## 第 6 章 计算引擎

捭阖者，道之大化，说之变也。必豫审其变化。

——《鬼谷子》

本章导读

Spark的计算是一个层层迭代的过程。本章将5.5.3节与5.6.3两节中有关计算的内容单独抽出来，是为了让本书的内容及轮廓更清晰。读者可以先阅读第5章，也可以直接阅读本章内容，了解计算的有趣过程。

RDD作为Spark对各种数据计算模型的统一抽象，被用于迭代计算过程以及任务输出结果的缓存读写。在所有MapReduce框架中，shuffle是连接map任务和reduce任务的桥梁。map任务的中间输出要作为reduce任务的输入，就必须经过shuffle，shuffle的性能优劣直接决定了整个计算引擎的性能和吞吐量。相比于Hadoop的MapReduce，我们可以看到Spark提供多种计算结果处理的方式，对shuffle过程进行了优化。

本章将继续以word count为例讲解。

### 6.1 迭代计算

MappedRDD的iterator方法实际是父类RDD的iterator方法，见代码清单6-1。如果分区任务初次执行，此时还没有缓存，所以会调用computeOrReadCheckpoint方法。

这里需要说一下iterator方法的容错处理过程：如果某个分区任务执行失败，但是其他分区任务执行成功，可以利用DAG重新调度。失败的分区任务将从检查点恢复状态，而那些执行成功的分区任务由于其执行结果已经缓存到存储体系，所以调用CacheManager的getOrCompute方法获取即可，不需要再次执行。

代码清单6-1 iterator方法实现

final def iterator(split: Partition, context: TaskContext): Iterator[T] = { if (storageLevel != StorageLevel.NONE) { SparkEnv.get.cacheManager.getOrCompute(this, split, context, storageLevel) } else { computeOrReadCheckpoint(split, context) } }

CacheManager的有关内容已在4.10节介绍过，我们主要分析computeOrReadCheckpoint方法。computeOrReadCheckpoint在存在检查点时直接获取中间结果，否则需要调用compute继续计算，代码如下。

private[spark] def computeOrReadCheckpoint(split: Partition, context: TaskContext): Iterator[T] = { if (isCheckpointed) firstParent[T].iterator(split, context) else compute(split,context) }

MappedRDD的compute方法实现如下。

override def compute(split: Partition, context: TaskContext) = firstParent[T].iterator(split, context).map(f)

MappedRDD的compute方法首先调用firstParent找到其父RDD（有关firstParent方法的内容，请阅读5.3.2节），本例中MappedRDD的父RDD为FlatMappedRDD。FlatMappedRDD的iterator方法，其实也是RDD的iterator。

经过RDD管道中对iterator和computeOrReadCheckpoint的层层调用，最终到达Hadoop-RDD。查看此时的线程栈更直观，如图6-1所示。

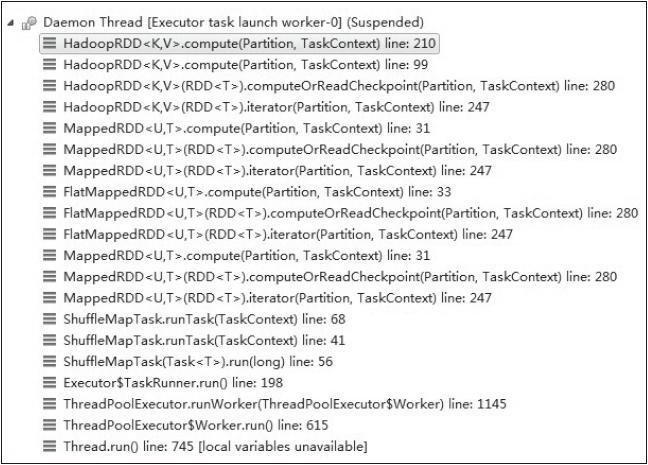


图6-1 word count例子的线程栈

HadoopRDD的compute方法用来创建NextIterator的匿名内部类，然后将其封装为InterruptibleIterator，见代码清单6-2。

代码清单6-2 HadoopRDD.compute的实现

override def compute(theSplit: Partition, context: TaskContext): Interruptible-Iterator[(K, V)] = { val iter = new NextIterator[(K, V)] { val split = theSplit.asInstanceOf[HadoopPartition] val jobConf = getJobConf() val inputMetrics = new InputMetrics(DataReadMethod.Hadoop) val bytesReadCallback = if (split.inputSplit.value.isInstanceOf[FileSplit]) { SparkHadoopUtil.get.getFSBytesReadOnThreadCallback( split.inputSplit.value.asInstanceOf[FileSplit].getPath, jobConf) } else { None } if (bytesReadCallback.isDefined) { context.taskMetrics.inputMetrics = Some(inputMetrics) } var reader: RecordReader[K, V] = null val inputFormat = getInputFormat(jobConf) HadoopRDD.addLocalConfiguration(new SimpleDateFormat("yyyyMMddHHmm").format(createTime), context.stageId, theSplit.index, context.attemptId.toInt, jobConf) reader = inputFormat.getRecordReader(split.inputSplit.value, jobConf, Reporter.NULL) context.addTaskCompletionListener{ context => closeIfNeeded() } val key: K = reader.createKey() val value: V = reader.createValue() var recordsSinceMetricsUpdate = 0 override def getNext() = { try { finished = !reader.next(key, value) } catch { case eof: EOFException => finished = true } if (recordsSinceMetricsUpdate == HadoopRDD.RECORDS\_BETWEEN\_BYTES\_READ\_METRIC\_UPDATES && bytesReadCallback.isDefined) { recordsSinceMetricsUpdate = 0 val bytesReadFn = bytesReadCallback.get inputMetrics.bytesRead = bytesReadFn() } else { recordsSinceMetricsUpdate += 1 } (key, value) } // 省略关闭RecordReader的代码 } new InterruptibleIterator[(K, V)](context, iter) }

构造NextIterator的过程如下：

1）从broadcast中获取jobConf，此处的jobConf正是代码清单5-3中的hadoopConfiguration。

2）创建InputMetrics用于计算字节读取的测量信息，然后在RecordReader正式读取数据之前创建bytesReadCallback。bytesReadCallback用于获取当前线程从文件系统读取的字节数。

3）获取inputFormat，此处的inputFormat正是代码清单5-2中的TextInputFormat。

4）使用addLocalConfiguration给JobConf添加Hadoop任务相关配置。addLocalConfiguration的实现见代码清单6-3。

代码清单6-3 HadoopRDD.addLocalConfiguration的实现

def addLocalConfiguration(jobTrackerId: String, jobId: Int, splitId: Int, attemptId: Int, conf: JobConf) { val jobID = new JobID(jobTrackerId, jobId) val taId = new TaskAttemptID(new TaskID(jobID, true, splitId), attemptId) conf.set("mapred.tip.id", taId.getTaskID.toString) conf.set("mapred.task.id", taId.toString) conf.setBoolean("mapred.task.is.map", true) conf.setInt("mapred.task.partition", splitId) conf.set("mapred.job.id", jobID.toString) }

5）创建RecordReader，调用reader.createKey（）和reader.createValue（）得到的正是代码清单5-2中的LongWritable和Text。NextIterator的getNext实际是代理了RecordReader的next方法并且每读取一些记录后使用bytesReadCallback更新InputMetrics的bytesRead字段。

6）将NextIterator封装为InterruptibleIterator。

InterruptibleIterator只是对NextIterator的代理，见代码清单6-4。

代码清单6-4 InterruptibleIterator的实现

class InterruptibleIterator[+T](val context: TaskContext, val delegate: Iterator[T]) extends Iterator[T] { def hasNext: Boolean = { if (context.isInterrupted) { throw new TaskKilledException } else { delegate.hasNext } } def next(): T = delegate.next() }

根据5.5.3节的内容，我们知道整个rdd.iterator调用结束，最后返回InterruptibleIterator对象后，会调用SortShuffleWriter的write方法（见代码清单6-5），其功能如下：

1）创建ExternalSorter，然后调用insertAll将计算结果写入缓存。

2）调用shuffleBlockManager.getDataFile方法获取当前任务要输出的文件路径，请参阅4.13节。

3）调用shuffleBlockManager.consolidateId创建blockId，请参阅4.13节所讲的ShuffleBlockId。

4）调用ExternalSorter的writePartitionedFile将中间结果持久化。

5）调用shuffleBlockManager.writeIndexFile方法创建索引文件。

6）创建MapStatus。

代码清单6-5 SortShuffleWriter.write的实现

override def write(records: Iterator[\_ <: Product2[K, V]]): Unit = { if (dep.mapSideCombine) { require(dep.aggregator.isDefined, "Map-side combine without Aggregator specified!") sorter = new ExternalSorter[K, V, C]( dep.aggregator, Some(dep.partitioner), dep.keyOrdering, dep.serializer) sorter.insertAll(records) } else { sorter = new ExternalSorter[K, V, V]( None, Some(dep.partitioner), None, dep.serializer) sorter.insertAll(records) } val outputFile = shuffleBlockManager.getDataFile(dep.shuffleId, mapId) val blockId = shuffleBlockManager.consolidateId(dep.shuffleId, mapId) val partitionLengths = sorter.writePartitionedFile(blockId, context, outputFile) shuffleBlockManager.writeIndexFile(dep.shuffleId, mapId, partitionLengths) mapStatus = MapStatus(blockManager.shuffleServerId, partitionLengths) }

