

Hash Based Shuffle Write

1，实现：

每个Shuffle Map Task根据Key的哈希值，计算出每个Key需要写入的Partition然后将数据单独写入一个文件，此时的Parititon就对应下游的一个Shuffle Map Task或者ResultTask。

Basic Shuffle Writer实现解析

1. Shuffle Map Task计算是调用ShuffleMapTask.runTask执行的。

核心代码如下：

1. 获得ShuffleManager为 Hash Shuffle。
2. 获得Hash Shuffle的Writer方法：HashShuffleWriter.
3. 调用HashShuffleWriter的write方法。其中调用RDD的iterator方法计算，然后将结果传入给write。

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

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

**//**调用HashShuffleWriter的write方法，  
writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[\_ <: Product2[Any, Any]]])

下面具体看write方法。

*/\*\* Write a bunch of records to this task's output \*/***override def** write(records: Iterator[Product2[K, V]]): Unit = {

*//判断*aggregator是否被定义  
 **val** iter = **if** (*dep*.aggregator.isDefined) {

*//判断数据是否需要聚合如果需要，聚合records*  
 **if** (*dep*.mapSideCombine) {  
 *dep*.aggregator.get.combineValuesByKey(records, context)  
//中间代码省略

//elem是(K,V)形式的，通过K计算出bucketId  
 **for** (elem <- iter) {  
 **val** bucketId = *dep*.partitioner.getPartition(elem.\_1)

//然后再通过bucketId具体写入那个partition

//此时Shuffle是FileShuffleBlockResolver  
 *shuffle*.writers(bucketId).write(elem.\_1, elem.\_2)  
 }

1. 具体看一下FileShuffleBlockResolver.writers：

**val** *writers*: Array[DiskBlockObjectWriter] = {  
 Array.*tabulate*[DiskBlockObjectWriter](numReducers) { bucketId =>  
 **val** blockId = *ShuffleBlockId*(shuffleId, mapId, bucketId)  
 **val** blockFile = *blockManager*.*diskBlockManager*.getFile(blockId)  
 **val** tmp = Utils.*tempFileWith*(blockFile)

**//tmp也就是blockFile如果已经存在则，在后面追加数据**  
 *blockManager*.getDiskWriter(blockId, tmp, *serializerInstance*, *bufferSize*, writeMetrics)  
 }

1. blockManager.getDiskWriter就会为每个文件创建一个DiskBlockObjectWriter

**new** DiskBlockObjectWriter(file, serializerInstance, bufferSize, compressStream,  
 syncWrites, writeMetrics, blockId)

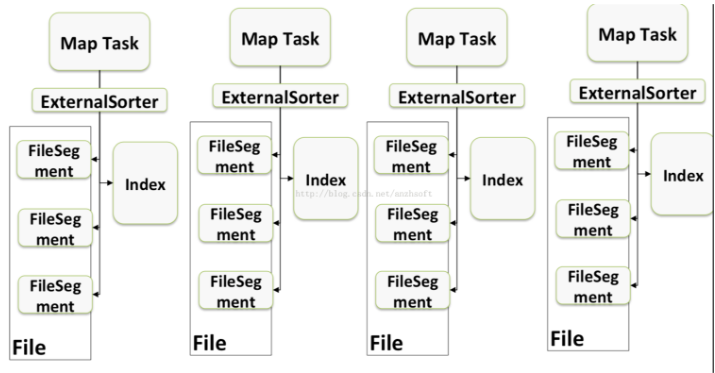
1. DiskBlockObjectWriter可以直接向一个在磁盘上的文件写数据，并且允许在后面追加数据

*\* A class for writing JVM objects directly to a file on disk. This class allows data to be appended  
\** *to an existing block and can guarantee atomicity in the case of faults as it allows the caller to  
\* revert partial writes.*

