**What is Spark?**

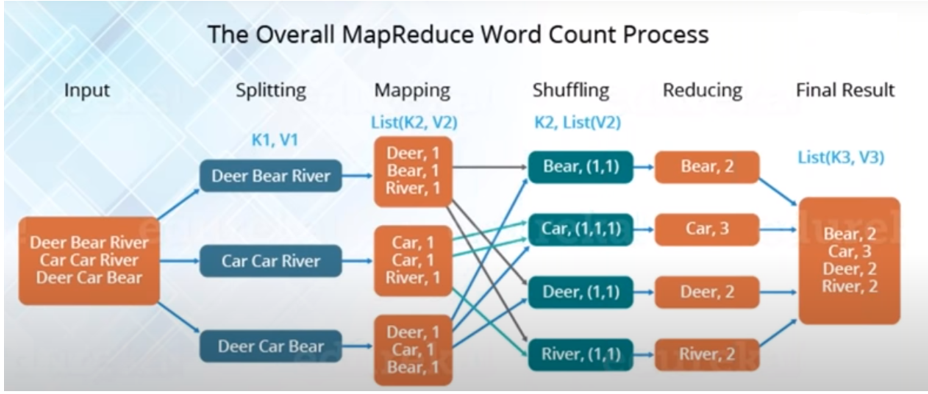
1) Spark is **lightning-fast cluster** **computing** technology, designed for **fast** computation for distributed parallel processing over large datasets on different nodes of the cluster.

2) The **main** feature of **Spark** is its **in-memory cluster computing** that **increases** the **processing** **speed** of an **application**.

3) Spark supports **APIs** such as **Java**, **Scala**, **Python** and **R**. It is basically built upon **Scala** language.

4) 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:**

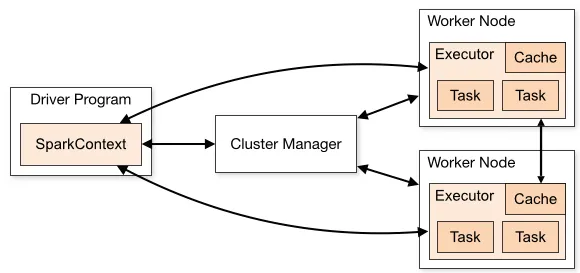
1) In-Memory Computing

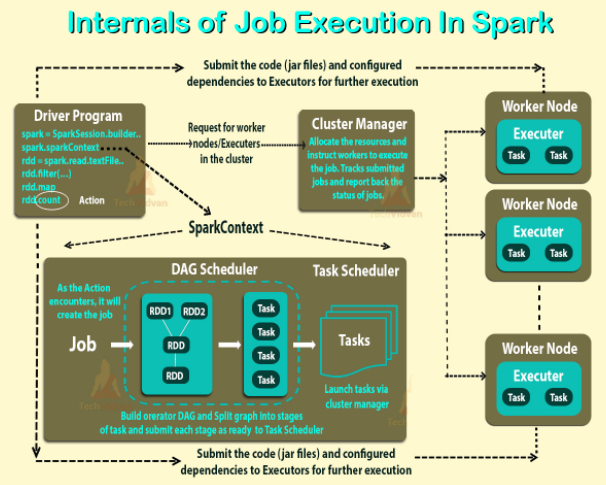
2) Support for Real Time Processing

3) One Single Framework

**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 **DISK.**





Spark uses **master/slave** architecture.

**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 (series of RDDs)**.

**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 Modules:**

* PySpark RDD (pyspark.RDD)
* PySpark DataFrame and SQL (pyspark.sql)
* PySpark Streaming (pyspark.streaming)
* PySpark MLib ([pyspark.ml](https://spark.apache.org/docs/latest/api/python/pyspark.ml.html), pyspark.mllib)
* PySpark GraphFrames (GraphFrames)

**Features of Spark:**

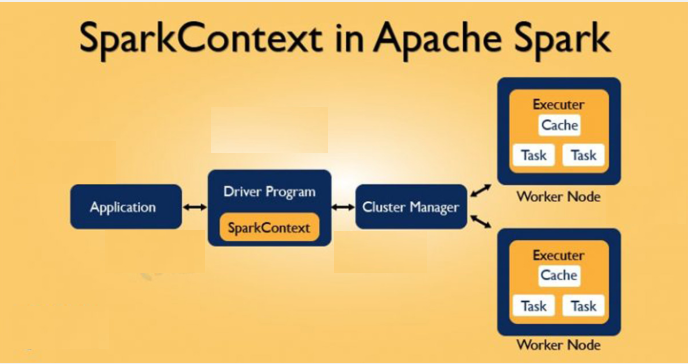
* In-memory computation & distributed processing framework
* Can be used with many CM (YARN, Mesos, and Kubernetes)
* Fault-Tolerant
* Immutable
* Lazy Evaluation
* Cache & Persistence
* Inbuild-Optimization when using DataFrame
* Supports ANSI SQL

**What is SparkContext in Spark?**

**1)** **SparkContext** is the **entry point** of Apache **Spark** **functionality**.

**2)** To create **SparkContext**, first **SparkConf** should be made.

**3)** 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**

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 [\*]”).getOrCreate( )

spark.sparkContext ( )

**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 Limitations:**

* RDDs does **not** have **in-built optimization**.
* RDD does **not** have **schema**.

