Table of Contents

Introduction 1.1

Best Practices To Run Faster 1.2

- Don’t collect large RDDs 1.2.1
- Don't use count() when you don't need to return the exact number of rows 1.2.2
- Picking the Right Operators 1.2.3
  - Avoid List of Iterators 1.2.3.1
  - Avoid groupByKey when performing a group of multiple items by key 1.2.3.2
  - Avoid groupByKey when performing an associative reductiove operation
  - Avoid reduceByKey when the input and output value types are different 1.2.3.3 1.2.3.4
  - Avoid the flatMap-join-groupBy pattern 1.2.3.5
  - Use TreeReduce/TreeAggregate instead of Reduce/Aggregate 1.2.3.6
  - Hash-partition before transformation over pair RDD 1.2.3.7
  - Use coalesce to repartition in decrease number of partition 1.2.3.8
- TreeReduce and TreeAggregate Demystified 1.2.4
- When to use Broadcast variable 1.2.5
  - Joining a large and a small RDD 1.2.5.1
  - Joining a large and a medium size RDD 1.2.5.2
- Which storage level to choose 1.2.6
- Avoiding Shuffle "Less stage, run faster" 1.2.7
- Use the right level of parallelism 1.2.8
- Serialization 1.2.9
- Tuning Java Garbage Collection 1.2.10

References 1.3

Apache Spark - Best Practices and Tuning

This is a collections of notes (see References about Apache Spark's best practices). The notes aim to help me design and develop better programs with Apache Spark.

Introduction

- Best Practices To Run Faster
  - Don’t collect large RDDs
  - Don't use count() when you don't need to return the exact number of rows
  - Picking the Right Operators
    - Avoid List of Iterators
    - Avoid groupByKey when performing a group of multiple items by key
    - Avoid groupByKey when performing an associative reductiove operation
    - Avoid reduceByKey when the input and output value types are different
    - Avoid the flatMap-join-groupBy pattern
    - Use TreeReduce/TreeAggregate instead of Reduce/Aggregate
    - Hash-partition before transformation over pair RDD
    - Use coalesce to repartition in decrease number of partition
  - TreeReduce and TreeAggregate Demystified
  - When to use Broadcast variable
    - Joining a large and a small RDD
    - Joining a large and a medium size RDD
  - Which storage level to choose
  - Avoiding Shuffle "Less stage, run faster"
  - Use the right level of parallelism
  - Serialization
  - Tuning Java Garbage Collection
- References

Best Practices To Run Faster

Learn techniques for tuning your Apache Spark jobs for optimal efficiency.

Don’t collect large RDDs

When a collect operation is issued on a RDD, the dataset is copied to the driver, i.e. the master node. A memory exception will be thrown if the dataset is too large to fit in memory; `take` or `takeSample` can be used to retrieve only a capped number of elements instead.

Another way has been showed in [8] where you can get the array of partition indexes:

```
val parallel = sc.parallelize(1 to 9)
val parts = parallel.partitions
```

and then create a smaller rdd filtering out everything but a single partition. Collect the data from smaller rdd and iterate over values of a single partition:

```
for(p <- parts){
  val idx = p.index
  val partRDD = parallel.mapPartitionsWithIndex((index: Int, it: Iterator[Int]) => if(
index == idx) it else Iterator(), true)
  val data = partRDD.collect
  // Data contains all values from a single partition in the form of array.
  // Now you can do with the data whatever you want: iterate, save to a file, etc.
}

// You can use also the foreachPartition operation
parallel.foreachPartition(partition => {
  partition.toArray
  // Your code
})
```

Of cause, it will work only if the partitions are small enough. If they aren't, you can always increase the number of partitions with `rdd.coalesce(numParts, true)`.

Don't use count() when you don't need to return the exact number of rows

When you don't need to return the exact number of rows use:

```
DataFrame inputJson = sqlContext.read().json(...);
if (inputJson.take(1).length == 0) {}
```

instead of

```
if (inputJson.count() == 0) {}
```

In RDD you can use **isEmpty()** because if you see the code: https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/rdd/RDD.scala

```
def isEmpty(): Boolean = withScope { partitions.length == 0 || take(1).length == 0 }
```

Picking the Right Operators

When you try to write an application with Spark, you can usually choose from many arrangements of actions and transformations that will produce the same results. However, not all these arrangements will result in the same performance: avoiding common pitfalls and picking the right arrangement can make a world of difference in an application’s performance.

The following rules and insights that I've collected will help you orient yourself when these choices come up.

Avoid List of Iterators

Often when reading in a file [22], we want to work with the individual values contained in each line separated by some delimiter. Splitting a delimited line is a trivial operation:

```
newRDD = textRDD.map(line => line.split(","))
```

But the issue here is the returned RDD will be an iterator composed of iterators. What we want is the individual values obtained after calling the split function. In other words, we need an `Array[String]` not an `Array[Array[String]]`. For this we would use the `flatMap` function. For those with a functional programming background, using a `flatMap` operation is nothing new. But if you are new to functional programming it's a great operation to become familiar with.

```
val inputData = sc.parallelize (Array ("foo,bar,baz", "larry,moe,curly", "one,two,thre
e") ).cache ()

val mapped = inputData.map (line => line.split (",") )
val flatMapped = inputData.flatMap (line => line.split (",") )

val mappedResults = mapped.collect ()
val flatMappedResults = flatMapped.collect ();

println ("Mapped results of split")
println (mappedResults.mkString (" : ") )

println ("FlatMapped results of split")
println (flatMappedResults.mkString (" : ") )
```

When we run the program we see these results:

```
Mapped results of split
[Ljava.lang.String;@45e22def : [Ljava.lang.String;@6ae3fb94 : [Ljava.lang.String;@4417
af13
FlatMapped results of split
foo : bar : baz : larry : moe : curly : one : two : three
```

As we can see the map example returned an Array containing 3 `Array[String]` instances, while the `flatMap` call returned individual values contained in one Array.

Avoid groupByKey when performing a group of multiple items by key

As already showed in [21] let's suppose we've got a RDD items like:

```
(3922774869,10,1)
(3922774869,11,1)
(3922774869,12,2)
(3922774869,13,2)
(1779744180,10,1)
(1779744180,11,1)
(3922774869,14,3)
(3922774869,15,2)
(1779744180,16,1)
(3922774869,12,1)
(3922774869,13,1)
(1779744180,14,1)
(1779744180,15,1)
(1779744180,16,1)
(3922774869,14,2)
(3922774869,15,1)
(1779744180,16,1)
(1779744180,17,1)
(3922774869,16,4)
...
```

which represent **(id, age, count)** and we want to group those lines to generate a dataset for which each line represent the distribution of age of each id like this ((id, age) is unique):

```
(1779744180, (10,1), (11,1), (12,2), (13,2) ...)
(3922774869, (10,1), (11,1), (12,3), (13,4) ...)
```

which is **(id, (age, count), (age, count) ...)**

The easiest way is first reduce by both fields and then use groupBy:

```
rdd
.map { case (id, age, count) => ((id, age), count) }.reduceByKey(_ + _)
.map { case ((id, age), count) => (id, (age, count)) }.groupByKey()
```

Which returns an `RDD[(Long, Iterable[(Int, Int)])]`, for the input above it would contain these two records:

```
(1779744180,CompactBuffer((16,3), (15,1), (14,1), (11,1), (10,1), (17,1)))
(3922774869,CompactBuffer((11,1), (12,3), (16,4), (13,3), (15,3), (10,1), (14,5)))
```

But if you have a **very large dataset**, in order to reduce shuffling, you should not to use `groupByKey`.

Instead you can use `aggregateByKey`:

```
import scala.collection.mutable

val rddById = rdd.map { case (id, age, count) => ((id, age), count) }.reduceByKey(_ +
_)
val initialSet = mutable.HashSet.empty[(Int, Int)]
val addToSet = (s: mutable.HashSet[(Int, Int)], v: (Int, Int)) => s += v
val mergePartitionSets = (p1: mutable.HashSet[(Int, Int)], p2: mutable.HashSet[(Int, I
nt)]) => p1 ++= p2
val uniqueByKey = rddById.aggregateByKey(initialSet)(addToSet, mergePartitionSets)
```

This will result in:

```
uniqueByKey: org.apache.spark.rdd.RDD[(AnyVal, scala.collection.mutable.HashSet[(Int,
Int)])]
```

And you will be able to print the values as:

```
scala> uniqueByKey.foreach(println)
(1779744180,Set((15,1), (16,3)))
(1779744180,Set((14,1), (11,1), (10,1), (17,1)))
(3922774869,Set((12,3), (11,1), (10,1), (14,5), (16,4), (15,3), (13,3)))
```

Shuffling can be a great bottleneck. Having many big HashSet's (according to your dataset) could also be a problem. However, it's more likely that you'll have a large amount of ram than network latency which results in faster reads/writes across distributed machines.

