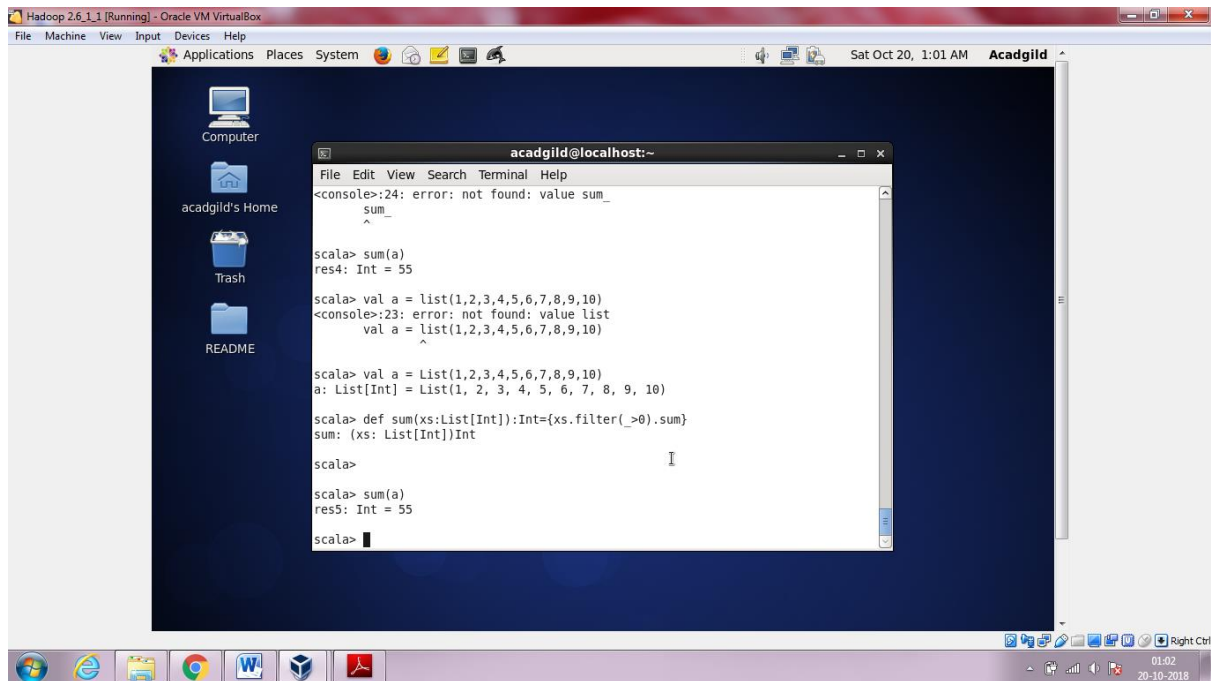


Task1

List[Int](1,2,3,4,5,6,7,8,9,10)

