Spark Shuffle

Spark Shuffle的处理方式是一个迭代的过程，初始时使用基于哈希的Shuffle写操作，避免Hadoop的多余排序（在Reduce之前获取的数据经过排序），但是这种方式在Map和Reduce数量较大的情况下会导致写文件数量大和缓存开销过大的问题，为了解决这个问题，在Spark 1.2版本中把默认Shuffle写替换为基于排序的Shuffle写，该操作中会把所有的结果写到一个文件中，同时生成一个索引文件进行定位。

基于排序的Shuffle写操作的核心类是SortShuffleWriter，其write方法中，先判断Shuffle Map Task输出结果在Map端是否需要合并（combile），如果需要合并，则外部排序中进行聚合并排序；如果不需要，则外部排序中不进行聚合和排序，例如sortByKey操作在Reduce端会进行聚合并排序。确认外部排序方式后，在外部排序中将使用PartitionedAppendOnlyMap来存放数据，当排序中的Map占用的内存已经超越了阈值，则将Map中的内容溢写到磁盘中，每一次溢写操作产生一个不同的文件。当所有数据处理完成后，在外部排序中有可能一部分计算结果在内存中，另外一部分结果溢写到一个或者多个文件中，这时通过merge操作将内存和spill文件中的内容合并到一个文件中，写操作图如下所示：



# 1.Shuffle写操作

SortShuffleWriter中的write代码如下：

*override def write(records: Iterator[Product2[K, V]]): Unit = {*

*//获取Shuffle Map Task输出结果的排序方式*

*sorter = if (dep.mapSideCombine) {*

*//当输出结果需要Combile，那么外部排序算法中进行聚合*

*require(dep.aggregator.isDefined, "Map-side combine without Aggregator specified!")*

*new ExternalSorter[K, V, C](*

*context, dep.aggregator, Some(dep.partitioner), dep.keyOrdering, dep.serializer)*

*} else {*

*//其他情况下，外部排序算法不进行聚合*

*new ExternalSorter[K, V, V](*

*context, aggregator = None, Some(dep.partitioner), ordering = None, dep.serializer)*

*}*

*//根据获取的排序方式，对数据进行排序并写入内存缓冲区中，如果排序中Map占用的内存已经超过*

*//阈值，则将map中的内容溢写到磁盘中，每一次溢写会产生一个不同的文件*

*sorter.insertAll(records)*

*//通过shuffle编号和Map编号获取该数据文件*

*val output = shuffleBlockResolver.getDataFile(dep.shuffleId, mapId)*

*val tmp = Utils.tempFileWith(output)*

*try {*

*val blockId = ShuffleBlockId(dep.shuffleId, mapId, IndexShuffleBlockResolver.NOOP\_REDUCE\_ID)*

*//在外部排序中有可能一部分计算结果在内存中，另一部分计算结果溢写到一或多个文件中*

*//这时通过merge操作将内存和spill文件中的内容合并整到一个文件中*

*val partitionLengths = sorter.writePartitionedFile(blockId, tmp)*

*//创建索引文件，将每个partition的在数据文件中的起始位置和结束位置写到索引文件中*

*shuffleBlockResolver.writeIndexFileAndCommit(dep.shuffleId, mapId, partitionLengths, tmp)*

*//将元数据信息写入到MapStatus中，后续的任务可以通过该MapStatus得到处理结果信息*

*mapStatus = MapStatus(blockManager.shuffleServerId, partitionLengths)*

*} finally {*

*if (tmp.exists() && !tmp.delete()) {*

*logError(s"Error while deleting temp file ${tmp.getAbsolutePath}")*

*}*

*}*

*}*

在ExternalSorter的insertAll方法中，先判断是否需要进行聚合（Aggregation），如果需要，则根据键值进行合并（Combile），然后把这些数据写到内存缓冲区中，如果排序中的Map占用的内存已经超越了使用的阈值，则将Map中的内容溢写到磁盘中，每一次溢写会产生不同的文件。如果不需要聚合，则直接把数据写到内存缓冲区中。