Here are more functions to prefer over `groupByKey`:

- `combineByKey`
  can be used when you are combining elements but your return type differs from your input value type. You can see an example here
- `foldByKey`
  merges the values for each key using an associative function and a neutral "zero value".

Avoid groupByKey when performing an associative reductive operation

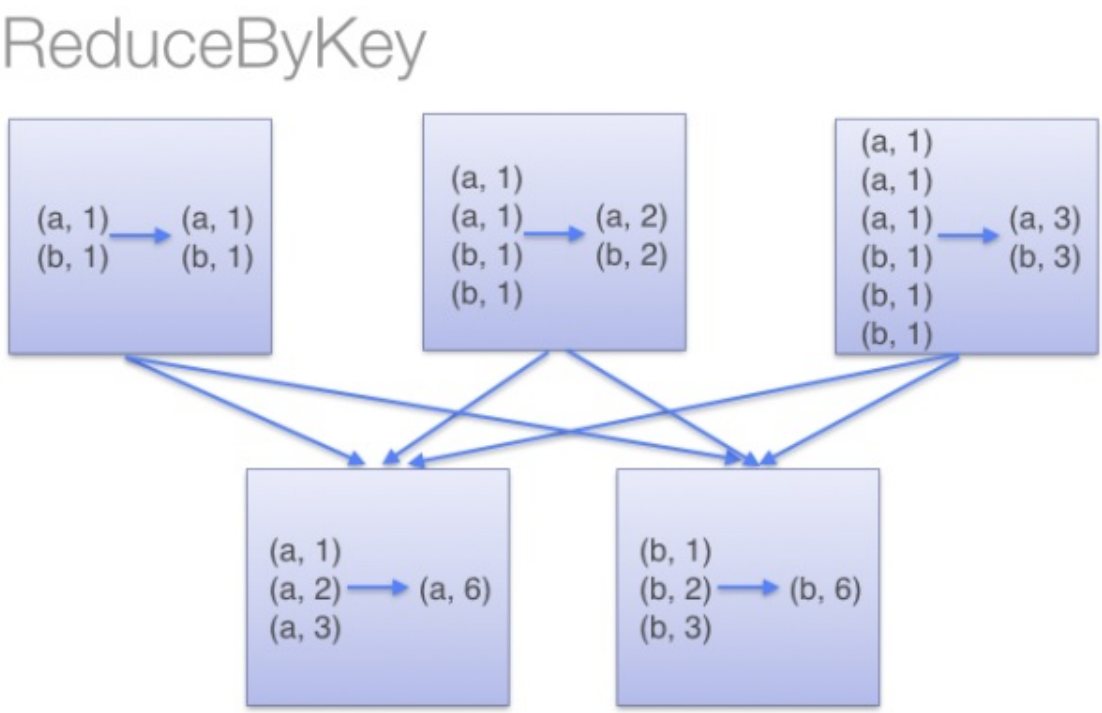
For example, `rdd.groupByKey().mapValues(_.sum)` will produce the same results as `rdd.reduceByKey(_ + _)`. However, the former will transfer the entire dataset across the network, while the latter will compute local sums for each key in each partition and combine those local sums into larger sums after shuffling.

As already showed in [2] let see example of word count you can process RDD and find the frequency of word using both the transformations `groupByKey` and `reduceBykey`.

word count using `reduceBykey`:

```
val wordPairsRDD = rdd.map(word => (word, 1))
val wordCountsWithReduce = wordPairsRDD
  .reduceByKey(_ + _)
  .collect()
```

See in diagram how RDD are process and shuffle over the network

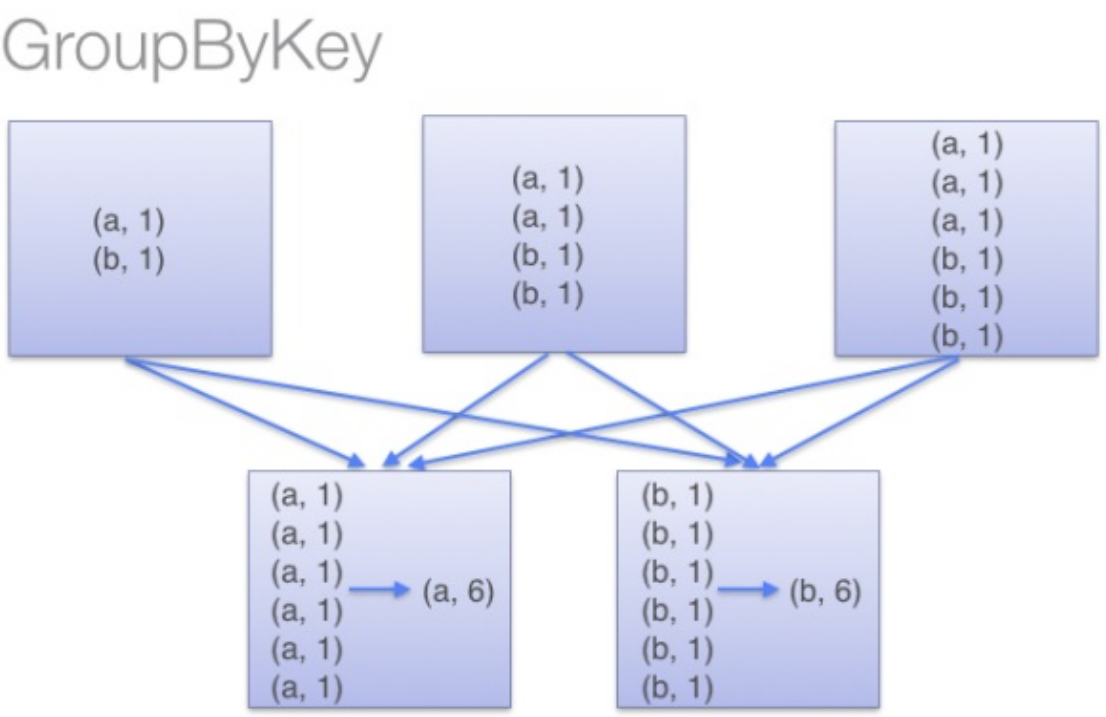


As you see in above diagram all worker node first process its own partition and count words on its own machine and then shuffle for final result.

On the other hand if we use `groupByKey` for word count as follow:

```
val wordCountsWithGroup = rdd
  .groupByKey()
  .map(t => (t._1, t._2.sum))
  .collect()
```

Let see diagram how RDD are process and shuffle over the network using **groupByKey**



As you see above all worker node shuffle data and at final node it will be count words so using `groupByKey` so lot of unnecessary data will be transfer over the network.

So avoid using `groupByKey` as much as possible.

Avoid reduceByKey when the input and output value types are different

For example, consider writing a transformation that finds all the unique strings corresponding to each key. One way would be to use map to transform each element into a Set and then combine the Sets with `reduceByKey`:

```
rdd.map(kv => (kv._1, new Set[String]() + kv._2)) .reduceByKey(_ ++ _)
```

This code results in tons of unnecessary object creation because a new Set must be allocated for each record. It's better to use `aggregateByKey`, which performs the map-side aggregation more efficiently:

```
val zero = new collection.mutable.Set[String]()
rdd.aggregateByKey(zero)( (set, v) => set += v, (set1, set2) => set1 ++= set2)
```

Avoid the flatMap-join-groupBy pattern

When two datasets are already grouped by key and you want to join them and keep them grouped, you can just use `cogroup`. That avoids all the overhead associated with unpacking and repacking the groups.

Use TreeReduce/TreeAggregate instead of Reduce/Aggregate

You'll see in TreeReduce and TeeAggregate Demystifed `treeReduce/treeAggregate` function are more efficient than `reduce/aggregate`.

