Task 1

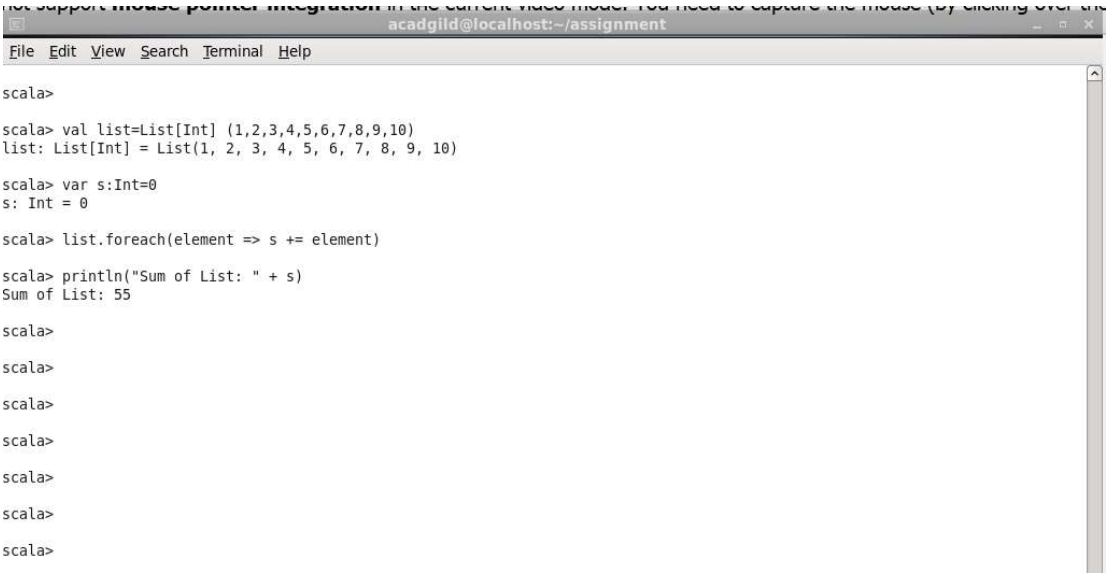
A: Find sum of numbers in the list

- Initialize a variable s used for calculating sum to 0
- For each element in List
- Add element to sum
- Print s

The scala code is as below:

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

var s:Int = 0

list.foreach(element => sum += element)

println("Sum of List" + sum)
```

Screenshot is:



B: Find number of element in the list

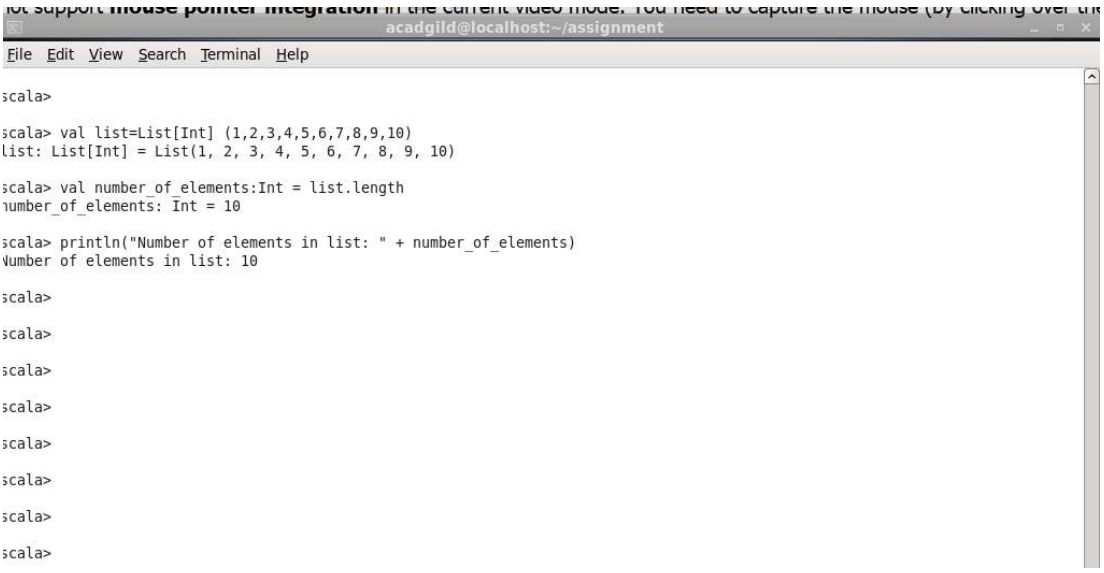
- Define a variable number_of_elements
- Assign length of list to number_of_elements
- Print number_of_elements

The scala code is as below:

