Because of the in-memory nature of most Spark computations, Spark programs can be bottlenecked by any resource in the cluster: CPU, network bandwidth, or memory. Most often, if the data fits in memory, the bottleneck is network bandwidth, but sometimes, you also need to do some tuning, such as storing RDDs in serialized form, to decrease memory usage. This guide will cover two main topics: data serialization, which is crucial for good network performance and can also reduce memory use, and memory tuning. We also sketch several smaller topics.

Data Serialization

Serialization plays an important role in the performance of any distributed application. Formats that are slow to serialize objects into, or consume a large number of bytes, will greatly slow down the computation. Often, this will be the first thing you should tune to optimize a Spark application. Spark aims to strike a balance between convenience (allowing you to work with any Java type in your operations) and performance. It provides two serialization libraries:

- Java serialization: By default, Spark serializes objects using Java's `ObjectOutputStream` framework, and can work with any class you create that implements `java.io.Serializable`. You can also control the performance of your serialization more closely by extending `java.io.Externalizable`. Java serialization is flexible but often quite slow, and leads to large serialized formats for many classes.
- Kryo serialization: Spark can also use the Kryo library (version 4) to serialize objects more quickly. Kryo is significantly faster and more compact than Java serialization (often as much as 10x), but does not support all `Serializable` types and requires you to *register* the classes you'll use in the program in advance for best performance.

You can switch to using Kryo by initializing your job with a SparkConf and calling `conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")`. This setting configures the serializer used for not only shuffling data between worker nodes but also when serializing RDDs to disk. The only reason Kryo is not the default is because of the custom registration requirement, but we recommend trying it in any network-intensive application. Since Spark 2.0.0, we internally use Kryo serializer when shuffling RDDs with simple types, arrays of simple types, or string type.

Spark automatically includes Kryo serializers for the many commonly-used core Scala classes covered in the AllScalaRegistrar from the Twitter chill library.

To register your own custom classes with Kryo, use the `registerKryoClasses` method.

```
val conf = new SparkConf().setMaster(...).setAppName(...)
conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))
val sc = new SparkContext(conf)
```

The Kryo documentation describes more advanced registration options, such as adding custom serialization code.

If your objects are large, you may also need to increase the `spark.kryoserializer.buffer` config. This value needs to be large enough to hold the *largest* object you will serialize.

Finally, if you don't register your custom classes, Kryo will still work, but it will have to store the full class name with each object, which is wasteful.

Memory Tuning

There are three considerations in tuning memory usage: the *amount* of memory used by your objects (you may want your entire dataset to fit in memory), the *cost* of accessing those objects, and the overhead of *garbage collection* (if you have high turnover in terms of objects).

By default, Java objects are fast to access, but can easily consume a factor of 2-5x more space than the "raw" data inside their fields. This is due to several reasons:

- Each distinct Java object has an "object header", which is about 16 bytes and contains information such as a pointer to its class. For an object with very little data in it (say one `Int` field), this can be bigger than the data.
- Java `String`s have about 40 bytes of overhead over the raw string data (since they store it in an array of `Char`s and keep extra data such as the length), and store each character as *two* bytes due to `String`'s internal usage of UTF-16 encoding. Thus a 10-character string can easily consume 60 bytes.
- Common collection classes, such as `HashMap` and `LinkedList`, use linked data structures, where there is a "wrapper" object for each entry (e.g. `Map.Entry`). This object not only has a header, but also pointers (typically 8 bytes each) to the next object in the list.
- Collections of primitive types often store them as "boxed" objects such as `java.lang.Integer`.

This section will start with an overview of memory management in Spark, then discuss specific strategies the user can take to make more efficient use of memory in his/her application. In particular, we will describe how to determine the memory usage of your objects, and how to improve it – either by changing your data structures, or by storing data in a serialized format. We will then cover tuning Spark's cache size and the Java garbage collector.

