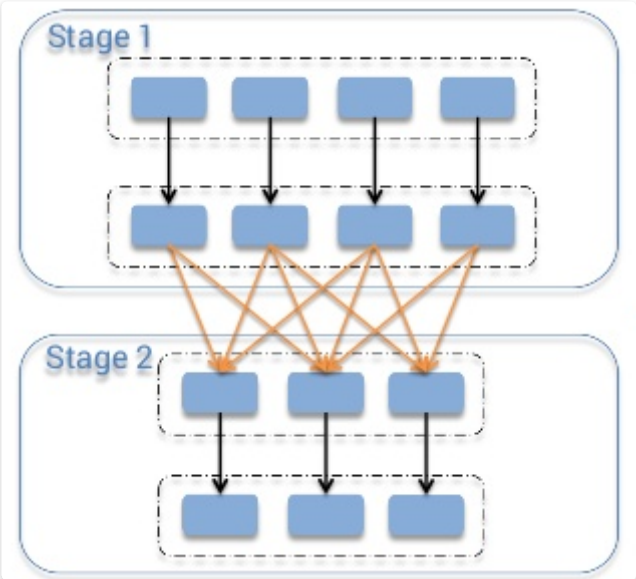
Spark Architecture: Shuffle

41 Replies

This is my second article about Apache Spark architecture and today I will be more specific and tell you about the shuffle, one of the most interesting topics in the overall Spark design. The previous part was mostly about general Spark architecture and its memory management. It can be accessed here. The next one is about Spark memory management and it is available here.



What is the shuffle in general? Imagine that you have a list of phone call detail records in a table and you want to calculate amount of calls happened each day. This way you would set the “day” as your key, and for each record (i.e. for each call) you would emit “1” as a value. After this you would sum up values for each key, which would be an answer to your question – total amount of records for each day. But when you store the data across the cluster, how can you sum up the values for the same key stored on different machines? The only way to do so is to make all the values for the same key be on the same machine, after this you would be able to sum them up.

There are many different tasks that require shuffling of the data across the cluster, for instance table join – to join two tables on the field “id”, you must be sure that all the data for the same values of “id” for both of the tables are stored in the same chunks. Imagine the tables with integer keys ranging from 1 to 1’000’000. By storing the data in same chunks I mean that for instance for both tables values of the key 1-100 are stored in a single partition/chunk, this way instead of going through the whole second table for each partition of the first one, we can join partition with partition directly, because we know that the key values 1-100 are stored only in these two partitions. To achieve this both tables should have the same number of partitions, this way their join would require much less computations. So now you can understand how important shuffling is.

Discussing this topic, I would follow the MapReduce naming convention. In the shuffle operation, the task that emits the data in the source executor is “mapper”, the task that consumes the data into the target executor is “reducer”, and what happens between them is “shuffle”.

Shuffling in general has 2 important compression parameters: ***spark.shuffle.compress*** – whether the engine would compress shuffle outputs or not, and ***spark.shuffle.spill.compress*** – whether to compress intermediate shuffle spill files or not. Both have the value “true” by default, and both would use ***spark.io.compression.codec*** codec for compressing the data, which is snappy by default.

