**一：为什么需要Sort-Based Shuffle?**

1. Shuffle一般包含两个阶段任务：

**第一部分：**产生Shuffle数据的阶段(Map阶段，额外补充，需要实现ShuffleManager中的getWriter来写数据(数据可以通过BlockManager写到Memory，Disk，Tachyon等，例如想非常快的Shuffle，此时可以考虑把数据写在内存中，但是内存不稳定，所以可以考虑增加副本。建议采用MEMONY\_AND\_DISK方式)；

**第二部分：**使用Shuffle数据的阶段(Reduce阶段，额外补充，Shuffle读数据：需要实现ShuffleManager的getReader，Reader会向Driver去获取上一个Stage产生的Shuffle数据)。

1. Spark的Job会被划分成很多Stage:

如果只要一个Stage，则这个Job就相当于只有一个Mapper段，当然不会产生Shuffle，适合于简单的ETL。如果不止一个Stage，则最后一个Stage就是最终的Reducer，最左侧的第一个Stage就仅仅是整个Job的Mapper，中间所有的任意一个Stage是其父Stage的Reducer且是其子Stage的Mapper；

**二：****Hash-based Shuffle**

1. Spark Shuffle在最开始的时候只支持Hash-based Shuffle:默认Mapper阶段会为Reducer阶段的每一个Task单独创建一个文件来保存该Task中要使用的数据。

**优点**：就是操作数据简单。

**缺点**：但是在一些情况下(例如数据量非常大的情况)会造成大量文件(M\*R,其中M代表Mapper中的所有的并行任务数量，R代表Reducer中所有的并行任务数据)大数据的随机磁盘I/O操作且会形成大量的Memory(极易造成OOM)。

2，**Hash-based Shuffle产生的问题:**

第一：不能够处理大规模的数据

第二：Spark不能够运行在大规模的分布式集群上！

3，**Consolidate机制:**

后来的改善是加入了Consolidate机制来将Shuffle时候产生的文件数量减少到C\*R个(C代表在Mapper端，同时能够使用的cores数量，R代表Reducer中所有的并行任务数量)。但是此时如果Reducer端的并行数据分片过多的话则C\*R可能已经过大，此时依旧没有逃脱文件打开过多的厄运！！！Consolidate并没有降低并行度，只是降低了临时文件的数量，此时Mapper端的内存消耗就会变少，所以OOM也就会降低，另外一方面磁盘的性能也会变得更好。

**Spark在引入Sort-Based Shuffle之前，适合中小型数据规模的大数据处理！**

**三：****Sort-Based Shuffle**

1， 为了让Spark在更大规模的集群上更高性能处理更大规模的数据，于是就引入了Sort-based Shuffle!从此以后(Spark1.1版本开始)，Spark可以胜任任何规模(包括PB级别及PB以上的级别)的大数据的处理，尤其是钨丝计划的引入和优化，Spark更快速的在更大规模的集群处理更海量的数据的能力推向了一个新的巅峰！

2， Spark1.6版本支持最少三种类型Shuffle：

// Let the user specify short names for shuffle managers  
**val** shortShuffleMgrNames = *Map*(  
 "hash" -> "org.apache.spark.shuffle.hash.HashShuffleManager",  
 "sort" -> "org.apache.spark.shuffle.sort.SortShuffleManager",  
 "tungsten-sort" -> "org.apache.spark.shuffle.sort.SortShuffleManager")  
**val** shuffleMgrName = conf.get("spark.shuffle.manager", "sort")  
**val** shuffleMgrClass = shortShuffleMgrNames.getOrElse(shuffleMgrName.toLowerCase, shuffleMgrName)  
**val** shuffleManager = instantiateClass[ShuffleManager](shuffleMgrClass)

