How-to: Tune Your Apache Spark Jobs (Part 1)

March 9, 2015
By Sandy Ryza
10 Comments
Categories: How-to Spark

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

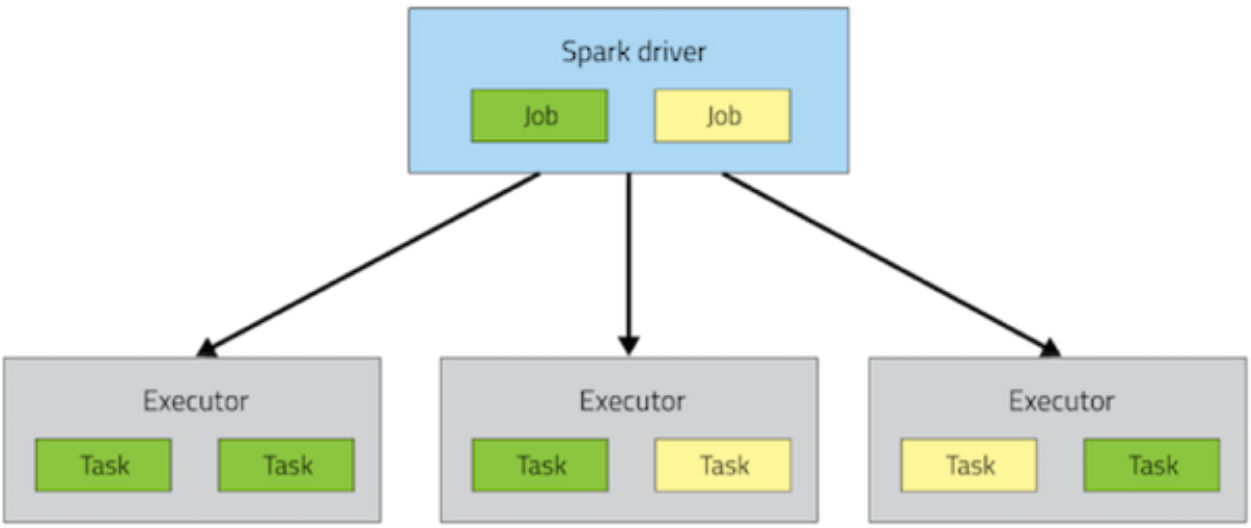
When you write Apache Spark code and page through the public APIs, you come across words like *transformation*, *action*, and *RDD*. Understanding Spark at this level is vital for writing Spark programs. Similarly, when things start to fail, or when you venture into the web UI to try to understand why your application is taking so long, you're confronted with a new vocabulary of words like *job*, *stage*, and *task*. Understanding Spark at this level is vital for writing *good* Spark programs, and of course by *good,* I mean *fast*. To write a Spark program that will execute efficiently, it is very, very helpful to understand Spark's underlying execution model.

In this post, you'll learn the basics of how Spark programs are actually executed on a cluster. Then, you'll get some practical recommendations about what Spark's execution model means for writing efficient programs.

How Spark Executes Your Program

A Spark application consists of a single *driver* process and a set of *executor* processes scattered across nodes on the cluster.

The driver is the process that is in charge of the high-level control flow of work that needs to be done. The executor processes are responsible for executing this work, in the form of *tasks*, as well as for storing any data that the user chooses to cache. Both the driver and the executors typically stick around for the entire time the application is running, although dynamic resource allocation changes that for the latter. A single executor has a number of slots for running tasks, and will run many concurrently throughout its lifetime. Deploying these processes on the cluster is up to the cluster manager in use (YARN, Mesos, or Spark Standalone), but the driver and executor themselves exist in every Spark application.



At the top of the execution hierarchy are *jobs*. Invoking an action inside a Spark application triggers the launch of a Spark job to fulfill it. To decide what this job looks like, Spark examines the graph of RDDs on which that action depends and formulates an execution plan. This plan starts with the farthest-back RDDs—that is, those that depend on no other RDDs or reference already-cached data–and culminates in the final RDD required to produce the action's results.

The execution plan consists of assembling the job's transformations into *stages*. A stage corresponds to a collection of *tasks* that all execute the same code, each on a different subset of the data. Each stage contains a sequence of transformations that can be completed without *shuffling* the full data.

