Big Data And Hadoop

Session 13 - Assignment 1

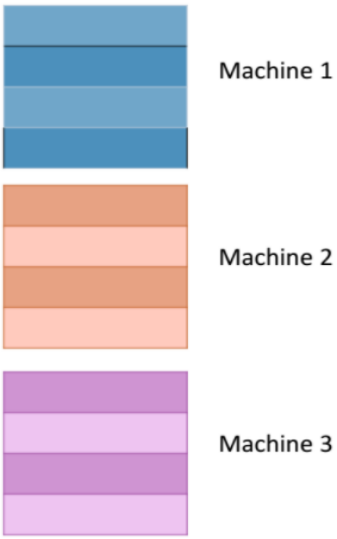
Problem Statement:

Answer the given questions.

1. **What is RDD?**

   Answer:

   a. Apache spark uses the concept of Resilient Distributed Dataset (RDD), which allows it to transparently store data in memory and persist it to disc when it's needed. This helps to reduce most of the disc read and write – the main time consuming factors – of data processing.
   b. RDDs load the data for us and are resilient, which means they can be recomputed.
   c. It is a fault-tolerant collection of elements partitioned across the nodes of the cluster that can be operated on in parallel.
   d. RDDs in Spark are immutable distributed collection of objects. They are not actual data, but they are Objects, which contain information about data residing on the cluster.
   e. Each RDD is split into multiple partitions, which may be computed on different nodes of the cluster.
   f. The RDDs can contain any type of Python, Java or Scala objects, including user-defined classes.
   g. RDD spreads across multiple machines



h. There are two ways to create RDDs

   i. Parallelized collections: Created by calling SparkContext's parallelize method on an existing collection in your driver program (a Scala Seq). The elements of the collection are copied to form a distributed dataset that can be operated on in parallel.

   Example: val data = Array(1,2,3,4,5)

   val distData = sc.parallelize(data)

   ii. External Datasets: Spark can create distributed datasets from any storage source supported by Hadoop, including your local file system, HDFS, Cassandra, HBase, Amazon S3, etc. Spark supports text files, SequenceFiles, and any other Hadoop InputFormat.

   Example: scala> val distFile = sc.textFile("data.txt")

i. Users may also ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations.

j. RDDs automatically recover from node failures. If a particular node crashes in the middle of an operation Z, which depended on operation Y, which in turn on operation X. The cluster manager (YARN/Mesos) finds out the node is dead and tries to assign another node to continue processing. This node will be told to operate on the particular partition of the RDD and the series of operations X->Y->Z (called lineage) that it has to execute, by passing in the Scala closures created from the application code. Now the new node can happily continue processing and there is effectively no data-loss.

2. Define Partitions.

Answer:

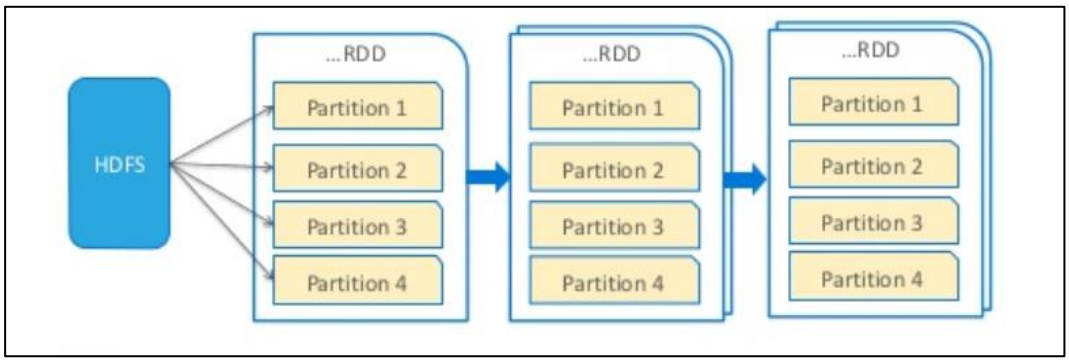
i. Resilient Distributed Datasets (RDD) is a simple and immutable distributed collection of objects. Each RDD is split into multiple partitions which may be computed on different nodes of the cluster.

ii. Partitioning is nothing but dividing RDDs into parts. By doing partitioning network I/O will be reduced so that data can be processed a lot faster.