实现ShuffleManager接口可以根据自己的业务实际需要最优化的使用自定义的Shuffle实现；

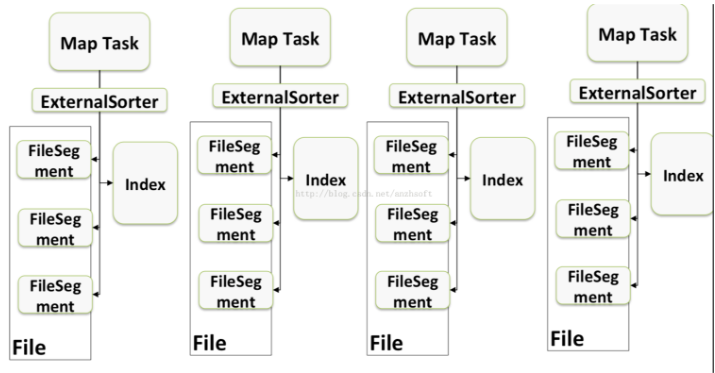
3，Spark1.6默认采用的就是Sort-based Shuffle的方式：

**val** shuffleMgrName = conf.get("spark.shuffle.manager", "sort")

上述源码说明，你可以在Spark配置文件中配置Spark框架运行时要使用的具体的ShuffleManager的实现。可以在conf/spark-default.conf加入如下内容：

spark.shuffle.manager SORT 配置Shuffle方式是SORT

1. **Sort-based Shuffle的工作方式如下**：Shuffle的目的就是：数据分类，然后数据聚集



1. 首先每个ShuffleMapTask不会为每个Reducer单独生成一个文件，相反，Sort-based Shuffle会把Mapper中每个ShuffleMapTask所有的输出数据Data只写到一个文件中。因为每个ShuffleMapTask中的数据会被分类，所以Sort-based Shuffle使用了index文件存储具体ShuffleMapTask输出数据在同一个Data文件中是如何分类的信息！！
2. 基于Sort-base的Shuffle会在Mapper中的每一个ShuffleMapTask中产生两个文件：Data文件和Index文件，其中Data文件是存储当前Task的Shuffle输出的。而index文件中则存储了Data文件中的数据通过Partitioner的分类信息，此时下一个阶段的Stage中的Task就是根据这个Index文件获取自己所要抓取的上一个Stage中的ShuffleMapTask产生的数据的，Reducer就是根据index文件来获取属于自己的数据。

涉及问题：Sorted-based Shuffle：会产生 2\*M(M代表了Mapper阶段中并行的Partition的总数量，其实就是ShuffleMapTask的总数量)个Shuffle临时文件。

Shuffle产生的临时文件的数量的变化一次为：

**Basic Hash Shuffle: M\*R;**

**Consalidate方式的Hash Shuffle: C\*R;**

**Sort-based Shuffle: 2\*M;**

**四：在集群中动手实战Sort-based Shuffle**

在Sorted-based Shuffle中Reducer是如何获取自己需要的数据呢？具体而言，Reducer首先找Driver去获取父Stage中的ShuffleMapTask输出的位置信息，根据位置信息获取index文件，解析index，从解析的index文件中获取Data文件中属于自己的那部分内容；

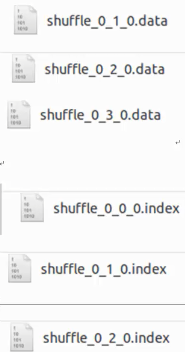
Sorted-based Shuffle与排序没有关系，Sorted-based Shuffle并没有对内容进行排序，Sorted-based Shuffle是对Shuffle进行Sort,对我们具体要执行的内容没有排序。

Reducer在什么时候去fetch数据？

当parent Stage的所有ShuffleMapTasks结束后再fetch。等所有的ShuffleMapTask执行完之后，边fetch边计算。

通过动手实践确实证明了Sort-based Shuffle产生了2M个文件。M是并行Task的数量。

Shuffle\_0\_0\_0.data



shuffle\_0\_3\_0.index

从上可以看出index文件和data文件数量是一样的。

**Sorted Shuffle Writer源码：**

1. ShuffleMapTask的runTask方法

反序列化RDD和Dependency

调用SortShuffleManager的getWriter方法。

Writer方法写入结果。

**override def** runTask(context: TaskContext): MapStatus = {  
 // Deserialize the RDD using the broadcast variable.  
 **val** deserializeStartTime = System.*currentTimeMillis*()  
 **val** ser = SparkEnv.*get*.closureSerializer.newInstance()