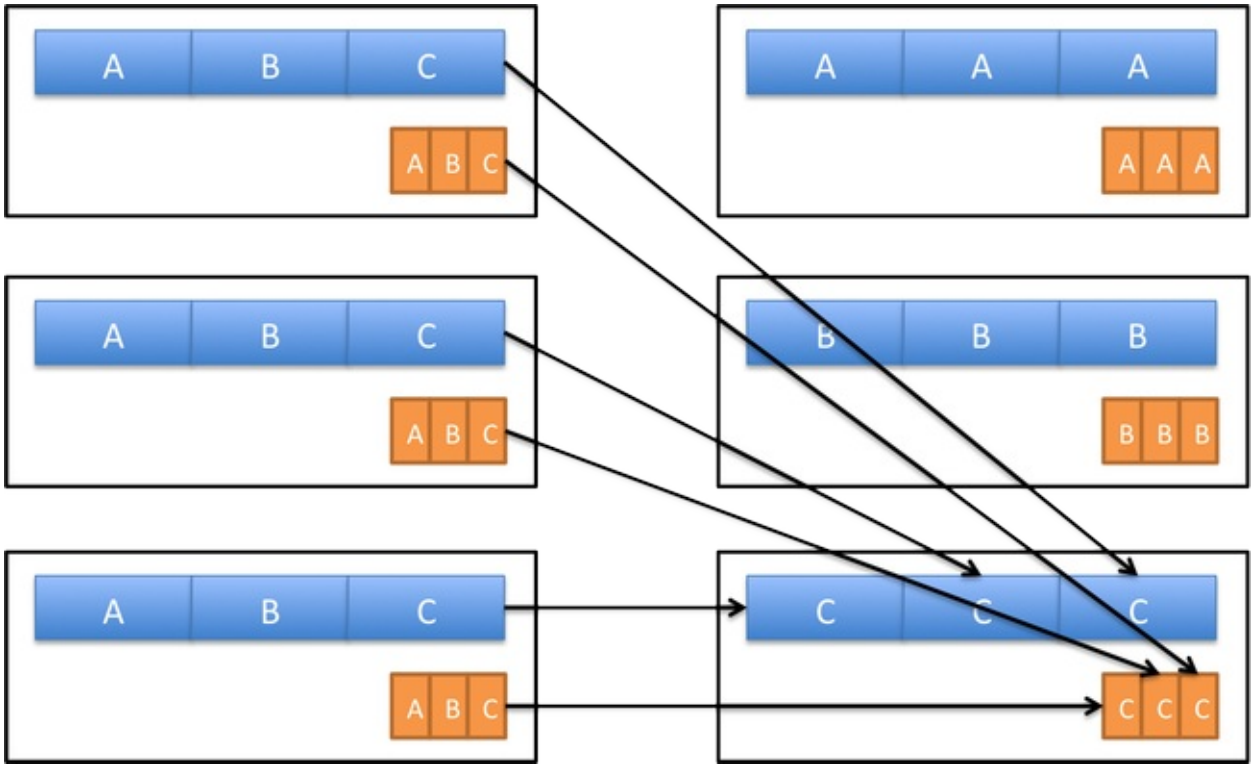
Hash-partition before transformation over pair RDD

Before perform any transformation we should shuffle same key data at the same worker so for that we use Hash-partition to shuffle data and make partition using the key of the pair RDD let see the example of the Hash-Partition data

```
val wordPairsRDD = rdd.map(word => (word, 1)).
                   partitonBy(new HashPartition(4))

val wordCountsWithReduce = wordPairsRDD
  .reduceByKey(_ + _)
  .collect()
```

When we are using Hash-partition the data will be shuffle and all same key data will shuffle at same worker, Let see in diagram



In the above diagram you can see all the data of “c” key will be shuffle at sameworker node. So if we use tansformation over pair RDD we should use hash-partitioning.

Use coalesce to repartition in decrease number of partition

Use **coalesce** if you decrease number of partition of the RDD instead of **repartition**. **coalesce** is usefull because its not shuffle data over network.

TreeReduce and TreeAggregate Demystified

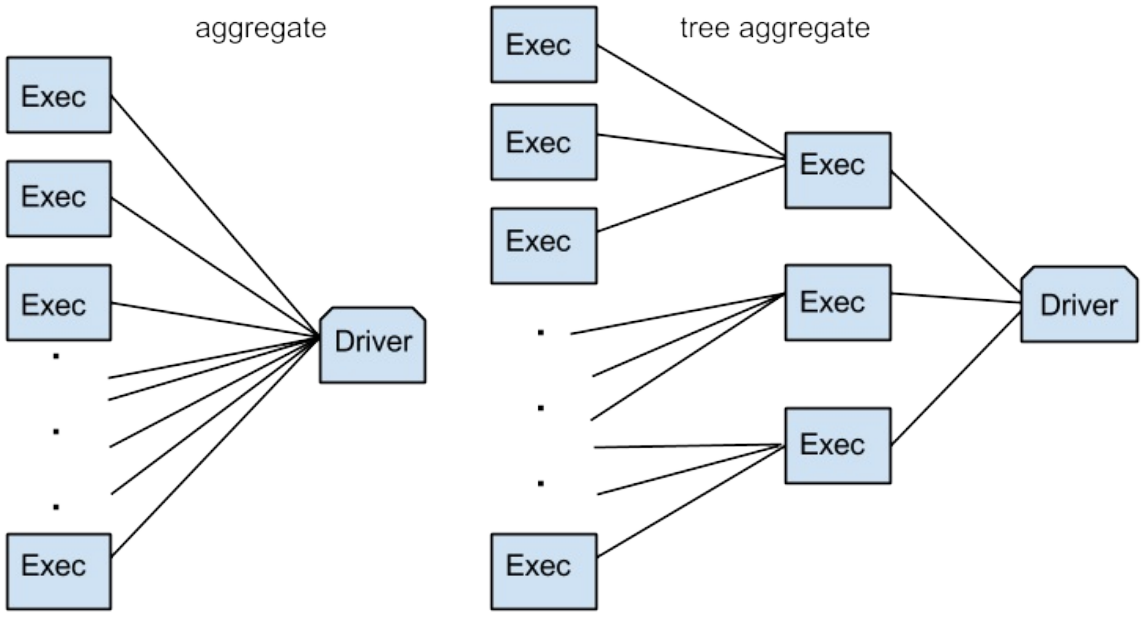
Introduction

In a regular **reduce** or **aggregate** functions in Spark (and the original MapReduce) all partitions have to send their reduced value to the driver machine, and that machine spends linear time on the number of partitions (due to the CPU cost in merging partial results and the network bandwidth limit). It becomes a **bottleneck** [13] when there are many partitions and the data from each partition is big.

Since Spark 1.1 [20] was introduced a new aggregation communication pattern based on multi-level aggregation trees. In this setup, data are combined partially on a small set of executors before they are sent to the driver, which dramatically reduces the load the driver has to deal with. Tests showed that these functions reduce the aggregation time by an order of magnitude, especially on datasets with a large number of partitions.

So, in **treeReduce** and in **treeAggregate**, the partitions talk to each other in a logarithmic number of rounds.

In case of **treeAggregate** imagine the follow n-ary tree that has all the partitions at its leaves and the root will contain the final reduced value. This way there is no single bottleneck machine.



Differences between reduceByKey and treeReduce

reduceByKey is only available on key-value pair RDDs, while **treeReduce** is a generalization of reduce operation on any RDD. **reduceByKey** is used for implementing **treeReduce** but they are not related in any other sense. **reduceByKey** performs reduction for each key, resulting in an RDD; it is not an action but a transformation that returns a Shuffled RDD.
On the other hand, **treeReduce** perform the reduction in parallel using **reduceByKey** (this is done by creating a key-value pair RDD on the fly, with the keys determined by the depth of the tree).

Differences between aggregate and treeAggregate

treeAggregate [19] is a specialized implementation of **aggregate** that iteratively applies the combine function to a subset of partitions. This is done in order to prevent returning all partial results to the driver where a single pass reduce would take place as the classic **aggregate** does.

Why you should use TreeReduce/TreeAggregate

Many of **MLib**'s algorithms uses **treeAggregate**, in the case of **GaussianMixture** (https://tinyurl.com/n3l68a8) the use of **treeAggregate** rather than **aggregate** have increased the performance about **20%**, while **Online Variational Bayes for LDA** (https://tinyurl.com/kt6kty6) uses a **treeAggregate** instead of a **reduce** to aggregate the expected word-topic count matrix (potentially a very large matrix) without scalability issues. Also **MLlib**'s implementation of **Gradient Descent** use **treeAggregate** (https://tinyurl.com/l6q5nn7).

In fact curious about this I've decided to use **treeAggregate** instead of a **reduce** to compute **Gradient** in my implementation of **Back Propagation**. In my test of dataset with 100 features and 10M instance partitioned in 96 partitions, performed on a cluster consists of 3 Worker nodes and one Application Master node (each with 16 CPUs and 52 GB memory), the **neural network** performed 100 epochs in only 36 minutes instead of hours.

