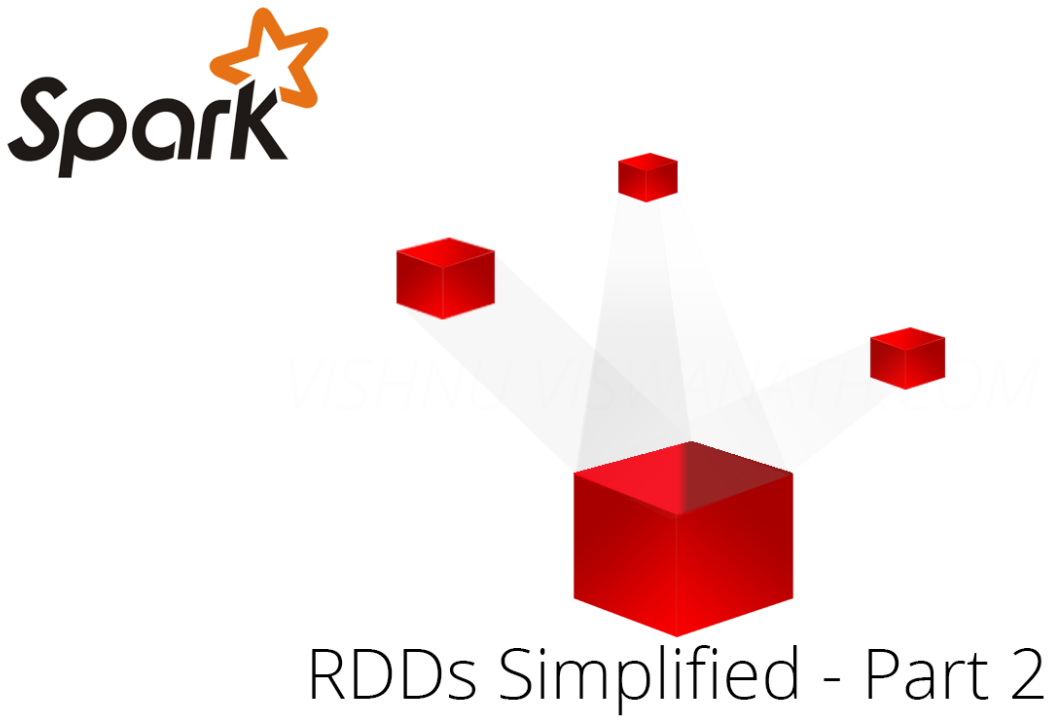
Spark RDDs Simplified - Part 2

February 28, 2016. Estimated read time: 5 minutes



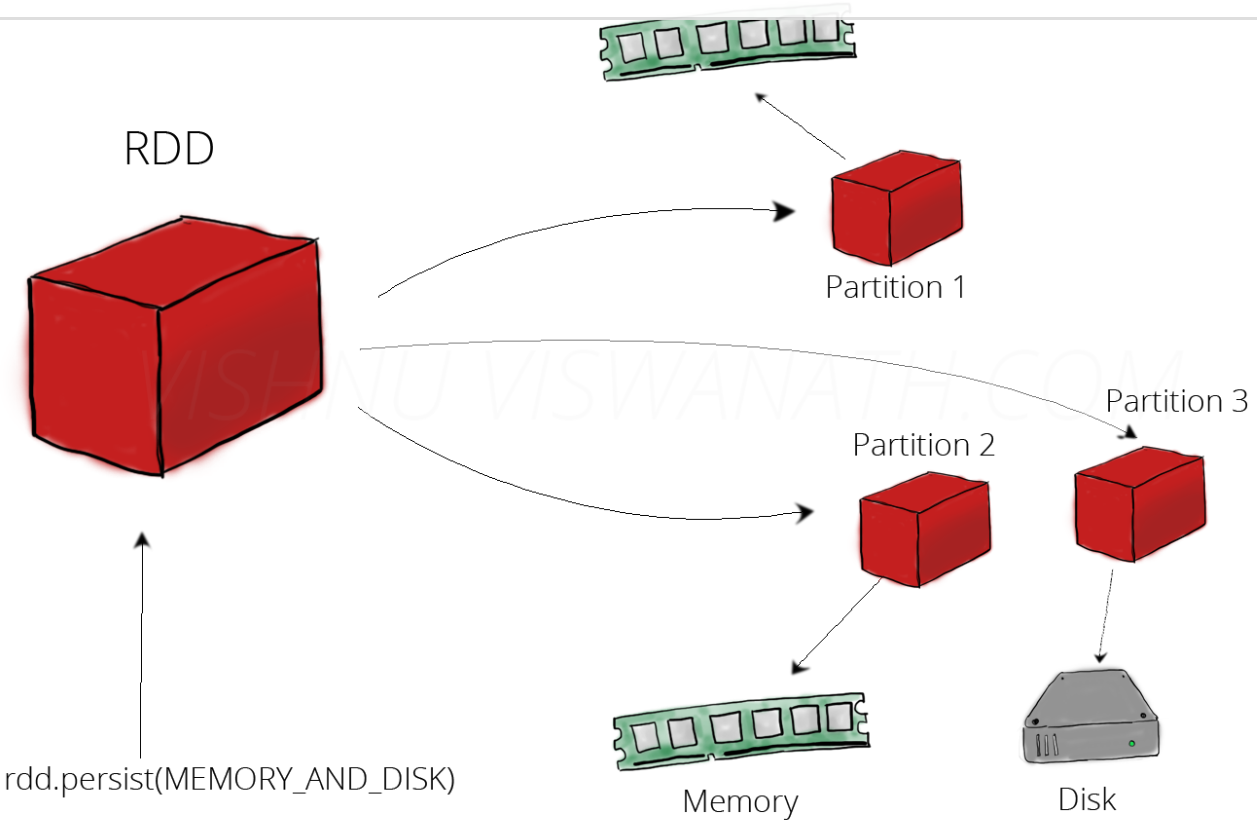
This is Part 2 of the blog **Spark RDDs Simplified.** In this part, I am trying to cover the topics **Persistence**, **Broadcast** variables and **Accumulators**. You can read the first part from here

Persistence

In my previous blog, I talked about caching which can be used to avoid recomputation of RDD lineage by saving its contents in memory. If there is not enough memory in the cluster, you can tell spark to use disk also for saving the RDD by using the method *persist()*.

```
rdd.persist(StorageLevel.MEMORY_AND_DISK)
```

In fact Caching is a type of persistence with StorageLevel -*MEMORY_ONLY*. If you use MEMORY_ONLY as the *Storage Level* and if there is not enough memory in your cluster to hold the entire RDD, then some partitions of the RDD cannot be stored in memory and will have to be recomputed every time it is needed. If you don't want this to happen, you can use the StorageLevel - *MEMORY_AND_DISK* in which if an RDD does not fit in memory, the partitions that do not fit are saved to disk.



In the above example, the RDD has 3 partitions and there are 2 nodes in the cluster. Also, memory available in the cluster can hold only 2 out of 3 partitions of the RDD. Here, partitions 1 and 2 can be saved in memory where as partition 3 will be saved to disk. Another StorageLevel, *DISK_ONLY* stores all the partitions on the disk.

In the above method, the RDDs are not serialized before saving to Memory, there

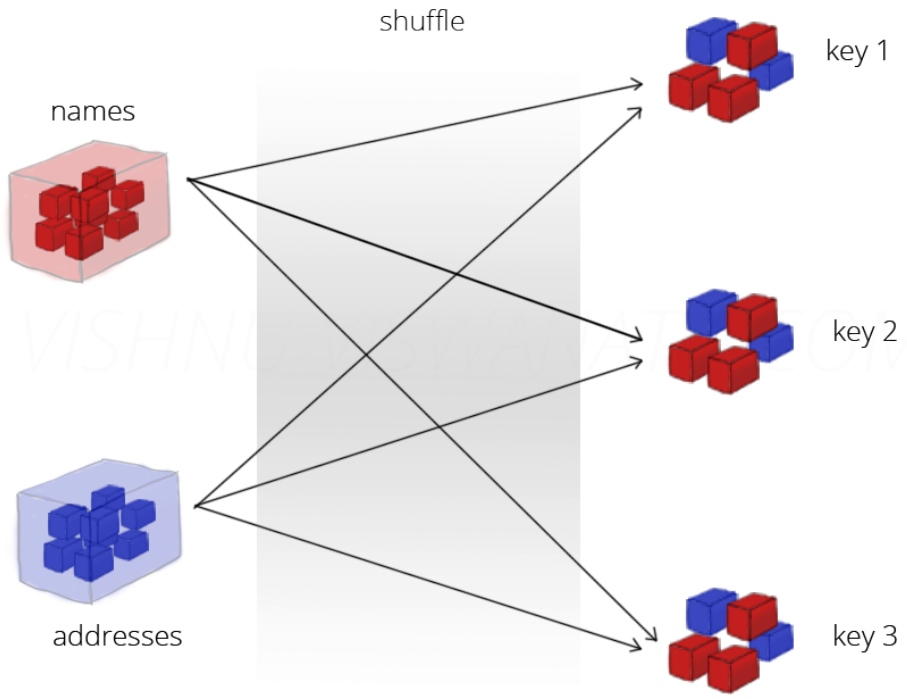
There are a few more StorageLevels which I did not mention here, you can find more details about it here

Broadcast variables

A broadcast variable, is a type of shared variable, used for broadcasting data across the cluster. Hadoop MapReduce users can relate this to distributed cache. Let us first understand why we need a broadcast variable. Take a look at the below example, where *names* is joined with *addresses*.

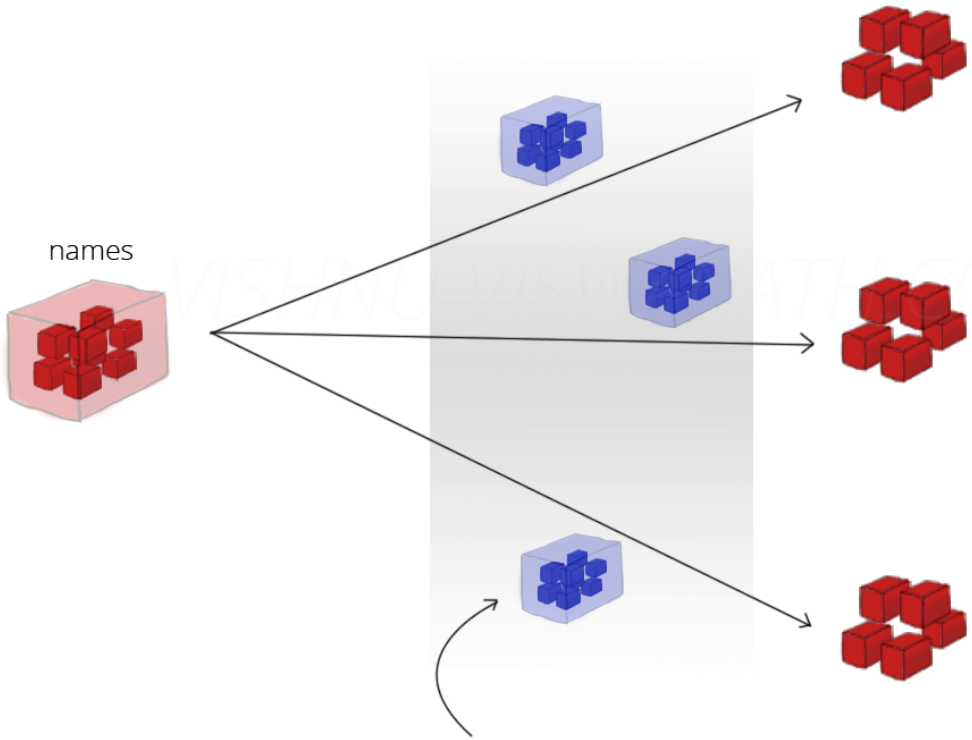
```
val names = sc.textFile("/names").map(line => (line.split(",")(3),line))
val addresses = sc.textFile("/address").map(line=>(line.split(",")(0),line))
names.join(addresses)
```

Here, both names and addresses will be shuffled over the network for performing the join which is not efficient since any data transfer over the network will reduce the execution speed.



Another approach is, if one of the RDDs is small in size, we can choose to send it along with each task. Consider the below example

```
val names = sc.textFile("/names").map(line => (line.split(",")(3),line))
val addresses = sc.textFile("/address").map(line=>(line.split(",")(0),line))
val addressesMap = addresses.collect().toMap
```



addresses is sent along with each task

This is also inefficient since we are sending sizable amount of data over the network for each task.

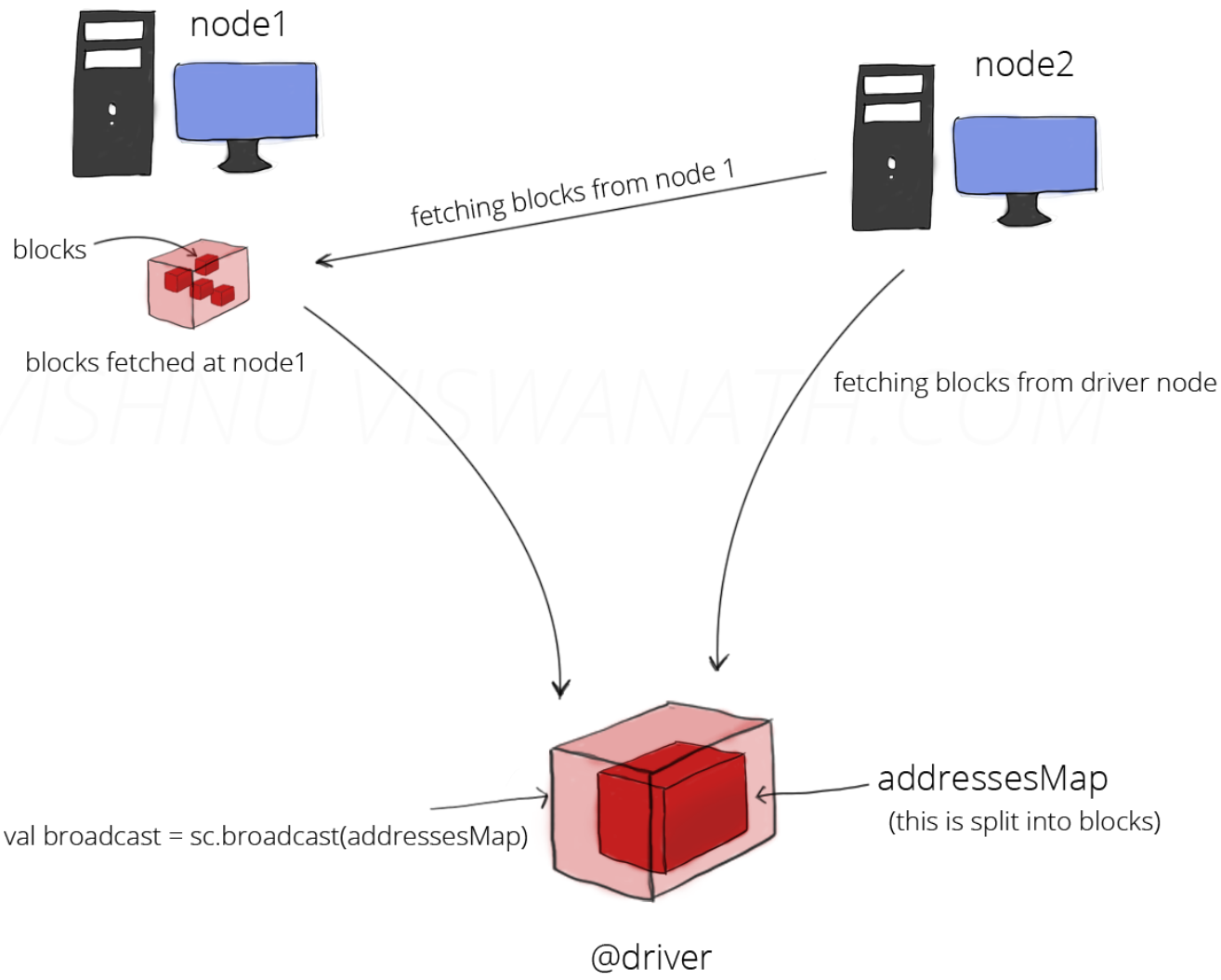
So how do we overcome this problem? By means of **broadcast** variables.

```
val names = sc.textFile("/names").map(line => (line.split(",")(3),line))
val addresses = sc.textFile("/address").map(line=>(line.split(",")(0),line))
val addressesMap = addresses.collect().toMap
```

If a variable is broadcasted, it will be sent to each node only once, thereby reducing network traffic.

Broadcast variables are read-only, broadcast.value is an immutable object

Spark uses BitTorrent like protocol for sending the broadcast variable across the cluster, i.e., for each variable that has to be broadcasted, initially the driver will act as the only source. The data will be split into blocks at the driver and each leecher *(receiver)* will start fetching the block to it's local directory. Once a block is completely received, then that leecher will also act as a source for this block for the rest of the leechers *(This reduces the load at the machine running driver).* This is continued for rest of the blocks. So initially, only the driver is the source and later on the number of sources increases - because of this, rate at which the blocks are fetched by a node increases over time.



(added) by each executors. Finally all these values are aggregated back at the driver.

```
val names = sc.textFile("/names").map(line => (line.split(",")(3),line))
val addresses = sc.textFile("/address").map(line=>(line.split(",")(0),line))
val addressesMap = addresses.collect().toMap
val broadcast = sc.broadcast(addressesMap)
val joined = names.map(v=>(v._2,(broadcast.value(v._1))))

val accum = sc.accumulator(0,"india_counter")
joined.foreach(v=> if (v._2.contains("india")) accum += 1)

//we cannot do below operations on accumulators of the type Int
//joined.foreach(v=> if (v._2.contains("india")) accum -= 1)
//joined.foreach(v=> if (v._2.contains("india")) accum *= 1)
//error: value *= is not a member of org.apache.spark.Accumulator[Int]
```

That concludes part 2 of the blog **Spark RDDs Simplified**, thanks for reading. Please leave a comment for any clarifications or queries.

Continue reading