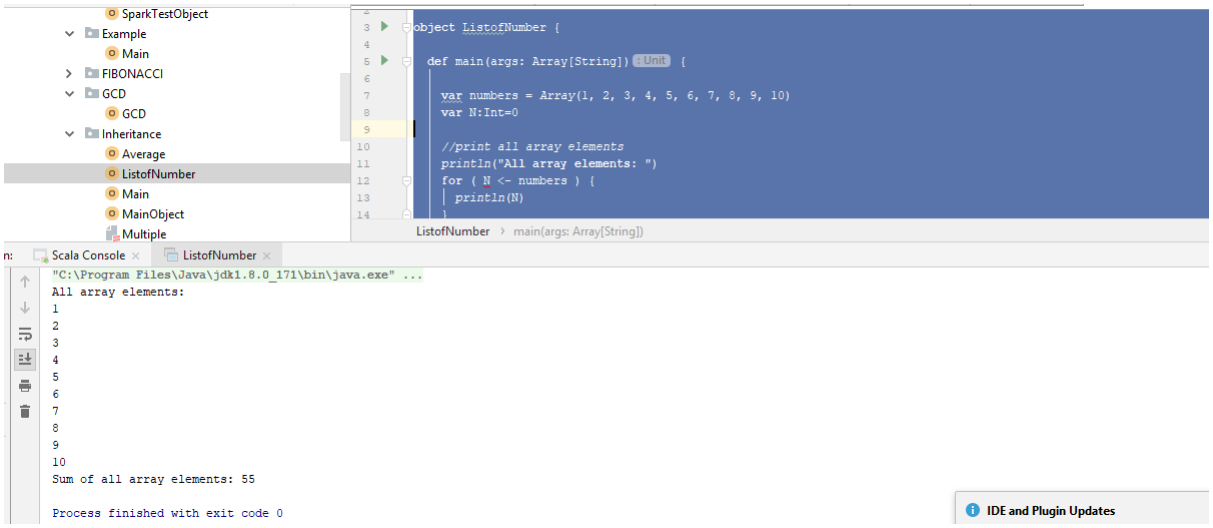
Session 18: INTRODUCTION TO SPARK Assignment 1

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

Solution:

```
package Inheritance

object ListofNumber {

  def main(args: Array[String]) {

    var numbers = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    var N:Int=0

    //print all array elements
    println("All array elements: ")
    for ( N <- numbers ) {
      println(N)
    }
    //calculating SUM of all elements
    var sum: Int=0
    for ( N <- numbers ) {
      sum+=N
    }
    println("Sum of all array elements: "+sum)


  }
}
```



- calculate the average of the numbers in the list

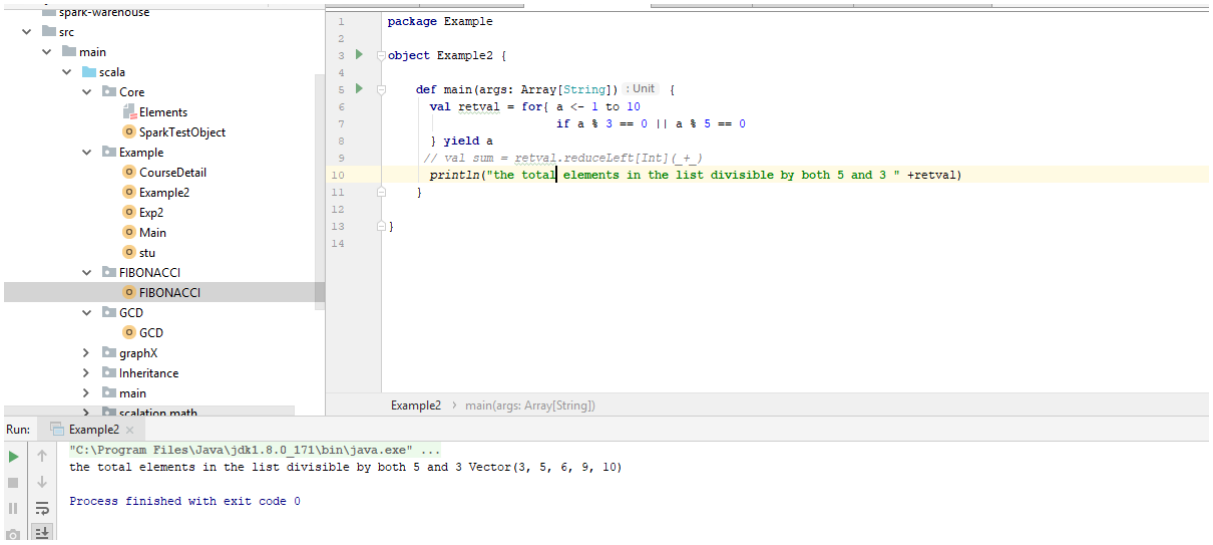
```
scala> def average[T]( ts: Iterable[T] )( implicit num: Numeric[T] ) = {
     |   num.toDouble( ts.sum ) / ts.size
     | }
average: [T](ts: Iterable[T])(implicit num: Numeric[T])Double

scala> average( List( 1,2,3,4,5,6,7,8,9,10) )
res0: Double = 5.5
```

