

ASSIGNMENT 18.1

Task 1

Given a list of numbers - List [Int] (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)


 acadgild@localhost:~

```
scala> val list = List(1,2,3,4,5,6,7,8,9,10)
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val listRDD = sc.parallelize(list)
listRDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:26

scala> █
```

1. Find the sum of all numbers.

 acadgild@localhost:~

```
scala> val sum = listRDD.sum
sum: Double = 55.0

scala> █
```


2. Find the total elements in the list

 acadgild@localhost:~

```
scala> val totalRDD = listRDD.count()
totalRDD: Long = 10

scala> █
```

3. Calculate the average of the numbers in the list

 acadgild@localhost:~

```
scala> val avgRDD = listRDD.sum/listRDD.count()
avgRDD: Double = 5.5

scala> █
```

4. Find the sum of all the even numbers in the list

acadgild@localhost:~

```
scala> val evenRDD = listRDD.filter(x => x%2==0)
evenRDD: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[3] at filter at <console>:28

scala> evenRDD.collect()
res0: Array[Int] = Array(2, 4, 6, 8, 10)

scala> val sumEvenRDD = evenRDD.sum
sumEvenRDD: Double = 30.0
```

5. Find the total number of elements in the list divisible by both 5 and 3

acadgild@localhost:~

```
scala> val divRDD = listRDD.filter(x => x%3==0 && x%5==0)
divRDD: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[5] at filter at <console>:28

scala> divRDD.collect()
res1: Array[Int] = Array()
```

Above result shows, there is no element in the list is divisible by both 5 and 3.

Task 2

1) Pen down the limitations of MapReduce.

1. Since MapReduce is suitable only for batch processing jobs, implementing interactive jobs and models becomes impossible.
2. Implementing iterative map reduce jobs is expensive due to the huge space consumption by each job.
3. In Hadoop, with a parallel and distributed algorithm, MapReduce process large data sets. There are tasks that need to be performed: **Map** and **Reduce** and, MapReduce requires a lot of time to perform these tasks thereby increasing latency. Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing speed
4. MapReduce is not suitable for real time data processing.

5. MapReduce is not so efficient for iterative processing, as Hadoop does not support cyclic data flow, (i.e. a chain of stages in which each output of the previous stage is the input to the next stage).

6. MapReduce is not efficient for caching. In Hadoop, MapReduce cannot cache the intermediate data in memory for a further requirement, which diminishes the performance of Hadoop.

7. Not suitable for graphs processing.

8. MapReduce is lengthy code. The number of lines produces the number of bugs. Hence, it will take more time to execute the programs.

9. Apache Hadoop is challenging in maintaining the complex applications. Hadoop is missing encryption at the storage and network levels, which is a major point of concern. Apache Hadoop supports Kerberos authentication, which is hard to manage.

2) What is RDD? Explain few features of RDD?

RDD stands for “**Resilient Distributed Dataset**”. It is the fundamental data structure of Apache Spark. RDD in Apache Spark is an immutable collection of objects, which computes on the different node of the cluster.

Decomposing the name RDD:

- **Resilient**, i.e. fault-tolerant with the help of RDD lineage graph (**DAG**) and so able to recompute missing or damaged partitions due to node failures.
- **Distributed**, since Data resides on multiple nodes.
- **Dataset** represents records of the data you work with. The user can load the data set externally, which can be either JSON file, CSV file, text file or database via JDBC with no specific data structure.

Each and every dataset in RDD is logically partitioned across many servers so that they can be computed on different nodes of the cluster. RDDs are fault tolerant i.e. it possess self-recovery in the case of failure.

Features of RDD in Spark

1. In-memory Computation

Spark RDDs have a provision of in-memory computation. It stores intermediate results in distributed memory (RAM) instead of stable storage (disk).

2. Lazy Evaluations

All transformations in Apache Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base data set.

Spark computes transformations when an action requires a result for the driver program.

3. Fault Tolerance

Spark RDDs are fault tolerant as they track data lineage information to rebuild lost data automatically on failure. They rebuild lost data on failure using lineage, each RDD remembers how it was created from other datasets (by transformations like a map, join or groupBy) to recreate itself.

