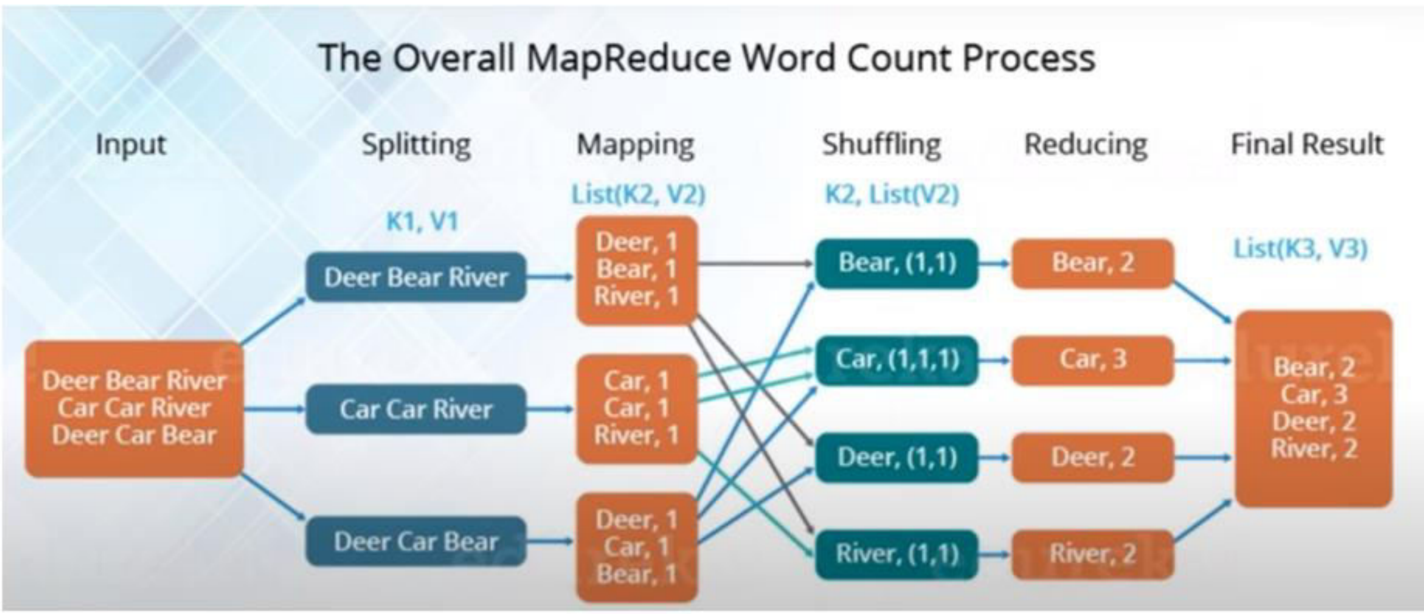
What is Spark?

1) Spark is **lightning-fast cluster computing** technology, designed for **fast** computation.
2) Spark is **primarily** based on **Hadoop**, supports **earlier model** to work **efficiently** and **offer** several **new computations** such as **interactive queries** and **stream processing**.
3) The **main** feature of **Spark** is its **in-memory cluster computing** that **increases** the **processing speed** of an **application**.
4) Spark supports **high-level APIs** such as **Java**, **Scala**, **Python** and **R**. It is basically built upon **Scala** language.
5) Spark **implements** the **processing** around **10 to 100** times **faster** than **MapReduce** because of its **in-memory** computing.

Why Spark:



So, why Spark than MapReduce:

1) In-Memory Computing:

Let's say there are **5 MR jobs** i.e. MR1, MR2, MR3, MR4 & MR5. So **MR1 brings data** from **HDFS** to **Memory** then **process** the **data** and **sends** the **result back** to **disk**. So total **10 disk seeks** i.e. **5** for **Read** & **5** for **Write** are required in this case.

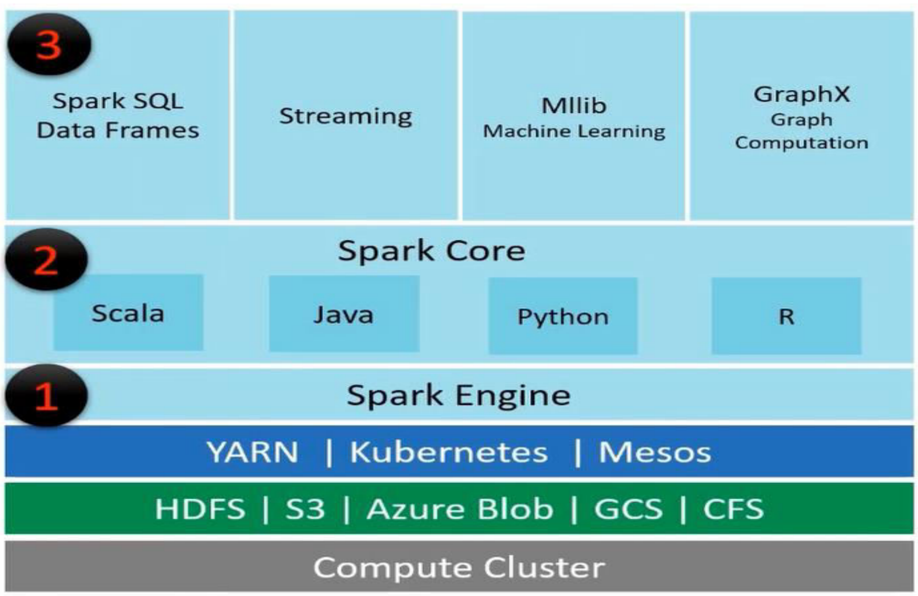
Spark brings **all data in memory** from the **disk**, **process** the **data** and **sends back** the **result** to **disk**. So **2 disk seeks** are required i.e. **1** for **Read** & **1** for **Write**.
Hence, Spark is fast as compared to MapReduce.