### 6.2 什么是shuffle

shuffle是所有MapReduce计算框架所必须经过的阶段，shuffle用于打通map任务的输出与reduce任务的输入，map任务的中间输出结果按照key值哈希后分配给某一个reduce任务，这个过程如图6-2所示。

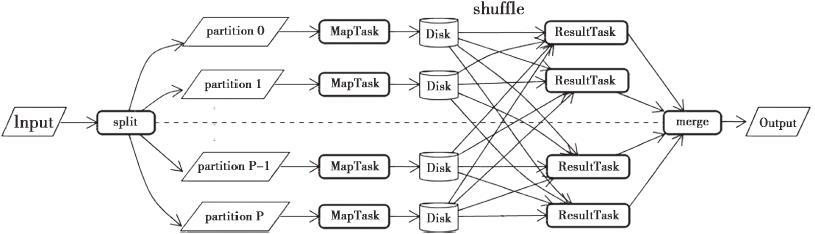


图6-2 MapReduce计算框架的shuffle过程示意图

在具体分析源码之前，我们先看看Spark早期版本的shuffle是怎样的，如图6-3所示。

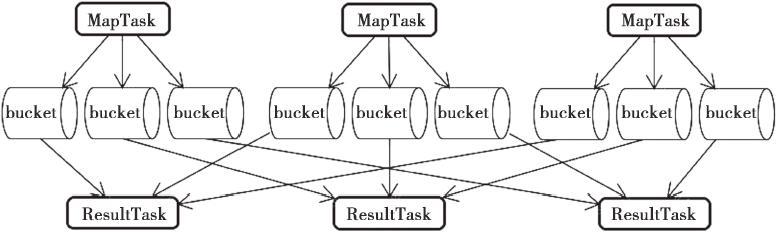


图6-3 Spark早期版本的shuffle过程

这里对图6-3做一些解释：

1）map任务会为每一个reduce任务创建一个bucket。假设有M个map任务，R个reduce任务，则map阶段一共会创建M×R个bucket；

2）map任务会将产生的中间结果按照partition写入不同的bucket中；

3）reduce任务从本地或者远端的map任务所在的BlockManager获取相应的bucket作为输入。

Spark早期的shuffle过程存在以下问题：

1）map任务的中间结果首先存入内存，然后才写入磁盘。这对于内存的开销很大，当一个节点上map任务的输出结果集很大时，很容易导致内存紧张，进而发生内存溢出（out of memory，OOM）；

2）每个map任务都会输出R（reduce任务数量）个bucket。假如M等于1000，R也等于1000，那么共计生成100万个bucket，在bucket本身不大，但是shuffle很频繁的情况下，磁盘I/O将成为性能瓶颈。

熟悉Hadoop的读者应该知道，Hadoop MapReduce的shuffle过程存在以下问题：

1）reduce任务获取到map任务的中间输出后，会对这些数据在磁盘上进行merge sort，虽然不怎么占用内存，但是却产生了更多的磁盘I/O；

2）当数据量很小，但是map任务和reduce任务数目很多时，会产生很多网络I/O。

为了解决以上Hadoop MapReduce和早期Spark在shuffle过程中的性能问题，目前Spark的shuffle已经做了多种性能优化，主要解决方法包括：

1）将map任务给每个partition的reduce任务输出的bucket合并到同一个文件中，这解决了bucket数量很多，但是本身数据体积不大时，造成shuffle很频繁，磁盘I/O成为性能瓶颈的问题；

2）map任务逐条输出计算结果，而不是一次性输出到内存，并使用AppendOnlyMap缓存及其聚合算法对中间结果进行聚合，这大大减小了中间结果所占的内存大小；

3）对SizeTrackingAppendOnlyMap和SizeTrackingPairBuffer等缓存进行溢出判断，当超出myMemoryThreshold的大小时，将数据写入磁盘，防止内存溢出；

4）reduce任务对拉取到的map任务中间结果逐条读取，而不是一次性读入内存，并在内存中进行聚合和排序（其本质上也使用了AppendOnlyMap缓存），这也大大减小了数据占用的内存；

5）reduce任务将要拉取的Block按照BlockManager地址划分，然后将同一BlockManager地址中的Block累积为少量网络请求，减少网络I/O。

在接下来的源码分析过程中，我们一起来看看这些问题是如何解决的。

### 6.3 map端计算结果缓存处理

在详细介绍map端对中间计算结果的细节之前，先理解两个概念：

·bypassMergeThreshold：传递到reduce端再做合并（merge）操作的阈值。如果partition的数量小于bypassMergeThreshold的值，则不需要在Executor执行聚合和排序操作，只需要将各个partition直接写到Executor的存储文件，最后在reduce端再做串联。通过配置spark.shuffle.sort.bypassMergeThreshold可以修改bypassMergeThreshold的大小，在分区数量小的时候提升计算引擎的性能。bypassMergeThreshold的默认值是200。

·bypassMergeSort：标记是否传递到reduce端再做合并和排序，即是否直接将各个partition直接写到Executor的存储文件。当没有定义aggregator、ordering函数，并且partition的数量小于等于bypassMergeThreshold时，bypassMergeSort为true。如果bypassMergeSort为true，map中间结果将直接输出到磁盘，此时不会占用太多内存，避免了内存撑爆问题。

map端计算结果缓存（见代码清单6-6）有三种处理方式：

·map端对计算结果在缓存中执行聚合和排序。

·map不使用缓存，也不执行聚合和排序，直接调用spillToPartitionFiles将各个partition直接写到自己的存储文件（即bypassMergeSort为true的情况），最后由reduce端对计算结果执行合并和排序。spillToPartitionFiles的实现，请参阅6.4.1节。

·map端对计算结果简单缓存。

代码清单6-6 ExternalSorter.insertAll的实现

def insertAll(records: Iterator[\_ <: Product2[K, V]]): Unit = { val shouldCombine = aggregator.isDefined if (shouldCombine) { // Combine values in-memory first using our AppendOnlyMap 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) maybeSpillCollection(usingMap = true) } } else if (bypassMergeSort) { if (records.hasNext) { spillToPartitionFiles(records.map { kv => ((getPartition(kv.\_1), kv.\_1), kv.\_2.asInstanceOf[C]) }) } } else { while (records.hasNext) { addElementsRead() val kv = records.next() buffer.insert((getPartition(kv.\_1), kv.\_1), kv.\_2.asInstanceOf[C]) maybeSpillCollection(usingMap = false) } } }

我们先来分析两种需要缓存的方式。

6.3.1 map端计算结果缓存聚合

一个任务的分区数量通常很多，如果只是简单地将数据存储到Executor上。在执行reduce任务时会存在大量的网络I/O操作，这时网络I/O将成为系统性能的瓶颈，reduce任务读取map任务的计算结果变慢，导致其他想要分配到被这些map任务占用的节点的任务不得不等待或者降低本地化选择分配到更远的节点上。对于更远节点的I/O本身会更慢，因此还会导致更多的任务得不到分配或者无法高效本地化。经过这样的恶性循环，整个集群将变得迟钝，新的任务长时间得不到执行或者执行变慢。

通过在map端对计算结果在缓存中执行聚合和排序，能够节省I/O操作，进而提升系统性能。这种情况下，必须要定义聚合器（aggregator）函数，以便于对计算结果按照partitionID和key聚合后排序。

ExternalSorter的insertAll方法（见代码清单6-6）中，如果定义了aggregator，则should-Combine为true。此分支执行过程如下：

1）由于设置了聚合函数aggregator，则从聚合函数获取mergeValue（word count例子中为Function2）、createCombiner（word count例子中为PairFunction）等函数。

2）定义update偏函数，此函数用于操作mergeValue和createCombiner。

3）迭代之前创建的iterator，每读取一条Product2[K，V]（此时真正执行代码清单5-1中的FlatMapFunction和PairFunction），将每行字符串按照空格切分，并且给每个文本设置1，比如（#，1）、（Apache，1）、（Spark，1）…。

4）以（分区索引，Product2[K，V].\_1）为参数调用SizeTrackingAppendOnlyMap的change-Value函数（见代码清单6-7），与update函数配合，按照key值叠加value。

5）调用maybeSpillCollection方法，来处理SizeTrackingAppendOnlyMap溢出（当Size-TrackingAppendOnlyMap的大小超过myMemoryThreshold时，将集合中的数据写入磁盘并新建SizeTrackingAppendOnlyMap）。这样做是为了防止内存溢出，解决了Spark早期版本shuffle的内存撑爆问题。

代码清单6-7 SizeTrackingAppendOnlyMap.changeValue方法的实现

override def changeValue(key: K, updateFunc: (Boolean, V) => V): V = { val newValue = super.changeValue(key, updateFunc) super.afterUpdate() newValue }

