Overview of Spark

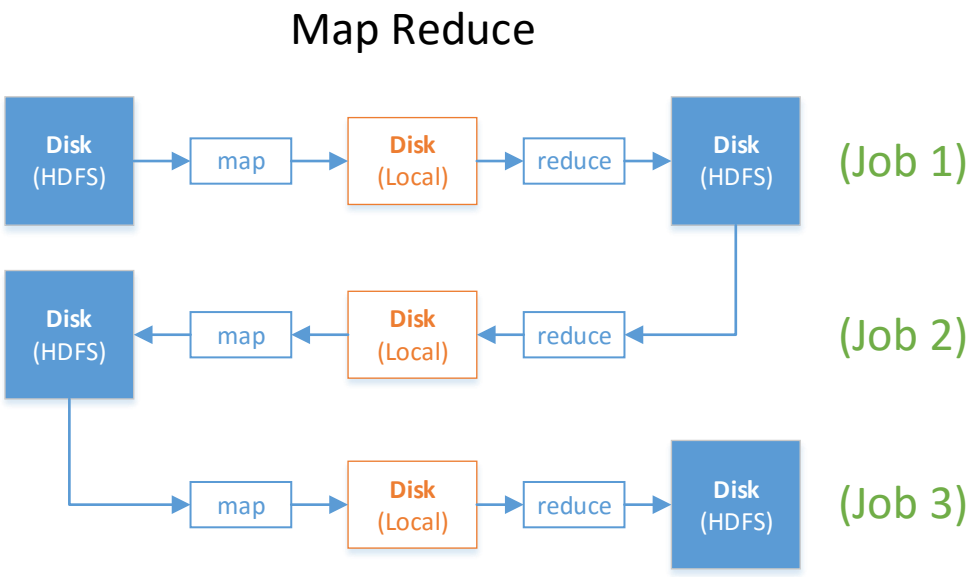
Author: Peter Bugaj

Table of Contents

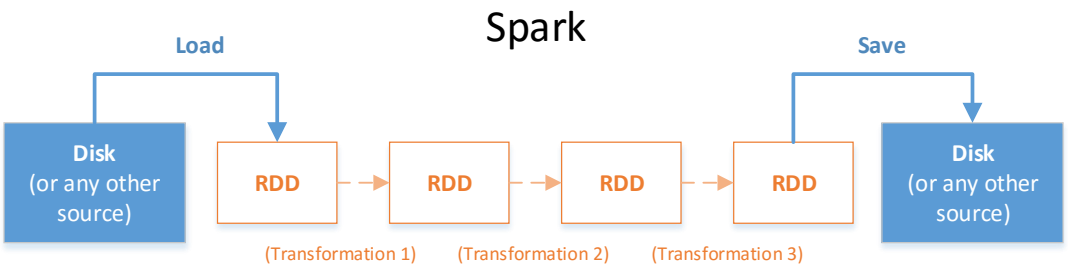
Overview

Spark is a tool used for processing and analyzing large amounts of data. It is an improvement built on top of the concept of the MapReduce framework, solving many of its problems while adding new features.

One key advantage of Spark is its use of in memory computation. In MapReduce, intermediate results are stored and read from the hard disk very frequently – for every map and reduce task performed. When running multiple jobs in sequence, these input and output operations are repeated each time. One such example is given below, showing three jobs reading and processing data from HDFS:



The problem here is that accessing the hard disk is much slower than accessing the RAM (in memory). Spark provides a solution by keeping as much of the computed results in RAM as possible, including the results that have to be frequently iterated over. This piece alone allows Spark to support many other different types of functionalities such as the data processing necessary for machine learning.



Another key advantage of Spark is its ease of use and powerful interface. As seen in the examples in MapReduce, whenever a job needs to run, it has to be first kicked of as a compiled jar from within Hadoop. An input and output location in HDFS then needs to be created – input and output formats need to be specified. The creation of a data pipeline is also very tedious. To transform one chunk of data into a different chunk of data, a mapping class needs to be provided for an input folder, along with a reduce class created for the output. To transform the data again and possibly multiple times, a separate Map and Reduce class has to be created every single time.

Defining a single MapReduce Job

```
class SampleMapper extends Mapper[Object, Text, Text, IntWritable] {
  override
  def map(
    key: Object,
    value: Text,
    context: Mapper[Object, Text, Text, IntWritable]
    . . .
```

```
class MyReducer extends Reducer[Text, IntWritable, Text, IntWritable] {
  override
  def reduce(
    key: Text,
    values: Iterable[IntWritable],
    context: Reducer[Text, IntWritable, Text, IntWritable]#Context) = {
    . . .
```

```
val job = new Job(conf, "Sample")
job.setJarByClass(classOf[MyMapper])
job.setMapperClass(classOf[MyMapper])
job.setCombinerClass(classOf[MyReducer])
job.setReducerClass(classOf[MyReducer])
. . .

job.setMapOutputKeyClass(classOf[CompositeKey]);
job.setMapOutputValueClass(classOf[IntWritable]);
. . .
```

and other key value definitions and path specifications . . .

Spark solves all this by removing this tedious detail. It can be told to read data from any source, whether from a database, a file, a chunk of data sitting in HDFS, a previous result computed by a pipeline in Spark, or a stream flowing over a network. Transforming this data from one type of source to another can then be done in one function, typically written in one line – rather than have the transformation be defined in two separate classes. Spark provides many of such functions, some supporting very complex operations on manipulating or evaluating the data.

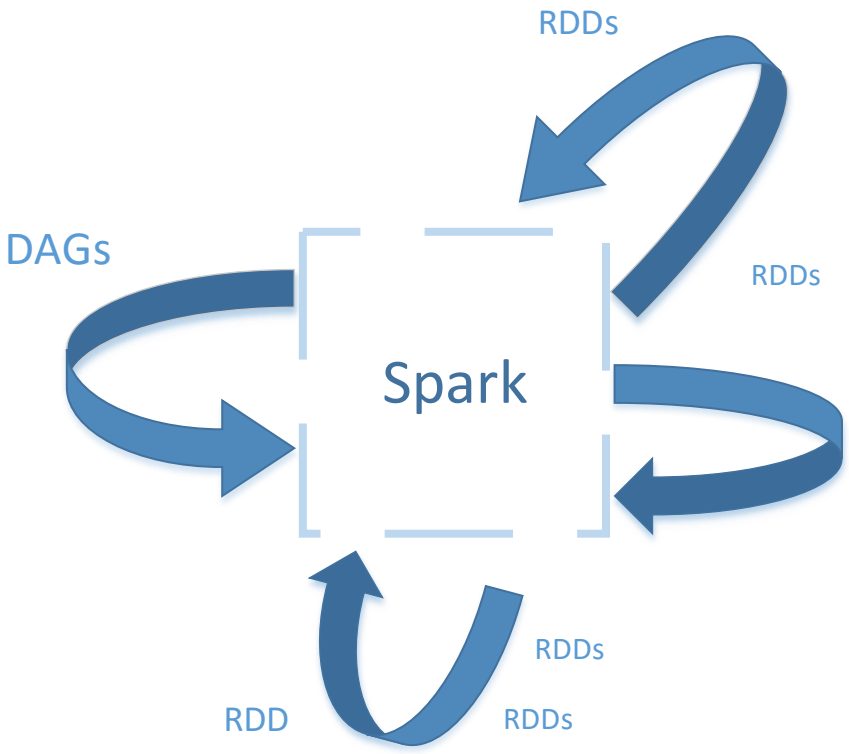
Spark Equivalent

```
val inputRdd = sc.textFile("gs://spark-examplespark-example.txt", 8);
val sum = inputRdd.map(x => x.split(",")(3)).reduce(_ + _);

println(sum);
```

Spark also allows you to string a series of transformation together to define a complete pipeline on the source of data, using lazy evaluation – lazy evaluation meaning that the transformations are executed only when the job itself runs. Since everything is stored in memory, each section of the pipeline can be executed on individual partitions of data, without the pipeline waiting for other functions running on other partitions to complete – assuming the data is not divided into stages. A pipeline in MapReduce would require for each individual job to complete, before moving on to run the next job.

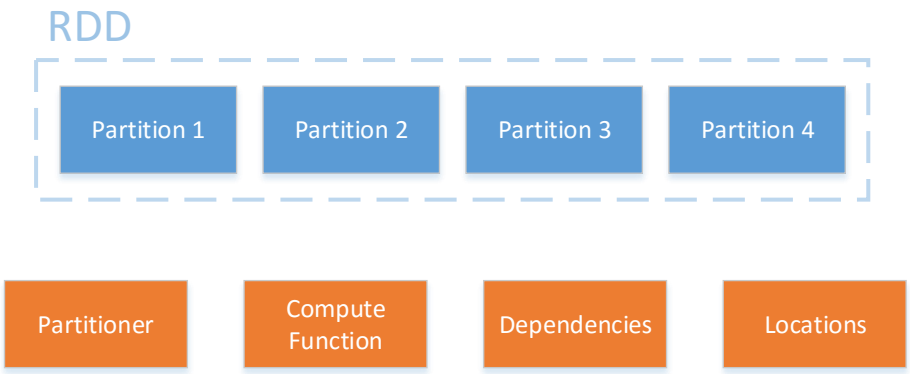
Interfaces

Two main interfaces create the simplicity in Spark and its' ease of use. One is the Resilient Distributed Dataset, abbreviating to just RDD. The second interface is the DAG – the Directed Acyclic Graph. All high level operations that you create to process and analyze your data work with the RDD interface. The DAG is then used by the framework to decide on how operations are to be scheduled, on the optimizations required, and what needs to be re-executed in case of failure.



RDD

As the un-abbreviated name implies, an RDD is a resilient distributed collection of data. The data can be of any variable type or object. Unlike in the MapReduce framework, the source of this data can come from anywhere, as long as there is an API available to read that source into memory. To make this collection distributed, the data within the RDD is divided into partitions. Each partition is processed in parallel by multiple cores and nodes within the cluster. Given prior knowledge of the data's structure and the transformation intended on it, custom partitioners can be specified. A visualization of the RDD is shown:



Structure

Partitions: Partitions are the chunks of data belonging to the RDD. The partitions are distributed among multiple nodes and can be processed in parallel. Depending on the transformations, they can be processed independently. For example, if each partition within an RDD is only dependent on one parent, then the pipeline can be computed for each partition in one go, without waiting for other partitions to complete.

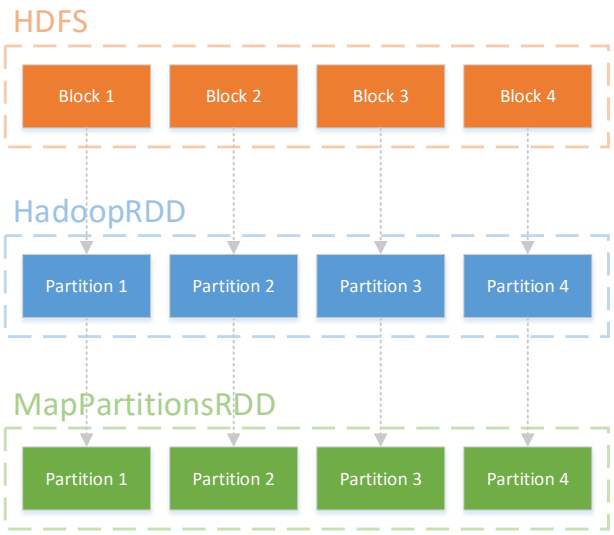
Dependencies: The parent RDD the current RDD is dependent on.

Partitioner: Determines how the collection of data within the RDD is split amongst the partitions. By default, Spark uses the Hash Partitioner. However based on the structure of the data, partitioners such as the Range Partitioner can be specified as well. For example, if you plan to sort a very large dataset in an RDD and the data is partially sorted (except for a few rows being off), then you can partition the data by range and and sort the data within each partition instead. Partitioners can also help lookup operations be more efficient because they narrow down the search to the partition a key belongs to.

Locations: The best place within the cluster for the Spark framework to process each partition on. This would be decided based on the locality of the data.

Compute function: The function to compute on each partition, given the partitions from the parent RDDs specified in the list of dependencies.

```
val exampleTextFile = sc.textFile("hdfs://sample.txt")
```



HadoopRDD

Dependencies:
The sample.txt file.

Partitions:
4 HDFS blocks.

Partitioner:
None (default Hash Partitioner).

Locations:
Location of HDFS blocks.

Compute function:
Load each block into memory.

MapPartitionsRDD

Dependencies:
Parent RDD.

Partitions:
4HDFS blocks.

Partitioner:
None (defaul Hash Partitioner).

Locations:
Location of HDFS blocks.

Compute function:
Compute the parent RDD, read each block, and output an array of Strings, one for each line.

Additional Properties

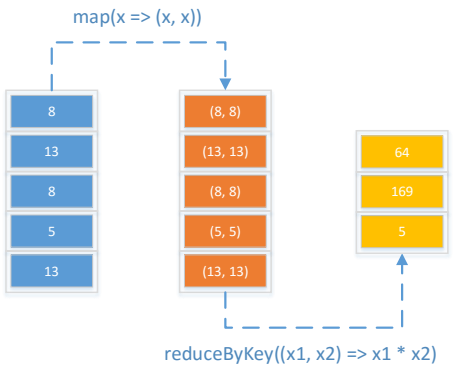
Aside to the structure defined above, RDDs have multiple properties used to help improve efficiency. RDDs are immutable, meaning that an RDD can only be transformed into a new RDD, without the old RDD being modified. This allows computations to remain consistent, because no matter how many times individual partitions are recomputed during failure, the final resulting collection remains the same – since the starting collection and the transformations are unchanged.

RDDs persist in memory and are only ever stored on the disk when data either needs to spill, needs to be shuffled, or if there is a need to cache the RDD directly in some form of stable storage. This in memory feature is very useful because it allows for both fast computations and re-computations in memory, without having the need to worry about IO calls. Recovery becomes easier as well.

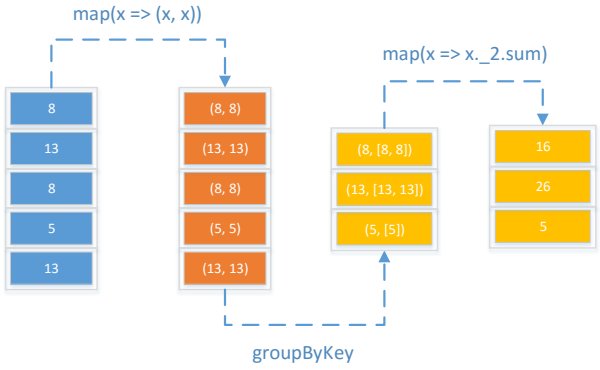
RDDs are resilient. Each RDD stores a log of transformations in a graph, allowing it to replay those transformations during failure. Since each RDD is divided into partitions, only those partitions existing on the failed nodes need to be re-computed. The in memory feature makes this process even faster.

Operations

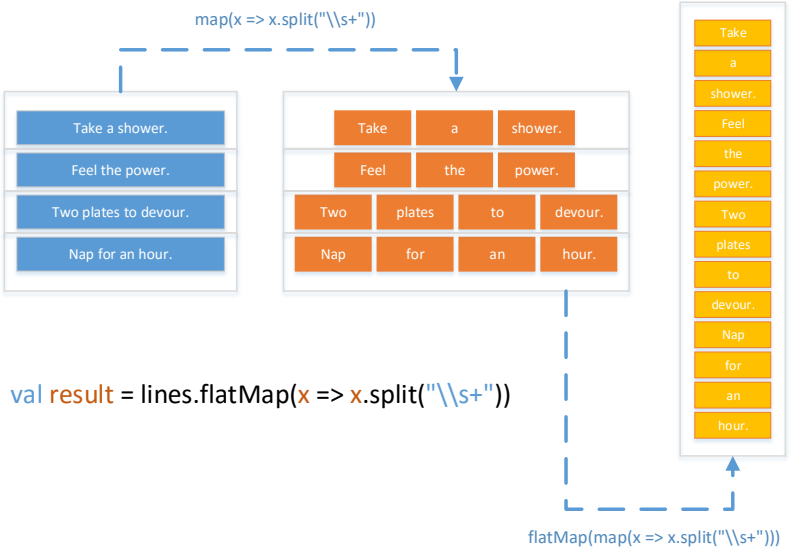
Transformations: There are two main types of operations performed on an RDD. Transformations and actions. Transformations change one RDD into another. They do this by applying the compute function on every partition within the RDD. They form the component of the DAG, creating the edges that exists between the RDDs. Each transformation for an RDD is logged within this graph, allowing it to be re-run on partitions lost during node failures. Transformation examples include the map, flatMap, groupByKey, reduceByKey, and coalesce functions. Other visual representations are shown below:




```
val result = numbers.
  map(x => (x, x)).
  reduceByKey((x1, y2) => x1 * x2))
```

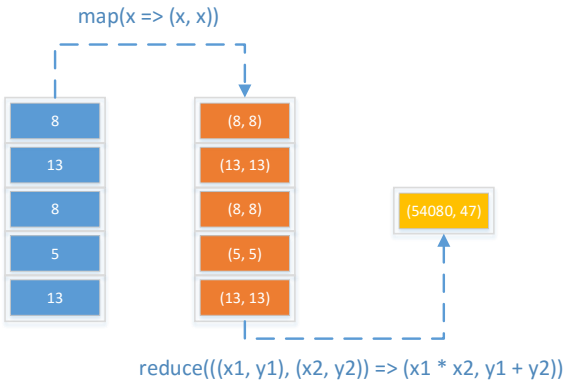



```
val result = numbers.
  map(x => (x, x)).
  groupByKey().
  map(x => x._2.sum)
```

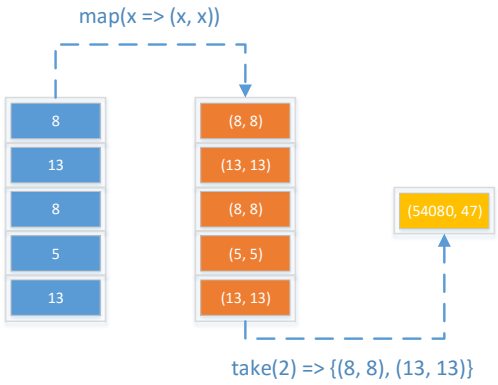



```
val result = lines.flatMap(x => x.split("\\s+"))
```

Actions: An action operates across all partitions within the RDD and returns a new value. In order for it to complete, all transformations must finish operating on each partition. Example of actions include reduce, foreach, count, take, and collect.




```
val result = numbers.
  map(x => (x, x)).
  reduce(((x1, y1), (x2, y2)) => (x1 * x2, y1 + y2))
```



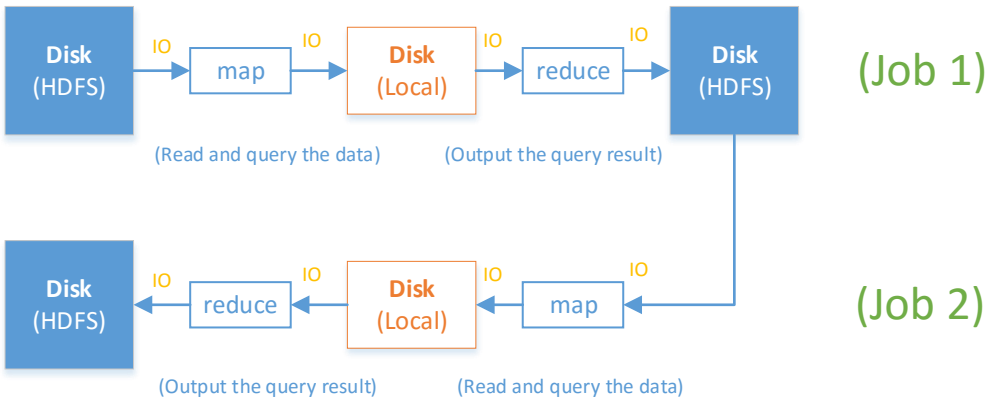

```
val result = numbers.
  map(x => (x, x)).
  take(2)
```

Benefits

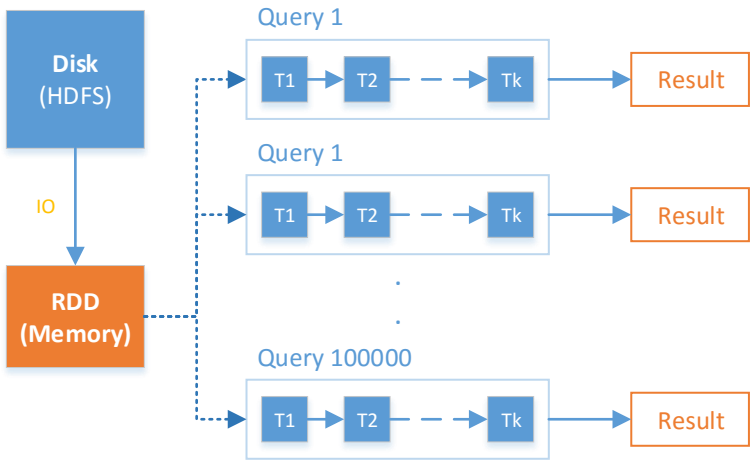
In Memory: The key benefit of the RDD is its persistence in memory within the cluster. An RDD is loaded from the disk once – or from any other third party API – when it needs to be read from the source, and no other IO operation is performed again. The key exception is when the data is spilled due to memory overflow, the RDD pipeline needs to be re-computed, there is a specific need to cache the data to more stable storage, or in some less common cases, when the data needs to be shuffled across partitions. The shuffling mechanism is similar to the mechanism in MapReduce, but Spark provides ways to avoid this so that you can maximize the time the data stays in memory.