Code examples (Scala)

The follow **Scala** code generates two random double **RDD** that contains **1 million values** and calculates the **Euclidean distance** using **map-reduce pattern**, **treeReduce** and **treeAggregate**:

```
import org.apache.commons.lang.SystemUtils
import org.apache.spark.mllib.random.RandomRDDs._
```

```
import org.apache.spark.sql.SQLContext
import org.apache.spark.{SparkConf, SparkContext}

import scala.math.sqrt

object Test{

  def main(args: Array[String]) {

    var mapReduceTimeArr : Array[Double]= Array.ofDim(20)
    var treeReduceTimeArr : Array[Double]= Array.ofDim(20)
    var treeAggregateTimeArr : Array[Double]= Array.ofDim(20)

    // Spark setup
    val config = new SparkConf().setAppName("TestStack")
    val sc: SparkContext = new SparkContext(config)
    val sql: SQLContext = new SQLContext(sc)

    // Generate a random double RDD that contains 1 million i.i.d. values drawn from t
he
    // standard normal distribution `N(0, 1)`, evenly distributed in 5 partitions.
    val input1 = normalRDD(sc, 1000000L, 5)

    // Generate a random double RDD that contains 1 million i.i.d. values drawn from t
he
    // standard normal distribution `N(0, 1)`, evenly distributed in 5 partitions.
    val input2 = normalRDD(sc, 1000000L, 5)

    val xy = input1.zip(input2).cache()
    // To materialize th RDD
    xy.count()

    for(i:Int <-0 until 20){
      val t1 = System.nanoTime()
      val euclideanDistanceMapRed = sqrt(xy.map { case (v1, v2) => (v1 - v2) * (v1 - v
2) }.reduce(_ + _))
      val t11 = System.nanoTime()
      println("Map-Reduce - Euclidean Distance "+euclideanDistanceMapRed)
      mapReduceTimeArr(i)=(t11 - t1)/1000000.0
      println("Map-Reduce - Elapsed time: " + (t11 - t1)/1000000.0  + "ms")
    }

    for(i:Int <-0 until 20) {
      val t2 = System.nanoTime()
      val euclideanDistanceTreeRed = sqrt(xy.map { case (v1, v2) => (v1 - v2) * (v1 -
v2) }.treeReduce(_ + _))
      val t22 = System.nanoTime()
      println("TreeReduce - Euclidean Distance "+euclideanDistanceTreeRed)
      treeReduceTimeArr(i)=(t22 - t2) / 1000000.0
      println("TreeReduce - Elapsed time: " + (t22 - t2) / 1000000.0 + "ms")
    }

    for(i:Int <-0 until 20) {
```

```
      val t3 = System.nanoTime()
      val euclideanDistanceTreeAggr = sqrt(xy.treeAggregate(0.0)(
        seqOp = (c, v) => {
          (c + ((v._1 - v._2) * (v._1 - v._2)))
        },
        combOp = (c1, c2) => {
          (c1 + c2)
        }))
      val t33 = System.nanoTime()
      println("TreeAggregate - Euclidean Distance " + euclideanDistanceTreeAggr)
      treeAggregateTimeArr(i) = (t33 - t3) / 1000000.0
      println("TreeAggregate - Elapsed time: " + (t33 - t3) / 1000000.0 + "ms")
    }

    val mapReduceAvgTime = mapReduceTimeArr.sum / mapReduceTimeArr.length
    val treeReduceAvgTime = treeReduceTimeArr.sum / treeReduceTimeArr.length
    val treeAggregateAvgTime = treeAggregateTimeArr.sum / treeAggregateTimeArr.length

    val mapReduceMinTime = mapReduceTimeArr.min
    val treeReduceMinTime = treeReduceTimeArr.min
    val treeAggregateMinTime = treeAggregateTimeArr.min

    val mapReduceMaxTime = mapReduceTimeArr.max
    val treeReduceMaxTime = treeReduceTimeArr.max
    val treeAggregateMaxTime = treeAggregateTimeArr.max

    println("Map-Reduce - Avg:" + mapReduceAvgTime+ "ms "+ "Max:" +mapReduceMaxTime+ "
ms "+ "Min:" +mapReduceMinTime+ "ms ")
    println("TreeReduce - Avg:" + treeReduceAvgTime + "ms "+ "Max:" +treeReduceMaxTime
+ "ms "+ "Min:" +treeReduceMinTime+ "ms ")
    println("TreeAggregate - Avg:" + treeAggregateAvgTime + "ms "+ "Max:" +treeAggrega
teMaxTime+ "ms "+ "Min:" +treeAggregateMinTime+ "ms ")
  }
}
```

To see **treeReduce/treeAggregate** shine, this code should be run on a cluster with a large number of partitions.

Code examples (Java)

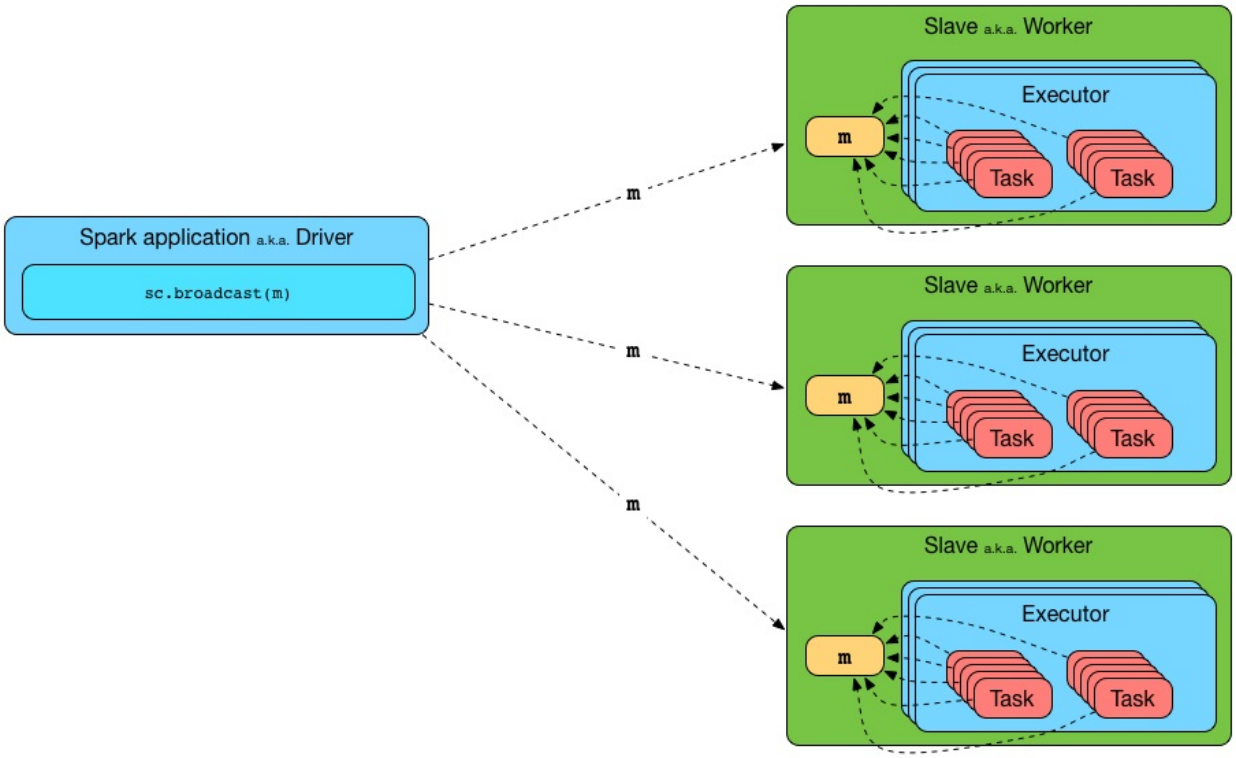
Basic Java treeReduce and treeAggregate examples

```
public void treeReduce() {
   JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(-5, -4, -3, -2, -1, 1, 2, 3, 4)
, 10);
     Function2<Integer, Integer, Integer> add = new Function2<Integer, Integer, Intege
r>() {
       @Override
       public Integer call(Integer a, Integer b) {
         return a + b;
       }
     };
     for (int depth = 1; depth <= 10; depth++) {
       int sum = rdd.treeReduce(add, depth);
       assertEquals(-5, sum);
     }
   }


public void treeAggregate() {
     JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(-5, -4, -3, -2, -1, 1, 2, 3, 4
), 10);
     Function2<Integer, Integer, Integer> add = new Function2<Integer, Integer, Intege
r>() {
       @Override
       public Integer call(Integer a, Integer b) {
         return a + b;
       }
     };
     for (int depth = 1; depth <= 10; depth++) {
       int sum = rdd.treeAggregate(0, add, add, depth);
       assertEquals(-5, sum);
     }
   }
```

When to use Broadcast variable

As documentation for Spark Broadcast variables states, they are immutable shared variable which are cached on each worker nodes on a Spark cluster.



When to use Broadcast variable?

Before running each tasks on the available executors, Spark computes the task's closure. The closure is those variables and methods which must be visible for the executor to perform its computations on the RDD.

If you have huge array that is accessed from Spark Closures, for example some reference data, this array will be shipped to each spark node with closure.
For example if you have 10 nodes cluster with 100 partitions (10 partitions per node), this Array will be distributed at least 100 times (10 times to each node).
If you use broadcast it will be distributed once per node using efficient p2p protocol.

```
val array: Array[Int] = ??? // some huge array
val broadcasted = sc.broadcast(array)
```

And some RDD

```
val rdd: RDD[Int] = ???
```

In this case array will be shipped with closure each time

```
rdd.map(i => array.contains(i))
```

and with broadcast you'll get huge performance benefit

```
rdd.map(i => broadcasted.value.contains(i))
```

Things to remember while using Broadcast variables:

Once we broadcasted the value to the nodes, we shouldn’t make changes to its value to make sure each node have exact same copy of data. The modified value might be sent to another node later that would give unexpected results.

