打造价值40万Offer的朋友圈
数据人的宝藏朋友圈
欢迎邀请你的同事同学参观

公众号：import_bigdata
知　乎：https://www.zhihu.com/people/wang-zhi-wu-66
CSDN ：https://blog.csdn.net/u013411339
Github：https://github.com/wangzhiwubigdata/God-of-Bigdata

本文已经加入「大数据成神之路PDF版」中提供下载，你可以关注公众号，后台回复：「**PDF**」即可获取。

更多PDF下载可以参考：**《重磅,大数据成神之路PDF可以分类下载啦!》**

Spark重点难点系列：

- **《【Spark重点难点01】你从未深入理解的RDD和关键角色》**
- **《【Spark重点难点02】你以为的Shuffle和真正的Shuffle》**
- **《【Spark重点难点03】你的数据存在哪了?》**
- **《【Spark重点难点04】你的代码跑起来谁说了算？(内存管理)》**

我们的【Spark重点难点】系列继续更新。

以往的系列：

- **我们在学习Spark的时候，到底在学习什么？**
- **我在B站读大学，大数据专业**

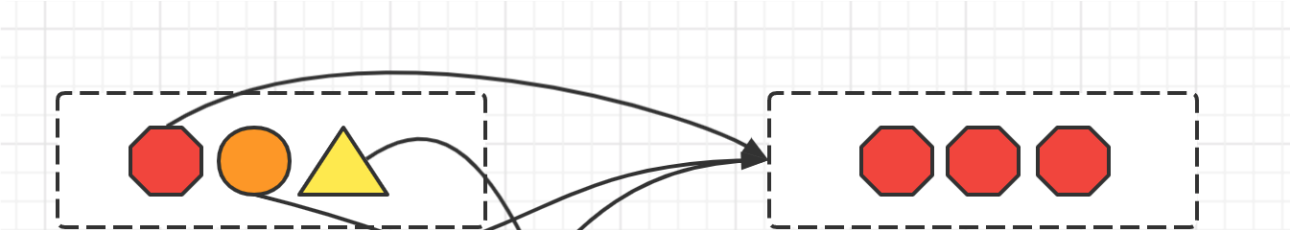
通俗解释

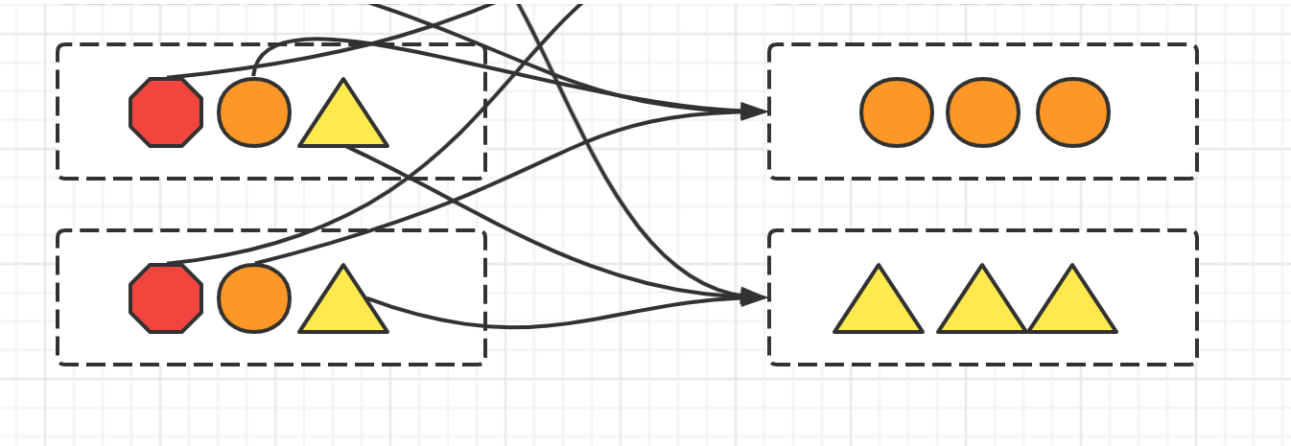
上节课我们讲了DAGScheduler划分Stage的原理: DAGScheduler调度时会根据 `是否需要经过Shuffle过程将Job划分为多个Stage` 。

Shuffle就这么重要?

正是由于Shuffle的计算几乎需要消耗所有类型的硬件资源，比如CPU、内存、磁盘与网络，在绝大多数的Spark作业中，Shuffle往往是作业执行性能的瓶颈。

> Shuffle的本质就是数据重组分发的过程。





Shuffle的过程

有了上图的形状分类的直观对比，我们现在就可以直接给 Shuffle 下一个正式的定义了。

> 集群范围内跨节点、跨进程的数据分发。

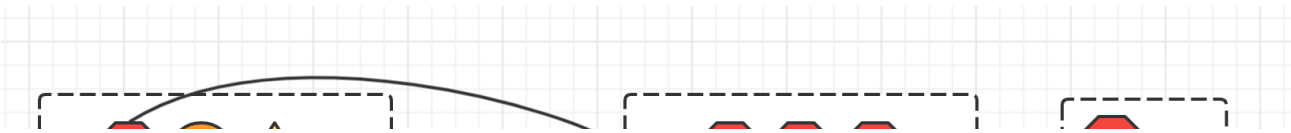
上图中在做形状分类时，集群会需要大量资源进行磁盘和网络的I/O。在DAG的计算链条中，Shuffle环节的执行性能往往是最差的。