- find the sum of all the even numbers in the list

```
scala> def evennum(array: List[Int]): Int = {
     |   def sum(array: List[Int], total: Int = 0): Int = {
     |     if (array.isEmpty) return total
     |     if (array.head % 2 == 0) return sum(array.tail, array.head + total)
     |     else return sum(array.tail, total)
     |   }
     |
     |   sum(array)
     | }
evennum: (array: List[Int])Int

scala> evennum(List(1,2,3,4,5,6,7,8,9,10))
res0: Int = 30

scala>
```

- find the total elements in the list divisible by both 5 and 3

```
package Example

object Example2 {

    def main(args: Array[String]) {
      val retval = for{ a <- 1 to 10
                        if a % 3 == 0 || a % 5 == 0
      } yield a

      println("the total elements in the list divisible by both 5 and 3 " +retval)
    }

}
```



Task 2

1) Pen down the limitations of MapReduce.

Solution:

1-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 element 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.

2-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

3-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

4-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 stage).

5. No Abstraction

Hadoop does not have any type of abstraction so MapReduce developers need to hand code for each and every operation which makes it very difficult to work.

6. Lengthy Line of Code

Hadoop has 1,20,000 line of code, the number of lines produces the number of bugs and it will take more time to execute the program.

7- Uncertainty

Hadoop only ensures that data job is complete, but it's unable to guarantee when the job will be complete.

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.

9 - Not fit for small files

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

Small files are the major problem in HDFS. A small file is significantly smaller than the HDFS block size (default 128MB). If we are storing these huge numbers of small files, HDFS can't handle these lots of files, as HDFS was designed to work properly with a small number of large files for storing large data sets rather than a large number of small files. If there are too many small files, then the NameNodewill be overloaded since it stores the namespace of HDFS.

10-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.

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

RDD (Resilient Distributed Dataset) is the fundamental data structure of **Apache Spark** which are an immutable collection of objects which computes on the different node of the cluster. Each and every dataset in **Spark RDD** is logically partitioned across many servers so that they can be computed on different nodes of the cluster.

- 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.

Feature of RDD

5.1. 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).

5.2. 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. Follow this guide for the deep study of **Spark Lazy Evaluation**.

5.3. 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. Follow this guide for the deep study of **RDD Fault Tolerance**.

5.4. 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.

5.5. 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.

5.6. Persistence

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

5.7. Coarse-grained Operations

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

5.8. 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
- 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.

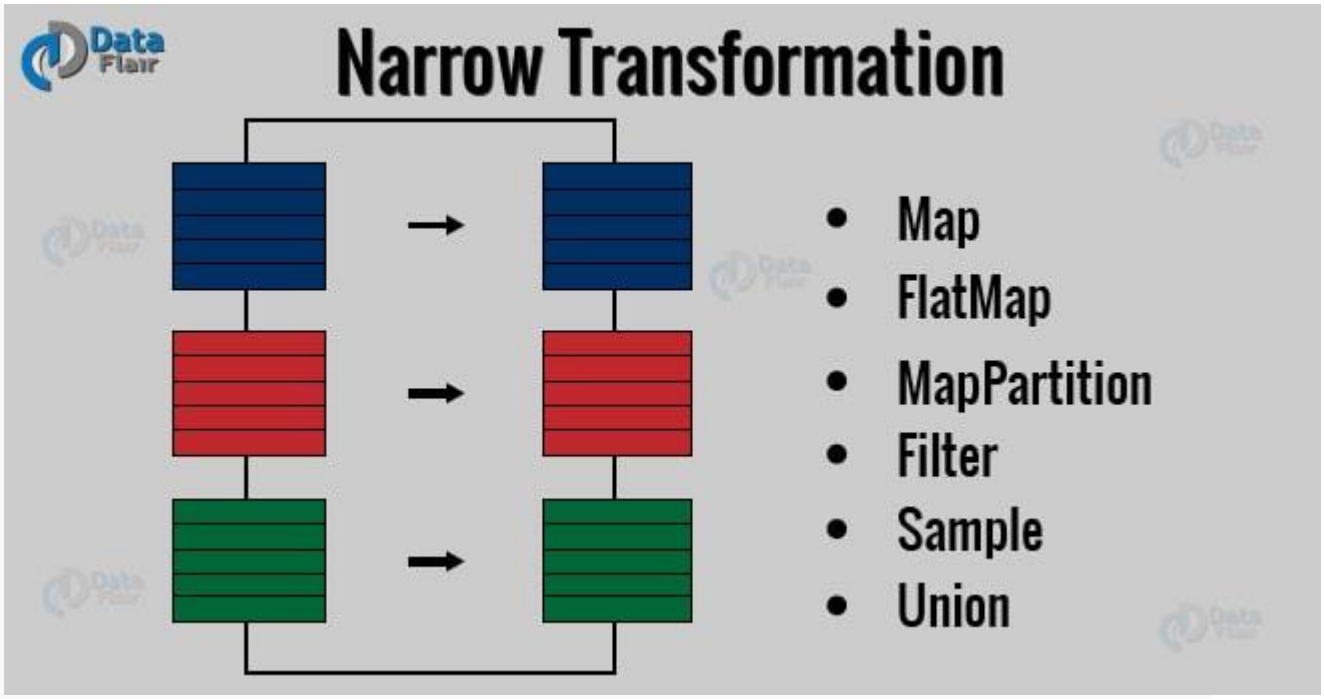
Transformations are lazy operations on an RDD in Apache Spark. It creates one or many new RDDs, which executes when an Action occurs. Hence, Transformation creates a new dataset from an existing one.