*def insertAll(records: Iterator[Product2[K, V]]): Unit = {*

*// 获取外部排序中是否需要进行聚合（Aggregation）*

*val shouldCombine = aggregator.isDefined*

*if (shouldCombine) {*

*// 如果需要聚合，则使用AppendOnlyMap根据键值进行合并*

*val mergeValue = aggregator.get.mergeValue*

*val createCombiner = aggregator.get.createCombiner*

*var kv: Product2[K, V] = null*

*val update = (hadValue: Boolean, oldValue: C) => {*

*if (hadValue) mergeValue(oldValue, kv.\_2) else createCombiner(kv.\_2)*

*}*

*while (records.hasNext) {*

*addElementsRead()*

*kv = records.next()*

*map.changeValue((getPartition(kv.\_1), kv.\_1), update)*

*//对数据进行排序并写入到内存缓冲区中，如果排序中的Map占用内存已经超过使用的阈值*

*//则将Map中的内容写到磁盘中，每一次溢写产生一个不同的文件*

*maybeSpillCollection(usingMap = true)*

*}*

*} else {*

*// 不需要聚合（Aggregation），对数据进行排序并写到内存缓冲区中*

*while (records.hasNext) {*

*addElementsRead()*

*val kv = records.next()*

*buffer.insert(getPartition(kv.\_1), kv.\_1, kv.\_2.asInstanceOf[C])*

*maybeSpillCollection(usingMap = false)*

*}*

*}*

*}*

# 2.Shuffle读操作

在下游调度阶段执行时，需要读取Map阶段的输出数据，需要解决两个问题：

1. Shuffle写有基于哈希和排序两种方式
2. 如果确认下游任务读取数据的位置信息，位置信息包括所在节点、Executor编号和读取数据块序列等

下图是Shuffle读流程：



1. 在SparkEnv启动时，会对ShuffleManager、BlockManager和MapOutputTracker等实例化。ShuffleManager配置项有HashShuffleManager和SortShuffleManager和自定义的ShuffleManager等3中选项，前两种在Shuffle读中均实例化BlockStoreShuffleReader，但是在HashShuffleManager中所持有的是FileShuffleBlockResolver实例，SortShuffleManager中持有的是IndexShuffleBlockResolver实例。
2. 在BlockStoreShuffleReader的read方法中，调用MapOutputTracker的getMapSizesByExecutorId方法，由Executor的MapOutputTracker发送获取结果状态GetMapOutputStatus消息给Driver端的MapoutputTrackerMaste，请求获取上游Shuffle输出结果对应的MapStatus，在该MapStatus存放结果数据的位置信息。
3. 知道Shuffle结果的位置信息后，对这些位置进行筛选，判断当前运行的数据是从本地还是从远程节点获取。如果是本地获取，直接调用BlockManager的getBlockData方法，在读取数据的时候会根据写入方式采取不同ShuffleBlockResolver读取；如果是在远程节点上，需要通过Netty网络方式读取数据，在远程读取的过程中使用多线程的方式进行读取，一般来说，会启动5个线程到5个节点进行读取所有，每次请求的数据大小不会超过系统设置的1/5，该大小由spark.reducer.maxSizeInFligth配置项进行设置，默认情况该配置为48MB
4. 读取数据后，判断ShuffleDependency是否定义聚合（Aggregation），如果需要，则根据键值进行聚合。如果上游ShuffleMapTask已经做了合并，则在合并数据的基础上进行键值聚合。待数据处理完毕后，使用外部排序（ExeternalSorter）对数据进行排序并放入存储中，至此完成Shuffle读数据操作。

从代码分析Shuffle读的实现，其类调用关系如下图所示：



1. Shuffle读的起始点是由ShuffledRDD.compute发起的，在该方法中会调用ShuffleManager的getReader方法，基于哈希和Shuffle读都是使用BlockStoreShuffleReader的read方法：

*override def compute(split: Partition, context: TaskContext): Iterator[(K, C)] = {*

*val dep = dependencies.head.asInstanceOf[ShuffleDependency[K, V, C]]*

*SparkEnv.get.shuffleManager.getReader(dep.shuffleHandle, split.index, split.index + 1, context)*

