for future transformations
- They are lazily evaluated, i.e., Spark delays the RDD evaluation until it is really needed
- We can perform transformations and actions on RDDs

RDD Workflow



Stopgaps in the Existing Computing Methodologies

The Existing Technologies

In-disk Computation	Before Spark RDDs, large volumes of data were processed using in-disk computation, which is slower than in-memory computation
	In-disk computation uses memory capacity to process stored data
Job Execution	The existing distributed computing systems (viz., MapReduce) need to store data in some intermediate stable distributed store, namely, HDFS
	It makes the overall computations of jobs slower as it involves multiple IO operations, replications, and serializations in the process
Parallel Processing	Besides Spark, most of the data processing frameworks are less efficient to perform parallel processing

How do **RDDs** solve the problem?

How do RDDs solve the problem?

- Spark RDDs have various capabilities to handle huge volumes of data by processing them parallelly over multiple logical partitions. We can create an RDD, anytime, but Spark RDDs are executed only when they are needed (lazy evaluation)
- Here are some more facts that explain why RDDs are better than the existing frameworks:

Enhanced Distributed Computing

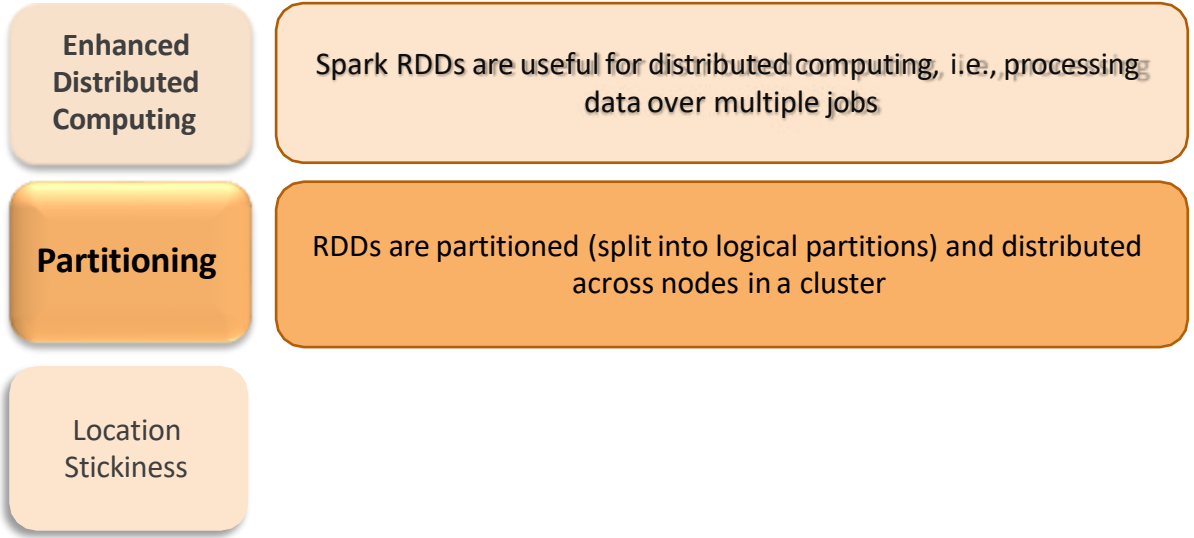
Spark RDDs are useful for distributed computing, i.e., processing data over multiple jobs

Partitioning

Location Stickiness

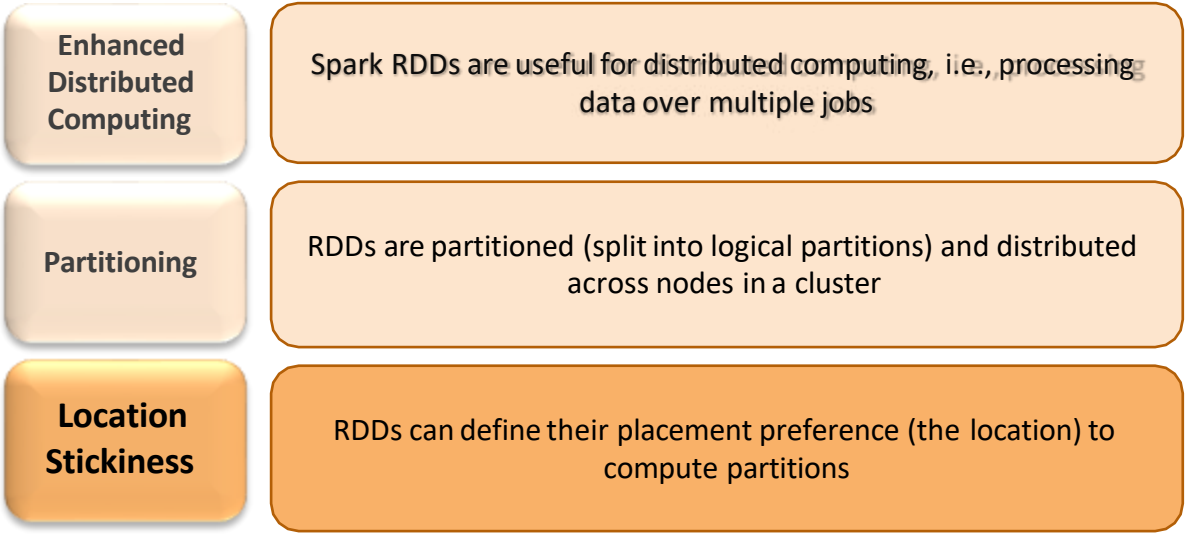
How do RDDs solve the problem?

- Spark RDDs have various capabilities to handle huge volumes of data by processing them parallelly over multiple logical partitions. We can create an RDD, anytime, but Spark RDDs are executed only when they are needed (lazy evaluation)
- Here are some more facts that explain why RDDs are better than the existing frameworks:



How do RDDs solve the problem?

- Spark RDDs have various capabilities to handle huge volumes of data by processing them parallelly over multiple logical partitions. We can create an RDD, anytime, but Spark RDDs are executed only when they are needed (lazy evaluation)
- Here are some more facts that explain why RDDs are better than the existing frameworks:



Features of **Spark RDDs**

Features of Spark RDDs

- **In-memory Computation**
- **Immutable or Read-only**
- **Lazy Evaluation**
- **Cacheable**
- **Parallel Data Processing**
- **Typed**

The data inside an RDD can be stored in memory for as long as possible

Features of Spark RDDs

- In-memory Computation
- **Immutable or Read-only**
- Lazy Evaluation
- Cacheable
- Parallel Data Processing
- Typed

An RDD can’t be changed once created and can only be transformed using transformations

Features of Spark RDDs

- In-memory Computation
- Immutable or Read-only
- **Lazy Evaluation**
- Cacheable
- Parallel Data Processing
- Typed

The data inside an RDD is not available or transformed until an action is executed that triggers the execution

Features of Spark RDDs

- In-memory Computation
- Immutable or Read-only
- Lazy Evaluation
- **Cacheable**
- Parallel Data Processing
- Typed

Cache stores the intermediate RDD results in memory only, i.e., the default storage of RDD cache is in memory

Features of Spark RDDs

- In-memory Computation
- Immutable or Read-only
- Lazy Evaluation
- Cacheable
- **Parallel Data Processing**
- Typed

RDDs can process data in parallel

Features of Spark RDDs

- In-memory Computation
- Immutable or Read-only
- Lazy Evaluation
- Cacheable
- Parallel Data Processing
- **Typed**

RDD records have types, e.g., Int, Long, etc.

Ways to Create RDDs in **PySpark**

Creating RDDs in PySpark

- The conf object is the configuration for a Spark application
- We define the App Name and the Master URL in it
- sc is an object of SparkContext

```
from pyspark.conf import SparkConf
from pyspark.context import SparkContext
conf = SparkConf()
conf.setMaster("local").setAppName("My app")

conf.get("spark.master")

conf.get("spark.app.name")

sc = SparkContext(conf=conf)
sc.master

sc.appName

sc.sparkHome is None
```

Creating RDDs in PySpark

Creating an RDD Using a List:

```
i values = [1, 2, 3, 4, 5]
  rdd = sc.paralleli e(values)
```

Printing the RDD Values:

```
 i rdd.take(5)

1, 2, 3, 4, S]
```

Creating RDDs in PySpark

Uploading a File to Google Colab:

```
from google.colab import files
uploaded = files.upload()
```

Initializing an RDD Using a Text File:

```
rdd = sc.textFile(”Sperk.txt”)
```

Printing the Text from the RDD:

```
rdd . take
'apache Span' ›:it: r›ythen :  POS:ark']
```

RDD Persistence and Caching

RDD Persistence and Caching

- Spark RDDs are lazily evaluated; thus when we wish to use the same RDD multiple times, it results in recomputing the RDD
- To avoid computing an RDD multiple times, we can ask Spark to persist the data
- In this case, the node that computes the RDD stores its partitions
- If a node has data persisting on it fails, Spark will recompute the lost partitions of the data when required
- RDDs come with a method called unpersist() that lets us manually remove them from the cache

```
aba = sc.parallelize(range(1,10000,2))
aba.persist()
```