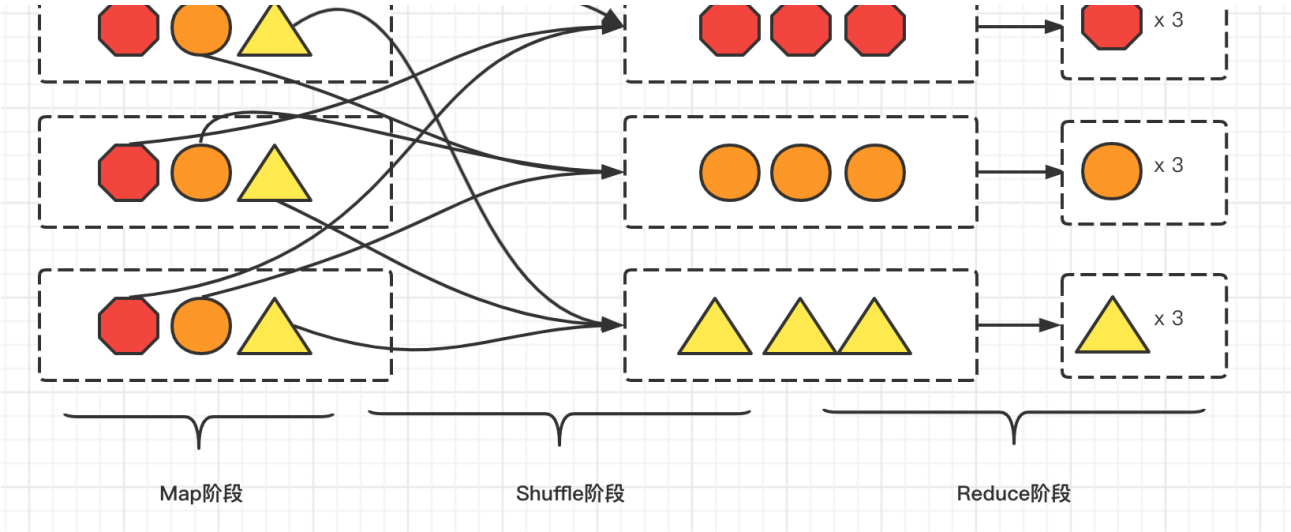
原理

我们用Word Count的例子来做说明。在这个示例中，引入Shuffle操作的是reduceByKey算子:

```
line.flatMap(_.split(" ")).map((_, 1)).reduceByKey(_+_).collect().foreach(println)
```

以Shuffle为边界，reduceByKey的计算被切割为两个执行阶段。约定俗成地，我们把Shuffle之前的Stage叫作Map阶段，而把Shuffle之后的 Stage称作Reduce阶段。在Map阶段，每个Executors先把自己负责的数据分区做初步聚合（又叫 Map 端聚合、局部聚合）;在Shuffle环节，不同的单词被分发到不同节点的Executors中；最后的Reduce阶段，Executors以单词为Key做第二次聚合，从而完成统计计数的任务。如下图所示。





中间文件

Map阶段与Reduce阶段，通过生产与消费Shuffle中间文件的方式，来完成集群范围内的数据交换。

在Map执行阶段，每个Task（以下简称 Map Task）都会生成包含data 文件与index文件的Shuffle中间文件，如上图所示。也就是说，Shuffle 文件的生成，是以Map Task为粒度的，Map阶段有多少个Map Task，就会生成多少份Shuffle中间文件。

Shuffle Write

Shuffle写入临时文件的过程叫做： `Shuffle Write` 。

Spark现支持三种writer，分为BypassMergeSortShuffleWriter， SortShuffleWriter 和 UnsafeShuffleWriter。

每种Shuffle witer都有非常复杂的实现机制。如果你对Shuffle的底层实现非常感兴趣可以参考:

> https://blog.csdn.net/wendelee/article/details/109818711

在生成中间文件的过程中，Spark 会借助一种类似于 Map 的数据结构，来计算、缓存并排序数据分区中的数据记录。这种 Map 结构的 Key 是（Reduce Task Partition ID，Record Key）的二元组，而 Value 是原数据记录中的数据值。

总结下来，Shuffle 中间文件的生成过程，分为如下几个步骤:

1. 对于数据分区中的数据记录，逐一计算其目标分区，然后填充内存数据结构；
2. 当数据结构填满后，如果分区中还有未处理的数据记录，就对结构中的数据记录按（目标分区 ID，Key）排序，将所有数据溢出到临时文件，同时清空数据结构；
3. 重复前 2 个步骤，直到分区中所有的数据记录都被处理为止；
4. 对所有临时文件和内存数据结构中剩余的数据记录做归并排序，生成数据文件和索引文件。

Shuffle Reader

对于所有 Map Task 生成的中间文件，Reduce Task 需要通过网络从不同节点的硬盘中下载并拉取属于自己的数据内容。不同的 Reduce Task 正是根据 index 文件中的起始索引来确定哪些数据内容是属于自己的。这个拉取数据的过程被叫做Shuffle Read。