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: narrow transformation, wide transformation.

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 calculate the result.

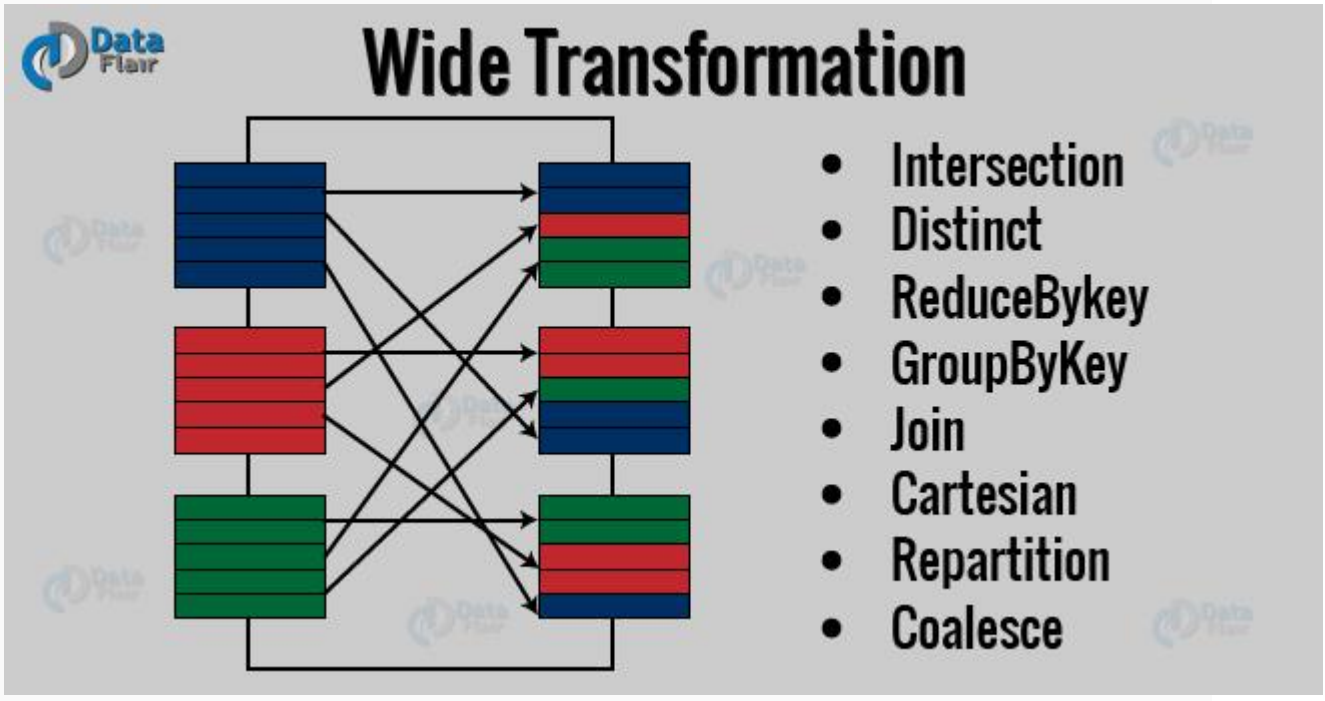
Spark groups narrow transformations as a stage known as pipelining.



Spark RDD – Narrow Transformation

2) Wide Transformations

It is the result of groupByKey() and reduceByKey() like functions. The data required to compute the records in a single partition may 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.



Wide Transformation

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 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. 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 or to the external storage system. It brings laziness of RDD into motion.