2) Support for Real Time Processing:
MapReduce supported only **batch** processing but Spark supports **batch**, **streaming** (near real time), graph processing etc.

3) One Single Framework:
a) To perform **stream** processing, we were using **Apache Storm / S4**.
b) For **interactive** processing, we were using **Apache Impala / Apache Tez**.
c) To perform **graph** processing, we were using **Neo4j / Apache Giraph**.
Hence there was **no powerful engine** in the industry, which can **process** the data **both** in **real-time** and **batch** mode. Also, there was a requirement that **one engine** can **respond** in **sub-second** and **perform in-memory processing** and that's where Spark emerged.

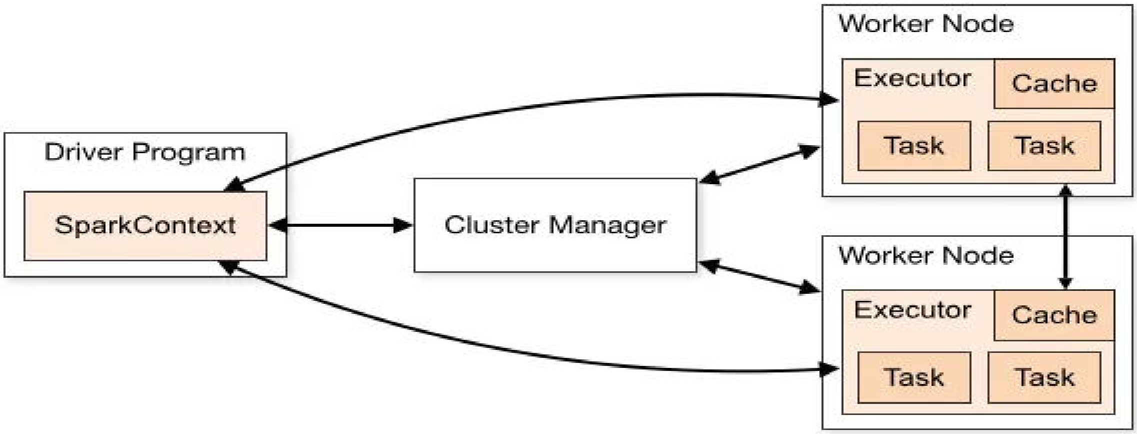
Apache Spark Ecosystem Components:

As we know, Spark offers faster computation and easy development. But it is not possible without following components of Spark.



Spark Architecture:

Apache **Spark** is an **open-source cluster computing** framework and when compared to Hadoop, Sparks' **performance** is upto **100 times faster** for **data** in **RAM** and **upto 10 times faster** for **data** in **storage.**



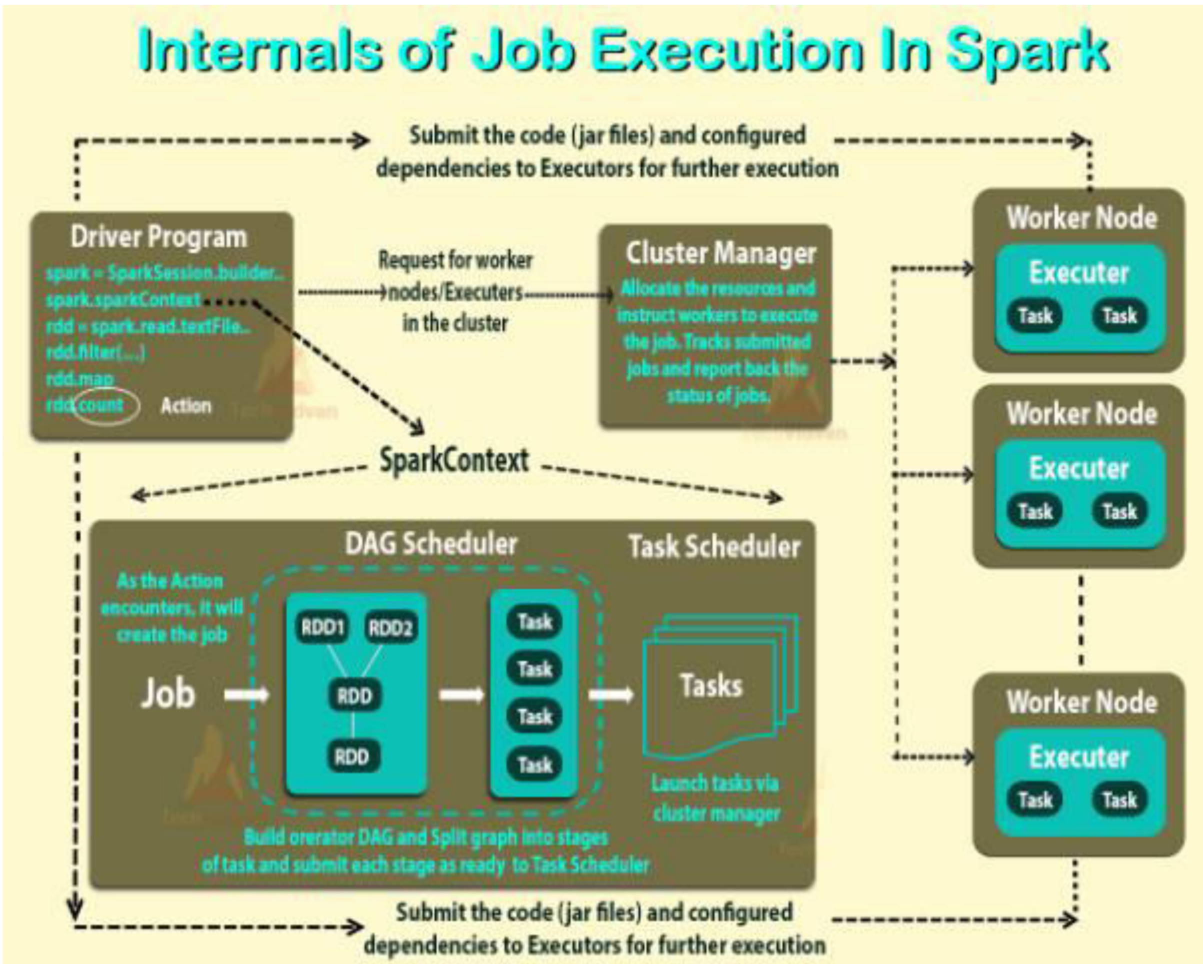
Spark uses **master/slave** architecture. As you can see in the figure, it has **one central coordinator** (**Driver**) that **communicates** with **many distributed workers** (**executors**). The **driver** and **each** of the **executors run** in their **own JVM**.