Persistence Level

Level	Space Used	CPU Time	In Memory	On Disk	Comments
MEMORY_ONLY	High	Low	Yes	No	Stores an RDD as a deserialized Java object in JVM. If it does not fit in memory, some partitions will not be cached and will be recomputed when needed
MEMORY_ONLY_SER	Low	High	Yes	No	Stores an RDD as a serialized Java object. It stores one-byte array per partition
MEMORY_AND_DISK	High	Medium	Some	Some	Stores an RDD as a deserialized Java object in JVM. If the full RDD does not fit in memory, the remaining partition is stored on the disk, instead of recomputing it
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to the disk if there is too much data to fit in memory and stores serialized representation in memory
DISK_ONLY	Low	High	No	Yes	Stores the RDD partitions only on disk

RDD Persistence

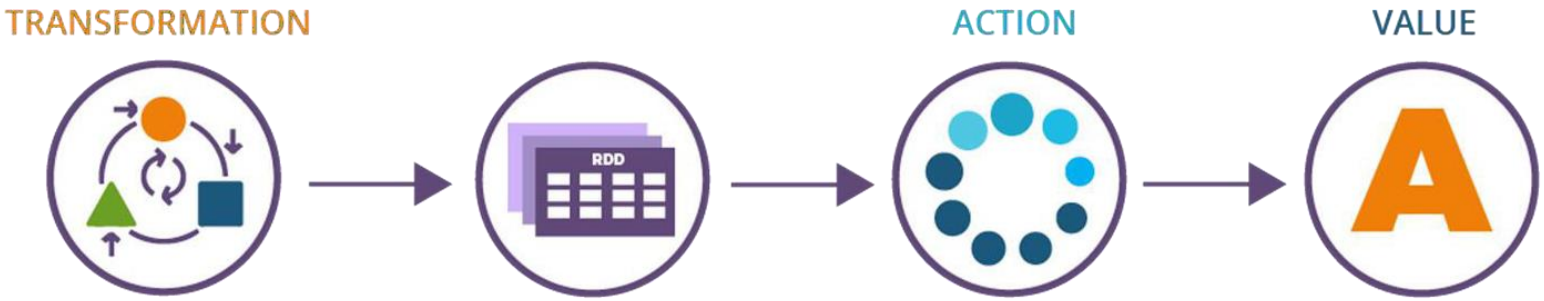
Caching

- It is very useful when data is accessed repeatedly, such as, querying a small ‘hot’ dataset or running an iterative algorithm
- Cache is fault-tolerant
- Let us revisit our example and mark our linesWithSpark dataset to be cached

```
df2 = spark.read.text(text.txt, lineSep=",")
df2.cache()
```

Operations on RDDs

Operations on RDDs

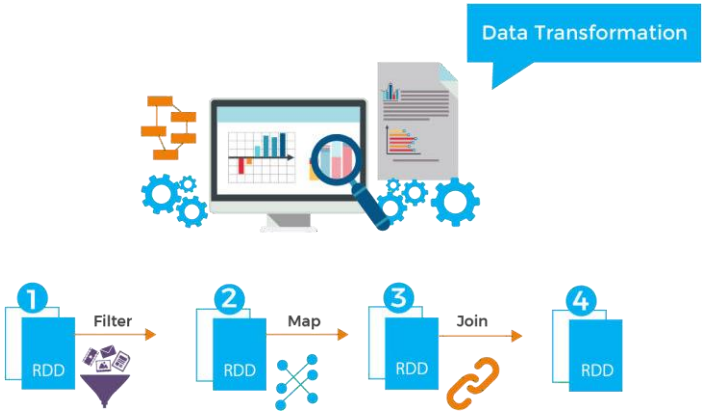


- There are two types of data operations we can perform on an RDD, transformations and actions
  - A **transformation** will return a new RDD as RDDs are generally immutable
  - An **action** will return a value

RDD
Transformations

Operations on RDDs: Transformations

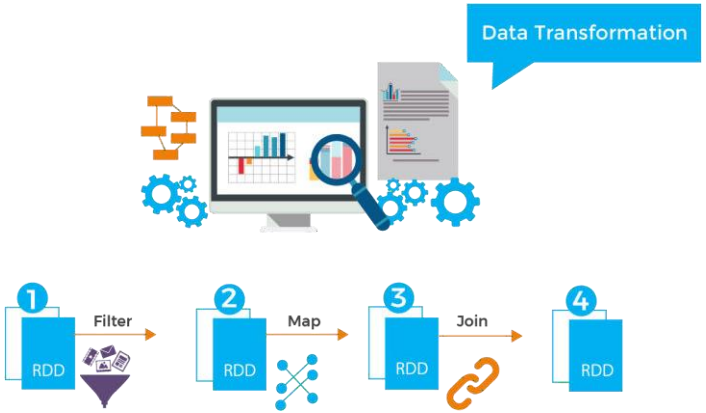
Transformations are lazy operations on an RDD that create one or more new RDDs



Operations on RDDs: Transformations

Transformations are lazy operations on an RDD that create one or more new RDDs

RDD transformations return a pointer to the new RDD and allow us to create dependencies between RDDs. Each RDD in a dependency chain (a string of dependencies) has a function for calculating its data and a pointer (dependency) to its parent RDD

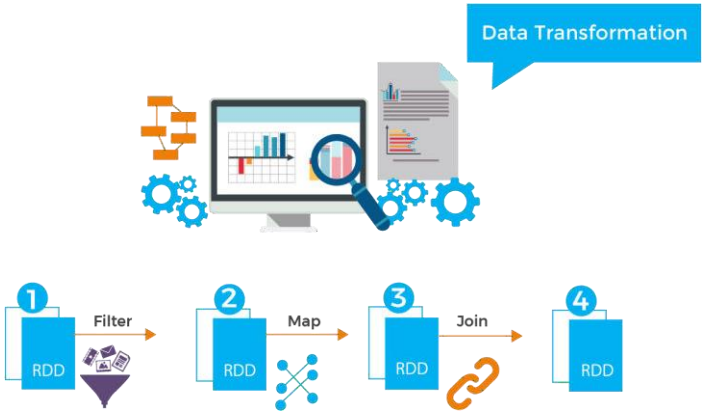


Operations on RDDs: Transformations

Transformations are lazy operations on an RDD that create one or more new RDDs

RDD transformations return a pointer to the new RDD and allow us to create dependencies between RDDs. Each RDD in a dependency chain (a string of dependencies) has a function for calculating its data and a pointer (dependency) to its parent RDD

Spark is lazy, so nothing will be executed unless we call some transformation or action that will trigger the job creation and execution



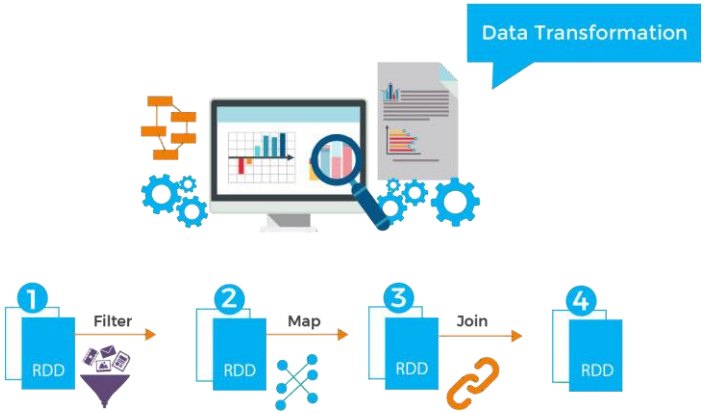
Operations on RDDs: Transformations

Transformations are lazy operations on an RDD that create one or more new RDDs

RDD transformations return a pointer to the new RDD and allow us to create dependencies between RDDs. Each RDD in a dependency chain (a string of dependencies) has a function for calculating its data and a pointer (dependency) to its parent RDD

