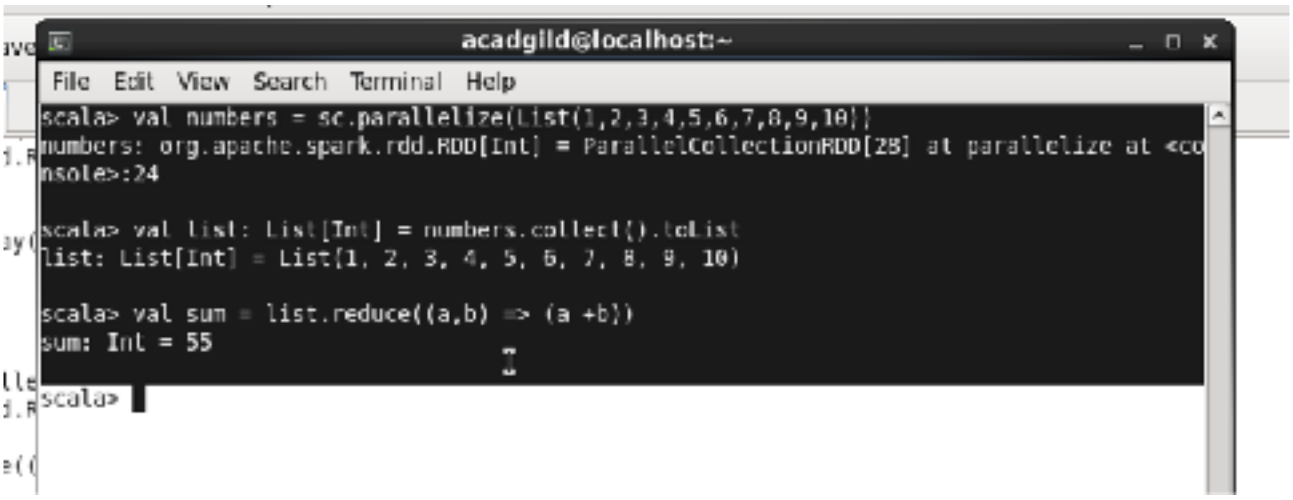
## Task 1

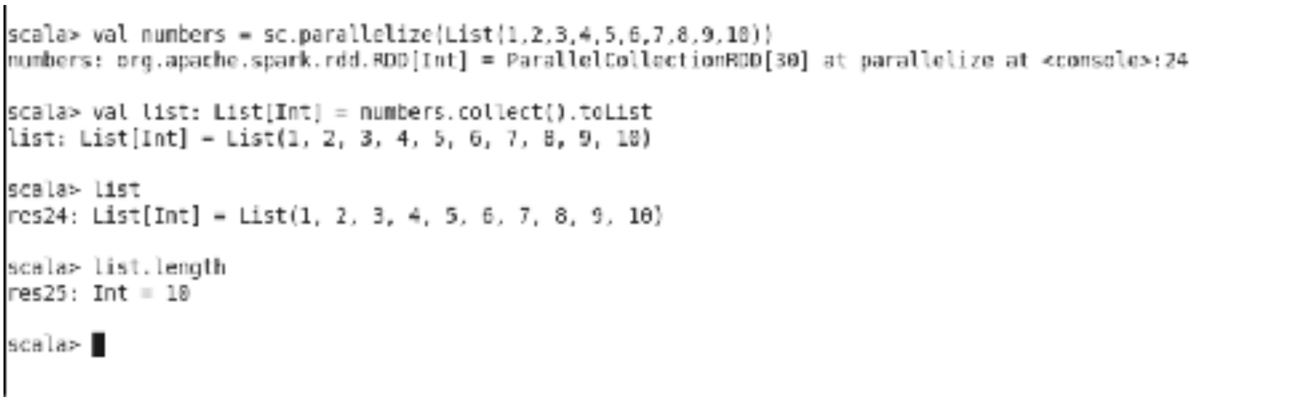
Given a list of numbers - List[Int] (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