DRIVER:
The **driver** is the **process** where the **main method runs. First** it **converts** the **user program** into **tasks** and **after** that it **schedules** the **tasks** on the **executors**.

EXECUTORS:
Executors are **worker nodes processes** in charge of **running individual tasks** in a given Spark job. They are **launched** at the **beginning** of a **Spark application** and typically **run** for the **entire lifetime** of an **application**. Once they have **run** the **task** they **send** the **results** to the **driver**.

Runtime Architecture of Spark Application OR Execution Flow of Spark Application.
1) Apache Spark uses **Master-Slave** Architecture.
2) Client submits **user application** code. When an **application** is **submitted** the **driver implicitly converts** the **application code** containing **transformations** & **actions** into a **DAG**.
3) At this stage it **performs pipeline optimization** by **resolving Unresolved Logical Plans** into a **Physical Execution Plan** which **contains jobs**, **stages** & **tasks**.
4) Now the **driver talks** to **Cluster Manager** & **negotiates** for **resources**. CM **launches executors** on **worker nodes** on **behalf** of the **driver**.
5) Now the **driver sends** the **tasks** to these **executors** based on **data placement**.
6) When **executor starts** they **register** themselves with **drivers** so that the **driver** will have **complete view** of **all executors**.

7) Executors starts **executing** the **task assigned** by the **driver** & will be **monitored** by your **driver program**.
8) Driver **schedules** future **tasks. Tracks** the **location** of **cached data** to **schedule** future **tasks**.
9) Driver **provides** all of the above **information** of **running application** on **Spark Web UI** on **port** http://localhost:4040
10) When the **driver's sc stop method** is **called** it will **terminate** all the **executors** & **release resources** from **CM**.



PySpark:

- **PySpark** is a **Python API** for **Apache Spark** to **process larger datasets** in a **distributed cluster**. It is **written** in **Python** to **run** a **Python application** using **Apache Spark capabilities.**
- **Spark** basically is written in **Scala**, and due to its adaptation in industry, its equivalent **PySpark API** has been released for **Python Py4J**.
- **Py4J** is a **Java library** that is **integrated** within **PySpark** and **allows python** to **dynamically interface** with **JVM objects**, hence to **run PySpark** you also need **Java** to be **installed** along with **Python**, and **Apache Spark**.

PySpark Modules:

- PySpark RDD (pyspark.RDD)
- PySpark DataFrame and SQL (pyspark.sql)
- PySpark Streaming (pyspark.streaming)
- PySpark MLib (pyspark.ml, pyspark.mllib)
- PySpark GraphFrames (GraphFrames)

Features of Spark:

- In-memory computation & distributed processing framework
- Can be used with many CM (Yarn, Mesos, and Kubernetes)
- Fault-Tolerant
- Immutable
- Lazy Evaluation
- Cache & Persistence
- Inbuild-Optimization when using DataFrame
- Supports ANSI SQL

Advantages of Spark:

- Spark is a **general-purpose**, **in-memory**, **distributed processing engine** that **allows** you to **process data efficiently** in a **distributed** environment.
- Applications **running** on Spark are **100x faster** than **traditional** systems.
- Using **Spark** we can **process** data from **HDFS**, **AWS S3**, and many **file systems**.
- Spark is also used to **process real-time** data using **Streaming** and **Kafka**.
- Using **Spark streaming** you can also **stream files** from the **file system** and also **stream** from the **socket**.
- Spark **natively** has **machine learning** and **graph libraries**.

What is SparkContext in Spark?

1) SparkContext is the **entry point** of Apache **Spark functionality**.
2) The **most important step** of any **driver application** is to **generate SparkContext**.
3) It **allows** your **application** to **access cluster** with the help of **Resource Manager**.
4) To create **SparkContext**, first **SparkConf** should be made.
5) The **SparkConf** has a **configuration parameter** that our **driver application** will **pass** to **SparkContext**.

How to Create SparkContext?

If you want to create **SparkContext**, first **SparkConf** should be made. The **SparkConf** has a **configuration parameter** that our **driver application** will **pass** to **SparkContext**.