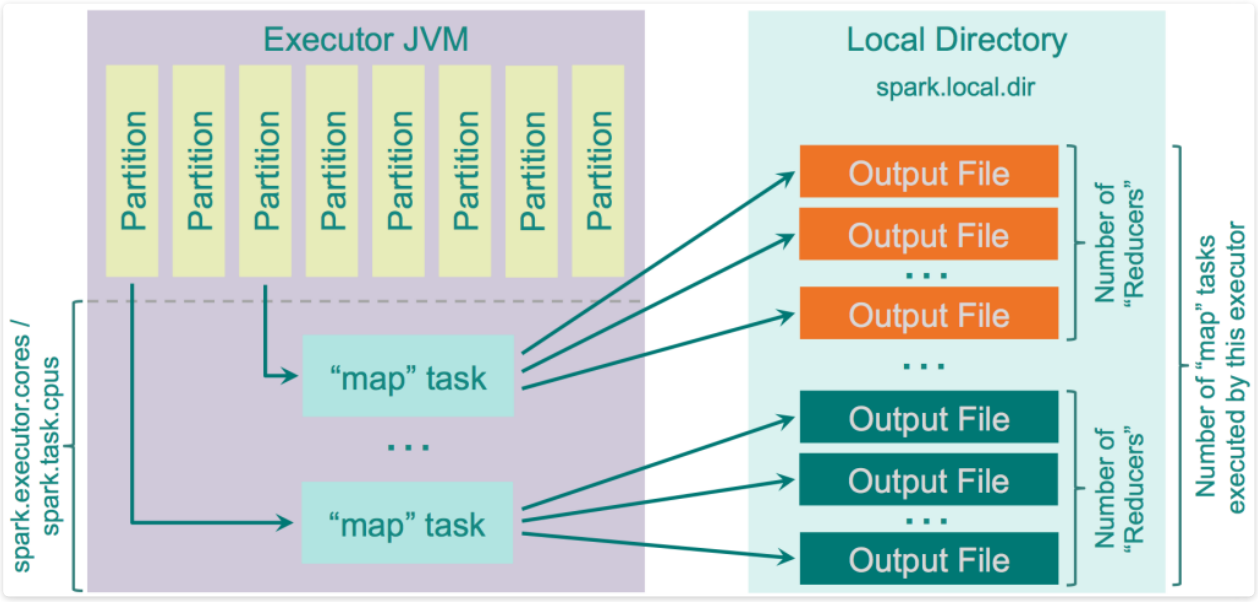
As you might know, there are a number of shuffle implementations available in Spark. Which implementation would be used in your particular case is determined by the value of ***spark.shuffle.manager*** parameter. Three possible options are: hash, sort, tungsten-sort, and the “sort” option is default starting from Spark 1.2.0.

Hash Shuffle

Prior to Spark 1.2.0 this was the default option of shuffle (***spark.shuffle.manager*** = *hash*). But it has many drawbacks, mostly caused by the amount of files it creates – each mapper task creates separate file for each separate reducer, resulting in **M** * **R**total files on the cluster, where **M** is the number of "mappers" and **R** is the number of "reducers". With high amount of mappers and reducers this causes big problems, both with the output buffer size, amount of open files on the filesystem, speed of creating and dropping all these files. Here's a good example of how Yahoo faced all these problems, with 46k mappers and 46k reducers generating 2 billion files on the cluster.

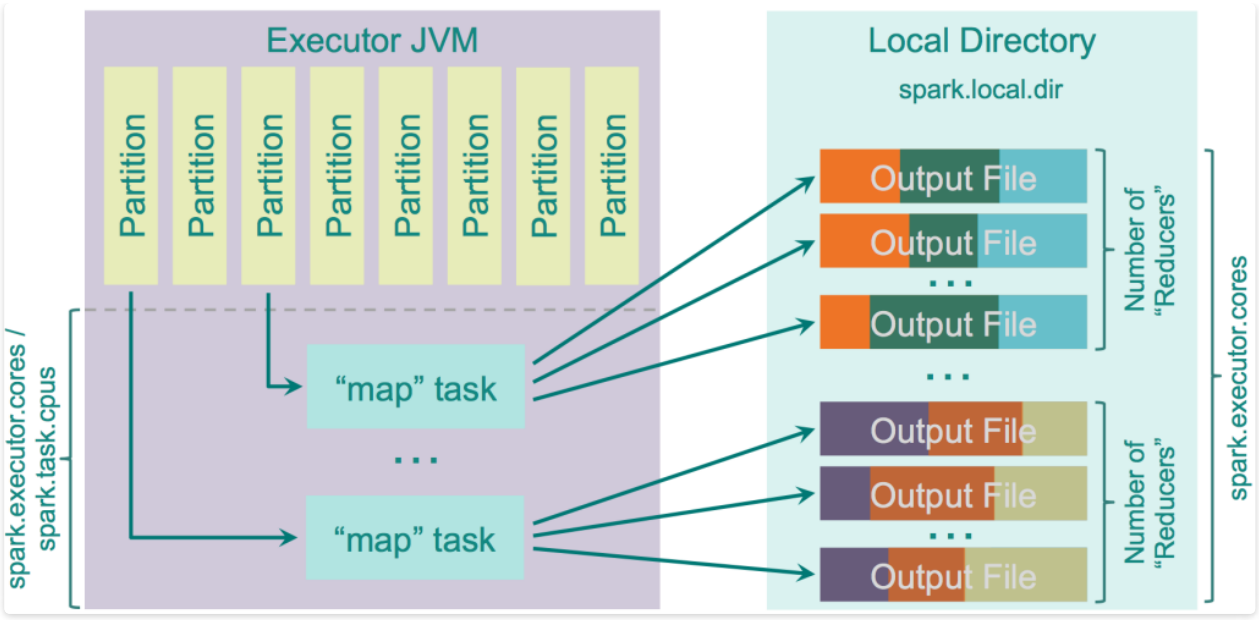
The logic of this shuffler is pretty dumb: it calculates the amount of "reducers" as the amount of partitions on the "reduce" side, creates a separate file for each of them, and looping through the records it needs to output, it calculates target partition for each of them and outputs the record to the corresponding file.