```
val total_elements = list.length
```

println(total_elements)

Screenshot is as below:



C: Find average of all the numbers

- Write a function average which takes list as input and return average as output
- Implementation of function average is as below:
- check if list is empty then throw Exception
- Initialize a variable s used for calculating sum to
- Iterate over the list and add element to sum
- Divide sum by number of elements in list and return

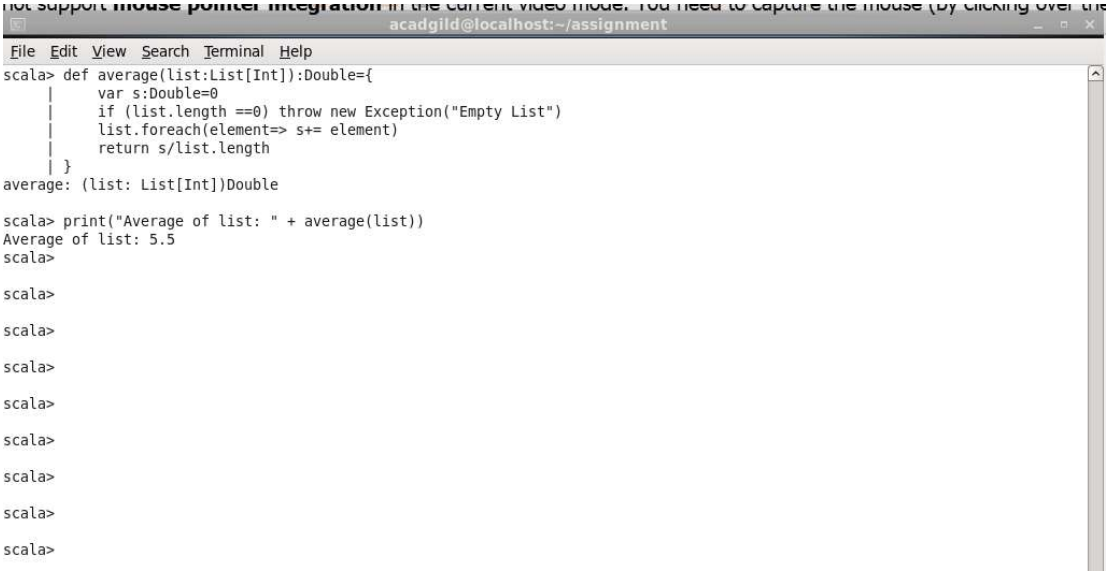
Code is as below:

```
def average(list:List[Int]):Double={
   var s:Double = 0
   if (list.length ==0) throw new Exception(“Empty List”)
    list.foreach(element => s+= element)
    return s/list.length
```

}

println(“Average of list: “ + average(list))

Screenshot is as below:



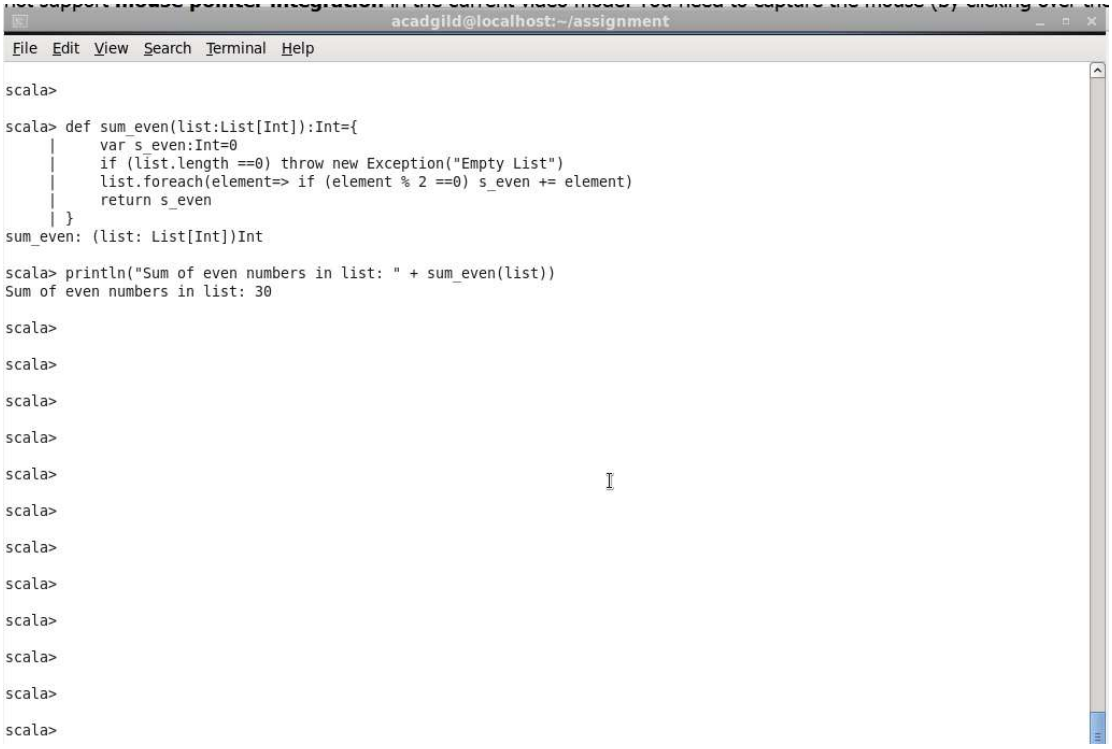
D: Find sum of even numbers in the list

- Write a function sum_even which takes list as input and return sum of even numbers
- Implementation of function average is as below:
- check if list is empty then throw Exception
- Initialize a variable s_even used for calculating sum to 0
- Iterate over the list and adding only those elements which is even (modulo 2 is 0)
- Return s_even

Scala code is as below:

```
def sum_even(list:List[Int]):Double={
  var s_even: Int = 0
  if (list.length ==0) throw new Exception(“Empty List”)
    list.foreach(element => if (element % 2 == 0) s_even += element)
    return s_even
}
```

Screenshot is as below:



E: Find count of numbers divisible by both 3 and 5

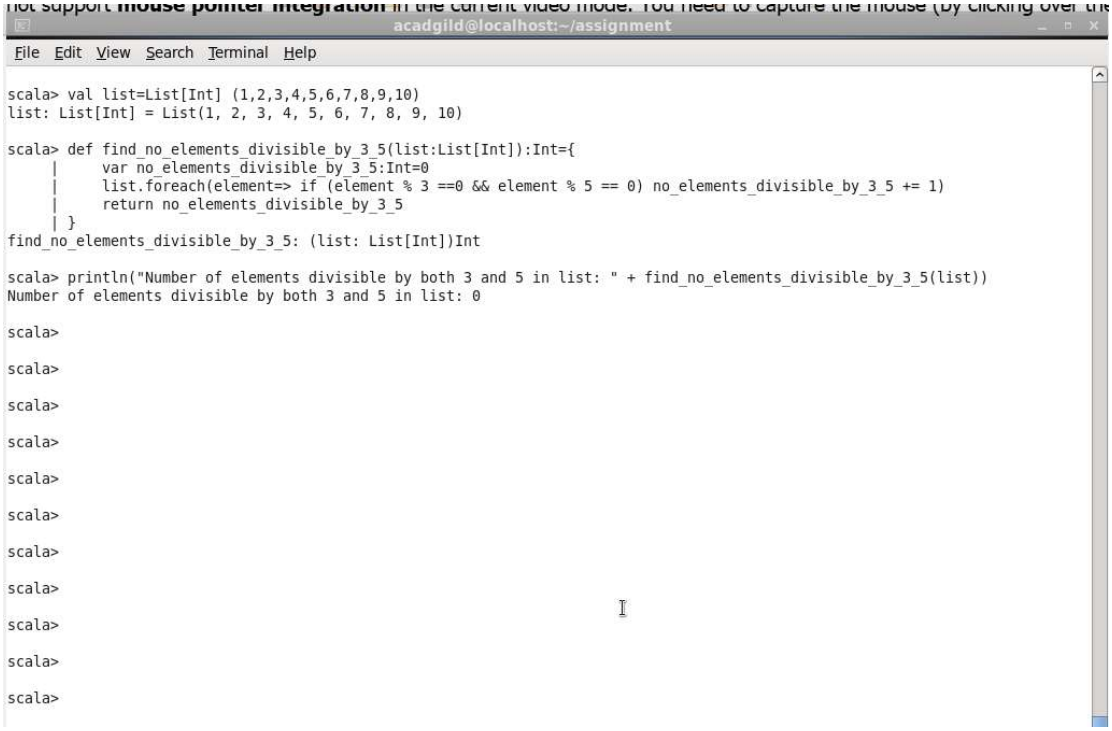
- Write a function find_no_elements_divisible_by_3_5 which takes list as input and return count of numbers divisible by both 3 and 5
- Implementation of function average is as below:
- check if list is empty then throw Exception
- Initialize a variable number_elements_divisible_by_3_5 to 0
- Iterate over the list and for each element which is divisible by both 3 and 5 increment variable number_elements_divisible_by_3_5 by 1
- Return number_elements_divisible_by_3_5

Scala code is as below:

```
def find_no_elements_divisible_by_3_5(list:List[Int]):Int={
  var no_elements_divisible_by_3_5: Int = 0
  if (list.length ==0) throw new Exception("Empty List")
   list.foreach(element => if (element % 3 == 0 && element % 5 == 0) no_elements_divisible_by_3_5 += 1)
```

