Consolidating Shuffle Files in Spark

Jason Dai (jason.dai@intel.com)

1. Introduction

In Spark, it is common practice (and usually preferred) to launch a large number of small tasks, which unfortunately can create an even larger number of very small shuffle files – one of the major shuffle performance issues.

To address this issue, this change will combine multiple such files (for the same reduce partition or bucket) into one single large shuffle file. One approach to achieve this is simply having different map tasks directly appending to shared shuffle files; however, this can make fault-tolerance very complex (if not impossible), in that:

- A map task could fail before it completes, leading to possible *partial map output*. While it is possible to revert/discard the partial output when a map task fails in writing the output, it can become very complex if the map task fails after it completes writing but before it reports its completion to the master, or when there are fails during the reverting.

- The same map tasks can run multiple times (either due to speculative executions, or re-execution of a previously completed map task in case of failures), which can lead to potential *redundant map output*.

In the following sections, we outline our design that addressees both shuffle file consolidations and the related failure handling.

2. Keeping the functional semantics

The original framework can efficiently support fault tolerance, mostly due to the facts that all the (RDD or map/reduce) operators are functional; i.e., they are stateless with no side effects, so that they can be simply re-tried repeatedly until they are ensured to be successful.

To maintain the functional semantics of shuffle file generation, two levels of guarantees are actually required:

- At slave level, different map tasks write different shuffle files; consequently, partial map output will not impact other shuffle files, and can be simply skipped and discarded.
- At cluster level, a map instance is considered successful only after its output is registered in the master. Consequently, no partial or redundant map shuffle files are possible.

Therefore, to efficiently support fault tolerance, it is critical for the shuffle file consolidation to maintain the functional semantics by providing similar guarantees.

3. Cluster level guarantee

In this change, a new shuffle manager layer is introduced in the cluster, with a shuffle block manager (*ShuffleBlockManager*) running on each slave. Each manager maintains for each shuffle a shuffle block pool (*ShuffleBlocksPool*), which in turn maintains a list of shuffle block group (ShuffleWriterGroup); each group maintains a list of bucket writers (ShuffleBucketWriter), each for a different bucket (or reduce partition). Conceptually, a bucket is a shuffle file containing a list of chunks followed by a list of chunk sizes, as shown in Figure 1.

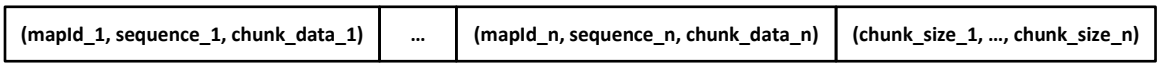

Figure 1: shuffle file format

Each chunk data is a completely encoded (compressed and serialized) output of a specific map task (as identified by *mapId*) for that bucket; *sequence* is a unique (monotonously increasing) sequence number assigned by the shuffle block pool to each map task. Different chunks can be separately located and decoded using the chunk size list.

A map task instance is considered successful only after it successfully registers with the master its output location, which is specified using a tuple *(blcokManagerId, sequence)*. When reading each shuffle file, unregistered chunks can be simply skipped (by checking its sequence), and consequently *no partial or redundant map shuffle files are possible*.

4. Slave level guarantee

On each slave, a map task writes its shuffle output as illustrated in Figure 2.

```
try {
  shuffle =  huffleBlockManager.forShuffle(shuffleId, …)
  group = shuffle.acquireWriters(partition)
  for (elem <- rdd.iterator(split, taskContext)) {
    …
    group.writers(bucketId).write(pair)
  }
  shuffle.commitWrites(group)
  return MapStatus (…)
}
finally {
  shuffle.releaseWriters(group)
}
```

Figure 2: writing map outputs

- When the map task tries to acquire a shuffle block group (*ShuffleWriterGroup*), the shuffle manager will reuse previously created groups (but assign different groups to concurrent map tasks to deal with concurrency).

- The commit of map output is complicated by the fact that a group contains a list of buckets and need to commit the current chunk of each bucket individually. In our implementation, each bucket writer maintains a chunk list (containing the end positions of all successfully committed chunks); we then implement a 2-phase commit by
  (1) Persisting current chunks of all buckets on disk
  (2) Completing current chunks by appending their end positions to respective bucket writers

- To ensure *partial map output can be skipped and discarded*, uncompleted writes need to be reverted; this is achieved by that, whenever a bucket is acquired for writing, it is truncated to the end position of its last completed chunk. Since *this process is idempotent, it can be safely re-tried* in case there are errors.

- When the reduce tasks need to fetch the shuffle file of a bucket, the shuffle manager will first close the bucket writer by (1) reverting uncompleted writes and (2) appending the chunk size list to the file. This is again *an idempotent process and can be safely retried* in case of failures.

5. Multiple segments in a bucket

It is possible that (reduce task) fetching and (map task) writing of the same bucket can happen at the same time, e.g., when a map task need to re-run due to some failures in the reduce stage. In our implementation, each bucket is actually organized as a list of segments, each of which is a shuffle file (as illustrated in Figure 1).

In each bucket, only the latest segment can be written, and previous segments are read-only. In addition, each bucket implements a small state machine as shown in Figure 3.

- A bucket is initially OPEN; when it is acquired for writing, its state is changed to WRITING; after the map task is done writing, its state is changed back to OPEN.
- When the fetcher tries to close the latest segment in the bucket and its state is WRITING, the close is refused with an exception thrown; otherwise, the latest segment in the bucket is closed and the state is changed to CLOSED
- When bucket is first CLOSED and then acquired for writing again, a new segment is generated in the bucket for writing and the bucket restarts from OPEN again.

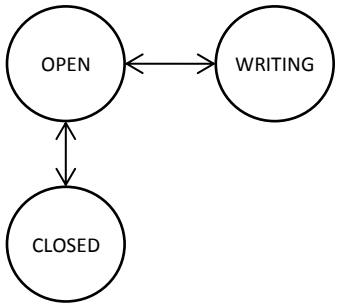

Figure 3: bucket state machine

Since the shuffle fetcher use the shuffle file sizes to control the parallelism of requests, the master needs to maintain the sizes of all segment files of each shuffle. In our implementation, the MapStatus returned by the map task contains

- The BlockManagerId
- The ID of the shuffle block group that it is writing to
- The sequence number it is assigned by the shuffle group
- The current sizes of all the segment files in the shuffle group

As the sequence is monotonously increasing within a group, the master can always maintain the latest segment file sizes.

In future, segments can also be used to break very large shuffle output into smaller files.