Once the **SparkContext** is **created**, it can be **used** to **create RDDs**, **broadcast variable**, **accumulator** and **run jobs**. All these things can be **carried out** until SparkContext is **stopped**.

Let's see how to create SparkContext using SparkConf:

1) sparkConf = SparkConf ( ) \
. setAppName ("WordCount") \
.setMaster ("local") → **Create conf object**
2) sc = SparkContext (conf=sparkConf) → **Create SparkContext object**

Stopping SparkContext:

Only **one SparkContext** may be **active per JVM**. You must **stop** the **active one** before creating a **new one** as shown:
sc.stop ( )
It will display message: ***INFO SparkContext: Successfully stopped SparkContext***

2) SparkSession:

spark = SparkSession.builder.appName("WordCount").master("local [3]").getOrCreate( )
spark.sparkContext ( )

********************************WordCount Code************************************

1) REPL Mode:

****

2) Jupyter Notebook:

****

3) PyCharm:

****

RDD:

- **Resilient Distributed Dataset (aka RDD)** is the **primary data abstraction** in Apache Spark and the **core** of **Spark** i.e. referred as "**Spark Core**".
- It is **immutable collection** of **objects** & **lazily evaluated**.
- Each **dataset** in **RDD** is **divided** into **logical partitions**, which may be **computed** on **different nodes** of the **cluster**.

RDD Benefits:

1) In-Memory Processing:

- Spark **loads** the **data** from **disk** and **process in memory** and keeps the **data in memory**, this is the **main difference** between **Spark** and **MapReduce** (**I/O intensive**).
- We can also **cache/persists** the **RDD** in **memory** to **reuse** the **previous computations**.

2) Immutability

- Spark **RDD's** are **immutable** in nature meaning, once **RDDs** are **created** you **cannot modify**.
- When we **apply transformations** on **RDD**, Spark **creates** a **new RDD** and **maintains** the **RDD Lineage**.

3) Fault Tolerance:

- Spark **operates** on **fault-tolerant data** stored on HDFS, S3 etc. hence any **RDD** operation **fails** it **automatically reloads** the **data** from other **partitions**.
- When Spark **application** is **running** on a **cluster** any **task failures** are **automatically recovered** for a **certain number** of **times** (**as per the configuration**) and **finish** the **application seamlessly**.

4) Lazy Evolution:

Spark **does not evaluate** the **RDD transformations** as they **appear/encountered** by driver **instead** it **keeps** the **all transformations** as it **encounters** (**DAG**) and **evaluates** the **all transformation** when it sees the **first RDD action**.

5) Partitioning:
When you **create RDD** from **data** by default it **partitions** the **elements** in a **RDD**. By default it **partitions** to the **number of cores** available.

RDD Limitations:

- Spark **RDDs** are **not much** suitable for **applications** that make **updates** to the **state store** such as **storage systems** for a **web application**.
- For these applications, it is **more efficient** to **use systems** that **perform traditional update, logging** and **data checkpointing** such as **databases**.
- The **goal** of **RDD** is to **provide** an **efficient programming model** for **batch analytics**.

There are three ways to create RDDs in Spark:

- **Parallelizing** via **collections** in driver program.
- Creating a **dataset** in an **external storage system** (e.g. HDFS, HBase, and Shared FS).
- Creating RDD from **existing RDDs**.

1) Parallelized collection (parallelizing):

a) Create RDD from parallelize:

Spark sets **number** of **partition** based on our **cluster**. But we can also **manually set** the **number** of **partitions**. This is **achieved** by **passing number** of **partition** as **second parameter** to **parallelize**.
e.g. sc.parallelize (data, 5), here we have **manually** given **number** of **partition** as **5**.

b) Create RDD with partition:

2) External Datasets (Referencing a dataset):
To **create RDD** from **external text file** we can use sc **textFile** method.

External Datasets (Referencing a dataset):

3) Creating RDD from existing RDD:

- Transformation **mutates one RDD** into **another RDD**, thus **transformation** is the way to **create** an **RDD** from **already existing RDD**.
- Transformation **acts** as a **function** that **intakes** an **RDD** and **produces one**.
- The **input** RDD **does not** get changed, because **RDDs** are **immutable** in nature.

Creating RDD from existing RDD:

Spark RDD Operations: RDD in Spark supports two types of operations:

- Transformation
- Actions

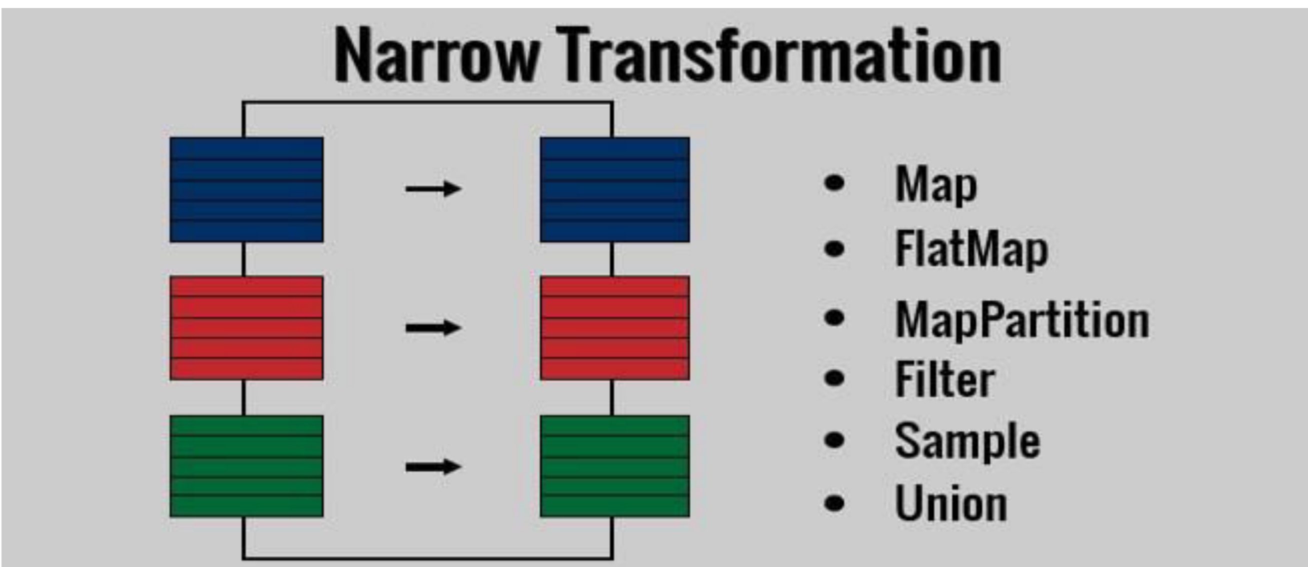
1) Transformations:

- **Transformations** are **operation** which will **transform** your **RDD data** from **one form** to **another**.
- And when you **apply** this **operation** on any **RDD**, you will get a **new RDD** of **transformed data** (RDDs in Spark are immutable).

There are two kinds of transformations: narrow transformation, **wide** transformation.

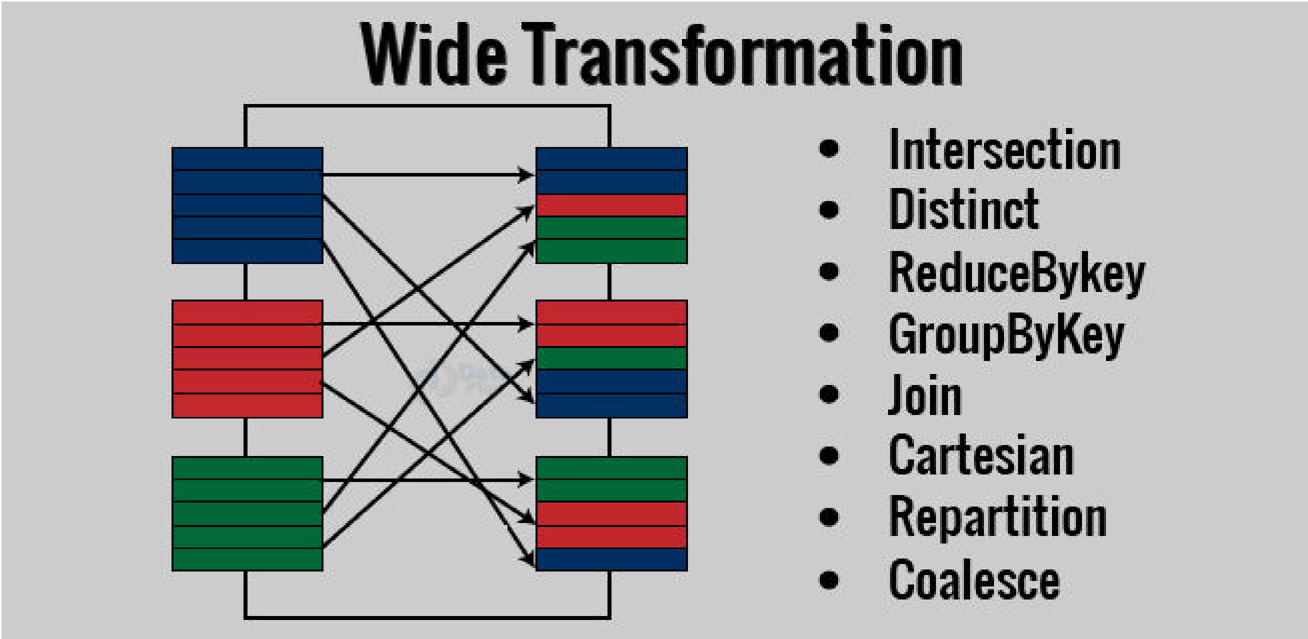
a) Narrow Transformations:

- In **Narrow** transformation, **all the elements** that are **required** to **compute** the **records** in a **single partition live** in the **single partition** of **parent RDD**.



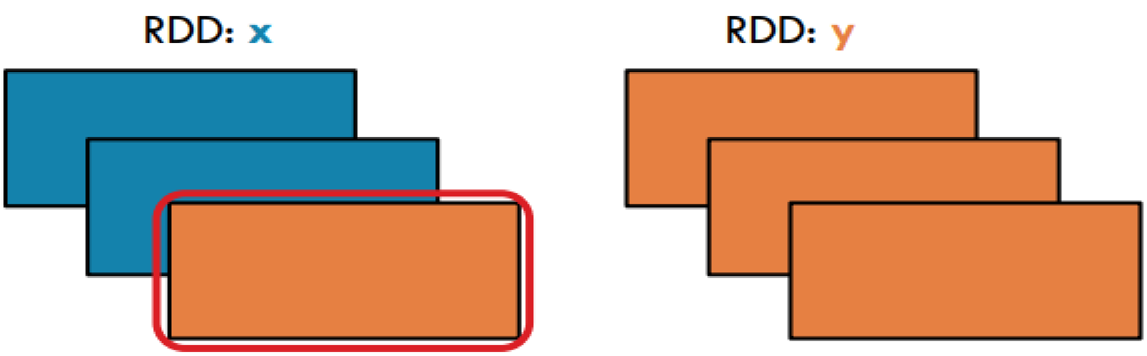
b) Wide Transformations:

- In **wide** transformation, **all the elements** that are **required** to **compute** the **records** in the **single partition** may live in **many partitions** of **parent RDD**.
- Wide transformations are also known as **shuffle transformations** because they **may** or **may not depend** on a **shuffle**.