What determines whether data needs to be shuffled? Recall that an RDD comprises a fixed number of partitions, each of which comprises a number of records. For the RDDs returned by so-called *narrow* transformations like map and filter, the records required to compute the records in a single partition reside in a single partition in the parent RDD. Each object is only dependent on a single object in the parent. Operations like coalesce can result in a task processing multiple input partitions, but the transformation is still considered narrow because the input records used to compute any single output record can still only reside in a limited subset of the partitions.

However, Spark also supports transformations with *wide* dependencies such as groupByKey and reduceByKey. In these dependencies, the data required to compute the records in a single partition may reside in many partitions of the parent RDD. All of the tuples with the same key must end up in the same partition, processed by the same task. To satisfy these operations, Spark must execute a shuffle, which transfers data around the cluster and results in a new stage with a new set of partitions.

For example, consider the following code:

```
sc.textFile("someFile.txt").
      map(mapFunc).
      flatMap(flatMapFunc).
      filter(filterFunc).
      count()
```

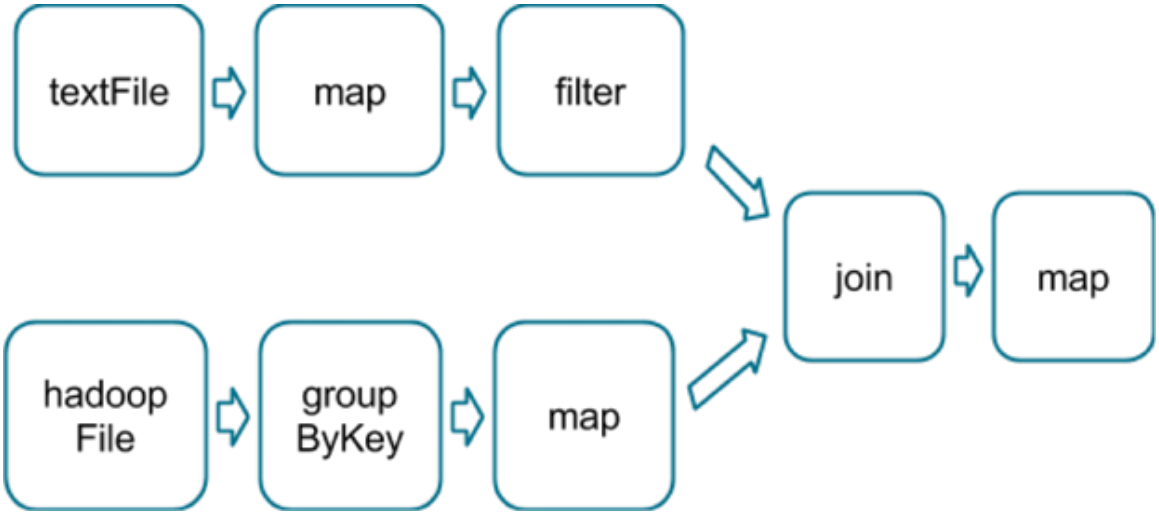
It executes a single action, which depends on a sequence of transformations on an RDD derived from a text file. This code would execute in a single stage, because none of the outputs of these three operations depend on data that can come from different partitions than their inputs.

In contrast, this code finds how many times each character appears in all the words that appear more than 1,000 times in a text file.

