**Assignment 16.2**

**Problem Statement**

**1) Pen down the limitations of MapReduce.**

MapReduce is a programming model designed for processing large volumes of

data in parallel by dividing the work into a set of independent tasks. MapReduce sends code to distributed data, instead of bringing data to the actual code. The individual concepts of functions called map and reduce have been derived from

functional programming languages (like C++ & Java) where they were applied to lists of input data. Another key underlying concept is that of "divide and conquer", where a single problem is broken into multiple individual subtasks. This approach becomes even more powerful when the subtasks are executed in parallel.

**Some Real-World Examples**

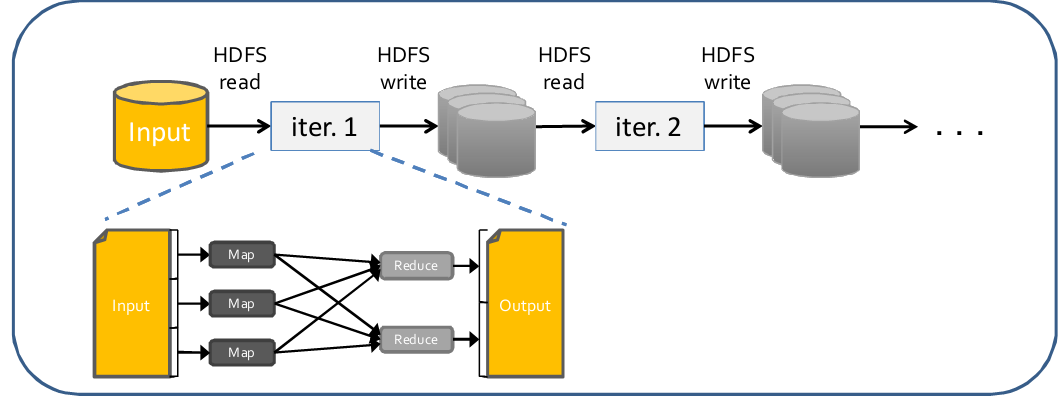
1. An address book relates a name (key) and to contact information (value).
2. A bank account uses an account number (key) to associate with the account

details (value).

1. The index of a book relates a word (key) to the pages on which it occurs (value)
2. On a computer file system, filenames (keys) allow access to any sort of data,such as text, images, and sound (values).

But MapReduce has some limitations such as

1. It’s based on disk based computing. Mapreduce writes the intermediate results to disk. Writing to and reading from disk is very slow



1. Suitable for single pass computations -not iterative computations.

In iterative algorithms and data minings processes mapreduce will be slow because of disk I/O, serialization and replication

1. Needs a sequence of MR jobs to run iterative tasks.
2. Needs integration with several other frameworks/tools to solve bigdata

usecases.

* Apache Storm for stream data processing
* Apache Mahout for machine learning

**2) What is RDD? Explain few features of RDD?**

RDD is immutable and partitioned collection of records, which can only be created

by coarse grained operations such as map, filter, group-by, etc. It can only be created by reading data from a stable storage like HDFS or by transformations on existing RDD’s. RDD (Resilient Distributed Dataset) is the primary abstraction in Spark and is the core of Apache Spark.One could compare RDDs to collections in Scala, i.e. a RDD is computed on many JVMs while a Scala collection lives on a single JVM.

**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. you 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.

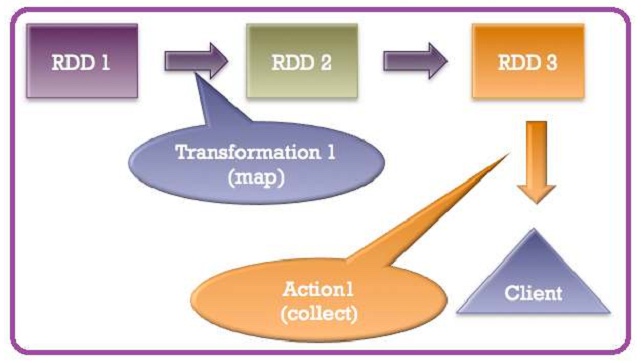
* **Location-Stickiness**—RDD can define placement preferences to

compute partitions (as close to the records as possible).

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

**Spark RDD operations** RDD supports two kinds of operations

1. **Actions** -operations that trigger computation and return values
2. **Transformations** -lazy operations that return another RDD



**Actions**

Actions are RDD operations that produce non-RDD values. In other words, a

RDD operation that returns a value of any type except RDD[T] is an action.

They trigger execution of RDD transformations to return values. Simply put,

an action evaluates the RDD lineage graph. You can think of actions as a valve and until action is fired, the data to be processed is not even in the pipes, i.e. transformations. Only actions can materialize the entire processing pipeline with real data. Actions are one of two ways to send data from executors to the driver (the other being accumulators).

**Some Actions supported by Spark**

**reduce(func)**

It aggregate the elements of the dataset using a function func(which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.