SizeTrackingAppendOnlyMap的changeValue方法的处理包括三步：

1）调用父类AppendOnlyMap的changeValue函数，应用缓存聚合算法。

2）调用继承特质SizeTracker的afterUpdate函数，增加对AppendOnlyMap大小的采样。

3）返回第1）步计算的结果。

1.AppendOnlyMap的缓存聚合算法

SizeTrackingAppendOnlyMap的父类AppendOnlyMap的changeValue函数（见代码清单6-8）用于回调update函数进行聚合操作。其实现可以说明，AppendOnlyMap支持null值的缓存，而Java的map默认是不支持的。changeValue方法利用一种使用数据缓存的算法完成聚合。在介绍此算法前先弄清一些定义：

·LOAD\_FACTOR：负载因子，常量值等于0.7。

·initialCapacity：初始容量值64。

·capacity：容量，初始时等于initialCapacity。

·curSize：记录当前已经放入data的key与聚合值的数量。

·data：数组，初始大小为2\*capacity，data数组的实际大小之所以是capacity的2倍，是因为key和聚合值各占一位。

·growThreshold：data数组容量增加的阈值，表达式为growThreshold＝LOAD\_FACTOR\*capacity。

·mask：计算数据存放位置的掩码值，表达式为capacity－1。

·k：要放入data的key。

·pos：k将要放入data的索引值。索引值等于k的哈希值再次计算哈希值的结果与mask按位&运算的值。表达式为pos＝rehash（k.hashCode）&mask。

·curKey：data（2\*pos）位置的当前key。

·newValue：key的聚合值。

在掌握以上概念的前提下，给出以下算法描述：

条件1：如果curKey等于null，那么newValue等于1；

条件2：如果curKey不等于null并且不等于k，那么从pos当前位置向后找，直到此位置的索引值与mask按位&运算后的新位置的key符合条件1或者条件3；

条件3：如果curKey不等于null并且等于k，那么newValue等于data（2\*pos＋1）与k对应的值按照mergeValue定义的函数运算（在word count例子中，mergeValue是代码清单5-1中的Function2函数）。

代码清单6-8 AppendOnlyMap.changeValue方法的实现

def changeValue(key: K, updateFunc: (Boolean, V) => V): V = { assert(!destroyed, destructionMessage) val k = key.asInstanceOf[AnyRef] if (k.eq(null)) { if (!haveNullValue) { incrementSize() } nullValue = updateFunc(haveNullValue, nullValue) haveNullValue = true return nullValue } var pos = rehash(k.hashCode) & mask var i = 1 while (true) { val curKey = data(2 \* pos) if (k.eq(curKey) || k.equals(curKey)) { val newValue = updateFunc(true, data(2 \* pos + 1).asInstanceOf[V]) data(2 \* pos + 1) = newValue.asInstanceOf[AnyRef] return newValue } else if (curKey.eq(null)) { val newValue = updateFunc(false, null.asInstanceOf[V]) data(2 \* pos) = k data(2 \* pos + 1) = newValue.asInstanceOf[AnyRef] incrementSize() return newValue } else { val delta = i pos = (pos + delta) & mask i += 1 } } null.asInstanceOf[V] // Never reached but needed to keep compiler happy }

2.AppendOnlyMap的容量增长

incrementSize方法（见代码清单6-9）用于扩充AppendOnlyMap的容量。当curSize＞growThreshold时，调用growTable方法将capacity容量扩大一倍，即newCapacity＝capacity\*2。

growTable方法（见代码清单6-10）先创建newCapacity的两倍大小的新数组，将老数组中的元素复制到新数组中，新数组索引位置采用新的mask重新使用rehash（k.hashCode）&mask计算。

代码清单6-9 AppendOnlyMap.incrementSize方法代码

private def incrementSize() { curSize += 1 if (curSize > growThreshold) { growTable() } }

代码清单6-10 AppendOnlyMap.growTable方法代码

protected def growTable() { val newCapacity = capacity \* 2 if (newCapacity >= (1 << 30)) { throw new Exception("Can't make capacity bigger than 2^29 elements") } val newData = new Array[AnyRef](2 \* newCapacity) val newMask = newCapacity - 1 var oldPos = 0 while (oldPos < capacity) { if (!data(2 \* oldPos).eq(null)) { val key = data(2 \* oldPos) val value = data(2 \* oldPos + 1) var newPos = rehash(key.hashCode) & newMask var i = 1 var keepGoing = true while (keepGoing) { val curKey = newData(2 \* newPos) if (curKey.eq(null)) { newData(2 \* newPos) = key newData(2 \* newPos + 1) = value keepGoing = false } else { val delta = i newPos = (newPos + delta) & newMask i += 1 } } } oldPos += 1 } data = newData capacity = newCapacity mask = newMask growThreshold = (LOAD\_FACTOR \* newCapacity).toInt }

经过以上算法的运算，word count例子的数据集合中间计算结果会变为（（0，site，），1）、（（0，which），2）、（（0，Hadoop），4）的样子，证明确实发生了聚合。

3.AppendOnlyMap大小采样

代码清单6-10列出了AppendOnlyMap的容量增长实现方法growTable，那是不是意味着AppendOnlyMap的容量可以无限制增长呢？当然不是，我们需要对AppendOnlyMap大小进行限制。很明显我们可以在AppendOnlyMap的每次更新操作之后计算它的大小，这应该没有什么问题。Spark为大数据平台需要提供实时计算能力，无论是数据量还是对CPU的开销都很大，每当发生update或者insert操作就计算一次大小会严重影响Spark的性能，因此Spark实际采用了采样并利用这些采样对AppendOnlyMap未来的大小进行估算或推测的方式。

SizeTrackingAppendOnlyMap继承了特质SizeTracker，其afterUpdate（见代码清单6-11）用于每次更新AppendOnlyMap的缓存后进行采样，采样前提是已经到达设定的采样间隔（nextSampleNum），其处理步骤如下：

1）将AppendOnlyMap所占的内存进行估算并且与当前编号（numUpdates）一起作为样本数据更新到samples＝new mutable.Queue[Sample]中。

2）如果当前采样数量大于2，则使samples执行一次出队操作，保证样本总数等于2。

3）计算每次更新增加的大小，公式如下：

00285.jpeg

如果样本数小于2，那么bytesPerUpdate＝0。

4）计算下次采样的间隔nextSampleNum。

代码清单6-11 SizeTracker.afterUpdate的实现

protected def afterUpdate(): Unit = { numUpdates += 1 if (nextSampleNum == numUpdates) { takeSample() } } private def takeSample(): Unit = { samples.enqueue(Sample(SizeEstimator.estimate(this), numUpdates)) // Only use the last two samples to extrapolate if (samples.size > 2) { samples.dequeue() } val bytesDelta = samples.toList.reverse match { case latest :: previous :: tail => (latest.size - previous.size).toDouble / (latest.numUpdates - previous.numUpdates) // If fewer than 2 samples, assume no change case \_ => 0 } bytesPerUpdate = math.max(0, bytesDelta) nextSampleNum = math.ceil(numUpdates \* SAMPLE\_GROWTH\_RATE).toLong }

AppendOnlyMap的大小采样数据用于推测AppendOnlyMap未来的大小，推测的实现见代码清单6-12。由于SizeTrackingPairBuffer也继承了SizeTracker，所以estimateSize方法不但对AppendOnlyMap也对SizeTrackingPairBuffer在内存中的容量进行限制以防内存溢出时发挥其作用。有关SizeTrackingPairBuffer的介绍请看6.3.2节。

代码清单6-12 SizeTracker.estimateSize方法代码

def estimateSize(): Long = { assert(samples.nonEmpty) val extrapolatedDelta = bytesPerUpdate \* (numUpdates - samples.last.numUpdates) (samples.last.size + extrapolatedDelta).toLong }

6.3.2 map端计算结果简单缓存

ExternalSorter的insertAll方法中，如果没有定义aggregator，那么shouldCombine为false，见代码清单6-6。这时会调用SizeTrackingPairBuffer的insert方法（见代码清单6-13），从其实现可以知道，它只不过是把计算结果简单地缓存到数组中了。

代码清单6-13 SizeTrackingPairBuffer.insert代码实现

def insert(key: K, value: V): Unit = { if (curSize == capacity) { growArray() } data(2 \* curSize) = key.asInstanceOf[AnyRef] data(2 \* curSize + 1) = value.asInstanceOf[AnyRef] curSize += 1 afterUpdate() }

下面我们来介绍SizeTrackingPairBuffer的容量增长。

SizeTrackingPairBuffer的容量增长是通过growArray方法实现的。growArray实现增长data数组容量的方式非常简单，只是新建2倍大小的新数组，然后简单复制而已，见代码清单6-14。

00126.jpeg

注意

新建的数组大小有可能超过Int类型的最大值，所以会抛出异常。

代码清单6-14 SizeTrackingPairBuffer.growArray代码实现

private def growArray(): Unit = { if (capacity == (1 << 29)) { throw new Exception("Can't grow buffer beyond 2^29 elements") } val newCapacity = capacity \* 2 val newArray = new Array[AnyRef](2 \* newCapacity) System.arraycopy(data, 0, newArray, 0, 2 \* capacity) data = newArray capacity = newCapacity resetSamples() }