Here is how it looks like:



There is an optimization implemented for this shuffler, controlled by the parameter "***spark.shuffle.consolidateFiles***" (default is "false"). When it is set to "true", the "mapper" output files would be consolidated. If your cluster has **E** executors ("***–num-executors***" for YARN) and each of them has **C** cores ("***spark.executor.cores***" or "***–executor-cores***" for YARN) and each task asks for **T** CPUs ("***spark.task.cpus***"), then the amount of execution slots on the cluster would be **E * C / T**, and the amount of files created during shuffle would be **E * C / T * R**. With 100 executors 10 cores each allocating 1 core for each task and 46000 "reducers" it would allow you to go from 2 billion files down to 46 million files, which is much better in terms of performance. This feature is implemented in a rather straightforward way: instead of creating new file for each of the reducers, it creates a pool of output files. When map task starts outputting the data, it requests a group of **R** files from this pool. When it is finished, it returns this **R** files group back to the pool. As each executor can execute only **C / T** tasks in parallel, it would create only **C / T** groups of output files, each group is of **R**files. After the first **C / T** parallel "map" tasks has finished, each next "map" task would reuse an existing group from this pool.

Here's a general diagram of how it works:



Pros:

1. Fast – no sorting is required at all, no hash table maintained;
2. No memory overhead for sorting the data;
3. No IO overhead – data is written to HDD exactly once and read exactly once.

Cons:

1. When the amount of partitions is big, performance starts to degrade due to big amount of output files
2. Big amount of files written to the filesystem causes IO skew towards random IO, which is in general up to 100x slower than sequential IO

Just for the reference, IO operation slowness at the scale of millions of files on a single filesystem.

And of course, when data is written to files it is serialized and optionally compressed. When it is read, the process is opposite – it is uncompressed and deserialized. Important parameter on the fetch side is "***spark.reducer.maxSizeInFlight***" (48MB by default), which determines the amount of data requested from the remote executors by each reducer. This size is split equally by 5 parallel requests from different executors to speed up the process. If you would increase this size, your reducers would request the data from "map" task outputs in bigger chunks, which would improve performance, but also increase memory usage by "reducer" processes.

If the record order on the reduce side is not enforced, then the "reducer" will just return an iterator with dependency on the "map" outputs, but if the ordering is required it would fetch all the data and sort it on the "reduce" side with ExternalSorter.

Sort Shuffle

Starting Spark 1.2.0, this is the default shuffle algorithm used by Spark (***spark.shuffle.manager*** = *sort*). In general, this is an attempt to implement the shuffle logic similar to the one used by Hadoop MapReduce. With hash shuffle you output one separate file for each of the "reducers", while with sort shuffle you're doing a smarted thing: you output a single file ordered by "reducer" id and indexed, this way you can easily fetch the chunk of the data related to "reducer x" by just getting information about the position of related data block in the file and doing a single fseek before fread. But of course for small amount of "reducers" it is obvious that hashing to separate files would work faster than sorting, so the sort shuffle has a "fallback" plan: when the amount of "reducers" is smaller than "***spark.shuffle.sort.bypassMergeThreshold***" (200 by default) we use the "fallback" plan with hashing the data to separate files and then joining these files together in a single file. This logic is implemented in a separate class BypassMergeSortShuffleWriter.

The funny thing about this implementation is that it sorts the data on the “map” side, but does not merge the results of this sort on “reduce” side – in case the ordering of data is needed it just re-sorts the data. Cloudera has put itself in a fun position with this idea: http://blog.cloudera.com/blog/2015/01/improving-sort-performance-in-apache-spark-its-a-double/. They started a process of implementing the logic that takes advantage of pre-sorted outputs of “mappers” to merge them together on the “reduce” side instead of resorting. As you might know, sorting in Spark on reduce side is done using TimSort, and this is a wonderful sorting algorithm which in fact by itself takes advantage of pre-sorted inputs (by calculating minruns and then merging them together). A bit of math here, you can skip if you’d like to. Complexity of merging **M**sorted arrays of **N** elements each is **O(MNlogM)** when we use the most efficient way to do it, using Min Heap. With TimSort, we make a pass through the data to find MinRuns and then merge them together pair-by-pair. It is obvious that it would identify **M** MinRuns. First **M/2** merges would result in **M/2** sorted groups, next **M/4**merges would give **M/4** sorted groups and so on, so its quite straightforward that the complexity of all these merges would be **O(MNlogM)** in the very end. Same complexity as the direct merge! The difference here is only in constants, and constants depend on implementation. So the patch by Cloudera engineers has been pending on its approval for already one year, and unlikely it would be approved without the push from Cloudera management, because performance impact of this thing is very minimal or even none, you can see this in JIRA ticket discussion. Maybe they would workaround it by introducing separate shuffle implementation instead of “improving” the main one, we’ll see this soon.

Fine with this. What if you don’t have enough memory to store the whole “map” output? You might need to spill intermediate data to the disk. Parameter ***spark.shuffle.spill*** is responsible for enabling/disabling spilling, and by default spilling is enabled. If you would disable it and there is not enough memory to store the “map” output, you would simply get OOM error, so be careful with this.

The amount of memory that can be used for storing “map” outputs before spilling them to disk is “JVM Heap Size” * ***spark.shuffle.memoryFraction*** * ***spark.shuffle.safetyFraction***, with default values it is “JVM Heap Size” * 0.2 * 0.8 = “JVM Heap Size” * 0.16. Be aware that if you run many threads within the same executor (setting the ratio of ***spark.executor.cores / spark.task.cpus*** to more than 1), average memory available for storing “map” output for each task would be “JVM Heap Size” * **spark.shuffle.memoryFraction** * **spark.shuffle.safetyFraction** / **spark.executor.cores * spark.task.cpus**, for 2 cores with other defaults it would give 0.08 * “JVM Heap Size”.

Spark internally uses AppendOnlyMap structure to store the “map” output data in memory. Interestingly, Spark uses their own Scala implementation of hash table that uses open hashing and stores both keys and values in the same array using quadratic probing. As a hash function they use murmur3_32 from Google Guava library, which is MurmurHash3.