Shuffle Reader的实现都被封装在了 `BlockStoreShuffleReader` 。

整个Reader的流程主要是：

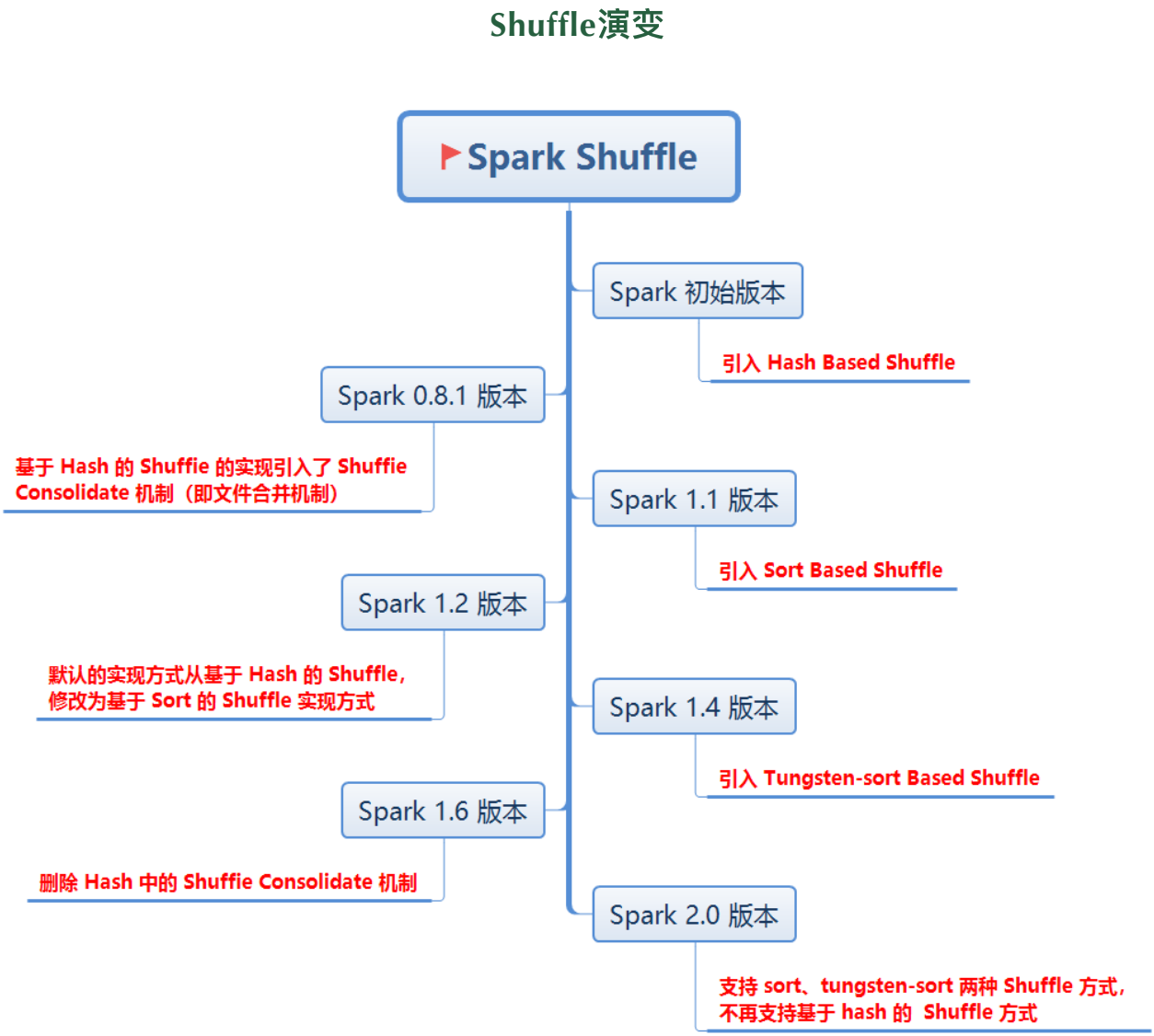
- 首先新建ShuffleBlockFetcherIterator获取数据迭代器，会返回(blockId, inputStream)的数据迭代器；
- 对每个block数据进行压缩和加密操作，是通过serializerManager进行的，对每个block数据进行反序列化，反序列化输入流成为<K,V>数据迭代器；
- 对迭代器添加监控和数据处理完成后的清洗函数处理工作；
- 如果要进行聚合操作，会对各个map的当前reduceId的数据进行聚合；
- 如果需要排序，对聚合后的数据进行排序操作。

需要特别注意的是，Shuffle Reader过程可以从两个地方来读取数据块，一个是本地的block，一个是远程的block。远程的block读取是通过向BlockTransferService这个服务发送读取数据块请求来获取数据数据。那么如何区分是从本地读，还是从远程读取呢?

> 是通过每个块的executorID来区分的，本地环境的executorID和块的id相等就是从本地读，若不相等就会从远端节点读取数据。

源码级别的分析你可以参考：

> https://blog.csdn.net/lidongmeng0213/article/details/109527227



Spark Shuffle的演变

我们可以看到，从Spark2.0以后，Hash Based Shuffle退出了历史舞台，本着过时不讲的原则，我们来看一下SortShuffleManager的运行机制。

目前Spark2.0及以上的版本，Shuffle框架主要包括以下几个部分：

- ShuffleManager

这是一个接口，负责管理shuffle相关的组件，比如：通过它来注册shuffle的操作函数，获取writer和reader等。在sparkenv中注册，通过sprkconf进行配置，配置参数是：spark.shuffle.manager，默认是sort，也就是：SortShuffleManager类。在早期的spark版本中，也实现过hashmanager后来全部统一成sort。