- find the sum of all numbers



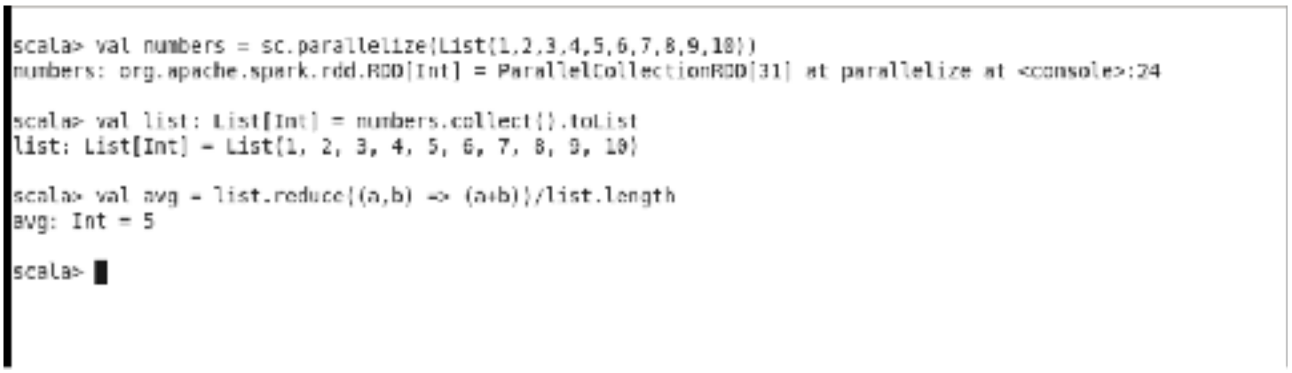
```
scala> val numbers = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
numbers: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[28] at parallelize at <co
nsole>:24

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

scala> val sum = list.reduce((a,b) => (a +b))
sum: Int = 55

scala> 
```

- find the total elements in the list



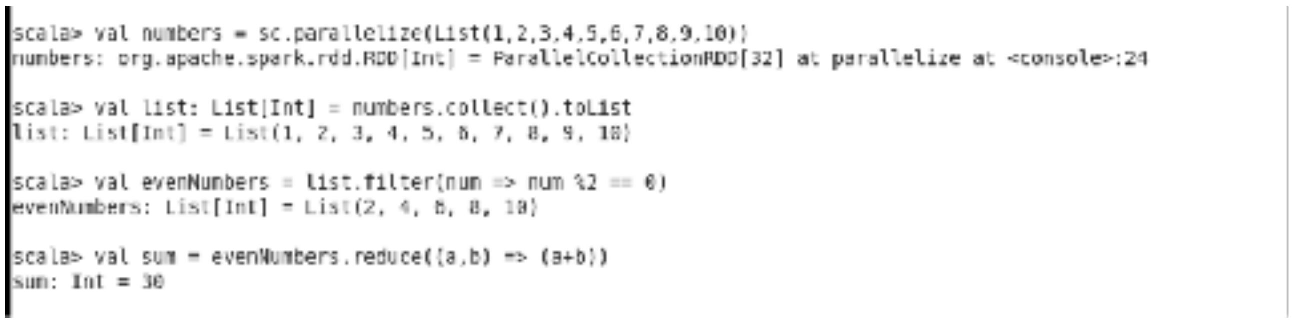
```
scala> val numbers = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
numbers: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[30] at parallelize at <console>:24

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

scala> list
res24: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> list.length
res25: Int = 10

scala> 
```