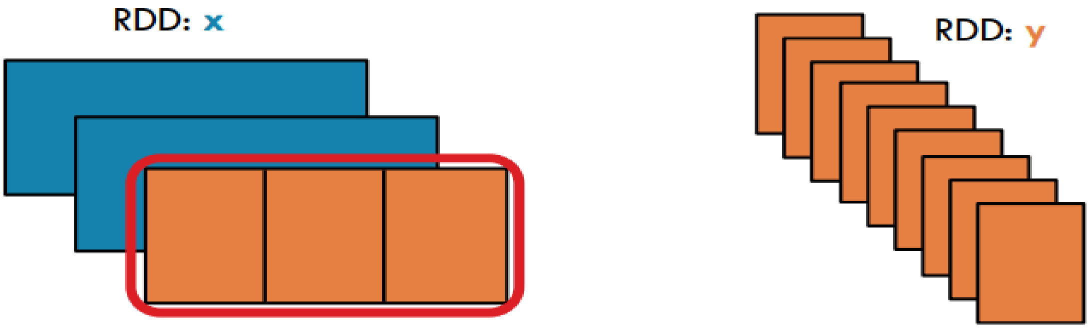
a) map (func):
Returns a new RDD by applying a function to each element of this RDD

E.g. in RDD {1, 2, 3, 4, 5} if we apply rdd.map (lambda x: (x+2)) we will get the result as (3, 4, 5, 6, 7).



E.g.
```
a = sc.parallelize([1,2,3,4,5])
result = a.map(lambda x: (x, x+2))
result.collect()
for element in result.collect():
   print(element, end="")
```

b) flatMap ( ):
- **Returns** a **new RDD** by **first** applying a **function** to **all elements** of this **RDD**, and then **flattening** the **results.**
- The **key difference** between **map** ( ) and **flatMap** ( ) is map ( ) **returns only one** element, while flatMap ( ) **can return a list of multiple elements**.



E.g.

```
a = sc.parallelize([1,2,3,4,5])
result = a.flatMap(lambda x: (x, x**2))
result.collect()
```

OR

```
sameline=True
for i in result.collect():
   print(i, end=' ')
   if not sameline:
      print()
   sameline=not sameline
```

Note: In above code, flatMap ( ) function splits each line when space occurs.

c) filter (func):

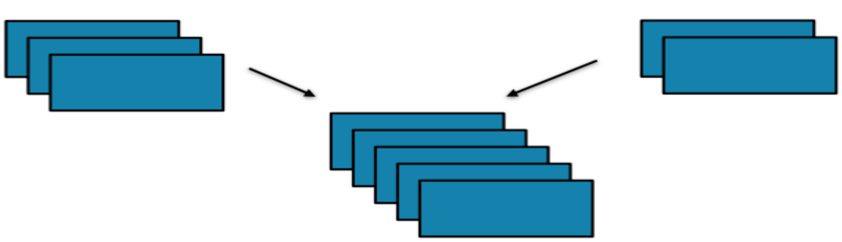
- **filter ( )** function **returns** a **new RDD**, containing **only** the **elements** that **meets condition**.
- It is a **narrow** operation because it **does not shuffle** data from **one partition** to **many partitions**.

E.g.

```
a = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
result = a.filter(lambda x: x % 2 == 0)
print(result.collect())
```

d) union (dataset):

With **union ( )** function, we **get** the **elements** of **both** the RDD in **new RDD**. The key rule is that the **two RDDs** should be of the **same type**. It can have **duplicates** also.



E.g.
a = sc.parallelize([1,2,3,4,5])
b = sc.parallelize([1,2,2,3,3])
a.union(b).collect()

e) intersection ( ):
intersection ( ) function, we get **only** the **common element** of **both** the **RDD** in **new RDD**. The key rule is that the **two RDDs** should be of the **same type**.

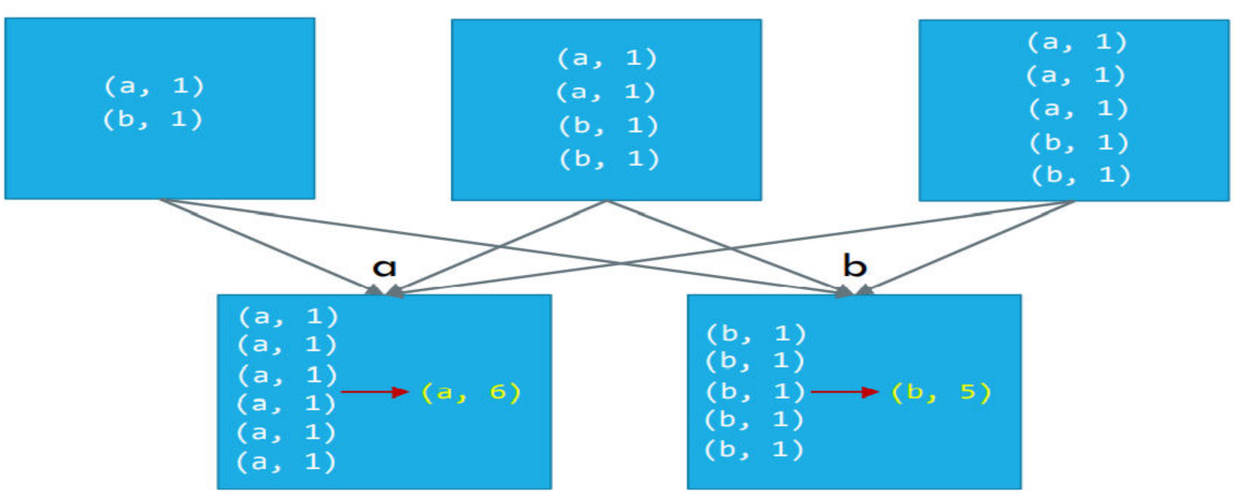
E.g.
a = sc.parallelize([1,2,3,4,5])
b = sc.parallelize([1,2,2,3,3])
a.intersection(b).collect()