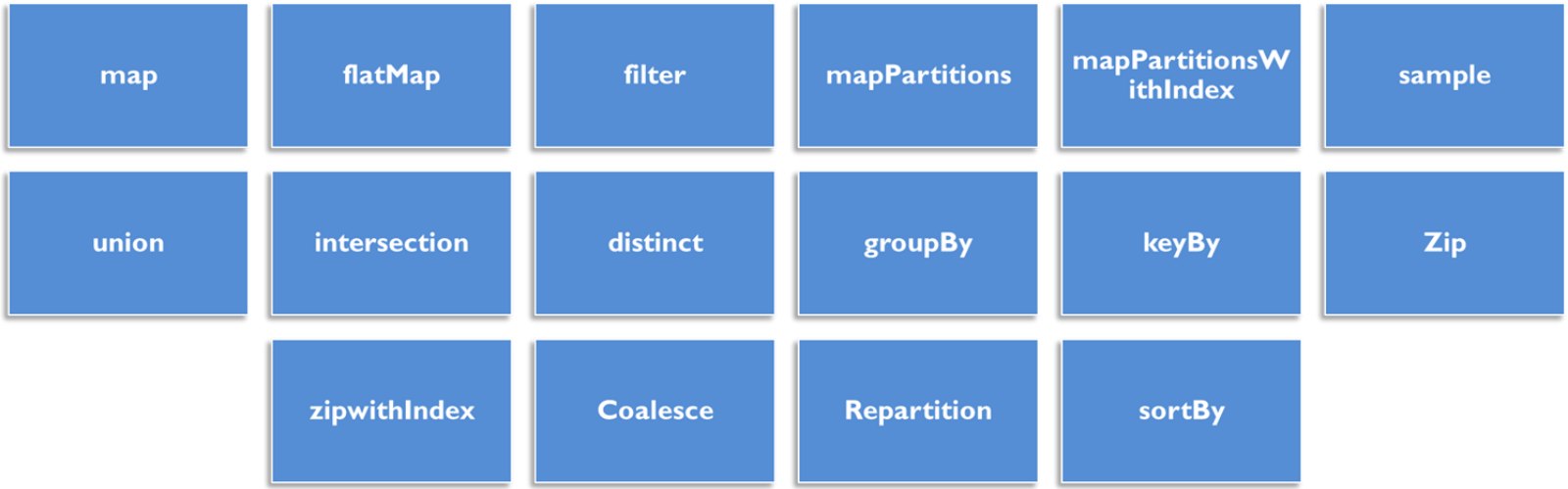
Spark is lazy, so nothing will be executed unless we call some transformation or action that will trigger the job creation and execution

Therefore, RDD transformation is not a set of data but a step in a program (might be the only step) telling Spark how to get data and what to do with it



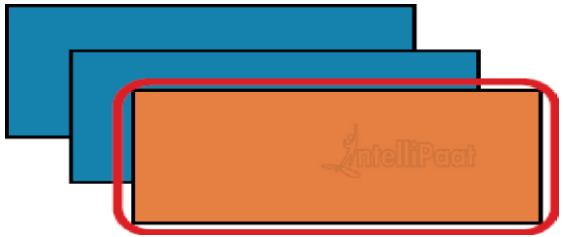
Operations on RDDs: Transformations

Given below is a list of RDD Transformations:



“map”

- Passes each element through a function



“map”

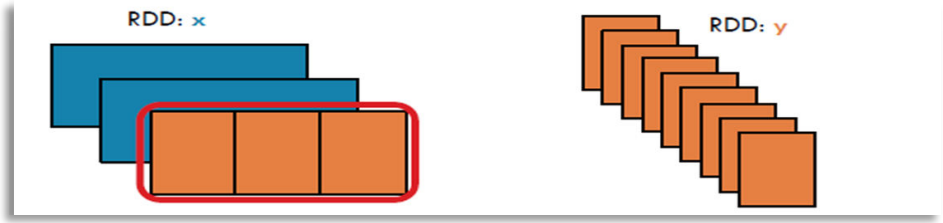
```
x = sc.parallelize(["spark", "rdd", "example",  "sample", "example"])
y = x.map(lambda  x:(x,  1))
y.collect()
```

```
x =:c.parollelize(["›:al’ñ", "l'JJ", ":-xal’:l:-",   "=auple", "°›:and.l°"],
y = x.map,lambda x:(x, 1))
y.collect{)
```

```
('spark', I) , ('rdd , 1) , ('exanpl e , 1),  ('saup1e', 1) , ('exanpie', I) ]
```

“flatMap”

- It is like map, but here each input item can be mapped to 0 or more output items (so, a function should return a sequence rather than a single item)



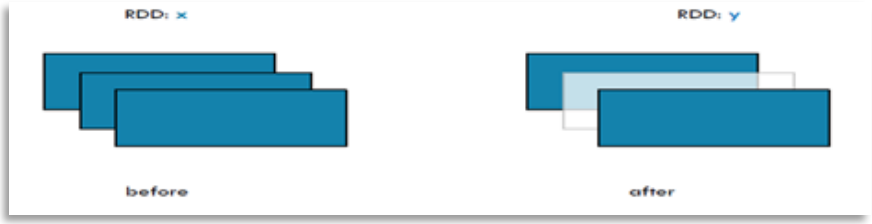

```
rdd = sc.parallelize([2, 3, 4])
sorted(rdd.flatMap(lambda x: range(1,
x)).collect())
```

```
1 rdd = sc.parallelize([2, 3, 4])
2 sorted(rdd.flatMap(lambda x: range(1, x)).collect())
```

```
[1, 1, 1, 2, 2, 3]
```

“filter”

- Returns a collection of elements on the basis of the condition provided in the function



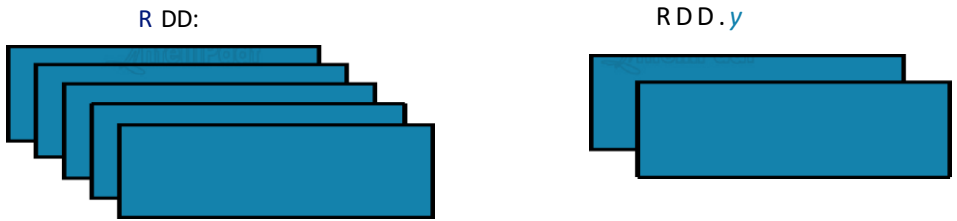

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd.filter(lambda x: x % 2 == 0).collect()
```

```
1 rdd = sc.parallelize([1, 2, 3, 4, 5])
2 rdd.filter(lambda x: x % 2 == 0).collect()
```

```
[2, 4]
```

“sample”

- Samples a fraction of the data, with or without replacement, using a given random number generator seed
- We can add following parameters to sample method:
  - withReplacement: Elements can be sampled multiple times (replaced when sampled out)
  - fraction: Makes the size of the sample as a fraction of the RDD's size without replacement
  - seed: A number as a seed for the random number generator



“sample”

```
parallel = sc.parallelize(range(9))
parallel.sample(True,.2).count()
parallel.sample(False,1).collect()
```

```
parollel = sc.pal’ollelize(range(9))
parollel.somple{Tl’ue,.2).county )
```

1

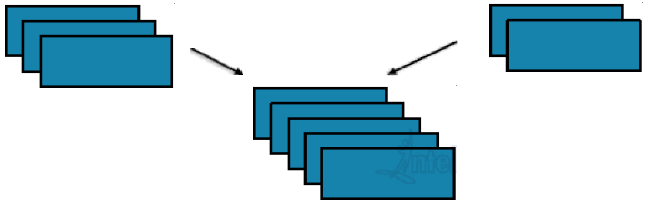
```
parollel. Somple.{Folse,1}.collrct()
```

8, 1, 2, 3, 4, 5, 8, 7, 8]

“union”

- Returns the union of two RDDs after concatenating their elements

```
parallel = sc.parallelize(range(1,9))
par = sc.parallelize(range(5,15))
parallel.union(par).collect()
```



```
parollel = :c.parollelize(range(1,9,a
par = sc.porallrli:e(range(5,15))
parollel.union(pol’).collect(,
```

1, 2, 3, 4, S, 8, 7, 8, 5, €, 7, 8, D, 18, 11, 12, 13, 14]

“intersection”

- Similar to union, but returns the intersection of two RDDs

```
parallel = sc.parallelize(range(1,9))
par = sc.parallelize(range(5,15))
parallel.intersection(par).collect()
```

```
parallel = sc.pal’ llelize(range(1,9,)
par = sc.p rall lize(range(5,15))
parallel.inters ction{p l’).collect(,
```

[G, 8, 5, 7]

“distinct”

- Returns a new RDD with distinct elements within the source data

```
parallel = sc.parallelize(range(1,9))
par = sc.parallelize(range(5,15))
parallel.union(par).distinct().collect()
```

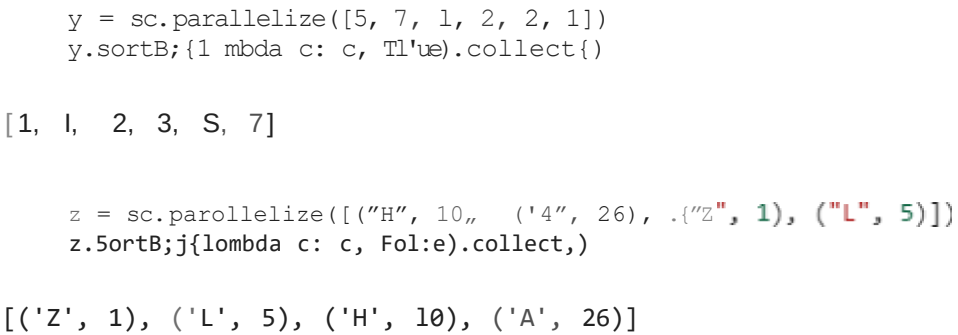
```
parollel = :c.pal’ollelize(range(1,9,)
par = sc.p rallrli:e(range(5,15))
parollel.union(po+’).distinct{,.collrct()
```