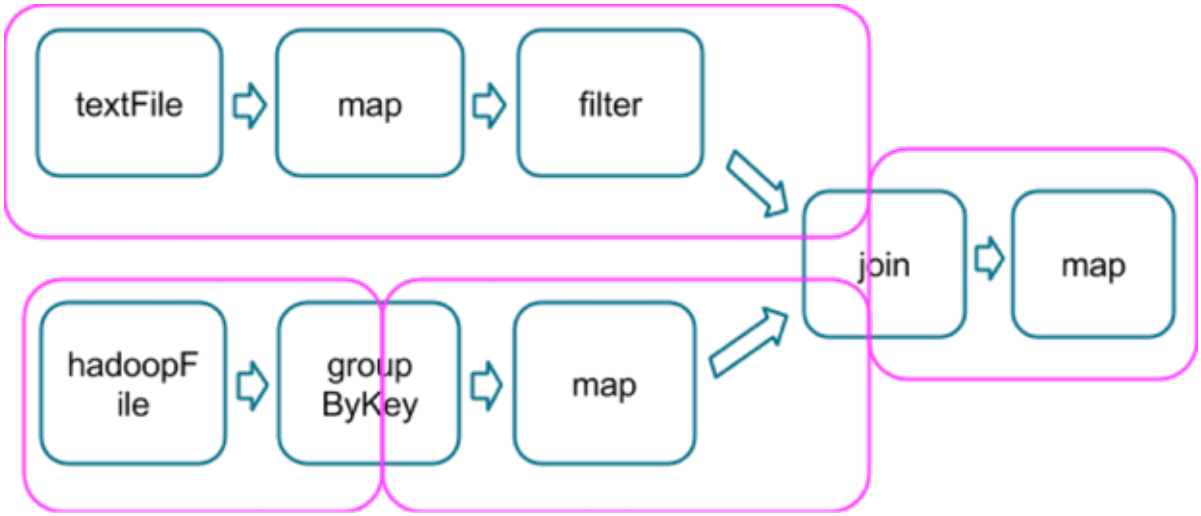
```
val tokenized = sc.textFile(args(0)).flatMap(_.split(' '))
val wordCounts = tokenized.map((_, 1)).reduceByKey(_ + _)
val filtered = wordCounts.filter(_._2 >= 1000)
val charCounts = filtered.flatMap(_._1.toCharArray).map((_, 1)).
      reduceByKey(_ + _)
charCounts.collect()
```

This process would break down into three stages. The `reduceByKey` operations result in stage boundaries, because computing their outputs requires repartitioning the data by keys.

Here is a more complicated transformation graph including a join transformation with multiple dependencies.



The pink boxes show the resulting stage graph used to execute it.



At each stage boundary, data is written to disk by tasks in the *parent* stages and then fetched over the network by tasks in the *child* stage. Because they incur heavy disk and network I/O, stage boundaries can be expensive and should be avoided when possible. The number of data partitions in the parent stage may be different than the number of partitions in the child stage. Transformations that may trigger a stage boundary typically accept a `numPartitions` argument that determines how many partitions to split the data into in the child stage.

Just as the number of reducers is an important parameter in tuning MapReduce jobs, tuning the number of partitions at stage boundaries can often make or break an application's performance. We'll delve deeper into how to tune this number in a later section.

Picking the Right Operators

When trying to accomplish something with Spark, a developer can usually choose from many arrangements of actions and transformations that will produce the same results. However, not all these arrangements will result in the same performance: avoiding common pitfalls and picking the right arrangement can make a world of difference in an application's performance. A few rules and insights will help you orient yourself when these choices come up.

Recent work in SPARK-5097 began stabilizing SchemaRDD, which will open up Spark's Catalyst optimizer to programmers using Spark's core APIs, allowing Spark to make some higher-level choices about which operators to use. When SchemaRDD becomes a stable component, users will be shielded from needing to make some of these decisions.

The primary goal when choosing an arrangement of operators is to reduce the number of shuffles and the amount of data shuffled. This is because shuffles are fairly expensive operations; all shuffle data must be written to disk and then transferred over the network. `repartition` , `join`, `cogroup`, and any of the `*By` or `*ByKey` transformations can result in shuffles. Not all these operations are equal, however, and a few of the most common performance pitfalls for novice Spark developers arise from picking the wrong one:

- **Avoid `groupByKey` when performing an associative reductive operation.** For example, `rdd.groupByKey().mapValues(_.sum)` will produce the same results as `rdd.reduceByKey(_ + _)`. However, the former will transfer the entire dataset across the network, while the latter will compute local sums for each key in each partition and combine those local sums into larger sums after shuffling.
- **Avoid `reduceByKey` When the input and output value types are different.** For example, consider writing a transformation that finds all the unique strings corresponding to each key. One way would be to use map to transform each element into a `Set` and then combine the `Set`s with `reduceByKey`:

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

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

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

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

When Shuffles Don't Happen

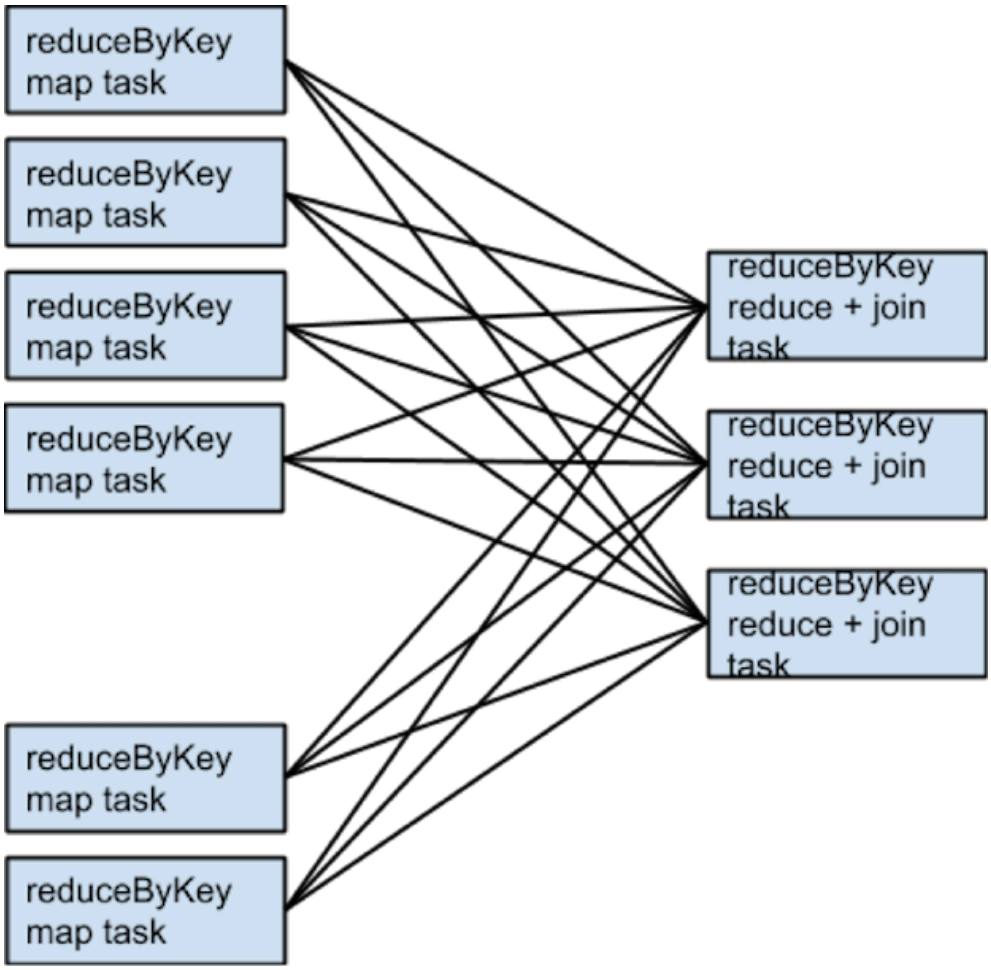
It's also useful to be aware of the cases in which the above transformations will *not* result in shuffles. Spark knows to avoid a shuffle when a previous transformation has already partitioned the data according to the same partitioner. Consider the following flow:

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

Because no partitioner is passed to `reduceByKey`, the default partitioner will be used, resulting in rdd1 and rdd2 both hash-partitioned. These two `reduceByKey`s will result in two shuffles. If the RDDs have the same number of partitions, the join will require no additional shuffling. Because the RDDs are partitioned identically, the set of keys in any single partition of rdd1 can only show up in