-find the sum of all members



The screenshot shows a Scala REPL window titled 'acadgild@localhost:~' within an Oracle VM VirtualBox environment. The window displays the following code and output:

```
File Edit View Search Terminal Help
<console>:24: error: not found: value sum_
  sum_
  ^

scala> sum(a)
res4: Int = 55

scala> val a = list(1,2,3,4,5,6,7,8,9,10)
<console>:23: error: not found: value list
  val a = list(1,2,3,4,5,6,7,8,9,10)
        ^

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

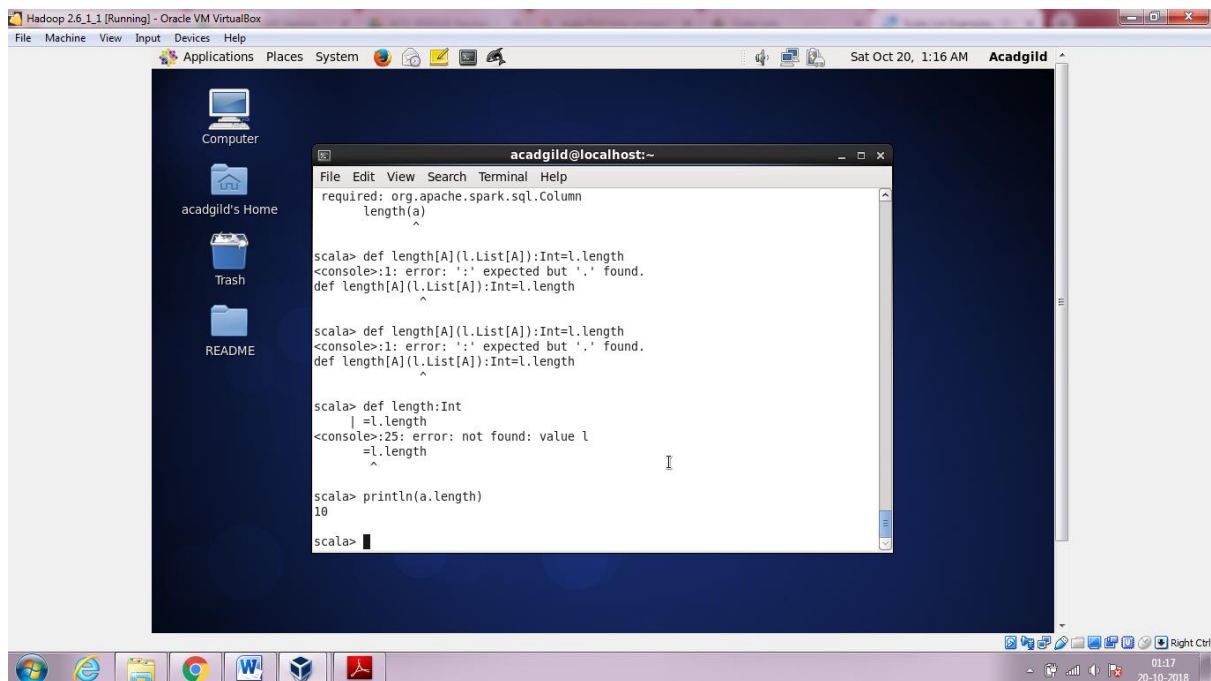
scala> def sum(xs:List[Int]):Int={xs.filter(_>0).sum}
sum: (xs: List[Int])Int

scala>

scala> sum(a)
res5: Int = 55

scala>
```

-find total element in list.