- ShuffleReader

在reduce任务中去获取来自多个mapper任务的合并记录数据。实现该接口的类只有一个：BlockStoreShuffleReader。

- ShuffleWriter

在mapper任务中把记录到shuffle系统。这是一个抽象类，实现该抽象类的有：SortShuffleWriter，UnsafeShuffleWriter，BypassMergeSortShuffleWriter三个。

- ShuffleBlockResolver

该接口的实现类需要理解：如何为逻辑的shuffle块标识(map,reduce,shuffle等)获取数据。实现者可以通过文件或文件片段来封装shuffle数据。当获取到shuffle数据时，BlockStore使用它来抽象不同的shuffle实现。该接口的实现类为：IndexShuffleBlockResolver。

SortShuffleManager

SortShuffleManager的运行机制分为三种：

1. 普通运行机制

2. bypass运行机制

当 shuffle read task 的数量小于等于 `spark.shuffle.sort.bypassMergeThreshold` 参数的值时（默认为 200），就会启用 bypass 机制；

3. Tungsten Sort运行机制

开启此运行机制需设置配置项 `spark.shuffle.manager=tungsten-sort` 。但是开启此项配置也不能保证就一定采用此运行机制。

普通运行机制

在该模式下，数据会先写入一个内存数据结构中，此时根据不同的 shuffle 算子，可能选用不同的数据结构。 `如果是 reduceByKey 这种聚合类的 shuffle 算子，那么会选用 Map`

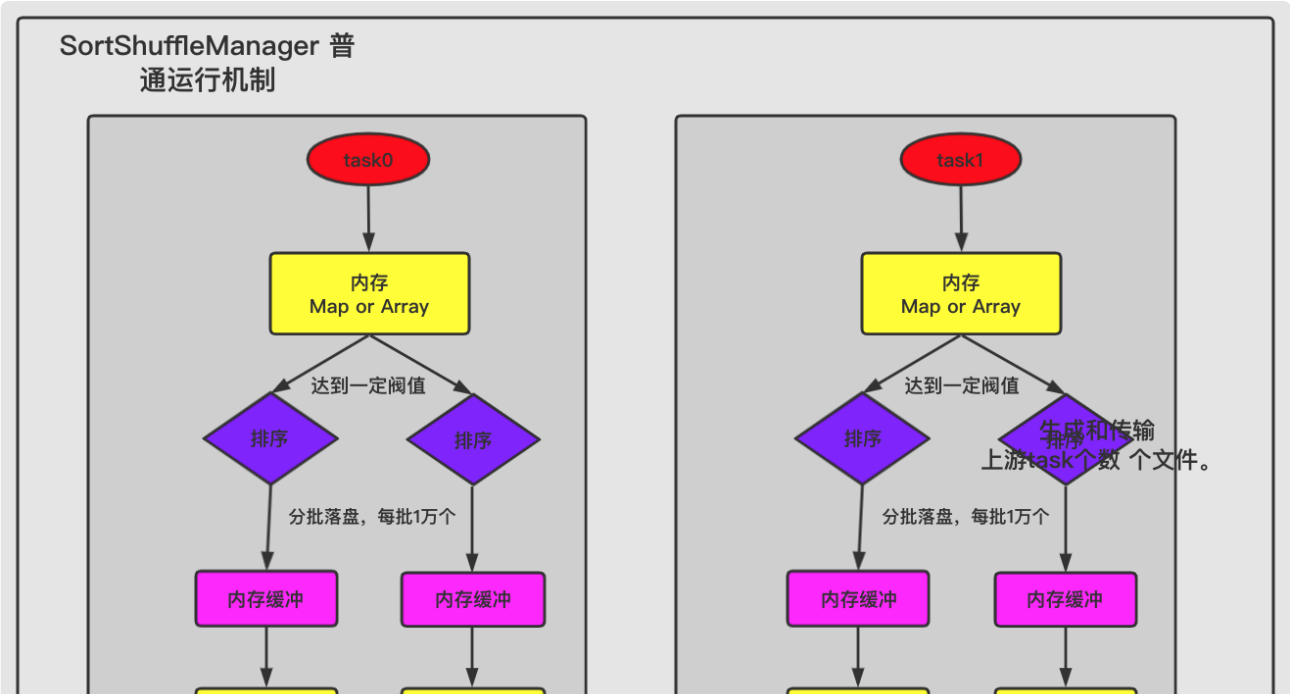
数据结构，一边通过 Map 进行聚合，一边写入内存；如果是 join 这种普通的 shuffle 算子，那么会选用 Array 数据结构，直接写入内存 。接着，每写一条数据进入内存数据结构之后，就会判断一下，是否达到了某个临界阈值。如果达到临界阈值的话，那么就会尝试将内存数据结构中的数据溢写到磁盘，然后清空内存数据结构。

