Shuffle Internals

NOTE: This Wiki is obsolete as of November 2016 and is retained for reference only.

This page explains how Spark's shuffle works, as of commit 95690a17d328f205c3398b9b477b4072b6fe908f (shortly after the 1.4 release). This explains what happens for a single task; this will happen in parallel for each task running on the machine, and Spark runs up to SPARK_WORKER_CORES (by default, the number of cores on the machine) tasks concurrently on each machine.

The key data structure used in fetching shuffle data is the “results” queue in ShuffleBlockFetcherIterator, which buffers data that we have in serialized (and maybe compressed) form, but haven’t yet deserialized / processed. The results queue is filled by many threads fetching data over the network (the number of concurrent threads fetching data is equal to the number of remote executors we’re currently fetching data from) [0], and is consumed by a single thread that deserializes the data and computes some function over it (e.g., if you’re doing rdd.count(), the thread deserializes the data and counts the number of items). As we fetch data over the network, we track bytesInFlight, which is data that has been requested (and possibly received) from a remote executor, but that hasn’t yet been deserialized / processed by the consumer thread. So, this includes all of the data in the “results” queue, and possibly more data that’s currently outstanding over the network. We always issue as many requests as we can, with the constraint that bytesInFlight remains less than a specified maximum [1].

In a little more detail, here’s exactly what happens when a task begins reading shuffled data:

(1) Issue requests [2] to fetch up to maxBytesInFlight bytes of data [1] over the network (this happens here).

These requests are all executed asynchronously using a ShuffleClient [3] via the shuffleClient.fetchBlocks call [4]. We pass in a callback that, once a block has been successfully fetched, sticks it on the “results” queue.

(2) Begin processing the local data. One by one, we request the local data from the local block manager (which memory maps the file) and then stick the result onto the results queue. Because we memory map the files, which is speedy, the local data typically all ends up on the results queue in front of the remote data.

(3) One the async network requests have been issued (note — issued, but not finished!) and we’ve “read” (memory-mapped) the local data (i.e., (1) and (2) have happened), ShuffleBlockFetcherIterator returns an iterator that gets wrapped too many times to count [5] and eventually gets unrolled [6]. Each time next() is called on the iterator, it blocks waiting for an item from the results queue. This may return right away, or if the queue is empty, will block waiting on new data from the network [6]. Before returning from next(), we update our accounting for the bytes in flight: the chunk of data we return is no longer considered in-flight, because it’s about to be processed, so we update the current bytesInFlight, and if it won’t result in > maxBytesInFlight outstanding, send some more requests for data.

Notes:

[0] Note that these threads consume almost no CPU resources, because they just receive data from the OS and then execute a callback that sticks the data on the results queue.

[1] We limit the data outstanding on the network to avoid using too much memory to hold the data we’ve fetched over the network but haven’t yet processed.

[2] Each request may include multiple shuffle blocks, where is a "block" is the data output for this reduce task by a particular map task. All of the reduce tasks for a shuffle read a total of # map tasks * # reduce tasks shuffle blocks; each reduce task reads # map tasks blocks. We do some hacks to try to size these requests in a "good" way: we limit each request to about maxBytesInFlight / 5, so that we can fetch from roughly 5 machines concurrently without exceeding maxBytesInFlight. 5 is completely a magic number here that was probably guessed by someone long long ago, and it seems to work ok.

[3] The default configuration uses NettyBlockTransferService as the ShuffleClient implementation (note that this extends BlockTransferService, which extends ShuffleClient).

[4] If you’re curious how the shuffle client fetches data, the default Spark configuration results in exactly one TCP connection from an executor to each other executor. If executor A is getting shuffle data from executor B, we start by sending an OpenBlocks message from A to B. The OpenBlocks message includes the list of blocks that A wants to fetch, and causes the remote executor, B, to start to pull the corresponding data into memory from disk (we typically memory map the files, so this may not actually result in the data being read yet), and also to store some state associated with this “stream” of data. The remote executor, B, responds with a stream ID that helps it to identify the connection. Next, A requests blocks one at a time from B using an ChunkFetchRequest message (this happens here in OneForOneBlockFetcher, which calls this code in TransportClient; currently, we have a one-to-one mapping from a chunk to a particular block). It’s possible that there are many sets of shuffle data being fetched concurrently between A and B (e.g., because many tasks are run concurrently). These requests are serialized, so one block is sent at a time from B, and they’re sent in the order that the requests were issued on A.

[5] In BlockStoreShuffleFetcher, which handles failures; then in HashShuffleReader, which helps aggregate some of the data; etc.

[6] This happens in BlockManager.putIterator, if the RDD is going to be cached; in the function passed in to ResultTask, if this is the last stage in a job; or via the writer.write() call in ShuffleMapTask, if this is a stage that generates intermediate shuffle data.

[7] We time how long we spend blocking on data from the network; this is what’s shown as “fetch wait time” in Spark’s UI.