The persistence of the RDDs staying in memory allows for iterative applications, because you can read your data thousands of times, without having to send a thousand operations to the disk. This leads to bigger practical uses such machine learning, where your machine learning algorithm might need to make a million, or perhaps a few billion, iterations. Of course, once you have your RDDs loaded into the cluster, you can share the data among multiple other applications. This in memory, cluster wide distribution of data, which you can share across nodes, is not possible in MapReduce.

MapReduce Query



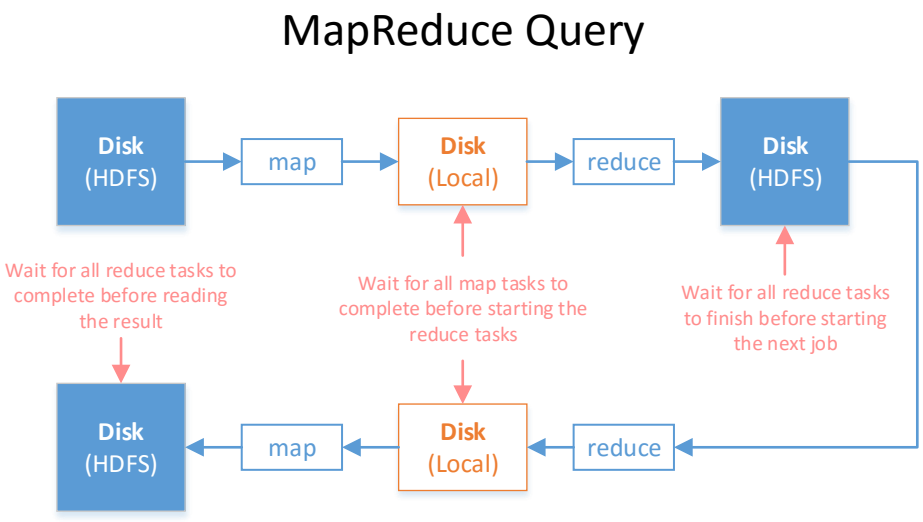
Spark Queries



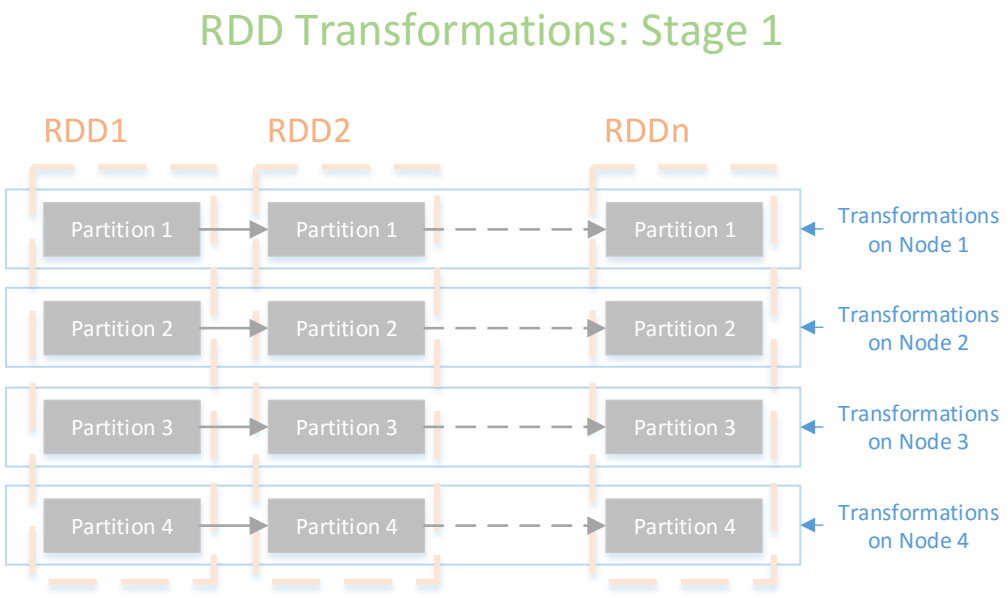
Fault Tolerance: Another key benefit of using RDDs is their ability to quickly recover from node failures. Each RDD logs its transformation in a graph. This graph is used to recover lost partitions by tracing down the transformations of their origin. Since the transformations are done in memory, re-computing each partition is quick. Back-up alternatives created to avoid the repetition of slow operations – such as the replication of partitions across multiple nodes – are avoided completely.

Because the graphs are direct and acyclic (resulting from the immutable property of the RDDs) the starting data and the sequence of transformations remains as a constant. This guarantees that the starting set of partitions will always generate the same output set. This in turn allows Spark to recover individual partitions, rather than recreating the entire RDD, knowing that the collection within the RDD will always remain the same.

Distribution: The third benefit of using RDDs is the ability for each partition to be processed in parallel across multiple nodes. An improvement provided by Spark is the ability to compute a chain of functions on a partition independently, rather than per one task. This is not possible in MapReduce, since there is a synchronization point between the map and reduce tasks when a result needs to be written to disk. This problem is visualized below:



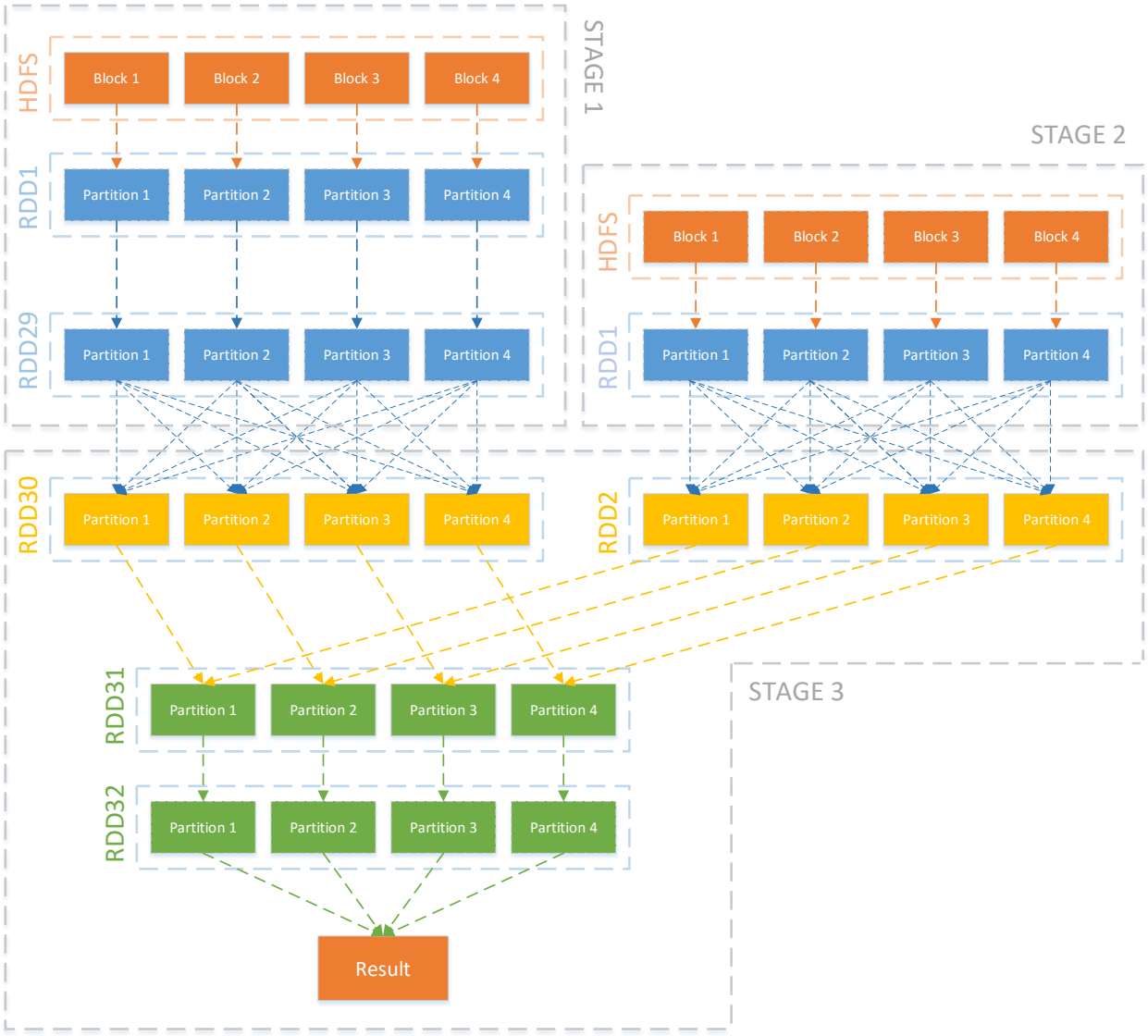
In spark, not only are the disk operation avoided, but individual partitions can undergo a pipeline of transformations independently, as long as they remain within the same stage of tasks. Staging of tasks is something that will be discussed later. For example, in the image below, transformations on Node 1 can complete without the node waiting for the other nodes to complete their transformations:



DAG

The DAG is an abbreviation for the Directed Acyclic Graph. Each node within the graph is a partition belonging to an RDD, and the edge is a transformation between two partitions. The direct and acyclic keywords are derived from the fact that RDDs are immutable. Each transformation done on an RDD does not affect the content of the existing partitions within. Each transformation can only create a new RDD with a new set of partitions. The DAG Scheduler is responsible for creating the DAG, using the information about each RDD and the transformations provided to the pipeline. It typically creates the DAG in stages, before sending it of to the task scheduler.

An example of DAG is shown on the below. Transformations requiring shuffling operations (something which will be discussed later) result in the DAG being divided into stages.


STAGE 1
STAGE 2
STAGE 3
HDFS
Block 1
Block 2
Block 3
Block 4
RDD1
Partition 1
Partition 2
Partition 3
Partition 4
RDD29
RDD30
RDD2
RDD31
RDD32
Result

Main Architecture

Driver: The starting point of the application. Responsible for launching the components that turn the code into the actual spark jobs.

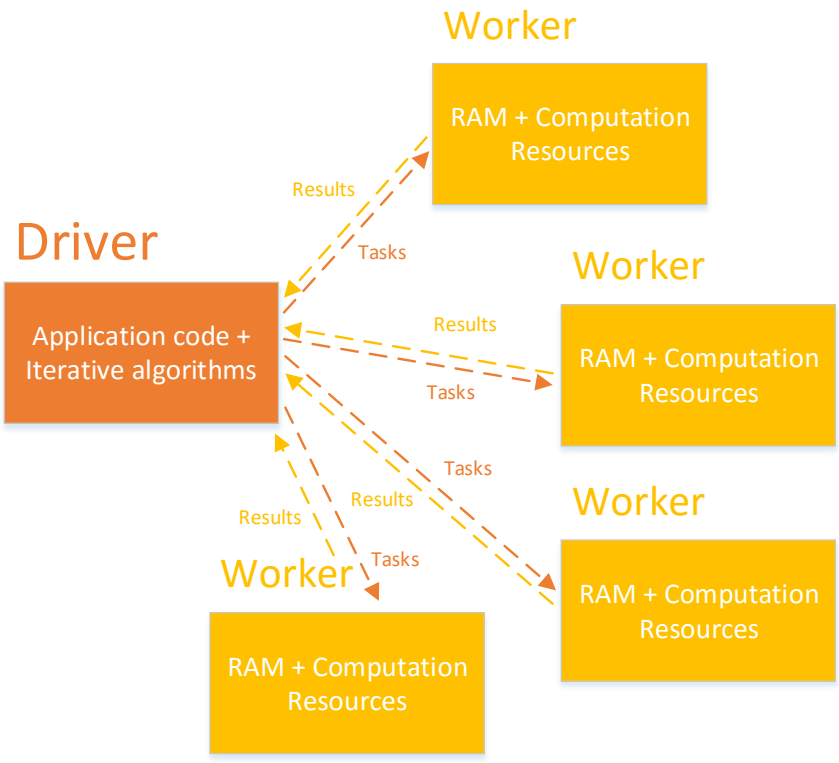
Worker: Responsible for running the transformations across the distributed data.

Stage: A set of transformations that can be executed together on partitions within the same worker.

Task: The execution of a single stage for a single partition.

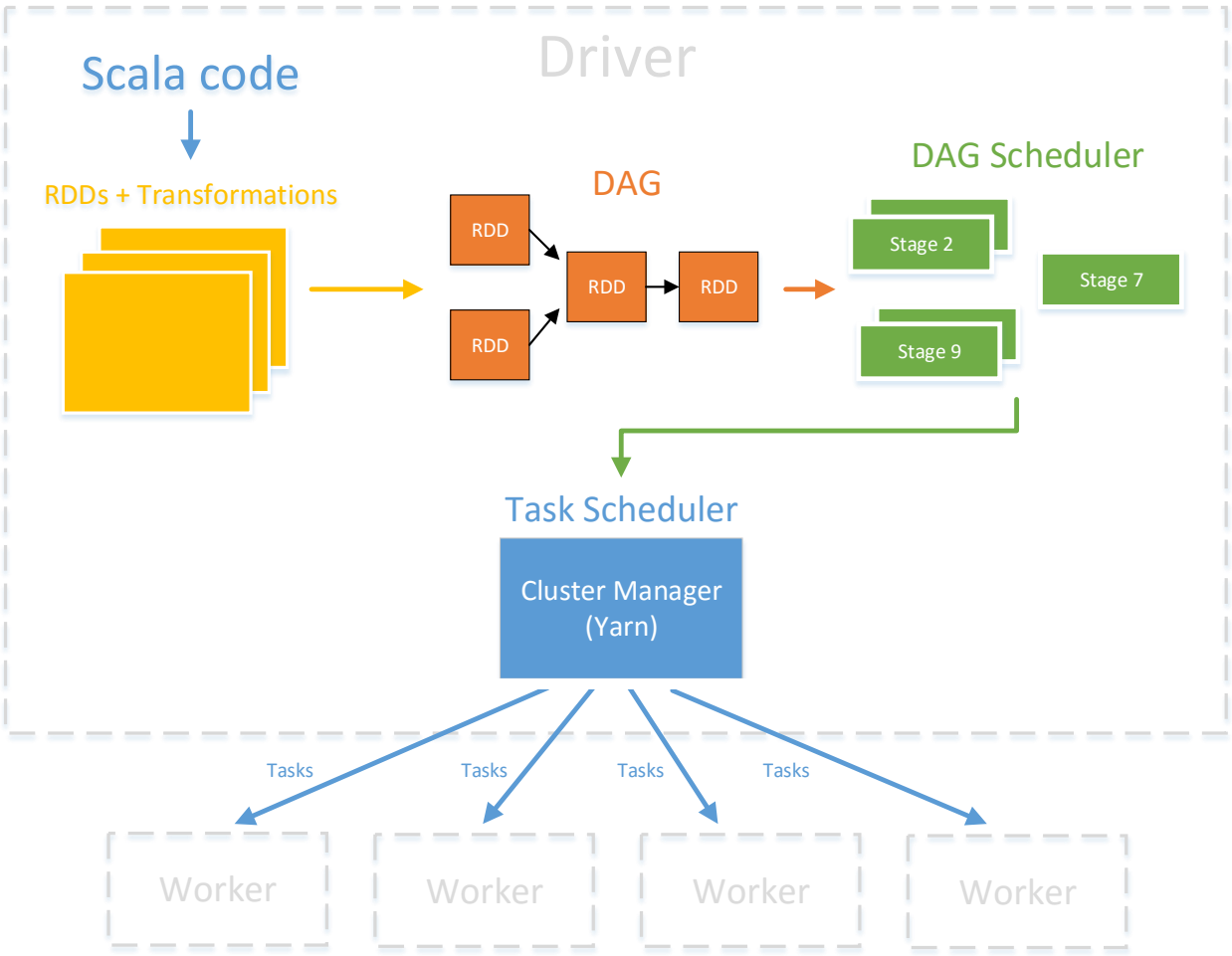
DAG Scheduler: Splits the DAG into stages of tasks.

Task Scheduler: Schedules tasks to be executed on a worker and ensures they complete.



Driver

The Driver is the main program that kicks of Spark. You would have a Driver on the master node, and any application dependent on Spark would communicate through this server. The Driver is where you first provide your code. Spark takes this code and turns it into the RDD and DAG abstractions. It then breaks everything down into stages and tasks and schedules everything on the available nodes running in your cluster – through the help of the cluster manager.



As shown above, the driver contains multiple components, such as the DAG Scheduler and the Task Scheduler. The first step within the driver is to take the RDDs and transformations – as defined by the application code provided – and to convert these into the Directed Acyclic Graph. This graph might have various dependencies between RDDs, and it is split up by the DAG scheduler into multiple stages based on the type of dependencies that exist.

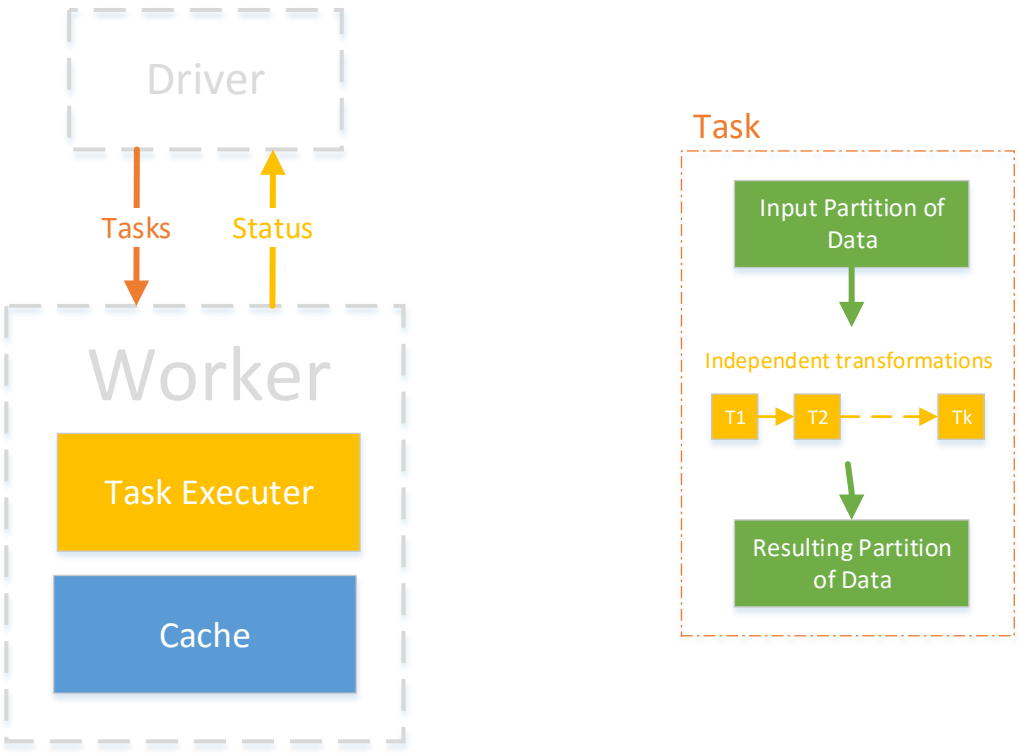
Once the driver has a set of stages, it goes to each stage and creates a sub-set of tasks that need to be executed – one task exists for every partition of data, for every RDD. These tasks are sent to the Task Manager. Its' responsibility is to communicate with the cluster manager for resources and to figure out which worker it is to schedule each task on for optimal execution. This location, by default, is based on data locality, assuming the data is properly distributed. So if a partition associated with a task has its chunk of data on one node, then that Task Manager will try to run a task in that location.