f) distinct ( ):
It **returns** a **new dataset** that **contains** the **distinct elements** of the **source dataset**. It is helpful to **remove duplicate** data.

E.g.
a = sc.parallelize([1,2,2,3,4,4,4,5])
a.distinct( ).collect()

g) groupByKey ():

- groupByKey function takes key-value pair (K, V) as an input and produces RDD with key and list of values.
- This function require to shuffle all data with same key to a single partition unless your source RDD is already partitioned by key. And this shuffling makes this transformation as a wider transformation.
- groupByKey can cause disk problems as data is sent over the network and collected on the reduce workers.



E.g.

```
x = sc.parallelize([('A', 2), ('B', 1), ('B', 5), ('A', 1), ('B', 10)])
result = x.groupByKey()
print(result.collect())
for j in result.collect():
    for i in j[1]:
        print(i)
```

OR

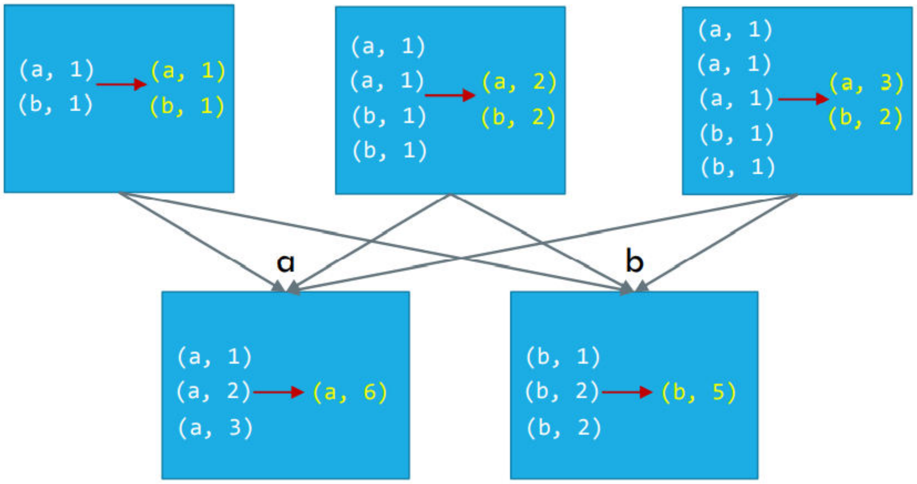
```
for j in result.collect():
   out = ''
   for i in j[1]:
      out = out + str(i) + ','
   print(out[:-1])
```

OR

```
print(list((j[0], list(j[1])) for j in result.collect()))
```

h) reduceByKey:

- Data is **combined** at **each partition**, only **one output** for **one key** at **each partition** is **sent over network**.
- **reduceByKey** is a **transformation operation** in Spark hence it is **lazily evaluated**.
- Before **sending data across** the **partitions**, it also **merges** the **data locally** using the same **associative** function for **optimized data shuffling**.
- It accepts a **Commutative** and **Associative function** as an **argument**.
  a) The **parameter function** should have **two arguments** of the **same data type**.
  b) The **return type** of the **function** also **must** be **same** as **argument types**.



E.g.

```
words = sc.parallelize(["Saif", "Ram", "Mitali", "Aniket", "Ram", "Ram", "Aniket"])
wordCount = words.map(lambda word: (word, 1)).reduceByKey(lambda a,b: a + b)
print(wordCount.collect())
```

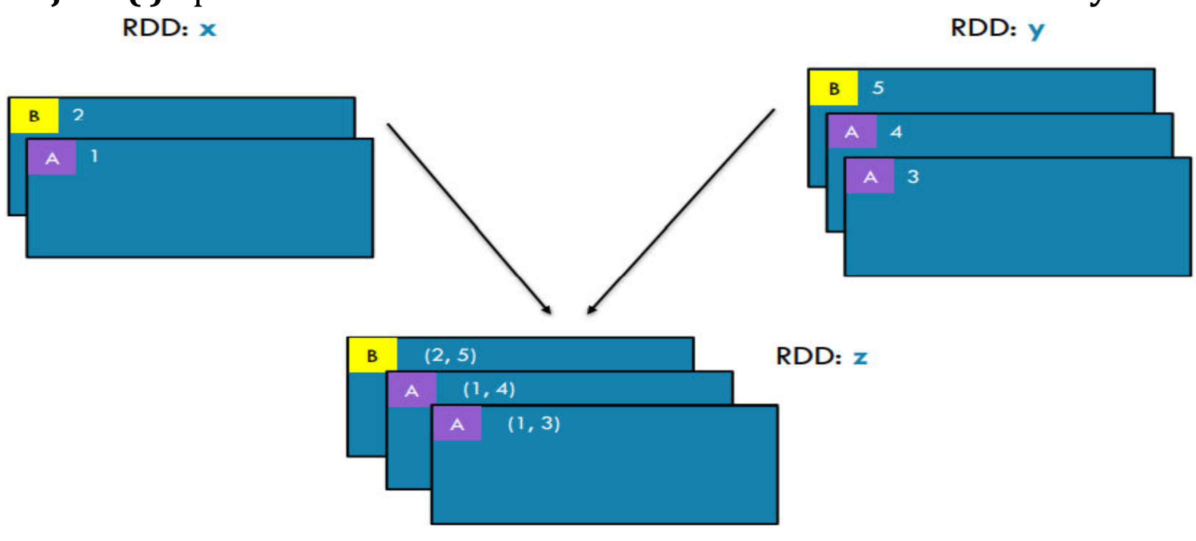
Note:
The above code will **parallelize** the **String**.
It will then **map each word** with **count 1**, then **reduceByKey** will **merge** the **count** of **values** having the **similar key**.

