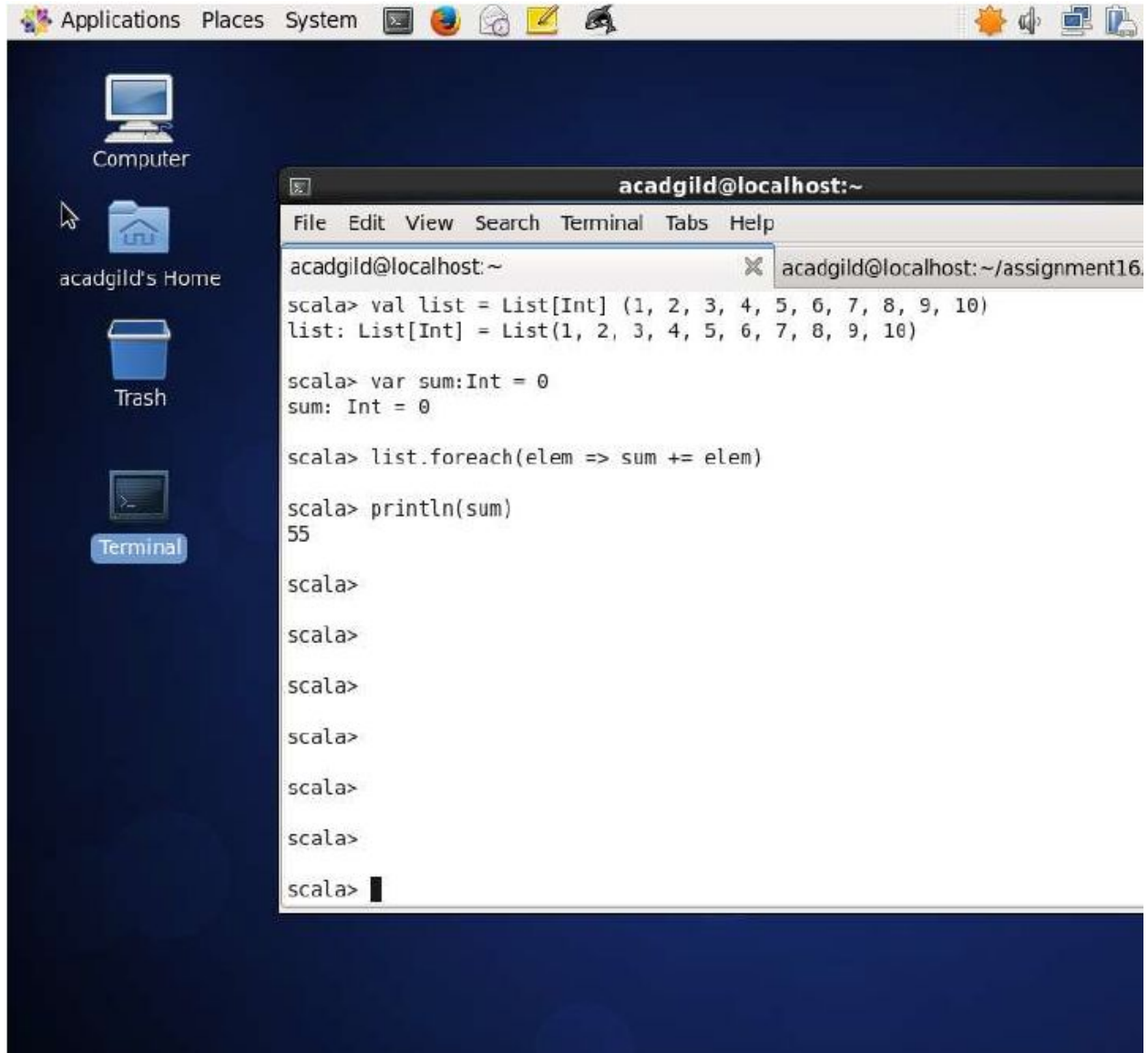


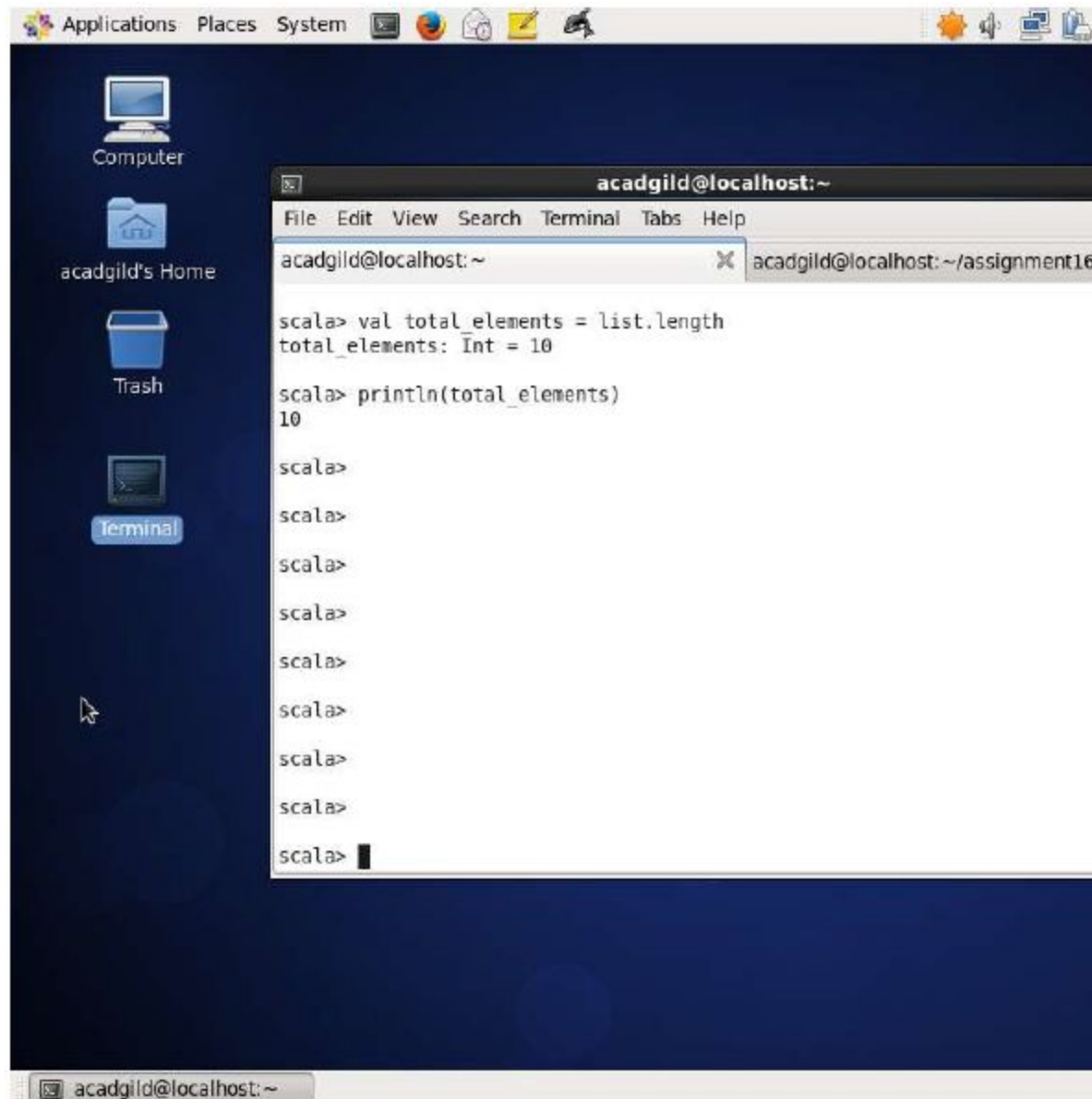
Task 1

A: Find sum of numbers in the list

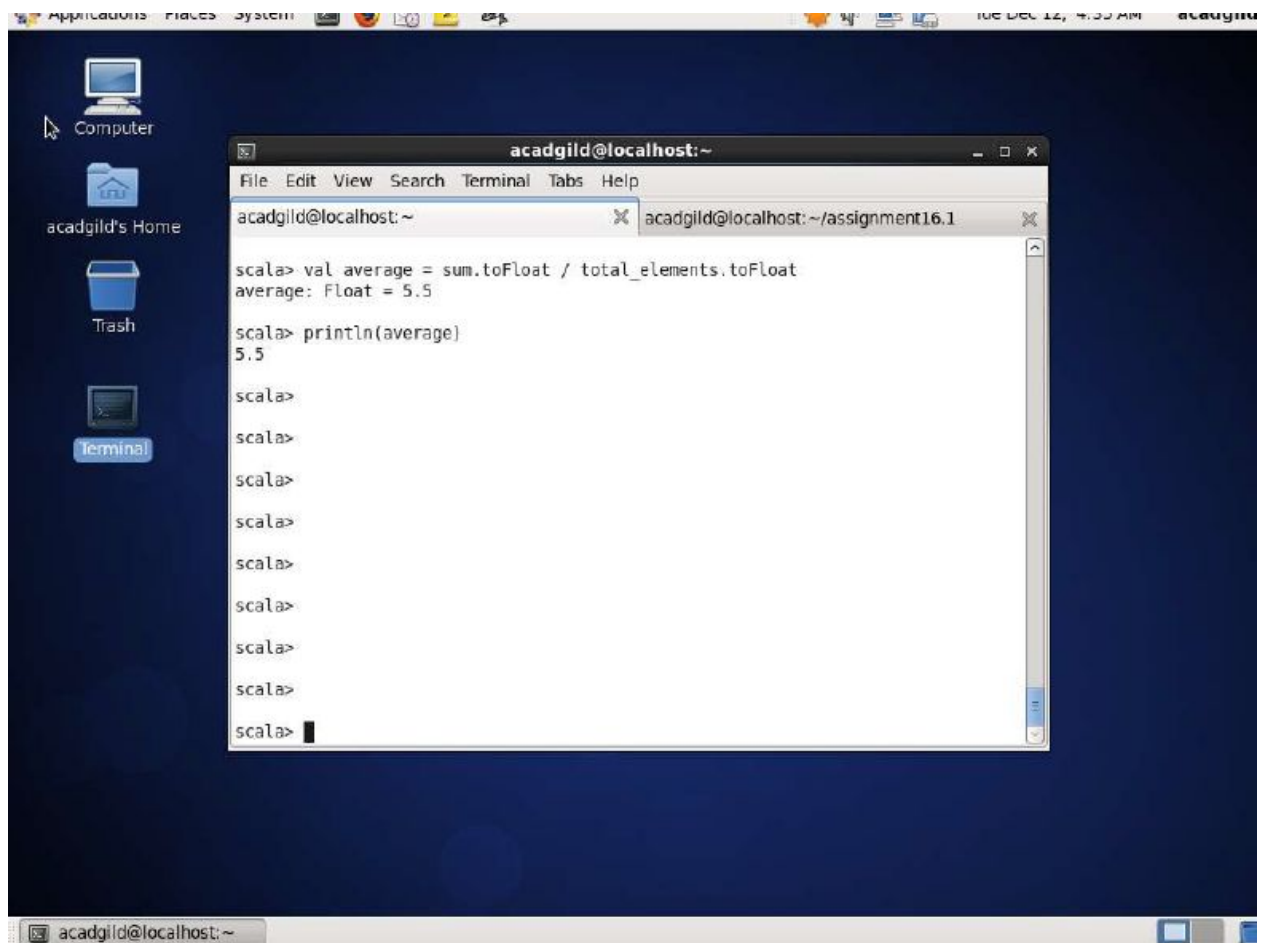
A screenshot of a Linux desktop environment. The desktop background is dark blue. On the left side, there is a vertical dock with icons for 'Computer', 'acadgild's Home' (a folder icon), 'Trash' (a trash can icon), and 'Terminal' (a terminal icon). The top of the screen shows a panel with 'Applications', 'Places', and 'System' menus, along with several system icons on the right. A terminal window is open in the center-right, titled 'acadgild@localhost:~'. The terminal has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', 'Tabs', and 'Help'. The terminal content shows the following Scala code being executed:

```
acadgild@localhost:~  
scala> val list = List[Int] (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
  
scala> var sum:Int = 0  
sum: Int = 0  
  
scala> list.foreach(elem => sum += elem)  
  
scala> println(sum)  
55  
  
scala>  
  
scala>  
  
scala>  
  
scala>  
  
scala>  
  
scala>  
  
scala>
```

