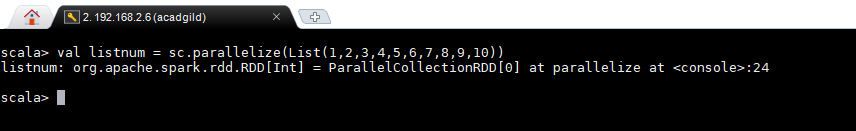
**Task 1**

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

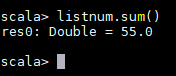
Create an RDD

*Command: val listnum = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))*



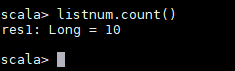
1. find the sum of all numbers

*Command : listnum.sum()*



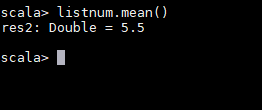
1. find the total elements in the list

*Command : listnum.count()*



1. calculate the average of the numbers in the list

*Command: listnum.mean()*



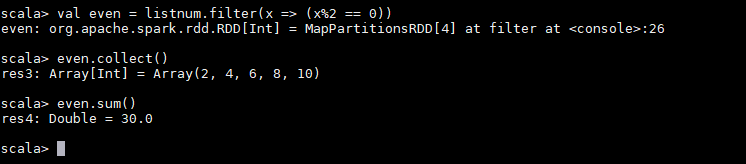
1. find the sum of all the even numbers in the list

*Command*

*val even = listnum.filter(x => (x%2 == 0)) -- to filter the even records*

*even.collect() – to check the filtered records in the above step*

*even.sum() -- to sum the set of even records*

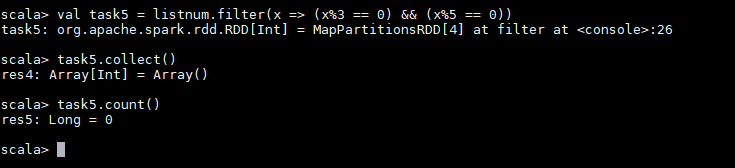


1. find the total number of elements in the list divisible by both 5 and 3

val task5 = listnum.filter(x => (x%3 == 0) && (x%5 == 0)) -- *to filter the numbers divisible by 3 and 5*

task5.collect() -- *to check the filtered records in the above step*

*task5.count() – to get the count of the filtered records*



**TASK-2**

1. **Pen down the limitations of MapReduce.**

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster.

* It’s based on disk computing
* Suitable for single pass computations - not iterative computations. Needs a sequence of MR jobs to run iterative tasks,
* Needs integration with several other frameworks/tools to solve bigdata use cases,

o Apache Storm for stream data processing

o Apache Mahout for machine learning

* Hadoop Map Reduce 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.
* Slow Processing Speed,No Real-time Data Processing Lengthy Line of Code MapReduce only ensures that data job is complete, but it’s unable to guarantee when the job will be complete.

1. **What is RDD? Explain few features of RDD?**

**RDD (Resilient Distributed Dataset)** is the fundamental data structure of [**Apache Spark**](http://data-flair.training/blogs/introduction-spark-tutorial-quickstart/) 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.

1. In-memory Computation

Spark RDDs have a provision of [in-memory computation](http://data-flair.training/blogs/apache-spark-in-memory-computing/). 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. Follow this guide for the deep study of[Spark Lazy Evaluation.](http://data-flair.training/blogs/lazy-evaluation-in-apache-spark-guide/)

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.](http://data-flair.training/blogs/apache-spark-streaming-fault-tolerance/)

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. Follow this guide to [learn What is DAG?](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/)

1. **List down few Spark RDD operations and explain each of them.**

RDD in Apache Spark supports two types of operations:

Transformation

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.

1.1. 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.

1.2. 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.

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 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.