Assignment 16.2:

Problem Statement:

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. Integration with Apache Storm is required for Stream data processing. Integration with Mahout required for Machine Learning
- Problems that cannot be trivially partitionable or recombinable becomes a candid limitation of MapReduce problem solving. For instance, Travelling Salesman problem.
- Due to the fixed cost incurred by each MapReduce job submitted, application that requires low latency time or random access to a large set of data is infeasible.
- Tasks that has a dependency on each other cannot be parallelized, which is not possible through MapReduce.

2. What is RDD? Explain 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.
- **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).
- **IParallel**, i.e. process data in parallel.
- **Typed** — RDD records have types, e.g. Long in RDD [Long] or (Int, String) in RDD[(Int, String)].
- **Partitioned** — records are partitioned (split into logical partitions) and distributed across nodes in a cluster.
- **Location-Stickiness** — RDD can define placement preferences to Compute partitions (as close to the records as possible).

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

There are two types of Apache Spark RDD operations, they are-

- Transformations
- Actions.

A Transformation is a function that produces new RDD from the existing RDDs but when we want to work with the actual dataset, at that point Action is performed. When the action is triggered after the result, new RDD is not formed like transformation.

Few transformation functions are:

- **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))
```

- **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(" "))
```

- **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")
```

- **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.
- **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)
```

- **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)
```

- **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
```

- **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)
```

- **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)
```

- **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(","))
```

Few Action functions:

- **Count():**Action count() returns the number of elements in RDD.
  Example**:**
  RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD "rdd.count()" will give the result 8.
- **Collect():**The action collect() is the common and simplest operation that returns our entire RDDs content to driver program. The application of collect() is unit testing where the entire RDD is expected to fit in memory. As a result, it makes easy to compare the result of RDD with the expected result.

  Action Collect() had a constraint that all the data should fit in the machine, and copies to the driver.
  Example:
  var rdd1 = sc.parallelize(Array(1,2,3,4,1,2))
  rdd1.distinct().collect()
- **take(n):** The action take(n) returns n number of elements from RDD. It tries to cut the number of partition it accesses, so it represents a biased collection. We cannot presume the order of the elements.
  Example**:** consider RDD {1, 2, 2, 3, 4, 5, 5, 6} in this RDD "take (4)" will give result { 2, 2, 3, 4}
- **top():**If ordering is present in our RDD, then we can extract top elements from our RDD using top(). Action top() use default ordering of data.
  Example**:** var x = sc.parallelize(Array(5,2,5,6,7,1))
  x.top(2)
- **countByValue():**The countByValue() returns, many times each element occur in RDD.
  Example**:**
  RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD "rdd.countByValue()" will give the result {(1,1), (2,2), (3,1), (4,1), (5,2), (6,1)}
- **reduce()**:The reduce() function takes the two elements as input from the RDD and then produces the output of the same type as that of the input elements. The simple forms of such function are an addition. We can add the elements of RDD, count the number of words. It accepts commutative and associative operations as an argument.
  Example: val rdd1 = spark.sparkContext.parallelize(List(20,32,45,62,8,5))
  val sum = rdd1.reduce(_+_)
  println(sum)
- **fold():**The signature of the fold() is like reduce(). Besides, it takes "zero value" as input, which is used for the initial call on each partition. But, the condition with zero value is that it should be the identity element of that operation. The key difference between fold() and reduce() is that, reduce() throws an exception for empty collection, but fold() is defined for empty collection.

For example, zero is an identity for addition; one is identity element for multiplication. The return type of fold() is same as that of the element of RDD we are operating on.
Example: rdd.fold(0)((x, y) => x + y)

- **aggregate()**: It gives us the flexibility to get data type different from the input type. The aggregate() takes two functions to get the final result. Through one function we combine the element from our RDD with the accumulator, and the second, to combine the accumulator. Hence, in aggregate, we supply the initial zero value of the type which we want to return.