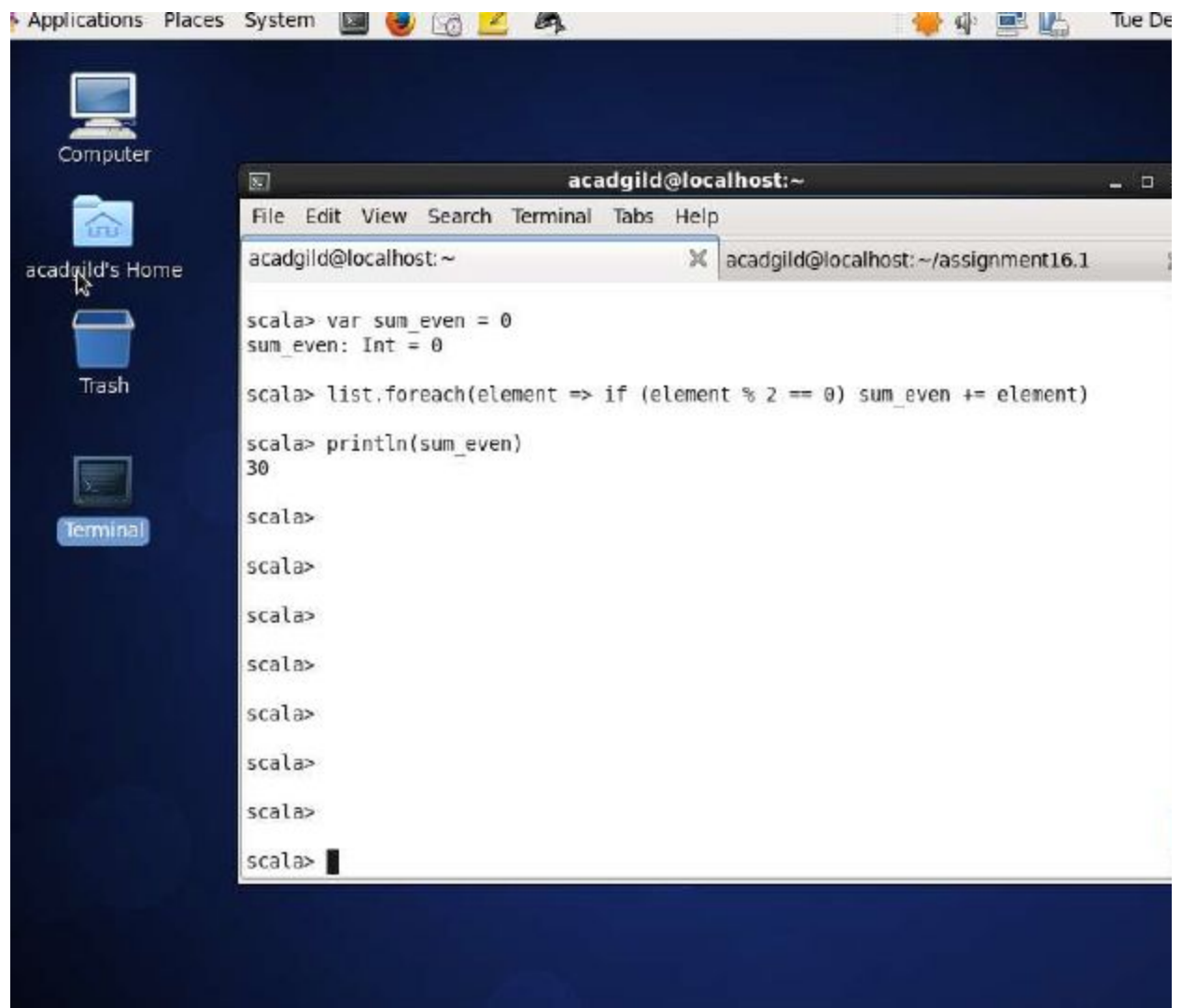
B: Find number of element in the list



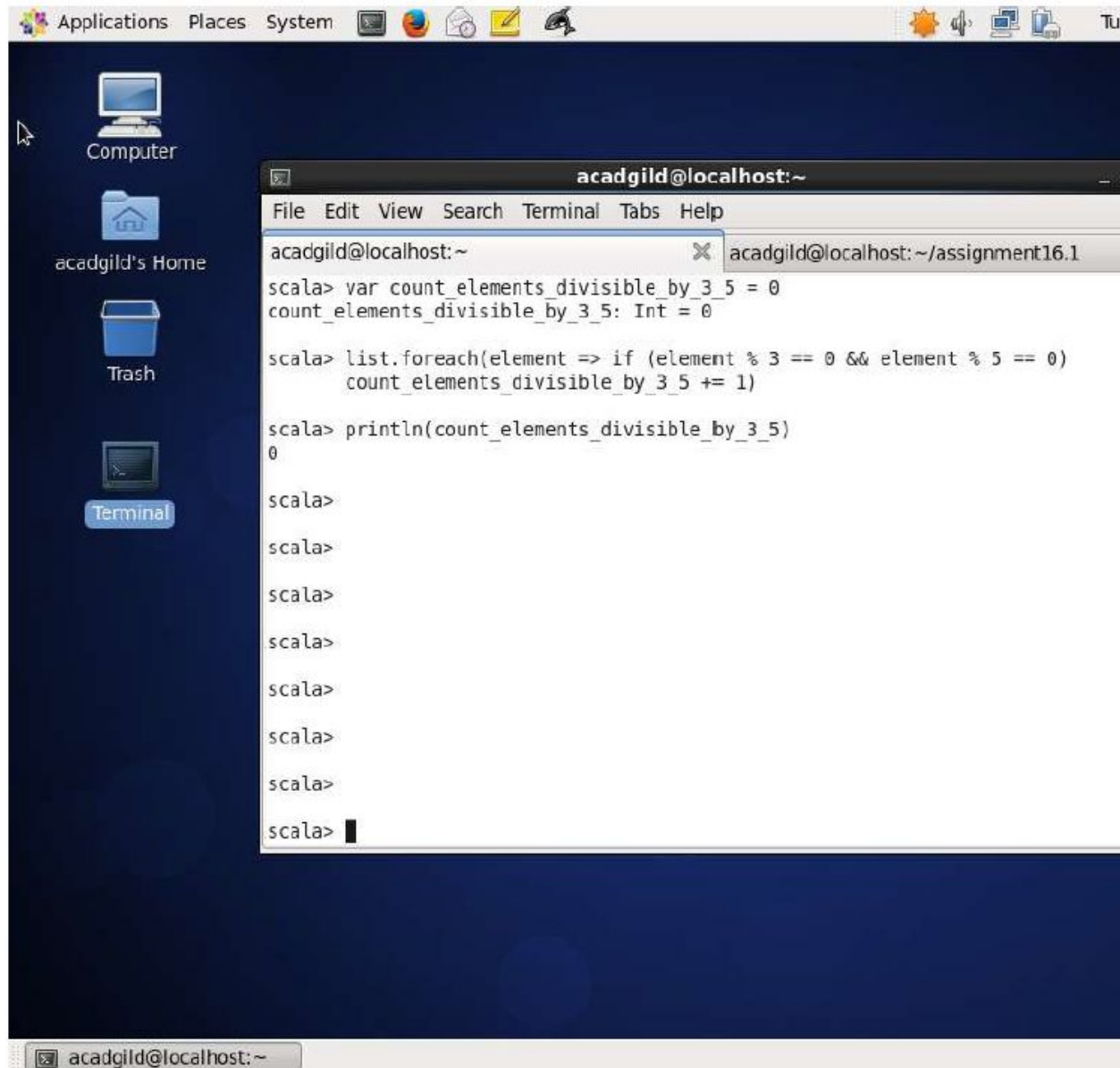
C: Find average of all the numbers



D: Find sum of event numbers in the list



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



Task 2

1) Pen down the limitations of MapReduce.

Limitation of Map Reduce are:

- It is based on disk based computation which makes computation jobs slower
- It is only meant for single pass computation, not iterative computations. It requires a sequence of Map Reduce jobs to run iterative task
- Needs integration with several tools to solve big data usecases.

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

RDD is a logical reference of a dataset which is partitioned across many server machines in the cluster. It is the primary abstraction in Spark and is the core of Apache Spark. Immutable and partitioned collection of records, which can only be created by coarse grained operations such as

map, filter, group-by, etc. Can only be created by reading data from a stable storage like HDFS or by transformations on existing RDD's.

Features of RDD:

- 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
- Distributed with data residing on multiple nodes in a cluster.
- Dataset is a collection of partitioned data with primitive values or values of values, e.g. tuples or other objects
- In-Memory, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.
- Immutable or Read-Only, i.e. it does not change once created and can only be transformed using transformations to new RDDs.
- Lazy evaluated, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.
- 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:

This 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 element 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. **SortByKey**: 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(","))
```