The driver then sends each task to the specific worker and monitors each process, ensuring each task completes. It will either wait for the workers to send their status of completion – after which it will start preparing for the next stage – or it will reschedule tasks if there is failure with one of the workers. Once all the stages complete and the result is computed, the driver terminates all executing processes – freeing up the allocated resources in the cluster – and exits.

Worker

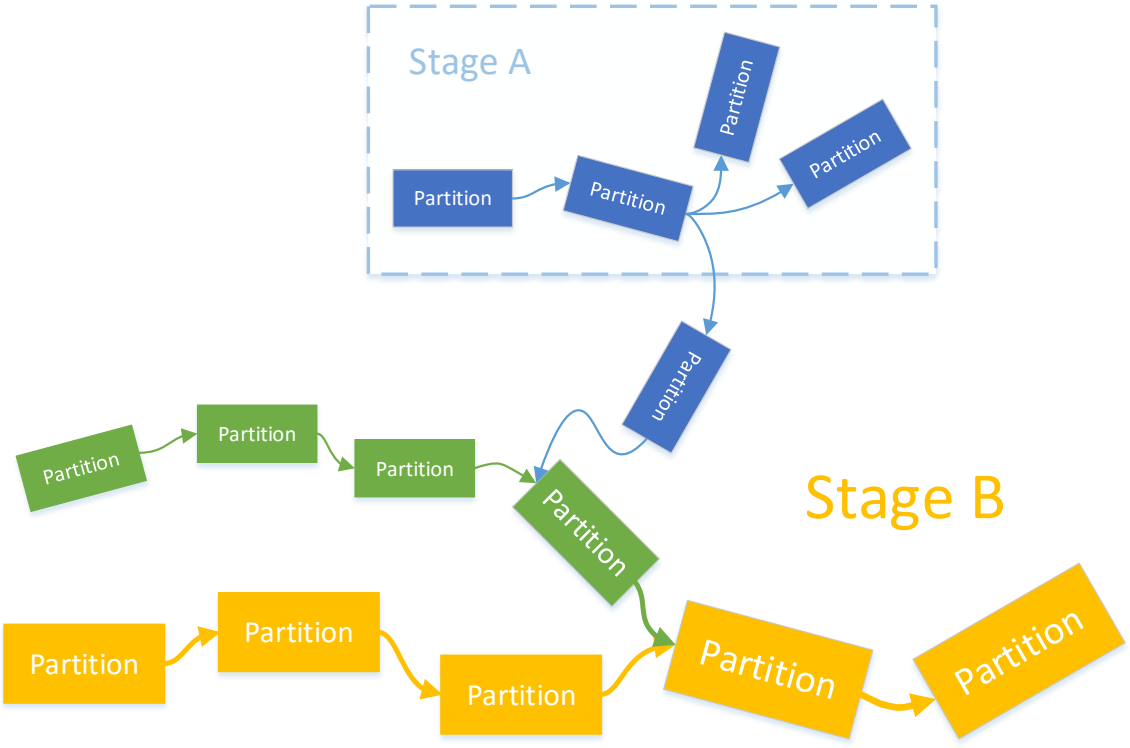
A worker is a unit responsible for the execution of tasks within a stage, on a subset of partitions instructed by the driver – one task being a set of one or more transformations running on a single partition. A worker runs independently, meaning that besides the driver who it sends the statuses to, the worker communicates to no one, including other workers. It also works only with the partitions assigned.

Unlike the driver, the worker reads and writes the actual data belonging to the RDD. It can write its output into memory, cache, the disk, or to any other source – if needed after the completion of the last stage or when performing a shuffle.



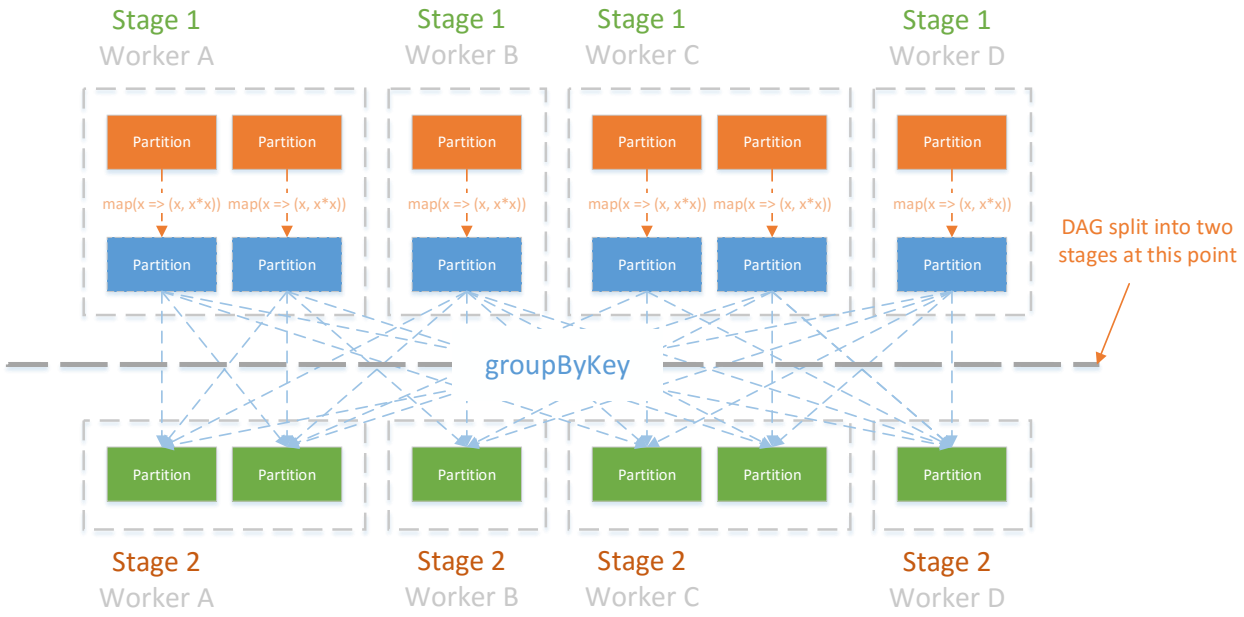
DAG Scheduler

The DAG Scheduler is responsible for breaking down the Direct Acyclic Graph into stages. A stage is a set of transformations that can be performed on one partition where the data associated remains local. During a stage, a worker can get tuples from partitions existing on other workers, but it never sends them out. Spark uses the definition of narrow and wide dependencies – which we will discuss in detail in the next secion – to determine what constitutes the escense of such a stage, but for now we can visualize it as a growing river. Within a stage, a child partition can be the converging point of multiple parent partitions, possibly from multiple workers, but a parent can only point to one child.



One example of a stage is a long sequence of mapping transformations. Each map only requires data from the previous transformed output. In order for all transformations to complete, the worker only needs to load the data once for its' assigned partitions at the starting point.

A counter example is the usage of the groupByKey. This transformation shuffles tuples across partitions between all nodes in the cluster and produces a new set of partitions with the tuples re-grouped per key. Since the tuples from a single parent might be sent to multiple children, the worker needs to receive data from other workers in the cluster and it needs to send the data out. Hence the DAG scheduler notices a more complex dependency and splits the graph at this point into two separate stages. A more detailed example of how this traffic appears in a cluster between the workers is illustrated in the next diagram:



As seen in the representation, the mapping operations only transition the partitions from one state to another. However, the groupByKey operator forces the keys from each partition to be shuffled across the different workers. The DAG scheduler splits the graph at this point into two separate stages. The accompanying code showing where this shuffling occurs is provided below:

```
val sampleB = sc.parallelize(Seq.fill(1000)(Random.nextInt(100)).toList, 20)
sampleB.partitions.length
// Int = 20

sampleB.toDebugString
// String = (50) ParallelCollectionRDD[1] at parallelize at <console>:25 []
```

```
val mapTrans = sampleB.map(x => (x, x * x))
mapTrans.partitions.length
// Int = 20

mapTrans.toDebugString
// String = (20) MapPartitionsRDD[3] at map at <console>:27 []
//   |   ParallelCollectionRDD[2] at parallelize at <console>:25 []
```

```
val groups = mapTrans.groupByKey
groups.partitions.length
// Int = 20

groups.toDebugString
// String = (20) ShuffledRDD[4] at groupByKey at <console>:29 []
//  +-(20) MapPartitionsRDD[3] at map at <console>:27 []
//    |   ParallelCollectionRDD[2] at parallelize at <console>:25 []
```

Task Scheduler

The task scheduler is responsible for sending out tasks to the workers – a task being a set of transformations belonging to one stage, all to be executed on one particular partition of an RDD. The task scheduler is used by the driver to keep track of transformations; to see if they have finished successfully on each worker, and to determine if the dependencies have been met so that the next stage within the job can begin. The task scheduler also tries to sends tasks to a specific node based on data locality.

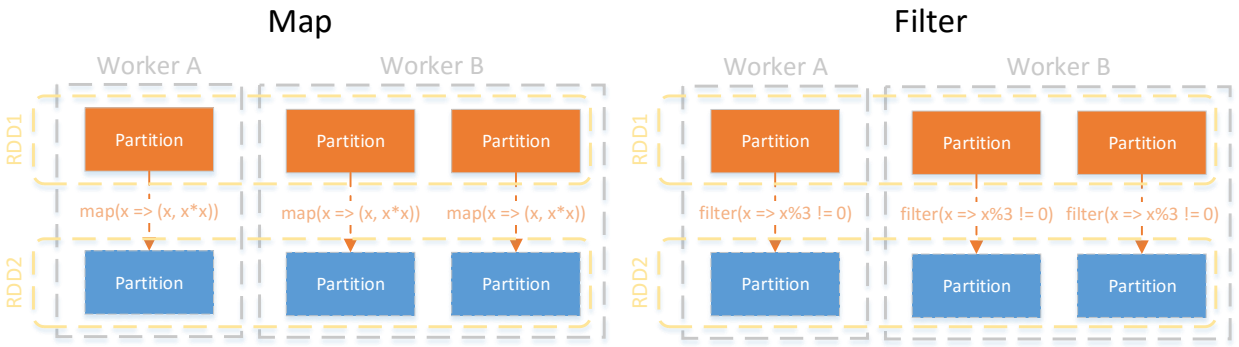
Dependency Management

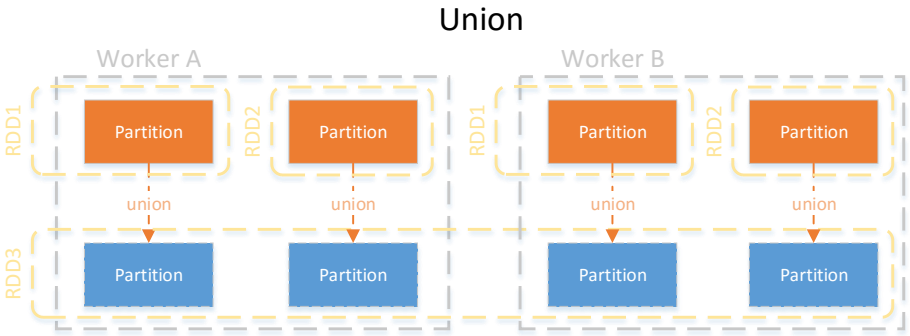
There are two types of dependencies that exist when transforming the state of an RDD . One being the narrow dependency and the second being the wide dependency. As we saw above, the DAG Scheduler will take the Directed Acyclic Graph and split it into stages wherever a transformation causes a wide dependency to occur. We will now discuss each of these dependencies separately.

Narrow Dependencies

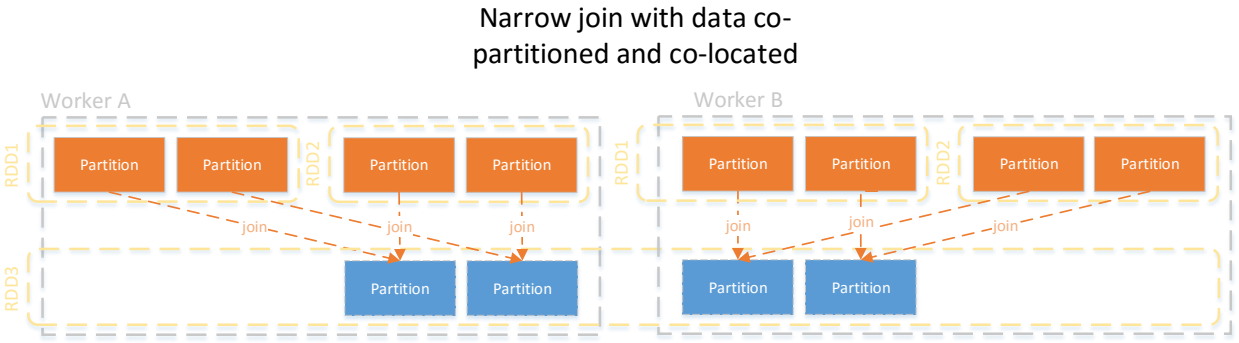
A narrow dependency is created when each partition in the parent RDD is associated with at most one partition forming the child. A child partition can have multiple parent partitions, and in most operations each parent partition will belong to a separate parent RDD as well – Coalesce being one of the exceptions.

Narrow dependencies between partitions are efficient because they often allow for transformations to be computed on the same worker. The driver creates a pipeline of transformations under one stage, and the worker loads the data once, reducing the need for traffic within the cluster. In the case of a transformation such as the join, the dependency between the RDDs is still considered narrow if all the RDDs share the same partitioner. Traffic does occur, however minimal. Each child partition only needs tuples to be sent from one parent partition for every parent RDD involved. This fact is useful during a node failure because the lost child partitions only require one partition from each parent RDD to be re-computed. Examples of transformations forming narrow dependencies are shown below:

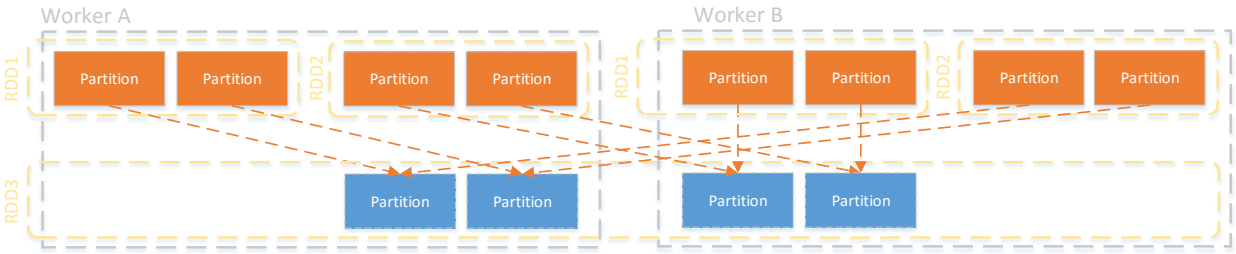




The examples above each involve a one to one dependency between the parent and child partition. However, as previously mentioned, a child partition can also be created from multiple parents. Such a narrow dependency can happen between the join of two RDDs, whose partitioner is shared.



Narrow join with data co-partitioned but not co-located



If the partitioning logic is shared between the RDDs, then during a join transformation – illustrated with the examples above – every partition within the first RDD will join with at most one partition belonging to the second RDD. Each child partition is therefore created as a result of two separate parents.

The first example also shows the benefit of having the partitions within each RDD co-located. If the partitions from each RDD are joined on the same node, then traffic is removed completely. Such co-location is possible when two RDDs are partitioned under the same job. This is not guaranteed however. Spark does not provide constraints to make partitions location specific. If two RDDs end up in separate jobs, or if nodes fail and partitions are recomputed, then the co-locality can easily be lost.

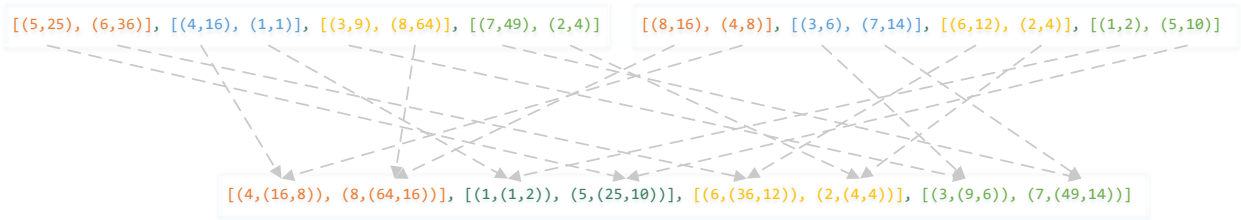
To show how such a join works more clearly, consider the code example and outputs of two RDDs joined together without the use of a shared partitioner:

```
val sampleOne = sc.parallelize(List(5, 6, 4, 1, 3, 8, 7, 2), 4).
  map(x => (x, x * x))

printPartitions(sampleOne);
// Partitions =
// [(5,25), (6,36)], [(4,16), (1,1)], [(3,9), (8,64)], [(7,49), (2,4)]
```

```
val sampleTwo = sc.parallelize(List(8, 4, 3, 7, 6, 2, 1, 5), 4).
  map(x => (x, x + x))

printPartitions(sampleTwo);
// Partitions =
// [(8,16), (4,8)], [(3,6), (7,14)], [(6,12), (2,4)], [(1,2), (5,10)]
```

```
val joinOne = sampleOne.join(sampleTwo)

printPartitions(join);
// Partitions = [(4,(16,8)), (8,(64,16))], [(1,(1,2)), (5,(25,10))],
                [(6,(36,12)), (2,(4,4))], [(3,(9,6)), (7,(49,14))]
```

As seen in each printed output for each partition – labelled by color for further visibility – tuples belonging together in sample one and sample two are no longer together in the output partitions of the joined result. This can also be visualized as follows:



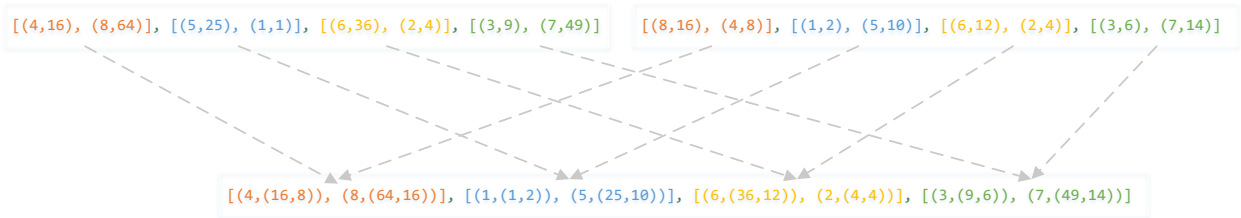
Now consider the same samples being joined together, but with shared partitioning logic applied to all the keys. The code and output for the partitioned samples and join will now look as follows:

```
val partitionedOne = sampleOne.partitionBy(new HashPartitioner(4))

printPartitions(partitionedOne);
// Partitions =
// [(4,16), (8,64)], [(5,25), (1,1)], [(6,36), (2,4)], [(3,9), (7,49)]
```

```
val partitionedTwo = sampleTwo.partitionBy(new HashPartitioner(4))

printPartitions(partitionedTwo);
// Partitions =
// [(8,16), (4,8)], [(1,2), (5,10)], [(6,12), (2,4)], [(3,6), (7,14)]
```

```
val partitionedJoin= partitionedOne.join(partitionedTwo)

printPartitions(partitionedJoin);
// Partitions =
// [(4,(16,8)), (8,(64,16))], [(1,(1,2)), (5,(25,10))],
// [(6,(36,12)), (2,(4,4))], [(3,(9,6)), (7,(49,14))]
```

The tuples in the partitioned samples now remain together inside the output of the joined RDD. Since both tuples from each parent partition are always sent to the same child, the dependency structure is also narrowed and the traffic itself is minimized – since the tuples can now be sent together. This network and dependency reduction is visualized below:



Spark will reduce the dependency and network traffic whenever possible, even if only one of the RDDs is partitioned. The CoGroupedRDD – created during a join transformation – checks how much shuffling is required. It will first check the partitioner of each parent RDD involved in the join. If all parents share the same partitioner, it will use that partitioner and the DAG scheduler will treat this dependency as narrow. Otherwise the CoGroupedRDD will pick the parent RDD with the existing partitioner, or the largest parent RDD – if multiple parent RDDs have a present but different partitioner – and use that RDD to define the new partition boundaries, forcing only the other RDDs to be shuffled. If zero partitioners exist, it will shuffle everything. This logic is seen in the source code itself:

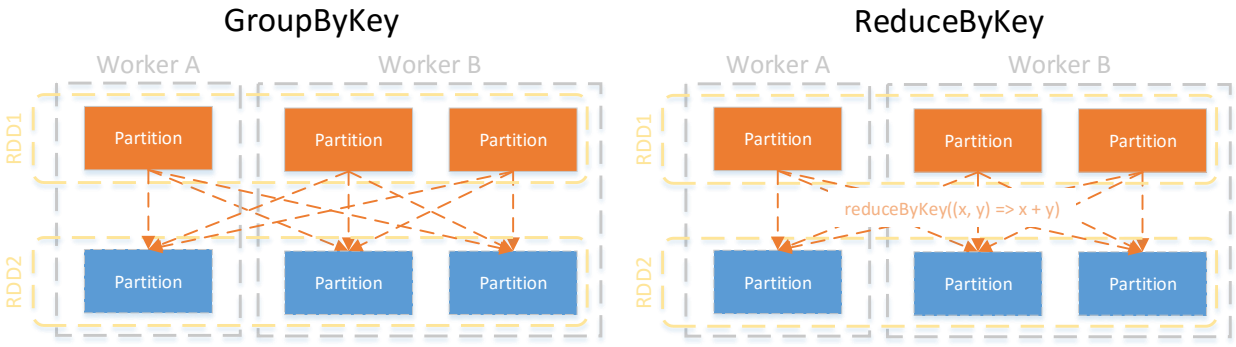
org/apache/spark/rdd/CoGroupedRDD.scala

```
. . .

 * @param rdds parent RDDs.
 * @param part partitioner used to partition the shuffle output
 */
class CoGroupedRDD[K](@transient var rdds: Seq[RDD[_ <: Product2[K, _]]], part: Partitioner)
  extends RDD[(K, Array[Iterable[_]])](rdds.head.context, Nil) {

. . .

override def getDependencies: Seq[Dependency[_]] = {
  rdds.map { rdd: RDD[_ <: Product2[K, _]] =>
    if (rdd.partitioner == Some(part)) {
      logDebug("Adding one-to-one dependency with " + rdd)
      new OneToOneDependency(rdd)
    } else {
      logDebug("Adding shuffle dependency with " + rdd)
      new ShuffleDependency[K, Any, CoGroupCombiner](rdd, part, serializer)
    }
  }
}
```

It is worth noting that even though a partitioned RDD can result in a narrow dependency during a join, and the use of a partitioner is often considered as part of an optimization strategy, calling the partitioner to re-group the tuples is a wide dependency and it can cause an expensive shuffle in itself. Therefore existing partioners on an RDD should be retainied whenever possible.

Wide Dependencies

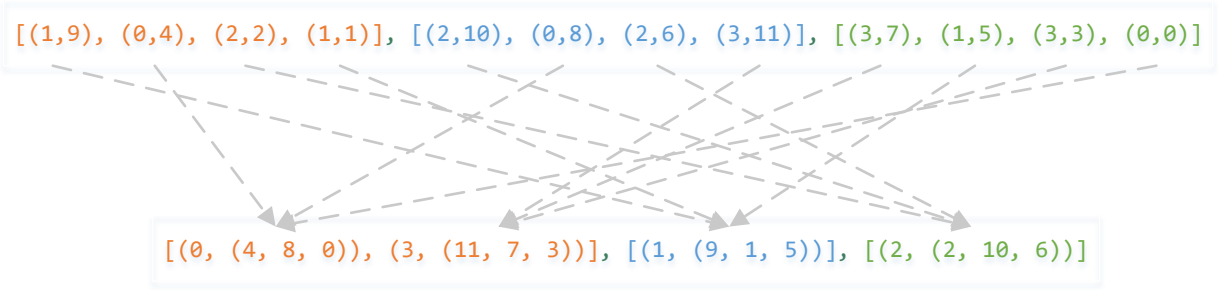
A wide dependency causes a shuffle between all partitions involved and forces the Directed Acyclic Graph to be divded into stages. Instead of a child partition being dependent on just one partition from each parent RDD – or being dependent on multiple partitions belonging to one parent but existing on the same worker, as is the case for Coalesce – the child now needs to fetch data from multiple partitions, spread across most, if not all, nodes within the cluster. The network traffic significantly increases. Examples of transformations causing wide dependencies is shown below:



As seen in these GroubByKey and ReduceByKey transformations, each partition belonging to the second RDD needs to pull data from every partition belonging to the first one. Data also needs to be moved between the workers A and B, thus creating more traffic. The occurrence of this type of shuffle can also be confirmed by running the following code:

```
val sampleOne = sc.parallelize(List(9, 4, 2, 1,10, 8, 6,11, 7, 5, 3, 0), 3).
  map(x => (x % 4, x))

printPartitions(sampleOne);
// Partitions =
// [(1,9), (0,4), (2,2), (1,1)],
// [(2,10), (0,8), (2,6), (3,11)],
// [(3,7), (1,5), (3,3), (0,0)]
```

```
val groupByKey = sampleOne.groupByKey

printPartitions(groupByKey);
// Partitions =
// [(0, (4, 8, 0)), (3, (11, 7, 3))],
// [(1, (9, 1, 5))],
// [(2, (2, 10, 6))]
```



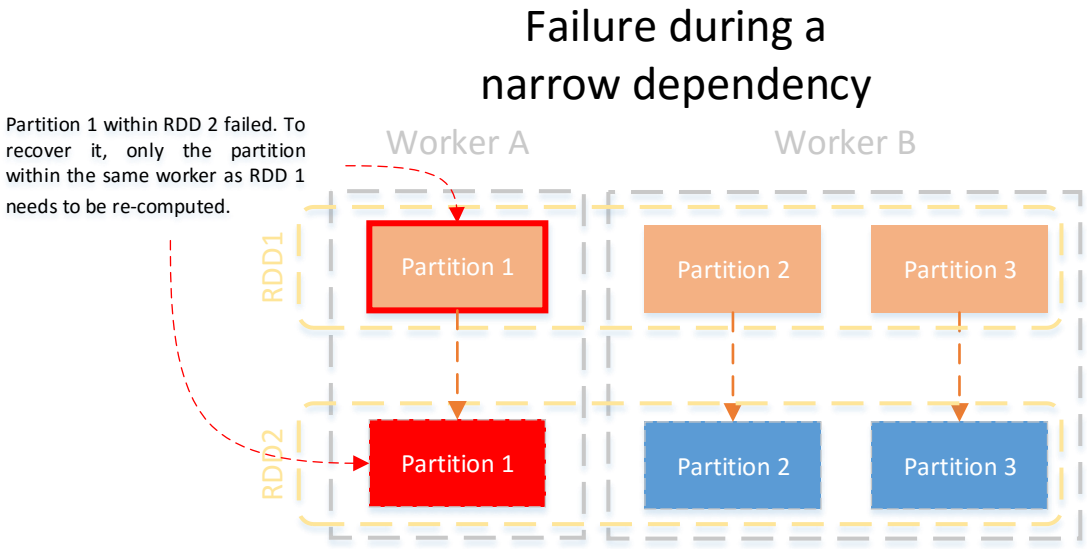
In the above example, we first generate a list of twelve numbers into four partitions – each partition labelled by color for better visibility. We then use the groupByKey operator to group the tuples into three different groups. A lot of the tuples are shuffled as a result.

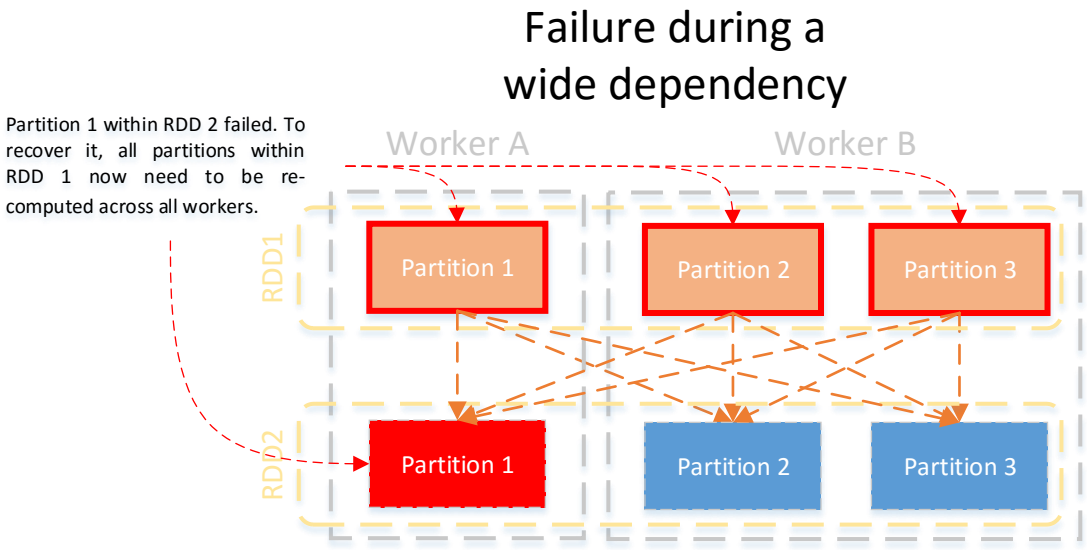
We can also confirm whether or not a transformation causes a shuffle by calling the toDebugString function. For most operations directly dependent on a shuffle, the ShuffledRDD will be created. An example of how toDebugString would look like for groupByKey and reduceByKey is shown below:

Shuffling occurs

```
val sampleE = sc.parallelize(Seq.fill(1000)(Random.nextInt(100)).toList, 20)
val groupByKey = sampleE.map(x => (x, x * x)).groupByKey
groupByKey.toDebugString
// String = (20) ShuffledRDD[2] at groupByKey at <console>:29 []
//  +-(20) MapPartitionsRDD[1] at map at <console>:27 []
//     |   ParallelCollectionRDD[0] at parallelize at <console>:25 []

val sampleF = sc.parallelize(Seq.fill(1000)(Random.nextInt(100)).toList, 20)
val reduceByKey = sampleF.map(x => (x, x * x)).reduceByKey((a, b) => a + b)
reduceByKey.toDebugString
// String = (20) ShuffledRDD[2] at reduceByKey at <console>:29 []
//  +-(20) MapPartitionsRDD[1] at map at <console>:27 []
//     |   ParallelCollectionRDD[0] at parallelize at <console>:25 []
```

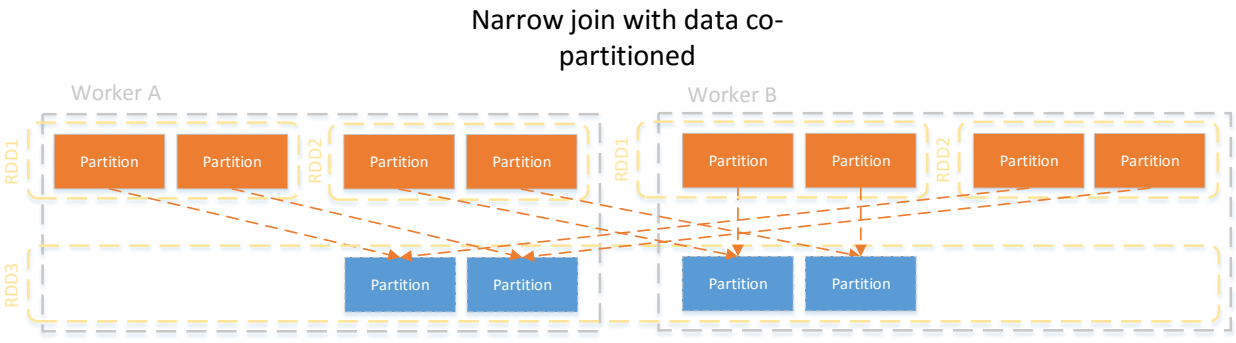
Wide dependencies make recoveries from node failures more expensive. If one node dies containing only a small subset of partitions belonging to an RDD, all the partitions belonging to the parent RDD need to be re-computed – since each lost child is depedent on everything. To see why a wide dependency is more expensive to recover than one which is narrow, consider the visualization below, comparing the two:



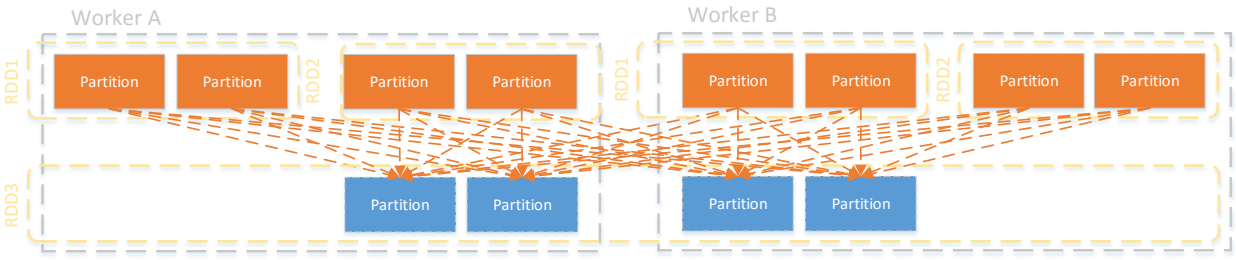


Because node failures can happen frequently, the transformations creating a wide dependency belong to an exceptional case when Spark writes to the disk. During a failure, Spark will re-use a set of **shuffle files** written at the start of the stage, whenever possible, to avoid the expensive re-computation.

Spark also performs multiple checks to avoid shuffling when possible. One such example is the use of the CoPartitionedRDD, which looks for existing partitioners when joining two or more parent RDDs together. Consider first the example of a co-partitioned join – shown previously – compared to a join between RDDs not using a partitioner at all:



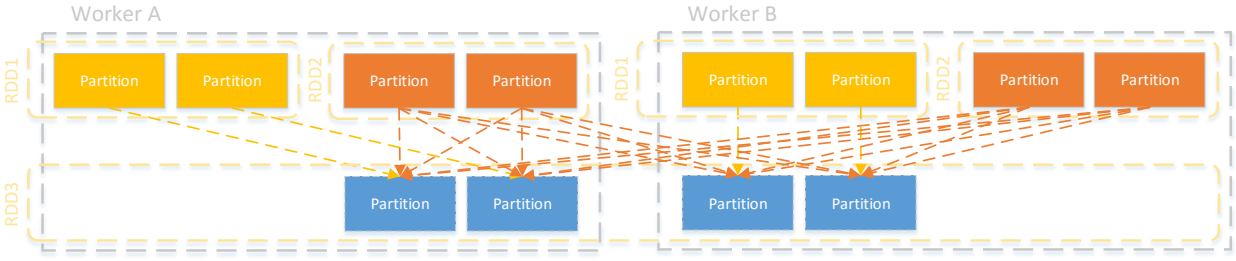
Wide join with data not partitioned



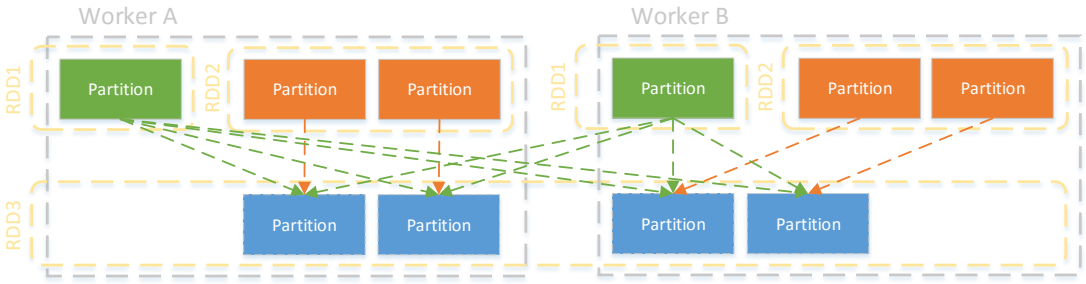
Despite both transformations involving a join, the first example – a narrow join using a shared partitioner – results in a lot less data being moved between the workers. This is because Spark is smart enough to know that both RDDs use the same partitioner, and can therefore map the two parent partitions, one from each input RDD, to the same location for the output child.

But the partitioners don't have to be equivalent in order for optimization to take place. Even if only one RDD has an associated partitioner, Spark will use that partitioner for the constructing the partitions for the resulting RDD, and only the second RDD being joined will be shuffled. If both RDDs have a partitioner, but different ones, Spark will re-use the partitioner belonging to the largest RDD so that shuffling is applied to the smaller dataset. To see this better, consider the visual example for these two cases below:

Wide join with only RDD1 partitioned

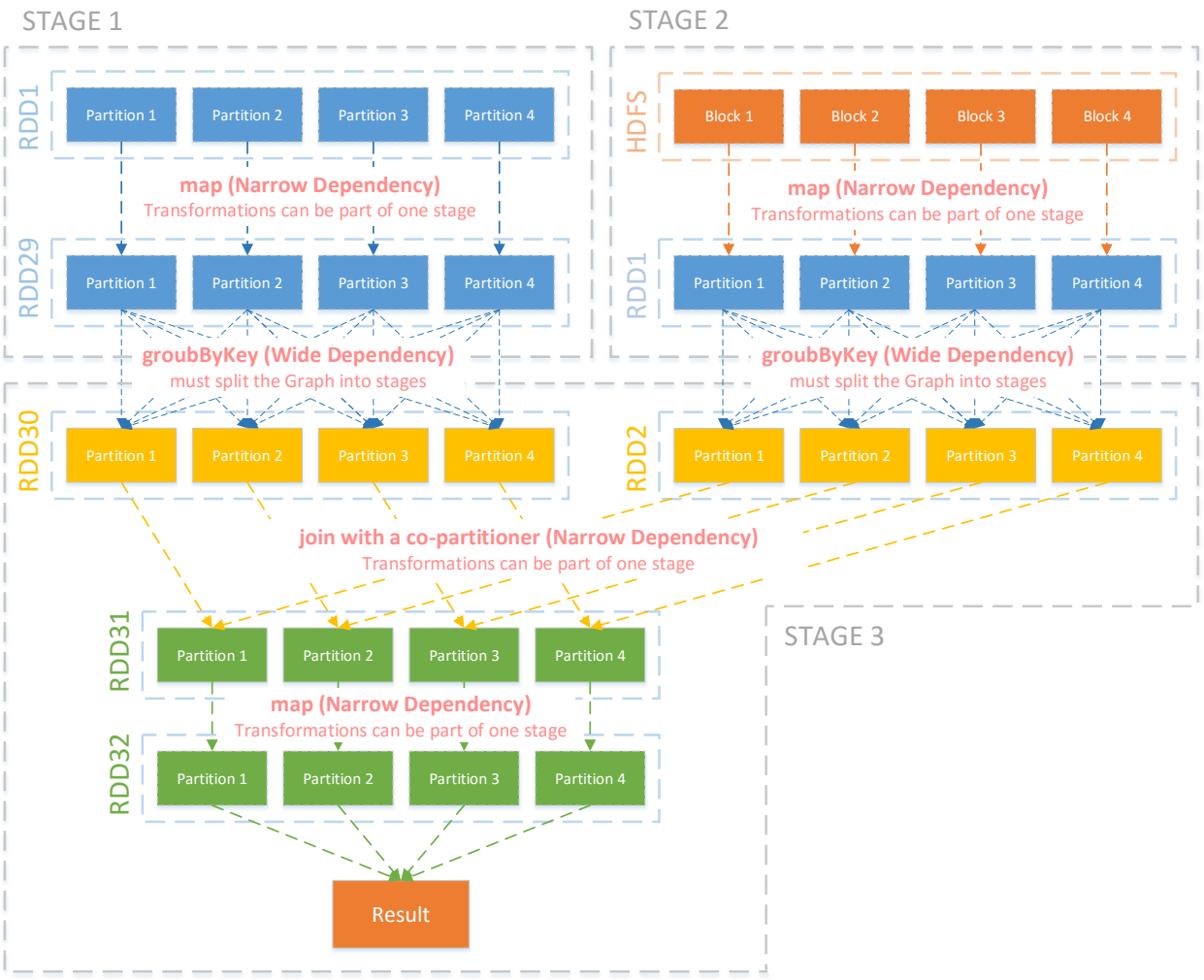


Wide join with RDDs having different
partitioners, with RDD 1 smaller than RDD 2



In the first example, only the tuples belonging to RDD number two are shuffled. In the second example, shuffling occurs only for RDD number one – the smallest RDD of the two sets. In each case, the traffic created is minor in comparison to the wide join displayed where the data was not partitioned at all.