Spark使用SizeTrackingPairBuffer的过程中，也会调用maybeSpillCollection方法，来处理SizeTrackingPairBuffer溢出（当SizeTrackingPairBuffer的大小超过myMemoryThreshold时，将集合中的数据写入磁盘并新建SizeTrackingPairBuffer）。这样做是为了防止内存溢出，解决了Spark早期版本shuffle的内存撑爆问题。

6.3.3 容量限制

既然AppendOnlyMap和SizeTrackingPairBuffer的容量都可以增长，那么数据量不大的时候不会有问题。但由于大数据处理的数据量往往都很大，全部都放入内存会将系统的内存撑爆。Spark为了防止这个问题的发生，提供了函数maybeSpillCollection，见代码清单6-15。

代码清单6-15 ExternalSorter.maybeSpillCollection代码实现

private def maybeSpillCollection(usingMap: Boolean): Unit = { if (!spillingEnabled) { return } if (usingMap) { if (maybeSpill(map, map.estimateSize())) { map = new SizeTrackingAppendOnlyMap[(Int, K), C] } } else { if (maybeSpill(buffer, buffer.estimateSize())) { buffer = new SizeTrackingPairBuffer[(Int, K), C] } } }

1.集合溢出判定

maybeSpillCollection判定集合是否溢出主要由maybeSpill函数来决定，见代码清单6-16。maybeSpill函数的处理步骤如下：

1）为当前线程尝试获取amountToRequest大小的内存（amountToRequest＝2\*current-Memory－myMemoryThreshold）。shuffleMemoryManager的tryToAcquire方法已在4.14节详细介绍过。

2）如果获得的内存依然不足（myMemoryThreshold＜＝currentMemory），则调用spill，执行溢出操作。内存不足可能是申请到的内存为0或者已经申请得到的内存大小超过了myMemoryThreshold。

3）溢出后续处理，如elementsRead归零，已溢出内存字节数（memoryBytesSpilled）增加线程当前内存大小（currentMemory），释放当前线程占用的内存。

代码清单6-16 Spillable.maybeSpill的实现

protected def maybeSpill(collection: C, currentMemory: Long): Boolean = { if (elementsRead > trackMemoryThreshold && elementsRead % 32 == 0 && currentMemory >= myMemoryThreshold) { val amountToRequest = 2 \* currentMemory - myMemoryThreshold val granted = shuffleMemoryManager.tryToAcquire(amountToRequest) myMemoryThreshold += granted if (myMemoryThreshold <= currentMemory) { \_spillCount += 1 logSpillage(currentMemory) spill(collection) \_elementsRead = 0 \_memoryBytesSpilled += currentMemory releaseMemoryForThisThread() return true } } false }

2.溢出

spill方法的实现，见代码清单6-17。如果bypassMergeSort为真，则调用spillToPartition-Files将内存中的数据溢出到分区文件，具体细节请参阅6.4.1节。如果bypassMergeSort不为真，则调用spillToMergeableFile。

代码清单6-17 ExternalSorter.spill方法的实现

override protected[this] def spill(collection: SizeTrackingPairCollection[(Int, K), C]): Unit = { if (bypassMergeSort) { spillToPartitionFiles(collection) } else { spillToMergeableFile(collection) } }

spillToMergeableFile方法（见代码清单6-18）的处理步骤如下：

1）调用createTempShuffleBlock创建临时文件，createTempShuffleBlock的实现已在4.4.3节讲述过。

2）新建ShuffleWriteMetrics用于测量。

3）调用getDiskWriter方法创建DiskBlockObjectWriter，具体内容参阅4.8.7节。

4）调用destructiveSortedIterator方法对集合元素排序，在6.4.2节将会详细介绍。

5）将集合内容写入临时文件。写入的时机有两个：

·集合遍历完的时候，执行flush。

·遍历过程中，每当写入DiskBlockObjectWriter的元素个数（objectsWritten）达到批量序列化尺寸（serializerBatchSize）时，也会执行flush，然后重新创建DiskBlock-ObjectWriter。

代码清单6-18 ExternalSorter.spillToMergeableFile方法的实现

private def spillToMergeableFile(collection: SizeTrackingPairCollection[(Int, K), C]): Unit = { val (blockId, file) = diskBlockManager.createTempShuffleBlock() curWriteMetrics = new ShuffleWriteMetrics() var writer = blockManager.getDiskWriter(blockId, file, ser, fileBufferSize, curWriteMetrics) var objectsWritten = 0 // Objects written since the last flush val batchSizes = new ArrayBuffer[Long] val elementsPerPartition = new Array[Long](numPartitions) def flush() = { val w = writer writer = null w.commitAndClose() \_diskBytesSpilled += curWriteMetrics.shuffleBytesWritten batchSizes.append(curWriteMetrics.shuffleBytesWritten) objectsWritten = 0 } var success = false try { val it = collection.destructiveSortedIterator(partitionKeyComparator) while (it.hasNext) { val elem = it.next() val partitionId = elem.\_1.\_1 val key = elem.\_1.\_2 val value = elem.\_2 writer.write(key) writer.write(value) elementsPerPartition(partitionId) += 1 objectsWritten += 1 if (objectsWritten == serializerBatchSize) { flush() curWriteMetrics = new ShuffleWriteMetrics() writer = blockManager.getDiskWriter(blockId, file, ser, fileBufferSize, curWriteMetrics) } } if (objectsWritten > 0) { flush() } else if (writer != null) { val w = writer writer = null w.revertPartialWritesAndClose() success = true } finally { if (!success) { if (writer != null) { writer.revertPartialWritesAndClose() } if (file.exists()) { file.delete() } } spills.append(SpilledFile(file, blockId, batchSizes.toArray, elementsPerPartition)) }

### 6.4 map端计算结果持久化

writePartitionedFile（见代码清单6-19）用于持久化计算结果。此方法有两个分支：

·溢出到分区文件后合并：将内存中缓存的多个partition的计算结果分别写入多个临时Block文件，然后将这些Block文件的内容全部写入正式的Block输出文件中。

·内存中排序合并：将缓存的中间计算结果按照partition分组后写入Block输出文件。此种方式还需要更新此任务与内存、磁盘有关的测量数据。

无论哪种排序方式，每个partition都会最终写入一个正式的Block文件，所以每个map任务实际上最后只会生成一个磁盘文件，最终解决了Spark早期版本中一个map任务输出的bucket文件过多和磁盘I/O成为性能瓶颈的问题。此外，无论哪种排序方式，每输出完一个partition的中间结果时，都会记录当前partition的长度，此长度将记录在索引文件中，以便下游任务的读取。

writePartitionedFile中有关DiskBlockObjectWriter的实现，请参阅4.12节。

代码清单6-19 ExternalSorter.writePartitionedFile的实现

def writePartitionedFile(blockId: BlockId, context: TaskContext, outputFile: File): Array[Long] = { val lengths = new Array[Long](numPartitions) if (bypassMergeSort && partitionWriters != null) { spillToPartitionFiles(if (aggregator.isDefined) map else buffer) partitionWriters.foreach(\_.commitAndClose()) var out: FileOutputStream = null var in: FileInputStream = null try { out = new FileOutputStream(outputFile, true) for (i <- 0 until numPartitions) { in = new FileInputStream(partitionWriters(i).fileSegment().file) val size = org.apache.spark.util.Utils.copyStream(in, out, false, transferToEnabled) in.close() in = null lengths(i) = size } } finally { if (out != null) { out.close() } if (in != null) { in.close() } } } else { for ((id, elements) <- this.partitionedIterator) { if (elements.hasNext) { val writer = blockManager.getDiskWriter( blockId, outputFile, ser, fileBufferSize, context.taskMetrics.shuffle-WriteMetrics.get) for (elem <- elements) { writer.write(elem) } writer.commitAndClose() val segment = writer.fileSegment() lengths(id) = segment.length } } } context.taskMetrics.memoryBytesSpilled += memoryBytesSpilled context.taskMetrics.diskBytesSpilled += diskBytesSpilled context.taskMetrics.shuffleWriteMetrics.filter(\_ => bypassMergeSort).foreach { m => if (curWriteMetrics != null) { m.shuffleBytesWritten += curWriteMetrics.shuffleBytesWritten m.shuffleWriteTime += curWriteMetrics.shuffleWriteTime } } lengths }

6.4.1 溢出分区文件

spillToPartitionFiles（见代码清单6-20）用于将内存中的集合数据按照每个partition创建一个临时Block文件，为每个临时Block文件生成一个DiskBlockObjectWriter，并且用Disk-BlockObjectWriter将计算结果分别写入这些临时Block文件中。createTempShuffleBlock方法创建临时的Block，具体内容请参阅4.4.3节。getDiskWriter方法创建DiskBlockObjectWriter，具体内容请参阅4.8.7节。