*.read()*

*.asInstanceOf[Iterator[(K, C)]]*

*}*

1. 在BlockStoreShuffleReader的read方法中先实例化ShuffleBlockFetcherIterator，在该实例化过程中，通过MapOutputTracker的getMapSizeByExecutorId获取上游ShuffleMapTask输出数据的元数据。先尝试在本地的mapStatuses获取，如果获取不到，则通过RPC通信框架发送给MapOutputTrackerMaster，请求获取该ShuffleMapTask输出数据的元数据，获取这些元数据转换成Seq[(BlockManagerId,Seq[BlockId,Long])]。在这个序列中的元素包括两部分信息，BlockManagerId可以定位数据所处的Executor，而Seq[(BlockId,Long)]可以定位Executor的数据块编号和获取数据的大小

*override def read(): Iterator[Product2[K, C]] = {*

*val wrappedStreams = new ShuffleBlockFetcherIterator(*

*context,*

*blockManager.shuffleClient,*

*blockManager,*

*//通过消息发送获取ShuffleMapTask存储数据位置的元数据*

*mapOutputTracker.getMapSizesByExecutorId(handle.shuffleId, startPartition, endPartition),*

*serializerManager.wrapStream,*

*//远程获取数据时，每次传输数据设置大小*

*SparkEnv.get.conf.getSizeAsMb("spark.reducer.maxSizeInFlight", "48m") \* 1024 \* 1024,*

*......*

*}*

在MapOutputTracker的getMapSizesByExecutorId方法代码如下：

*override def getMapSizesByExecutorId(shuffleId: Int, startPartition: Int, endPartition: Int)*

*: Seq[(BlockManagerId, Seq[(BlockId, Long)])] = {*

*//通过ShuffleId获取上游ShuffleMapTask输出数据的元数据*

*val statuses = getStatuses(shuffleId)*

*try {*

*MapOutputTracker.convertMapStatuses(shuffleId, startPartition, endPartition, statuses)*

*} catch {}}*

1. 获取读取数据的位置信息后，返回到ShuffleBlockFetcherIterator的initialize方法，该方法是Shuffle读的核心代码所在。在该方法中先通过调用splitLocalRemoteBlocks方法获取的数据位置信息进行区分，判断数据所处的位置是本地节点还是远程节点。如果是远程节点使用fetchUpToMaxBytes方法，从远程节点获取数据；如果是本地节点使用fetchLocalBlocks方法获取数据。

*private[this] def initialize(): Unit = {*

*context.addTaskCompletionListener(\_ => cleanup())*

*// 对获取的数据位置的元数据进行区分，区分本地节点还是远程节点*

*val remoteRequests = splitLocalRemoteBlocks()*

*fetchRequests ++= Utils.randomize(remoteRequests)*

*//对于远程节点的数据，使用Netty网络方式读取*

*fetchUpToMaxBytes()*

*val numFetches = remoteRequests.size - fetchRequests.size*

*logInfo("Started " + numFetches + " remote fetches in" + Utils.getUsedTimeMs(startTime))*

*//对于本地节点的数据，不同的写入方式，采取的读取方式不同，基于Shuffle写使用的是FileShuffleBlockResolver*

*//getBlockData方式获取数据，而基于Shuffle写使用的是IndexShuffleBlockResolver的getBlockData方法获取数据*

*fetchLocalBlocks()*

*}*

1. 数据读取完毕后，回到BlockStoreShuffleReader的read方法，判断是否聚合，如果需要，根据键值调用Aggregator的combileCombinersByKey方法进行聚合。聚合完毕后，使用外部排序（ExternalSorter）对数据进行排序并放入内存中
2. 读取数据后，判断ShuffleDependency是否定义聚合（Aggregation），如果需要，根据根据键值进行聚合，如果在上游ShuffleMapTask已经做了合并，则在合并数据的基础上做键值聚合。待数据处理完毕后，使用ExernalSorter(外部排序)的insertAll方法对数据进行排序，该操作和Shuffle写排序是类似的。