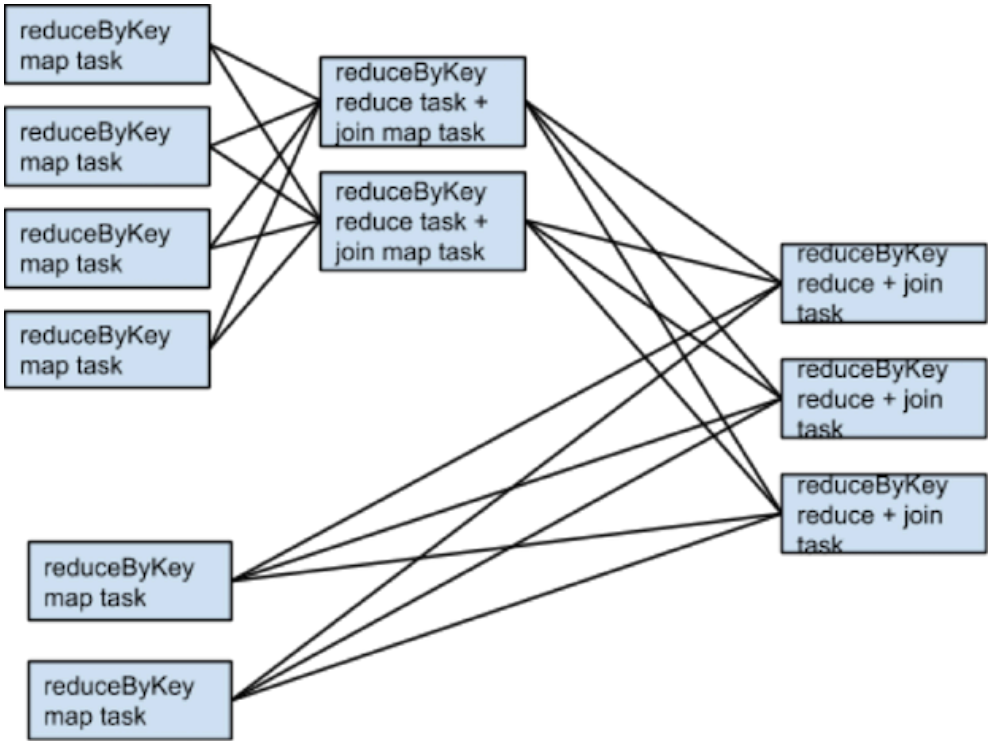
a single partition of rdd2. Therefore, the contents of any single output partition of rdd3 will depend only on the contents of a single partition in rdd1 and single partition in rdd2, and a third shuffle is not required.

For example, if `someRdd` has four partitions, `someOtherRdd` has two partitions, and both the `reduceByKey`s use three partitions, the set of tasks that execute would look like:



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

Same transformations, same inputs, different number of partitions:



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

When More Shuffles are Better

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

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

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

approach, which can be accomplished with `aggregateByKey`, is to perform the count in a fully distributed way, and then simply `collectAsMap` the results to the driver.

Secondary Sort

Another important capability to be aware of is the `repartitionAndSortWithinPartitions` transformation. It's a transformation that sounds arcane, but seems to come up in all sorts of strange situations. This transformation pushes sorting down into the shuffle machinery, where large amounts of data can be spilled efficiently and sorting can be combined with other operations.

For example, Apache Hive on Spark uses this transformation inside its join implementation. It also acts as a vital building block in the *secondary sort* pattern, in which you want to both group records by key and then, when iterating over the values that correspond to a key, have them show up in a particular order. This issue comes up in algorithms that need to group events by user and then analyze the events for each user based on the order they occurred in time. Taking advantage of `repartitionAndSortWithinPartitions` to do secondary sort currently requires a bit of legwork on the part of the user, but SPARK-3655 will simplify things vastly.

Conclusion

You should now have a good understanding of the basic factors in involved in creating a performance-efficient Spark program! In Part 2, we'll cover tuning resource requests, parallelism, and data structures.

Sandy Ryza is a Data Scientist at Cloudera, an Apache Spark committer, and an Apache Hadoop PMC member. He is a co-author of the O'Reilly Media book, Advanced Analytics with Spark.

How-to: Tune Your Apache Spark Jobs (Part 2)

March 30, 2015
By Sandy Ryza
25 Comments
Categories: How-to Spark

In the conclusion to this series, learn how resource tuning, parallelism, and data representation affect Spark job performance.

In this post, we'll finish what we started in "How to Tune Your Apache Spark Jobs (Part 1)". I'll try to cover pretty much everything you could care to know about making a Spark program run fast. In particular, you'll learn about resource tuning, or configuring Spark to take advantage of everything the cluster has to offer. Then we'll move to tuning parallelism, the most difficult as well as most important parameter in job performance. Finally, you'll learn about representing the data itself, in the on-disk form which Spark will read (spoiler alert: use Apache Avro or Apache Parquet) as well as the in-memory format it takes as it's cached or moves through the system.

Tuning Resource Allocation

The Spark user list is a litany of questions to the effect of "I have a 500-node cluster, but when I run my application, I see only two tasks executing at a time. HALP." Given the number of parameters that control Spark's resource utilization, these questions aren't unfair, but in this section, you'll learn how to squeeze every last bit of juice out of your cluster. The recommendations and configurations here differ a little bit between Spark's cluster managers (YARN, Mesos, and Spark Standalone), but we're going to focus only on YARN, which Cloudera recommends to all users.