**//反序列化获得RDD和Dependency**  
 **val** (rdd, dep) = ser.deserialize[(RDD[\_], ShuffleDependency[\_, \_, \_])](  
 ByteBuffer.*wrap*(taskBinary.value), Thread.*currentThread*.getContextClassLoader)  
 *\_executorDeserializeTime* = System.*currentTimeMillis*() - deserializeStartTime  
  
 *metrics* = *Some*(context.taskMetrics)  
 **var** writer: ShuffleWriter[Any, Any] = **null  
 try** {

**//获得SortShuffleManager**  
 **val** manager = SparkEnv.*get*.shuffleManager

**//获得SortShuffleManager的getWriter方法，获得partitionId**  
 writer = manager.getWriter[Any, Any](dep.*shuffleHandle*, partitionId, context)

//将结果写入到文件中。  
 writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[\_ <: Product2[Any, Any]]])  
 writer.stop(success = **true**).get

1. SortShuffleManager复写了ShuffleManager中的getWriter方法，源码如下：

*/\*\* Get a writer for a given partition. Called on executors by map tasks. \*/***override def** getWriter[K, V](  
 handle: ShuffleHandle,  
 mapId: Int, //也就是partitionId  
 context: TaskContext): ShuffleWriter[K, V] = {  
 *numMapsForShuffle*.putIfAbsent(

*//中间代码省略*  
 **case** other: BaseShuffleHandle[K @unchecked, V @unchecked, \_] =>

*//创建*SortShuffleWriter对象

*//shuffleBlockResolver：index文件*  
 **new** SortShuffleWriter(*shuffleBlockResolver*, other, mapId, context)  
 }

1. SorShuffleWriter的write方法源码如下：

*/\*\* Write a bunch of records to this task's output \*/***override def** write(records: Iterator[Product2[K, V]]): Unit = {  
 *sorter* = **if** (*dep*.mapSideCombine) {  
 *require*(*dep*.aggregator.isDefined, "Map-side combine without Aggregator specified!")

*//创建*ExternalSorter实例。  
 **new** ExternalSorter[K, V, C](  
 context, *dep*.aggregator, *Some*(*dep*.partitioner), *dep*.keyOrdering, *dep*.serializer)  
 } **else** {  
 // In this case we pass neither an aggregator nor an ordering to the sorter, because we don't  
 // care whether the keys get sorted in each partition; that will be done on the reduce side  
 // if the operation being run is sortByKey.  
 **new** ExternalSorter[K, V, V](  
 context, aggregator = None, *Some*(*dep*.partitioner), ordering = None, *dep*.serializer)  
 }

*//然后将结果通过insertAll写入缓存中。*  
 *sorter*.insertAll(records)  
  
 // Don't bother including the time to open the merged output file in the shuffle write time,  
 // because it just opens a single file, so is typically too fast to measure accurately  
 // (see SPARK-3570).

*//获得当前文件的输出路径*  
 **val** output = shuffleBlockResolver.getDataFile(*dep*.*shuffleId*, mapId)  
 **val** tmp = Utils.*tempFileWith*(output)

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

*//调用ExternalSorter.* writePartitionedFile将中间结果持久化  
 **val** partitionLengths = *sorter*.writePartitionedFile(blockId, tmp)

*//创建索引文件。*  
 shuffleBlockResolver.writeIndexFileAndCommit(*dep*.*shuffleId*, mapId, partitionLengths, tmp)  
 *mapStatus* = *MapStatus*(*blockManager*.*shuffleServerId*, partitionLengths)  
}

