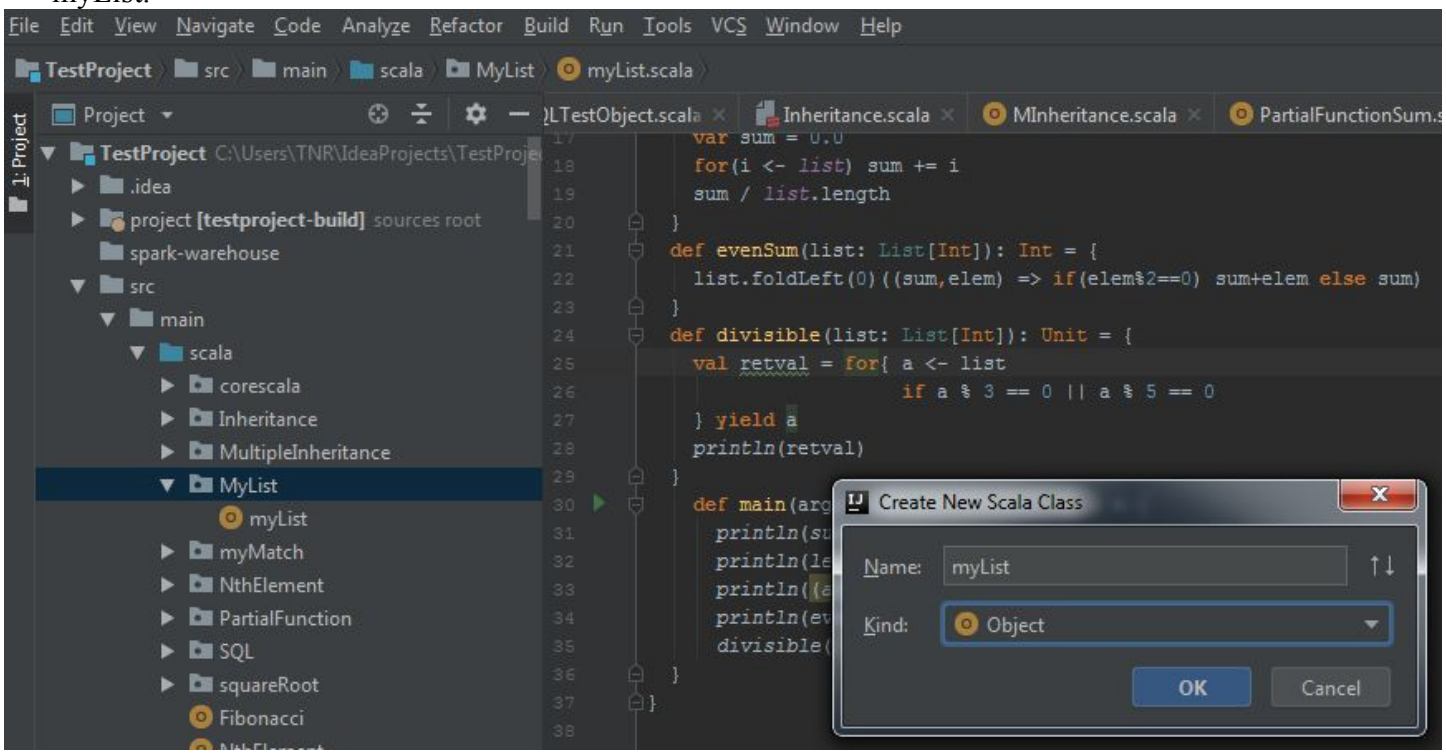
Session 18 – Introduction to Spark
Assignment 1

Task 1

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

- **Find the sum of all numbers.**
- **Find the total elements in the list**
- **Calculate the average of the numbers in the list.**
- **Find the sum of all the even numbers in the list.**
- **Find the total number of elements in the list divisible by both 5 and 3.**