With a better understanding of how dependencies work, we can now go back to the previous image of the Directed Acyclic Graph and see where and why the DAG scheduler creates the stages:


STAGE 1
STAGE 2
STAGE 3
RDD1
Partition 1
Partition 2
Partition 3
Partition 4
map (Narrow Dependency)
Transformations can be part of one stage
RDD29
Partition 1
Partition 2
Partition 3
Partition 4
groubByKey (Wide Dependency)
must split the Graph into stages
RDD30
Partition 1
Partition 2
Partition 3
Partition 4
HDFS
Block 1
Block 2
Block 3
Block 4
map (Narrow Dependency)
Transformations can be part of one stage
RDD1
Partition 1
Partition 2
Partition 3
Partition 4
groubByKey (Wide Dependency)
must split the Graph into stages
RDD2
Partition 1
Partition 2
Partition 3
Partition 4
join with a co-partitioner (Narrow Dependency)
Transformations can be part of one stage
RDD31
Partition 1
Partition 2
Partition 3
Partition 4
map (Narrow Dependency)
Transformations can be part of one stage
RDD32
Partition 1
Partition 2
Partition 3
Partition 4
Result

Partitioning

A partition is the most basic unit for distributing data. It stores a sub-collection of the dataset and ensures that the sub-collection is stored entirely on that same node. Every RDD is first divided into these partitions and Spark then processes them in parallel across multiple nodes. This increases time performance and allows us to utilize entire cloud servers. However, as we also saw, the flow of data between the partitions can cause a lot of network traffic between the workers. This is especially true for transformations that move data around for ordering and where the data is scattered across all nodes.

By default, when loading data from a source like HDFS, Spark creates one partition for every HDFS file. However this does not guarantee that the data is loaded with the neighbouring tuples having the desired relationship. As a result, Spark provides us with different preferences to partition our data wisely, so that our next subsequent partitioning schemes become easier on our resources. A common option is deciding how the data should be partitioned – whether by hash values or by range boundaries.

Hash Partitioning

The Hash Partitioner is the default partitioner used by Spark when a shuffle occurs between two or more RDDs. It works by assigning each RDD tuple to a partition number equal to the hash value of the associated key, modulo the total number of partitions involved:

$$\textbf{partition number for } (\text{key}_n, \text{value}_n) = \text{key}_n\textbf{.hashValue \% number of partitions}$$

Unless the **spark.default.parallelism** configuration is set, during a shuffling operation, Spark will also call the Hash Partitioner with a default number of partitions, set to the number of partitions belonging a the largest parent RDD. This default behavior is seen in the source code of the partitioner directly:

org/apache/spark/Partitioner.scala

```
. . .

def defaultPartitioner(rdd: RDD[_], others: RDD[_]*): Partitioner = {
  val rdds = (Seq(rdd) ++ others)
  val hasPartitioner = rdds.filter(_.partitioner.exists(_.numPartitions > 0))
  if (hasPartitioner.nonEmpty) {
    hasPartitioner.maxBy(_.partitions.length).partitioner.get
  } else {
    if (rdd.context.conf.contains("spark.default.parallelism")) {
      new HashPartitioner(rdd.context.defaultParallelism)
    } else {
      new HashPartitioner(rdds.map(_.partitions.length).max)
    }
  }
}
```

The Hash Partitioner works really well when there is no prior knowledge to how the data is structured. Consider the result of the Hash Partitioner below, with number of partitions specified to four:

```
val food = sc.parallelize(List(
  "asparagus", "coconut", "coffee", "grapes", "lime",
  "meat", "pizza", "steak", "yogurt", "zucchini" )).
  map(x => (x, 1)).
  partitionBy(new HashPartitioner(4));

printPartitions(food);
// Partitions =
// [coffee, grapes, pizza], [coconut, lime], [steak, yogurt], [asparagus, meat, zucchini]
```

The above example has a list of food items chosen at random. Despite no complex logic specified, each partition already contains a good distribution of the words given. This distribution can be expected for a much larger collection as well. Unless the words are generated in a specific manner, computed hash values are very random, creating a high probability that each partition receives an even proportion of tuples.

Range Partitioning

Hash partitioning is advantageous when you have no knowledge about the structure of your data. Assume on the other hand that the data you have is sorted, even if just partially. Then it might be beneficial to keep this order consistent for further processing, whenever an ordered relationship between the keys is needed. As the usage of the Hash Partitioner shows however, the order of the output keys isn't maintained.

The Range Partitioner solves this issue by grouping keys together that happen to fall in the same range. It first starts of by estimating the lowest and highest limits, using samples from the RDD. Using these limits, it then creates the boundary index for each partition. To see this behavior, consider the example below:

```
val food = sc.parallelize(List(
  "asparagus", "coconut", "coffee", "grapes", "lime",
  "meat", "pizza", "steak", "yogurt", "zucchini" )).
  map(x => (x, 1));

val rangePartitioner = new RangePartitioner(4, food);

val foodPartitioned = food.partitionBy(rangePartitioner);

printPartitions(food);
// Partitions =
// [asparagus, coconut, coffee], [grapes, lime], [meat, pizza, steak], [yogurt, zucchini]
```

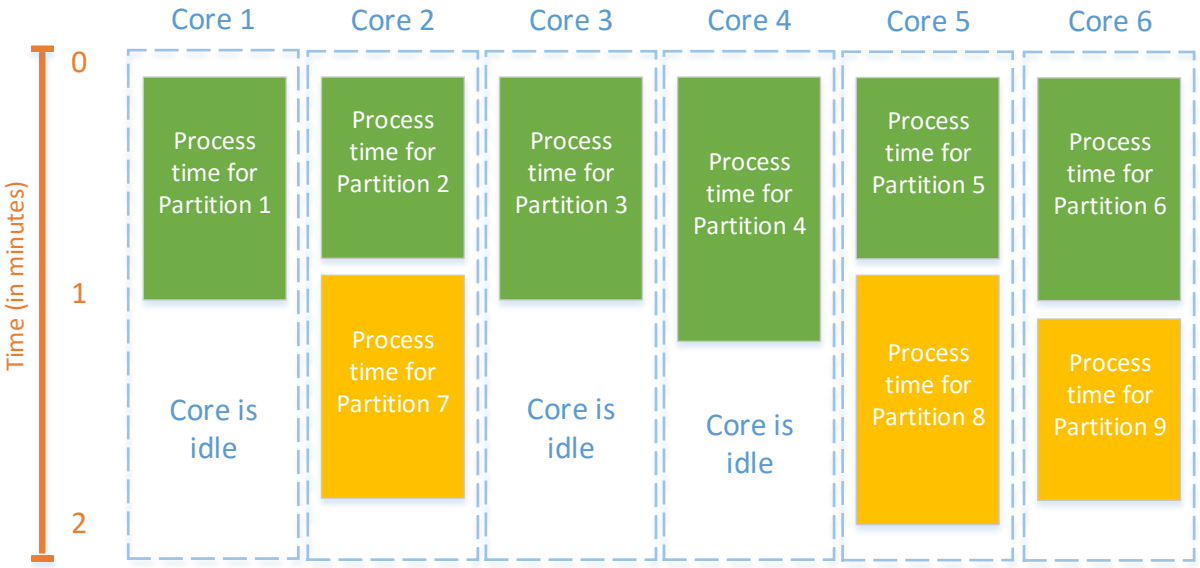
Order is maintained

In this example, the input to the Range Partitioner is the sorted list of food items. As shown in the output, by printing the content of each partition, the order on the list of food items is now retained.

The Range Partitioner also serves as a less expensive alternative to sorting data across an entire RDD. One would first use the partitioner to split the data across the defined boundaries. A call such as mapPartitions would then be used to sort the tuples per partition. Furthermore, if the data already maintains an order – but its' order is not thorough due to inconsistencies in the source producing it – then the amount of sorting done per partition can be cheaper.

Configuring parallelism

Spark gives you the option to specify the amount of partitions to create using the **spark.default.parallelism** setting. By defult, Spark sets this to the number of cores available on all the workers. The reasoning behind this is efficiency. Each core can only process one partition at a time. At the end of a stage, the remaining partitions that still need to be processed, will indirectly cause other cores and workers to become idle. To see this, consider the distribution below where you have six cores and nine partitions:



In the illustration above, six cores are shown processing the first six partitions. Once these partitions are finished – in a time frame of about one minute – the remaining cores start processing the last three. Three of the cores then become idle for almost another extra minute. If the result depends on all partitions to be ready, waiting for it will now take more time, because of these idle resources. Therefore the partition number should at least be chosen as a multiple of the cores available.

At times it is necessary to divide the dataset into smaller partitions. If the individual transformations are expensive, then this can help save memory. Smaller partitions can also benefit from increased parallelism – if each transformation done on the RDD is short and involves a quick and simple operation such as a map. But partitions cause the data to become more distributed – as is expected – and this can exponentially increase the cost of shuffling. Therefore operators such as groupByKey work better when the number of partitions remains smaller.

Repartitioning

Having gone through the benefit of using different partitioners, we will now go through the three main mechanisms most used for repartitioning the RDD. These are: **partitionBy**, **coalesce**, and **repartition**.

PartitionBy

PartitionBy is used when we plan to partition our RDD using logic from a specific partitioner. To apply partitioners such as the HashPartitioner and the RangePartitioner, partitionBy would be the function called on the RDD, with the partioner type specified as input. Partitioners are useful when we plan to keep related keys together and when we plan to optimize operations such as a join.

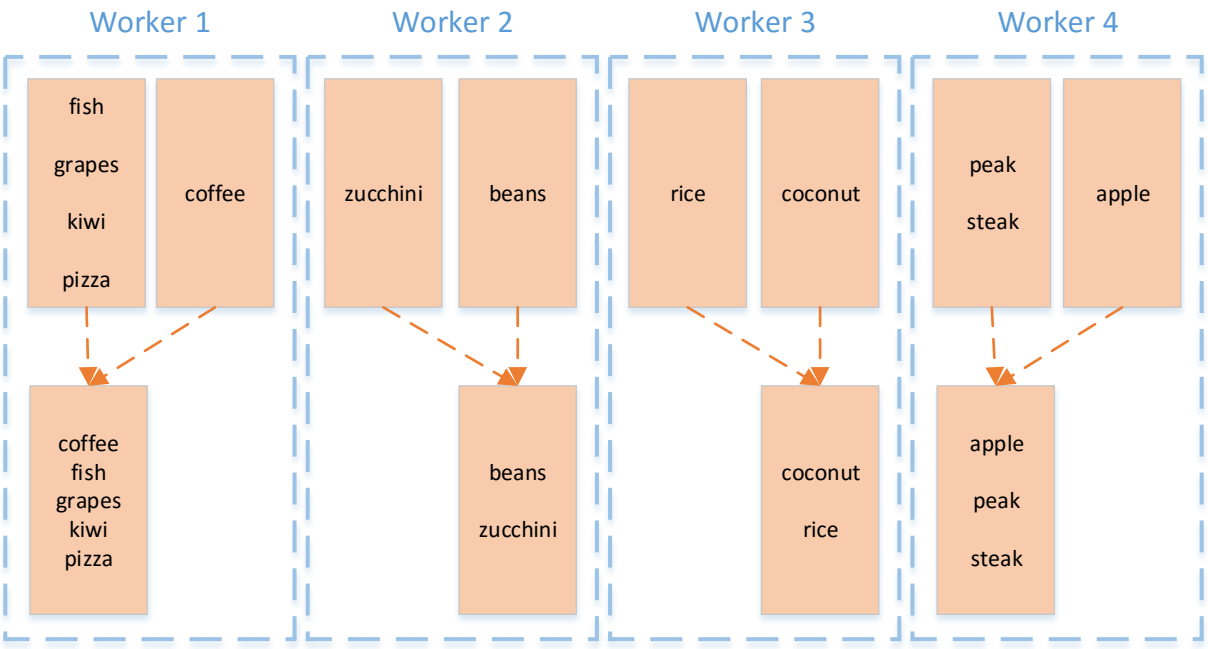
Coalesce

The Coalesce operator is used for reducing the number of partitions for an RDD. Its' input parameter is the number of partitions it is to reduce to, and a flag specifying the option for shuffling. Unlike the partitionBy mechanism, Coalesce does not shuffle the tuples by default. When multiple parent partitions exist on a worker, Coalesce will combine them locally to avoid the network traffic. The exception is when Coalese is called with a reduction number set to be less than the number of workers available in the cluster. In such a case it must pull data from other workers to combine the remaining partitions. To see the basic use case of this call, consider the following example:

```
val foodBasketOne = sc.parallelize(List(
    "apple", "beans", "coconut", "coffee", "fish", "grapes",
    "kiwi", "pear", "pizza", "rice", "steak", "zucchini" )).
  map(x => (x, 1)).
  partitionBy(new HashPartitioner(8));

printPartitions(foodBasketOne);
// Partitions =
// [fish,grapes,kiwi,pizza],[rice],[apple],[beans],[coffee],[coconut],[pear,steak],[zucchini]
```

```
val foodBasketTwo = foodBasketOne.coalesce(4);

printPartitions(foodBasketTwo);
// Partitions =
// [coffee,fish, grapes, kiwi, pizza],[coconut, rice], [apple, pear, steak],[beans, zucchini]
```

In this example, we first create a list of food items, which we transform into an RDD and group into eight partitions. The output of each partition is printed, labelled by color for better distinction. We then create a second RDD by calling Coalesce on the first set, with the number of partitions specified to four. As seen in the next output, the number of partitions printed has indeed been reduced. A visual representation of how such an operation would take place on a cluster of four workers is shown below:

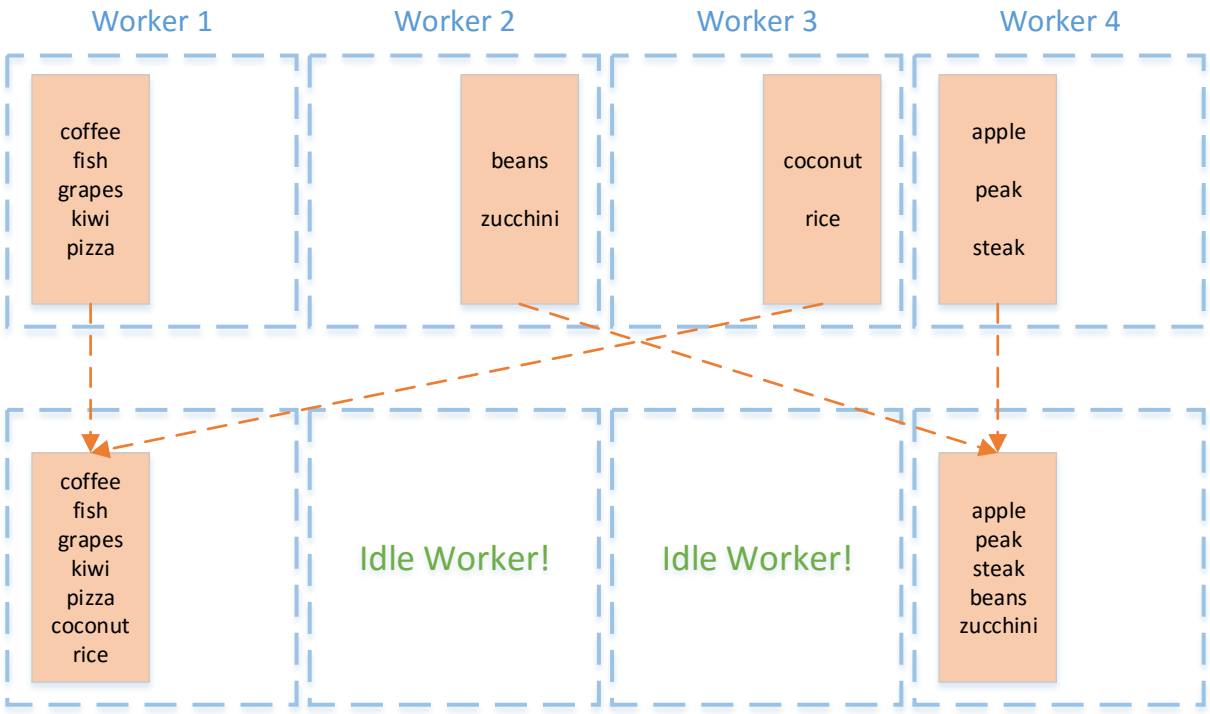


The eight partitions are first split amongst the four workers – two partitions exist on each. After calling Coalesce, Spark combines the partitions on every worker into one, creating the four partitions, as required. Each combination was local – the workers did not have to communicate.

One thing to notice in the visualization is how the keys are distributed after the step. Due to its quick execution, Coalesce does not bother re-distributing the data. If the original partitions are not evenly distributed, then combining them together might further increase the variance. For a very large datasets with multiple transformations, it might be ideal in some case to shuffle the data completely – especially if done at the start of a pipeline – since the value of running parallel computations on a perfect distribution will quickly outweigh the re-grouping costs.

Next we will see the consequence of what happens when we reduce the number of partitions to be less than the workers available. In the following example, we call Coalesce to merge the remaining four partitions into two:

```
val foodBasketThree = foodBasketTwo.coalesce(2);

printPartitions(foodBasketThree);
// Partitions =
// [coffee, fish, grapes, kiwi, pizza, coconut, rice], [apple, pear, steak, beans, zucchini]
```



As illustrated above, reducing the number of partitions to just two suddenly caused data to be transferred between the workers, making the overall operation more expensive. Two of the workers also became idle – this reduces the parallelism of Spark. Working with such an RDD afterwards heavily impacts performance. Knowing when and how to call Coalesce should therefore be done with caution.

Using Coalesce can increase both the discrepancy between the size of the partitions and it can result in idle workers. But when can it be used efficiently?

One use case is when we have a heavily partitioned RDD and we need to perform an operation that involves a lot of shuffling. Transformation such as groupByKey might end up creating millions of intermediate results, especially if the shuffling mechanics are not configured properly. Reducing the partition number would therefore help with the performance.

Another use case is when we plan to take a small sample from a large dataset. To see the reason behind this, consider a dataset of a million rows existing in a large cluster in Hadoop, spread across a hundred HDFS files. By default, Spark will load this data into memory as an RDD with a hundred partitions – one partition per file. The requirement now is to pick a random sample of one hundred rows. Based on the mechanism of Sample, Spark will compute this by taking an individual sample from each partition, rather than from the RDD as a whole. What we get is an RDD containing the sample, split into one hundred partitions. An example of this can be simulated using the code below:

```
val sample = sc.parallelize(Seq.fill(1000000)(Random.nextInt(100)).toList, 100).
  map(x => (x, 1)).sample(true, 0.0001);

// Print the number of elements in each partition.
var sizePerPartition = sample.
  mapPartitions(x => { List(x.toList.size).iterator }, true).collect
// Output:
// Array(1, 1, 2, 2, 2, 0, 1, 1, 0, 1, 0, 1, 3, 0, 0, 2, 0, 1, 1, 0, 2, 1, 0, 0, 1, 1,
// 0, 0, 4, 2, 2, 3, 2, 0, 1, 1, 1, 0, 1, 1, 2, 0, 2, 0, 1, 0, 1, 1, 0, 3, 1, 0, 1, 1,
// 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 2, 1, 2, 1, 0, 2, 0, 2, 0, 0, 1, 0,
// 2, 1, 0, 1, 2, 1, 3, 0, 0, 0, 0, 1, 1, 3, 0, 2, 2, 1)

// Produces a hundred separate files.
sample.saveAsTextFile("./output")
```

In the example above, we first generate an RDD with a million random numbers, split across a hundred partitions. We then fetch a sample, creating a new RDD with just that subset. However, when we print the size of each partition making up our sample, we get a hundred numbers, most of them being just ones and zeros. This is both silly and problematic. First of all, the number zero indicates that a partition has no tuples – but why would we want such an empty partitions in the first place? Second, why produce a hundred partitions at all? When calling saveAsTextFile, we generate a hundred files, one for each partition created. Coalese fixes these issues by allowing the sample to be grouped into a single partition:

```
val sample = sc.parallelize(Seq.fill(1000000)(Random.nextInt(100)).toList, 100).
  map(x => (x, 1)).sample(true, 0.0001).coalesce(1);
```

```
// Print the number of elements in each partition.
val sizePerPartition = sample.
  mapPartitions(x => { List(x.toList.size).iterator }, true).collect
// Output: Array(97)

// Produces one file.
sample.saveAsTextFile("./output")
```

In the above example, after calling Coalesce on the sample data, we end up with one partition containing all ninety-seven elements. The call to saveAsTextFile also produces a single file, instead of the hundred files created before.