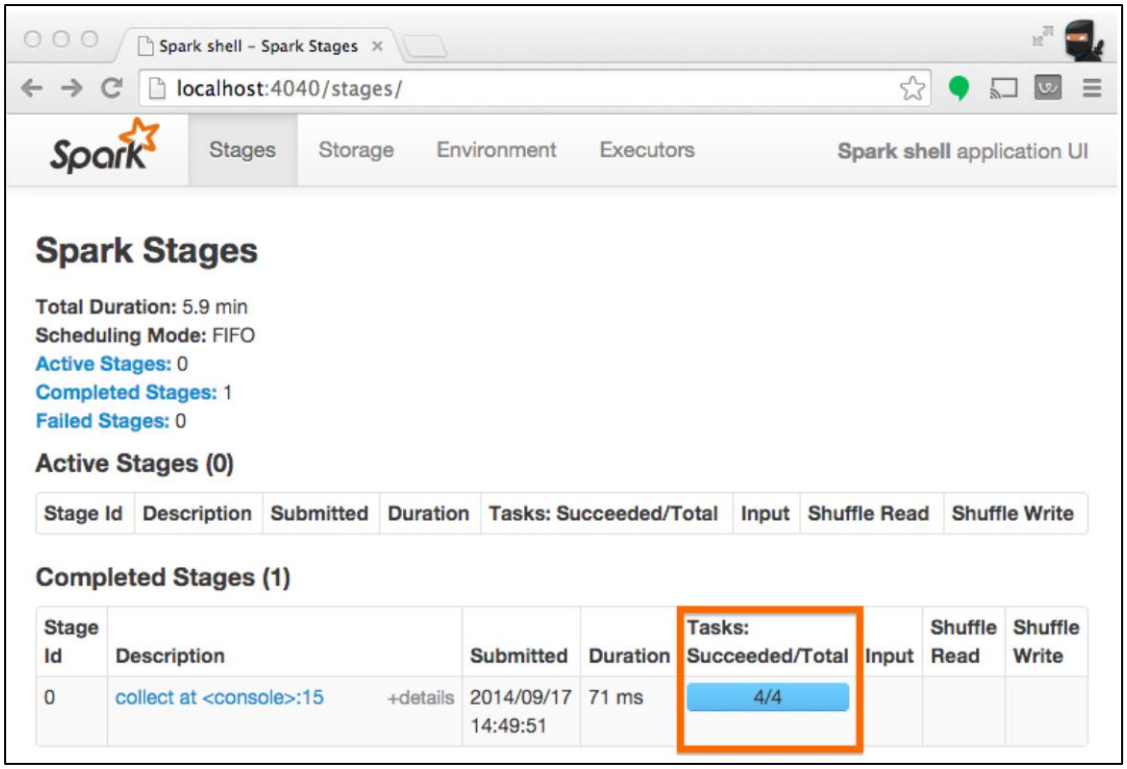
iii. By default, Spark tries to read data into an RDD from the nodes that are close to it. Since Spark usually accesses distributed partitioned data, to optimize transformation operations it creates partitions to hold the data chunks.

iv. RDDs get partitioned automatically without programmer intervention. However, there are times when you'd like to adjust the size and number of partitions or the partitioning scheme according to the needs of your application.

v. This is a pictorial representation of how data in the form of RDD is partitioned and transformed over the stages of transformation.

vi. 

When a stage executes, you can see the number of partitions for a given stage in the Spark UI.



3. What operations does RDD support?

Answer:

1. RDDs support two types of operations:
   i. Transformations:

      These operations create a new dataset from an existing one, and actions, which return a value to the driver program after running a computation on the dataset.

   ii. Actions: These operations aggregate all the elements of the RDD using some function and return the final result to the driver program (although there is also a parallel reduceByKey that returns a distributed dataset).
2. RDDs keeps track of Transformations and check them periodically. If a node fails, it can rebuild the lost RDD partition on the other nodes, in parallel.
3. Let us consider an example of RDD operations.

   val lines = sc.textFile("data.txt")

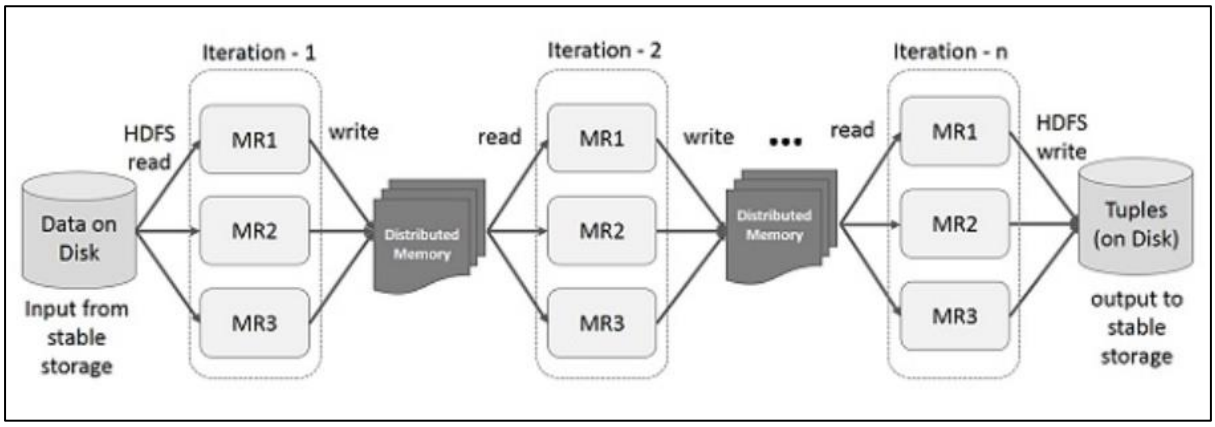
   val lineLengths = lines.map(s => s.length)

   val totalLength = lineLengths.reduce((a, b) => a + b)

