Python Basics

Normal Vs Lambda Function

Defining Normal function

```
def my_sum(x,y):
        return(x+y)
```

Calling a Normal function -

```
total = my_sum(5,7)
```

Lambda Function

- Equivalent to anonymous functions in Scala, where the code body / functionality is directly embedded in-line and passed as parameter rather than explicitly defining and then calling the function.
- Such functions are used when a specific functionality is required to be run not more than once.

Python map function Vs Spark map transformation

Say you want the sum of numbers in a list : my_list = [5,7,8,2,5,9]

- **The normal python map way** - function works on a single machine which is the local/gateway node.

```
//defining a normal function
def sumlist(x):
        total = 0
        for i in x:
                total = total + i
                return total
//calling a normal function
sumlist(my_list)
```

- **The distributed spark way** - map transformation works on a cluster of nodes in a distributed way.

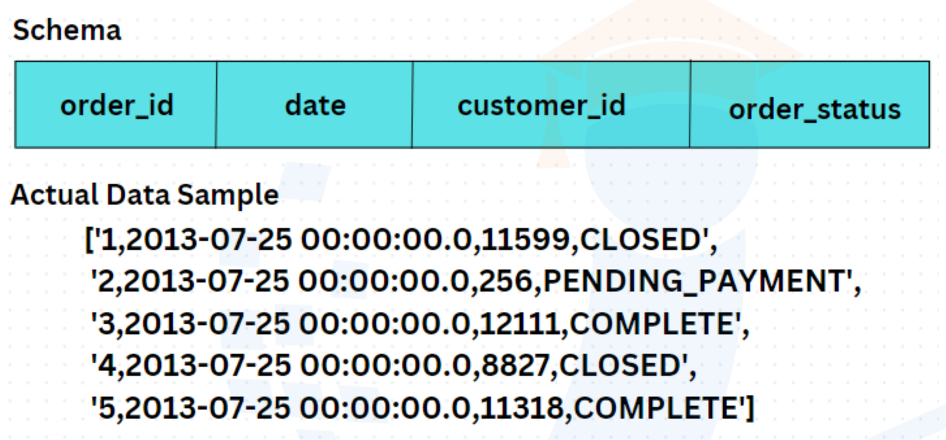
```
//importing the reduce functionality
from functools import reduce
```

```
        //defining lambda function to sum the numbers in-line and passing as a
parameter
        reduce(lambda x,y : x+y, my_list)
```

Higher order functions - Functions that accept another function as a parameter or provide another function as an output. Ex - reduce, map

Pyspark Use case 1 :

Consider the Business is requiring to know the following from their orders data with the schema



Schema

order_id	date	customer_id	order_status

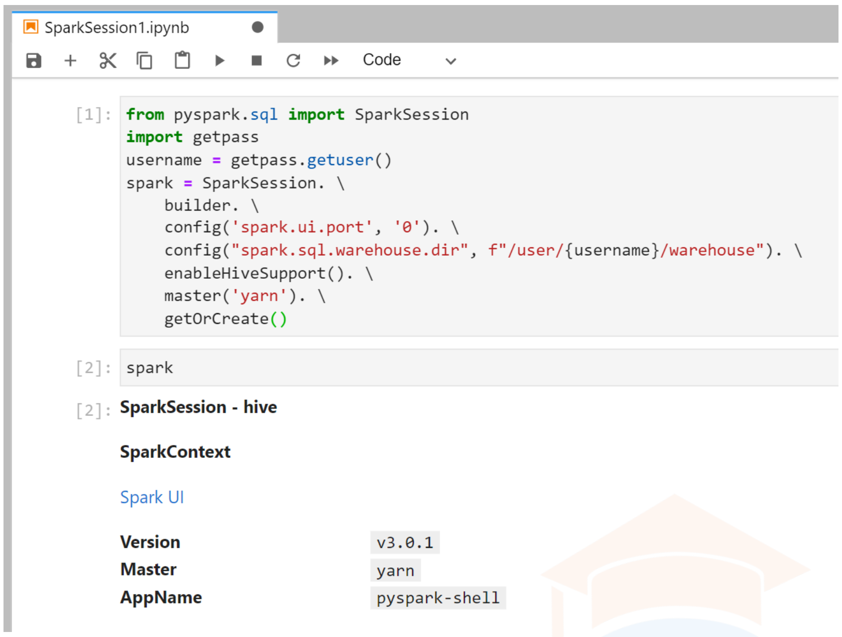
Actual Data Sample

```
['1,2013-07-25 00:00:00.0,11599,CLOSED',
 '2,2013-07-25 00:00:00.0,256,PENDING_PAYMENT',
 '3,2013-07-25 00:00:00.0,12111,COMPLETE',
 '4,2013-07-25 00:00:00.0,8827,CLOSED',
 '5,2013-07-25 00:00:00.0,11318,COMPLETE']
```