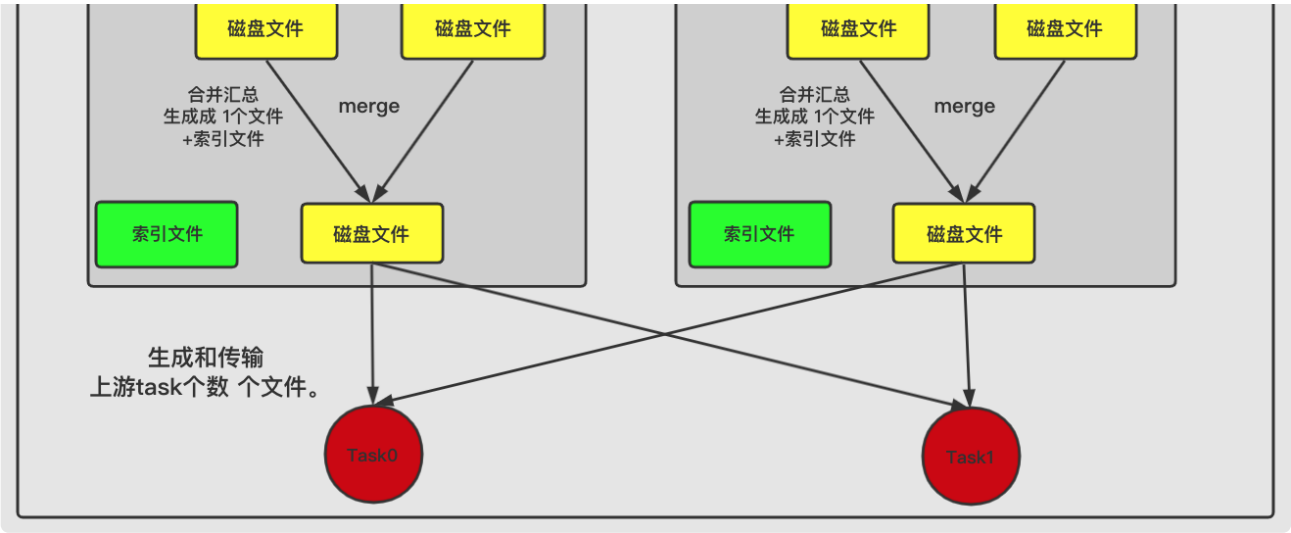
在溢写到磁盘文件之前，会先根据 key 对内存数据结构中已有的数据进行排序。 排序过后，会分批将数据写入磁盘文件。默认的 batch 数量是 10000 条，也就是说，排序好的数据，会以每批 1 万条数据的形式分批写入磁盘文件 。写入磁盘文件是通过 Java 的 BufferedOutputStream 实现的。 BufferedOutputStream 是 Java 的缓冲输出流，首先会将数据缓冲在内存中，当内存缓冲满溢之后再一次写入磁盘文件中，这样可以减少磁盘 IO 次数，提升性能。

一个 task 将所有数据写入内存数据结构的过程中，会发生多次磁盘溢写操作，也就会产生多个临时文件。最后会将之前所有的临时磁盘文件都进行合并，这就是merge 过程，此时会将之前所有临时磁盘文件中的数据读取出来，然后依次写入最终的磁盘文件之中。此外，由于一个 task 就只对应一个磁盘文件，也就意味着该 task 为下游 stage 的 task 准备的数据都在这一个文件中，因此还会单独写一份索引文件，其中标识了下游各个 task 的数据在文件中的 start offset 与 end offset。

SortShuffleManager 由于有一个磁盘文件 merge 的过程，因此大大减少了文件数量。比如第一个 stage 有 50 个 task，总共有 10 个 Executor，每个 Executor 执行 5 个 task，而第二个 stage 有 100 个 task。由于每个 task 最终只有一个磁盘文件，因此此时每个 Executor 上只有 5 个磁盘文件，所有 Executor 只有 50 个磁盘文件。

普通运行机制的 SortShuffleManager 工作原理如下图所示：





bypass 运行机制

Reducer 端任务数比较少的情况下，基于 `Hash Shuffle` 实现机制明显比基于 `Sort Shuffle` 实现机制要快，因此基于 `Sort huffle` 实现机制提供了一个回退方案，就是 bypass 运行机制。对于 Reducer 端任务数少于配置属性 `spark.shuffle.sort.bypassMergeThreshold` 设置的个数时，使用带 Hash 风格的回退计划。

bypass 运行机制的触发条件如下：

- shuffle map task 数量小于 `spark.shuffle.sort.bypassMergeThreshold=200` 参数的值。
- 不是聚合类的 shuffle 算子。