Memory Management Overview

Memory usage in Spark largely falls under one of two categories: execution and storage. Execution memory refers to that used for computation in shuffles, joins, sorts and aggregations, while storage memory refers to that used for caching and propagating internal data across the cluster. In Spark, execution and storage share a unified region (M). When no execution memory is used, storage can acquire all the available memory and vice versa. Execution may evict storage if necessary, but only until total storage memory usage falls under a certain threshold (R). In other words, `R` describes a subregion within `M` where cached blocks are never evicted. Storage may not evict execution due to complexities in implementation.

This design ensures several desirable properties. First, applications that do not use caching can use the entire space for execution, obviating unnecessary disk spills. Second, applications that do use caching can reserve a minimum storage space (R) where their data blocks are immune to being evicted. Lastly, this approach provides reasonable out-of-the-box performance for a variety of workloads without requiring user expertise of how memory is divided internally.

Although there are two relevant configurations, the typical user should not need to adjust them as the default values are applicable to most workloads:

- `spark.memory.fraction` expresses the size of `M` as a fraction of the (JVM heap space - 300MiB) (default 0.6). The rest of the space (40%) is reserved for user data structures, internal metadata in Spark, and safeguarding against OOM errors in the case of sparse and unusually large records.
- `spark.memory.storageFraction` expresses the size of `R` as a fraction of `M` (default 0.5). `R` is the storage space within `M` where cached blocks immune to being evicted by execution.

The value of `spark.memory.fraction` should be set in order to fit this amount of heap space comfortably within the JVM's old or "tenured" generation. See the discussion of advanced GC tuning below for details.

Determining Memory Consumption

The best way to size the amount of memory consumption a dataset will require is to create an RDD, put it into cache, and look at the "Storage" page in the web UI. The page will tell you how much memory the RDD is occupying.

To estimate the memory consumption of a particular object, use `SizeEstimator`'s `estimate` method. This is useful for experimenting with different data layouts to trim memory usage, as well as determining the amount of space a broadcast variable will occupy on each executor heap.

Tuning Data Structures

The first way to reduce memory consumption is to avoid the Java features that add overhead, such as pointer-based data structures and wrapper objects. There are several ways to do this:

1. Design your data structures to prefer arrays of objects, and primitive types, instead of the standard Java or Scala collection classes (e.g. `HashMap`). The fastutil library provides convenient collection classes for primitive types that are compatible with the Java standard library.
2. Avoid nested structures with a lot of small objects and pointers when possible.
3. Consider using numeric IDs or enumeration objects instead of strings for keys.
4. If you have less than 32 GiB of RAM, set the JVM flag `-XX:+UseCompressedOops` to make pointers be four bytes instead of eight. You can add these options in `spark-env.sh`.

Serialized RDD Storage

When your objects are still too large to efficiently store despite this tuning, a much simpler way to reduce memory usage is to store them in *serialized* form, using the serialized StorageLevels in the RDD persistence API, such as `MEMORY_ONLY_SER`. Spark will then store each RDD partition as one large byte array. The only downside of storing data in serialized form is slower access times, due to having to deserialize each object on the fly. We highly recommend using Kryo if you want to cache data in serialized form, as it leads to much smaller sizes than Java serialization (and certainly than raw Java objects).

Garbage Collection Tuning

JVM garbage collection can be a problem when you have large "churn" in terms of the RDDs stored by your program. (It is usually not a problem in programs that just read an RDD once and then run many operations on it.) When Java needs to evict old objects to make room for new ones, it will need to trace through all your Java objects and find the unused ones. The main point to remember here is that *the cost of garbage collection is proportional to the number of Java objects*, so using data structures with fewer objects (e.g. an array of `Int`s instead of a `LinkedList`) greatly lowers this cost. An even better method is to persist objects in serialized form, as described above: now there will be only *one* object (a byte array) per RDD partition. Before trying other techniques, the first thing to try if GC is a problem is to use serialized caching.