1. **Orders in each category (COMPLETE, PENDING_PAYMENT, CLOSED..)**
2. **Premium Customers (Top 10 customers who placed most of the orders)**
3. **Distinct customers who placed atleast 1 order**
4. **Customers having maximum number of CLOSED orders**

Creating a Spark Session (boilerplate code) :

```
from pyspark.sql import SparkSession
import getpass
username = getpass.getuser()
spark = SparkSession. \
    builder. \
    config('spark.ui.port', '0'). \
    config("spark.sql.warehouse.dir", f"/user/{username}/warehouse"). \
    enableHiveSupport(). \
    master('yarn'). \
    getOrCreate()
```



```
[1]: from pyspark.sql import SparkSession
     import getpass
     username = getpass.getuser()
     spark = SparkSession. \
         builder. \
         config('spark.ui.port', '0'). \
         config("spark.sql.warehouse.dir", f"/user/{username}/warehouse"). \
         enableHiveSupport(). \
         master('yarn'). \
         getOrCreate()

[2]: spark

[2]: SparkSession - hive

     SparkContext

     Spark UI

     Version      v3.0.1
     Master       yarn
     AppName      pyspark-shell
```

Loading Data into RDD :

orders_rdd =
spark.sparkContext.textFile(“/public/trendytech/retail_db/orders/*”)

Applying Transformations :

- Orders in each Category

mapped_rdd = orders_rdd.map(lambda x : (x.split(",")[3],1))

Transformation to aggregate the values based on the key

reduced_rdd = mapped_rdd.reduceByKey(lambda x,y : x+y)

Sort the values using sortBy

reduced_sorted = reduced_rdd.sortBy(lambda x : x[1], False)

[**Note** : reduce Vs reduceByKey

Reduce gives a single value after aggregation.

ReduceByKey gives aggregated result for each distinct key]

- Find the Premium Customers

```
customers_mapped = orders_rdd.map(lambda x : (x.split(“,”)[2],1))

customers_aggregated = customers_mapped.reduceByKey(lambda x,y : x+y)
```

Gives top ten premium customers who had placed most orders

```
customers_aggregated.sortBy(lambda x : x[1], False).take(10)
```

- Distinct Customers who placed atleast 1 order

```
distinct_customers = orders_rdd.map(lambda x : (x.split(“,”)[2]).distinct()

distinct_customers.count()
```

- Customers having maximum number of CLOSED orders

```
filtered_orders = orders_rdd.filter(lambda x : x.split(“,”)[3] == ‘CLOSED’))

filtered_mapped = filtered_orders.map(lambda x : (x.split(“,”)[2],1))

filtered_aggregated = filtered_mapped.reduceByKey(lambda x,y : x+y)

filtered_sorted = filtered_aggregated.sortBy(lambda x : x[1], False)
```