Unless specified with a shuffling option set to true, one thing Coalesce cannot be used for is increasing the number of partitions. In code example below, we attempt to use Coalesce to increase the number of partitions to eight, for an RDD that we first distributed into four partitions using the HashPartitioner. As seen in the output however, the number of partitions still remains at four.

```
val food = sc.parallelize(List(
  "apple", "beans", "coconut", "coffee", "fish", "grapes",
  "kiwi", "pear", "pizza", "rice", "steak", "zucchini" )).
  map(x => (x, 1)).
  partitionBy(new HashPartitioner(4)).coalesce(8).

printPartitions(food);
// Partitions =
// [coffee, fish,grapes,kiwi,pizza], [coconut, rice], [apple, pear, steak], [beans, zucchini]
```

Repartition

Repartition does what the name implies. It repartitions the data within an RDD, creating a new RDD with the tuples re-grouped. Unlike Coalese, Repartition can increase the number of partitions present – in fact, it calls Coalesce directly, but with the Shuffle flag always set to true.

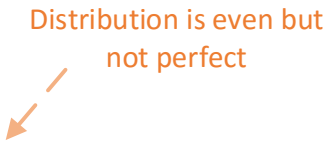
org/apache/spark/rdd/RDD.scala

```
. . .

 * If you are decreasing the number of partitions in this RDD, consider using `coalesce`,
 * which can avoid performing a shuffle.
 */
def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T] = withScope {
  coalesce(numPartitions, shuffle = true)
}
```

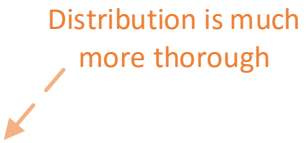
The shuffling makes the Repartition call more expensive. In general however, this allows for the distribution to become more even, and in practice, the subsequent transformations running on the evenly distributed partitions make the overall job cheaper – outweighing the starting cost.

This efficiency of the distribution can be verified by simulating a hash partitioned RDD with a million elements and by seeing side by side how these elements are re-grouped using Coalesce and Repartition.

```
val sampleOne = sc.parallelize(Seq.fill(1000000)(Random.nextInt(100)).toList).
  partitionBy(new HashPartitioner(10))

// Re-group using Coalesce.
val distributionOne = sampleOne.map(x => (x, 1)).coalesce(5);

// Print the size of each partition.
val sizePerPartition = distributionOne.
  mapPartitions(x => {
    List(x.toList.size).iterator
  }, true).collect
// Output: Array(199860, 200347, 199350, 200432, 200011)
```




```
val sampleTwo = sc.parallelize(Seq.fill(1000000)(Random.nextInt(100)).toList).
  partitionBy(new HashPartitioner(10))

// Re-group using Coalesce.
val distributionTwo = sampleTwo.map(x => (x, 1)).repartition(5);

// Print the size of each partition.
val sizePerPartition = distributionTwo.
  mapPartitions(x => {
    List(x.toList.size).iterator
  }, true).collect
// Output: Array(20000, 20000, 20000, 200001, 19999)
```



In the example above, we generate a million random numbers, once for sample one, and for sample two. Each sample has ten partitions, partitioned using the HashPartitioner. We then re-group the elements within each sample again to generate a new distribution using Coalesce, and a new distribution using Repartition. The sizes of the partitions belonging to each new distribution are then printed . As expected, the proportions are more equivalent in the second output.

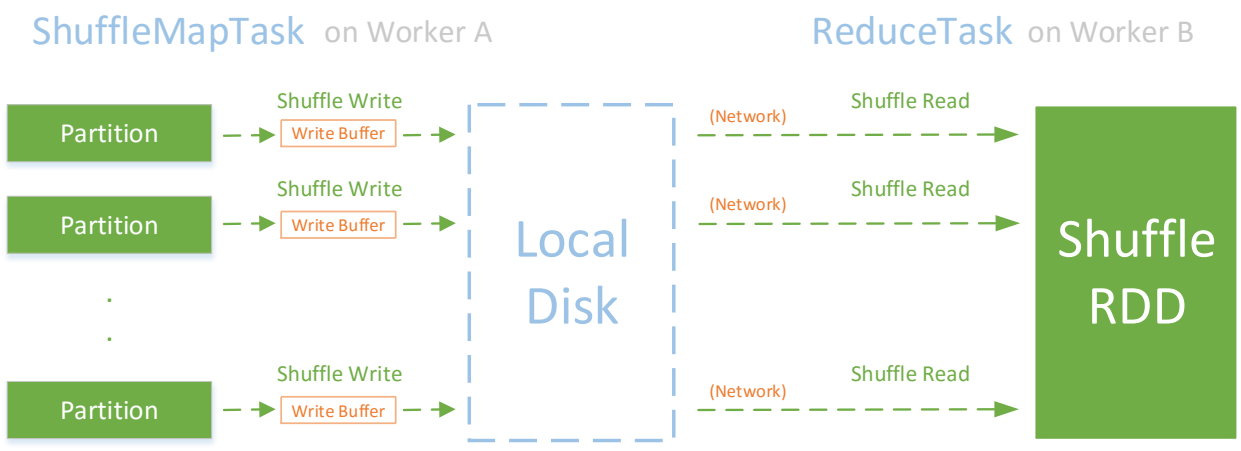
It is also worth noting that the distribution logic only works for a sufficiently large dataset, hence the choice of one million picked as the number in this example. For a data set with a smaller size, say, such as the number twenty, Repartition would do no better at distributing the elements than Coalesce.

Shuffling

Shuffling occurs when a transformation causes a parent partition to send tuples to multiple children, in turn, causing a wide dependency. As we seen in various examples so far, this is an expensive process that causes a lot of network traffic. Spark provides us with a few options – such as allowing us to decide how to partition our data – to help us avoid this process, or to prevent it from being more expensive than it needs to be. In the case where shuffling cannot be sidestepped, Spark gives alternatives for controlling the shuffling mechanism itself. Mainly for how the data is written to the disk.

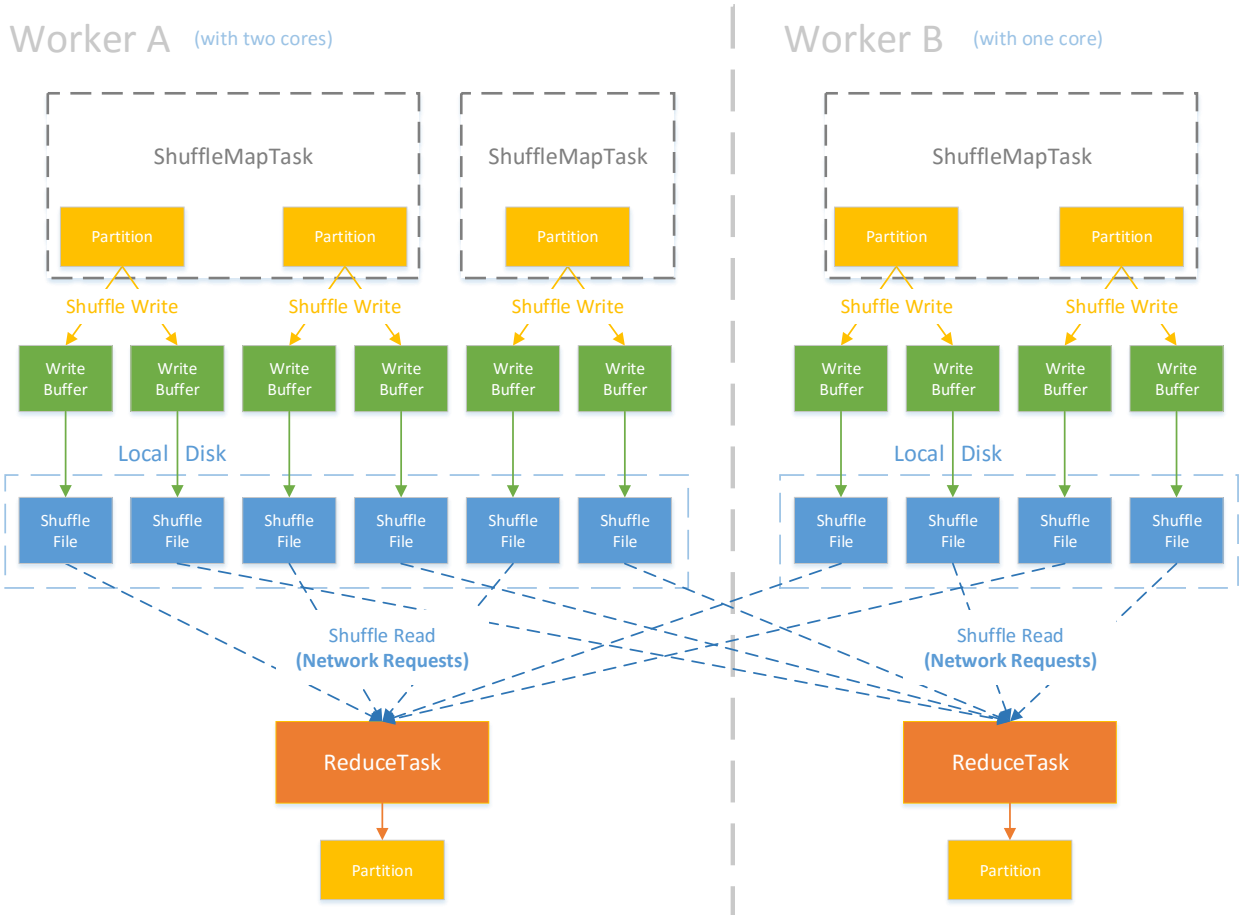
Internal Overview

In Spark, a shuffling operation looks similar to the one in MapReduce. When tuples are transferred between partitions, either between one or multiple RDDs, they are first written to disk. Afterwards they are sent to the partitions belonging to the appropriate child. The ShuffleMapTask is the process that writes to disk, whereas the ReduceTask is the process that reads. The overall process looks as follows:



At the start of a shuffling operation, each worker launches a ShuffleMapTask. Based on the number of cores, a worker will launch multiple tasks to utilize the maximum processing power. A ShuffleMapTask then iterates through every partition and breaks it down into files, which it saves onto the disk. The number of files created is dependent on the number of partitions in the resulting RDD. For example, if the new RDD has eight partitions, then each ShuffleMapTask will output at eight files, one for each partition.

Once the files are written, a ReduceTask, running on a separate worker, will read a subset of the files over the network. It will fetch the files it needs in order to create the partitions for its' part of the shuffled RDD. An alternative visualization, that explains how this process works for multiple workers, is shown below:

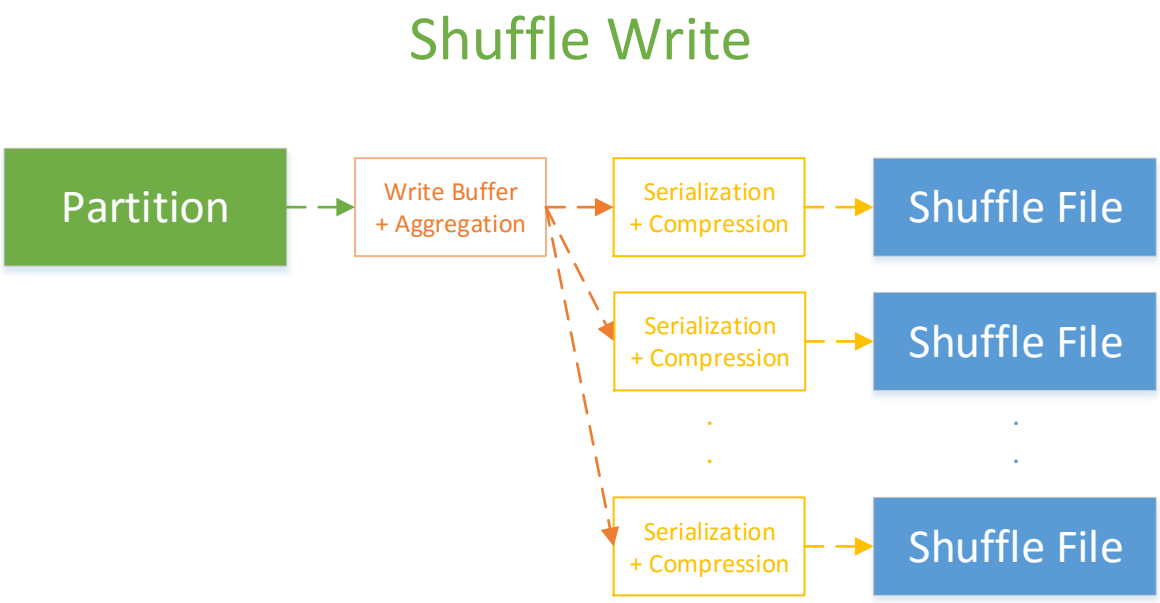


Examining the visualization above, we begin to notice a few challenges. Each ShuffleMapTask breaks down the partitions into multiple files, equal to the number of reducers – one created for every partition in the dependent RDD. Although the number within this example is small – only ten shuffle files are produced – the number grows exponentially large. For an RDD within a thousand partitions mapped to a thousand reducers, every partition will be broken down a thousand times, resulting in a million files in total. Such a scenario can overload the RAM – due to the amount of write buffers maintained – and it forces the worker to manage a lot of files. In some cases the worker opens more files than the maximum number allowed. On top of that the reducers have to make a lot of individual network requests.

Another issue we see is that the reducer and mapping tasks run on the same worker. This raises questions such as how much memory is to be allocated for each operation and whether or not the reducers are to start fetching data after all the mappers complete. By default, mappers and reducers do run separately. A Shuffle Read will only start after every Shuffle Write finishes. We will first discuss the reducer and mapping tasks in a bit more detail, before re-visiting the first challenge.

Shuffle Write

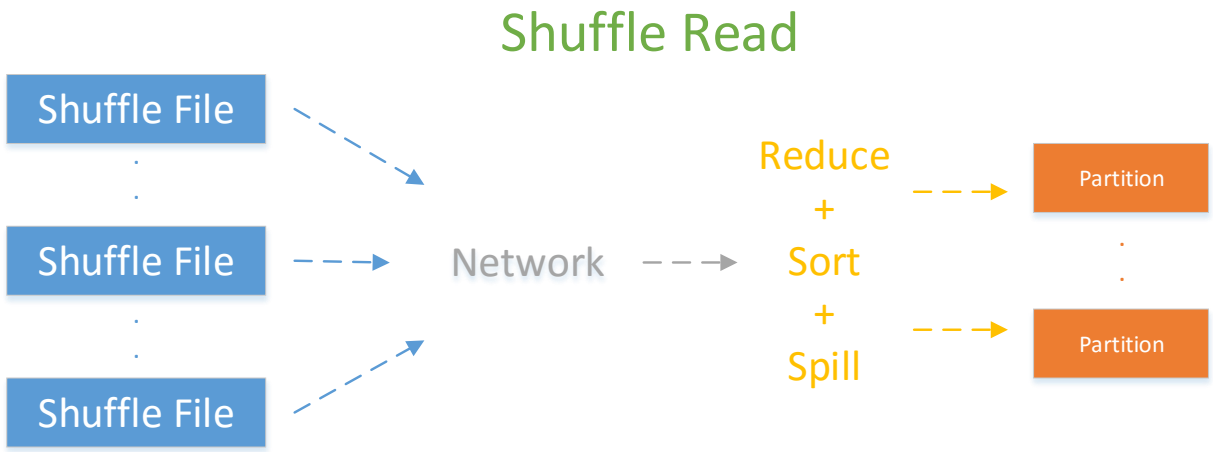
Shuffle Write is the process that distributes a partition across multiple local files based on the number of reducers present. To speed up the IO operations, Shuffle Write buffers the data in memory beforehand. Inside the buffer, the data is aggregated whenever a combination between the tuples is possible. The tuples are become serialized and compressed, before being written to a file. Although this last operation is expensive, it is necessary if the data is to be transferred efficiently over the network.



A Shuffle Write will produce multiple Shuffle Files per partition. At the minimum, this number is equal to the number of partitions the data from the first partition will be distributed across to. During a scenario where thousands of partitions exist between two RDDs joined by a shuffle dependency, the creation of files starts to become problematic. As a result, Spark offers two main implementations on how the data is written across each file. The first alternative, known for occasionally using up all the system resources, is the Hash Shuffle. A newer, more efficient alternative is the Sort Shuffle. Both will be discussed shortly.

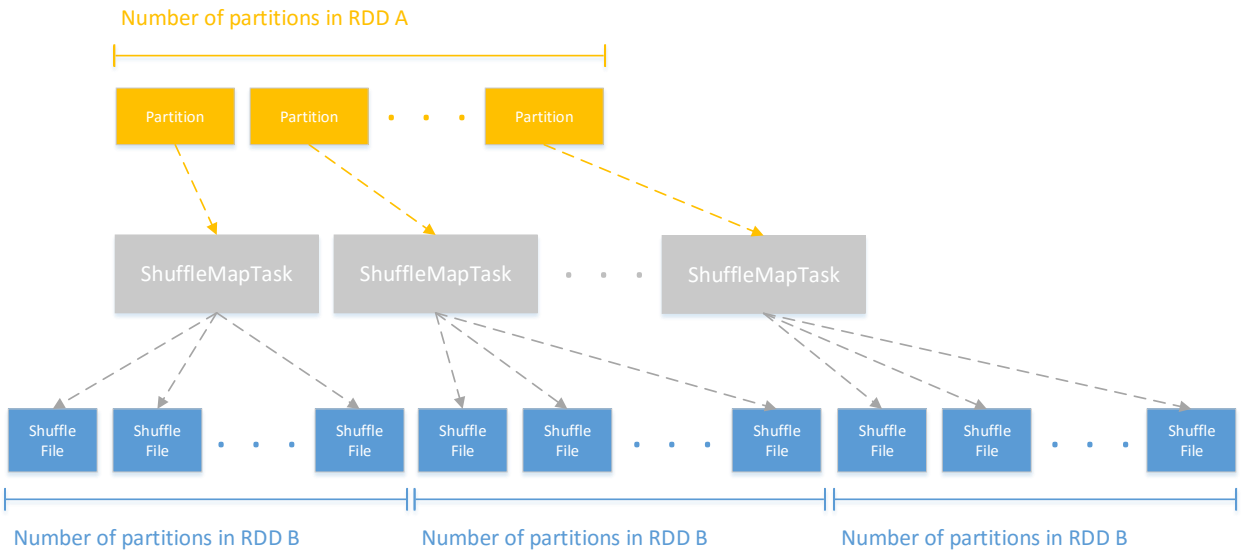
Shuffle Read

A shuffle read is launched by the Reducer Task. It fetches files over the network to create the set of partitions for the Shuffled RDD. By default, a Shuffle Read will start running on a worker once all the Shuffle Writers complete. While fetching the files, it will process them in memory, decreasing the time it takes to creating the resulting partitions. If necessary, the Shuffle Read will sort the data as well.