The screenshot shows a Scala REPL window titled 'acadgild@localhost:~' within an Oracle VM VirtualBox environment. The window displays the following code and output:

```
File Edit View Search Terminal Help
required: org.apache.spark.sql.Column
length(a)
^

scala> def length[A](l:List[A]):Int=l.length
<console>:1: error: ':' expected but '.' found.
def length[A](l:List[A]):Int=l.length
              ^

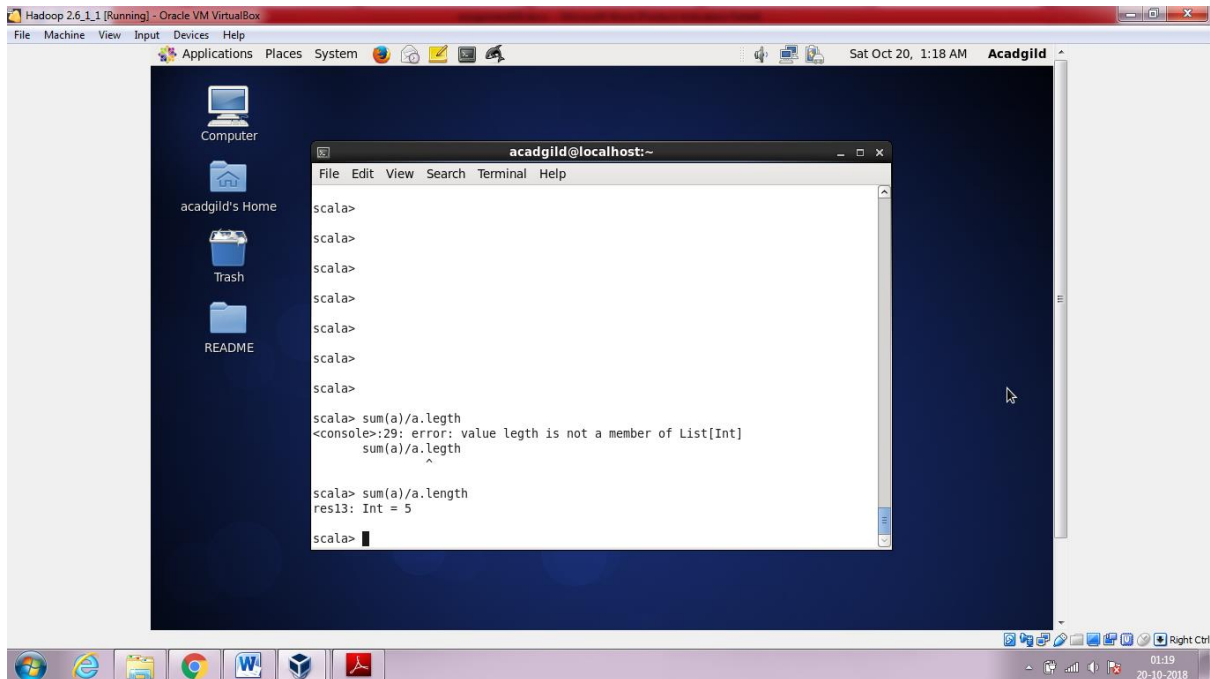
scala> def length[A](l:List[A]):Int=l.length
<console>:1: error: ':' expected but '.' found.
def length[A](l:List[A]):Int=l.length
              ^

scala> def length:Int
| =l.length
<console>:25: error: not found: value l
  =l.length
  ^

scala> println(a.length)
10

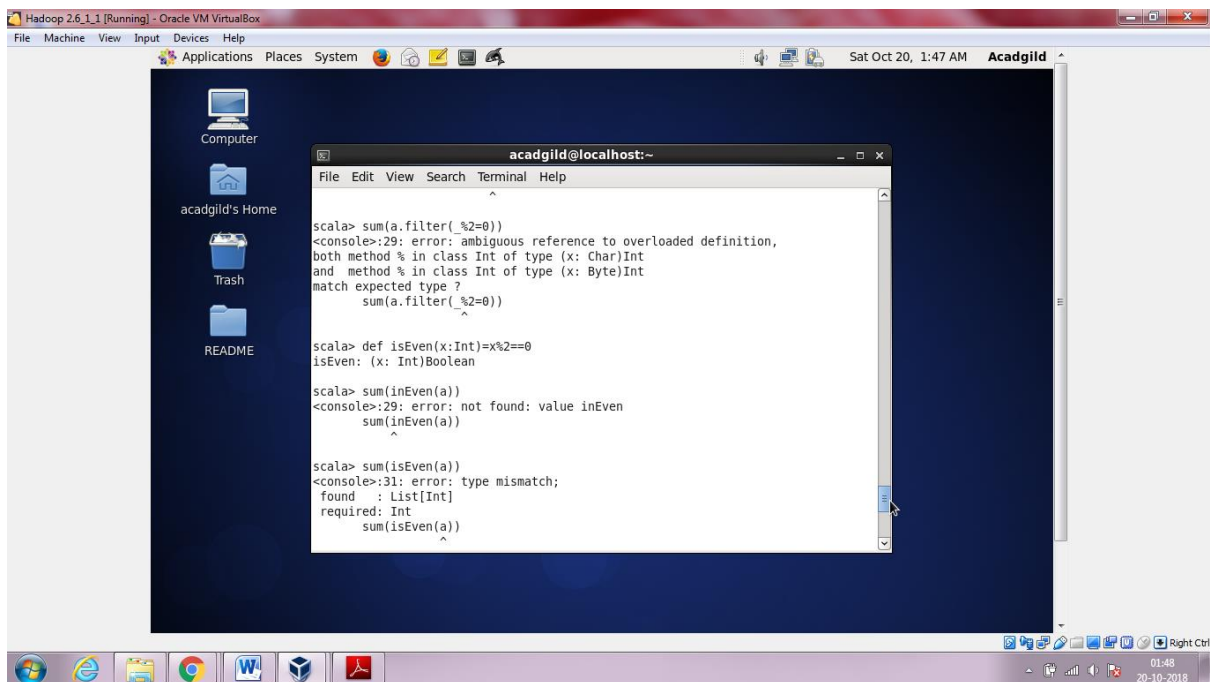
scala>
```