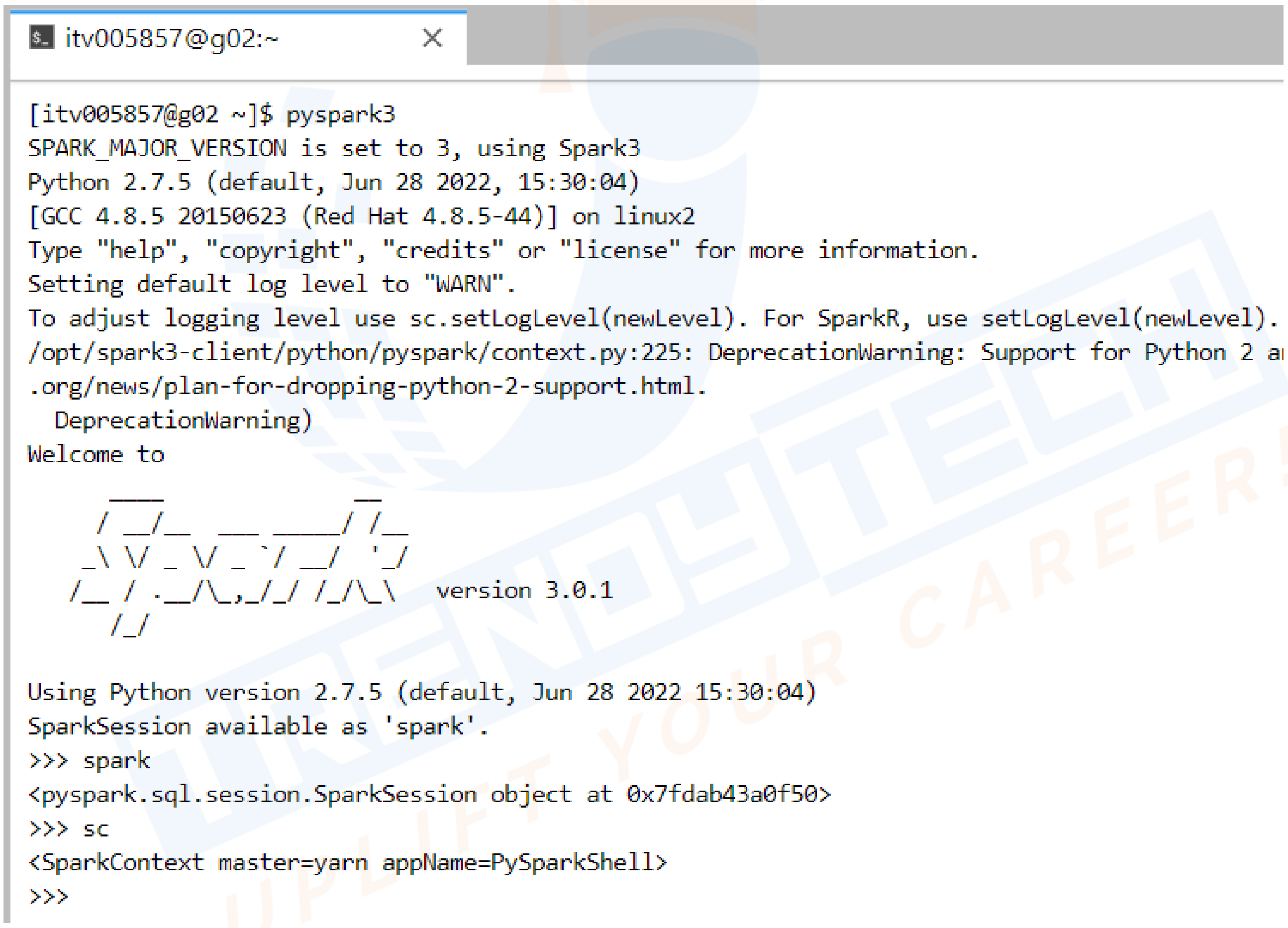
map No.of output rows = No.of input rows Ex : 1000 input rows (on map transformation) => 1000 output rows	**reduce** A single output for all the input rows Ex : 1000 input rows (on reduce transformation) => 1 output	**reduceByKey** No.of output rows is equal to the No.of distinct keys Say if there are 100 distinct keys Ex : 1000 input rows (on reduceByKey transformation) => 100 output rows
filter No.of output rows <= input rows Ex : 1000 input rows (on filter transformation) <= 1000 output rows	**sortBy / sortByKey** No.of output rows = No.of input rows sortBy - sorts the values in ascending(by default) or descending order sortByKey - sorts based on Key	**distinct** No.of output rows < No.of input rows gives unique / distinct values as output