For some background on what it looks like to run Spark on YARN, check out my post on this topic.

The two main resources that Spark (and YARN) think about are *CPU* and *memory*. Disk and network I/O, of course, play a part in Spark performance as well, but neither Spark nor YARN currently do anything to actively manage them.

Every Spark executor in an application has the same fixed number of cores and same fixed heap size. The number of cores can be specified with the `--executor-cores` flag when invoking spark-submit, spark-shell, and pyspark from the command line, or by setting the `spark.executor.cores` property in the `spark-defaults.conf` file or on a `SparkConf` object. Similarly, the heap size can be controlled with the `--executor-memory` flag or the `spark.executor.memory` property. The `cores` property controls the number of concurrent tasks an executor can run. `--executor-cores 5` means that each executor can run a maximum of five tasks at the same time. The memory property impacts the amount of data Spark can cache, as well as the maximum sizes of the shuffle data structures used for grouping, aggregations, and joins.

The `--num-executors` command-line flag or `spark.executor.instances` configuration property control the number of executors requested. Starting in CDH 5.4/Spark 1.3, you will be able to avoid setting this property by turning on dynamic allocation with the `spark.dynamicAllocation.enabled` property. Dynamic allocation enables a Spark application to request executors when there is a backlog of pending tasks and free up executors when idle.

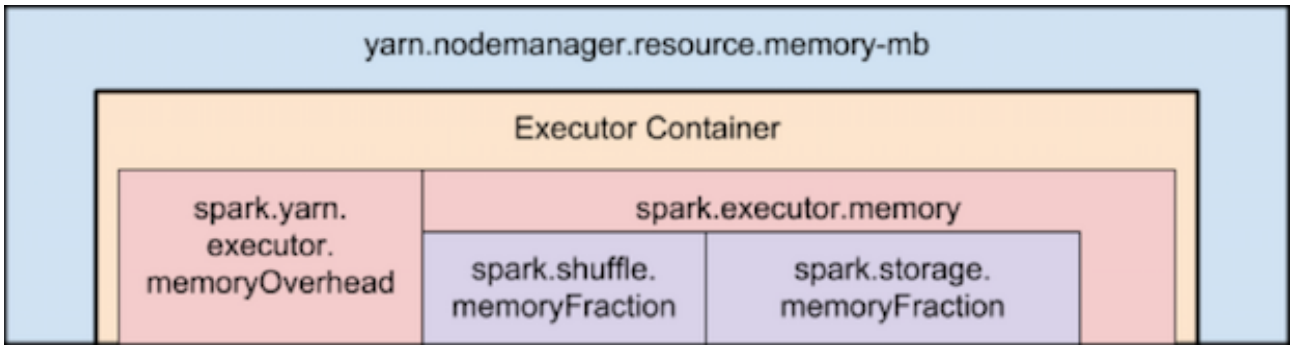
It's also important to think about how the resources requested by Spark will fit into what YARN has available. The relevant YARN properties are:

- `yarn.nodemanager.resource.memory-mb` controls the maximum sum of memory used by the containers on each node.
- `yarn.nodemanager.resource.cpu-vcores` controls the maximum sum of cores used by the containers on each node.

Asking for five executor cores will result in a request to YARN for five virtual cores. The memory requested from YARN is a little more complex for a couple reasons:

- `--executor-memory/spark.executor.memory` controls the executor heap size, but JVMs can also use some memory off heap, for example for interned Strings and direct byte buffers. The value of the `spark.yarn.executor.memoryOverhead` property is added to the executor memory to determine the full memory request to YARN for each executor. It defaults to max(384, .07 * spark.executor.memory).
- YARN may round the requested memory up a little. YARN's `yarn.scheduler.minimum-allocation-mb` and `yarn.scheduler.increment-allocation-mb` properties control the minimum and increment request values respectively.

