**Phase I**

1. **RDD Persistence**

RDDs are recomputed

RDDs by default is recomputed each time an action is run on them. For example,

scala> val lines = sc.textFile("words.txt")

...

scala> lines.first()

res4: String = line1 word1

scala> lines.count()

res5: Long = 4

Here the call to action first () computes the RDD 'lines'. Again when we use another action 'count()' on the same RDD, the RDD is recomputed once again

Persisting RDDs

The default behavior of recomputing the RDDs on each action can be overridden by persisting the RDDs, so that no re-computation is done each time an action is called on the RDD. When persisted, each node that compute the RDD store the result in their Partitions

We use persist() method to persist an RDD. In Scala & Java, by default, persist() will store the data in JVM as unserialized object. In Python, calling persist() will serialize the data before persisting. Options to store in Memory/Disk combination is also possible.

scala> val lines = sc.textFile("words.txt")

...

scala> import org.apache.spark.storage.StorageLevel

...

scala> lines.persist(StorageLevel.MEMORY\_ONLY) //We can also use cache() method if we need MEMORY\_ONLY storage level

...

scala> lines.count() (1)

...

The actual persistence takes place during the first (1) action call on the RDD. Spark provides multiple Storage options(Memory/Disk) to persist the data as well as Replication Levels. More information can be found here

We use unpersist() to unpersist RDD. When the cached data exceeds the Memory capacity, Spark automatically evicts the old partitions(it will be recalculated when needed). This is called Last Recently used Cache(LRU) policy

1. **Combiner in Pair RDDs : combineByKey()**

Similar to combiner in MapReduce, when working with key/value pairs, combineByKey() interface can be used to customize the combiner functionality. Methods like reduceByKey() by default use their own combiner to combine the data locally in each Partition, for a given key

Similar to aggregate()(which is used with single element RDD), combineByKey() allows user to return different RDD element type compared to the element type of Input RDD

Syntax

def combineByKey[C](createCombiner: (V) ⇒ C, mergeValue: (C, V) ⇒ C, mergeCombiners: (C, C) ⇒ C, partitioner: Partitioner, mapSideCombine: Boolean = true, serializer: Serializer = null): RDD[(K, C)]

Generic function to combine the elements for each key using a custom set of aggregation functions. Turns an RDD[(K, V)] into a result of type RDD[(K, C)], for a "combined type" C Note that V and C can be different -- for example, one might group an RDD of type (Int, Int) into an RDD of type (Int, Seq[Int]). Users provide three functions:

- createCombiner, which turns a V into a C (e.g., creates a one-element list) - mergeValue, to merge a V into a C (e.g., adds it to the end of a list) - mergeCombiners, to combine two C's into a single one.

In addition, users can control the partitioning of the output RDD, and whether to perform map-side aggregation (if a mapper can produce multiple items with the same key).

Here,

1st Argument : createCombiner is called when a key(in the RDD element) is found for the first time in a given Partition. This method creates an initial value for the accumulator for that key

2nd Argument : mergeValue is called when the key already has an accumulator

3rd Argument : mergeCombiners is called when more that one partition has accumulator for the same key

Example

Let us calculate the average in each subject using combineByKey()

scala> val inputrdd = sc.parallelize(Seq(

| ("maths", 50), ("maths", 60),

| ("english", 65),

| ("physics", 66), ("physics", 61), ("physics", 87)),

| 1)

inputrdd: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[41] at parallelize at <console>:27

scala> inputrdd.getNumPartitions

res55: Int = 1

scala> val reduced = inputrdd.combineByKey(

| (mark) => {

| println(s"Create combiner -> ${mark}")

| (mark, 1)

| },

| (acc: (Int, Int), v) => {

| println(s"""Merge value : (${acc.\_1} + ${v}, ${acc.\_2} + 1)""")

| (acc.\_1 + v, acc.\_2 + 1)

| },

| (acc1: (Int, Int), acc2: (Int, Int)) => {

| println(s"""Merge Combiner : (${acc1.\_1} + ${acc2.\_1}, ${acc1.\_2} + ${acc2.\_2})""")

| (acc1.\_1 + acc2.\_1, acc1.\_2 + acc2.\_2)

| }

| )

reduced: org.apache.spark.rdd.RDD[(String, (Int, Int))] = ShuffledRDD[42] at combineByKey at <console>:29

scala> reduced.collect()

Create combiner -> 50

Merge value : (50 + 60, 1 + 1)

Create combiner -> 65

Create combiner -> 66

Merge value : (66 + 61, 1 + 1)

Merge value : (127 + 87, 2 + 1)

res56: Array[(String, (Int, Int))] = Array((maths,(110,2)), (physics,(214,3)), (english,(65,1)))

scala> val result = reduced.mapValues(x => x.\_1 / x.\_2.toFloat)

result: org.apache.spark.rdd.RDD[(String, Float)] = MapPartitionsRDD[43] at mapValues at <console>:31

scala> result.collect()

res57: Array[(String, Float)] = Array((maths,55.0), (physics,71.333336), (english,65.0))

The map side aggregation done using combineByKey() can also be disabled(which is the case with methods like groupByKey() where the functionality of the combiner is not needed)

1. **Components and Drivers**

Following are some important components of Spark

**Cluster Manager**

Is used to run the Spark Application in Cluster Mode

**Application**

User program built on Spark. Consists of,

**Driver Program**

The Program that has SparkContext. Acts as a coordinator for the Application

**Executors**

* Runs computation & Stores Application Data
* Are launched at the beginning of an Application & runs for the entire life time of an Application
* Each Application gets it own Executors
* An Application can have multiple Executors
* An Executor is not shared by Multiple Applications
* Provides in-memory storage for RDDs
* For an Application, No >1 Executors run in the same Node

**Task**

Represents a unit of work in Spark

Gets executed in Executor

**Job**

Parallel Computation consisting of multiple Tasks that gets spawned in response to Spark action

Check out the Diagram & Glossary [**here**](https://spark.apache.org/docs/1.3.1/cluster-overview.html) for more information on the Componens