Spark Core APIs - RDD

Alternate environments where spark code can be developed and executed other than Jupyter Notebook

In the terminal : execute the following command to start a pyspark execution environment which has an already created SparkSession in it. (This avoids the usage of boilerplate code to create SparkSession as in Jupyter Notebook)

```
pyspark3
```



```
[itv005857@g02 ~]$ pyspark3
SPARK_MAJOR_VERSION is set to 3, using Spark3
Python 2.7.5 (default, Jun 28 2022, 15:30:04)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-44)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
/opt/spark3-client/python/pyspark/context.py:225: DeprecationWarning: Support for Python 2 a
.org/news/plan-for-dropping-python-2-support.html.
  DeprecationWarning)
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.0.1
      /_/

Using Python version 2.7.5 (default, Jun 28 2022 15:30:04)
SparkSession available as 'spark'.
>>> spark
<pyspark.sql.session.SparkSession object at 0x7fdab43a0f50>
>>> sc
<SparkContext master=yarn appName=PySparkShell>
>>>
```

However, Jupyter Notebooks have more user friendly UI and are better to work on (code can be saved for future reference with comments in the form of pdf)

Parallelize :

How things work in the industry

- Firstly, Develop the logic to meet the given Use Case around sample data
- Then, test the functionality on a small dataset (During the Development / testing phase, loading huge files of several terabytes will lead to wastage of resources.)

  parallelize() can be used to create an RDD from locally generated small data.

- After the desired results are achieved, the same can be applied to the large Datasets.

[One of the features of Functional Programing - Chaining of functions]

Example logic to find the frequency of each word on a sample data for testing (The same logic can then be applied to a large file of 10TB by replacing the creation of RDD using parallelize with the actual file)

```
words = ("big","Data","Is","SUPER","Interesting","BIG","data","IS","A","Trending","technology")

words_rdd = spark.sparkContext.parallelize(words)

words_normalized = words_rdd.map(lambda x:x.lower())

words_normalized.collect()

mapped_words = words_normalized.map(lambda x:(x,1))

aggregated_result = mapped_words.reduceByKey(lambda x,y:x+y)

aggregated_result.collect()
```

Chaining functions:

```
spark.sparkContext.parallelize(words).map(lambda x:x.lower()).map(lambda x:(x,1)).reduceByKey(lambda x,y:x+y).collect()
```

(or)

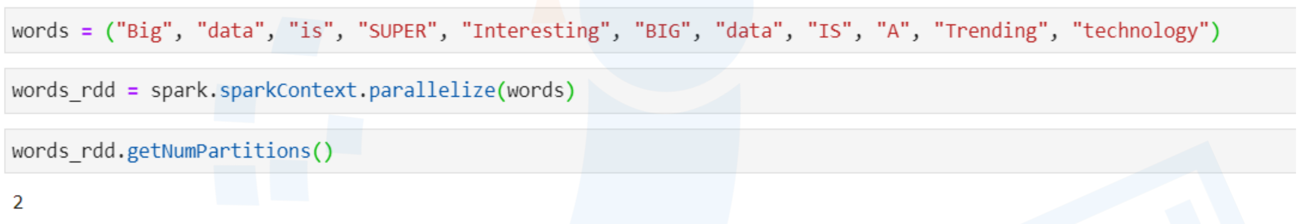
```
result = spark. \
sparkContext. \
parallelize(words). \
map(lambda x:x.lower()). \
map(lambda x:(x,1)). \
reduceByKey(lambda x,y:x+y)
result.collect()
```