- calculate the average of the numbers in the list



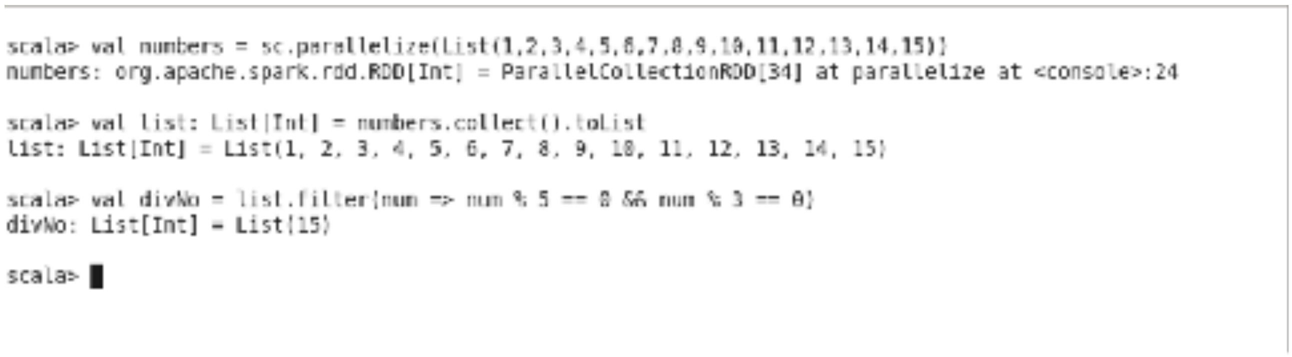
```
scala> val numbers = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
numbers: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[31] at parallelize at <console>:24

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

scala> val avg = list.reduce((a,b) => (a+b))/list.length
avg: Int = 5

scala> 
```

find the sum of all the even numbers in the list



```
scala> val numbers = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
numbers: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[32] at parallelize at <console>:24

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

scala> val evenNumbers = list.filter(num => num %2 == 0)
evenNumbers: List[Int] = List(2, 4, 6, 8, 10)

scala> val sum = evenNumbers.reduce((a,b) => (a+b))
sum: Int = 30
```

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



```
scala> val numbers = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15))
numbers: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[34] at parallelize at <console>:24

scala> val list: List[Int] = numbers.collect().toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)

scala> val divNo = list.filter(num => num % 5 == 0 && num % 3 == 0)
divNo: List[Int] = List(15)

scala> █
```

**Task 2**

1) Pen down the limitations of MapReduce.

- Issue with 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 NameNode will be overloaded since it stores the namespace of HDFS.

Solution to this Drawback of Hadoop to deal with small file issue is simple. Just merge the small files to create bigger files and then copy bigger files to HDFS.

- 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.

Solution to this Limitation of Hadoop spark has overcome this issue, by in-memory processing of data. In-memory processing is faster as no time is spent in moving the data/processes in and out of the disk. Spark is 100 times faster than MapReduce as it processes everything in memory. Flink is also used, as it processes faster than spark because of its streaming architecture and Flink may be instructed to process only the parts of the data that have actually changed, thus significantly increases the performance of the job.

- 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.

To solve these limitations of Hadoop spark is used that improves the performance, but Spark stream processing is not as much efficient as Flink as it uses micro-batch processing. Flink improves the overall performance as it provides single run-time for the streaming as well as batch processing. Flink uses native closed loop iteration operators which make machine learning and graph processing faster.

- 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.
Solution-

- Apache Spark supports stream processing. Stream processing involves continuous input and output of data. It emphasizes on the velocity of the data, and data is processed within a small period of time. Learn more about Spark Streaming APIs.
- Apache Flink provides single run-time for the streaming as well as batch processing, so one common run-time is utilized for data streaming application and batch processing application. Flink is a stream processing system that is able to process row after row in real time.

- 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).
Solution-
Apache Spark can be used to overcome this type of Limitations of Hadoop, as it accesses data from RAM instead of disk, which dramatically improves the performance of iterative algorithms that access the same dataset repeatedly. Spark iterates its data in batches. For iterative processing in Spark, each iteration has to be scheduled and executed separately.

- 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.
Solution-
Spark is used to reduce this limitation of Hadoop, Apache spark is yet another batch system but it is relatively faster since it caches much of the input data on memory by RDD(Resilient Distributed Dataset) and keeps intermediate data in memory itself. Flink's data streaming achieves low latency and high throughput.

- 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. To solve this Drawback of Hadoop, we can use spark. Spark has interactive mode so that developers and users alike can have intermediate feedback for queries and other action. Spark is easy to program as it has tons of high-level operators. Flink can also be easily used as it also has high-level operators. This way spark can solve many limitations of Hadoop.