```
[2, 4, 6, 8, 10, 12, 14, I, 3, 5, 7, 9, 11, 13]
```

“sortBy”

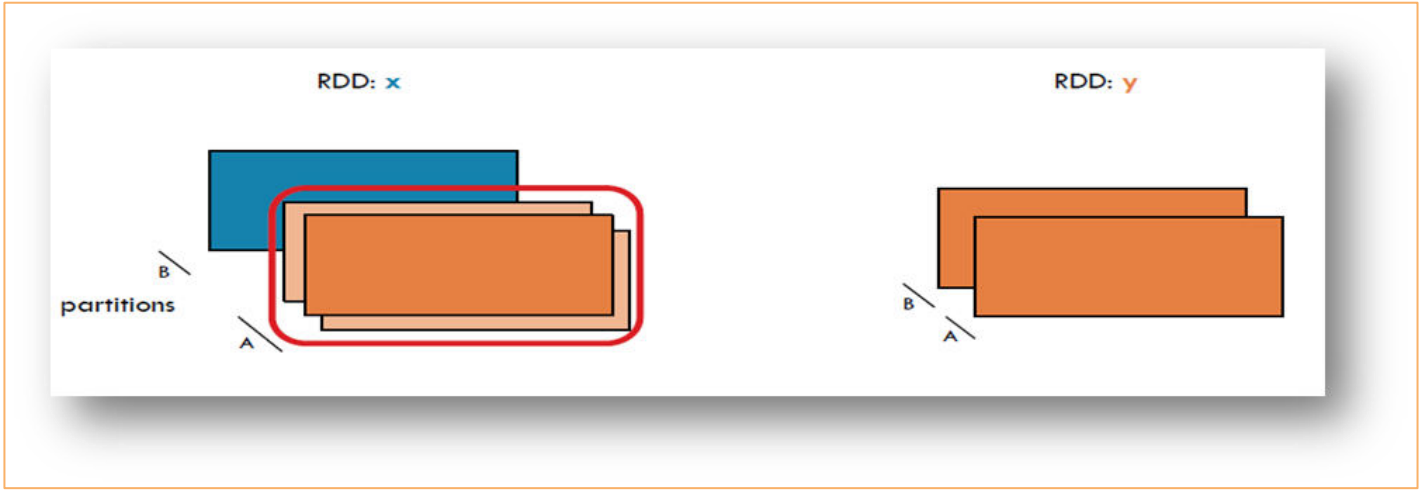
- Returns the RDD sorted by the given key function

```
y = sc.parallelize([5, 7, 1, 3, 2, 1])
y.sortBy(lambda c: c, True).collect()
z = sc.parallelize([(”H“, 10), (“A”, 26), (“Z”, 1), (“L”, 5)])
z.sortBy(lambda c: c, False).collect()
```



“mapPartitions”

- Can be used as an alternative to map() and foreach()
- mapPartitions() is called once for each partition unlike map() and foreach(), which are called for each element in the RDD



“mapPartitions”

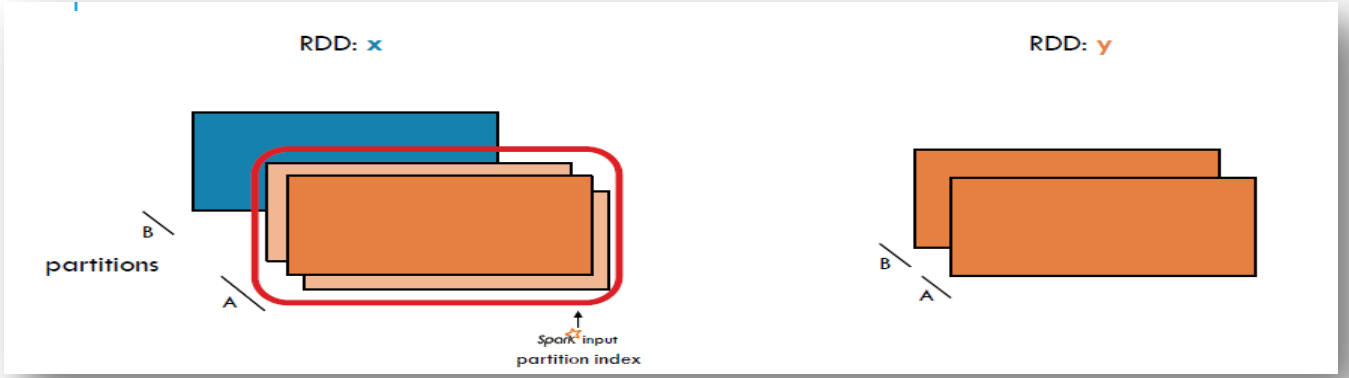
```
rdd = sc.parallelize([1, 2, 3, 4], 2)
def f(iterator): yield sum(iterator)
rdd.mapPartitions(f).collect()
```

```
1 rdd = sc.parallelize([1, 2, 3, 4], 2)
2 def f(iterator): yield sum(iterator)
3 rdd.mapPartitions(f).collect()
```

```
[3, 7]
```

“mapPartitions” - withIndex

- Returns a new RDD by applying a function to each partition of the RDD, while tracking the index of the original partition



“mapPartitions” - WithIndex

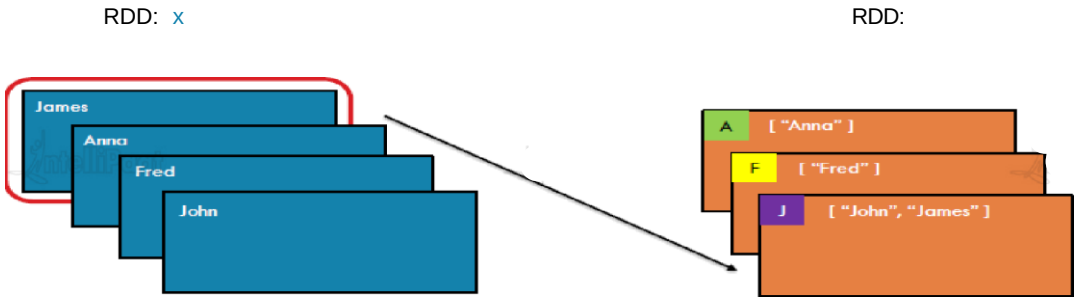
- Returns a collection of elements on the basis of the condition provided in the function

```
rdd = sc.parallelize([1, 2, 3, 4], 4)
def f(splitIndex, iterator): yield splitIndex
rdd.mapPartitionsWithIndex(f).sum()
```

```
rdd = sc.porallrlize([1, 2, ñ, 4], 4)
def f(splitIndex, iterator): yield splitInder
rdd.mapPartitionsl1ithIndex(fj.sum()
```

“groupBy”

- Returns a new RDD by grouping objects in the existing RDD using the given grouping key

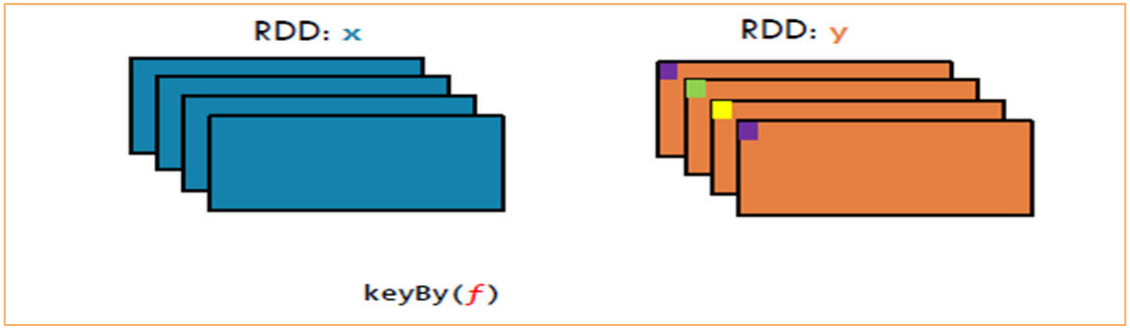



```
rdd = sc.porallrli:e([1, 1, l, 3, 5, 8])
result = rdd.groupBy(lambda x: x % 2).collrct()
sorted([{x, :orted(y), for (x, y) in result])
```