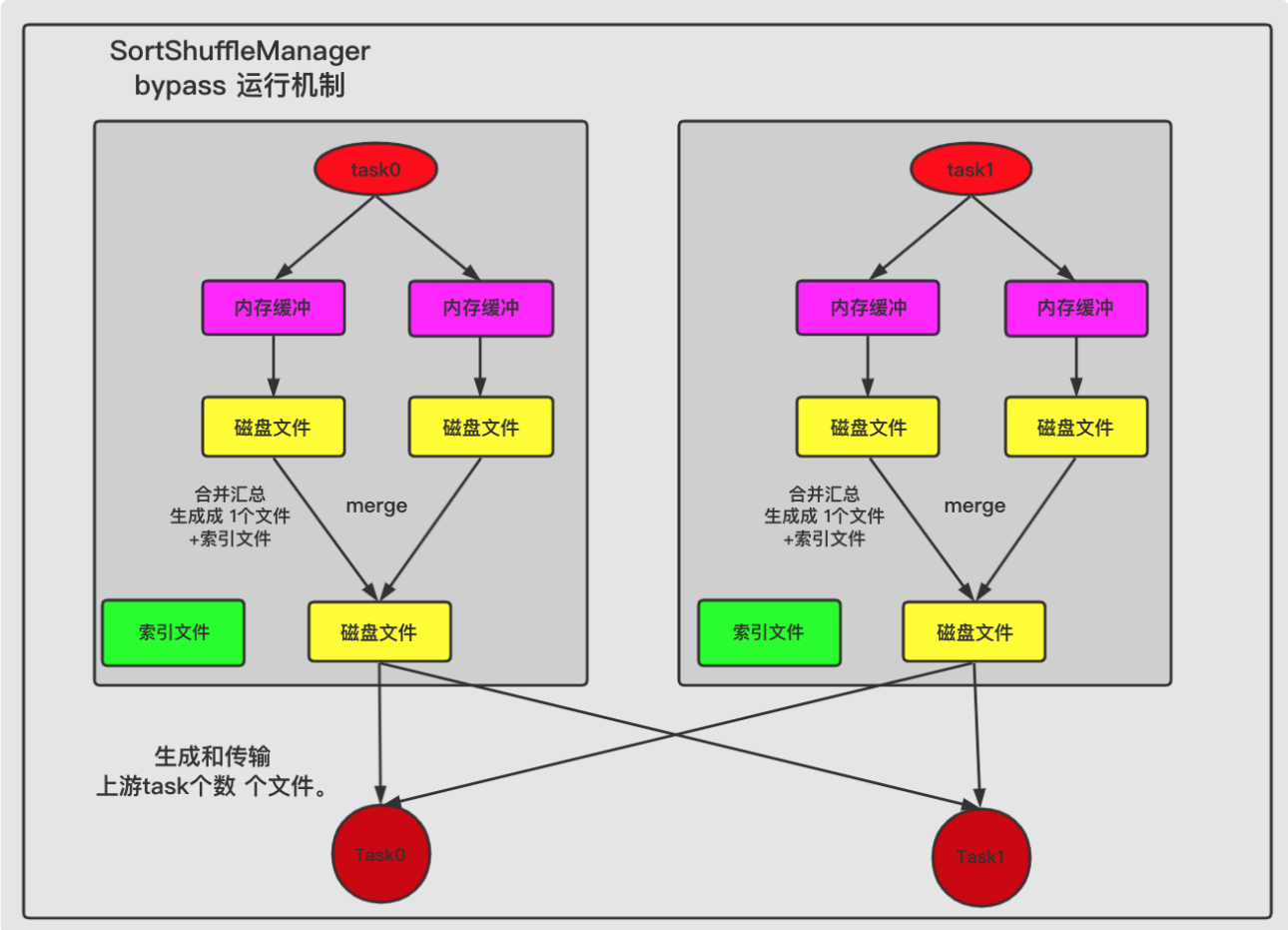
此时，每个 task 会为每个下游 task 都创建一个临时磁盘文件，并将数据按 key 进行 hash 然后根据 key 的 hash 值，将 key 写入对应的磁盘文件之中。当然，写入磁盘文件时也是先写入内存缓冲，缓冲写满之后再溢写到磁盘文件的。最后，同样会将所有临时磁盘文件都合并成一个磁盘文件，并创建一个单独的索引文件。

该过程的磁盘写机制其实跟未经优化的 `HashShuffleManager` 是一模一样的，因为都要创建数量惊人的磁盘文件，只是在最后会做一个磁盘文件的合并而已。因此少量的最终磁盘文件，也让该机制相对未经优化的 `HashShuffleManager` 来说， `shuffle read` 的性能会更好。

而该机制与普通 `SortShuffleManager` 运行机制的不同在于：第一，磁盘写机制不同；第二，不会进行排序。也就是说，启用该机制的最大好处在于， `shuffle write` 过程

中，不需要进行数据的排序操作，也就节省掉了这部分的性能开销。

`bypass` 运行机制的 `SortShuffleManager` 工作原理如下图所示：



Tungsten Sort Shuffle 运行机制

基于 Tungsten Sort 的 Shuffle 实现机制主要是借助 Tungsten 项目所做的优化来高效处理 Shuffle。

Spark 提供了配置属性，用于选择具体的 Shuffle 实现机制，但需要说明的是，虽然默认情况下 Spark 默认开启的是基于 SortShuffle 实现机制，但实际上，参考 Shuffle 的框架内核部分可知基于 SortShuffle 的实现机制与基于 Tungsten Sort Shuffle 实现机制都是使用 SortShuffleManager，而内部使用的具体的实现机制，是通过提供的两个方法进行判断的：

对应非基于 Tungsten Sort 时，通过 SortShuffleWriter.shouldBypassMergeSort 方法判断是否需要回退到 Hash 风格的 Shuffle 实现机制，当该方法返回的条件不满足时，则通过 SortShuffleManager.canUseSerializedShuffle 方法判断是否需要采用基于 Tungsten Sort

Shuffle 实现机制，而当这两个方法返回都为 false，即都不满足对应的条件时，会自动采用普通运行机制。

因此，当设置了 `spark.shuffle.manager=tungsten-sort` 时，也不能保证就一定采用基于 Tungsten Sort 的 Shuffle 实现机制。

要实现 Tungsten Sort Shuffle 机制需要满足以下条件：

- Shuffle 依赖中不带聚合操作或没有对输出进行排序的要求。
- Shuffle 的序列化器支持序列化值的重定位（当前仅支持 KryoSerializer Spark SQL 框架自定义的序列化器）。
- Shuffle 过程中的输出分区个数少于 16777216 个。

实际上，使用过程中还有其他一些限制，如引入 Page 形式的内存管理模型后，内部单条记录的长度不能超过 128 MB （具体内存模型可以参考 PackedRecordPointer 类）。另外，分区个数的限制也是该内存模型导致的。