**private**[spark] **class** DiskBlockObjectWriter(

**Sort Based Shuffle**

在Sort Based Shuffle中，对传入的records根据他们的目标分区(就是Shuffle之后要生成的RDD)id进行排序。然后写入一个map文件中，Reducers从map文件中 fetch出属于自己的部分，如果map文件过大的话，就要将内存中的数据spill到磁盘，并且将磁盘中的文件合并，以产生最终输出文件。



为了便于下游的Task获取到其需要的Partition，这里会生成一个index文件，去记录不同的partition的位置信息。

核心实现逻辑是通过SortShuffleWriter和依赖类实现

1. Spark1.6中默认的是Sort

// 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)

Sort Based Write

1. Shuffle Map Task计算是调用ShuffleMapTask.runTask执行的。
2. 反序列化获得RDD和ShuffleDependency.
3. 获得ShuffleManager为 SortShuffleManager。
4. 然后获得SortShuffleManager的Writer方法。

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

**//反序列化得到RDD和ShuffleDependency**  
 **val** (rdd, dep) = ser.deserialize[(RDD[\_], ShuffleDependency[\_, \_, \_])](  
 **var** writer: ShuffleWriter[Any, Any] = **null  
 try** {  
 **val** manager = SparkEnv.*get*.shuffleManager //  
 writer = manager.getWriter[Any, Any](dep.*shuffleHandle*, partitionId, context)  
 writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[\_ <: Product2[Any, Any]]])

下面具体看一下SortShuffleManager中的getWriter方法：调用SortShuffleManager的getWriter方法获得partitionId指定分区的SortShuffleWriter.

*/\*\** *Get a writer for a given partition. Called on executors by map tasks. \*/***override def** getWriter[K, V](  
 handle: ShuffleHandle,  
 mapId: Int, //partitionId  
//中间代码省略  
 **case** other: BaseShuffleHandle[K @unchecked, V @unchecked, \_] =>

*//创建SortShuffleWriter对象*  
 **new** SortShuffleWriter(*shuffleBlockResolver*, other, mapId, context)  
 }

1. SortShuffleWriter负责将数据持久化。

ExternalSorter：所有的数据都添加到这个ExternalSorter，然后将数据写入到一个文件在磁盘中存储。

*/\*\* 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对象  
 **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)  
 }

*//然后通过insertAll将数据缓存。*  
 *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).  
 **val** output = shuffleBlockResolver.getDataFile(*dep*.*shuffleId*, mapId)  
 **val** tmp = Utils.*tempFileWith*(output)  
 **val** blockId = *ShuffleBlockId*(*dep*.*shuffleId*, mapId, IndexShuffleBlockResolver.*NOOP\_REDUCE\_ID*)

//调用ExternalSorter的writePartitionedFile将输出结果持久化到磁盘上

//其中tmp是output输出结果。  
 **val** partitionLengths = *sorter*.writePartitionedFile(blockId, tmp)

//创建索引文件  
 shuffleBlockResolver.writeIndexFileAndCommit(*dep*.*shuffleId*, mapId, partitionLengths, tmp)

//创建mapStatus，也就是通过它来告诉下游的Stage  
 *mapStatus* = *MapStatus*(*blockManager*.*shuffleServerId*, partitionLengths)  
}

1. 而数据读取过程中则需要使用IndexShuffleBlockManager来获取Partiton的具体位置。
2. IndexShuffleBlockResolver：
3. 创建和维护Shuffle blocks的映射，逻辑块和物理文件位置之间的映射关系。
4. 来自同一个map Task的Shuffle blocks被存储在一个统一的数据文件中。而在数据文件中的数据块的索引被存储在一个单独的文件中。

*\* Create and maintain the shuffle blocks' mapping between logic block and physical file location.  
 \* Data of shuffle blocks from the same map task are stored in a single consolidated data file.  
 \* The offsets of the data blocks in the data file are stored in a separate index file.  
 \*  
 \* We use the name of the shuffle data's shuffleBlockId with reduce ID set to 0 and add ".data"  
 \* as the filename postfix for data file, and ".index" as the filename postfix for index file.*

**private**[spark] **class** IndexShuffleBlockResolver(