**一：到底什么是Shuffle？**

Shuffle中文翻译为“洗牌”，需要Shuffle的关键性原因是某种具有共同特征的数据需要最终汇聚到一个计算节点上进行计算。

二：Shuffle可能面临的问题？运行Task的时候才会产生Shuffle（Shuffle已经融化在Spark的算子中了）。

1. 数据量非常大，从其他各台机器收集数据占用大量网络。
2. 数据如何分类，即如何Partition，Hash、Sort、钨丝计算；
3. 负载均衡（数据倾斜），因为采用不同的Shuffle方式对数据不同的分类，而分类之后又要跑到具体的节点上计算，如果不恰当的话，很容易产生数据倾斜；
4. 网络传输效率，需要在压缩和解压缩之间做出权衡，序列化和反序列也是要考虑的问题；

说明：具体的Task进行计算的时候尽一切最大可能使得数据具备Process Locality的特性；退而求次是增加数据分片，减少每个Task处理的数据量。

注：除非你的计算特别复杂，否则的话，都要要求所有的数据在内存中，即是说任务很多需要排队，但是总体的运行速度更快，一般情况下，进行持久化是没有什么收益的，而读中间结果cache得到的价值，还不如出错的情况下重算一遍，除非计算链条特别长，计算特别的复杂。

Cache的风险：

1. 有风险，例如,Memory溢出的风险，例如说被别人占用掉这个内存的风险。
2. 读磁盘是一个高风险的事，但是读内存就更安全，这是在一个Stage内提倡的。

但是Shuffle是需要网络通信，这个时候就需要持久化，因为所有的父Stage算完，才能进行下一个Stage的计算，所以没有持久化，下一个Stage出错的话就需要全部重计算，Shuffle默认持久化到磁盘里，也可以持久化到内存中，Tachyon，Locality System中。Spark默认遇到Shuffle都会将结果进行持久化的。

**二：Hash Shuffle**

1. 要求key不能是Array；
2. Hash Shuffle不需要排序，此时从理论上讲就节省了Hadoop MapReduce中进行Shuffle需要排序时候的时间浪费，因为实际生产环境有大量的不需要排序的Shuffle类型；

**思考：不需要排序的Hash Shuffle是否一定比需要排序的Sorted Shuffle速度更快？**不一定！如果数据规模比较小的情形下，Hash Shuffle会比Sorted Shuffle速度快（很多）！但是如果数据量大，此时Sorted Shuffle一般都会比Hash Shuffle快（很多），因为如果数据规模比较大，Hash Shuffle甚至无法处理，因为Hash Shuffle会产生很多的句柄，小文件，这时候磁盘和内存会变成瓶颈，而Sorted Shuffle就会极大的节省内存和磁盘的访问，所以更有利于更大规模的计算。Hash Shuffle适合中小型规模的数据计算。

1. 每个ShuffleMapTask会根据key的哈希值计算出当前的key需要写入的Partition，然后把决定后的结果写入单独的文件，此时会导致每个Task产生R（指下一个Stage的Task并行度）个文件，如果当前的Stage中有M个ShuffleMapTask，则会产生M\*R个文件！！！此时数据已经分好类了，下一个Stage就会通过网络根据Driver端的注册信息，因为上一个Stage写过的内容会注册给Driver，然后向Driver获取上一个Stage的输出位置，就会通过网络去读取数据，数据分成几种类型只跟下一个阶段分成多少个任务有关系，因为下一个阶段的任务数表示数据被分成多少类。跟并行度没有关系，也就是说跟实际并行运行多少任务没有关系。

注意：Shuffle操作绝大多数情况下都要通过网络，如果Mapper和Reducer在同一台机器上，此时只需要读取本地磁盘即可。Spark中的Executor是线程池中的线程复用的，这个线程有可能运行上一个Stage的Task，也有可能运行下一个Stage的Task。

Hash 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

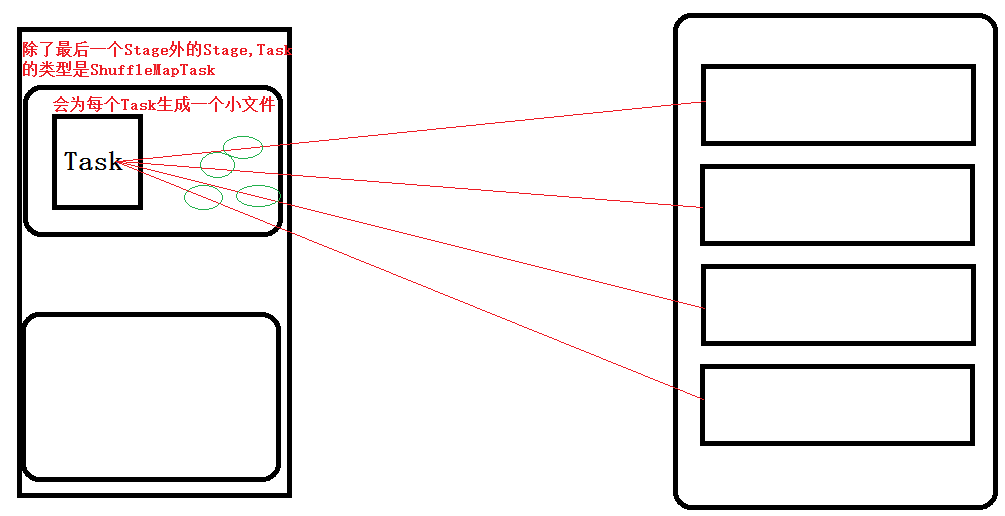
*\* A class for writing JVM objects directly to a file on disk. This class allows data to be appended  
\** *to an existing block and can guarantee atomicity in the case of faults as it allows the caller to  
\* revert partial writes.*