```
[(0, [2, 8]), (1, [1, 1, 3, 5])]
```

“keyBy”

- Returns a new RDD by changing the key of the RDD element using the given key object



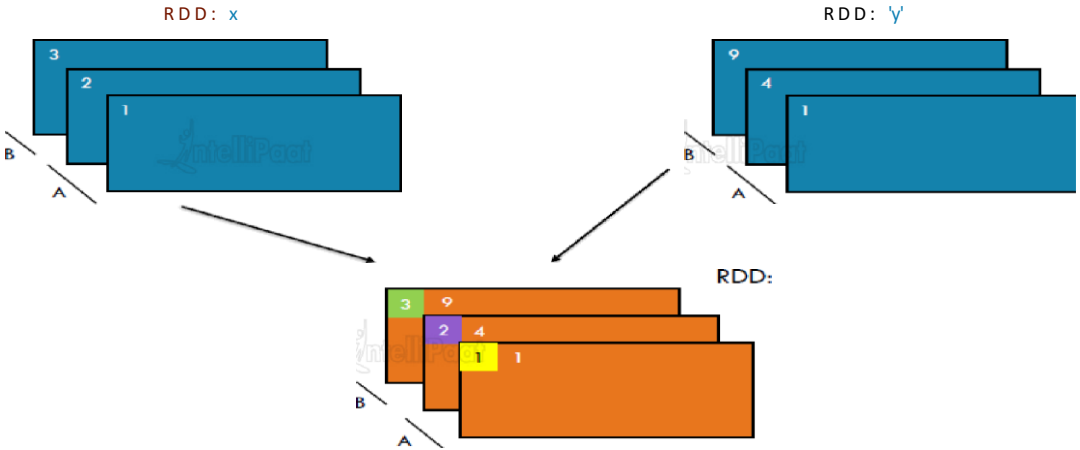
“keyBy”

```
x = sc.parallelize(range(0,3)).keyBy(lambda x: x*x)
y = sc.parallelize(zip(range(0,5), range(0,5)))
[(x, list(map(list, y))) for x, y in sorted(x.cogroup(y).collect())]
```

```
1 x = sc.parallelize(range(0,3)).keyBy(lambda x: x*x)
2 y = sc.parallelize(zip(range(0,5), range(0,5)))
3 [(x, list(map(list   y))) for x, y in sorted(x.cogroup(y).collect())]

[(0, [[0], [B]]),
 (1, [[1], [1]]),
 (2, [[], [2]]),
 (3, [[], [3]]),
 (4, [[2], [4]])]
```

“zip”

- Joins two RDDs by combining the i^{th} part of either partition with each other
- Returns an RDD formed from this list and another iterable collection by combining the corresponding elements in pairs
- If one of the two collections is longer than the other, its remaining elements are ignored



“zip”

```
x = sc.parallelize(range(0,5))
y = sc.parallelize(range(1000, 1005))
x.zip(y).collect()
```

```
i x = sc.parallelize(rarge(u,5))
i y = sc.panallelize(nzrge(i98 , 1905))
I x.:Ip(y).collect()
```

```
[(0, i000), (1, 1001), (2, }002), (3, 1003), (4, 1004)]
```

"zip" - WithIndex

- Returns a new RDD that contains pairs consisting of all elements of the given list paired with their indices. Indices start at 0

```
sc.parallelize([“a”, "b", "c", “d“],
3).zipWithIndex().collect()
```

```
i sc.parallelize([”a”, "o", "c", ”d”], 3).zipMithIndex().collect(}

[('a', 0), ('b', 1), ('c', 2), ('d', 3)]
```

“repartition”

- Used to either increase or decrease the number of partitions in an RDD
- Does a full shuffle and creates new partitions with the data that are distributed evenly

```
rdd = sc.parallelize([1,2,3,4,5,6,7], 4)
sorted(rdd.glom().collect())
len(rdd.repartition(2).glom().collect())
```

```
i rdd = sc.paralleli e( i,2,3,4,S,C,7], 4)
i sor: d(rdd.glom().collect())

[[1] , [2,  3] , [4,  5] , [6,  7] ]

i  ea(rdd.nepantition(2).glom().collect())
```

“coalesce”

- This method is used to reduce the number of partitions in an RDD

```
sc.parallelize([1, 2, 3, 4, 5], 3).glom().collect()
sc.parallelize([1, 2, 3, 4, 5],
3).coalesce(2).glom().collect()
```

```
i sc.parallelize([1,   , 3, 4, 5], 3}.glom().collect()

[[i], [2, 3], [4, 5]]

i sc.panallelize([i, l, 8, 4, 5], ñ}.coalesce(2).glom().collect()

[[1], [2, 3, 4, 5]]
```

“coalesce” vs “repartition”

Coalesce()	Repartition()
Uses the existing partitions to minimize the amount of data that's shuffled	Creates new partitions and does a full shuffle
Results in partitions with different amounts of data (at times with much different sizes)	Results in roughly equal-sized partitions
Faster	Not so faster

RDD Actions

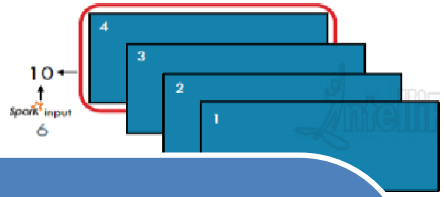
Operations on RDDs: Actions

- Unlike transformations that produce RDDs, action functions produce a value back to the Spark driver program
- Actions may trigger a previously constructed, lazy RDD to be evaluated

Given below is a list of commonly used RDD actions:

“reduce”

- Aggregates elements of a dataset through a function




```
from operator import add
# Create an RDD
rdd = sc.parallelize((2 for _ in range(10)))
# Use map to transform each element to 1
mapped_rdd = rdd.map(lambda x: 1)
# Cache the transformed RDD
cached_rdd = mapped_rdd.cache()
# Use reduce with add to sum all elements
sum_result = cached_rdd.reduce(add)
# Print the result
print(sum_result)  # Output: 10
```

“first”

- Returns the first element in an RDD

```
rdd = sc.parallelize([2, 3, 4])     # Use the first() action to get the first element
first_element = rdd.first()         # Print the result
print(first_element) # Output: 2
```

“takeOrdered”

- Returns an array with the given number of ordered values in an RDD

```
# Create an RDD rdd = sc.parallelize([4, 2, 3, 7, 1, 5])
# Use takeOrdered to get the first 3 elements in ascending order
top_3_elements = rdd.takeOrdered(3)
# Print the result print(top_3_elements)
# Output: [1, 2, 3]
# Use takeOrdered with a custom comparator to get the first 3 elements in descending order
top_3_descending = rdd.takeOrdered(3, key=lambda x: -x)
# Print the result
print(top_3_descending) # Output: [7, 5, 4]
```

“take”

- Returns an array as the specified number in the take method.

```
rdd = sc.parallelize([4, 2, 3, 7, 1, 5]) # Use take to get the first 3 elements
first_3_elements = rdd.take(3) # Print the result
print(first_3_elements) # Output: [4, 2, 3]
```

“count”

- Returns a long value indicating the number of elements present in an RDD

```
# Create an RDD
rdd = sc.parallelize([4, 2, 3, 7, 1, 5])
# Use count to get the number of elements in the RDD
element_count = rdd.count()
# Print the result
print(element_count) # Output: 6
```

“collect”

- Returns the elements of the dataset as an array back to the driver program.
- Should be used wisely as all worker nodes return the data to the driver node.
- If the dataset is huge in size, then this may result in an OutOfMemoryError.

```
# Create an RDD
rdd = sc.parallelize([4, 2, 3, 7, 1, 5])
# Use collect to retrieve all elements of the RDD
all_elements = rdd.collect()
# Print the result
print(all_elements)
 # Output: [4, 2, 3, 7, 1, 5]
```

“SaveAsTextfile” - filepath

- Writes the entire RDD's dataset as a text file on the path specified in the local filesystem or HDFS

```
a = sc.parallelize(range(1,10000), 3)
a.saveAsTextFile(”/usr/bin/mydata_a1”)

x = sc.parallelize([1,2,3,4,5,6,6,7,9,8,10,21],3)

x.saveAsTextFile(”/usr/bin/sample1.txt“)
```

“foreach”

- Passes each element in an RDD through the specified function

```
rdd = sc.parallelize([4, 2, 3, 7, 1, 5])
# Define a function to print each element
def print_element(x):
        print(x)
# Use foreach to apply the function to each element of the RDD
rdd.foreach(print_element)
```

“foreach” - Partition

- Executes the function for each partition. Access to the data items contained in the partition is provided via the iterator argument

```
def f(iterator):
        for x in iterator:
print(x)
 sc.parallelize([1, 2, 3, 4, 5]).foreachPartition(f)
```

“max”, “min”, “sum”, “mean”, “variance”, and “stdev”

- Spark RDD supports some mathematical actions like “max”, “min”, “sum”, “mean”, “variance”, and “stdev”

```
numbers = sc.parallelize(range(1,100))
numbers.sum
numbers.min
numbers.variance
numbers.max
numbers.mean
numbers.stdev
```

“max”, “min”, “sum”, “mean”, “variance”, and “stdev”