Joining a large and a small RDD

If the small RDD is small enough to fit into the memory of each worker we can turn it into a broadcast variable and turn the entire operation into a so called map side join for the larger RDD [23]. In this way the larger RDD does not need to be shuffled at all. This can easily happen if the smaller RDD is a dimension table.

```
val smallLookup = sc.broadcast(smallRDD.collect.toMap)
largeRDD.flatMap { case(key, value) =>
  smallLookup.value.get(key).map { otherValue =>
    (key, (value, otherValue))
  }
}
```

Joining a large and a medium size RDD

If the medium size RDD does not fit fully into memory but its key set does, it is possible to exploit this [23]. As a join will discard all elements of the larger RDD that do not have a matching partner in the medium size RDD, we can use the medium key set to do this before the shuffle. If there is a significant amount of entries that gets discarded this way, the resulting shuffle will need to transfer a lot less data.

```
val keys = sc.broadcast(mediumRDD.map(_._1).collect.toSet)
val reducedRDD = largeRDD.filter{ case(key, value) => keys.value.contains(key) }
reducedRDD.join(mediumRDD)
```

It is important to note that the efficiency gain here depends on the filter operation actually reducing the size of the larger RDD. If there are not a lot of entries lost here (e.g., because the medium size RDD is some king of large dimension table), there is nothing to be gained with this strategy.

You can find more details for efficient shuffles in this Databricks presentation.

Which storage level to choose

Cache judiciously

Just because you can cache a RDD in memory doesn’t mean you should blindly do so. Depending on how many times the dataset is accessed and the amount of work involved in doing so, recomputation can be faster than the price paid by the increased memory pressure.

It should go without saying that if you only read a dataset once there is no point in caching it, i it will actually make your job slower. The size of cached datasets can be seen from the Spark Shell.

Which storage level to choose

By dafault Spark will **cache()** data using **MEMORY_ONLY** level, **MEMORY_AND_DISK_SER** can help cut down on GC and avoid expensive recomputations.

- MEMORY_ONLY
    - Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
- MEMORY_AND_DISK
    - Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
- MEMORY_ONLY_SER
    - Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
- MEMORY_AND_DISK_SER
    - Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
- DISK_ONLY

    - Store the RDD partitions only on disk.
- MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.
    - Same as the levels above, but replicate each partition on two cluster nodes.
- OFF_HEAP (experimental)
    - Store RDD in serialized format in Tachyon. Compared to MEMORY_ONLY_SER, OFF_HEAP reduces garbage collection overhead and allows executors to be smaller and to share a pool of memory, making it attractive in environments with large heaps or multiple concurrent applications. Furthermore, as the RDDs reside in Tachyon, the crash of an executor does not lead to losing the in-memory cache. In this mode, the memory in Tachyon is discardable. Thus, Tachyon does not attempt to reconstruct a block that it evicts from memory.

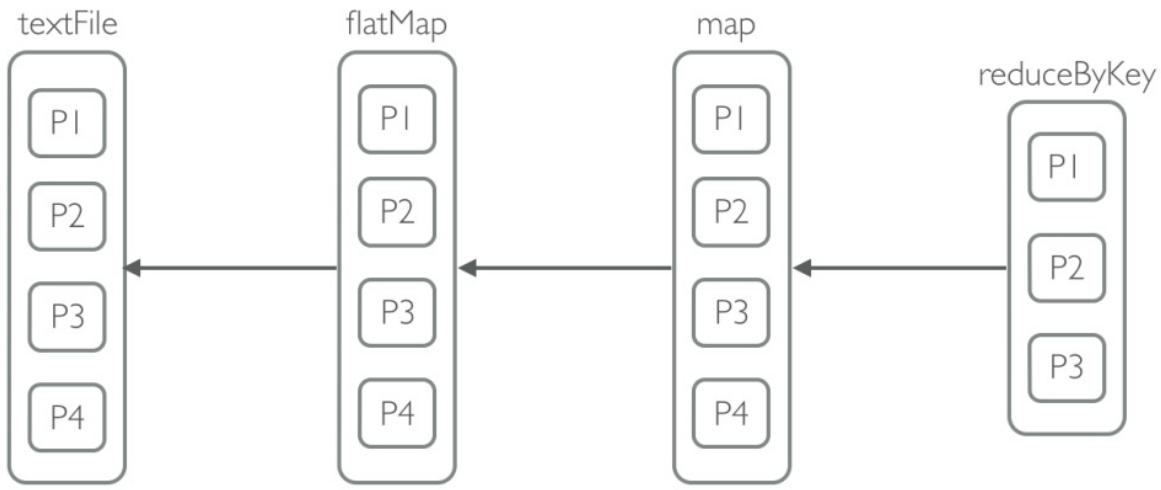
Avoiding Shuffle "Less stage, run faster"

Introduction to shuffling

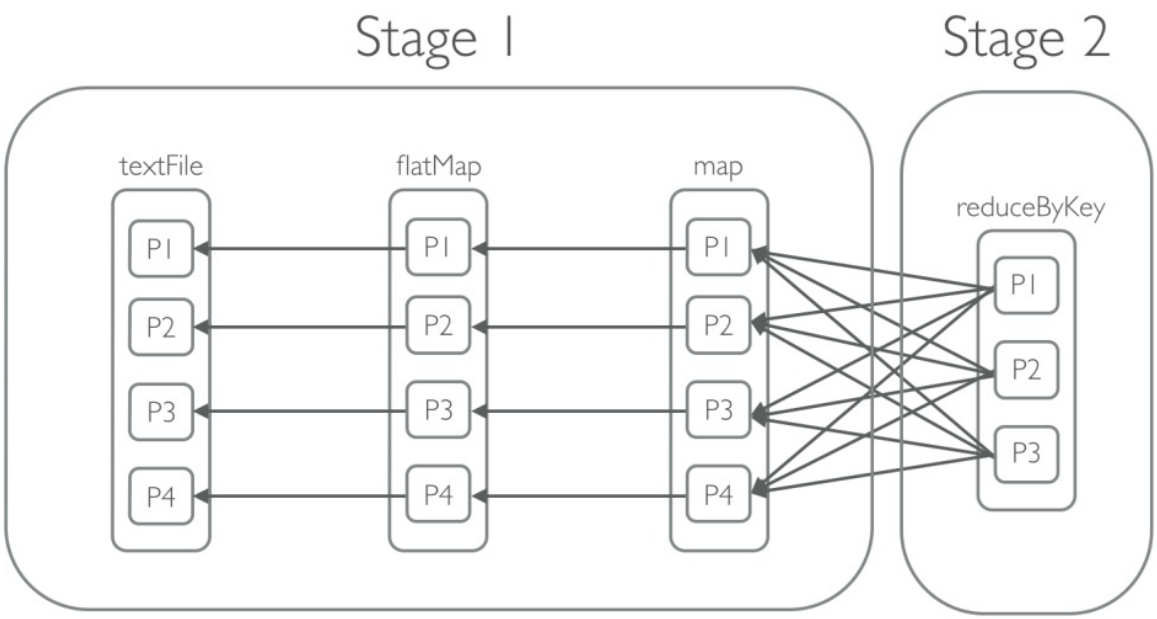
Let's start by taking our good old word-count friend as starting example:

```
rdd = sc.textFile("input.txt")\
.flatMap(lambda line: line.split())\
.map(lambda word: (word, 1))\
.reduceByKey(lambda x, y: x + y, 3)\
.collect()
```

RDD operations are compiled into a **Direct Acyclic Graph** of RDD objects, where each RDD points to the parent it depends on:



At shuffle boundaries, the DAG is partitioned into so-called stages that are going to be executed in order, as shown in next figure. The shuffle is Spark's mechanism for re-distributing data so that it's grouped differently across partitions. This typically involves copying data across executors and machines, making the shuffle a complex and costly operation.



Stages, tasks and shuffle writes and reads are concrete concepts that can be monitored from the Spark shell. The shell can be accessed from the driver node on port 4040.