*// reduce numbers 1 to 10 by adding them up*

scala> val x = sc.parallelize(1 to 10, 2)

scala> val y = x.**reduce**((accum,n) => (accum + n))

y: Int = 55

**collect()**

it return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.

scala> val x = sc.parallelize(List("spark", "rdd", "example", "sample", "example"), 3)

scala> val y = x.map(x => (x, 1))

scala> y.**collect**

res0: Array[(String, Int)] = Array((spark,1), (rdd,1), (example,1), (sample,1), (example,1))

**count()**

it return the number of elements in the dataset.

scala> val names2 = sc.parallelize(List("apple", "beatty", "beatrice"))

names2: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[1476] at parallelize at <console>:12

scala> names2.**count**

res784: Long = 3

**first()**

it return the first element of the dataset (similar to take(1)).

scala> val names2 = sc.parallelize(List("apple", "beatty", "beatrice"))

names2: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[1477] at parallelize at <console>:12

scala> names2.**first**

res785: String = apple

**take(n)**

it return an array with the first n elements of the dataset.

scala> val names2 = sc.parallelize(List("apple", "beatty", "beatrice"))

names2: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[1478] at parallelize at <console>:12

scala> names2.**take**(2)

res786: Array[String] = Array(apple, beatty)

**saveAsTextFile(path)**

it write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported

file system. Spark will call toString on each element to convert it to a line of text in the file.

scala> val onlyInterestedIn = sc.textFile("baby\_names.csv").map(line => line.split(",")).map(n => (n(1), n(4)))

onlyInterestedIn: org.apache.spark.rdd.RDD[(String, String)] = MappedRDD[27] at map at <console>:12

scala> onlyInterestedIn.**saveAsTextFile**("results.csv")

**Transformations**

Transformations are lazy operations on a RDD that create one or many new RDDs,

e.g. map, filter, reduceByKey, join, cogroup, randomSplit. They are functions that take a RDD as the input and produce one or many RDDs as the output. They do not change the input RDD (since RDDs are immutable), but always produce one or more new RDDs by applying the computations they represent. Transformations are lazy, i.e. are not executed immediately. Only after calling an action are transformations executed. After executing a transformation, the result RDD(s) will always be different from their parents and can be smaller (e.g. filter, count, distinct, sample), bigger (e.g. flatMap, union, cartesian) or the same size (e.g. map).

**Transformations supported by Spark**

**map(func)**

Return a new distributed dataset formed by passing each element of the source through a function func.

scala> sc.parallelize(List(1,2,3)).**map**(x=>List(x,x,x)).collect

res201: Array[List[Int]] = Array(List(1, 1, 1), List(2, 2, 2), List(3, 3, 3))

**Filter(func)**

Return a new dataset formed by selecting those elements of the source on which func returns true.

val file = sc.textFile("catalina.out")

val errors = file.**filter**(line => line.contains("ERROR"))

**flatMap(func)**

it is similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).

scala> sc.parallelize(List(1,2,3)).**flatMap**(x=>List(x,x,x)).collect

res200: Array[Int] = Array(1, 1, 1, 2, 2, 2, 3, 3, 3)

**mapPartitions(func)**

it is similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.

// compare to the same, but with default parallelize

scala> val parallel = sc.parallelize(1 to 9)

parallel: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[452] at parallelize at <console>:12

scala> parallel.**mapPartitions**( x => List(x.next).iterator).collect

res384: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8)

**mapPartitionsWithIndex(func)**

it is similar to mapPartitions, but also provides func with an integer value representing the index of the partition, so funcmust be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.

scala> val parallel = sc.parallelize(1 to 9)

parallel: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[455] at parallelize at <console>:12

scala> parallel.**mapPartitionsWithIndex**( (index: Int, it: Iterator[Int]) => it.toList.map(x => index + ", "+x).iterator).collect

res389: Array[String] = Array(0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 7, 9)

**distinct([numTasks]))**

It return a new dataset that contains the distinct elements of the source dataset.

scala> val parallel = sc.parallelize(1 to 9)

parallel: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[477] at parallelize at <console>:12

scala> val par2 = sc.parallelize(5 to 15)

par2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[478] at parallelize at <console>:12

scala> parallel.union(par2).**distinct**.collect

res412: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)

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

When it is called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

scala> val filteredRows = babyNames.filter(line => !line.contains("Count")).map(line => line.split(","))

filteredRows: org.apache.spark.rdd.RDD[Array[String]] = MappedRDD[546] at map at <console>:14

scala> filteredRows.map(n => (n(1),n(4).toInt)).**reduceByKey**((v1,v2) => v1 + v2).collect

res452: Array[(String, Int)] = Array((BRADEN,39), (MATTEO,279), (HAZEL,133), (SKYE,63), (JOSUE,404), (RORY,12), (NAHLA,16), (ASIA,6), (MEGAN,581), (HINDY,254), (ELVIN,26), (AMARA,10), (CHARLOTTE,1737), (BELLA,672), (DANTE,246), (PAUL,712), (EPHRAIM,26), (ANGIE,295), (ANNABELLA,38), (DIAMOND,16), (ALFONSO,6), (MELISSA,560), (AYANNA,11), (ANIYAH,365), (DINAH,5), (MARLEY,32), (OLIVIA,6467), (MALLORY,15), (EZEQUIEL,13), (ELAINE,116), (ESMERALDA,71), (SKYLA,172), (EDEN,199), (MEGHAN,128), (AHRON,29), (KINLEY,5), (RUSSELL,5), (TROY,88), (MORDECHAI,521), (JALIYAH,10), (AUDREY,690), (VALERIE,584), (JAYSON,285), (SKYLER,26), (DASHIELL,24), (SHAINDEL,17), (AURORA,86), (ANGELY,5), (ANDERSON,369), (SHMUEL,315), (MARCO,370), (AUSTIN,1345), (MITCHELL,12), (SELINA,187), (FATIMA,421), (CESAR,292), (CAR...