1. Start the intelliJ IDEA and create a package called myList. And create a scala package with name myList.

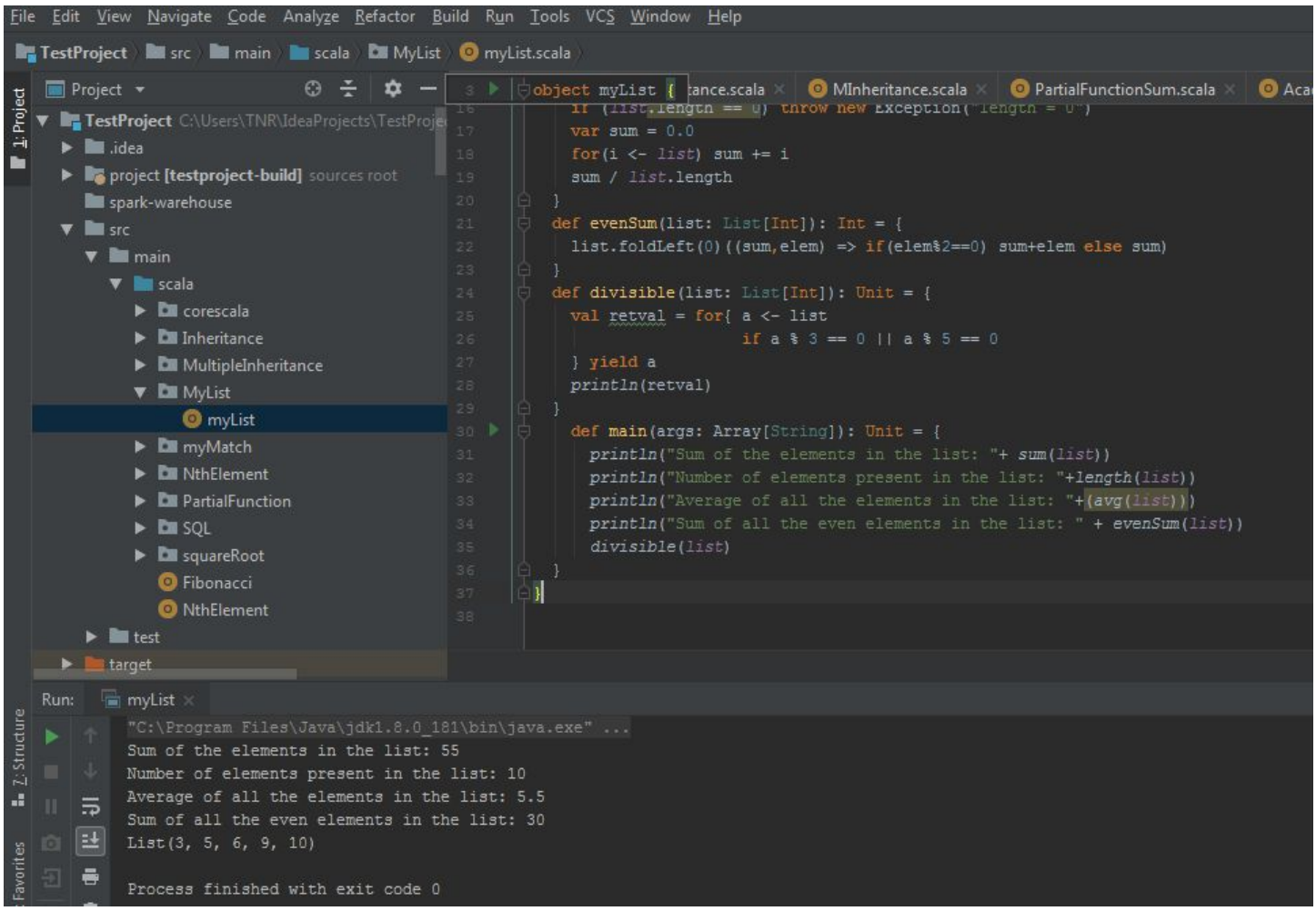


2. Now write the following code inside the class myList.

```
package MyList

object myList {
    val list = List(1,2,3,4,5,6,7,8,9,10)

  def length[A](list:List[A]):Int = {
    list.length
  }

  def sum(list: List[Int]): Int = list match {
    case Nil => 0
    case x :: xs => x + sum(xs)
  }

  def avg(nums: List[Int]): Double = {
    if (list.length == 0) throw new Exception("length = 0")
    var sum = 0.0
    for(i <- list) sum += i
    sum / list.length
  }
  def evenSum(list: List[Int]): Int = {
    list.foldLeft(0)((sum,elem) => if(elem%2==0) sum+elem else sum)
  }
```

```
  def divisible(list: List[Int]): Unit = {
    val retval = for{ a <- list
                      if a % 3 == 0 || a % 5 == 0
    } yield a
    println(retval)
  }
    def main(args: Array[String]): Unit = {
        println("Sum of the elements in the list: "+ sum(list))
        println("Number of elements present in the list: "+length(list))
        println("Average of all the elements in the list: "+(avg(list)))
        println("Sum of all the even elements in the list: " + evenSum(list))
        divisible(list)
  }
}
```

