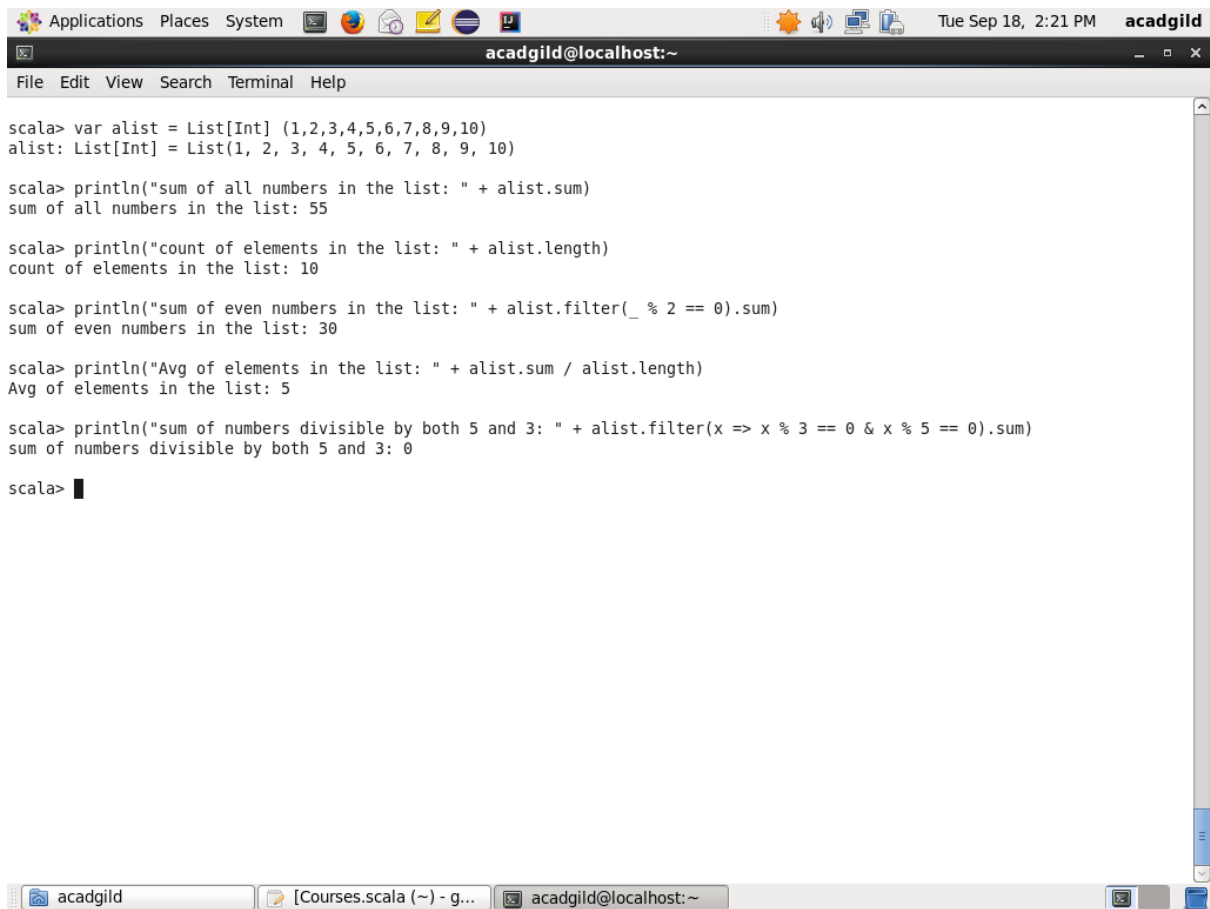


Task 1

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



The screenshot shows a terminal window titled 'acadgild@localhost:~' with a menu bar (File, Edit, View, Search, Terminal, Help) and a system bar at the top (Applications, Places, System, Tue Sep 18, 2:21 PM, acadgild). The terminal contains the following Scala code and its output:

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

scala> println("sum of all numbers in the list: " + alist.sum)
sum of all numbers in the list: 55

scala> println("count of elements in the list: " + alist.length)
count of elements in the list: 10

scala> println("sum of even numbers in the list: " + alist.filter(_ % 2 == 0).sum)
sum of even numbers in the list: 30

scala> println("Avg of elements in the list: " + alist.sum / alist.length)
Avg of elements in the list: 5

scala> println("sum of numbers divisible by both 5 and 3: " + alist.filter(x => x % 3 == 0 & x % 5 == 0).sum)
sum of numbers divisible by both 5 and 3: 0

scala> █
```

The terminal window is part of a desktop environment with a taskbar at the bottom showing three open windows: 'acadgild', '[Courses.scala (~) - g...', and 'acadgild@localhost:~'.

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.
- Latency: MR persists back to disk after a map-reduce job and the process introduces latency.
- Suitable for Batch Processing only: Because of latency issues, Hadoop is best suited for batch processing and not for processing large volumes of streamed or real-time data.
- No Delta Iteration: Hadoop is not so efficient for iterative processing, as it does not support cyclic data flow. Jobs run in isolation.
- It's not always very easy to implement each and everything as a MR program.
- MR is not suitable for a large number of short on-line transactions (OLTP).
- It cannot handle Graph Processing.
- Unlike Spark, MR doesn't have an interactive mode.

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 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. 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 (e.g., in-memory storage or on Disk).

3) List down few Spark RDD operations and explain each of them.

Two types of Apache Spark RDD operations are - **Transformations and Actions**:

- 1) **Spark Transformation** is a function that produces a new RDD from the existing RDDs. It takes an RDD as input and produces one or more RDDs as output. It creates a new RDD when we apply a transformation. Thus, the input RDDs are not changed since RDDs are immutable in nature.
 - a. **Map**: The map function iterates over every line in a RDD and splits it into new RDDs. Using map() transformation, we can take in any function, and that function is applied to every element of the RDD.

For example, in RDD {1, 2, 3, 4, 5} if we apply "rdd.map(x=>x+2)" we will get the result as (3, 4, 5, 6, 7).
 - b. **Filter**: Spark filter() function returns a new RDD containing only the elements that meet a predicate.
 - c. **groupByKey()**: When we use groupByKey() on a dataset of (K, V) pairs, the data is shuffled according to the key value K in another RDD.