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

Hash Shuffle的两大死穴：

第一：Shuffle前会产生海量的小文件于磁盘之上，此时会产生大量耗时低效的IO操作；

第二：内存不够用！！！由于内存中需要保存海量的文件操作句柄和临时缓存信息，如果数据处理规模比较庞大的话，内存不可承受，出现OOM等问题！



从图上可以看到，进行HashShuffle的时候会根据后面的Task数，生成对应数量的小文件，而每个小文件也就是一种类型，在数据处理的时候，Task就从前面的小文件抓取需要的数据即可，它会导致同时打开过多的文件，这样就会占用过多的内存，写文件通过Write Handler默认是50KB。

**三: Consalidate:**

为了改善上述的问题（**同时打开过多文件导致Writer Handler内存使用过大以及产生过度文件导致大量的随机读写带来的效率极为低下的磁盘IO操作**），Spark后来推出了Consalidate机制，来把小文件(指的是每个Map都要为所有的Reducer产生Reducer个数的小文件)合并，此时Shuffle时文件产生的数量为cores\*R，对于ShuffleMapTask的数量明显多于同时可用的并行Cores的数量的情况下，Shuffle产生的文件会大幅度减少，会极大降低OOM的可能；

**如何将小文件进行合并？**

Consalidate机制：把同一个Task的输出变成一个文件进行合并，根据CPU的个数来决定具体产生多少文件，对于运行在同一个core的Shuffle Map Task，第一个Shuffle Map Task会创建一个，之后的就会将数据追加到这个文件上而不是新建一个文件。

但是在生成环境下，并行度特别大的话，还是会产生原来的问题。

为此Spark推出了Shuffle Pluggable开放框架，方便系统升级的时候定制Shuffle功能模块，也方便第三方系统改造人员根据实际的业务场景来开发具体最佳的Shuffle模块；核心接口ShuffleManager，具体默认实现有HashShuffleManager、SortShuffleManager等，Spark 1.6.0中具体的配置如下：默认情况下是Sort Shuffle

**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")

"tungsten-sort" -> "org.apache.spark.shuffle.sort.SortShuffleManager")  
**val** shuffleMgrName = conf.get("spark.shuffle.manager", "sort")

**Sorted Shuffle是如何解决大量小文件？**

每个ShuffleMapTask不会为每个Reduce单独生成一个文件，它会将结果生成到一个文件里，同时会生成一个索引，那每个reduce可以根据这个index索引，取得它所要的文件数据，这样就避免产生大量的文件，也就不需要大量文件写的句柄，节省了内存，磁盘上的文件也变小了，这个时候不是随机读取，而是顺序Disk IO带来了低延迟，节省了内存，另外可以减少GC的分享的频率，而减少具体的文件数量避免同时写多个文件时给系统带来的压力。

具体实现的时候：

* ShuffleMapTask会根据Key，相应的Partition进行Sort，如果属于同一个Partition本身不会进行Sort。
* 在进行Sort的时候如果内存不够用的话，它会将哪些已经排序的数据写入到外部磁盘，结束的时候再进行归并排序，这时候基本上就不受内存限制了，同时由于在一个文件中，在一个File中，分为File不同的segment，为了高效的读取不同的segment，它就有一个index文件，会记录不同的Partition信息，BlockManager也会对它的寻址算法进行优化。
* Sort Shuffle Writer对于每个Partition会创建一个数组，存储他所包含的Key,Value.每个需要处理的Key,Value,都会被插入到相应的数组中，如果数组的大小超过了具体规定大小的限定值的时候，就需要将内容写入到外部存储。
* 文件开始的部分记录我们这个Partition的ID，以及这个文件具体保存了有哪些数量的数据。
* 最后将所有写入到外部文件进行归并排序，归并排序的时候文件不能过多，如果过多的话会消耗很多的内存，可能会有OOM的风险，或者垃圾回收过多的风险，过少性能不好，会有延迟，最优一般同时打开10~100个文件。
* 最后生成文件的时候需要生成index索引文件，由于它已经排序好了索引文件，所以Reducer去读取索引文件的时候就会非常方便。

对于ShuffleMapTask具体的归并排序的方式也就是extend sort，sort之后其实会产生两个文件，这两个文件，一个是索引文件，另一个是具体文件的内容，我们在Reducer端读数据的时候其实，首先访问Index文件，也就是具体工作的时候BlockManager，首选会帮我们访问Index，Index去定位具体文件的内容。Reducer本身因为它是通过index文件获取它需要处理的数据，这样就可以避免产生大量句柄，节省内存。上游的数据ShuffleMapTask是在一个文件中的。

怎么获取文件内容呢？通过索引去获取，从工作的角度上来讲，只有一个文件句柄，文件句柄和文件数目大大的减少。