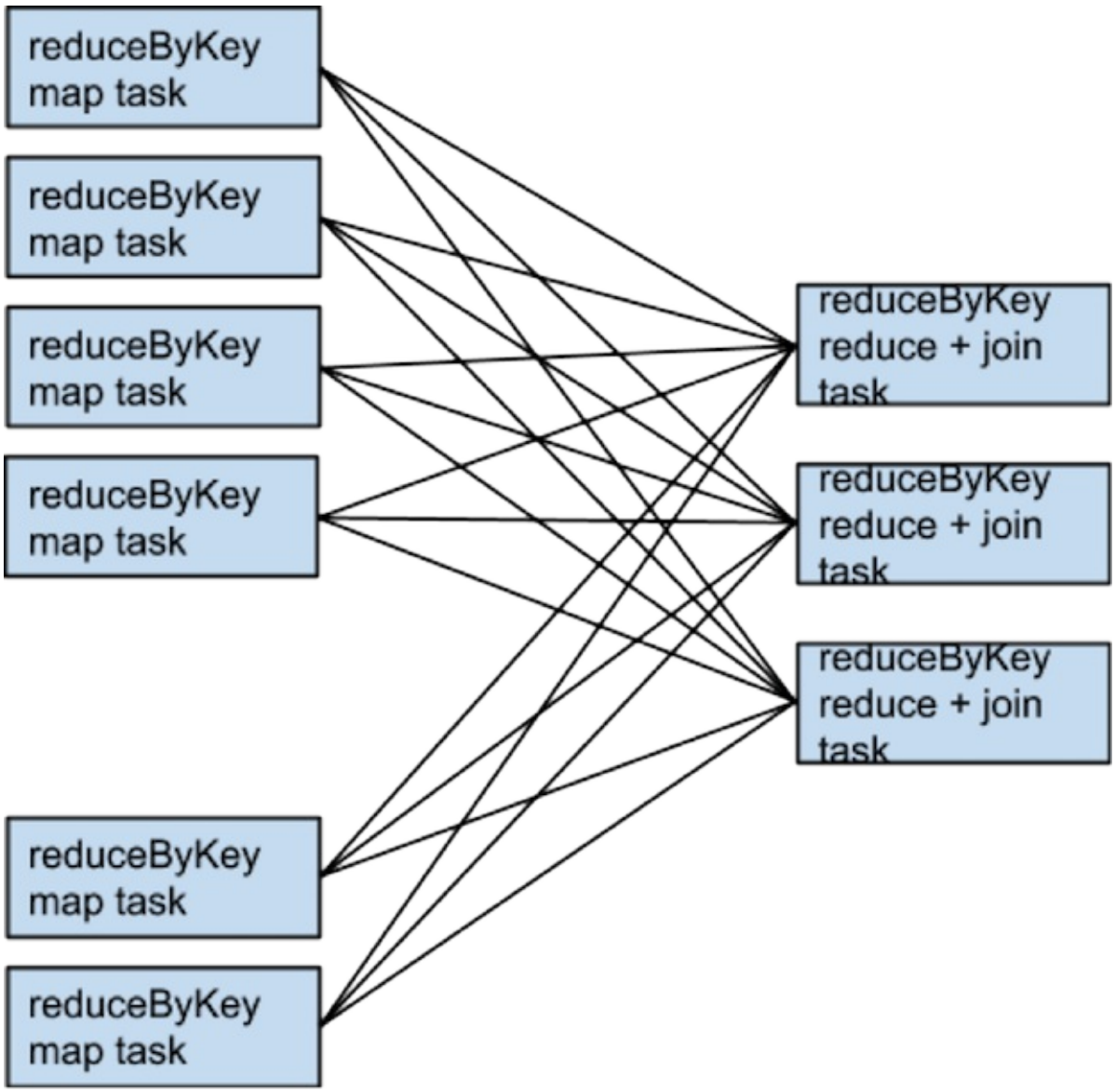
When Shuffles Don’t Happen

It’s also useful to be aware of the cases in which the above transformations will not result in shuffles. Spark knows to avoid a shuffle when a previous transformation has already partitioned the data according to the same partitioner. Consider the following flow:

```
rdd1 = someRdd.reduceByKey(...)
rdd2 = someOtherRdd.reduceByKey(...)
rdd3 = rdd1.join(rdd2)
```

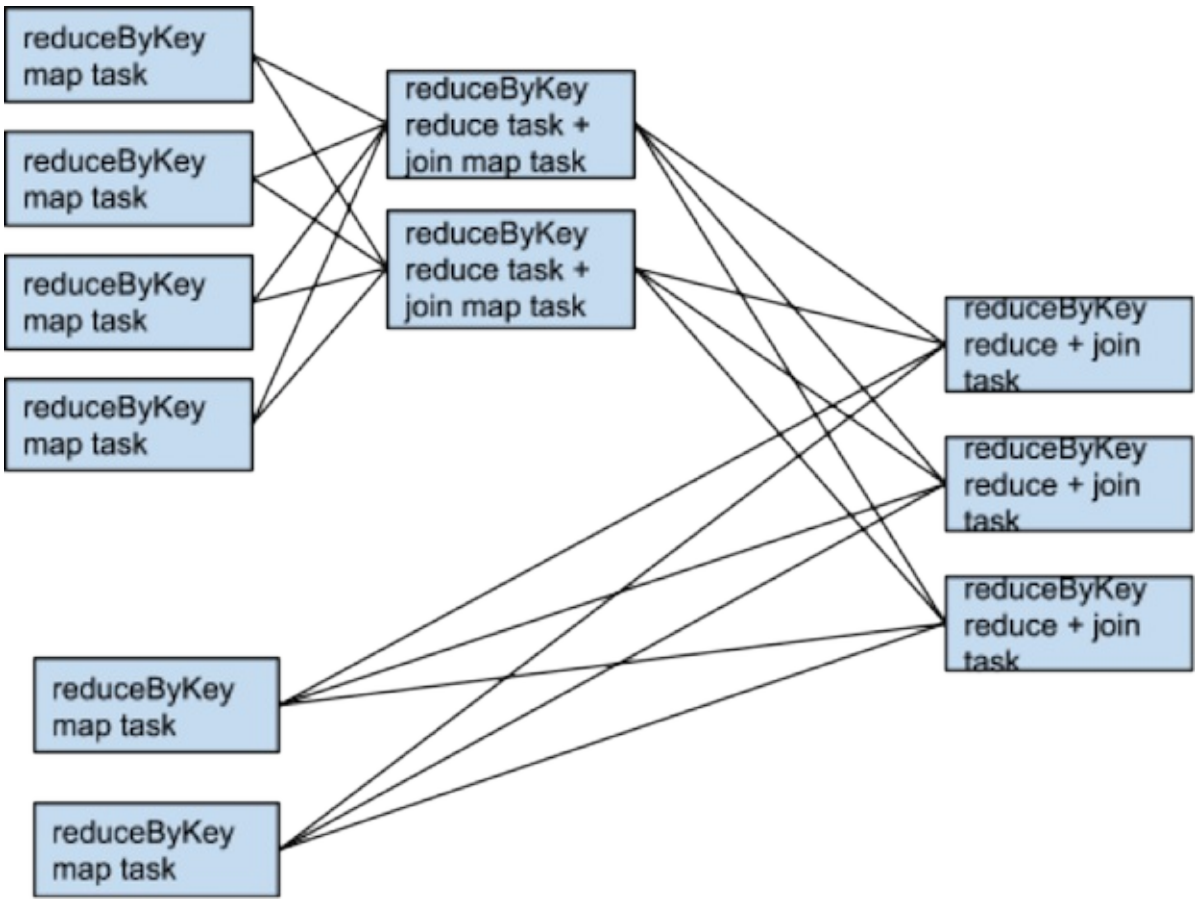
Because no partitioner is passed to `reduceByKey`, the default partitioner will be used, resulting in rdd1 and rdd2 both hash-partitioned. These two `reduceByKeys` will result in two shuffles. If the RDDs have the same number of partitions, the join will require no additional shuffling. Because the RDDs are partitioned identically, the set of keys in any single partition of rdd1 can only show up in a single partition of rdd2. Therefore, the contents of any single output partition of rdd3 will depend only on the contents of a single partition in rdd1 and single partition in rdd2, and a third shuffle is not required.

For example, if some Rdd has four partitions, someOther Rdd has two partitions, and both the `reduceByKeys` use three partitions, the set of tasks that execute would look like:



What if rdd1 and rdd2 use different partitioners or use the default (hash) partitioner with different numbers partitions? In that case, only one of the rdds (the one with the fewer number of partitions) will need to be reshuffled for the join.

Same transformations, same inputs, different number of partitions:



One way to avoid shuffles when joining two datasets is to take advantage of broadcast variables. When one of the datasets is small enough to fit in memory in a single executor, it can be loaded into a hash table on the driver and then broadcast to every executor. A map transformation can then reference the hash table to do lookups.

Best Practices

In general, avoiding shuffle will make your program run faster. All shuffle data must be written to disk and then transferred over the network.
Each time that you generate a shuffling shall be generated a new stage. So between a stage and another one I have a shuffling.

1. **repartition**, **join**, **cogroup**, and any of the ***By** or ***ByKey** transformations can result in shuffles.
2. **map**, **filter** and **union** generate a only stage (no shuffling).
3. Use the built in **aggregateByKey()** operator instead of writing your own aggregations.
4. Filter input earlier in the program rather than later.