3. Now run the code to print all the



4. We can observe that the definitions for sum, length, average and sum of even numbered list all called from the main definition.

Task 2:

1. **Pen down the limitation of MapReduce?**

Ans:

MapReduce Cannot handle:

1. Interactive Processing.
2. Real-time (stream) Processing.

3. Iterative (delta) Processing.
4. In-memory Processing.
5. Graph Processing.

1. **Issue with small files**

Hadoop is not suited for small data. (HDFS) lacks the ability to efficiently support the random reading of small files because of its high capacity design.

2. **Slow processing speed**

   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.

3. **Support for Batch Processing only**

Hadoop supports batch processing only, it does not process streamed data, and hence overall performance is slower. MapReduce framework of Hadoop does not leverage the memory of the Hadoop cluster to the maximum..

4. **No Real-time Data processing**

Apache Hadoop is designed for batch processing, that means it take a huge amount of data in input, process it and produce the result. Although batch processing is very efficient for processing a high volume of data, but depending on the size of the data being processed and computational power of the system, an output can be delayed significantly. Hadoop is not suitable for Real-Time data processing.

5. **No Delta Iteration**

Hadoop 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 state).

6. **Latency**

In Hadoop, MapReduce framework is comparatively slower, since it is designed to support different format, structure and huge volume of data. In **MapReduce,** Map takes a set of data and converts it into another set of data, where individual elements are broken down into **key value pair** and Reduce takes the output from the map as input and process further and MapReduce requires a lot of time to perform these tasks thereby increasing latency.

7. **Not easy to Use**

In Hadoop, MapReduce developers need to hand code for each and every operation which makes it very difficult to work. MapReduce has no interactive mode, but adding one such as hive and pig makes working with MapReduce a little easier for adopters.

8. **No caching.**

Hadoop 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.

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

Resilient Distributed Datasets Resilient Distributed Datasets (RDD) is a fundamental data structure of spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java or scala objects, including user defined classes.

Formally, an RDD is a read-only partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.

There are two ways to create RDDs − **parallelizing** an existing collection in your driver program, or **referencing a dataset** in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations. Let us first discuss how MapReduce operations take place and why they are not so efficient.

Data Sharing is Slow in MapReduce

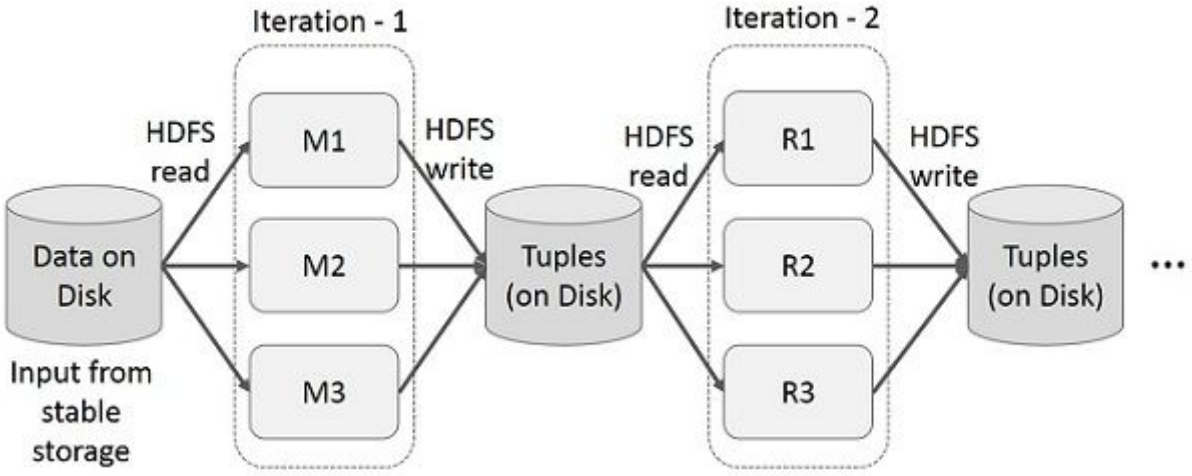
MapReduce is widely adopted for processing and generating large datasets with a parallel, distributed algorithm on a cluster. It allows users to write parallel computations, using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Unfortunately, in most current frameworks, the only way to reuse data between computations (Ex − between two MapReduce jobs) is to write it to an external stable storage system (Ex − HDFS). Although this framework provides numerous abstractions for accessing a cluster's computational resources, users still want more.

Both **Iterative** and **Interactive** applications require faster data sharing across parallel jobs. Data sharing is slow in MapReduce due to **replication, serialization**, and **disk IO**. Regarding storage system, most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

Iterative Operations on MapReduce

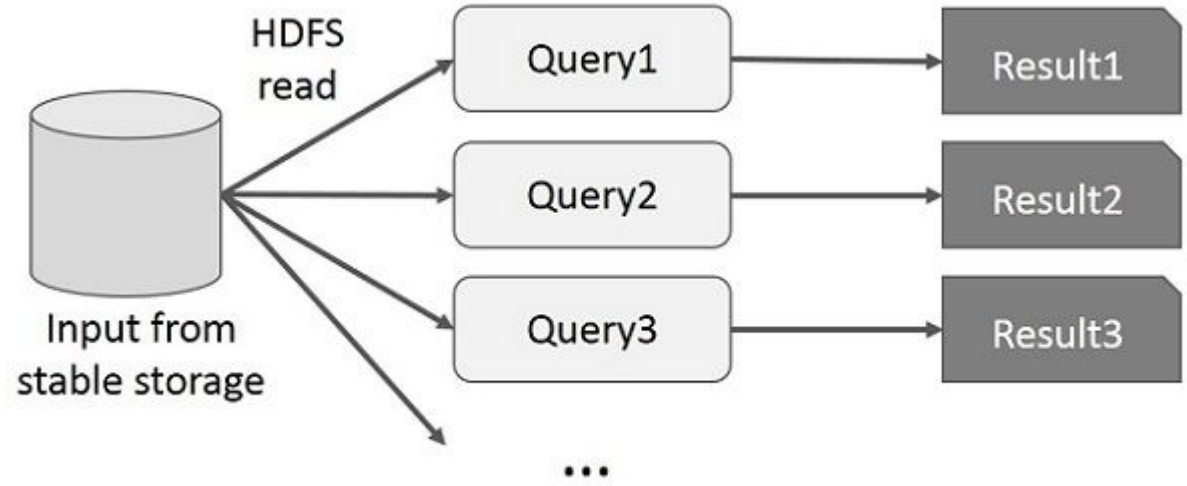
Reuse intermediate results across multiple computations in multi-stage applications. The following illustration explains how the current framework works, while doing the iterative operations on MapReduce. This incurs substantial overheads due to data replication, disk I/O, and serialization, which makes the system slow.



Interactive Operations on MapReduce

User runs ad-hoc queries on the same subset of data. Each query will do the disk I/O on the stable storage, which can dominate application execution time.

The following illustration explains how the current framework works while doing the interactive queries on MapReduce.



Data Sharing using Spark RDD

Data sharing is slow in MapReduce due to **replication, serialization**, and **disk IO**. Most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