```
    return no_elements_divisible_by_3_5
}
```

Screenshot is as below:



Task 2

1) Pen down the limitations of MapReduce.

Limitation of Map Reduce are:

i. **Slow Disk Based computation**: It is based on disk based computation which makes computation jobs slower

ii. **Not suitable for iterative computation**: It is only meant for single pass computation, not iterative computations. It requires a sequence of Map Reduce jobs to run iterative task

iii. **Can not perform non parallelized computation**: Tasks that has a dependency on each other cannot be parallelized, which is not possible through MapReduce

iv. **Does not work on non partitionable/recombinable Problems**: problems that cannot be trivially partitionable or recombinable can not be solved with Map Reduce e.g. Travelling Salesman problem.

v. **Needs integration with several tools**: Map reduce requires integration with different tools to solve big data usecases. E.g. Mahout for Machine

2) What is RDD. Specify a few features of RDD

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Features of RDD:

i. **Resilient**, i.e. fault-tolerant with the help of RDD lineage graph and so able to recompute missing or damaged partitions due to node failures
ii. **Distributed** with data residing on multiple nodes in a cluster.
iii. **Dataset** is a collection of partitioned data with primitive values or values of values, e.g. tuples or other objects
iv. **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.
v. **Immutable or Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs.
vi. **Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.
vii. **Cacheable**, i.e. we can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).
viii. **IParallel**, i.e. process data in parallel.
ix. **Typed** — RDD records have types, e.g. Long in RDD[Long] or (Int, String) in RDD[(Int, String)].
x. **Partitioned** — records are partitioned (split into logical partitions) and distributed across nodes in a cluster.
xi. **Location-Stickiness** — RDD can define placement preferences to compute partitions (as close to the records as possible).

3) List a number of RDD operations and explain each of them

i. **map**: The map function iterates over every line in RDD and split into new RDD. Using **map()** transformation we take in any function, and that function is applied to every element of RDD.
Example:
tupleRDD has 4 fields (name, subject, grade, marks). Using map operations two fields are taken (subject, marks)

```
val studentMarksSubjectRDD = tupleRDD.map(t=> (t._2, t._4)
```

ii. **flatMap**: With the help of **flatMap()** function, to each input element, we have many elements in an output RDD. The most simple use of flatMap() is to split each input string into words. flatMap returns a collection of elements

Example:
Tis example takes a input file and split them into words

```
val rdd = sc.textFile("/home/acadgild/assignment_17.1/wordcount_input_file")

val rdd_words = rdd.flatMap(line=> line.split(" "))
```

iii. **filter**: Spark RDD **filter()** function returns a new RDD, containing only the elements that meet a predicate. It is a *narrow operation* because it does not shuffle data from one partition to many partitions.

Example:

tupleRDD has 4 fields (name, subject, grade, marks). Using filter operation only students who are in grade-2 taken

```
val grade2StudentRDD = tupleRDD.filter(t=> t._3 == "grade-2")
```

iv. **mapPartition**: The **MapPartition** converts each *partition* of the source RDD into many elements of the result (possibly none). In mapPartition(), the map() function is applied on each partitions simultaneously. MapPartition is like a map, but the difference is it runs separately on each partition(block) of the RDD.

v. **Union**: With the **union()** function, we get the elements of both the RDD in new RDD. The key rule of this function is that the two RDDs should be of the same type.

Example:

In the example rdd1 has three dates, rdd2 has two dates and rdd3 has two dates, union operation is done on rdd1, rdd2, rdd3 to get new RDD rddUnion

```
val rdd1 = parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014)))

val rdd2 = spark.sparkContext.parallelize(Seq((5,"dec",2014),(17,"sep",2015)))
val rdd3 = spark.sparkContext.parallelize(Seq((6,"dec",2011),(16,"may",2015)))
val rddUnion = rdd1.union(rdd2).union(rdd3)
rddUnion.foreach(Println)
```

vi. **Intersection:** With the **intersection()** function, we get only the common element of both the RDD in new RDD. The key rule of this function is that the two RDDs should be of the same type.

Example:

In the example rdd1 has three dates, rdd2 has two dates and rdd2 has two dates, intersection operation is done on rdd1, rdd2 to get new RDD rddCommon

```
val rdd1 = sc.parallelize(Seq((1,"jan",2016),(3,"nov",2014,
(16,"feb",2014)))
val rdd2 =
spark.sparkContext.parallelize(Seq((5,"dec",2014),(1,"jan",2016)))
val rddCommon = rdd1.intersection(rdd2)
rddCommon.foreach(Println)
```

vii. **Distinct**: It returns a new dataset that contains the **distinct** elements of the source dataset. It is helpful to remove duplicate data.

Example:

In this example tuple RDD is created from a file which has student records. Distict is used to get distinct tuples

```
val baseRDD = sc.textFile("/home/acadgild/assignment_17.2/17.2_Dataset.txt")

val tupleRDD = baseRDD.map(x => (x.split(",")(0), x.split(",")(1), x.split(",")(2), x.split(",")(3).toInt))

val distinctTupleRDD = tupleRDD.distinct
```

viii. **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.

Example:

In this example, distinctGradeStudentMapCountRDD has tuples with first elementa as key grade and second element as value marks. Using reduceByKey operations Marks for each grade are summed and put to gradeStudentCountRDD

```
val gradeStudentCountRDD = distinctGradeStudentMapCountRDD.reduceByKey((x, y) => x+y)
```

ix. **SortByKe**y: When we apply the **sortByKey() function** on a dataset of (K, V) pairs, the data is sorted according to the key K in another RDD.

Example:

```
val data =
spark.sparkContext.parallelize(Seq(("maths",52), ("english",75), ("science",82), ("computer",65),
("maths",85)))
val sorted = data.sortByKey()
sorted.foreach(println)
```

x. **Join:**

join() operation in Spark is defined on pair-wise RDD. Pair-wise RDDs are RDD in which each element is in the form of tuples. Where the first element is key and the second element is the value.

The advantage of using keyed data is that we can combine the data together. The join() operation combines two data sets on the basis of the key.

Example:

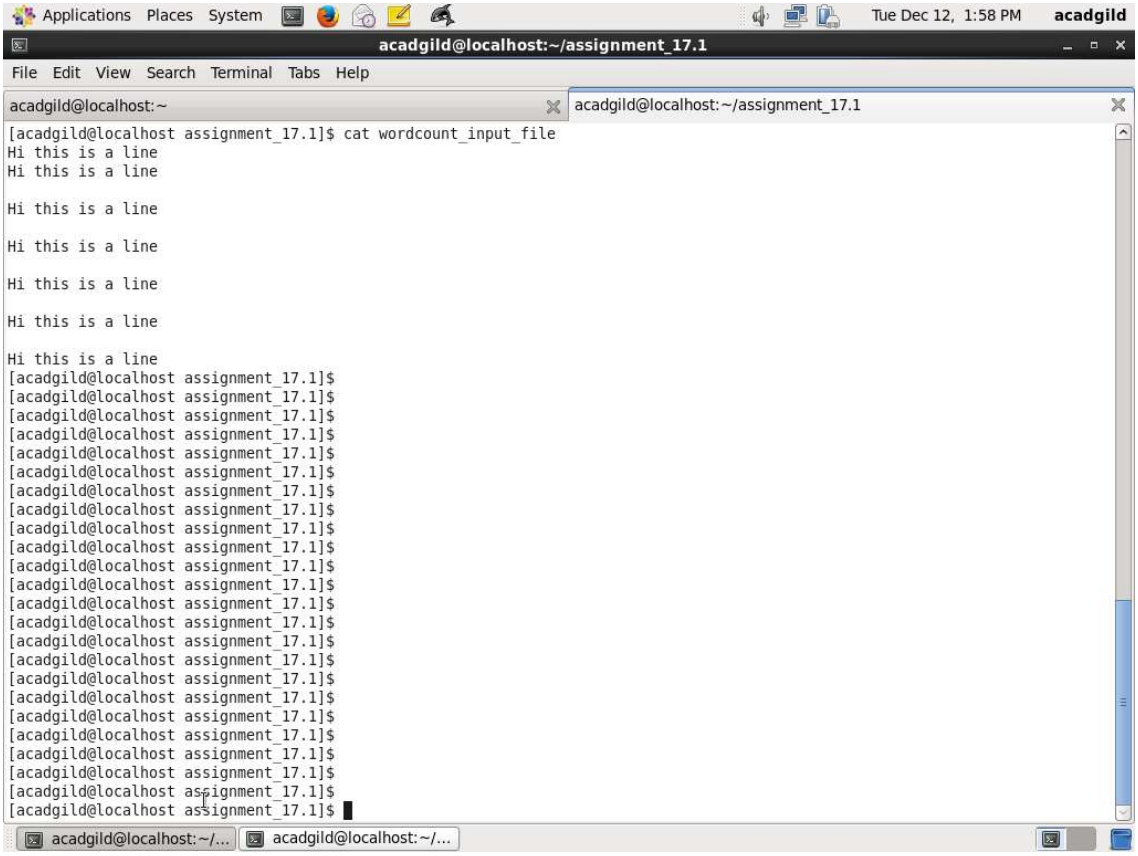
```
val data = spark.sparkContext.parallelize(Array(('A',1),('b',2),('c',3)))
val data2 =spark.sparkContext.parallelize(Array(('A',4),('A',6),('b',7),('c',3),('c',8)))
val result = data.join(data2)
println(result.collect().mkString(","))
```

Task 3