When More Shuffles are Better

There is an occasional exception to the rule of minimizing the number of shuffles. An extra shuffle can be advantageous to performance when it increases parallelism. For example, if your data arrives in a few large unsplittable files, the partitioning dictated by the InputFormat might place large numbers of records in each partition, while not generating enough partitions to take advantage of all the available cores. In this case, invoking repartition with a high number of partitions (which will trigger a shuffle) after loading the data will allow the operations that come after it to leverage more of the cluster's CPU.

Another instance of this exception can arise when using the reduce or aggregate action to aggregate data into the driver. When aggregating over a high number of partitions, the computation can quickly become **bottlenecked** on a single thread in the driver merging all the results together. To loosen the load on the driver, one can first use **reduceByKey** or **aggregateByKey** to carry out a round of distributed aggregation that divides the dataset into a smaller number of partitions. The values within each partition are merged with each other in parallel, before sending their results to the driver for a final round of aggregation. Take a look at **treeReduce** and **treeAggregate** for examples of how to do that. (Note that in 1.2, the most recent version at the time of this writing, these are marked as developer APIs, but **SPARK-5430** seeks to add stable versions of them in core.)

This trick is especially useful when the aggregation is already grouped by a key. For example, consider an app that wants to count the occurrences of each word in a corpus and pull the results into the driver as a map. One approach, which can be accomplished with the aggregate action, is to compute a local map at each partition and then merge the maps at the driver. The alternative approach, which can be accomplished with **aggregateByKey**, is to perform the count in a fully distributed way, and then simply **collectAsMap** the results to the driver.

Use the right level of parallelism

Clusters will not be fully utilized unless the level of parallelism for each operation is high enough. **Spark automatically sets the number of partitions of an input file according to its size and for distributed shuffles**. By default **spark create one partition for each block of the file in HDFS it is 64MB by default**.

You can also pass second argument as a number of partition when creating RDD.

Let see example of creating RDD of text file:

```
val rdd= sc.textFile(“file.txt”,5)
```

above statement make a RDD of textFile with 5 partition. Now if we have a cluster with 4 cores then each partition need to process 5 minutes so 4 partition process parallel and 5 partition process after that whenever core will be free so it so final result will be completed in 10 minutes and resources also ideal while only one partition process.

So to overcome this problem we should **make RDD with number of partition is equal to number of cores in the cluster** by this *all partition will process parallel and resources are also used equally.*

As a rule of thumb tasks should take at least 100 ms to execute; you can ensure that this is the case by monitoring the task execution latency from the Spark Shell. If your tasks take considerably longer than that keep increasing the level of parallelism, by say 1.5, until performance stops improving.

DataFrame create a number of partitions equal to **spark.sql.shuffle.partitions** parameter. **spark.sql.shuffle.partitions**'s default value is 200.

How to estimate the size of a Dataset

An approximated calculation for the size of a dataset is

```
number Of Megabytes = M = (N*V*W) / 1024^2
```

where

```
N  =  number of records

V  =  number of variables

W  =  average width in bytes of a variable
```

In approximating **W**, remember:

Type of variable	**Width**
Integers, −127 <= x <= 100	1
Integers, 32,767 <= x <= 32,740	2
Integers, -2,147,483,647 <= x <= 2,147,483,620	4
Floats single precision	4
Floats double precision	8
Strings	maximum lenght

Say that you have a 20,000-observation dataset. That dataset contains

```
1  string identifier of length 20                    20

10  small integers (1 byte each)                     10

4  standard integers (2 bytes each)                   8

5  floating-point numbers (4 bytes each)             20

-------------------------------------------------------

20  variables total                                  58
```

Thus the average width of a variable is

```
W = 58/20 = 2.9  bytes
```

The size of your dataset is

```
M = 20000*20*2.9/1024^2 = 1.13 megabytes
```

This result slightly understates the size of the dataset because we have not included any variable labels, value labels, or notes that you might add to the data. That does not amount to much. For instance, imagine that you added variable labels to all 20 variables and that the

average length of the text of the labels was 22 characters.

That would amount to a total of 20*22=440 bytes or 440/10242=.00042 megabytes.

Explanation of formula

```
M = 20000*20*2.9/1024^2 = 1.13 megabytes
```

N*V*W is, of course, the total size of the data. The 1,0242 in the denominator rescales the results to megabytes.

Yes, the result is divided by 1,0242 even though 1,0002 = a million. Computer memory comes in binary increments. Although we think of k as standing for kilo, in the computer business, k is really a “binary” thousand, 210 = 1,024. A megabyte is a binary million—a binary k squared:

```
1 MB = 1024 KB = 1024*1024 = 1,048,576 bytes
```

With cheap memory, we sometimes talk about a gigabyte. Here is how a binary gig works:

```
1 GB = 1024 MB = 10243 = 1,073,741,824 bytes
```

How to estimate the number of partitions, executor's and driver's params (YARN Cluster Mode)

```
yarn.nodemanager.resource.memory-mb = ((Node's Ram GB - 2 GB) * 1024) MB
```

```
Total Number Of Node's Core = yarn.nodemanager.resource.cpu-vcores
```

- Executor's params (Worker Node):

- `Executor (VM) x Node = ((total number of Node's core) / 5) - 1`
  - 5 is the upper bound for cores per executor because more than 5 cores per executor can degrade HDFS I/O throughput.
  - If the total number of Node's core is less than 12 we divide It by 2
- `numExecutors (Number of executorns to launch for this session) = number of Nodes * Executor (VM) x Node`
- `executorCores (Number of cores to use for each executor) = (total number of Node's`

`core - 5) / Executor x Node`

- `executorMemory (Amount of memory to use per executor process) = (yarn.nodemanager.resource.memory-mb - 1024) / (Executor (VM) x Node + 1)`

For the **executorMemory** We have to take a further reasoning. If you review the **BlockManager** source code:
./core/src/main/scala/org/apache/spark/storage/BlockManager.scala

You will note that the memory allocation is based on the algorithm:

`Runtime.getRuntime.maxMemory * memoryFraction * safetyFraction.`

where **memoryFraction = spark.storage.memoryFraction** and **safetyFraction = spark.storage.safetyFraction**

The default values of **spark.storage.memoryFraction** and **spark.storage.safetyFraction** are respectively **0.6** and **0.9** so the real **executorMemory** is:

`executorMemory = ((yarn.nodemanager.resource.memory-mb - 1024) / (Executor (VM) x Node + 1)) * memoryFraction * safetyFraction.`

- Driver's params (Application Master Node):

- `driverCores = executorCores`
- `driverMemory = executorMemory`

Example

I have 3 Worker nodes and one Application Master Node each with **16 vCPUs, 52 GB memory**

`yarn.nodemanager.resource.memory-mb = (52 - 2) * 1024 = 51200 MB`

`yarn.scheduler.maximum-allocation-mb = 20830 MB (Must be greater than executorMemory)`

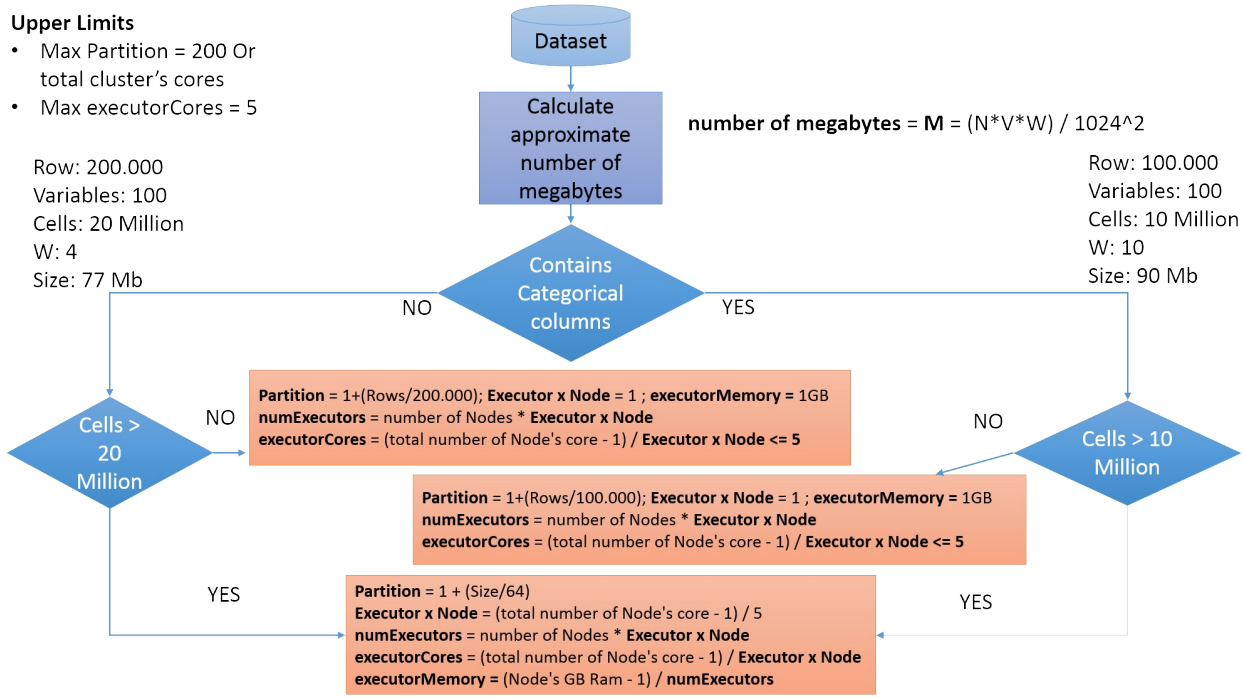
- Executor's params (Worker Node):