```
1 numbers = sc.parallelize(range(1,100))
```

```
1 numbers.sum()
```

```
4950
```

```
1 numbers.min()
```

```
1
```

```
1 numbers.variance()
```

```
816.6666666666666
```

```
1 numbers.max()
```

```
99
```

```
1 numbers.mean()
```

```
50.0
```

```
1 numbers.stdev()
```

```
28.577380332470412
```

RDD Functions

RDD Functions

- Some of the general RDD functions are given in the below table:

Functions	Arguments	Returns
cache	()	Caches an RDD to use without computing again
collect	()	Returns an array of all elements in an RDD
countByValue	()	Returns a map with the number of times each value occurs
distinct	()	Returns an RDD containing only distinct elements
filter	(f: T => Boolean)	Returns an RDD containing only those elements that match with the function **f**
foreach	(f: T => Unit)	Applies the function **f** to each of the elements of an RDD
persist	(); (newLevel: StorageLevel)	Sets an RDD with the default storage level (MEMORY_ONLY); sets the storage level that causes the RDD to be stored after it is computed (different storage levels are there in StorageLevel)

RDD Functions

Functions	Arguments	Returns
sample	(fraction: double)	Returns an RDD of that fraction
toDebugString	()	Returns a handy function that outputs the recursive steps of an RDD
count	()	Returns the number of elements in an RDD
unpersist	()	Removes all the persistent blocks of an RDD from the memory/disk
union	(other: RDD[T])	Returns an RDD containing elements of two RDDs; duplicates are not removed

“countByValue”

```
a=sc.parallelize([1,2,3,4,5,6,7,8,2,4,2,3,3,3,1,1,1])
a.countByValue()
```

“toDebugString”

```
# Create an RDD
rdd = sc.parallelize([1, 2, 3, 4, 5])
# Apply some transformations
mapped_rdd = rdd.map(lambda x: x * 2)
filtered_rdd = mapped_rdd.filter(lambda x: x > 5)
# Get the debug string representation
debug_string = filtered_rdd.toDebugString()
# Print the debug string print(debug_string)
```

Working with **Key–Value Pairs**

Creating Paired RDDs

- There are a number of ways to get paired RDDs in Spark
- There are many formats that directly return paired RDDs for their key–value data; in other cases, we have regular RDDs that need to be turned into paired RDDs
- We can do this by running a map() function that returns key–value pairs

Creating Paired RDDs

```
rdd  =  sc.parallelize([(”al“, ”b1“,”c1“,“d1","el"),(“a2”,“b2”,“c2”,“d2“,“e2“)])
result =rdd.map(lambda x: (x[0], list(x[1:])))
result.collect()
```

Transformations on Paired RDDs

- Transformations on one paired RDD
- **E.g.: RDD =** {(1, 2), (3, 4), (3, 6)}

Function	Purpose	Example	Result
reduceByKey(func)	Combines values with the same key	rdd.reduceByKey(add)	{(1, 2), (3, 10)}
groupByKey()	Groups values with the same key	rdd.groupByKey()	{(1, [2]), (3, [4, 6])}
mapValues(func)	Applies a function to each value of a paired RDD without changing the key	rdd.mapValues(lambda x: x+1)	{(1, 3), (3, 5), (3, 7)}
flatMapValues(func)	Applies a function that returns an iterator to each value of a paired RDD and, for each element returned, produces a key–value entry with the old key; often used for tokenization	rdd.flatMapValues(lambda x: range(x, 5))	{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}
keys()	Returns an RDD of just the keys	rdd.keys()	{1, 3, 3}
sortByKey()	Returns an RDD sorted by the key	rdd.sortByKey()	{(1, 2), (3, 4), (3, 6)}

Transformations on Paired RDDs

- Transformations on two paired RDDs
- **E.g.: RDD1 =** {(1, 2), (3, 4), (3, 6)} **RDD2 =** {(3, 9)}

Function	Purpose	Example	Result
subtractByKey	Removes elements with the key present in the other RDD	rdd.subtractByKey(other)	{(1, 2)}
join	Performs an inner join between both RDDs	rdd.join(other)	{(3, (4, 9)), (3, (6, 9))}
rightOuterJoin	Performs a join between two RDDs where the key must be present in the first RDD	rdd.rightOuterJoin(other)	{(3,(Some(4),9)), (3,(Some(6),9))}
leftOuterJoin	Performs a join between two RDDs where the key must be present in the other RDD	rdd.leftOuterJoin(other)	{(1,(2,None)), (3, (4,Some(9))), (3, (6,Some(9)))}
cogroup	Groups data from both RDDs sharing the same key	rdd.cogroup(other)	{(1,([2],[])), (3, ([4, 6],[9]))}

RDD Lineage

RDD Lineage

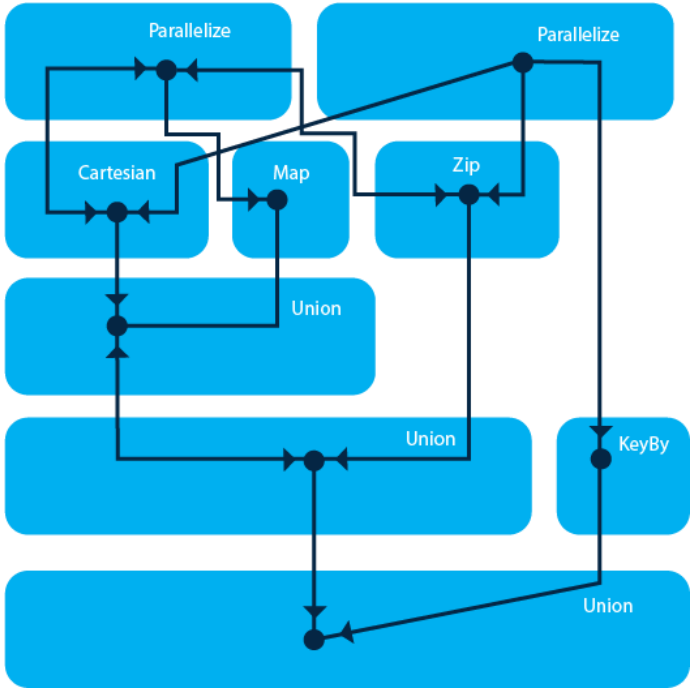
- RDD Lineage is a graph of all parent RDDs of an RDD
- It is built by applying transformations to the RDD and creating a logical execution plan
- Consider the following series of transformations:

```
rdd.toDebugString #to print rdd lineage
```

```
PythonRDD[141] at RDD at PythonRDD.scala:53 []\n
MapPartitionsRDD[140] at mapPartitions at
PythonRDD.scala:133 []\n
ShuffledRDD[139] at partitionBy at
NativeMethodAccessorImpl.java:0 []\n
PairwiseRDD[138] at subtract at <ipython-input-58-
e1f9a4054d92>:3 []\n
```

RDD Lineage

- The following RDD graph is the result of the series of transformations mentioned earlier
- An RDD lineage graph is hence a graph of transformations that need to be executed after an action has been called
- We can create an RDD lineage graph using the **RDD.toDebugString** method



Word Count Using **RDD** Concepts

Word Count Program

- A word count program will return the frequency of every word
- An RDD is created to store a text file data
- A few RDD operations can complete this program

```
rdd =sc.textFile("/content/Pyspark.txt")
nonempty_lines = rdd.filter(lambda x: len(x) > 0)
words = nonempty_lines.flatMap(lambda x: x.split(' '))
wordcount = words.map(lambda x:(x,1)).reduceByKey(lambda x,y: x+y).map(lambda x: (x[1], x[0])).sortByKey(False)
for word in wordcount.collect():
   print(word)
wordcount.saveAsTextFile("/content/Wordcount")
```

Word Count Program

Creating an RDD Using a Text File:

```
1 rdd =sc.textFile("/content/Pyspark.txt")
```

Filtering Text Data from an RDD:

```
1 nonempty_lines = rdd.filter(lambda x: len(x) > 0)
```

Word Count Program

Splitting Words Using the flatMap Method:

```
1 words = nonempty_lines.flatMap(lambda x: x.split(' '))
```

Counting the Frequency of Each Word:

```
1 wordcount = words.map(lambda x:(x,1)).reduceByKey(lambda x,y: x+y)
2 .map(lambda x: (x[1], x[0])).sortByKey(False)
```

Word Count Program

Printing the Frequency of Each Word:

```
1 for word in wordcount.collect():
2    print(word)
```

```
(2, '=')
(1, 'rdd')
(1, 'sc.parallelize([("a1","b1","c1","d1","e1"),')
(1, '("a2","b2","c2","d2","e2")])')
(1, 'result')
(1, 'rdd.map(lambda')
(1, 'x:')
(1, '(x[0],')
(1, 'list(x[1:]))))')
(1, 'result.collect()')
```

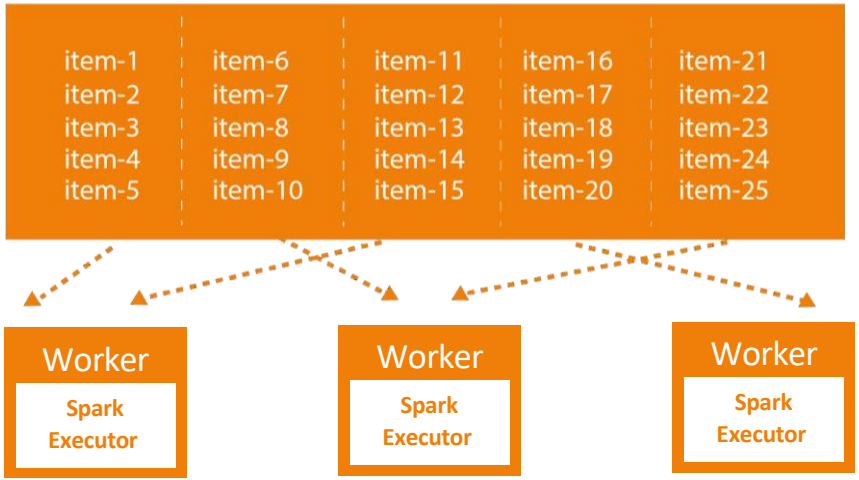
Saving the Output in a Folder:

```
1 wordcount.saveAsTextFile("/content/Wordcount")
```

RDD Partitioning and Achieving Parallelism

RDD Partitioning

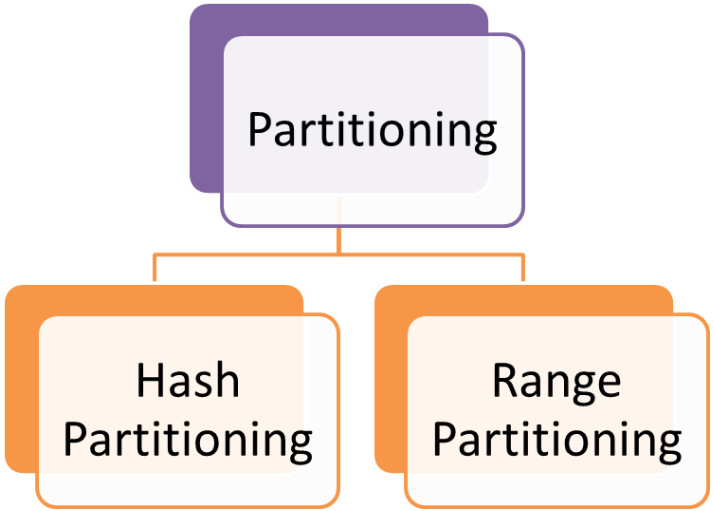
- A partition is a logical chunk of a large distributed dataset
- Spark manages data using partitions that help parallelize distributed data processing with minimal network traffic for sending data between executors
- By default, Spark tries to read data into an RDD from the nodes that are close to it
- Here, an RDD is split into 5 partitions:



RDD Partitioning

- Since Spark usually accesses distributed partitioned data, to optimize transformations, it creates partitions that can hold the data chunks
- RDDs get partitioned automatically without a programmer's intervention
- However, there are times when we would like to adjust the size and number of partitions or the partitioning scheme according to the needs of our application
- We can use the def getPartitions: Array[Partition] method on an RDD to know the number of partitions in the RDD

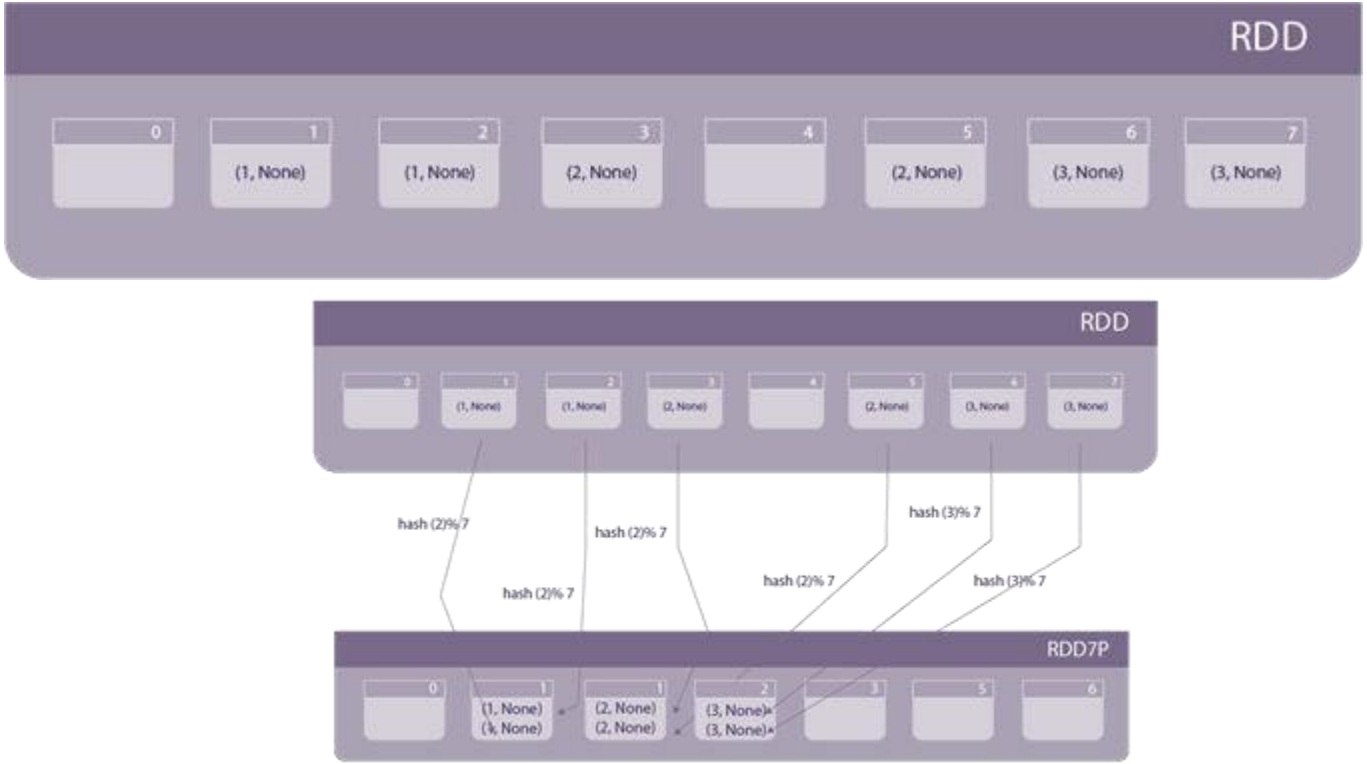
RDD Partitioning: Types



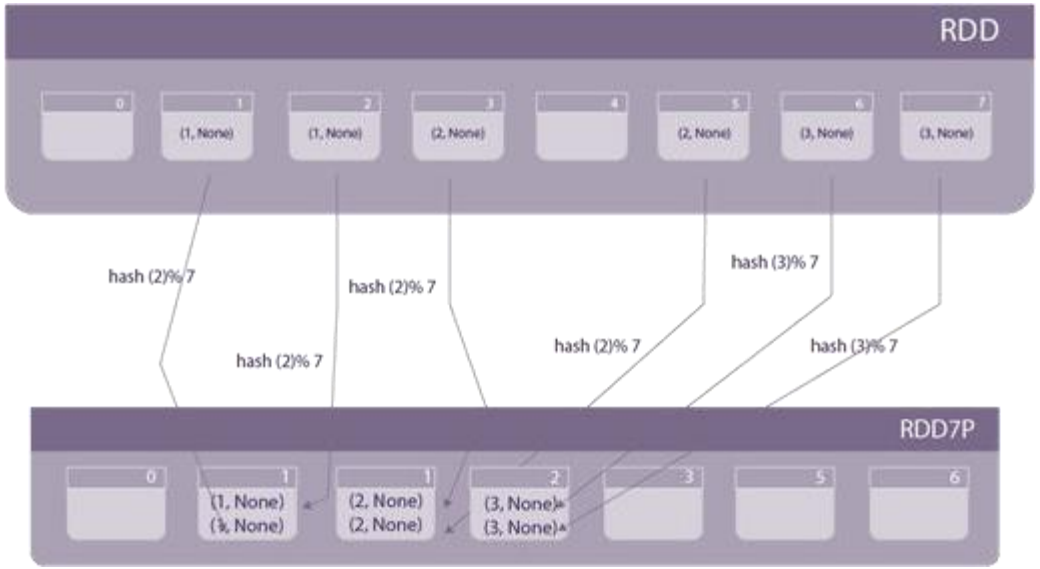
- Customizing partitioning is possible only on paired RDDs
- One of the basic advantages is that as similar kinds of data is co-located, shuffling of data across clusters reduces in transformations like groupByKey, reduceByKey, etc. which in turn increases job performance

HashPartitioner

- Hash partitioning is a partitioning technique where a hash key is used to distribute elements evenly across different partitions

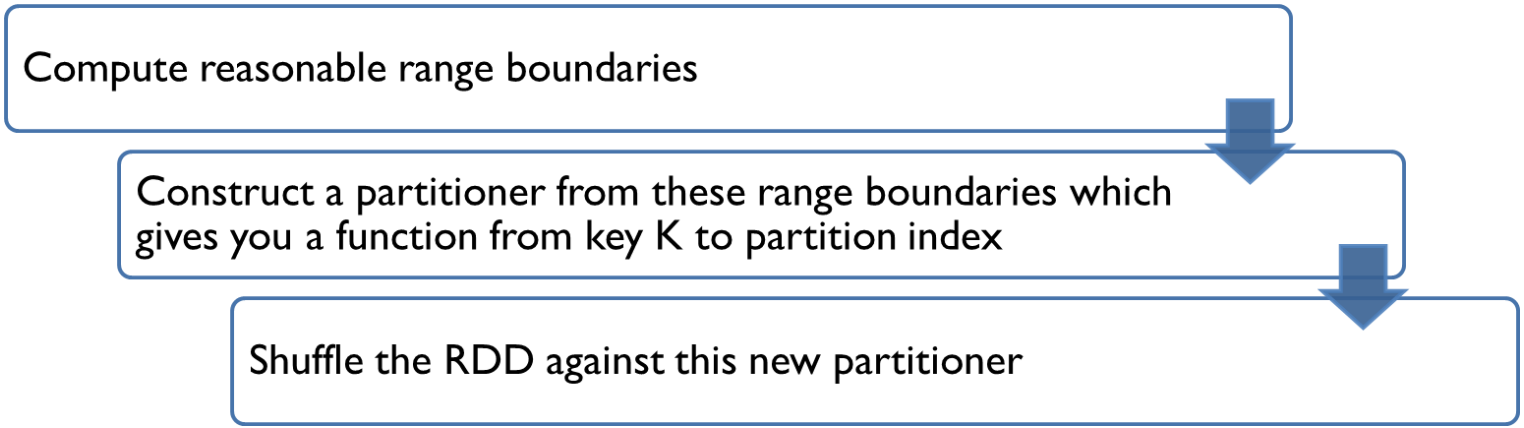


HashPartitioner

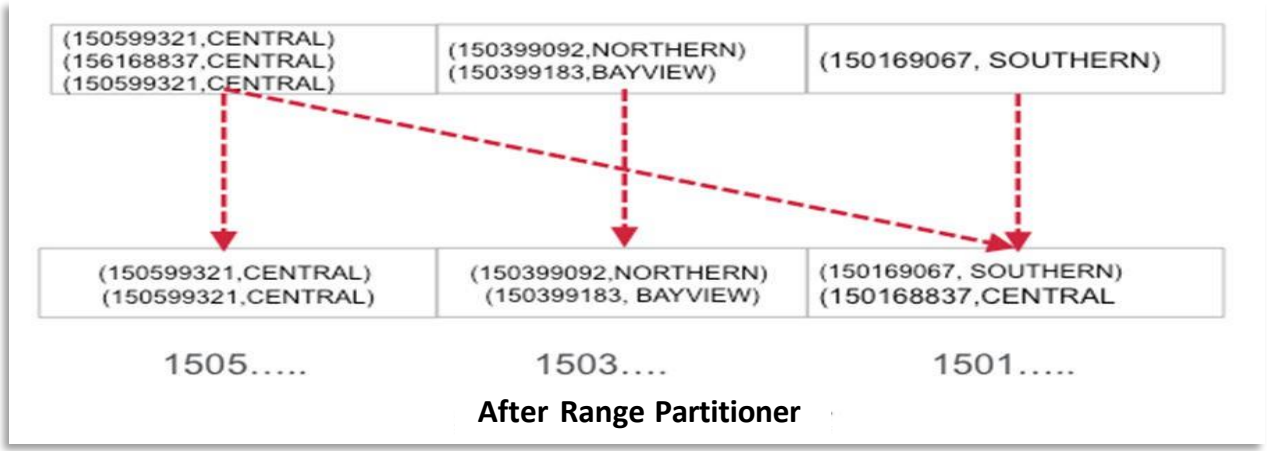


RangePartitioner

- Some Spark RDDs have keys that follow a particular order; for such RDDs, range partitioning is an efficient partitioning technique
- In the range partitioning method, tuples having keys within the same range will appear on the same machine
- Keys in a RangePartitioner are partitioned based on the set of sorted range of keys and ordering of keys
- This involves three steps:



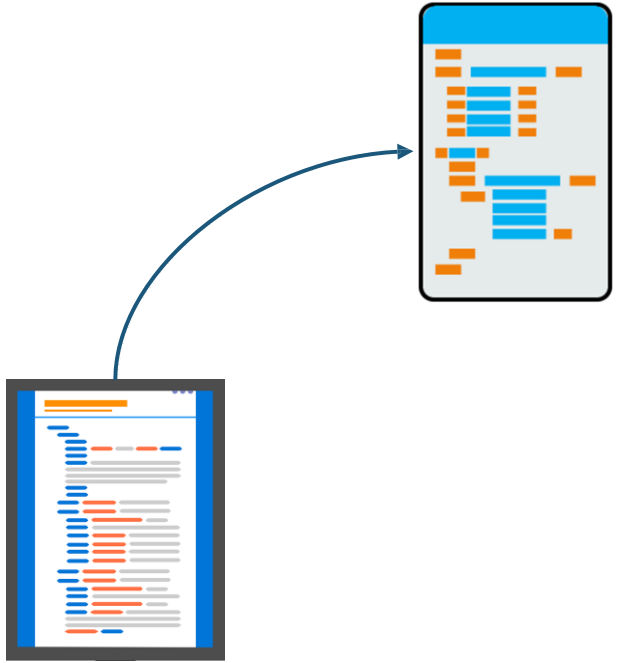
RangePartitioner



Passing Functions to **Spark**

Passing Functions to Spark

- Most of Spark's transformations, and some of its actions, depend on passing in functions that are used by Spark to compute data
- Each of the core languages (Python, Java, and Scala) has a slightly different mechanism for passing functions to Spark
- In Python, we can pass a function inside another function, similar to Python's other functional APIs
- Although, some other considerations come into play—namely, the function we pass, and the data referenced in it, needs to be serializable
- Spark's API relies heavily on passing functions in the driver program to run on the cluster

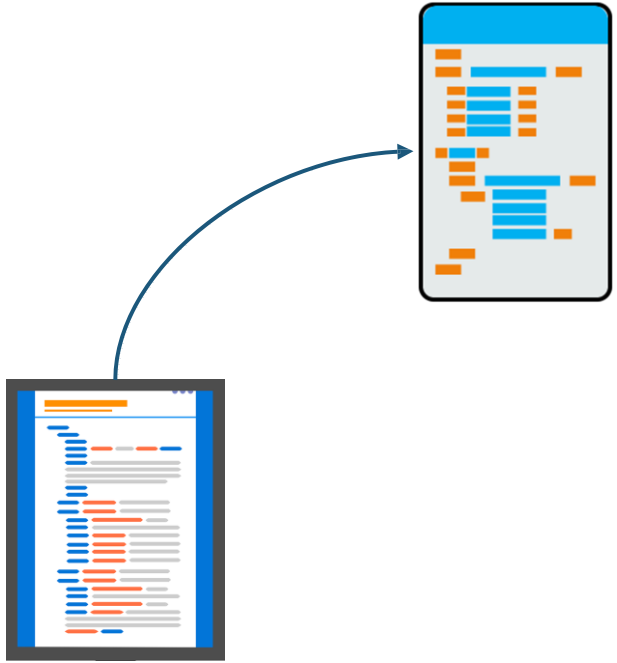


Passing Functions to Spark

- There are two recommended ways to do this:

By defining an anonymous function, a method used for short pieces of code

Static methods, used for a global singleton object



Passing Functions to Spark

By defining an anonymous function, a method used for short pieces of code

- We can define an anonymous function (i.e., one with no name) that returns a given integer plus 2:

```
1 lambda x: x+2
```

- On the left of ":" is a list of parameters and on the right is an expression involving the parameters
- We can name the functions as well

```
1 rdd = sc.paralleli e([i,2,3,4,S])
  rdd.mao(lambda x: xa2).collect()
```

- Functions may take multiple parameters, or they can take no parameters

Passing Functions to Spark

Static methods, used for a global singleton object

- For example, we can define the object MyFunctions and then pass MyFunctions.func1 as follows:

```
object MyFunctions {
  def func1(s: String): String = { ... } }

myRdd.map(MyFunctions.func1)
```

- It is also possible to pass a reference to a method in a class instance (as opposed to a singleton object), which requires sending the object that contains that class along with the method

Passing Functions to Spark

Static methods, used for a global singleton object

- For example, consider the following:

```
class MyClass {
  def func1(s: String): String = { ... }
  def doStuff(rdd: RDD[String]): RDD[String] = { rdd.map(func1) } }
```

- Here, if we create a new MyClass instance and call doStuff on it, the map inside references the func1 method of that MyClass instance, so the whole object needs to be sent to the cluster
- It is similar to writing rdd.map(x => this.func1(x))

Quiz!

Question #1

Which of the following is a module for structured data processing?

A	GraphX
B	MLlib
C	Spark SQL
D	Spark R

Answer #1

Which of the following is a module for structured data processing?

A	GraphX
B	MLlib
C	**Spark SQL**
D	Spark R

Question #2

Which is a distributed graph processing framework on top of Spark?

A	GraphX
B	MLlib
C	Spark SQL
D	Spark R

Answer #2

Which is a distributed graph processing framework on top of Spark?

A	**GraphX**
B	MLlib
C	Spark SQL
D	Spark R

+919030485102

rganesh0203@gmail.com

https://topmate.io/rganesh_0203