GC can also be a problem due to interference between your tasks' working memory (the amount of space needed to run the task) and the RDDs cached on your nodes. We will discuss how to control the space allocated to the RDD cache to mitigate this.

Measuring the Impact of GC

The first step in GC tuning is to collect statistics on how frequently garbage collection occurs and the amount of time spent GC. This can be done by adding `-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps` to the Java options. (See the configuration guide for info on passing Java options to Spark jobs.) Next time your Spark job is run, you will see messages printed in the worker's logs each time a garbage collection occurs. Note these logs will be on your cluster's worker nodes (in the `stdout` files in their work directories), *not* on your driver program.

Advanced GC Tuning

To further tune garbage collection, we first need to understand some basic information about memory management in the JVM:

- Java Heap space is divided in to two regions Young and Old. The Young generation is meant to hold short-lived objects while the Old generation is intended for objects with longer lifetimes.
- The Young generation is further divided into three regions [Eden, Survivor1, Survivor2].
- A simplified description of the garbage collection procedure: When Eden is full, a minor GC is run on Eden and objects that are alive from Eden and Survivor1 are copied to Survivor2. The Survivor regions are swapped. If an object is old enough or Survivor2 is full, it is moved to Old. Finally, when Old is close to full, a full GC is invoked.

The goal of GC tuning in Spark is to ensure that only long-lived RDDs are stored in the Old generation and that the Young generation is sufficiently sized to store short-lived objects. This will help avoid full GCs to collect temporary objects created during task execution. Some steps which may be useful are:

- Check if there are too many garbage collections by collecting GC stats. If a full GC is invoked multiple times before a task completes, it means that there isn't enough memory available for executing tasks.
- If there are too many minor collections but not many major GCs, allocating more memory for Eden would help. You can set the size of the Eden to be an over-estimate of how much memory each task will need. If the size of Eden is determined to be `E`, then you can set the size of the Young generation using the option `-Xmn=4/3*E`. (The scaling up by 4/3 is to account for space used by survivor regions as well.)
- In the GC stats that are printed, if the OldGen is close to being full, reduce the amount of memory used for caching by lowering `spark.memory.fraction`; it is better to cache fewer objects than to slow down task execution. Alternatively, consider decreasing the size of the Young generation. This means lowering `-Xmn` if you've set it as above. If not, try changing the value of the JVM's `NewRatio` parameter. Many JVMs default this to 2, meaning that the Old generation occupies 2/3 of the heap. It should be large enough such that this fraction exceeds `spark.memory.fraction`.
- Try the G1GC garbage collector with `-XX:+UseG1GC`. It can improve performance in some situations where garbage collection is a bottleneck. Note that with large executor heap sizes, it may be important to increase the G1 region size with `-XX:G1HeapRegionSize`.
- As an example, if your task is reading data from HDFS, the amount of memory used by the task can be estimated using the size of the data block read from HDFS. Note that the size of a decompressed block is often 2 or 3 times the size of the block. So if we wish to have 3 or 4 tasks' worth of working space, and the HDFS block size is 128 MiB, we can estimate the size of Eden to be `4*3*128MiB`.
- Monitor how the frequency and time taken by garbage collection changes with the new settings.

Our experience suggests that the effect of GC tuning depends on your application and the amount of memory available. There are many more tuning options described online, but at a high level, managing how frequently full GC takes place can help in reducing the overhead.

GC tuning flags for executors can be specified by setting `spark.executor.defaultJavaOptions` or `spark.executor.extraJavaOptions` in a job's configuration.

Other Considerations

Level of Parallelism

Clusters will not be fully utilized unless you set the level of parallelism for each operation high enough. Spark automatically sets the number of "map" tasks to run on each file according to its size (though you can control it through optional parameters to `SparkContext.textFile`, etc), and for distributed "reduce" operations, such as `groupByKey` and `reduceByKey`, it uses the largest parent RDD's number of partitions. You can pass the level of parallelism as a second argument (see the `spark.PairRDDFunctions` documentation), or set the config property `spark.default.parallelism` to change the default. In general, we recommend 2-3 tasks per CPU core in your cluster.