How to check the no.of RDD partitions

Let's consider an example file of 1GB(1024MB) in HDFS -

No.of blocks in HDFS for this 1GB file (with default block size of 128MB)
1024MB / 128MB = 8 Blocks

This indicates no.of RDD partitions will be 8 Partitions (Because no.of RDD partitions = no.of Blocks in HDFS)



```
words = ("Big", "data", "is", "SUPER", "Interesting", "BIG", "data", "IS", "A", "Trending", "technology")
```

```
words_rdd = spark.sparkContext.parallelize(words)
```

```
words_rdd.getNumPartitions()
```

```
2
```

The file size in the above example is less than 128MB => There would be only one block in HDFS => no.of RDD partitions should be 1.

no.of RDD partitions = no.of blocks in HDFS

getNumPartitions()

However, on executing the getNumPartitions() function, it is resulting in 2 partitions.

This is because of the property defaultMinPartitions set to 2 (i.e., if the number of RDD partitions is < 2, consider the default value. If > 2 then consider the evaluated value based on number of blocks)

defaultMinPartitions -

spark.SparkContext.defaultMinPartitions

defaultParallelism -

spark.SparkContext.defaultParallelism is a property that indicates the no.of default tasks that can run in parallel.

countByValue - Is an action that gives the same output as map and reduceByKey combined.

countByValue = map + reduceByKey

Seems that countByValue is a better choice as it requires less lines of code to achieve the same results.

However, countByValue cannot replace map & reduceByKey always because

- countByValue is an action where output is captured on a local / gateway node and not on the cluster (implies, no further processing in parallel can happen on the results from countByValue)
- reduceByKey on the other hand is just a transformation (implies, there can be further parallel processing on the results from reduceByKey)

Note:

- If you need to do further parallel processing, then use map+reduceByKey
- If the resultant output would be the final output where no further processing in parallel is required, then countByValue would be the right choice.

Categories of Transformations :

1. **Narrow Transformation** - No Shuffling takes place

   In such transformations, the transformed data will not be transferred to a different machine but resides on the same machine.