i) sortByKey ( ):
When we **apply** the **sortByKey ( )** function on a **dataset** of **(K, V) pair**, the **data** is **sorted according** to the **key K** in **another RDD.**

E.g.
```
words = sc.parallelize(["Saif", "Ram", "Mitali", "Aniket", "Ram", "Ram", "Aniket"])
wordCount = words.map(lambda word: (word, 1)).reduceByKey(lambda a,b: a + b)
print(wordCount.sortByKey().collect())
print(wordCount.sortByKey(False).collect())
```

Note: In above code, **sortByKey ( )** transformation **sort** the **RDD data** into **Ascending order** of the **Key.**

j) join ():
- It **combines** the **fields** from **two rdd** using **common values.** Join ( ) operation in Spark is defined on **pair-wise RDD.**
- **Pair-wise RDDs** are **RDD** in which **each element** is in the **form** of **tuples**.
- Where the **first element** is **key** and the **second element** is the **value**.
- The **join ( )** operation **combines two data sets** on the **basis** of the **key**.



E.g.
```
a = sc.parallelize([('C', 4), ('B', 3), ('A', 2), ('A', 1)])
b = sc.parallelize([('A', 8), ('B', 7), ('A', 6), ('D', 5)])
print(a.join(b).collect())
```

Types of joins:

- join
- leftOuterJoin
- rightOuterJoin
- fullOuterJoin
- Cartesian

k) coalesce ( ):

- In **coalesce ( )** we use **existing partition** so that **less data** is **shuffled.** To avoid **full shuffling** of **data** we **use coalesce ( )** function. Using this we can **reduce** the **number** of the **partition**.
- Creates **unequal sized partitions**.

l) repartition ( ):

- Used to **increase** or **decrease** the **number** of **partitions**.
- A **network shuffle** will be **triggered** which **can increase data movement**.
- Creates **equal sized partitions.**

#repartition & coalesce

```
a = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
b = a.getNumPartitions()
print(b)
c = a.glom().collect()
print(c)

d = a.coalesce(1)
print("After Coalesce: "+str(d.getNumPartitions()))
e = a.repartition(5)
print("After Repartition: "+str(e.getNumPartitions()))
```

2) Actions:

- When **action** is **triggered new RDD** is **not formed** like **transformations**. Thus, **actions** are **operation** that **gives non-RDD values.**
- The **values** of **action** are **sent** to **drivers** or to the **external storage system**.
- It brings **laziness** of **RDD** into **motion.**
- An **action** is **one** of the **ways** of **sending data** from ***executer*** to the ***driver****.*

a) count ( ):

count ( ) returns the **number** of **elements** in RDD.

```
a = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
a.count( )
```

b) collect ( ):
collect ( ) is the **common** and **simplest** operation that **returns** our **entire RDDs** content to **driver** program.
a.collect()

c) take (n):
take (n) **returns n number** of **elements** from **RDD**.
a.take(5)

d) top ( ):
If **ordering** is **present** in our **RDD**, then we can **extract top elements** from our **RDD** using **top ( )**. Action **top ( )** use **default ordering** of **data**.
a.top(4)

e) countByValue ():
- **No Key, Value** is required.
- It **returns** the **count** of **each unique value** in an **RDD**.
- **Care** must be **taken** to **use this API** since it **returns** the **value** to **driver program** so it's **suitable** only for **small values**.

E.g.
a = sc.parallelize(["Saif", "Mitali", "Ram", "Ram", "Ram", "Mitali"])
a.countByValue()
a.countByValue().keys()
a.countByValue().values()
Other way of writing code:
b = dict(a.countByValue())
print(b)
c = list(b.keys())
d = list(b.values())
print(c)
print(d)
Note: reduceByKey return **Array** whereas **countByValue** returns **Map**.

f) countByKey ():
- It is an **action** operation which **returns** (**key**, **noofkeycount**) **pairs**.
- It **counts** the **value** of **RDD** consisting of **two components tuple** for each **distinct key**.
- It actually **counts** the **number** of **elements** for **each key** and **return** the **result** to the driver as **lists** of **(key, count) pairs**.

E.g.

```
x = sc.parallelize([('A', 2), ('A', 1), ('C',1), ('B',5)])
x.countByKey()
x.countByKey().values()
```

g) reduce ( ):

- **reduce ( )** function takes **two elements** as **input** from the **RDD** and then **produces** the **output** of the **same type** as that of the **input elements**.
- We can **add** the **elements** of **RDD**, **count** the **number** of **words**.
- It accepts **commutative** and **associative operations** as an **argument**.

E.g.

```
a = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
a.reduce(lambda a, b: a + b)
```