Sort-Based Shuffle in Spark

Motivation

A sort-based shuffle can be more scalable than Spark's current hash-based one because it doesn't require writing a separate file for each reduce task from each mapper. Instead, we write a single sorted file and serve ranges of it to different reducers. In jobs with a lot of reduce tasks (say 10,000+), this saves significant memory for compression and serialization buffers and results in more sequential disk I/O.

Implementation

To perform a sort-based shuffle, each map task will produce one or more output files sorted by a key's partition ID, then merge-sort them to yield a single output file. Because it's only necessary to group the keys together into partitions, we won't bother to also sort them within each partition. Operators like sortByKey can do that in the reduce task, as they do in the hash-based shuffle. Once the map tasks produce these files, reducers will be able to request ranges of these files to get their particular data. For this purpose, we'll have an index file for each output file saying where each partition is located, and we'll update the BlockManager to support using this index.

NOTE: If the keys have an Ordering, it might be useful to sort the keys within each range to allow combining across merged files. We may leave this out in the initial implementation and add it in later. In the current codebase, the ShuffleMapTask already combines data using an ExternalAppendOnlyMap, so the elements going into the shuffle phase will have unique keys. However, we can move the merging/combining to happen as part of the sort later.

Shuffle Map Tasks

Map tasks will write data through a SortedFileWriter that creates one or more sorted files, merges them, and then creates an index file for the merged file. Because we need to be able to serve any partition of this file, this SortedFileWriter must reset compression and serialization streams when writing each range. This makes it different from the ExternalAppendOnlyMap, though some of the code will be similar. In addition, to avoid calling our Partitioner on each key multiple times as it gets sorted, we'll make each intermediate file track the start and end location of each partition it contains.

The SortedFileWriter will work as follows:

- Given a stream of incoming key-value pairs, first write them into buckets in memory based on their partition ID. These buckets can just be ArrayBuffers for each partition ID if

we assume input keys are unique (see above), or later on we can have a hashtable to support combining.
- When the total size of the buckets gets too large, write the current in-memory output to a new file. This intermediate file will contain a header saying at which position each partition ID begins. Unlike our final serving use case, these positions can be given by # of objects, and we don't need to reset compression and serialization streams.
- After all the intermediate files are written, merge-sort them into a final file. We can have a maximum merge factor to avoid opening too many files at once, though in practice you probably want to merge at least 10-100 at a time.
- When writing the final file, reset the serialization and compression streams after writing each partition and track the byte positions of each partition to create an index file.

After it runs through all the data, the SortedFileWriter will delete any intermediate files and just leave the final file and its index in the block manager.

NOTE: It may be useful to store intermediate files in the same format as final ones, with a reset before each partition, so that we can merge them without deserializing data. However, this could backfire when the amount of data per partition is small (since we start a new compression and serialization stream). We can investigate this later.

File Format

In both the intermediate and final merged files, it makes sense to use a sparse index that contains (partitionID, startLocation) pairs. This is to avoid having the index take much more space than the data if you have a large number of reduce tasks but few keys. (This is similar to how we avoid requesting zero-length blocks in the current shuffle). For the intermediate files, the index can be at the start of each file, and the locations will be given in # of objects within the file. We can do this because we know the # of objects before starting to write the file, both in the case of merging two files and the case of creating one from in-memory data. For the final file, it's easiest to put the index in a separate file after we finish because we won't know the byte positions for each range until we finish writing them.

File Serving

The BlockManager will need to be modified to serve ranges of a file based on an index. Right now it has some similar code for dealing with consolidated shuffle files: when someone requests a shuffle block, it asks a class called ShuffleBlockManager to get a FileSegment for this block, which may be a segment inside a "real" file managed by the DiskStore. We can do something similar here though it might also be nice to make the BlockManager explicitly aware of indices and of "sub-blocks". (One benefit of that is an easier move to in-memory shuffle.)

Reduce Tasks

Reduce tasks will fetch and hash together data the same way they do now. Because they use ExternalAppendOnlyMap, they already have a sort-based way of spilling to disk if they receive too many values, so we don't need to do anything special for them. Note, however, that the ExternalAppendOnlyMap merges all spilled files at once, so it may not perform well if there are a huge amount of files. We can update it to use a tiered merging policy in a separate patch.