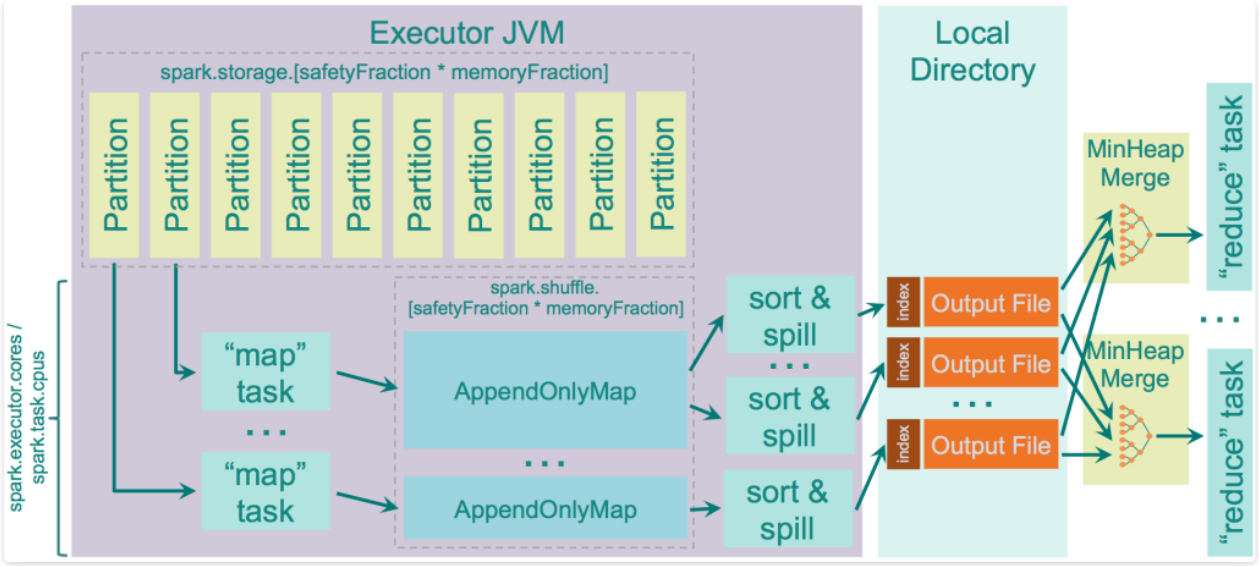
This hash table allows Spark to apply “combiner” logic in place on this table – each new value added for existing key is getting through “combine” logic with existing value, and the output of “combine” is stored as the new value.

When the spilling occurs, it just calls “sorter” on top of the data stored in this AppendOnlyMap, which executes TimSort on top of it, and this data is getting written to disk.

Sorted output is written to the disk when the spilling occurs or when there is no more mapper output, i.e. the data is guaranteed to hit the disk. Whether it will really hit the disk depends on OS settings like file buffer cache, but it is up to OS to decide, Spark just sends it “write” instructions.

Each spill file is written to the disk separately, their merging is performed only when the data is requested by “reducer” and the merging is real-time, i.e. it does not call somewhat “on-disk merger” like it happens in Hadoop MapReduce, it just dynamically collects the data from a number of separate spill files and merges them together using Min Heap implemented by Java PriorityQueue class.

This is how it works:



So regarding this shuffle:

Pros:

1. Smaller amount of files created on “map” side
2. Smaller amount of random IO operations, mostly sequential writes and reads

Cons:

1. Sorting is slower than hashing. It might worth tuning the bypassMergeThreshold parameter for your own cluster to find a sweet spot, but in general for most of the clusters it is even too high with its default
2. In case you use SSD drives for the temporary data of Spark shuffles, hash shuffle might work better for you

Unsafe Shuffle or Tungsten Sort

Can be enabled with setting ***spark.shuffle.manager*** = *tungsten-sort* in Spark 1.4.0+. This code is the part of project “Tungsten”. The idea is described here, and it is pretty interesting. The optimizations implemented in this shuffle are:

1. Operate directly on serialized binary data without the need to deserialize it. It uses unsafe (sun.misc.Unsafe) memory copy functions to directly copy the data itself, which works fine for serialized data as in fact it is just a byte array
2. Uses special cache-efficient sorter ShuffleExternalSorter that sorts arrays of compressed record pointers and partition ids. By using only 8 bytes of space per record in the sorting array, it works more efficienly with CPU cache
3. As the records are not deserialized, spilling of the serialized data is performed directly (no deserialize-compare-serialize-spill logic)
4. Extra spill-merging optimizations are automatically applied when the shuffle compression codec supports concatenation of serialized streams (i.e. to merge separate spilled outputs just concatenate them). This is currently supported by Spark’s LZF serializer, and only if fast merging is enabled by parameter “***shuffle.unsafe.fastMergeEnabled***”

As a next step of optimization, this algorithm would also introduce off-heap storage buffer.

This shuffle implementation would be used only when all of the following conditions hold:

- The shuffle dependency specifies no aggregation. Applying aggregation means the need to store deserialized value to be able to aggregate new incoming values to it. This way you lose the main advantage of this shuffle with its operations on serialized data