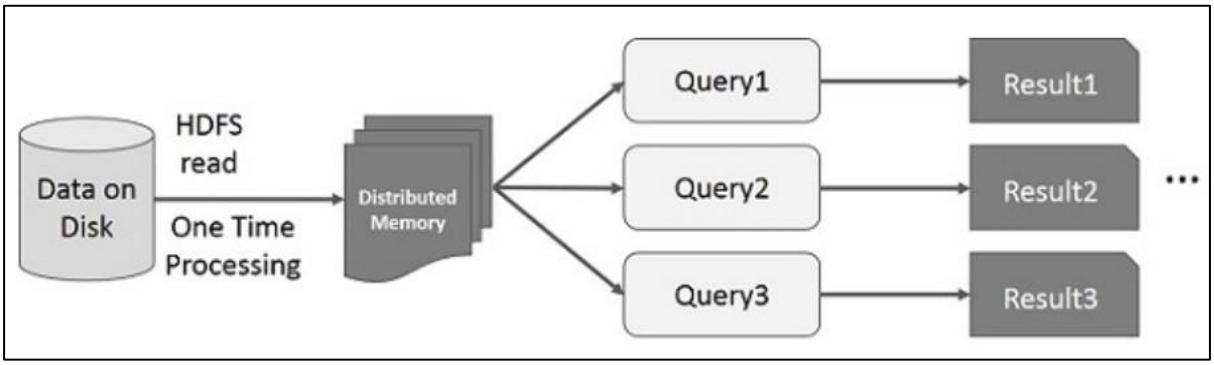
The first line defines a base RDD from an external file. This dataset is not loaded in memory or otherwise acted on: lines is merely a pointer to the file. The second line defines lineLengths as the result of a map transformation.

Again, lineLengths is not immediately computed, due to laziness. Finally, we run reduce, which is an action. At this point Spark breaks the computation into tasks to run on separate machines, and each machine runs both its part of the map and a local reduction, returning only its answer to the driver program.

4. Spark RDDs can perform both iterative and interactive operations.
5. Iterative operations will store intermediate results in distributed memory instead of stable storage and make the system faster. Pictorial representation is as follows:



6. Interactive Operations: If different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times.



7. By default, each transformed RDD may be recomputed each time you run an action on it

8. What do you understand by Transformations in Spark?

Answer:

a) Transformations are operations that create a new dataset from an existing one, and actions, which return a value to the driver program after running a computation on the dataset.
b) For example, map is a transformation that passes each dataset element through a function and returns a new RDD representing the results.
c) All transformations in Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file).
d) The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently.
e) For example, we can realize that a dataset created through map will be used in a reduce and return only the result of the reduce to the driver, rather than the larger mapped dataset.
f) An example of transformation operation is

   val input = sc.parallelize(List(1,2,3,4,5))

   val mapRDD = input.map(x=>x*x)

   Here, an RDD under goes a transformation to create another RDD, which contains square of every data element in previous RDD.
g) The following table lists some of the common transformations supported by Spark.

Transformation	**Meaning**
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func*returns true.
flatMap(*func*)	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).

mapPartitions(*func*)	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.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numTasks*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can

	pass an optional `numTasks` argument to set a different number of tasks.
reduceByKey(*func*, [*numTasks*])	When 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.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numTasks*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument.
join(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.
cogroup(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K,

	(Iterable<V>, Iterable<W>)) tuples. This operation is also called `groupWith`.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(*command*, *[envVars]*)	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithinPartitions(*partitioner*)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling `repartition` and then sorting within each partition because it can push the sorting down into the shuffle machinery.

9. Define Actions.

Answer:

a) Actions are a type of operation supported by RDD.

b) These operations aggregate all the elements of the RDD using some function return the final result to the driver program (although there is also a parallel reduceByKey that returns a distributed dataset).

c) Example of action is:

```
val lines = sc.textFile("data.txt")
val pairs = lines.map(s => (s, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
```

In this, we find the count per key. The aggregation result is computed and sent to the driver and hence this opaeration is an action.

d) The following table lists some of the common actions supported by Spark.

Action	**Meaning**
reduce(*func*)	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.
collect()	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.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset.
take(*n*)	Return an array with the first *n* elements of the dataset.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.

takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(*path*)	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.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using`SparkContext.objectFile()`.
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems.