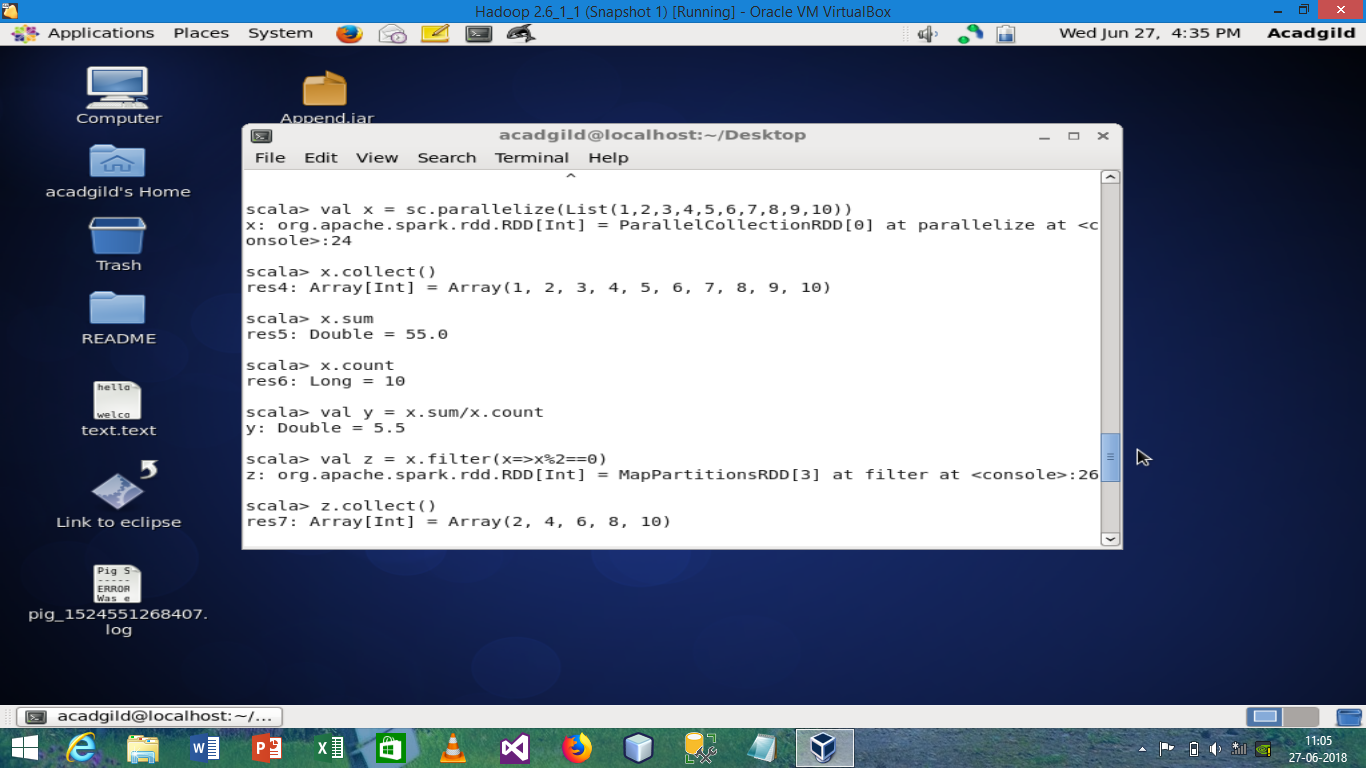
**TASK1 - find the sum of all the element present in in list List(1,2,3,4,5,6,7,8,9,10)**

**Solution**: for sum of list first of all we have to create the rdd for the list

**RDD:- val x = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))**

**Sum => x.sum**

Result as shown in figure



**TASK1 - find the count of all the element present in in list List(1,2,3,4,5,6,7,8,9,10)**

**Solution –** for count first we have to create the rdd as we did previously and them count function as shown in figure, first create a rdd and store it in variable x

**For count => x.count**

**for displaying => x.collect()** as shown in screenshot(above)

**TASK1 - calculate the average of the numbers in the list.**

**Solution –**  for calculating the average of number first we need to create the rdd and then apply this function, first create a rdd and store it in variable x

**For calculating average – val y = x.sum/x.count**

**for displaying => y.collect()** as shown in screenshot(above)

**TASK1 - find the sum of all the even numbers in the list.**

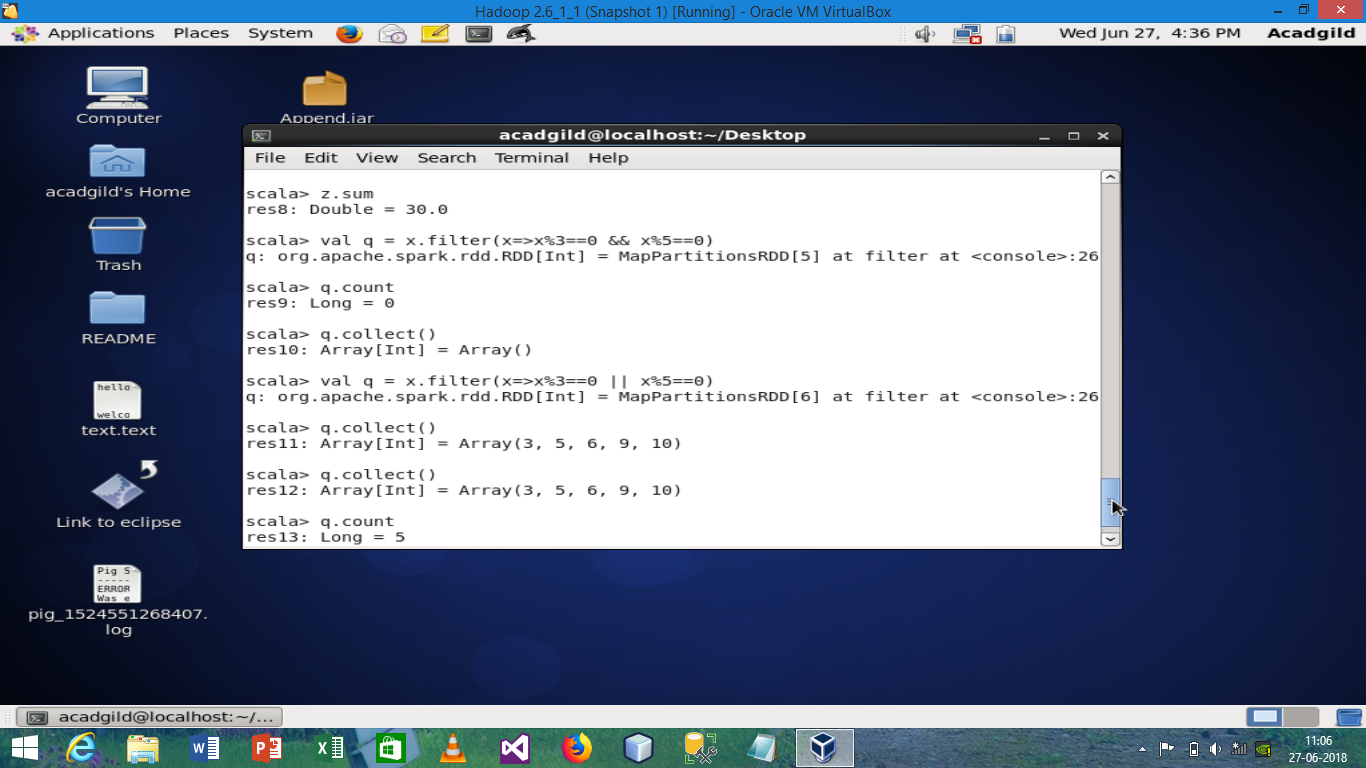
**Solution:**

First we create the rdd and store it in val x and on this variable we apply filter function

We use the filter function and store the result in z variable and then apply sum function on it to get the desired result

**SYNTAX or Command : val z = x.filter(x=>x%2==0)**

**z.sum**

****

**Task1:- find the total number of elements in the list divisible by both 5 and 3**

**Solution:-**

First we create the rdd and store it in val x and on this variable we apply filter function

We use the filter function and store the result in z variable and then apply sum function on it to get the desired result

**SYNTAX or Command : val q = x.filter(x=>x%3==0 || x%5==0)**

**q.count**

**Task2 :- Pen down the limitations of MapReduce.**

**Solution:**

**MapReduce cannot handle:**

1. Interactive Processing
2. Real-time (stream) Processing
3. Iterative (delta) Processing
4. In-memory Processing
5. Graph Processing

**Task2 :- What is RDD? Explain few features of RDD?**

**Solution:**

**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](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/)**) 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**

**In-memory Computation**

SparkRDDs have a provision of [**in-memory computation**](http://data-flair.training/blogs/apache-spark-in-memory-computing/). It stores intermediate results in distributed memory(RAM) instead of stable storage(disk).

**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**.](http://data-flair.training/blogs/lazy-evaluation-in-apache-spark-guide/)

**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**.](http://data-flair.training/blogs/apache-spark-streaming-fault-tolerance/)

**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.

**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.

**Persistence**

Users can state which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage or on Dis

**Task2:- List down few Spark RDD operations and explain each of them.**

**Solution:-**

**1-map(func)**

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.

**Map() example:**

1. import org.apache.spark.SparkContext
2. import org.apache.spark.SparkConf
3. import org.apache.spark.sql.SparkSession
4. object mapTest{
5. def main(args: Array[String]) = {
6. val spark = SparkSession.builder.appName("mapExample").master("local").getOrCreate()
7. val data = spark.read.textFile("spark\_test.txt").rdd
8. val mapFile = data.map(line => (line,line.length))
9. mapFile.foreach(println)
10. }
11. }

**2 - 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() example:**

1. val data = spark.read.textFile("spark\_test.txt").rdd
2. val flatmapFile = data.flatMap(lines => lines.split(" "))
3. flatmapFile.foreach(println)

**3 - filter(func)**

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.

**Filter() example:**

1. val data = spark.read.textFile("spark\_test.txt").rdd
2. val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")
3. println(mapFile.count())

* ***Note****–* In above code, flatMap function map line into words and then count the word “Spark” using count() Action after filtering lines containing “Spark” from mapFile.

**4 - mapPartitions(func)**

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.

**mapPartitionWithIndex()**

It is like mapPartition; Besides mapPartition it provides *func* with an integer value representing the index of the partition, and the map() is applied on partition index wise one after the other.

**5 - union(dataset)**

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.

**Union() example:**

1. val rdd1 = spark.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014)))
2. val rdd2 = spark.sparkContext.parallelize(Seq((5,"dec",2014),(17,"sep",2015)))
3. val rdd3 = spark.sparkContext.parallelize(Seq((6,"dec",2011),(16,"may",2015)))
4. val rddUnion = rdd1.union(rdd2).union(rdd3)
5. rddUnion.foreach(Println)

**6 - intersection(other-dataset)**

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.

**7 - distinct()**

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

**Distinct() example:**

1. val rdd1 = park.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014),(3,"nov",2014)))
2. val result = rdd1.distinct()
3. println(result.collect().mkString(", "))

* ***Note –*** In the above example, the distinct function will remove the duplicate record i.e. (3,'”nov”,2014).

**8 - 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. In this transformation, lots of unnecessary data get to transfer over the network.

**groupByKey() example:**

1. val data = spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
2. val group = data.groupByKey().collect()
3. group.foreach(println)

**reduceByKey(func, [numTasks])**

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.

**reduceByKey() example:**

1. val words = Array("one","two","two","four","five","six","six","eight","nine","ten")
2. val data = spark.sparkContext.parallelize(words).map(w => (w,1)).reduceByKey(\_+\_)
3. data.foreach(println)

***9 -* 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.

**sortByKey() example:**

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

**10 - join()**

The **Join** is database term. It combines the fields from two table using common values. 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 boon 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.

**Join() example:**

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

* ***Note*** *–*  The join() transformation will join two different RDDs on the basis of Key.

**11 - coalesce()**

To avoid full shuffling of data we use coalesce() function. In **coalesce()** we use existing partition so that less data is shuffled. Using this we can cut the number of the partition. Suppose, we have four nodes and we want only two nodes. Then the data of extra nodes will be kept onto nodes which we kept.

**Coalesce()**

**example:**

1. val rdd1 = spark.sparkContext.parallelize(Array("jan","feb","mar","april","may","jun"),3)
2. val result = rdd1.coalesce(2)
3. result.foreach(println).