**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:**

data = [1,2,3,4,5,6,7,8,9,10,11,12]

rdd = sc.parallelize (data)

rdd.collect ( )

print (rdd.take(20))

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:**

data = [1,2,3,4,5,6,7,8,9,10,11,12]

rdd = sc.parallelize(data)

print("Initial Partition Count:"+str(rdd.getNumPartitions()))

**2) External Datasets (Referencing a dataset):**

To **create** **RDD** from **external** **text** **file** we can use sc **textFile** method.

**External Datasets (Referencing a dataset):**

readFile = sc.textFile("file:///home/saif/LFS/datasets/emp.txt")

readFile.collect()

**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:**

print("Initial Partition Count:"+str(readFile.getNumPartitions()))

repartition\_Rdd = readFile.repartition(2)

print("Re-partition count:"+str(repartition\_Rdd.getNumPartitions()))

coalesce\_Rdd = readFile.repartition(3)

print("Re-partition count:"+str(coalesce\_Rdd.getNumPartitions()))

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

* Transformation
* Actions

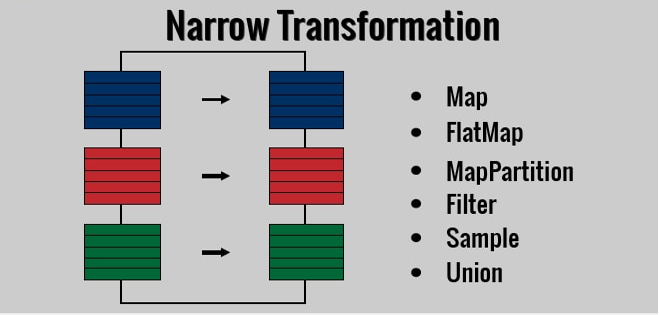
**1) Transformations:**

* **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.

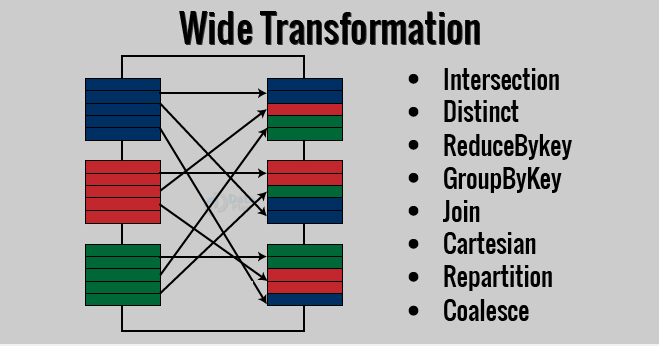
**a) Narrow Transformations:**

* 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**.

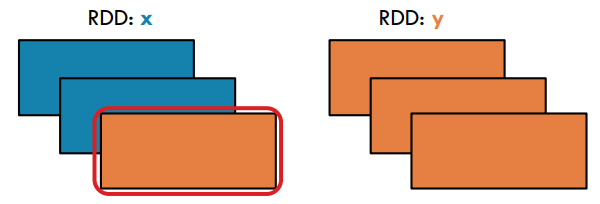


**a) map (func):**

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))

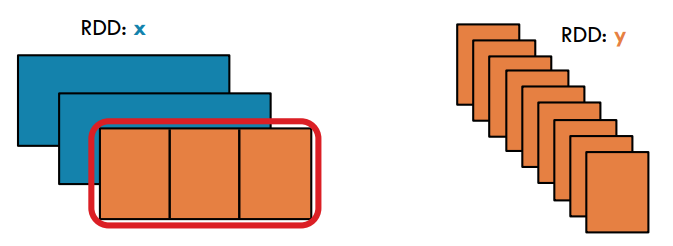
result.collect()

for element in result.collect():

print(element, end="")

**b) flatMap ( ):**

* **Returns** a **new** **RDD** by **first** applying a **map** **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.**

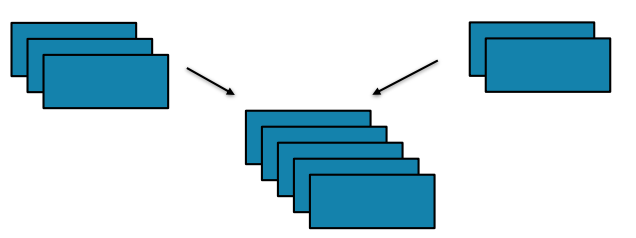
a = sc.parallelize([1,2,3,4,5,6,7,8,9,10])

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.