The following (not to scale with defaults) shows the hierarchy of memory properties in Spark and YARN:



And if that weren't enough to think about, a few final concerns when sizing Spark executors:

- The application master, which is a non-executor container with the special capability of requesting containers from YARN, takes up resources of its own that must be budgeted in. In *yarn-client* mode, it defaults to a 1024MB and one vcore. In *yarn-cluster* mode, the application master runs the driver, so it's often useful to bolster its resources with the `--driver-memory` and `--driver-cores` properties.
- Running executors with too much memory often results in excessive garbage collection delays. 64GB is a rough guess at a good upper limit for a single executor.
- I've noticed that the HDFS client has trouble with tons of concurrent threads. A rough guess is that at most five tasks per executor can achieve full write throughput, so it's good to keep the number of cores per executor below that number.
- Running tiny executors (with a single core and just enough memory needed to run a single task, for example) throws away the benefits that come from running multiple tasks in a single JVM. For example, broadcast variables need to be replicated once on each executor, so many small executors will result in many more copies of the data.

To hopefully make all of this a little more concrete, here's a worked example of configuring a Spark app to use as much of the cluster as possible: Imagine a cluster with six nodes running NodeManagers, each equipped with 16 cores and 64GB of memory. The NodeManager capacities, `yarn.nodemanager.resource.memory-mb` and `yarn.nodemanager.resource.cpu-vcores`, should probably be set to 63 * 1024 = 64512 (megabytes) and 15 respectively. We avoid allocating 100% of the resources to YARN containers because the node needs some resources to run the OS and Hadoop daemons. In this case, we leave a gigabyte and a core for these system processes. Cloudera Manager helps by accounting for these and configuring these YARN properties automatically.

The likely first impulse would be to use `--num-executors 6 --executor-cores 15 --executor-memory 63G`. However, this is the wrong approach because:

- 63GB + the executor memory overhead won't fit within the 63GB capacity of the NodeManagers.
- The application master will take up a core on one of the nodes, meaning that there won't be room for a 15-core executor on that node.
- 15 cores per executor can lead to bad HDFS I/O throughput.

A better option would be to use `--num-executors 17 --executor-cores 5 --executor-memory 19G`. Why?

- This config results in three executors on all nodes except for the one with the AM, which will have two executors.
- `--executor-memory` was derived as
  $(63/3 \text{ executors per node}) = 21.\ 21 * 0.07 = 1.47.\ 21 - 1.47 \sim 19.$

Tuning Parallelism

Spark, as you have likely figured out by this point, is a parallel processing engine. What is maybe less obvious is that Spark is not a "magic" parallel processing engine, and is limited in its ability to figure out the optimal amount of parallelism. Every Spark stage has a number of tasks, each of which processes data sequentially. In tuning Spark jobs, this number is probably the single most important parameter in determining performance.

How is this number determined? The way Spark groups RDDs into stages is described in the previous post. (As a quick reminder, transformations like `repartition` and `reduceByKey` induce stage boundaries.) The number of tasks in a stage is the same as the number of partitions in the last RDD in the stage. The number of partitions in an RDD is the same as the number of partitions in the RDD on which it depends, with a couple exceptions: the`coalesce` transformation allows creating an RDD with fewer partitions than its parent RDD, the `union` transformation creates an RDD with the sum of its parents' number of partitions, and `cartesian` creates an RDD with their product.

What about RDDs with no parents? RDDs produced by `textFile` or `hadoopFile` have their partitions determined by the underlying MapReduce InputFormat that's used. Typically there will be a partition for each HDFS block being read. Partitions for RDDs produced by parallelize come from the parameter given by the user, or `spark.default.parallelism` if none is given.

To determine the number of partitions in an RDD, you can always call `rdd.partitions().size()`.

The primary concern is that the number of tasks will be too small. If there are fewer tasks than slots available to run them in, the stage won't be taking advantage of all the CPU available.

A small number of tasks also mean that more memory pressure is placed on any aggregation operations that occur in each task. Any `join`, `cogroup`, or `*ByKey` operation involves holding objects in hashmaps or in-memory buffers to group or sort. `join`, `cogroup`, and `groupByKey` use these data structures in the tasks for the stages that are on the fetching side of the shuffles they trigger. `reduceByKey` and `aggregateByKey` use data structures in the tasks for the stages on both sides of the shuffles they trigger.

When the records destined for these aggregation operations do not easily fit in memory, some mayhem can ensue. First, holding many records in these data structures puts pressure on garbage collection, which can lead to pauses down the line. Second, when the records do not fit in memory, Spark will spill them to disk, which causes disk I/O and sorting. This overhead during large shuffles is probably the number one cause of job stalls I have seen at Cloudera customers.