所以，目前使用基于 Tungsten Sort Shuffle 实现机制条件还是比较苛刻的。

基于Sort的Shuffle机制的优缺点

- 优点

1. 小文件的数量大量减少，Mapper 端的内存占用变少；

2. Spark 不仅可以处理小规模的数据，即使处理大规模的数据，也不会很容易达到性能瓶颈。

- 缺点

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 传数据的过程中到 Reducer 端， Reducer 会需要同时大量地记录进行反序列化，导致大量内存消耗和 GC 负担巨大，造成系统缓慢，甚至崩溃；

2. 强制了在 Mapper 端必须要排序，即使数据本身并不需要排序；

3. 它要基于记录本身进行排序，这就是 `Sort-Based Shuffle` 最致命的性能消耗。

优化

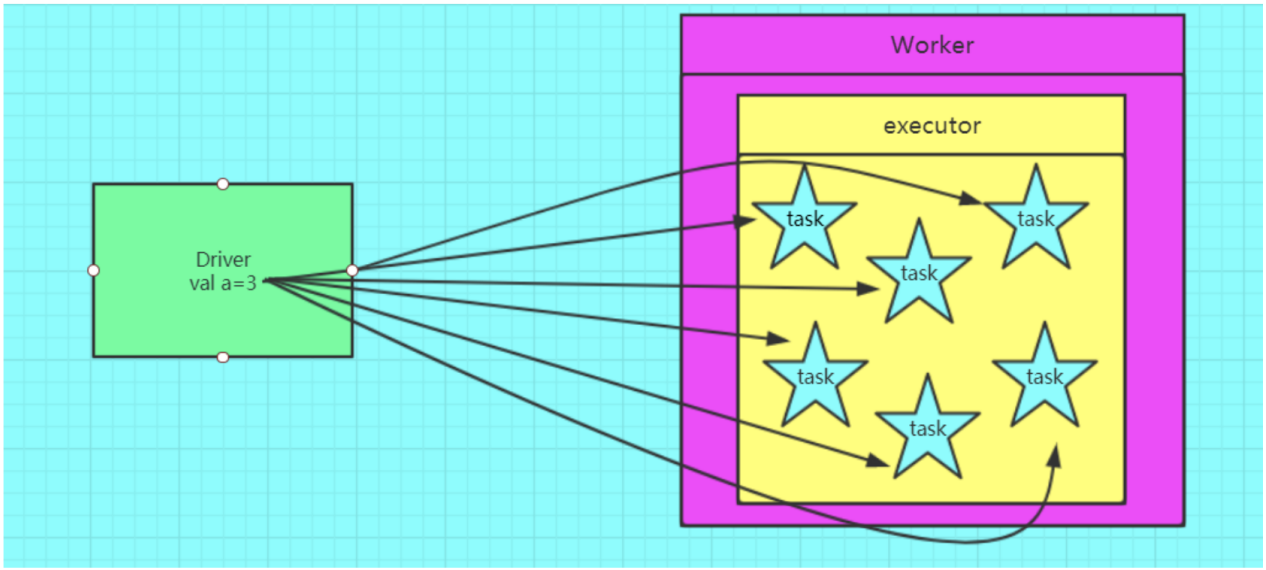
在之前的文章中我们对于Shuffle的优化也讲过很多次。我们可以引用美团博客中关于Shuffle的一手调优资料。

广播变量

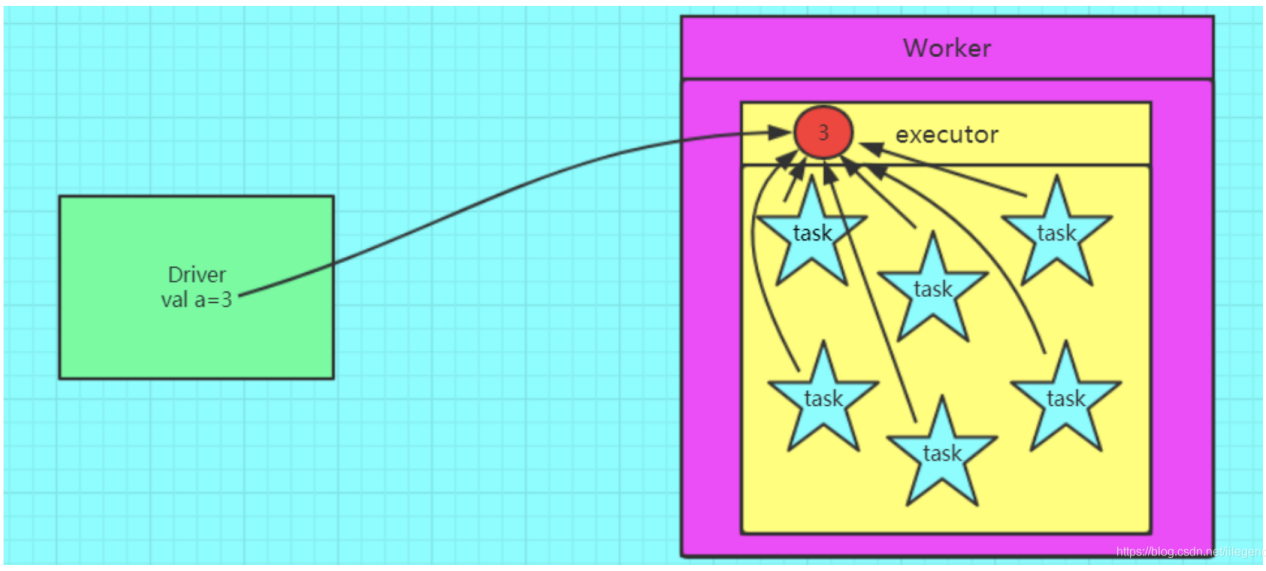
在数据关联场景中，广播变量是克制 Shuffle 的杀手锏。

一个形象的图例如下：

错误的，不使用广播变量



正确的，使用广播变量的情况



图来源于网络

在广播变量的运行机制下，普通变量存储的数据封装成广播变量，由 Driver 端以 Executors 为粒度进行分发，每一个 Executors 接收到广播变量之后，将其交由 BlockManager管理。