代码清单6-20 ExternalSorter的溢出分区文件实现

private def spillToPartitionFiles(collection: SizeTrackingPairCollection[(Int, K), C]): Unit = { spillToPartitionFiles(collection.iterator) } private def spillToPartitionFiles(iterator: Iterator[((Int, K), C)]): Unit = { assert(bypassMergeSort) if (partitionWriters == null) { curWriteMetrics = new ShuffleWriteMetrics() partitionWriters = Array.fill(numPartitions) { val (blockId, file) = diskBlockManager.createTempShuffleBlock() blockManager.getDiskWriter(blockId, file, ser, fileBufferSize, curWrite-Metrics).open() } } while (iterator.hasNext) { val elem = iterator.next() val partitionId = elem.\_1.\_1 val key = elem.\_1.\_2 val value = elem.\_2 partitionWriters(partitionId).write((key, value)) } }

代码清单6-19的持久化分支（即满足bypassMergeSort&&partitionWriters！＝null条件的代码分支）说明spillToPartitionFiles方法为每个partition生成的临时文件最后会逐个读取并统一写入正式的Block文件，如图6-4所示。spillToPartitionFiles方法在bypass-MergeSort为true，SizeTrackingAppend-OnlyMap或者SizeTrackingPairBuffer的大小超过myMemoryThreshold时被调用，以防止内存撑爆问题。此外，由于每个partition生成的临时文件最后会逐个读取并统一写入正式的Block文件，所以每个map任务实际上最后只会生成一个磁盘文件（相当于多个bucket被合并到一个文件中），最终解决了产生bucket文件过多和磁盘I/O成为性能瓶颈的问题。

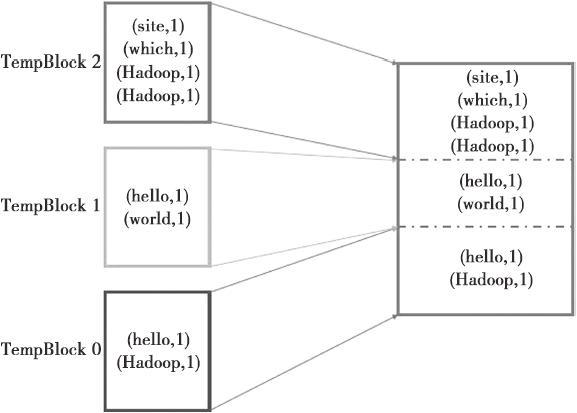


图6-4 溢出到分区文件后合并

6.4.2排序与分区分组

partitionedIterator（见代码清单6-21）通过对集合按照指定的比较器进行排序，并且按照partition id分组，生成迭代器。

代码清单6-21 ExternalSorter.partitionedIterator的实现

def partitionedIterator: Iterator[(Int, Iterator[Product2[K, C]])] = { val usingMap = aggregator.isDefined val collection: SizeTrackingPairCollection[(Int, K), C] = if (usingMap) map else buffer if (spills.isEmpty && partitionWriters == null) { if (!ordering.isDefined) { groupByPartition(collection.destructiveSortedIterator(partitionComparator)) } else { groupByPartition(collection.destructiveSortedIterator(partitionKeyComparator)) } } else if (bypassMergeSort) { val collIter = groupByPartition(collection.destructiveSortedIterator(partitionComparator)) collIter.map { case (partitionId, values) => (partitionId, values ++ readPartitionFile(partitionWriters(partitionId))) } } else { merge(spills, collection.destructiveSortedIterator(partitionKeyComparator)) } }

1.比较器

代码清单6-22列出了目前的三种比较器：

·keyComparator：按照指定的key进行排序；

·partitionComparator：按照partition id进行比较；

·partitionKeyComparator：先按照partition id进行比较，再按照指定的key进行第二级排序。当没有指定排序字段并且没有指定聚合函数时会退化为partitionComparator。

代码清单6-22 ExternalSorter中提供的几种比较器

private val keyComparator: Comparator[K] = ordering.getOrElse(new Comparator[K] { override def compare(a: K, b: K): Int = { val h1 = if (a == null) 0 else a.hashCode() val h2 = if (b == null) 0 else b.hashCode() if (h1 < h2) -1 else if (h1 == h2) 0 else 1 } }) private val partitionComparator: Comparator[(Int, K)] = new Comparator[(Int, K)] { override def compare(a: (Int, K), b: (Int, K)): Int = { a.\_1 - b.\_1 } } private val partitionKeyComparator: Comparator[(Int, K)] = { if (ordering.isDefined || aggregator.isDefined) { new Comparator[(Int, K)] { override def compare(a: (Int, K), b: (Int, K)): Int = { val partitionDiff = a.\_1 - b.\_1 if (partitionDiff != 0) { partitionDiff } else { keyComparator.compare(a.\_2, b.\_2) } } } } else { partitionComparator } }

由于partitionedIterator方法实际是通过调用destructiveSortedIterator和groupByPartition来实现，下面详细分析这两个方法。

2.排序

destructiveSortedIterator方法（见代码清单6-23）的处理步骤如下：

1）将data数组向左整理排列。

2）利用Sorter、KVArraySortDataFormat以及指定的比较器进行排序。这其中用到了TimSort，也就是优化版的归并排序。

3）生成新的迭代器。

代码清单6-23 AppendOnlyMap.destructiveSortedIterator的排序实现

def destructiveSortedIterator(keyComparator: Comparator[K]): Iterator[(K, V)] = { destroyed = true var keyIndex, newIndex = 0 while (keyIndex < capacity) { if (data(2 \* keyIndex) != null) { data(2 \* newIndex) = data(2 \* keyIndex) data(2 \* newIndex + 1) = data(2 \* keyIndex + 1) newIndex += 1 } keyIndex += 1 } assert(curSize == newIndex + (if (haveNullValue) 1 else 0)) new Sorter(new KVArraySortDataFormat[K, AnyRef]).sort(data, 0, newIndex, keyComparator) new Iterator[(K, V)] { var i = 0 var nullValueReady = haveNullValue def hasNext: Boolean = (i < newIndex || nullValueReady) def next(): (K, V) = { if (nullValueReady) { nullValueReady = false (null.asInstanceOf[K], nullValue) } else { val item = (data(2 \* i).asInstanceOf[K], data(2 \* i + 1).asInstanceOf[V]) i += 1 item } } } }

3.分区分组

groupByPartition（见代码清单6-24）主要用于对destructiveSortedIterator生成的迭代器按照partition id分组。

代码清单6-24 ExternalSorter.groupByPartition的分区分组代码

private def groupByPartition(data: Iterator[((Int, K), C)]) : Iterator[(Int, Iterator[Product2[K, C]])] = { val buffered = data.buffered (0 until numPartitions).iterator.map(p => (p, new IteratorForPartition(p, buffered))) }

IteratorForPartition如何区分partition呢？见代码清单6-25。可见其hasNext会判断数据的partitionId。

代码清单6-25 ExternalSorter.IteratorForPartition的实现

private[this] class IteratorForPartition(partitionId: Int, data: Buffered-Iterator[((Int, K), C)]) extends Iterator[Product2[K, C]] { override def hasNext: Boolean = data.hasNext && data.head.\_1.\_1 == partitionId override def next(): Product2[K, C] = { if (!hasNext) { throw new NoSuchElementException } val elem = data.next() (elem.\_1.\_2, elem.\_2) } }

6.4.3 分区索引文件

根据代码清单6-19我们知道，无论采用哪种缓存处理，在持久化的时候都会被写入同一文件，那么reduce任务如何从此文件中按照分区读取数据呢？还记得在代码清单6-5中调用IndexShuffleBlockManager的writeIndexFile方法生成的分区索引文件吗？此文件使用偏移量来区分各个分区的计算结果，偏移量来自于合并排序过程中记录的各个partition的长度。writeIndexFile的详细介绍请阅读4.13节。

这里用图6-5展示内存中排序、分组后生成分区索引文件的全过程。

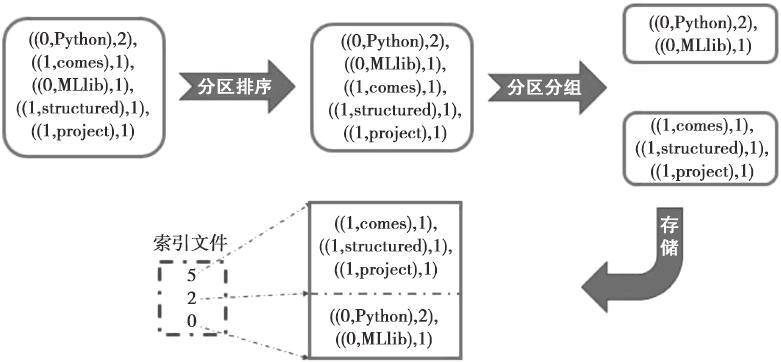


图6-5 排序、分组、生成分区索引文件的全过程

### 6.5 reduce端读取中间计算结果

先简单说下，当map任务相关Stage的任务都执行完毕后，会唤起下游Stage的提交及任务的执行。上游任务的执行结果必然是下游任务的输入，我们就下游任务如何读取上游任务计算结果来展开。