4. Immutability

Data is safe to share across processes. It can also be created or retrieved anytime which makes caching, sharing & replication easy. Thus, it is a way to reach consistency in computations.

5. Partitioning

Partitioning is the fundamental unit of parallelism in Spark RDD. Each partition is one logical division of data, which is mutable. One can create a partition through some transformations on existing partitions.

6. Persistence

Users can state which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage or on Disk).

7. Coarse-grained Operations

It applies to all elements in datasets through maps or filter or group by operation.

8. Location-Stickiness

RDDs are capable of defining placement preference to compute partitions. Placement preference refers to information about the location of RDD. The DAGScheduler places the partitions in such a way that task is close to data as much as possible. Thus, speed up computation.

3) List down few Spark RDD operations and explain each of them.

RDD in Apache Spark supports two types of operations: **Transformation** and **Actions**.

1.Transformations: Spark RDD Transformations are functions that take an RDD as the input and produce one or many RDDs as the output. They do not change the input RDD (since RDDs are immutable and hence one cannot change it), but always produce one or more new RDDs by applying the computations they represent e.g. Map(), filter(), reduceByKey() etc.

Transformations are lazy operations on an RDD in Apache Spark. It creates one or many new RDDs, which executes when an Action occurs. Hence, Transformation creates a new dataset from an existing one.

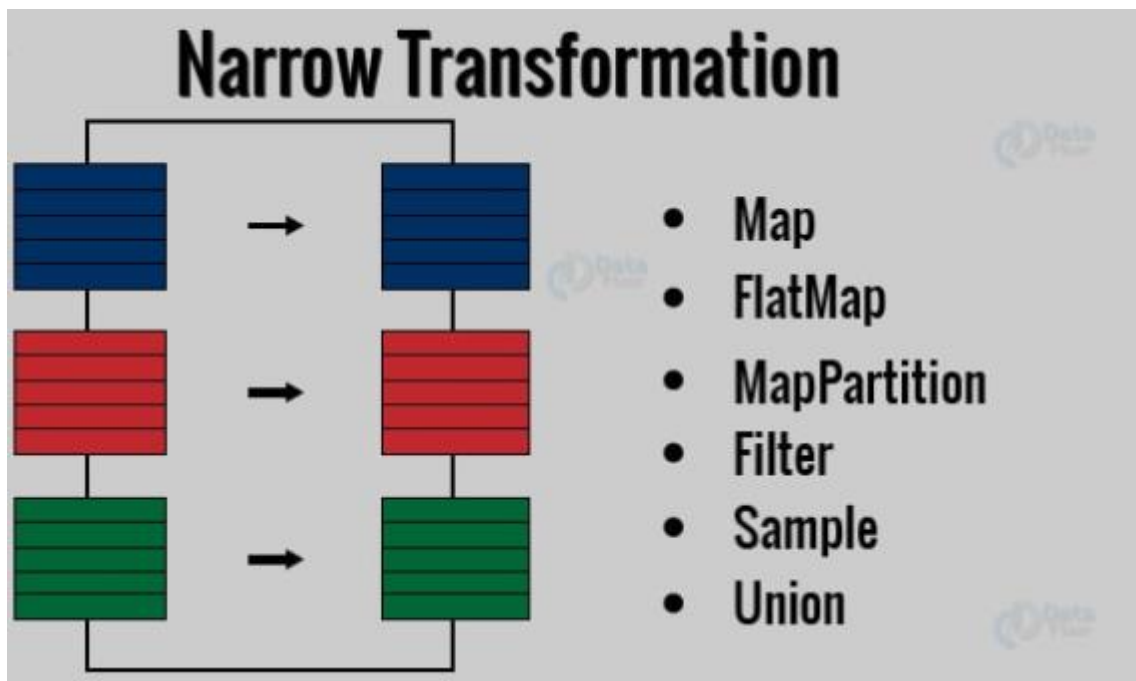
Certain transformations can be pipelined which is an optimization method, that Spark uses to improve the performance of computations.