- `Executor x Node = ((16) / 5) - 1 = 2`
- `numExecutors = 2 * 4 = 8`
- `executorCores = (16 - 5) / 2 = 5`
- `executorMemory = ((51200 - 1024) / 3) * 0.6 * 0.9 = 16725,33 MB * 0.6 * 0.9 = 9031,68 MB`

- Driver's params (Application Master Node):

- `driverCores = 5`
- `driverMemory = 16725,33 MB * 0.6 * 0.9 = 9031,68 MB`

See in diagram how params are estimated:



Example

I have to process a dataset that have 10.000.000 of rows and 100 double variables.

```
number of megabytes  =  M  =  10.000.000*100*8/1024^2 =  5.722 megabytes
```

`Partition = 5.722/64 = 89`

As in the previous example, I have 3 **Worker** nodes and one **Application Master** Node each with **16 vCPUs, 52 GB memory**

`yarn.nodemanager.resource.memory-mb = (52 - 2) * 1024 = 51200 MB`

`yarn.scheduler.maximum-allocation-mb = 20830 MB (Must be greater than executorMemory)`

- Executor's params (Worker Node):

- `Executor x Node = ((16) / 5) - 1 = 2`
- `numExecutors = 2 * 4 = 8`
- `executorCores = (16 - 5) / 2 = 5`
- `executorMemory = ((51200 - 1024) / 3) * 0.6 * 0.9 = 16725,33 MB * 0.6 * 0.9 = 9031,68 MB`

- Driver's params (Application Master Node):

- `driverCores = 5`
- `driverMemory = 16725,33 MB * 0.6 * 0.9 = 9031,68 MB`

Serialization

Serialization plays an important role in the performance of any distributed application. Formats that are slow to serialize objects into, or consume a large number of bytes, will greatly slow down the computation. Often, this will be the first thing you should tune to optimize a Spark application. **The Java default serializer has very mediocre performance** regarding runtime as well as the size of its results. Therefore the Spark team recommends to use the Kryo serializer instead.

The following code shows an example of how you can turn on Kryo and how you can register the classes that you will be serializing:

```
val conf = new SparkConf().setAppName(...).setMaster(...)
     .set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
     .set("spark.kryoserializer.buffer.max", "128m")
     .set("spark.kryoserializer.buffer", "64m")
     .registerKryoClasses(
       Array(classOf[ArrayBuffer[String]], classOf[ListBuffer[String]])
     )
```

You can find another example here BasicAvgWithKryo

Tuning Java Garbage Collection

Understanding Memory Management in Spark

A Resilient Distributed Dataset (RDD) is the core abstraction in Spark. Creation and caching of RDD's closely related to memory consumption. Spark allows users to persistently cache data for reuse in applications, thereby avoid the overhead caused by repeated computing. One form of persisting RDD is to cache all or part of the data in JVM heap. Spark's executors divide JVM heap space into two fractions: one fraction is used to store data persistently cached into memory by Spark application; the remaining fraction is used as JVM heap space, responsible for memory consumption during RDD transformation. We can adjust the ratio of these two fractions using the **spark.storage.memoryFraction** parameter to let Spark control the total size of the cached RDD by making sure it doesn't exceed RDD heap space volume multiplied by this parameter's value.

The unused portion of the RDD cache fraction can also be used by JVM.

Therefore, GC analysis for Spark applications should cover memory usage of both memory fractions.

When an efficiency decline caused by GC latency is observed, we should first check and make sure the Spark application uses the limited memory space in an effective way. The less memory space RDD takes up, the more heap space is left for program execution, which increases GC efficiency; on the contrary, excessive memory consumption by RDDs leads to significant performance loss due to a large number of buffered objects in the old generation.

So when GC is observed as too frequent or long lasting, it may indicate that memory space is not used efficiently by Spark process or application.

You can improve performance by explicitly cleaning up cached RDD's after they are no longer needed.

Choosing a Garbage Collector

The Hotspot JVM version 1.6 introduced the **Garbage-First GC (G1 GC)**. The **G1** collector is planned by Oracle as the long term replacement for the **CMS GC**. Most importantly, respect to the **CMS** the **G1** collector aims to achieve both **high throughput** and **low latency**.

Is recommend trying the **G1 GC** because Finer-grained optimizations can be obtained through GC log analysis [17].

To avoid full GC in **G1 GC**, there are two commonly-used approaches:

1. Decrease the **InitiatingHeapOccupancyPercent** option's value (the default value is 45), to let G1 GC starts initial concurrent marking at an earlier time, so that we are more likely to avoid full GC.
2. Increase the **ConcGCThreads** option's value, to have more threads for concurrent marking, thus we can speed up the concurrent marking phase. Take caution that this option could also take up some effective worker thread resources, depending on your workload CPU utilization.

```
spark.executor.extraJavaOptions = -XX:+UseConcMarkSweepGC -XX:+CMSClassUnloadingEnable
d -XX:UseG1GC XX:InitiatingHeapOccupancyPercent=35 -XX:ConcGCThread=20
```

References

- [1] Controlling Parallelism in Spark
  - http://www.bigsynapse.com/spark-input-output
- [2] Avoid GroupByKey
  - https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html
- [3] Everyday I'm Shuffling - Tips for Writing Better Spark Programs, Stra…
  - http://www.slideshare.net/databricks/strata-sj-everyday-im-shuffling-tips-for-writing-better-spark-programs
- [4] How-to: Tune Your Apache Spark Jobs (Part 1)
  - http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/
- [5] How-to: Tune Your Apache Spark Jobs (Part 2)
  - http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/
- [6] Top 5 Mistakes to Avoid When Writing Apache Spark Applications
  - https://intellipaat.com/blog/top-5-mistakes-writing-apache-spark-applications/
- [7] Spark best practices
  - https://robertovitillo.com/2015/06/30/spark-best-practices/
- [8] Best practice for retrieving big data from RDD to local machine
  - http://stackoverflow.com/questions/21698443/spark-best-practice-for-retrieving-big-data-from-rdd-to-local-machine
- [9] Optimizing Spark Machine Learning for Small Data
  - http://eugenezhulenev.com/blog/2015/09/16/spark-ml-for-big-and-small-data/
- [10] Tuning and Debugging in Apache Spark
  - http://www.slideshare.net/pwendell/tuning-and-debugging-in-apache-spark
- [11] Advantage of Broadcast Variables
  - http://stackoverflow.com/questions/26884871/advantage-of-broadcast-variables
- [12] When to use Broadcast variable?
  - https://blog.knoldus.com/2016/04/30/broadcast-variables-in-spark-how-and-when-to-use-them/
- [13] Implement treeReduce and treeAggregate
  - https://issues.apache.org/jira/browse/SPARK-2174
- [14] Shufflling and repartitioning of RDD's in apache spark
  - https://blog.knoldus.com/2015/06/19/shufflling-and-repartitioning-of-rdds-in-apache-spark/
- [15] Resource Allocation Configuration for Spark on YARN:
  - https://www.mapr.com/blog/resource-allocation-configuration-spark-yarn
- [16] Apache Spark: Config Cheatsheet:

    - http://c2fo.io/c2fo/spark/aws/emr/2016/07/06/apache-spark-config-cheatsheet/
- [17] Tuning Java Garbage Collection for Apache Spark Applications:
    - https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html
- [18] How to set Apache Spark Executor memory
    - http://stackoverflow.com/questions/26562033/how-to-set-apache-spark-executor-memory
- [19] How to interpret RDD.treeAggregate
    - http://stackoverflow.com/questions/29860635/how-to-interpret-rdd-treeaggregate
- [20] Apache Spark 1.1: MLlib Performance Improvements
    - https://databricks.com/blog/2014/09/22/spark-1-1-mllib-performance-improvements.html
- [21] Spark group multiple rdd items by key
    - http://stackoverflow.com/questions/36447057/spark-group-multiple-rdd-items-by-key
- [22] Spark Corner Cases
    - http://codingjunkie.net/spark-corner-cases/
- [23] Writing efficient Spark jobs
    - http://fdahms.com/2015/10/04/writing-efficient-spark-jobs/