根据5.6.3节中“ResultTask任务的结果处理”部分的内容，我们知道ResultTask的计算也是由RDD的iterator方法驱动，这在介绍ShuffleMapTask的时候已经介绍过。其计算过程最终会落实到ShuffledRDD的compute方法。ShuffledRDD的compute方法（见代码清单6-26）首先调用SortShuffleManager的getReader方法创建HashShuffleReader，然后执行Hash-ShuffleReader的read方法读取依赖任务的中间计算结果。

代码清单6-26 ShuffledRDD.compute代码

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)]] }

SortShuffleManager的getReader方法的实现如下。

override def getReader[K, C]( handle: ShuffleHandle, startPartition: Int, endPartition: Int, context: TaskContext): ShuffleReader[K, C] = { new HashShuffleReader( handle.asInstanceOf[BaseShuffleHandle[K, \_, C]], startPartition, endPartition, context) }

HashShuffleReader用来读取上游任务计算结果，它的read方法（见代码清单6-27）的处理步骤如下：

1）从远端节点或者本地读取中间计算结果。

2）对InterruptibleIterator执行聚合。

3）对InterruptibleIterator排序，由于使用ExternalSorter的insertAll，不再赘述。

代码清单6-27 HashShuffleReader的read方法

override def read(): Iterator[Product2[K, C]] = { val ser = Serializer.getSerializer(dep.serializer) val iter = BlockStoreShuffleFetcher.fetch(handle.shuffleId, startPartition, context, ser) val aggregatedIter: Iterator[Product2[K, C]] = if (dep.aggregator.isDefined) { if (dep.mapSideCombine) { new InterruptibleIterator(context, dep.aggregator.get.combineCombiners-ByKey(iter, context)) } else { new InterruptibleIterator(context, dep.aggregator.get.combineValues-ByKey(iter, context)) } } else { require(!dep.mapSideCombine, "Map-side combine without Aggregator specified!") iter.asInstanceOf[Iterator[Product2[K, C]]].map(pair => (pair.\_1, pair.\_2)) } dep.keyOrdering match { case Some(keyOrd: Ordering[K]) => val sorter = new ExternalSorter[K, C, C](ordering = Some(keyOrd), serializer = Some(ser)) sorter.insertAll(aggregatedIter) context.taskMetrics.memoryBytesSpilled += sorter.memoryBytesSpilled context.taskMetrics.diskBytesSpilled += sorter.diskBytesSpilled sorter.iterator case None => aggregatedIter } }

从远端节点或者本地读取中间计算结果通过调用BlockStoreShuffleFetcher的fetch方法（见代码清单6-28）实现，它的处理步骤如下：

1）获取map任务执行的状态信息。

2）按照中间结果所在节点划分各个Block。

3）创建ShuffleBlockFetcherIterator（即从远端节点或者本地读取中间计算结果）。

4）将ShuffleBlockFetcherIterator封装为InterruptibleIterator。

代码清单6-28 BlockStoreShuffleFetcher的fetch方法

def fetch[T](shuffleId: Int, reduceId: Int, context: TaskContext, serializer: Serializer) : Iterator[T] = { logDebug("Fetching outputs for shuffle %d, reduce %d".format(shuffleId, reduceId)) val blockManager = SparkEnv.get.blockManager val startTime = System.currentTimeMillis val statuses = SparkEnv.get.mapOutputTracker.getServerStatuses(shuffleId, reduceId) logDebug("Fetching map output location for shuffle %d, reduce %d took %d ms".format( shuffleId, reduceId, System.currentTimeMillis - startTime)) val splitsByAddress = new HashMap[BlockManagerId, ArrayBuffer[(Int, Long)]] for (((address, size), index) <- statuses.zipWithIndex) { splitsByAddress.getOrElseUpdate(address, ArrayBuffer()) += ((index, size)) } val blocksByAddress: Seq[(BlockManagerId, Seq[(BlockId, Long)])] = splitsByAddress.toSeq.map { case (address, splits) => (address, splits.map(s => (ShuffleBlockId(shuffleId, s.\_1, reduceId), s.\_2))) } def unpackBlock(blockPair: (BlockId, Try[Iterator[Any]])) : Iterator[T] = { val blockId = blockPair.\_1 val blockOption = blockPair.\_2 blockOption match { case Success(block) => { block.asInstanceOf[Iterator[T]] } case Failure(e) => { //异常省略 } } } val blockFetcherItr = new ShuffleBlockFetcherIterator(context, SparkEnv.get.blockManager.shuffleClient, blockManager, blocksByAddress, serializer, SparkEnv.get.conf.getLong("spark.reducer.maxMbInFlight", 48) \* 1024 \* 1024) val itr = blockFetcherItr.flatMap(unpackBlock) val completionIter = CompletionIterator[T, Iterator[T]](itr, { context.taskMetrics.updateShuffleReadMetrics() }) new InterruptibleIterator[T](context, completionIter) }

6.5.1 获取map任务状态

Spark通过调用MapOutputTracker的getServerStatuses（见代码清单6-29）来获取map任务执行的状态信息，其处理步骤如下：

1）从当前BlockManager的MapOutputTracker中获取MapStatus，若没有就进入第2）步，否则直接到第4）步。

2）如果获取列表（fetching）中已经存在要取的shuffleId，那么就等待其他线程获取。如果获取列表中不存在要取的shuffleId，那么就将shuffleId放入获取列表。

3）调用askTracker方法（见代码清单6-30）向MapOutputTrackerMasterActor发送Get-MapOutputStatuses消息获取map任务的状态信息。MapOutputTrackerMasterActor接收到GetMapOutputStatuses消息后，将请求的map任务状态信息序列化后发送给请求方，见代码清单6-31。请求方接收到map任务状态信息后进行反序列化操作，然后放入本地的map-Statuses中。

4）调用MapOutputTracker的convertMapStatuses方法（见代码清单6-32）将获得的Map-Status转换为map任务所在的地址（即BlockManagerId）和map任务输出中分配给当前reduce任务的Block大小。

代码清单6-29 MapOutputTracker.getServerStatuses的代码

def getServerStatuses(shuffleId: Int, reduceId: Int): Array[(BlockManagerId, Long)] = { val statuses = mapStatuses.get(shuffleId).orNull if (statuses == null) { logInfo("Don't have map outputs for shuffle " + shuffleId + ", fetching them") var fetchedStatuses: Array[MapStatus] = null fetching.synchronized { // Someone else is fetching it; wait for them to be done while (fetching.contains(shuffleId)) { try { fetching.wait() } catch { case e: InterruptedException => } } fetchedStatuses = mapStatuses.get(shuffleId).orNull if (fetchedStatuses == null) { fetching += shuffleId } } if (fetchedStatuses == null) { try { val fetchedBytes = askTracker(GetMapOutputStatuses(shuffleId)).asInstance-Of[Array[Byte]] fetchedStatuses = MapOutputTracker.deserializeMapStatuses(fetchedBytes) logInfo("Got the output locations") mapStatuses.put(shuffleId, fetchedStatuses) } finally { fetching.synchronized { fetching -= shuffleId fetching.notifyAll() } } } if (fetchedStatuses != null) { fetchedStatuses.synchronized { return MapOutputTracker.convertMapStatuses(shuffleId, reduceId, fetchedStatuses) } } else { logError("Missing all output locations for shuffle " + shuffleId) throw new MetadataFetchFailedException( shuffleId, reduceId, "Missing all output locations for shuffle " + shuffleId) } } else { statuses.synchronized { return MapOutputTracker.convertMapStatuses(shuffleId, reduceId, statuses) } } }

代码清单6-30 MapOutputTracker.askTracker的实现

protected def askTracker(message: Any): Any = { try { val future = trackerActor.ask(message)(timeout) Await.result(future, timeout) } catch { case e: Exception => logError("Error communicating with MapOutputTracker", e) throw new SparkException("Error communicating with MapOutputTracker", e) } }

代码清单6-31 MapOutputTrackerMasterActor处理GetMapOutputStatuses消息

case GetMapOutputStatuses(shuffleId: Int) => val hostPort = sender.path.address.hostPort logInfo("Asked to send map output locations for shuffle " + shuffleId + " to " + hostPort) val mapOutputStatuses = tracker.getSerializedMapOutputStatuses(shuffleId) val serializedSize = mapOutputStatuses.size if (serializedSize > maxAkkaFrameSize) { val msg = s"Map output statuses were $serializedSize bytes which " + s"exceeds spark.akka.frameSize ($maxAkkaFrameSize bytes)." val exception = new SparkException(msg) logError(msg, exception) throw exception } sender ! mapOutputStatuses

代码清单6-32 map任务地址转换

private def convertMapStatuses( shuffleId: Int, reduceId: Int, statuses: Array[MapStatus]): Seq[(BlockManagerId, Seq[(BlockId, Long)])] = { assert (statuses != null) val splitsByAddress = new HashMap[BlockManagerId, ArrayBuffer[(BlockId, Long)]] for ((status, mapId) <- statuses.zipWithIndex) { if (status == null) { val errorMessage = s"Missing an output location for shuffle $shuffleId" logError(errorMessage) throw new MetadataFetchFailedException(shuffleId, reduceId, errorMessage) } else { splitsByAddress.getOrElseUpdate(status.location, ArrayBuffer()) += ((ShuffleBlockId(shuffleId, mapId, reduceId), status.getSizeForBlock(reduceId))) } } splitsByAddress.toSeq }