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

a) Narrow Transformations

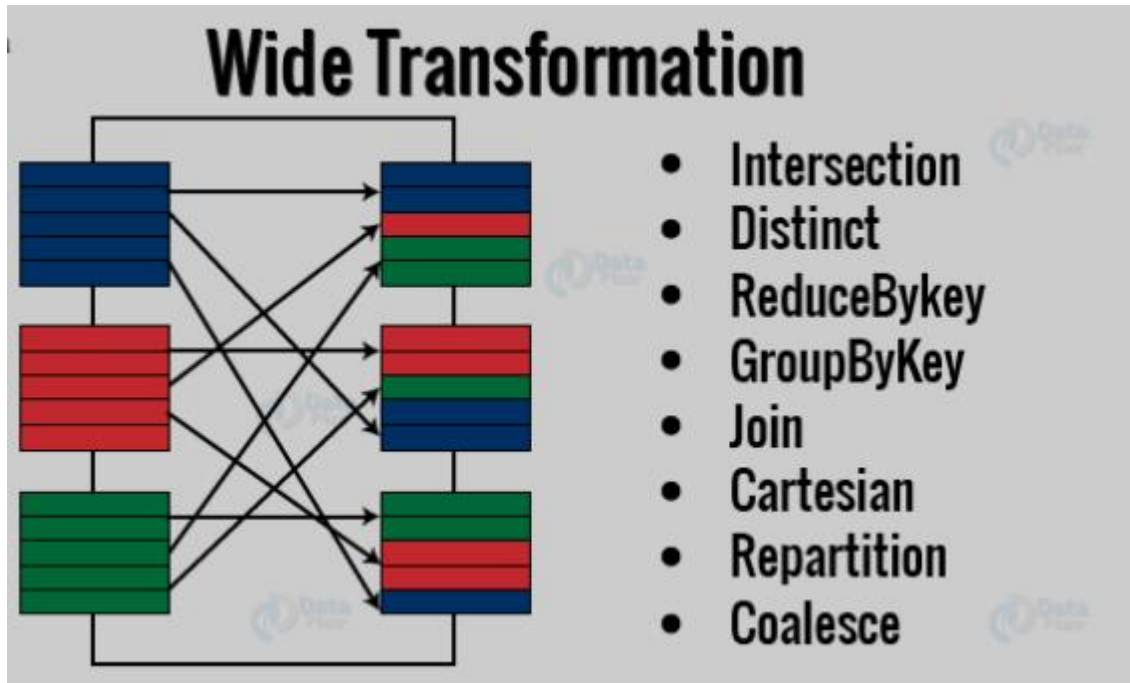
It is the result of map, filter and such that the data is from a single partition only, i.e. it is self-sufficient. An output RDD has partitions with records that originate from a single partition in the parent RDD. Only a limited subset of partitions used to calculate the result.

Spark groups narrow transformations as a stage known as **pipelining**.



b) Wide Transformations:

It is the result of `groupByKey()` and `reduceByKey()` like functions. The data required to compute the records in a single partition might live in many partitions of the parent RDD. Wide transformations are also known as shuffle transformations because they may or may not depend on a shuffle.



2. Actions: An Action in Spark returns final result of RDD computations. It triggers execution using lineage graph to load the data into original RDD, carry out all intermediate transformations and return results to Driver program or write it out to file system.

Lineage graph is dependency graph of all parallel RDDs of RDD.

Actions are RDD operations that produce non-RDD values. They materialize a value in a Spark program.

An Action is one of the ways to send result from executors to the driver. `First()`, `take()`, `reduce()`, `collect()`, the `count()` is some of the Actions in spark.

Using transformations, one can create RDD from the existing one. However, when we want to work with the actual dataset, at that point we use Action. When the Action occurs, it does not create the new RDD, unlike transformation. Thus, actions are RDD operations that give no RDD values. Action stores its value either to drivers or to the external storage system. It brings laziness of RDD into motion.