当然使用广播变量也有很多的制约，例如：

- 当创建完广播变量，后续不可以对广播变量进行修改，保证所有的节点都能获得相同的广播变量。
- 在数据量较大的情况下，Driver可能会成为瓶颈

shuffle相关参数调优

spark.shuffle.file.buffer

- 默认值：32k
- 参数说明：该参数用于设置shuffle write task的BufferedOutputStream的buffer缓冲大小。将数据写到磁盘文件之前，会先写入buffer缓冲中，待缓冲写满之后，才会溢写到磁盘。
- 调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如64k），从而减少shuffle write过程中溢写磁盘文件的次数，也就可以减少磁盘IO次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1%~5%的提升。

spark.reducer.maxSizeInFlight

- 默认值：48m
- 参数说明：该参数用于设置shuffle read task的buffer缓冲大小，而这个buffer缓冲决定了每次能够拉取多少数据。
- 调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如96m），从而减少拉取数据的次数，也就可以减少网络传输的次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1%~5%的提升。

spark.shuffle.io.maxRetries

- 默认值：3

- 参数说明：shuffle read task从shuffle write task所在节点拉取属于自己的数据时，如果因为网络异常导致拉取失败，是会自动进行重试的。该参数就代表了可以重试的最大次数。如果在指定次数之内拉取还是没有成功，就可能会导致作业执行失败。
- 调优建议：对于那些包含了特别耗时的shuffle操作的作业，建议增加重试最大次数（比如60次），以避免由于JVM的full gc或者网络不稳定等因素导致的数据拉取失败。在实践中发现，对于针对超大数据量（数十亿~上百亿）的shuffle过程，调节该参数可以大幅度提升稳定性。

spark.shuffle.io.retryWait

- 默认值：5s
- 参数说明：具体解释同上，该参数代表了每次重试拉取数据的等待间隔，默认是5s。
- 调优建议：建议加大间隔时长（比如60s），以增加shuffle操作的稳定性。

spark.shuffle.memoryFraction

- 默认值：0.2
- 参数说明：该参数代表了Executor内存中，分配给shuffle read task进行聚合操作的内存比例，默认是20%。
- 调优建议：在资源参数调优中讲解过这个参数。如果内存充足，而且很少使用持久化操作，建议调高这个比例，给shuffle read的聚合操作更多内存，以避免由于内存不足导致聚合过程中频繁读写磁盘。在实践中发现，合理调节该参数可以将性能提升10%左右。

spark.shuffle.manager

- 默认值：sort
- 参数说明：该参数用于设置ShuffleManager的类型。Spark 1.5以后，有三个可选项：hash、sort和tungsten-sort。HashShuffleManager是Spark 1.2以前的默认选项，但是Spark 1.2以及之后的版本默认都是SortShuffleManager了。tungsten-sort与sort类似，但是使用了tungsten计划中的堆外内存管理机制，内存使用效率更高。
- 调优建议：由于SortShuffleManager默认会对数据进行排序，因此如果你的业务逻辑中需要该排序机制的话，则使用默认的SortShuffleManager就可以；而如果你的业务逻辑不需要对数据进行排序，那么建议参考后面的几个参数调优，通过bypass机制或优化的HashShuffleManager来避免排序操作，同时提供较好的磁盘读写性能。这里要注意的是，tungsten-sort要慎用，因为之前发现了一些相应的bug。

spark.shuffle.sort.bypassMergeThreshold

- 默认值：200
- 参数说明：当ShuffleManager为SortShuffleManager时，如果shuffle read task的数量小于这个阈值（默认是200），则shuffle write过程中不会进行排序操作，而是直接按照未经优化的HashShuffleManager的方式去写数据，但是最后会将每个task产生的所有临时磁盘文件都合并成一个文件，并会创建单独的索引文件。
- 调优建议：当你使用SortShuffleManager时，如果的确不需要排序操作，那么建议将这个参数调大一些，大于shuffle read task的数量。那么此时就会自动启用bypass机制，map-side就不会进行排序了，减少了排序的性能开销。但是这种方式下，依然会产生大量的磁盘文件，因此shuffle write性能有待提高。

spark.shuffle.consolidateFiles

- 默认值：false
- 参数说明：如果使用HashShuffleManager，该参数有效。如果设置为true，那么就会开启consolidate机制，会大幅度合并shuffle write的输出文件，对于shuffle read task数量特别多的情况下，这种方法可以极大地减少磁盘IO开销，提升性能。
- 调优建议：如果的确不需要SortShuffleManager的排序机制，那么除了使用bypass机制，还可以尝试将spark.shffle.manager参数手动指定为hash，使用HashShuffleManager，同时开启consolidate机制。在实践中尝试过，发现其性能比开启了bypass机制的SortShuffleManager要高出10%~30%。