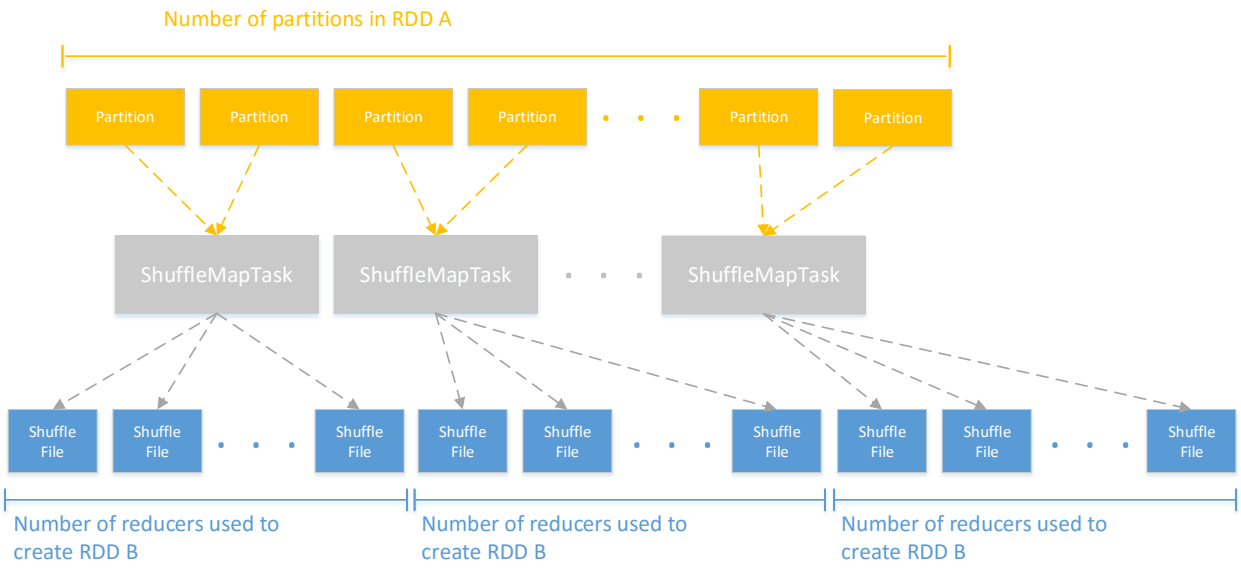


Hash Shuffle

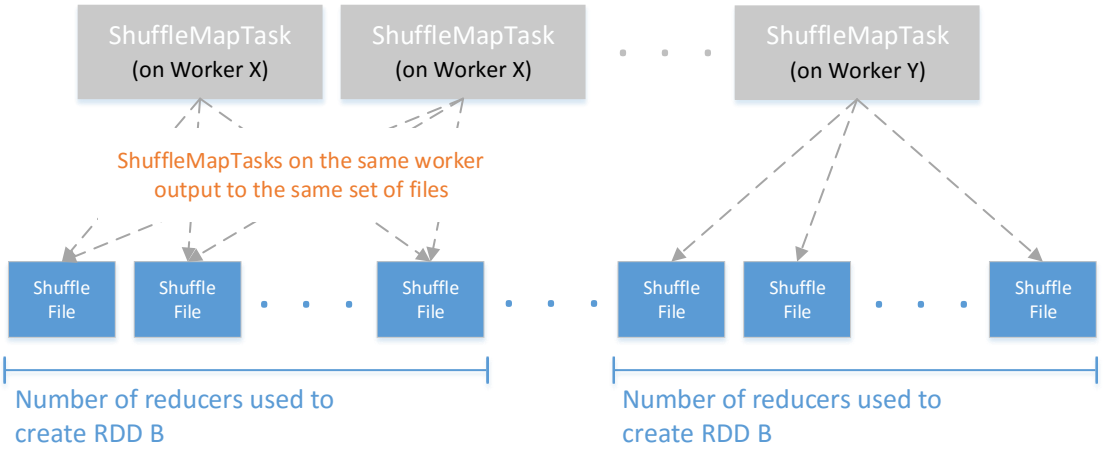
In order for one partition from an RDD to be distributed across multiple partitions in the transformed RDD, it has to be split and written across several files. Afterwards the reduce task from each worker reads the files it needs for its's subset of partitions. Following this logic, if there are a thousand partitions between two RDDs, then every partition will be split into a thousand files. Clearly, this is problematic. An example of how this would look like between two RDDs is shown below:



The Hash Shuffle, an older implementation of the read and write mechanisms, runs an operation similar to the approach described. It goes through each partition within the RDD and splits it across multiple files. It optimizes a bit by ensuring that each ShuffleMapTask outputs data into the same set of files and that the number of files produced is equal in amount to the number of reducers and not the partitions in the resulting RDD. For example, if twenty partitions are being shuffled into a set of thirty partitions, and there are four ShuffleMapTasks and four ReduceTasks present, then the number of files produced will be sixteen (4 x 4) instead of six hundred (20 x 30). A revised diagram showing the Hash Shuffle is shown below:



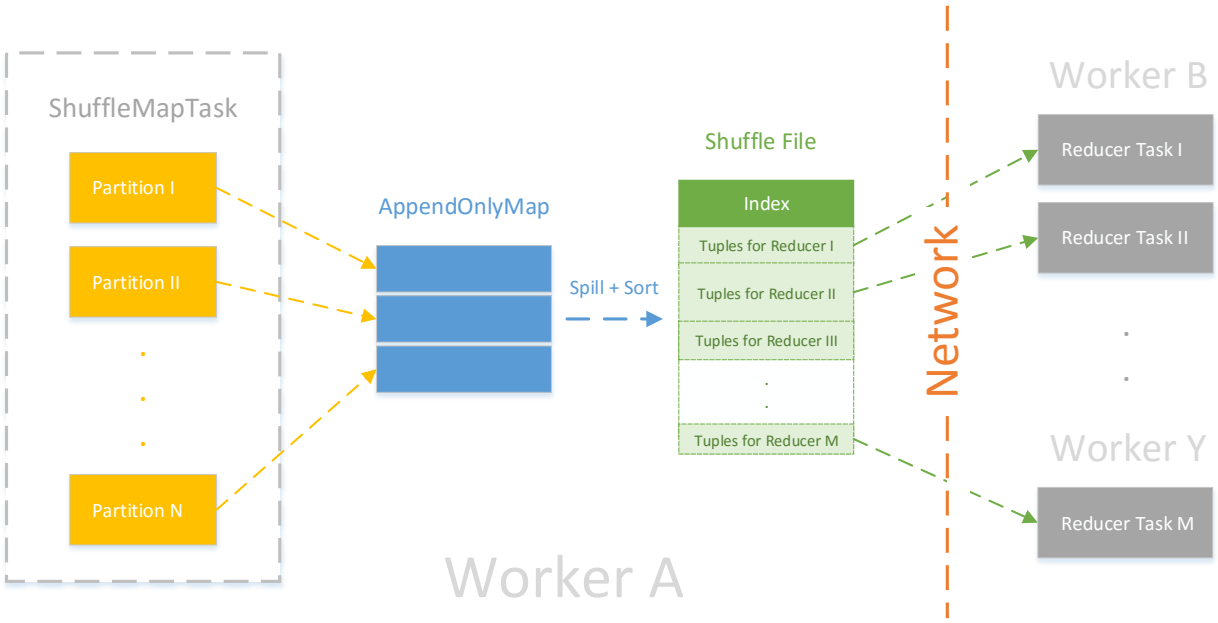
Hash Shuffle also provides a **consolidate** option for tasks running on the same worker to share the files written to. On a cluster with multiple cores per worker, this greatly reduces the number of files created.



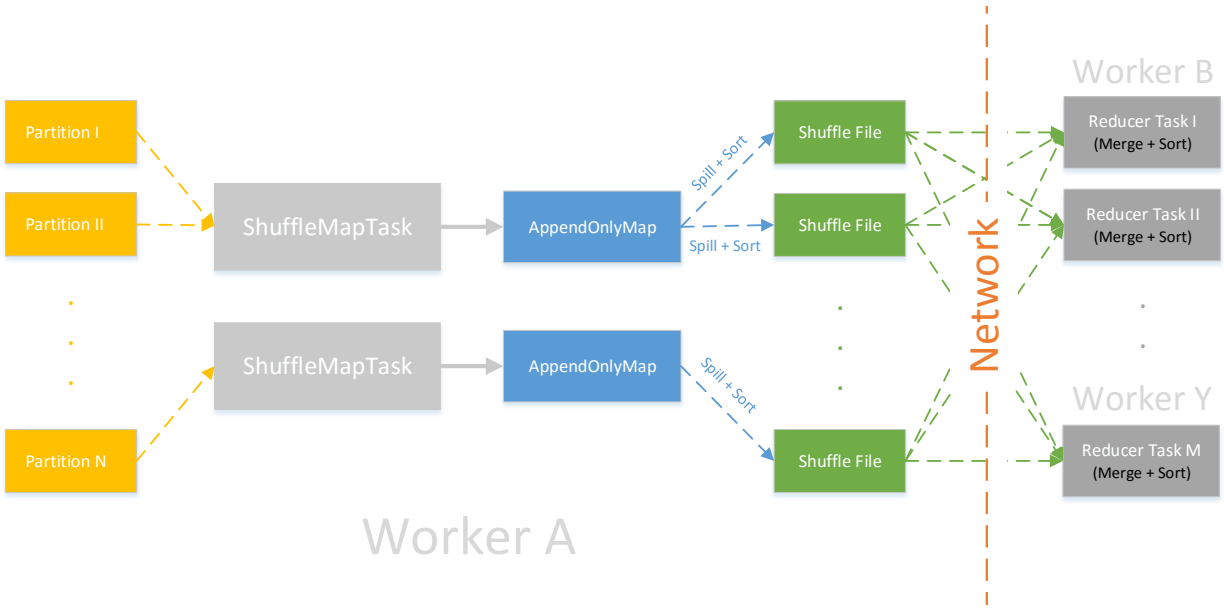
In either case however, the Hash Shuffle still generates a lot of files, especially within a Hadoop cluster containing thousands of nodes. As mentioned before, the output to these files also requires an in memory buffer for all the IO operations. Given that Spark's performance is largely dependent on how much memory is available, having the Shuffle hog too much memory is unacceptable.

Sort Shuffle

Sort shuffle is a newer implementation in Spark that fixes the issues addressed. Unlike the Hash Shuffle, which creates multiple files, the Sort Shuffle tries to create one file per mapping task. The file contains the mapping output grouped by the ID of the reducer, as well as an index indicating where each associated output starts. Each reducer then fetches the section of the file it needs using the help of the index.



The most important step in the above implementation is taking the tuples created from each input partition and sorting them by the ID of the reducer. This is done by first saving and updating them in memory using the AppendOnlyMap structure. From there they are spilled and sorted into a Shuffle File. If they cannot all fit into memory, they are spilled and sorted multiple times and the data from the files is only then merged when requested by a reducer. A visual representation of this is shown below:



As seen above, each mapping task can still output multiple files. This happens if the mapped data is very large in comparison to the memory allocated for the worker, or if the worker happens to have several cores kicking of each task. However the number of buffers kept in memory is constant. Each mapping task spills data to the disk in sequence and only after the current memory fills up – hence keeping one file opened at a time. The Hash Shuffle might keep thousands of files opened in parallel to match the number of reducers present, regardless if the data loaded is large or small. This is indeed an optimization.

Updates to the AppendOnlyMap table are dependent on the shuffling operation. For functions such as GroupByKey, each value stored in the table is a list of input tuples grouped by the key associated to that value. For ReduceByKey, each value updated for a given key is combined with the value already mapped at that location – or it is inserted if no previous value has been mapped for that key. This allows for the grouping and combination to be done together. An example of these two operations is shown:

Updates in ReduceByKey

```
function insert(key, val) {
  map.put(key, combine(map.get(key), val))
}
```

```
insert(K1, Tuple1)
insert(K4, Tuple2)
insert(K2, Tuple3)
insert(K1, Tuple4)
insert(K3, Tuple5)
insert(K3, Tuple6)
insert(K2, Tuple7)
insert(K3, Tuple8)
```

AppendOnlyMap

K2	combine(Tuple3, Tuple7)
K1	combine(Tuple1, Tuple4)
K4	Tuple2
K3	combine(combine(Tuple5, Tuple6), Tuple8)

Updates in GroupByKey

```
function insert(key, val) {
  map.put(key, append(map.get(key), val))
}
```

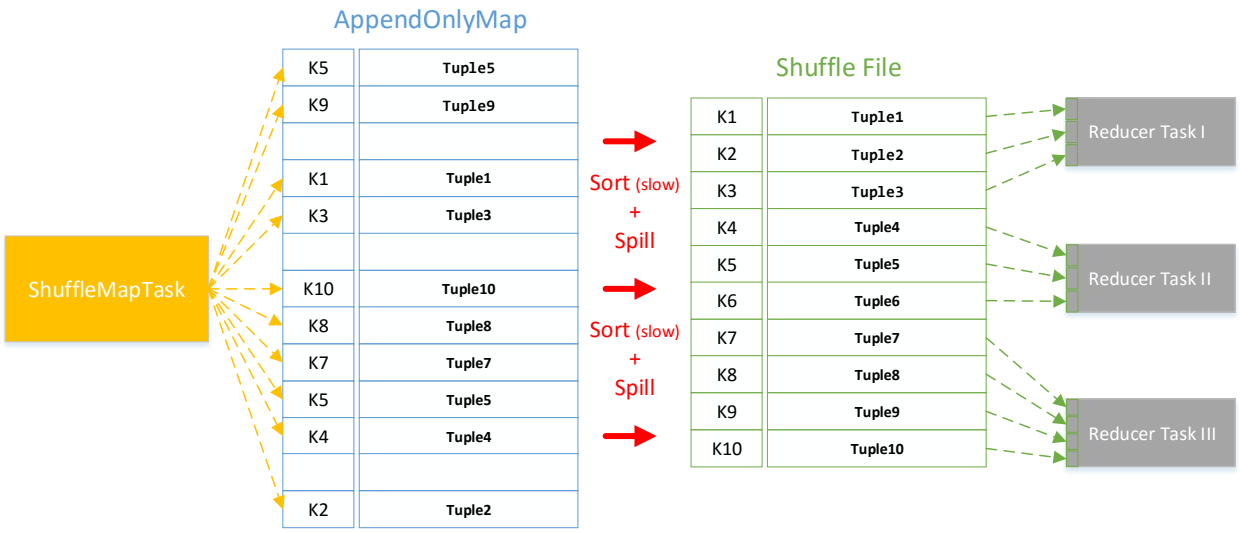
```
insert(K1, Tuple1)
insert(K4, Tuple2)
insert(K2, Tuple3)
insert(K1, Tuple4)
insert(K3, Tuple5)
insert(K3, Tuple6)
insert(K2, Tuple7)
insert(K3, Tuple8)
```


AppendOnlyMap

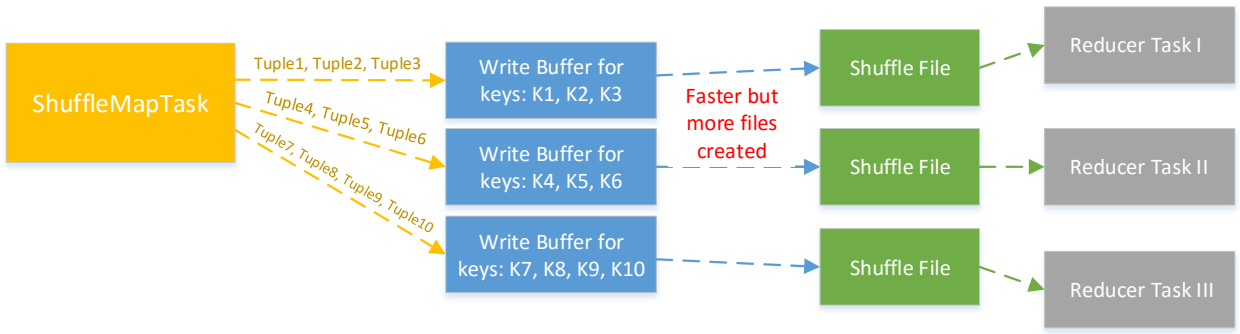
K4	[Tuple2]
K2	[Tuple3, Tuple7]
K3	[Tuple5,Tuple6,Tuple8]
K1	[Tuple1, Tuple4]

The only consequence of the Sort Shuffle is that the mapped data within the AppendOnlyMap table needs to be sorted first before it is written to a file – a process that is slower than if the data was to be mapped out to separate files directly. Also note that sorting the data within the AppendOnlyMap table is absolutely necessary because the reducers need to be able to fetch their section of the file. Therefore, if the number of reducers is below the threshold specified in the **spark.shuffle.sort.bypassMergeThreshold** value, Sort Shuffle falls back to creating a separate file for each reducer, before merging the files together. This value is set to two hundred by default, but it can be configured to anything custom for best performance.

Stand Sort Shuffle



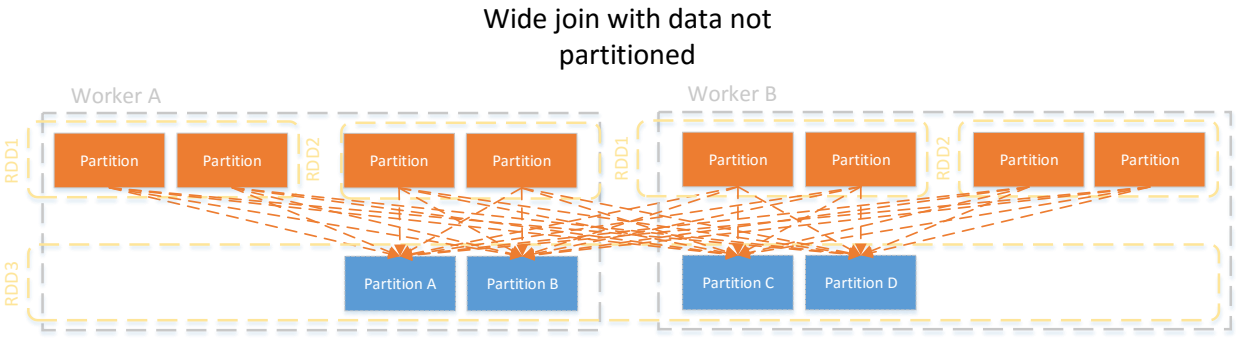
Sorting bypassed



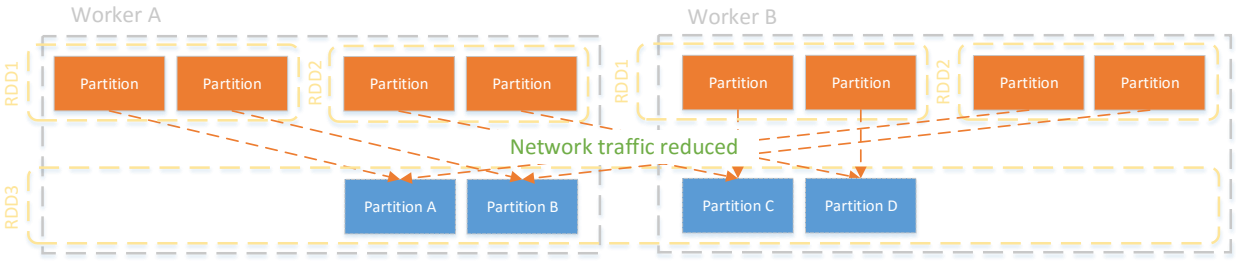
Preventions and Optimizations

As we saw, the Sort Shuffle adds a huge improvement for distributing partitions to their respective workers. However there are still a few inefficiencies. One is the serialization and deserialization of data to and from a file, for reading and object merging purposes. The second is the latency of the network involved. Even with efficient disk writes and cases where data is kept in memory for as long as possible, the data transfer from worker to worker can still lbe a bottleneck.

Use a partitioner: One optimization we have already described before is using a shared partitioner for operations such unions and intersections between two or more RDDs. Doing so will not fully prevent a shuffle because the RDDs might still have their partitions on different workers, and these partitions will need to be moved over the network to their shared location. However a worker in the subsequent stage only needs to request the partitions from a fixed set of locations, rather than pulling partitions from every worker in the cluster. This reduces the additional network traffic.