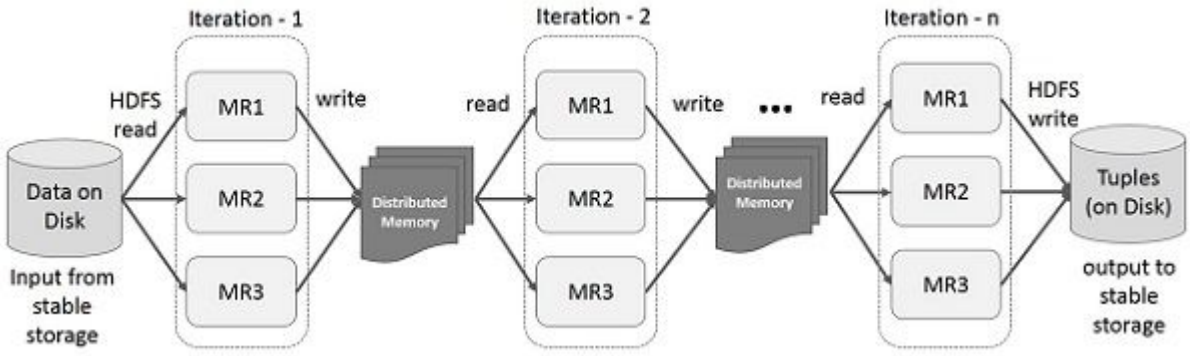
Recognizing this problem, researchers developed a specialized framework called Apache Spark. The key idea of spark is **R**esilient **D**istributed **D**atasets (RDD); it supports in-memory processing computation. This means, it stores the state of memory as an object across the jobs and the object is sharable between those jobs. Data sharing in memory is 10 to 100 times faster than network and Disk.

Let us now try to find out how iterative and interactive operations take place in Spark RDD.

Iterative Operations on Spark RDD

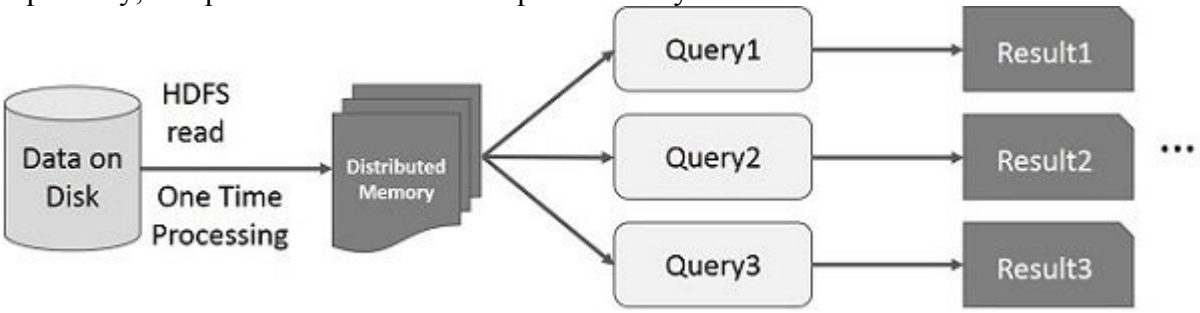
The illustration given below shows the iterative operations on Spark RDD. It will store intermediate results in a distributed memory instead of Stable storage (Disk) and make the system faster.

Note − If the Distributed memory (RAM) is not sufficient to store intermediate results (State of the JOB), then it will store those results on the disk.



Interactive Operations on Spark RDD

This illustration shows interactive operations on Spark RDD. If different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times.



By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also **persist** an RDD in memory, in which case Spark will keep the elements around on the cluster for much faster access, the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

Features of RDD in Spark

Several features of Apache Spark RDD are:

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 result right away. Instead, they just remember the transformations applied to somebase 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 away 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**

RDD 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 computations.

3. **List down few spark RDD operations and explain each of them?**

Spark RDD Operations

RDD in Spark supports two types of operations:

- Transformation
- Actions

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.

