Sort Based Shuffle Read Rough Design

Background

Currently Spark has already integrated sort-based shuffle write, which greatly improve the IO performance and reduce the memory consumption when reducer number is very large. But for the reducer side, it still adopts the implementation of hash-based shuffle reader, which may still has some improvements:

1. In some situations (aggregate in map-side) records have already sorted within each partition in map-side, currently we neglect this ordering and still takes another round of sorting if necessary in reduce-side. We should take this ordering into consideration and avoid unnecessary sorting.

2. Current implementation of sort-based shuffle only adopts **sort-by-partition-and-key** when map-side aggregation is needed, otherwise adopts **sort-by-partition**. In our test, compared to **sort-by-partition**, **sort-by-partition-and-key** brings no obvious performance overhead (one more comparison). So we can adopt **sort-by-partition-and-key** in map-side for some operations (sortByKey), this will greatly ease the processing in reduce-side.

So here we propose a sort-merge like shuffle reader for sort-based shuffle manager. The basic idea is to take within-partition-ordering into consideration, merge the multiple map outputs into a new ordered one without re-sorting.

Implementation

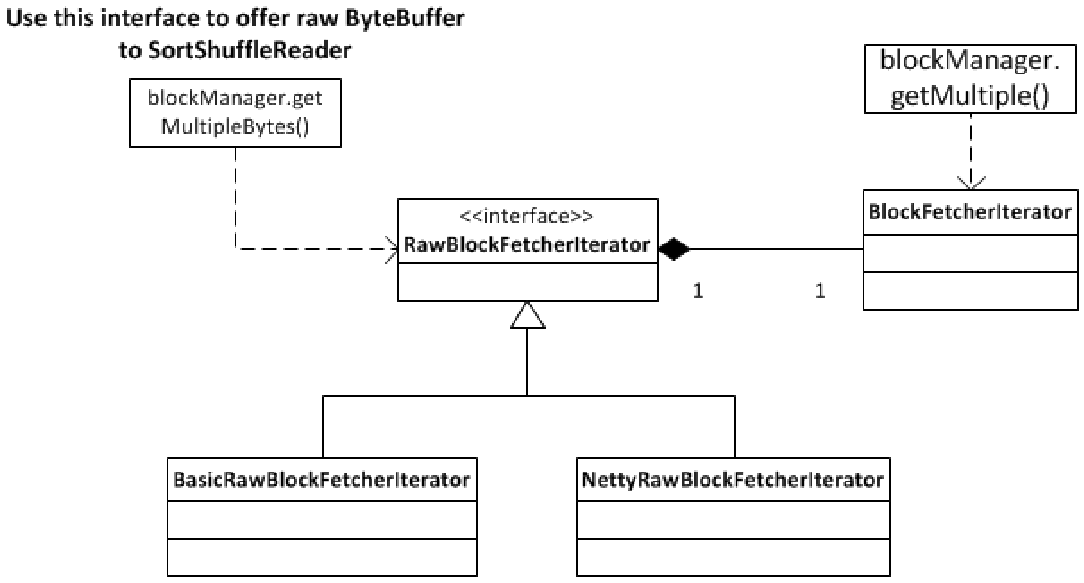
Firstly we should divide Spark’s various shuffle related operations into 3 main parts, and use different solutions for each part:

1. Has Aggregator and map-side-combine is true (reduceByKey):
    - map-side: records sort by partition-and-key.
    - reduce-side: merge the records with same key and aggregate them using Aggregator’s defined `mergeCombiners()` function.

2. Has Aggregator, but map-side-combine is false (groupByKey):
    - map-side: records sort by partition or directly writes to disk (small reducer number).
    - reduce-side: use Aggregator to do `combineValuesByKey()`

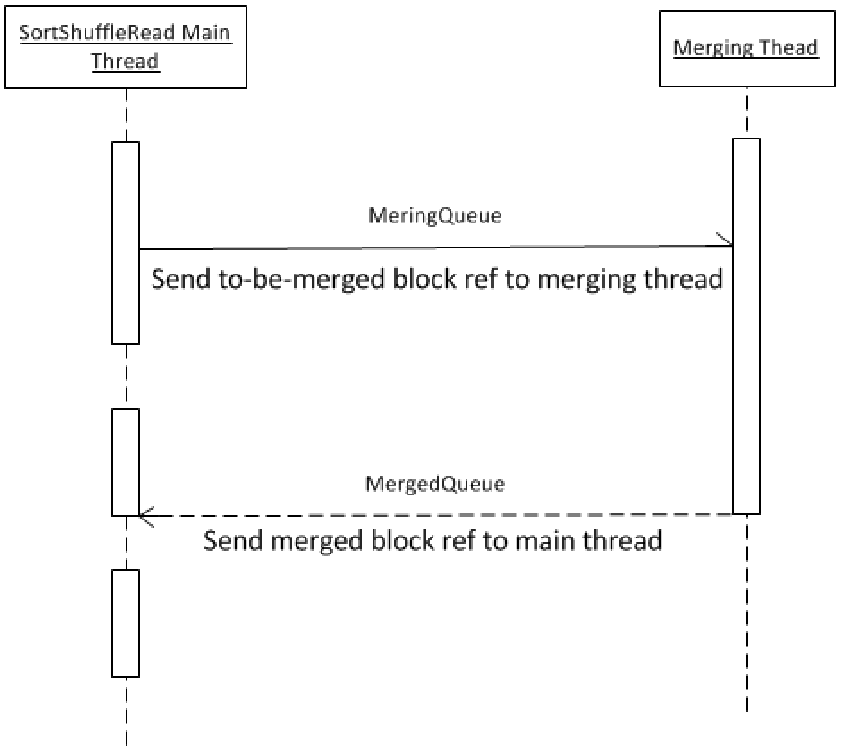
3. No Aggregator, has KeyOrdering (sortByKey):
    - map-side: records sort by partition-and-key.
    - reduce-side: merge the records with same key.

BlockFetcherIterator modification to get raw ByteBuffer iterator


Use this interface to offer raw ByteBuffer to SortShuffleReader
blockManager.getMultipleBytes()
<<interface>> RawBlockFetcherIterator
blockManager.getMultiple()
BlockFetcherIterator
1
1
BasicRawBlockFetcherIterator
NettyRawBlockFetcherIterator

The above UML chart is the basic chart of BlockFetcherIterator related classes, we will modify the existed `BasicBlockFetcherIterator` and `NettyBlockFetcherIterator` to get raw ByteBuffer iterator as `Iterator[(BlockId, Option[ByteBuffer])]`, also wrap it to get the deserialized iterator, which is the same as previous code path.

SortShuffleReader


SortShuffleRead Main Thread
Merging Thead
MeringQueue
Send to-be-merged block ref to merging thread
MergedQueue
Send merged block ref to main thread

Basic work flow is like this:

1. Main thread will fetch map out raw data and store it into memory or local disk (according to shuffle memory fraction), then it will push this block ref into MergingQueue for another thread to merge.
2. Merging thread is created from main thread and used for merging the data from MergingQueue and push the merged data into MergedQueue.
3. Main thread will wait until the merged block is under specified threshold, and then do the merge-sort and combine (if needed), then offer the sorted iterator to the downstream operators.

The merging threshold is controlled by the configuration `spark.shuffle.ioSortFactor`, default is 100, basic idea is the same as MapReduce's `io.sort.factor`, the control principle is to do as less merge as possible to meet the threshold.

SortShuffleWriter

Modify to use **sort-by-partition-and-key** when `keyOrdering` is defined.