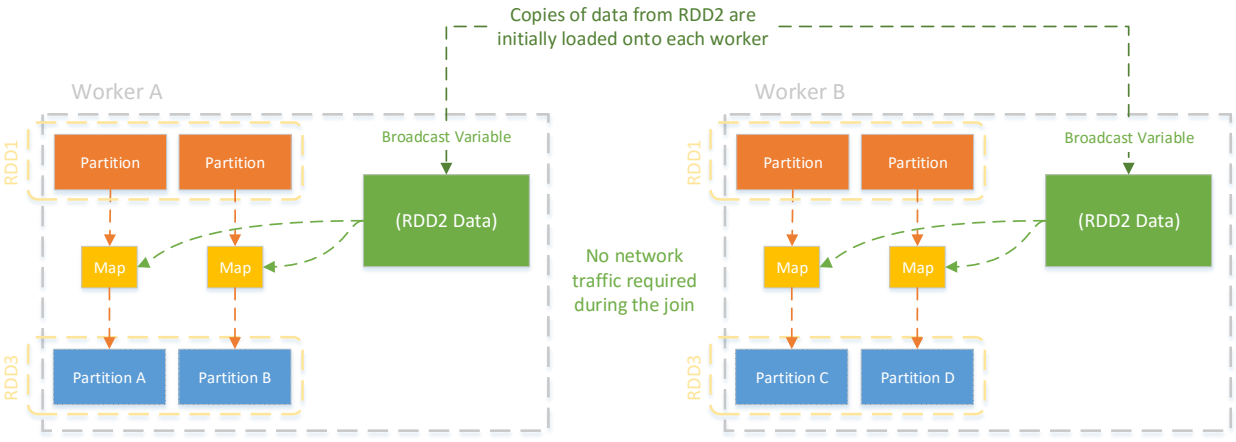


Narrow join with data co-partitioned

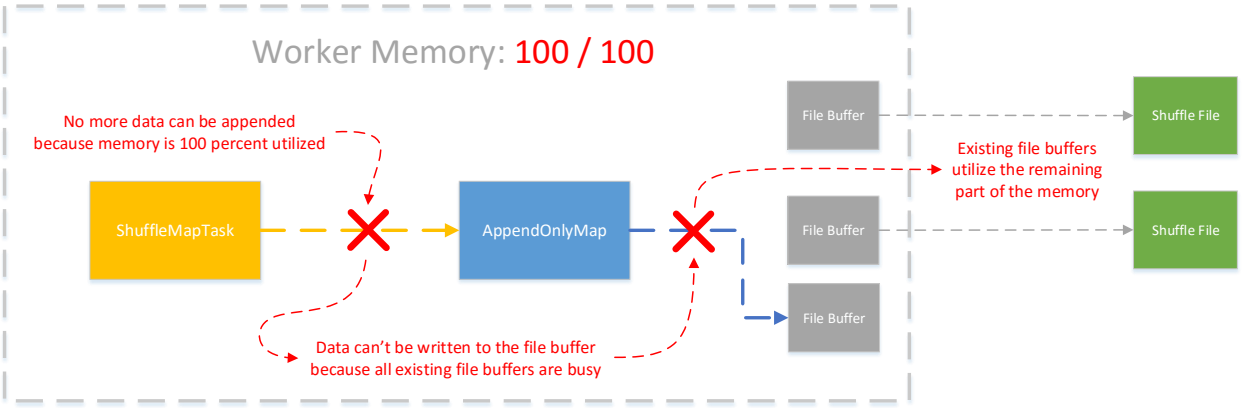


Use broadcast if possible: Broadcast variables can be used in place of a direct join or intersection. Rather than having both RDDs in memory and then distributing their partitions to shared locations where the individual joins or intersections can occur, the entire data from one RDD can be loaded into memory and distributed as a copy to every worker. Then through a mapping function, partitions belonging to the second RDD can be updated while making use of the local data from that variable. This requires that the data broadcasted takes up a small fraction of the worker's memory, so as to not cause a memory overflow. Spark can use the broadcast functionality automatically if the size of one of the RDDs is smaller than the value set in the **spark.sql.autoBroadcastJoinThreshold** configuration.

Joining data in RDD1 using a Broadcast variable

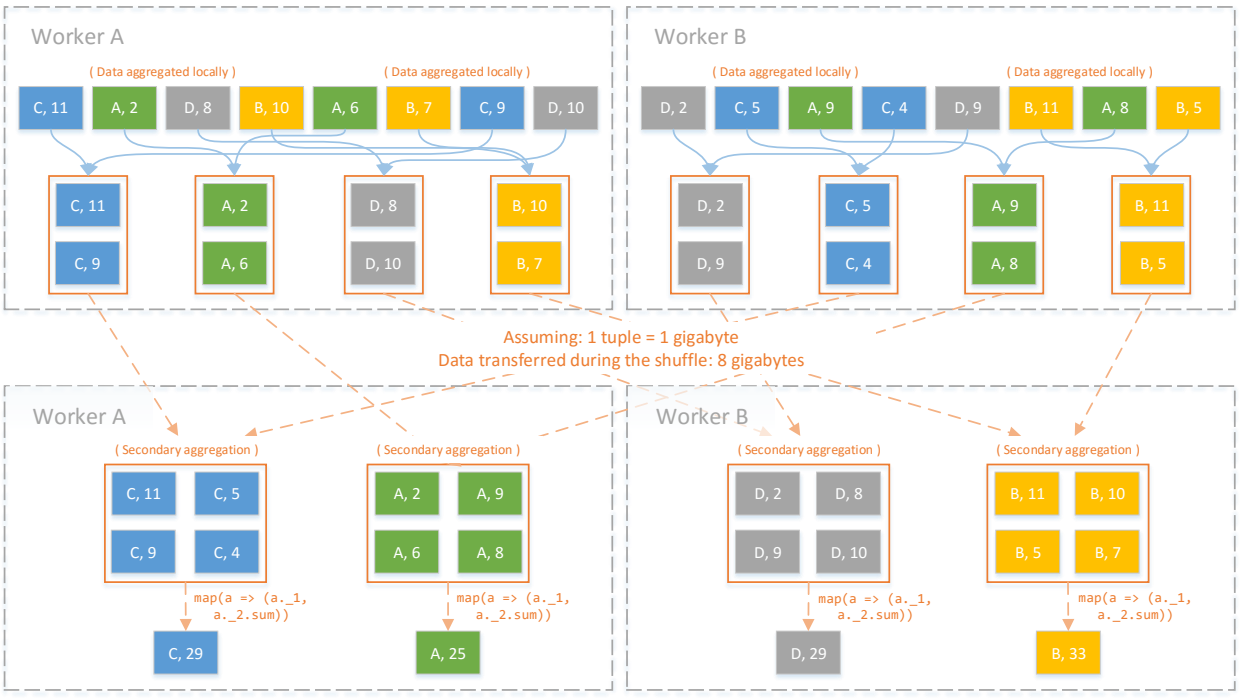


Do not overflow the buffers: If a complete shuffle cannot be avoided, an important piece to look out for are the memory buffers. As seen previously, both Hash and Sort implementations utilize buffers for outputting data to a file and for organizing it in memory for purposes such as tuple combinations. However once the worker's memory limit is reached, data needs to be spilled over before tasks can resume. Spilling the data to a file takes time and a pile up can be caused if new data needs to be spilled while the maximum number of file buffers are already active. **spark.shuffle.memoryFraction** sets the amout of worker memory allocated to the shuffling operations – set to twenty percent by default. The **spark.shuffle.safetyFraction** configuration then determines the percentage of that allocated memory that can be safely used. After this threshold, data will start spilling simultaneously.

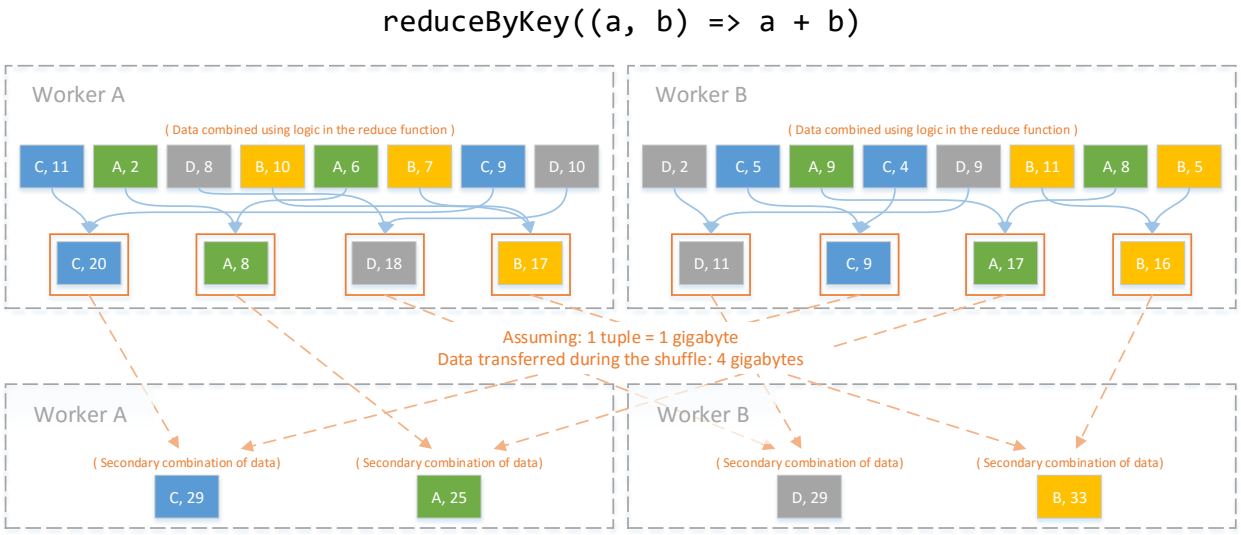


Use reduce over group by: Use ReduceByKey over GroupByKey. As mentioned before, the most expensive operation during the Shuffle can occur during the network requests. This is true nowadays where most drives are solid states, making the file IO operations fast in comparison. A reduce operation is similar in functionality to the combiner used in MapReduce. Rather than sending millions of tuples over the network, all which have to be separately serialized and deserialized afterwards, it is better to combine them first locally into as few entities as possible. Consider the next example below that makes use of the groupByKey operator. The input is a set of tuples, each tuple consists of a letter and number. The goal is to output the sum of numbers for each letter. The logic to accomplish this would look as follows:

```
groupByKey().map(a => (a._1, a._2.sum))
```

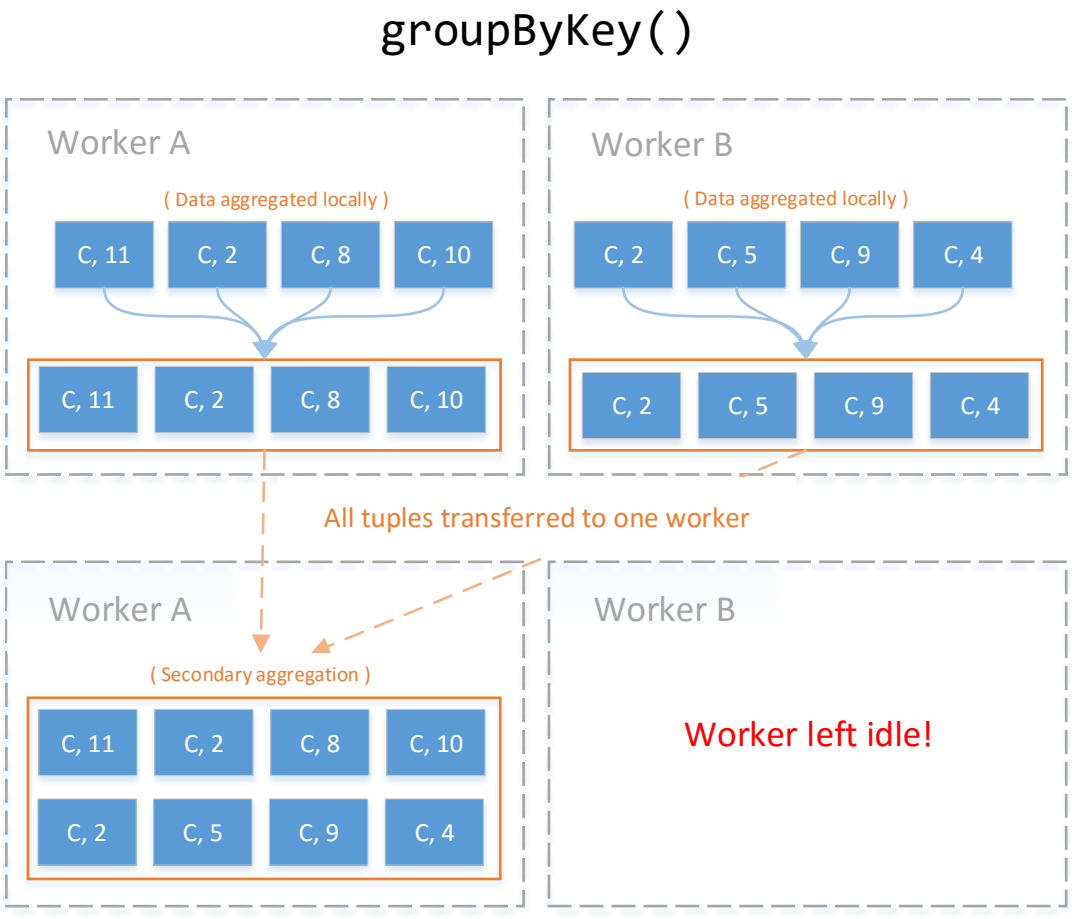


The groupByKey operator first groups the tuples together by letter. Afterwards a mapping function is used on each group to sum up the numbers. As we can see in the shuffling operation created, half the tuples from each worker are transferred over, amounting to eight tuples transferred in total. For the purpose of this example, if each tuple was a gigabyte in size, this would generate eight gigabytes in traffic. Now consider the reduceByKey operator, which produces the same output, without the mapping function:



The first major difference seen with the reduceByKey operator is that the overall process looks cleaner. Less tuples move around, both before and after the shuffling takes place. Tuples are immediately combined locally using the logic defined inside the reduceBy function – this combination results in half the network traffic. The example we are using has sixteen tuples as input, and the combination results in just eight. However we can use a much larger number in place of sixteen and the reduction in the number of tuples can become an order of magnitude times bigger – assuming that the number of unique letters in the input stays the same.

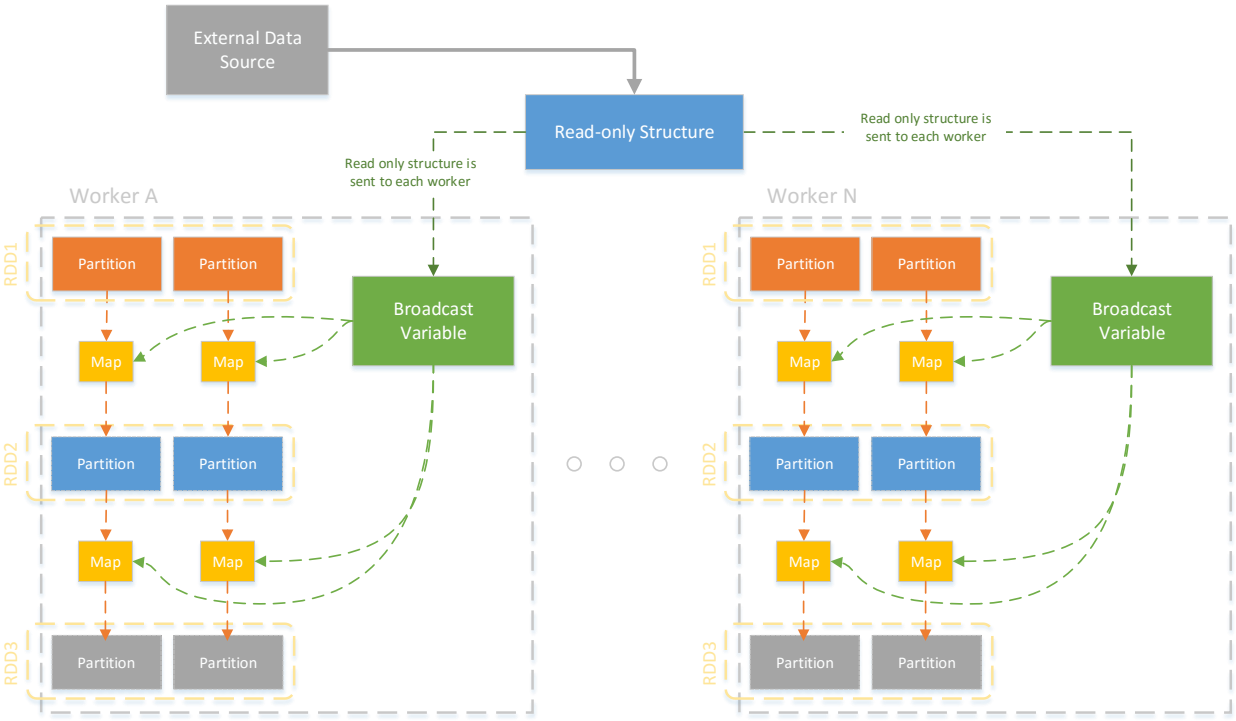
Do not group data into a single partition: As we seen in the example above, using reduceByKey in place of groupByKey reduces the amount of tuples that Spark has to deal with – as the name implies. However when the aggregation logic is neither commutative or associative, tuples cannot be combined – leaving groupByKey as the only option. In such a case, it becomes important to avoid grouping data into a single key. Otherwise all the tuples belonging to that one key will be brought to a single worker and the remaining part of the cluster will be left idle. We can see this below:



Shared Structures

Broadcast Variables

Broadcast variables allow datasets to be loaded into a read only structure in memory. This structure is then sent as a copy to every worker. Tasks running on each worker can then make use of the data within this structure. For example, while processing or updating each tuple of an RDD, a mapping function would look up data within the Broadcast variable if needed, rather then fetching data from an external source.



In the example above, there are two separate mapping stages making use of the same broadcast variable. This is the recommended approach. A broadcast variable still needs to be loaded onto each worker, serialized and de-serialized in the process. Rather than broadcasting data for a mapping operation each time, it is better to use a broadcast the data once for multiple transformations at a time.

As described previously, broadcast variables are a great optimization for avoiding a shuffle. Rather than joining two RDDs and having tuples shuffled across the network – a job that creates small individual requests – data from the first RDD can be loaded once as a broadcast variable onto each worker. A mapping transformation running on the second RDD can then use this variable as a look up table to do the join. However since the entire RDD is broadcasted, one must ensure that its' data is small enough to fit into the memory of each worker.

Online References

Overviews

- https://www.dezyre.com/article/apache-spark-architecture-explained-in-detail/338
- http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/
- https://www.infoq.com/articles/apache-spark-introduction
- https://www.slideshare.net/AGrishchenko/apache-spark-architecture

RDDs

- https://www.dezyre.com/article/working-with-spark-rdd-for-fast-data-processing/273
- https://dirtysalt.github.io/spark-rdd-paper.html

Partitioning

- https://www.edureka.co/blog/demystifying-partitioning-in-spark
- http://parrotprediction.com/partitioning-in-apache-spark/
- https://hackernoon.com/managing-spark-partitions-with-coalesce-and-repartition-4050c57ad5c4
- https://stackoverflow.com/questions/31610971/spark-repartition-vs-coalesce
- http://www.bigsynapse.com/controlling-the-number-of-partitions-in-spark
- https://techmagie.wordpress.com/2015/12/19/understanding-spark-partitioning/
- https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies

Memory

- https://0x0fff.com/spark-memory-management/

Fault tolerance

- https://databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html
- https://stackoverflow.com/questions/39189483/on-which-way-does-rdd-of-spark-finish-fault-tolerance

Shuffling

- https://0x0fff.com/spark-architecture-shuffle/
- https://www.slideshare.net/colorant/spark-shuffle-introduction
- http://apache-spark-user-list.1001560.n3.nabble.com/How-does-shuffle-work-in-spark-td584.html
- http://apache-spark-user-list.1001560.n3.nabble.com/Co-Partitioned-Joins-td25956.html
- https://trongkhoanguyen.com/spark/understand-the-shuffle-component-in-spark-core/
- https://stackoverflow.com/questions/40949835/does-spark-write-to-disk-intermidiate-shuffle-outputs
- https://stackoverflow.com/questions/41661849/spill-to-disk-and-shuffle-write-spark
- http://www.waitingforcode.com/apache-spark/shuffling-in-spark/read
- https://github.com/JerryLead/SparkInternals/blob/master/markdown/english/4-shuffleDetails.md

Broadcasting and Accumulators

- http://www.sparktutorials.net/spark-broadcast-variables---what-are-they-and-how-do-i-use-them
- https://www.supergloo.com/fieldnotes/spark-broadcast-accumulator-examples-scala/