其中ShuffleBlockId记录shuffleId和mapId获得Block。

// Format of the shuffle block ids (including data and index) should be kept in sync with  
//org.apache.spark.network.shuffle.ExternalShuffleBlockResolver#getBlockData().  
@DeveloperApi  
**case class** ShuffleBlockId(shuffleId: Int, mapId: Int, reduceId: Int) **extends** BlockId {

//ShuffleBlockId的格式如下：  
 **override def** name: String = "shuffle\_" + shuffleId + "\_" + mapId + "\_" + reduceId  
}

1. 其中writeIndexFileAndCommit方法：

用于在Block的索引文件中记录每个block的偏移量，其中getBlockData方法可以根据ShuffleId和mapId读取索引文件，获得前面partition计算之后，，将结果写入文件中的偏移量和结果的大小。

*/\*\*  
 \* Write an index file with the offsets of each block, p**lus a final offset at the end for the  
 \* end of the output file. This will be used by getBlockData to figure out where each block  
 \* begins and ends.  
 \*  
 \* It will commit the data and index file as an atomic operation, use the existing ones, or  
 \* replace them with new ones.  
 \*  
 \* Note: the* `*lengths*` *will be updated to match the existing index file if use the existing ones.  
 \* \*/***def** writeIndexFileAndCommit(  
 shuffleId: Int,  
 mapId: Int,  
 lengths: Array[Long],  
 dataTmp: File): Unit = {  
 **val** indexFile = getIndexFile(shuffleId, mapId)  
 **val** indexTmp = Utils.*tempFileWith*(indexFile)  
 **val** out = **new** DataOutputStream(**new** BufferedOutputStream(**new** FileOutputStream(indexTmp)))  
 Utils.*tryWithSafeFinally* {  
 // We take in lengths of each block, need to convert it to offsets.  
 **var** offset = 0L  
 out.writeLong(offset)  
 **for** (length <- lengths) {  
 offset += length  
 out.writeLong(offset)  
 }  
 } {  
 out.close()  
 }  
  
 **val** dataFile = getDataFile(shuffleId, mapId)  
 // There is only one IndexShuffleBlockResolver per executor, this synchronization make sure  
 // the following check and rename are atomic.  
 synchronized {  
 **val** existingLengths = checkIndexAndDataFile(indexFile, dataFile, lengths.length)  
 **if** (existingLengths != **null**) {  
 // Another attempt for the same task has already written our map outputs successfully,  
 // so just use the existing partition lengths and delete our temporary map outputs.  
 System.*arraycopy*(existingLengths, 0, lengths, 0, lengths.length)  
 **if** (dataTmp != **null** && dataTmp.exists()) {  
 dataTmp.delete()  
 }  
 indexTmp.delete()  
 } **else** {  
 // This is the first successful attempt in writing the map outputs for this task,  
 // so override any existing index and data files with the ones we wrote.  
 **if** (indexFile.exists()) {  
 indexFile.delete()  
 }  
 **if** (dataFile.exists()) {  
 dataFile.delete()  
 }  
 **if** (!indexTmp.renameTo(indexFile)) {  
 **throw new** IOException("fail to rename file " + indexTmp + " to " + indexFile)  
 }  
 **if** (dataTmp != **null** && dataTmp.exists() && !dataTmp.renameTo(dataFile)) {

总结如下：



三：默认Sort-based Shuffle的几个缺陷：

1. 如果Mapper中Task的数量过大，依旧会产生很多小文件，此时在Shuffle传递数据的过程中到Reducer端，reduce会需要同时打开大量的记录来进行反序列化，导致大量的内存消耗和GC的巨大负担，造成系统缓慢甚至崩溃！

2．如果需要在分片内也进行排序的话，此时需要进行Mapper端和Reducer端的两次排序！！！

优化：

可以改造Mapper和Reducer端，改框架来实现一次排序。

频繁GC的解决办法是：钨丝计划！！

全文总结如下：

