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

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)" (http://blog.cloudera.com/blog/2015/03/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 (http://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/).

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 (https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-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:

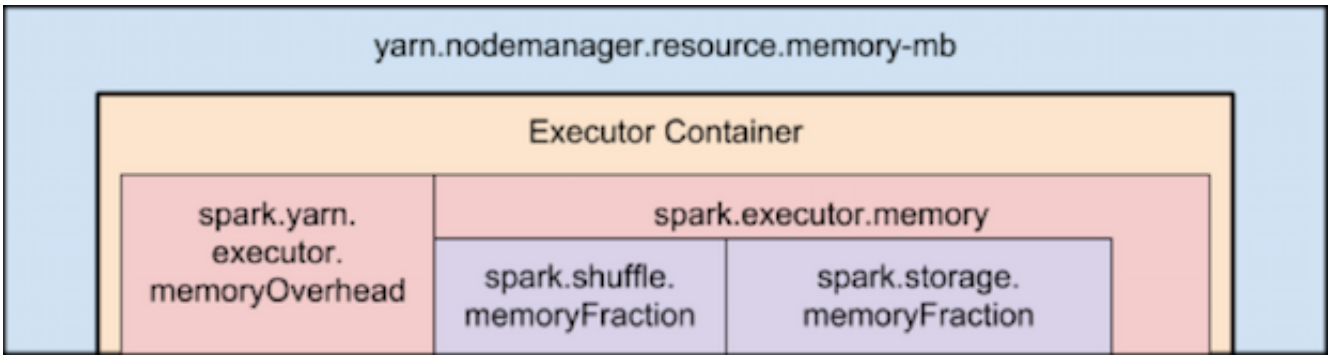
- `yarn.nodemanager.resource.memory-mb` controls the maximum sum of memory used by the containers on each node.
- `yarn.nodemanager.resource.cpu-vcores` controls the maximum sum of cores used by the containers on each node.

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-al-`

`location-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 executors per node) = 21. 21 * 0.07 = 1.47. 21 – 1.47 ~ 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 (http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/). (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 (`spark.executor.memory` * `spark.shuffle.memoryFraction` * `spark.shuffle.safetyFraction`)/`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:

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

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 (https://issues.apache.org/jira/browse/SPARK-4550) planned (https://issues.apache.org/jira/browse/SPARK-2926) 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 (http://spark.apache.org/docs/1.2.0/tuning.html#memory-tuning) 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` (https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.SparkConf) 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 (https://hadoop.apache.org/docs/current/api/org/apache/hadoop/io/SequenceFile.html). 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.