So how do you increase the number of partitions? If the stage in question is reading from Hadoop, your options are:

- Use the repartition transformation, which will trigger a shuffle.
- Configure your InputFormat to create more splits.
- Write the input data out to HDFS with a smaller block size.

If the stage is getting its input from another stage, the transformation that triggered the stage boundary will accept a `numPartitions` argument, such as

```
val rdd2 = rdd1.reduceByKey(_ + _, numPartitions = X)
```

What should "X" be? The most straightforward way to tune the number of partitions is experimentation: Look at the number of partitions in the parent RDD and then keep multiplying that by 1.5 until performance stops improving.

There is also a more principled way of calculating X, but it's difficult to apply a priori because some of the quantities are difficult to calculate. I'm including it here not because it's recommended for daily use, but because it helps with understanding what's going on. The main goal is to run enough tasks so that the data destined for each task fits in the memory available to that task.

The memory available to each task is:

$$\frac{(\text{spark.executor.memory} * \text{spark.shuffle.memoryFraction} * \text{spark.shuffle.safetyFraction})}{\text{spark.executor.cores}}$$

Memory fraction and safety fraction default to 0.2 and 0.8 respectively.

The in-memory size of the total shuffle data is harder to determine. The closest heuristic is to find the ratio between Shuffle Spill (Memory) metric and the Shuffle Spill (Disk) for a stage that ran. Then multiply the total shuffle write by this number. However, this can be somewhat compounded if the stage is doing a reduction:

$$\frac{(observed\ shuffle\ write) * (observed\ shuffle\ spill\ memory) * (spark.executor.cores)}{(observed\ shuffle\ spill\ disk) * (spark.executor.memory) * (spark.shuffle.memoryFraction) * (spark.shuffle.safetyFraction)}$$

Then round up a bit because too many partitions is usually better than too few partitions.

In fact, when in doubt, it's almost always better to err on the side of a larger number of tasks (and thus partitions). This advice is in contrast to recommendations for MapReduce, which requires you to be more conservative with the number of tasks. The difference stems from the fact that MapReduce has a high startup overhead for tasks, while Spark does not.

Slimming Down Your Data Structures

Data flows through Spark in the form of records. A record has two representations: a deserialized Java object representation and a serialized binary representation. In general, Spark uses the deserialized representation for records in memory and the serialized representation for records stored on disk or being transferred over the network. There is work planned to store some in-memory shuffle data in serialized form.

The `spark.serializer` property controls the serializer that's used to convert between these two representations. The Kryo serializer, `org.apache.spark.serializer.KryoSerializer`, is the preferred option. It is unfortunately not the default, because of some instabilities in Kryo during earlier versions of Spark and a desire not to break compatibility, but the Kryo serializer should *always* be used

The footprint of your records in these two representations has a massive impact on Spark performance. It's worthwhile to review the data types that get passed around and look for places to trim some fat.

Bloated deserialized objects will result in Spark spilling data to disk more often and reduce the number of deserialized records Spark can cache (e.g. at the `MEMORY` storage level). The Spark tuning guide has a great section on slimming these down.

Bloated serialized objects will result in greater disk and network I/O, as well as reduce the number of serialized records Spark can cache (e.g. at the `MEMORY_SER` storage level.) The main action item here is to make sure to register any custom classes you define and pass around using the `SparkConf#registerKryoClasses` API.

Data Formats

Whenever you have the power to make the decision about how data is stored on disk, use an extensible binary format like Avro, Parquet, Thrift, or Protobuf. Pick *one* of these formats and stick to it. To be clear, when one talks about using Avro, Thrift, or Protobuf on Hadoop, they mean that each record is a Avro/Thrift/Protobuf struct stored in a sequence file. JSON is just not worth it.

Every time you consider storing lots of data in JSON, think about the conflicts that will be started in the Middle East, the beautiful rivers that will be dammed in Canada, or the radioactive fallout from the nuclear plants that will be built in the American heartland to power the CPU cycles spent parsing your files over and over and over again. Also, try to learn people skills so that you can convince your peers and superiors to do this, too.

Sandy Ryza is a Data Scientist at Cloudera, an Apache Spark committer, and an Apache Hadoop PMC member. He is a co-author of the O'Reilly Media book, Advanced Analytics with Spark.