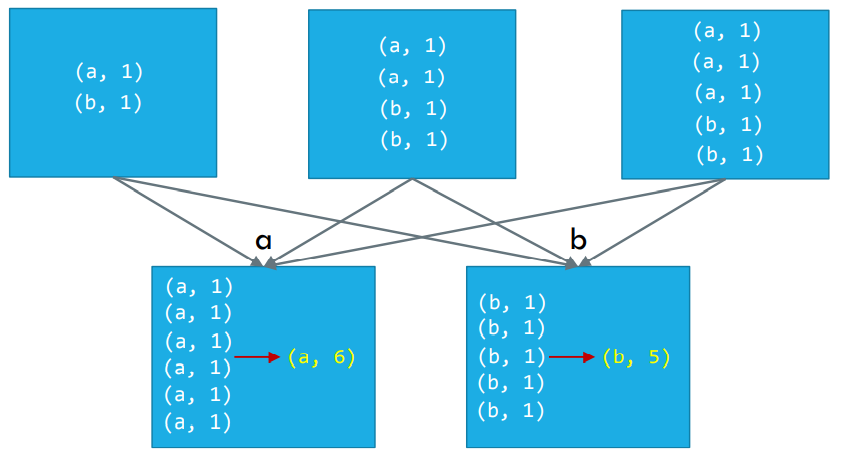
**E.g.**

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())

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

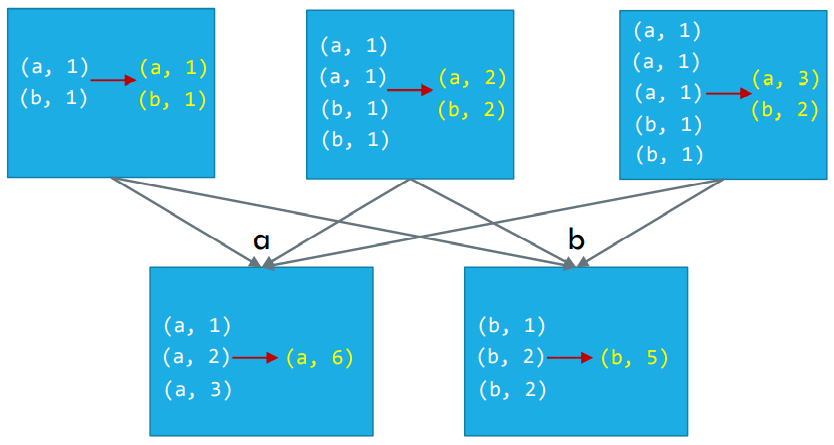
result = grouped\_rdd.mapValues(list).collect()

print(result)

**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**.

1. The **parameter** **function** should have **two** **arguments** of the **same** **data** **type**.
2. 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**.

**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 **e*xecuter*** 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) 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)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***WordCount Code**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**1) REPL Mode:**

**A) Read data from text file:**

readFile = sc.textFile("hdfs://localhost:9000/user/saif/HFS/Input/wordcount.txt")

**Validations:**

**a)** type(readFile)

**b)** readFile.collect()

['Saif Ram Aniket Saif', 'Mitali Saif Ram Aniket', 'Mitali Ram Aniket Saif']

**c)** type(readFile.collect())

<class 'list'>

**d)** for i in readFile.collect():

print(i)

Saif Ram Aniket Saif

Mitali Saif Ram Aniket

Mitali Ram Aniket Saif

**B) Split each line into words:**

splitWords = readFile.flatMap(lambda line: line.split(" "))

splitWords.collect()

for i in splitWords.take(5):

print(i)

**C) Assign the word with Value as 1:**

wordAssign = splitWords.map(lambda word: (word, 1))

**D) Count the occurrence of each word:**

wordCount = wordAssign.reduceByKey(lambda a, b: a + b)

**F) Save the counts to output:**

wordCount.saveAsTextFile("hdfs://localhost:9000/user/saif/HFS/Output/wordcount\_repl\_op")

wordCount.getNumPartitions()

hdfs dfs -cat /user/saif/HFS/Output/wordcount\_repl\_op/part\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**2) Jupyter Notebook:**

import findspark

findspark.init()

from pyspark import SparkConf

from pyspark import SparkContext

conf = SparkConf()

sc = SparkContext(conf=conf)

conf.setMaster("local").setAppName("WordCount")

# conf.get("spark.master")

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

# sc.master

# sc.appName

words = sc.textFile("/user/saif/HFS/Input/wordcount.txt")\

.flatMap(lambda line: line.split(" "))

wordCounts = words.map(lambda word: (word, 1)).reduceByKey(lambda a,b:a +b)

wordCounts.saveAsTextFile("/user/saif/HFS/Output/wordcount\_jupyter\_op")

!hdfs dfs -cat /user/saif/HFS/Output/wordcount\_jupyter\_op/part\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*