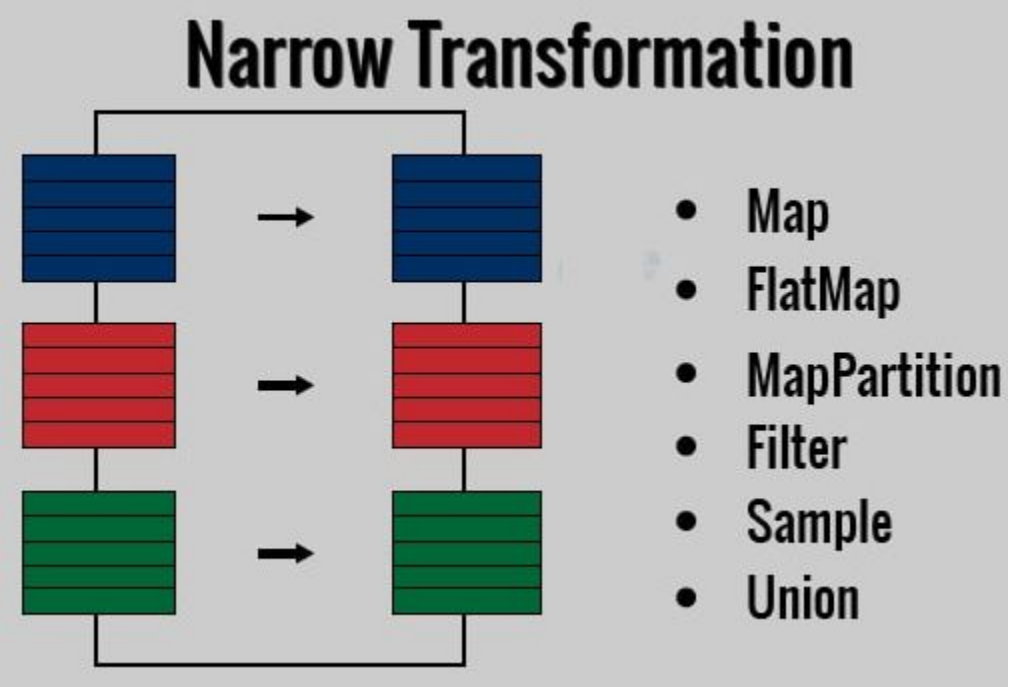
Certain transformations can be pipelined which is an optimization method, that spark uses to improve the performance of computations. There are two kinds of transformations:

1. **Narrow Transformations**
2. **Wide transformations.**

1. **Narrow Transformations**

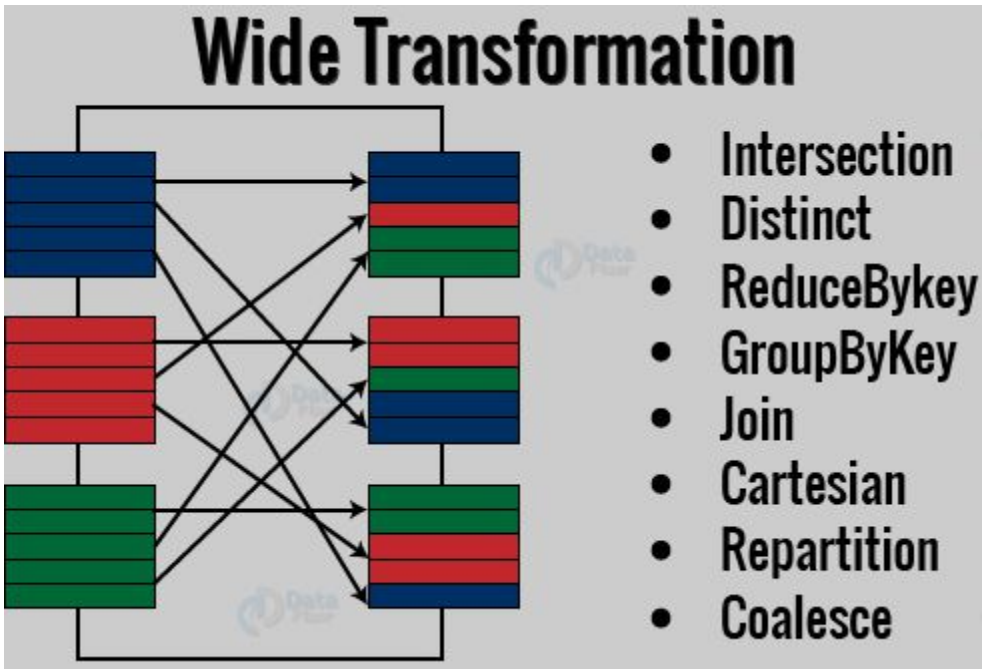
It is the result of map, filter and such that the data is from a single partition only, i.e., it is self-sufficient. An output RDD has partitions with records that originate from a single partition in the parent RDD. Only a limited subset of partitions used to create the result.

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



2. **Wide Transformations**

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



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 final results to Driver program or write it out to the 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. Frist(), take(), reduce(), collect() the count() is some of the Actions in spark.

Using transformations one can create RDD from the exiting one.. But 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 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.