 - d. **reduceByKey**: When we use reduceByKey on a dataset (K, V), the pairs on the same machine with the same key are combined before the data is shuffled.
- 2) **Actions**:
 - a. **count()**: Action count() returns the number of elements in the RDD.
 - b. **collect()**: The action collect() is the most common and the simplest operation that returns our entire RDDs content to the driver program.
 - c. **reduce()**: The reduce() function takes two elements as input from the RDD and then produces the output of the same type as that of the input elements. The simple form of such function is addition.
 - d. **countByValue**: The countByValue() returns the number of times an element occurs in the RDD.
 - e. **foreach()**: When we have a situation where we want to apply operation on each element of RDD, but it should not return value to the driver, foreach() function is useful.

Task 3

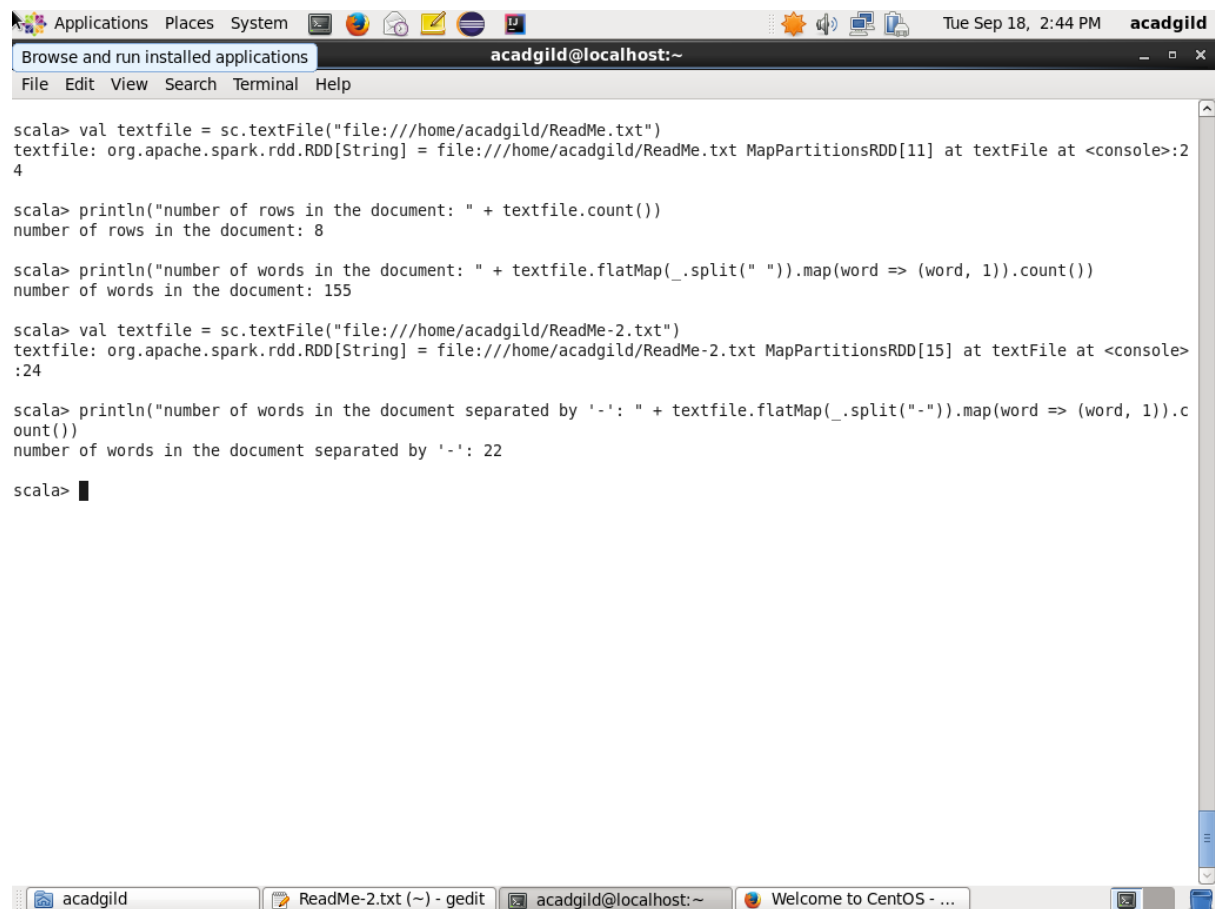
1. Write a program to read a text file and print the number of rows of data in the document.
2. Write a program to read a text file and print the number of words in the document.
3. We have a document where the word separator is -, instead of space. Write a spark code, to obtain the count of the total number of words present in the document.

Sample document :

This-is-my-first-assignment.

It-will-count-the-number-of-lines-in-this-document.

The-total-number-of-lines-is-3



The screenshot shows a Linux desktop with a terminal window titled 'acadgild@localhost:~'. The terminal displays the following Scala code and its output:

```
scala> val textfile = sc.textFile("file:///home/acadgild/ReadMe.txt")
textfile: org.apache.spark.rdd.RDD[String] = file:///home/acadgild/ReadMe.txt MapPartitionsRDD[11] at textFile at <console>:24

scala> println("number of rows in the document: " + textfile.count())
number of rows in the document: 8

scala> println("number of words in the document: " + textfile.flatMap(_.split(" ")).map(word => (word, 1)).count())
number of words in the document: 155

scala> val textfile = sc.textFile("file:///home/acadgild/ReadMe-2.txt")
textfile: org.apache.spark.rdd.RDD[String] = file:///home/acadgild/ReadMe-2.txt MapPartitionsRDD[15] at textFile at <console>:24

scala> println("number of words in the document separated by '-': " + textfile.flatMap(_.split("-")).map(word => (word, 1)).count())
number of words in the document separated by '-': 22

scala>
```

The desktop environment includes a taskbar at the bottom with icons for 'acadgild', 'ReadMe-2.txt (~) - gedit', 'acadgild@localhost:~', and 'Welcome to CentOS - ...'. The top of the window shows system icons and the date 'Tue Sep 18, 2:44 PM'.