6.5.2 划分本地与远程Block

无论从本地还是从MapOutputTrackerMasterActor获取的状态信息，都需要按照地址划分并且转换为BlockId。ShuffleBlockFetcherIterator是读取中间结果的关键。构造Shuffle-BlockFetcherIterator的时候会调用到initialize方法（见代码清单6-33），它的初始化过程如下：

1）使用splitLocalRemoteBlocks方法划分本地读取和远程读取的Block的请求。

2）将FetchRequest随机排序后存入fetchRequests：newQueue[FetchRequest]。

3）遍历fetchRequests中的所有FetchRequest，远程请求Block中间结果。

4）调用fetchLocalBlocks获取本地Block。

代码清单6-33 ShuffleBlockFetcherIterator的初始化

private[this] def initialize(): Unit = { context.addTaskCompletionListener(\_ => cleanup()) val remoteRequests = splitLocalRemoteBlocks() fetchRequests ++= Utils.randomize(remoteRequests) while (fetchRequests.nonEmpty && (bytesInFlight == 0 || bytesInFlight + fetchRequests.front.size <= maxBytes-InFlight)) { sendRequest(fetchRequests.dequeue()) } val numFetches = remoteRequests.size - fetchRequests.size logInfo("Started " + numFetches + " remote fetches in" + Utils.getUsedTimeMs(startTime)) fetchLocalBlocks() logDebug("Got local blocks in " + Utils.getUsedTimeMs(startTime)) }

splitLocalRemoteBlocks方法（见代码清单6-34）用于划分哪些Block从本地获取，哪些需要远程拉取，是获取中间计算结果的关键。为便于描述，先解释以下定义：

·targetRequestSize：每个远程请求的最大尺寸。

·totalBlocks：统计Block总数。

·localBlocks：ArrayBuffer[BlockId]，缓存可以在本地获取的Block的blockId。

·remoteBlocks：HashSet[BlockId]，缓存需要远程获取的Block的blockId。

·curBlocks：ArrayBuffer[（BlockId，Long）]，远程获取的累加缓存，用于保证每个远程请求的尺寸不超过targetRequestSize。为什么要累加缓存？如果向一个机器节点频繁地请求字节数很小的Block，那么势必造成网络拥塞并增加节点负担。将多个小数据量的请求合并为一个大的请求将避免这些问题，提高系统性能。

·curRequestSize：当前累加到curBlocks中的所有Block的大小之和，用于保证每个远程请求的尺寸不超过targetRequestSize。

·remoteRequests：new ArrayBuffer[FetchRequest]，缓存需要远程请求的FetchRequest对象。

·numBlocksToFetch：一共要获取的Block数量。

·maxBytesInFlight：单次航班请求的最大字节数。什么叫航班？其实就是一批请求，这批请求的字节总数不能超过maxBytesInFlight，而且每个请求的字节数不能超过maxBytesInFlight的五分之一。可以通过参数spark.reducer.maxMbInFlight来控制大小。为什么每个请求的字节数不能超过maxBytesInFlight的五分之一？这样做是为了提高请求的并发度，允许5个请求分别从5个节点获取数据，最大限度利用各节点的资源。

明白了这些定义，我们一起来看看splitLocalRemoteBlocks的处理逻辑吧。

1）遍历已经在代码清单6-28中按照BlockManagerId分组的blockInfo，如果blockInfo所在的Executor与当前Executor相同，则将它的BlockId存入localBlocks；否则，将blockInfo的BlockId和size累加到curBlocks，将blockId存入remoteBlocks，curRequestSize增加size的大小，每当curRequestSize＞＝targetRequestSize，则新建FetchRequest放入remoteRequests，并且为生成下一个FetchRequest做一些准备（如新建curBlocks，curRequestSize置为0）。

2）遍历结束，curBlocks中如果仍然有缓存的（BlockId，Long），新建FetchRequest放入remoteRequests。此次请求不受maxBytesInFlight和targetRequestSize的影响。

代码清单6-34 ShuffleBlockFetcherIterator.splitLocalRemoteBlocks方法

private[this] def splitLocalRemoteBlocks(): ArrayBuffer[FetchRequest] = { val targetRequestSize = math.max(maxBytesInFlight / 5, 1L) val remoteRequests = new ArrayBuffer[FetchRequest] var totalBlocks = 0 for ((address, blockInfos) <- blocksByAddress) { totalBlocks += blockInfos.size if (address.executorId == blockManager.blockManagerId.executorId) { localBlocks ++= blockInfos.filter(\_.\_2 != 0).map(\_.\_1) numBlocksToFetch += localBlocks.size } else { val iterator = blockInfos.iterator var curRequestSize = 0L var curBlocks = new ArrayBuffer[(BlockId, Long)] while (iterator.hasNext) { val (blockId, size) = iterator.next() if (size > 0) { curBlocks += ((blockId, size)) remoteBlocks += blockId numBlocksToFetch += 1 curRequestSize += size } else if (size < 0) { throw new BlockException(blockId, "Negative block size " + size) } if (curRequestSize >= targetRequestSize) { remoteRequests += new FetchRequest(address, curBlocks) curBlocks = new ArrayBuffer[(BlockId, Long)] logDebug(s"Creating fetch request of $curRequestSize at $address") curRequestSize = 0 } } if (curBlocks.nonEmpty) { remoteRequests += new FetchRequest(address, curBlocks) } } } remoteRequests }

6.5.3 获取远程Block

sendRequest方法（见代码清单6-35）用于远程请求中间结果。sendRequest利用Fetch-Request里封装的blockId、size、address等信息，调用shuffleClient的fetchBlocks方法获取其他节点上的中间计算结果。shuffleClient.fetchBlocks方法可以参阅4.2.5节。

代码清单6-35 ShuffleBlockFetcherIterator.sendRequest方法

private[this] def sendRequest(req: FetchRequest) { logDebug("Sending request for %d blocks (%s) from %s".format( req.blocks.size, Utils.bytesToString(req.size), req.address.hostPort)) bytesInFlight += req.size val sizeMap = req.blocks.map { case (blockId, size) => (blockId.toString, size) }.toMap val blockIds = req.blocks.map(\_.\_1.toString) val address = req.address shuffleClient.fetchBlocks(address.host, address.port, address.executorId, blockIds.toArray, new BlockFetchingListener { override def onBlockFetchSuccess(blockId: String, buf: ManagedBuffer): Unit = { if (!isZombie) { buf.retain() results.put(new SuccessFetchResult(BlockId(blockId), sizeMap(blockId), buf)) shuffleMetrics.remoteBytesRead += buf.size shuffleMetrics.remoteBlocksFetched += 1 } logTrace("Got remote block " + blockId + " after " + Utils.getUsed-TimeMs(startTime)) } override def onBlockFetchFailure(blockId: String, e: Throwable): Unit = { logError(s"Failed to get block(s) from ${req.address.host}:${req.address.port}", e) results.put(new FailureFetchResult(BlockId(blockId), e)) } } ) }

6.5.4 获取本地Block

fetchLocalBlocks（见代码清单6-36）用于对本地中间计算结果的获取。fetchLocalBlocks方法很简单，利用熟悉的BlockManager的getBlockData方法获取本地Block，最后将取到的中间结果存入results＝new LinkedBlockingQueue[FetchResult]中。

代码清单6-36 ShuffleBlockFetcherIterator.fetchLocalBlocks方法

private[this] def fetchLocalBlocks() { val iter = localBlocks.iterator while (iter.hasNext) { val blockId = iter.next() try { val buf = blockManager.getBlockData(blockId) shuffleMetrics.localBlocksFetched += 1 buf.retain() results.put(new SuccessFetchResult(blockId, 0, buf)) } catch { case e: Exception => logError(s"Error occurred while fetching local blocks", e) results.put(new FailureFetchResult(blockId, e)) return } } }

### 6.6 reduce端计算

6.6.1 如何同时处理多个map任务的中间结果

reduce任务的上游map任务可能有多个，根据之前的分析，知道这些中间结果的Block及数据缓存在ShuffleBlockFetcherIterator的results：new LinkedBlockingQueue[FetchResult]中。ShuffleBlockFetcherIterator作为迭代器，它的实现见代码清单6-37。从其实现可知，每次迭代ShuffleBlockFetcherIterator，会先从results：new LinkedBlockingQueue[FetchResult]中取出一个FetchResult，并构造此FetchResult的迭代器iteratorTry，具体迭代的数据就是从iteratorTry中获取。每当iteratorTry迭代结束，才会再次迭代ShuffleBlockFetcherIterator。

