Shuffle Read

无论是Hash Based Shuffle或者是Sort Based Shuffle，内置的Reader都是org.apache.spark.shuffle.hash.BlockStoreShuffleReader，

SortShuffleManager的getReader方法如下：

1. override def getReader[K, C](

2. handle: ShuffleHandle,

3. startPartition: Int,

4. endPartition: Int,

5. context: TaskContext): ShuffleReader[K, C] = {

6. new BlockStoreShuffleReader(

7. handle.asInstanceOf[BaseShuffleHandle[K, \_, C]], startPartition, endPartition, context)

8. }

HashShuffleManager中的getReader方法如下:

1. override def getReader[K, C](
2. handle: ShuffleHandle,
3. startPartition: Int,
4. endPartition: Int,
5. context: TaskContext): ShuffleReader[K, C] = {
6. new BlockStoreShuffleReader(
7. handle.asInstanceOf[BaseShuffleHandle[K, \_, C]], startPartition, endPartition, context)
8. }

总上可以看出都是通过创建BlockStoreShuffleReader对象来读取计算结果。

BlockStoreShuffleReader是用来读取上游任务计算结果，它是通过read方法实现。

1. /\*\* Read the combined key-values for this reduce task \*/
2. override def read(): Iterator[Product2[K, C]] = {
3. val blockFetcherItr = new ShuffleBlockFetcherIterator(
4. context,
5. blockManager.shuffleClient,
6. blockManager,
7. mapOutputTracker.getMapSizesByExecutorId(handle.shuffleId, startPartition, endPartition),
8. // Note: we use getSizeAsMb when no suffix is provided for backwards compatibility
9. SparkEnv.get.conf.getSizeAsMb("spark.reducer.maxSizeInFlight", "48m") \* 1024 \* 1024)
10. // Wrap the streams for compression based on configuration
11. val wrappedStreams = blockFetcherItr.map { case (blockId, inputStream) =>
12. blockManager.wrapForCompression(blockId, inputStream)
13. }
14. val ser = Serializer.getSerializer(dep.serializer)
15. val serializerInstance = ser.newInstance()
16. // Create a key/value iterator for each stream
17. val recordIter = wrappedStreams.flatMap { wrappedStream =>
18. // Note: the asKeyValueIterator below wraps a key/value iterator inside of a
19. // NextIterator. The NextIterator makes sure that close() is called on the
20. // underlying InputStream when all records have been read.
21. serializerInstance.deserializeStream(wrappedStream).asKeyValueIterator
22. }
23. // Update the context task metrics for each record read.
24. val readMetrics = context.taskMetrics.createShuffleReadMetricsForDependency()
25. val metricIter = CompletionIterator[(Any, Any), Iterator[(Any, Any)]](
26. recordIter.map(record => {
27. readMetrics.incRecordsRead(1)
28. record
29. }),
30. context.taskMetrics().updateShuffleReadMetrics())
31. // An interruptible iterator must be used here in order to support task cancellation
32. val interruptibleIter = new InterruptibleIterator[(Any, Any)](context, metricIter)
33. val aggregatedIter: Iterator[Product2[K, C]] = if (dep.aggregator.isDefined) {
34. if (dep.mapSideCombine) {
35. // We are reading values that are already combined
36. val combinedKeyValuesIterator = interruptibleIter.asInstanceOf[Iterator[(K, C)]]
37. dep.aggregator.get.combineCombinersByKey(combinedKeyValuesIterator, context)
38. } else {
39. // We don't know the value type, but also don't care -- the dependency \*should\*
40. // have made sure its compatible w/ this aggregator, which will convert the value
41. // type to the combined type C
42. val keyValuesIterator = interruptibleIter.asInstanceOf[Iterator[(K, Nothing)]]
43. dep.aggregator.get.combineValuesByKey(keyValuesIterator, context)
44. }
45. } else {
46. require(!dep.mapSideCombine, "Map-side combine without Aggregator specified!")
47. interruptibleIter.asInstanceOf[Iterator[Product2[K, C]]]
48. }
49. // Sort the output if there is a sort ordering defined.
50. dep.keyOrdering match {
51. case Some(keyOrd: Ordering[K]) =>
52. // Create an ExternalSorter to sort the data. Note that if spark.shuffle.spill is disabled,
53. // the ExternalSorter won't spill to disk.
54. val sorter =
55. new ExternalSorter[K, C, C](context, ordering = Some(keyOrd), serializer = Some(ser))
56. sorter.insertAll(aggregatedIter)
57. context.taskMetrics().incMemoryBytesSpilled(sorter.memoryBytesSpilled)
58. context.taskMetrics().incDiskBytesSpilled(sorter.diskBytesSpilled)
59. context.internalMetricsToAccumulators(
60. InternalAccumulator.PEAK\_EXECUTION\_MEMORY).add(sorter.peakMemoryUsedBytes)
61. CompletionIterator[Product2[K, C], Iterator[Product2[K, C]]](sorter.iterator, sorter.stop())
62. case None =>
63. aggregatedIter
64. }
65. }