Parallel Listing on Input Paths

Sometimes you may also need to increase directory listing parallelism when job input has large number of directories, otherwise the process could take a very long time, especially when against object store like S3. If your job works on RDD with Hadoop input formats (e.g., via `SparkContext.sequenceFile`), the parallelism is controlled via `spark.hadoop.mapreduce.input.fileinputformat.list-status.num-threads` (currently default is 1).

For Spark SQL with file-based data sources, you can tune `spark.sql.sources.parallelPartitionDiscovery.threshold` and `spark.sql.sources.parallelPartitionDiscovery.parallelism` to improve listing parallelism. Please refer to Spark SQL performance tuning guide for more details.

Memory Usage of Reduce Tasks

Sometimes, you will get an OutOfMemoryError not because your RDDs don't fit in memory, but because the working set of one of your tasks, such as one of the reduce tasks in `groupByKey`, was too large. Spark's shuffle operations (`sortByKey`, `groupByKey`, `reduceByKey`, `join`, etc) build a hash table within each task to perform the grouping, which can often be large. The simplest fix here is to *increase the level of parallelism*, so that each task's input set is smaller. Spark can efficiently support tasks as short as 200 ms, because it reuses one executor JVM across many tasks and it has a low task launching cost, so you can safely increase the level of parallelism to more than the number of cores in your clusters.

Broadcasting Large Variables

Using the broadcast functionality available in `SparkContext` can greatly reduce the size of each serialized task, and the cost of launching a job over a cluster. If your tasks use any large object from the driver program inside of them (e.g. a static lookup table), consider turning it into a broadcast variable. Spark prints the serialized size of each task on the master, so you can look at that to decide whether your tasks are too large; in general, tasks larger than about 20 KiB are probably worth optimizing.

Data Locality

Data locality can have a major impact on the performance of Spark jobs. If data and the code that operates on it are together, then computation tends to be fast. But if code and data are separated, one must move to the other. Typically, it is faster to ship serialized code from place to place than a chunk of data because code size is much smaller than data. Spark builds its scheduling around this general principle of data locality.

Data locality is how close data is to the code processing it. There are several levels of locality based on the data's current location. In order from closest to farthest:

- `PROCESS_LOCAL` data is in the same JVM as the running code. This is the best locality possible.
- `NODE_LOCAL` data is on the same node. Examples might be in HDFS on the same node, or in another executor on the same node. This is a little slower than `PROCESS_LOCAL` because the data has to travel between processes.
- `NO_PREF` data is accessed equally quickly from anywhere and has no locality preference.
- `RACK_LOCAL` data is on the same rack of servers. Data is on a different server on the same rack so needs to be sent over the network, typically through a single switch.
- `ANY` data is elsewhere on the network and not in the same rack.

Spark prefers to schedule all tasks at the best locality level, but this is not always possible. In situations where there is no unprocessed data on any idle executor, Spark switches to lower locality levels. There are two options: a) wait until a busy CPU frees up to start a task on data on the same server, or b) immediately start a new task in a farther away place that requires moving data there.

What Spark typically does is wait a bit in the hopes that a busy CPU frees up. Once that timeout expires, it starts moving the data from far away to the free CPU. The wait timeout for fallback between each level can be configured individually or all together in one parameter; see the `spark.locality` parameters on the configuration page for details. You should increase these settings if your tasks are long and see poor locality, but the default usually works well.

Summary

This has been a short guide to point out the main concerns you should know about when tuning a Spark application – most importantly, data serialization and memory tuning. For most programs, switching to Kryo serialization and persisting data in serialized form will solve most common performance issues. Feel free to ask on the Spark mailing list about other tuning best practices.