- Security
Hadoop can be challenging in managing the complex application. If the user doesn't know how to enable platform who is managing the platform, your data could be at huge risk. At storage and network levels, Hadoop is missing encryption, which is a major point of concern. Hadoop supports Kerberos authentication, which is hard to manage. HDFS supports access control lists (ACLs) and a traditional file permissions model. However, third party vendors have enabled an organization to leverage Active Directory Kerberos and LDAP for authentication.
Solution-
Spark provides security bonus to overcome these limitations of Hadoop. If we run spark in HDFS, it can use HDFS ACLs and file-level permissions. Additionally, Spark can run on YARN giving it the capability of using Kerberos authentication.

- 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.
Solution-
To overcome these Drawback of Hadoop, Spark is used in which we have RDD abstraction for batch. Flink has Dataset abstraction.

- Vulnerable by Nature
Hadoop is entirely written in java, a language most widely used, hence java been most heavily exploited by cyber criminals and as a result, implicated in numerous security breaches.

- 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.
Solution-
Spark and Flink can overcome this limitation of hadoop, as Spark and Flink cache data in memory for further iterations which enhance the overall performance.

- 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.
Solution-
Although Spark and Flink are written in scala and java but they are implemented in Scala, so the number of line of code is lesser than Hadoop. So it will also take less time to execute the program and solve the lengthy line of code limitations of Hadoop.

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.

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.

Hence, 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 posses self-recovery in the case of failure.
There are three ways to create RDDs in Spark such as – Data in stable storage, other RDDs, and parallelizing already existing collection in driver program. One can also operate Spark RDDs in parallel with a low-level API that
offers transformations and actions. We will study these Spark RDD Operations later in this section.

Spark RDD can also be cached and manually partitioned. Caching is beneficial when we use RDD several times. And manual partitioning is important to correctly balance partitions. Generally, smaller partitions allow distributing RDD data more equally, among more executors. Hence, fewer partitions make the work easy.
Programmers can also call a persist method to indicate which RDDs they want to reuse in future operations. Spark keeps persistent RDDs in memory by default, but it can spill them to disk if there is not enough RAM. Users can also request other persistence strategies, such as storing the RDD only on disk or replicating it across machines, through flags to persist.

The key motivations behind the concept of RDD are-

- Iterative algorithms.
- Interactive data mining tools.
- DSM (Distributed Shared Memory) is a very general abstraction, but this generality makes it harder to implement in an efficient and fault tolerant manner on commodity clusters. Here the need of RDD comes into the picture.
- In distributed computing system data is stored in intermediate stable distributed store such as HDFS or Amazon S3. This makes the computation of job slower since it involves many IO operations, replications, and serializations in the process.

In first two cases we keep data in-memory, it can improve performance by an order of magnitude.
The main challenge in designing RDD is defining a program interface that provides fault tolerance efficiently. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformation rather than fine-grained updates to shared state.
Spark exposes RDD through language integrated API. In integrated API each data set is represented as an object and transformation is involved using the method of these objects.
Apache Spark evaluates RDDs lazily. It is called when needed, which saves lots of time and improves efficiency. The first time they are used in an action so that it can pipeline the transformation. Also, the programmer can call a persist method to state which RDD they want to use in future operations.
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 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.

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.

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.

Spark RDD Operations
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.

### 1.2Wide 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.

## 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.

## Limitation of Spark RDD

### 1. No inbuilt optimization engine

When working with structured data, RDDs cannot take advantages of Spark's advanced optimizers including catalyst optimizer and Tungsten execution engine. Developers need to optimize each RDD based on its attributes.

### 2. Handling structured data

Unlike Dataframe and datasets, RDDs don't infer the schema of the ingested data and requires the user to specify it.

### 3. Performance limitation

Being in-memory JVM objects, RDDs involve the overhead of Garbage Collection and Java Serialization which are expensive when data grows.

4. Storage limitation
RDDs degrade when there is not enough memory to store them. One can also store that partition of RDD on disk which does not fit in RAM. As a result, it will provide similar performance to current data-parallel systems.