-calculate the average of the numbers in list

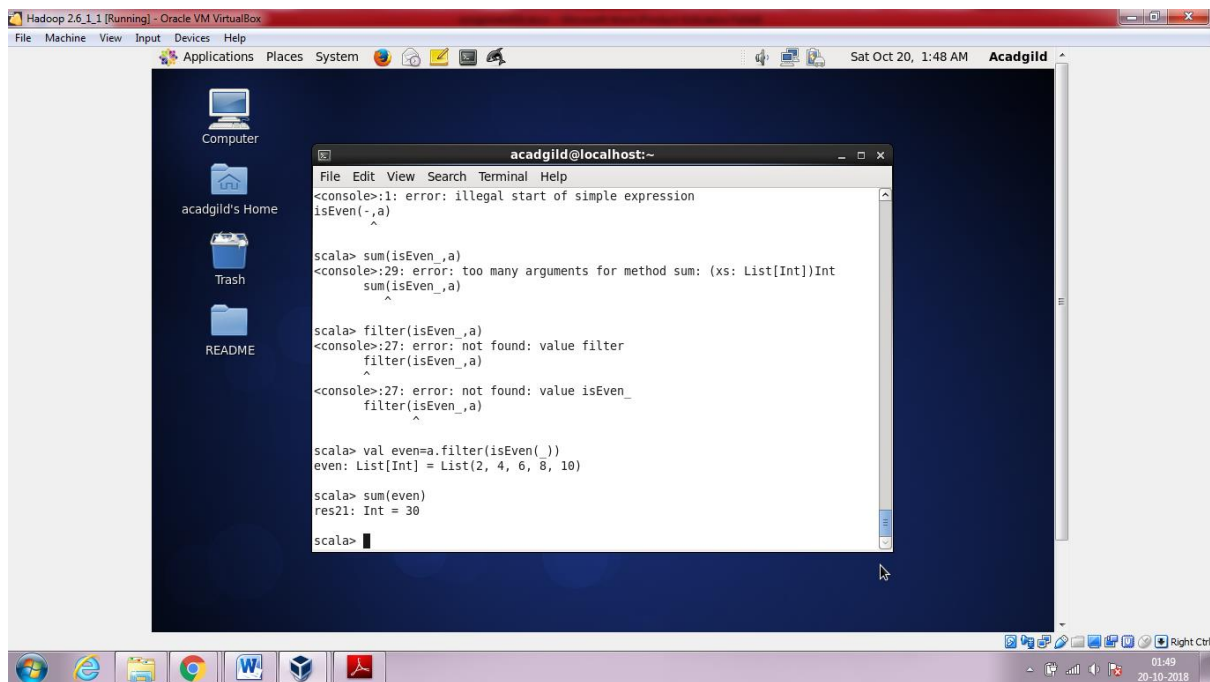


```
acadgild@localhost:~  
File Edit View Search Terminal Help  
scala>  
scala>  
scala>  
scala>  
scala>  
scala>  
scala>  
scala>  
scala> sum(a)/a.length  
<console>:29: error: value length is not a member of List[Int]  
      sum(a)/a.length  
                ^  
scala> sum(a)/a.length  
res13: Int = 5  
scala>
```

-find sum of all even numbers in list



```
acadgild@localhost:~  
File Edit View Search Terminal Help  
scala> sum(a.filter(_%2==0))  
<console>:29: error: ambiguous reference to overloaded definition,  
both method % in class Int of type (x: Char)Int  
and method % in class Int of type (x: Byte)Int  
match expected type ?  
      sum(a.filter(_%2==0))  
                ^  
scala> def isEven(x: Int) = x % 2 == 0  
isEven: (x: Int)Boolean  
scala> sum(isEven(a))  
<console>:29: error: not found: value isEven  
      sum(isEven(a))  
            ^  
scala> sum(isEven(a))  
<console>:31: error: type mismatch;  
found   : List[Int]  
required: Int  
      sum(isEven(a))  
            ^
```



Task2:

- 1) Pen down the limitations of MapReduce.

Ans:-Processing speed

MapReduce algorithm contains 2 important task i.e.map and reduce.It requires lot of time to perform task thereby increase latency.Data is distributed and processed over the cluster.

Catching

MapReduce cannot cache the intermediate data in-memory for a further requirement which diminishes the performance of Hadoop

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

Ease of 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 add one such as hive and pig, make working with MapReduce a little easier for adopters.

Latency

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)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.

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.

Features of RDD

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

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.

There are 2 types of operations

1.Transformation:

Transformations create a new data set from an existing one by passing each dataset element through a function and returns a new RDD representing the results. In short, creating an RDD from an existing RDD is 'transformation'.

All transformations in Spark are *lazy*. They do not compute their results right away.

Instead, they just remember the transformations applied to some base data set (e.g. a file). The transformations are only computed when an action requires a result that needs to be returned to the driver program.E.g.Filter is a transformation operation of rdd.

2.Actions:

Actions return final results of RDD computations. Actions trigger execution using lineage graph to load the data into original RDD and carry out all intermediate transformations and return the final results to the Driver program or writes it out to the file system.

E.g.Count is a action operation of rdd.