   Ex - map, filter, flatmap

2. **Wide Transformation** - Shuffling takes place

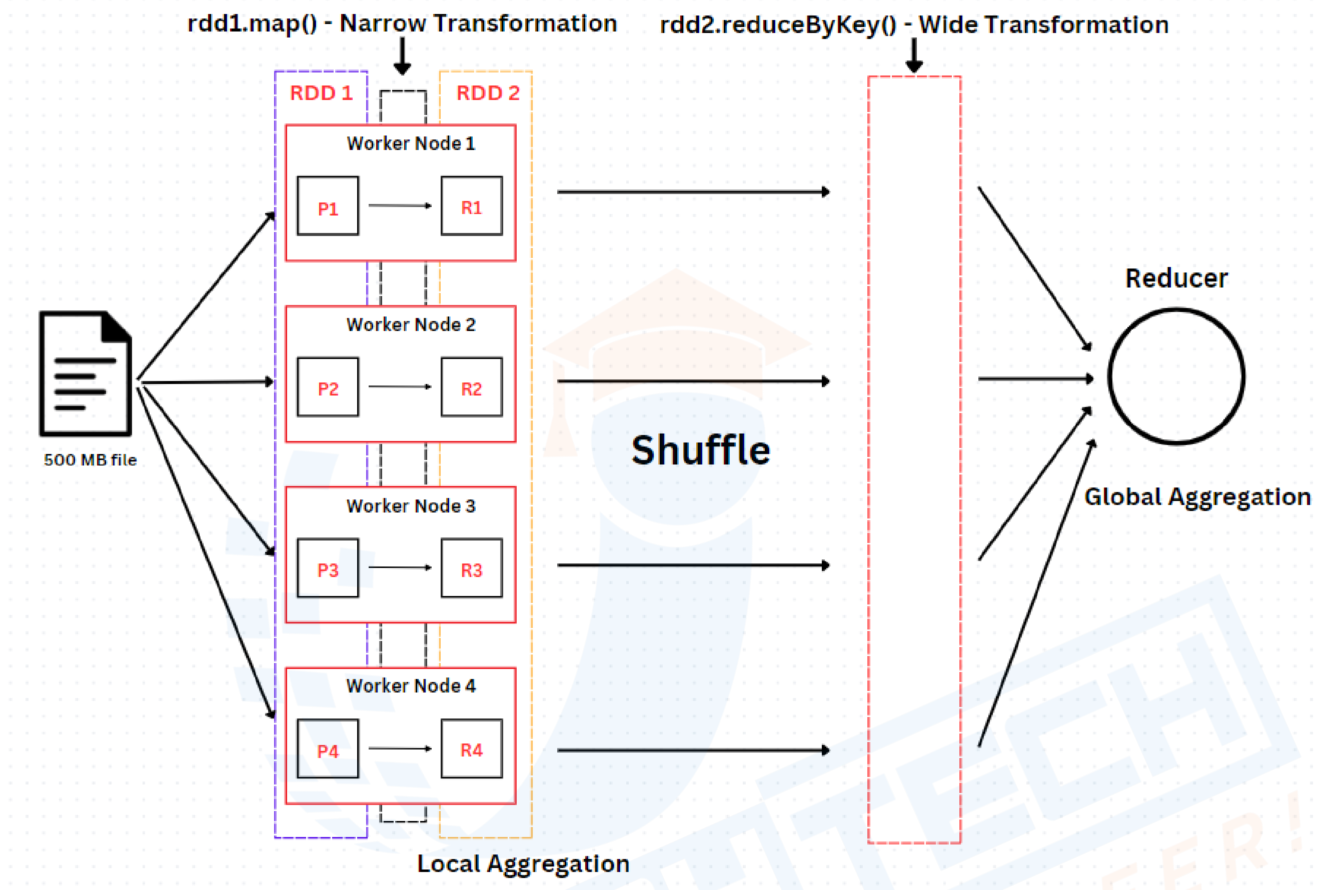
   In such transformations, the transformed data is transferred to a different machine for final global aggregation.

   Ex - reduceByKey, groupByKey

Wide Transformations

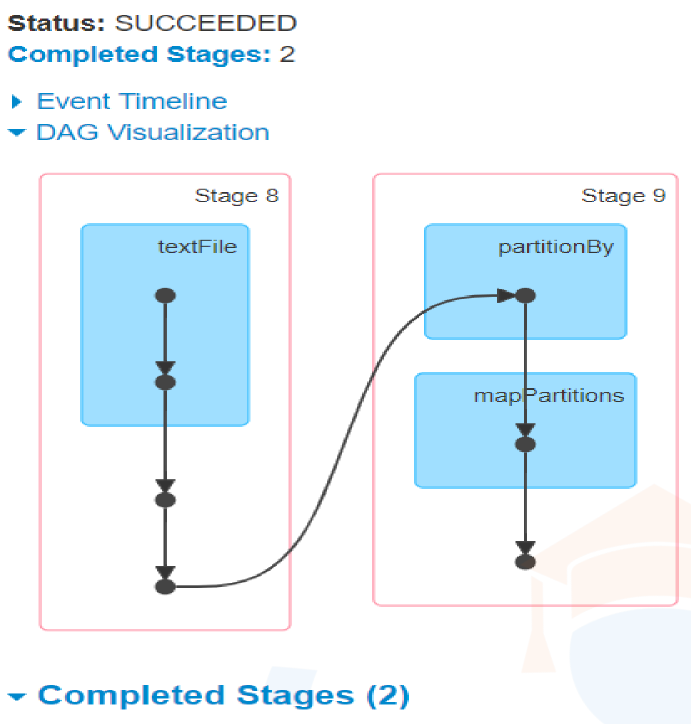
- Involve shuffle which is a costly transformation.

- Should be minimized and performed towards the end after the data is narrowed down to almost the desired state and thereby requires only minimal data to be transferred.



Visualizing Spark Jobs on History Server - Jobs will be visible on the history server only after they are terminated.