1. Write a program to read a text file and print the number of rows of data in the document.

Step1:

First copy the wordcount_input_file. Contents of file is displayed below



Step2:

Create a RDD using the text file. Count number of lines in rdd and put to line_count. Print line_count.

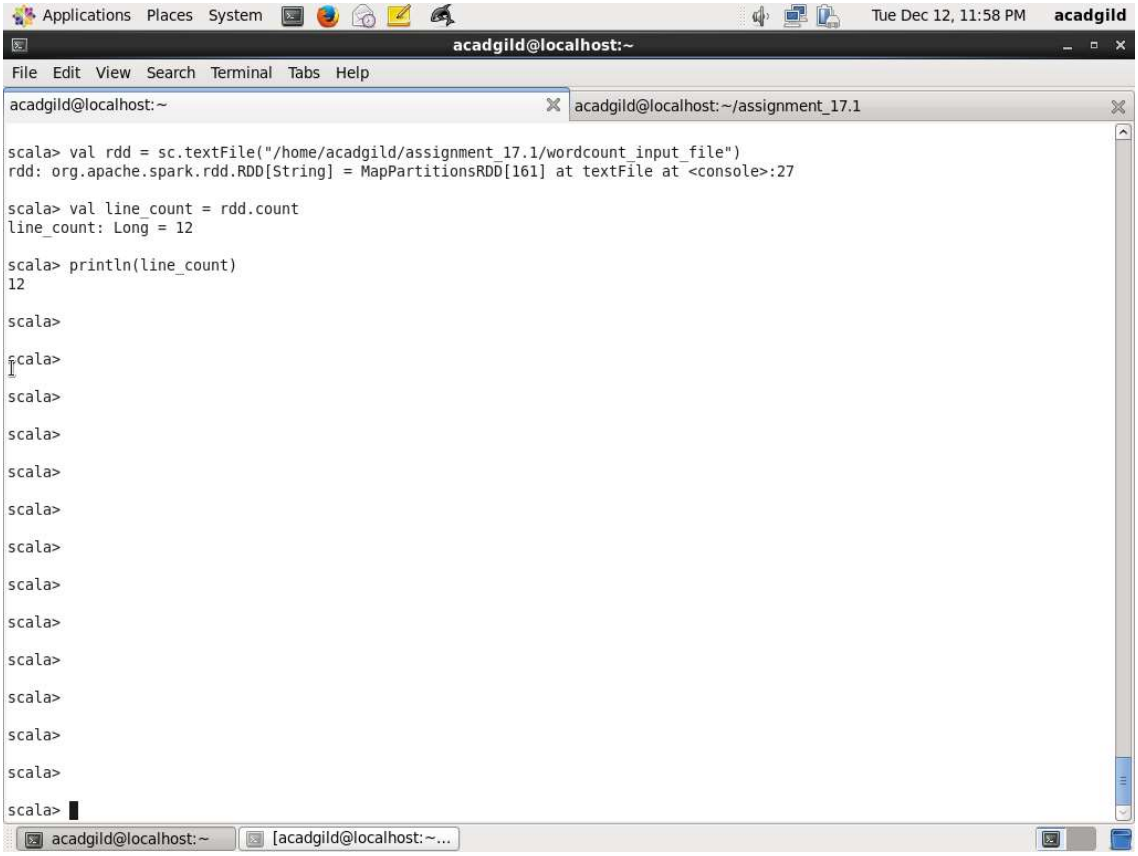
The code is as below:

val rdd = sc.textFile("/home/acadgild/assignment_17.1/wordcount_input_file")

val line_count = rdd.count

println(line_count)

Screenshot is as below:



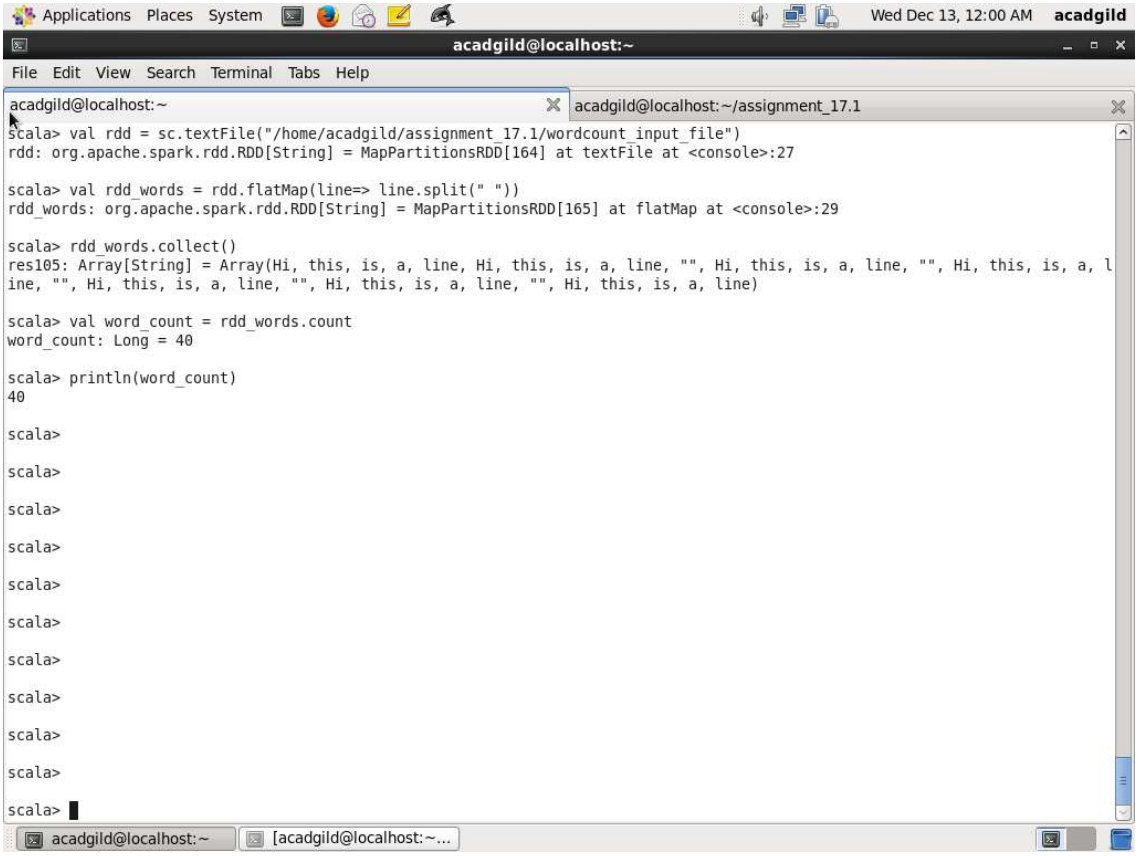
2. Write a program to read a text file and print the number of words in the document.

Create a rdd from the input file wordcount_input_file. Create a rdd_words by splitting line based on blank space and using flatMap on rdd. Count the number of words using count method and assign to word_count. Print word_count

Code is as below:

```
val rdd = sc.textFile("/home/acadgild/assignment_17.1/wordcount_input_file")

val rdd_words = rdd.flatMap(line=> line.split(" "))

rdd_words.collect()

val word_count = rdd_words.count

println(word_count)
```

Screenshot is as below:

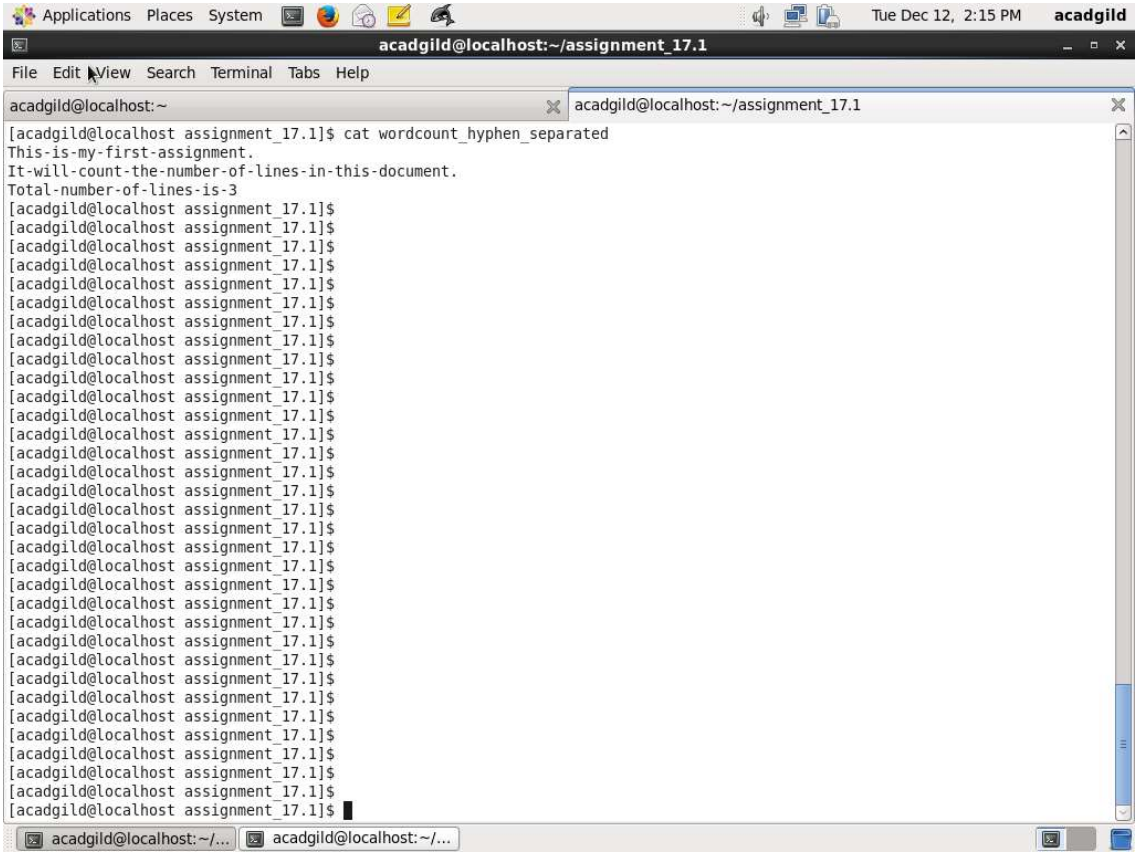


3. We have a document where the word separator is -, instead of space. Write a spark code, to obtain the count of the total number of words present in the document.

Step1:

Copy the file wordcount_hyphen_separated to /home/acadgild/assignment_17.1

Contents are as below:



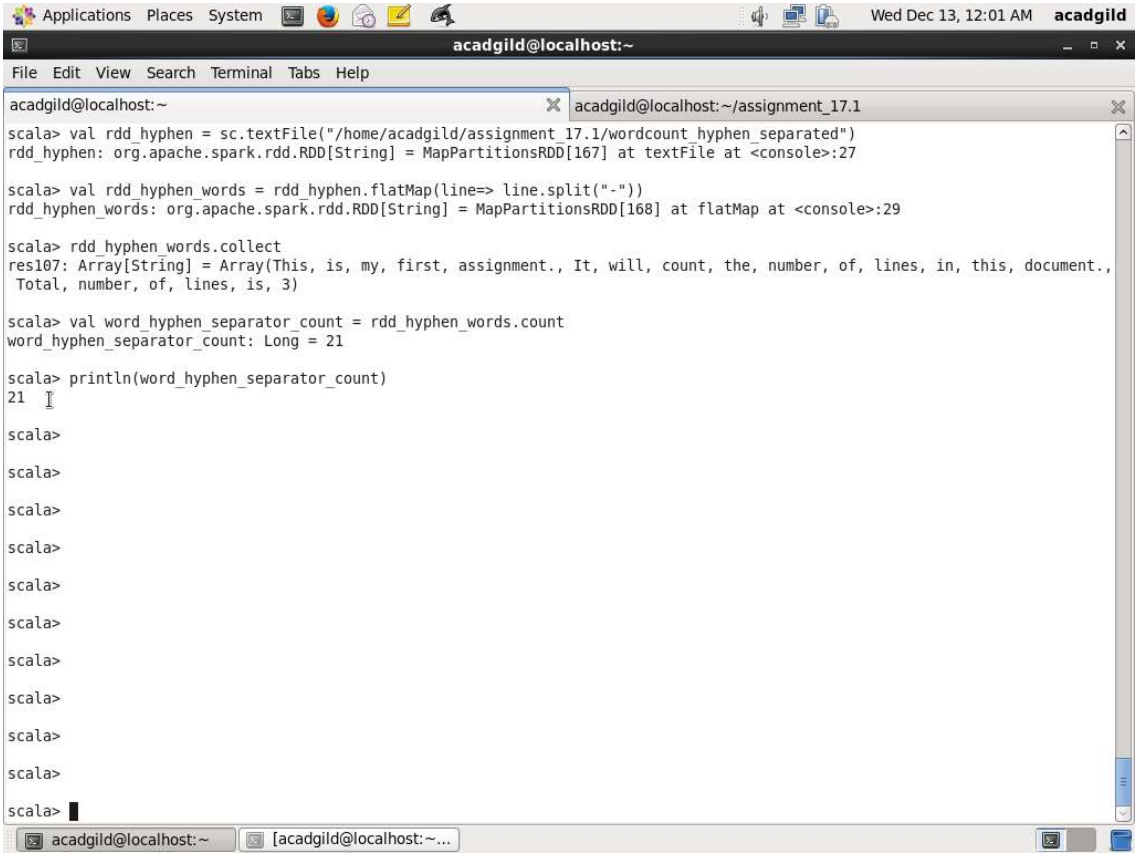
Step2:

Create a RDD rdd_hyphen from the wordcount_hyphen_separated file. Split each line based on separator hyphen (-) and use flatMap to combine the results. Use count method to get the count of works and assign to word_hyphen_separator_count and print word_hyphen_separator_count

Code is as below:

```
val rdd_hyphen = sc.textFile("/home/acadgild/assignment_17.1/wordcount_hyphen_separated")

val rdd_hyphen_words = rdd_hyphen.flatMap(line=> line.split("-"))

rdd_hyphen_words.collect

val word_hyphen_separator_count = rdd_hyphen_words.count

println(word_hyphen_separator_count)
```

Screenshot is as below:


Applications Places System
Wed Dec 13, 12:01 AM acadgild
acadgild@localhost:~
File Edit View Search Terminal Tabs Help
acadgild@localhost:~
acadgild@localhost:~/assignment_17.1
scala> val rdd_hyphen = sc.textFile("/home/acadgild/assignment_17.1/wordcount_hyphen_separated")
rdd_hyphen: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[167] at textFile at <console>:27
scala> val rdd_hyphen_words = rdd_hyphen.flatMap(line=> line.split("-"))
rdd_hyphen_words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[168] at flatMap at <console>:29
scala> rdd_hyphen_words.collect
res107: Array[String] = Array(This, is, my, first, assignment., It, will, count, the, number, of, lines, in, this, document.,
Total, number, of, lines, is, 3)
scala> val word_hyphen_separator_count = rdd_hyphen_words.count
word_hyphen_separator_count: Long = 21
scala> println(word_hyphen_separator_count)
21
scala>
acadgild@localhost:~
[acadgild@localhost:~...