History Server URL - http://m02.itversity.com:18080/


Status: SUCCEEDED
Completed Stages: 2

▸ Event Timeline
▾ DAG Visualization

▾ **Completed Stages (2)**

TASK , JOB & STAGE

TASK	JOB	STAGE
Number of Tasks = Number of Partitions	No.of Jobs = No.of Actions executed	Number of Stages = Number of Wide Transformations + 1
-Task is at the smallest level. Every partition has an associated task.	- Every Action executed will have a corresponding Job	-On every Wide transformation a new stage is created

reduce Vs reduceByKey

reduce	reduceByKey
• Is an Action • Single Value as a result • Results are captured on the local driver machine	• Is a Transformation. • Resultset is determined by the no.of distinct keys • Results are on the cluster

Understanding the internals of processing large datasets

Use Case - Find the no.of orders in each category (Complete, Closed, Cancelled, Processing…)

File - Orders Dataset of 3.5GB (3500 MB)

Structure of the data -

order_id , order_date, customer_id, order_status

No.of blocks = 3500 MB /128 MB(default block size) = 28 Block = 28 Partitions

Solution : Using reduceByKey to get the final aggregate of the no.of orders in each category.

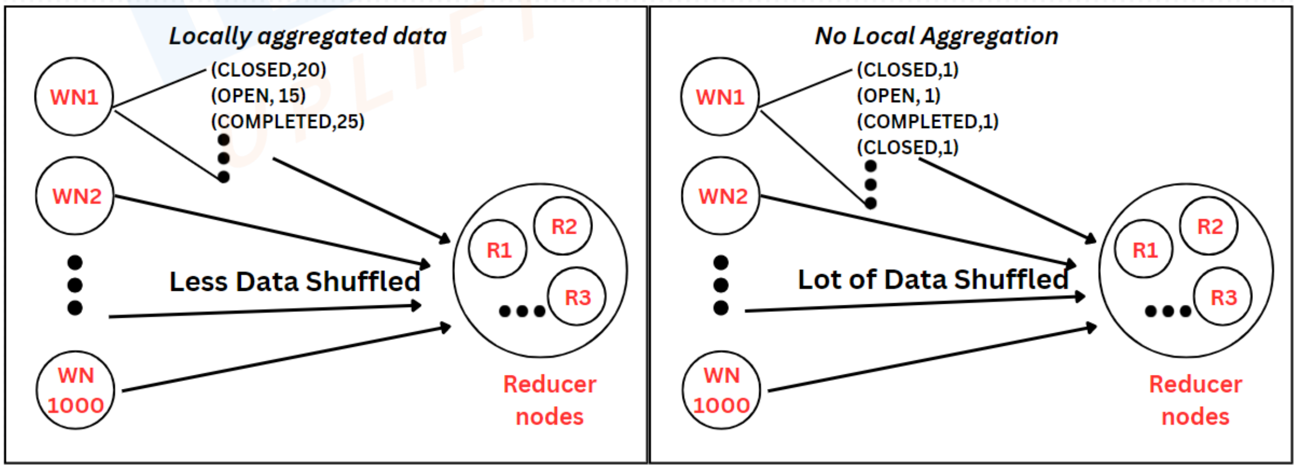
reduceByKey Vs groupByKey

Usecase :

-Consider a 1000 node cluster
-We need to process 1TB orders data on this cluster
-1TB = 8000 blocks (128MB default block size)
-Each node will hold 8 blocks
-RDD has 8000 partitions (no.of partitions = no.of blocks)
-9 different order status

So, 1000 nodes will work in parallel to aggregate the count for each order status

reduceByKey	groupByKey
• Local Aggregation is done before shuffling the data • Less data is shuffled • More tasks are executed in parallel • Doesn't lead to out of memory error as the local aggregation helps in reducing the volume of data to be shuffled.	• No Local Aggregation is done before shuffling the data • All the data is shuffled • Less tasks are executed in parallel • Can lead to out of memory error as huge volumes of data are shuffled down to a very few nodes.



Spark JOIN

Consider an Example Use case with 2 datasets and both are in HDFS

1. **orders** (file of ~ 1GB) no.of blocks = 9 = no.of partitions in RDD

   with columns (order_id, order_date, customer_id, order_status)

2. **customers** (file of 1MB) no.of blocks = 1 & no.of partitions = 2

   with columns (customer_id, fname, lname, username, password, address, city, state, pincode)

Common Column **(customer_id in the above example)** is used to join 2 datasets. Data with the common column has to be shuffled to a single partition for the join to happen.

joined_rdd = customers.join(orders)

(Joined based on the keys of the pair RDDs)

After executing the above example job, a **complicated DAG** is created with

2 Stages (because Join is a wide transformation, another stage gets created)

- Spark RDD Joins are wide transformations that involves data shuffling over the network
- They could lead to performance issues if not designed carefully as it involves a lot of data transfer
- Data has to be present on the same partition in order to perform Join.

BROADCAST JOIN

Normal Join is a costly operation as it is a wide transformation and involves a lot of data shuffle. To overcome this issue, Broadcast join provides a more optimized solution.

How does a Broadcast Join Work?

The larger dataset (orders of ~ 1GB) is distributed across the cluster. Whereas, the smaller dataset (customers of 1MB), the complete copy of the dataset, is made available on every machine on the cluster. This hugely reduces the shuffling of Data and thereby optimizing the performance.

spark.sparkContext.broadcast(customers.collect())

-Since the complete smaller dataset is broadcasted across all the nodes, every individual node is capable of performing join locally.

-No Shuffling happens over the network for broadcast join

-No additional stage is created in the DAG as there is no shuffling involved in a broadcast join.

On executing the example job (on orders and customers data) for retrieving the pincode of a given customer_id, a **simple DAG** is created with

1 Stage (because broadcast joins don't involve shuffling of data. Not a wide transformation)

Repartition Vs Coalesce

Suppose there is a need to increase or decrease the number of partitions, then **repartition** can be used as it can do both, i.e., increase / decrease the no.of partitions in a RDD.

Scenario 1 : When to increase the number of partitions

- Say you have a 5GB file => 40 blocks (= 40 partitions) running on a 100 node cluster.
- Since there are only 40 partitions, they will run on 40 nodes and the remaining 60 nodes are idle.
- This leads to underutilization of resources. In such cases, the partitions can be increased to achieve more parallelism.

How to increase the no.of partitions?

repartitioned_rdd = base_rdd.repartition(<increased partitions number>)

Scenario 2 : When to decrease the number of partitions

- Say you have 1 TB file => 8000 blocks (= 8000 partitions) running on a 100 node cluster.
- Each node will handle around 80 partitions.
- After applying transformations like - filter, the data would reduce significantly (say from 128MB[default] to 1MB)
- This will lead to a lot of sparse data and also would be tedious to maintain large no.of partitions.
- It would be more efficient to have few no.of completely filled partitions than having large no.of sparsely filled partitions.

How to decrease the no.of partitions?

repartitioned_rdd = base_rdd.repartition(<decreased partitions number>)

Coalesce

Coalesce can only decrease the number of partitions and cannot increase the no.of partitions.

When repartition can both increase / decrease the partitions, then what is the need for coalesce?

Repartition does a complete reshuffle of data while changing the no.of partitions with the intent to have equal sized partitions.

Repartition works well while increasing the number of partitions but is not efficient when decreasing the partitions.

Coalesce on the other hand tries to merge the partitions on the same node to form a new partition that could be of unequal sizes but the shuffling is avoided. Therefore, coalesce is **preferred when decreasing the no.of partitions as it avoids shuffling of data.**

Cache

- Suppose there are multiple costly transformations that need to be executed
- When an action is called for the first time, all the transformations are executed.
- If the results of these transformations are cached, then…
- When an action is called again, the cached results of transformations are used without having to re-execute all the transformations again.