代码清单6-37 ShuffleBlockFetcherIterator.scala

override def hasNext: Boolean = numBlocksProcessed < numBlocksToFetch override def next(): (BlockId, Try[Iterator[Any]]) = { numBlocksProcessed += 1 val startFetchWait = System.currentTimeMillis() currentResult = results.take() val result = currentResult val stopFetchWait = System.currentTimeMillis() shuffleMetrics.fetchWaitTime += (stopFetchWait - startFetchWait) result match { case SuccessFetchResult(\_, size, \_) => bytesInFlight -= size case \_ => } while (fetchRequests.nonEmpty && (bytesInFlight == 0 || bytesInFlight + fetchRequests.front.size <= maxBytesInFlight)) { sendRequest(fetchRequests.dequeue()) } val iteratorTry: Try[Iterator[Any]] = result match { case FailureFetchResult(\_, e) => Failure(e) case SuccessFetchResult(blockId, \_, buf) => Try(buf.createInputStream()).map { is0 => val is = blockManager.wrapForCompression(blockId, is0) val iter = serializer.newInstance().deserializeStream(is).asIterator CompletionIterator[Any, Iterator[Any]](iter, { currentResult = null buf.release() }) } } (result.blockId, iteratorTry) } }

00247.jpeg

注意

由于之前远程获取Block时，一小部分请求可能就达到了maxBytesInFlight的限制，所以很有可能会剩余很多请求没有发送。所以每此迭代ShuffleBlockFetcher-Iterator的时候还有个附加动作用于发送剩余请求。

6.6.2 reduce端在缓存中对中间计算结果执行聚合和排序

reduce端获取map端任务计算中间结果后，将ShuffleBlockFetcherIterator封装为Interru-ptibleIterator并聚合。聚合操作主要依赖aggregator的combineCombinersByKey方法，见代码清单6-38。如果isSpillEnabled为false，会再次使用AppendOnlyMap的changeValue方法，这些内容已在6.2节介绍AppendOnlyMap的缓存聚合算法时做过详细介绍。isSpillEnabled默认是true，此时会使用ExternalAppendOnlyMap完成聚合。

代码清单6-38 Aggregator.combineCombinersByKey的实现

def combineCombinersByKey(iter: Iterator[\_ <: Product2[K, C]], context: TaskContext) : Iterator[(K, C)] = { if (!isSpillEnabled) { val combiners = new AppendOnlyMap[K,C] var kc: Product2[K, C] = null val update = (hadValue: Boolean, oldValue: C) => { if (hadValue) mergeCombiners(oldValue, kc.\_2) else kc.\_2 } while (iter.hasNext) { kc = iter.next() combiners.changeValue(kc.\_1, update) } combiners.iterator } else { val combiners = new ExternalAppendOnlyMap[K, C, C](identity, mergeCombiners, mergeCombiners) while (iter.hasNext) { val pair = iter.next() combiners.insert(pair.\_1, pair.\_2) } Option(context).foreach { c => c.taskMetrics.memoryBytesSpilled += combiners.memoryBytesSpilled c.taskMetrics.diskBytesSpilled += combiners.diskBytesSpilled } combiners.iterator } }

ExternalAppendOnlyMap的insert方法的实际工作是由insertAll完成的，见代码清单6-39。从代码实现可以看到其实质也是使用SizeTrackingAppendOnlyMap，已在介绍AppendOnlyMap的缓存聚合算法时做过详细介绍。

代码清单6-39 ExternalAppendOnlyMap的insert方法

def insert(key: K, value: V): Unit = { insertAll(Iterator((key, value))) } def insertAll(entries: Iterator[Product2[K, V]]): Unit = { var curEntry: Product2[K, V] = null val update: (Boolean, C) => C = (hadVal, oldVal) => { if (hadVal) mergeValue(oldVal, curEntry.\_2) else createCombiner(curEntry.\_2) } while (entries.hasNext) { curEntry = entries.next() if (maybeSpill(currentMap, currentMap.estimateSize())) { currentMap = new SizeTrackingAppendOnlyMap[K, C] } currentMap.changeValue(curEntry.\_1, update) addElementsRead() } }

经过以上处理，数据结果为类似（##，8），（N，1），（set，2），（use，3）， （Hadoop-supported，1）的样子。

### 6.7 map端与reduce端组合分析

这一节主要对计算引擎部分的内容进行串联，用图来展示最常见的几种组合，以便大家对计算引擎有个宏观的认识。其中的具体执行过程，已在本章之前的内容中介绍过，此处不再赘述。

6.7.1 在map端溢出分区文件，在reduce端合并组合

bypassMergeSort标记是否传递到reduce端再做合并和排序，此种情况不使用缓存，而是先将数据按照partition写入不同文件，最后按partition顺序合并写入同一文件。当没有指定聚合、排序函数，且partition数量较小时，一般采用这种方式。此种方式将多个bucket合并到同一个文件，通过减少map输出的文件数量，节省了磁盘I/O，最终提升了性能，见图6-6。

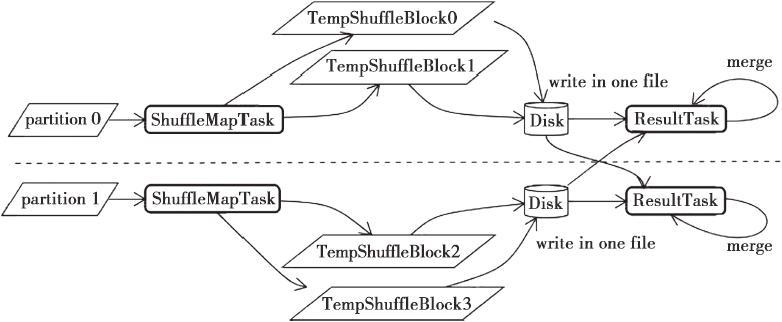


图6-6 map端溢出分区文件与reduce端合并组合

6.7.2 在map端简单缓存、排序分组，在reduce端合并组合

此种情况在缓存中利用指定的排序函数对数据按照partition或者key进行排序，最后按partition顺序合并写入同一文件。当没有指定聚合函数，且partition数量大时，一般采用这种方式，见图6-7。此种方式将多个bucket合并到同一个文件，通过减少map输出的文件数量，节省了磁盘I/O，提升了性能；对SizeTrackingPairBuffer的缓存进行溢出判断，当超出myMemoryThreshold的大小时，将数据写入磁盘，防止内存溢出。

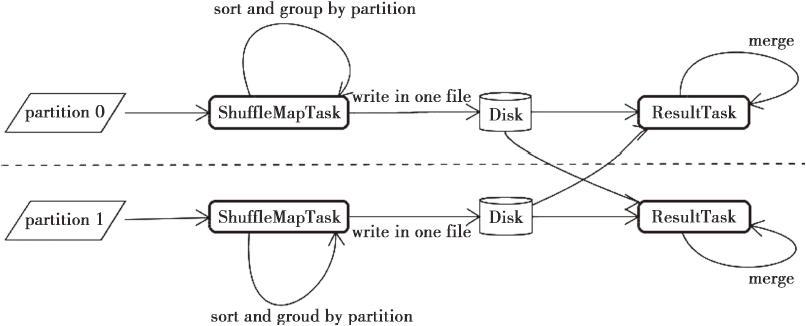


图6-7 map端简单缓存、排序分组与reduce端合并组合

6.7.3 在map端缓存中聚合、排序分组，在reduce端组合

此种情况在缓存中对数据按照key聚合，并且利用指定的排序函数对数据按照partition或者key进行排序，最后按partition顺序合并写入同一文件。当指定了聚合函数时，一般采用这种方式，见图6-8。此种方式将多个bucket合并到同一个文件，通过减少map输出的文件数量，节省了磁盘I/O，提升了性能；对中间输出数据不是一次性读取，而是逐条放入AppendOnlyMap的缓存，并对数据进行聚合，减少了中间结果占用的内存大小；对AppendOnlyMap的缓存进行溢出判断，当超出myMemoryThreshold的大小时，将数据写入磁盘，防止内存溢出。

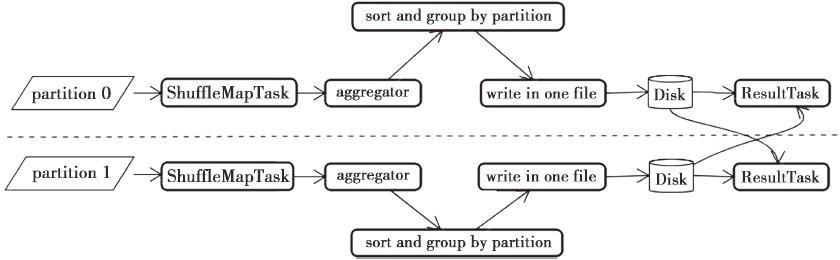


图6-8 map端缓存中聚合、排序分组与reduce端组合

### 6.8 小结

本章从迭代计算的层层剥离开始，分别分析了map和reduce任务的处理逻辑。这其中不乏值得读者学习的代码实现，比如：

·RDD迭代计算是如何实现容错的？基于缓存和检查点。

·map任务是如何聚合的？使用AppendOnlyMap提供的数据结构来实现聚合。

·为什么有时候需要在map端聚合？为了降低网络I/O，提升性能。

·map任务如何输出？按照分区排序或者分组后生成分区文件，并创建分区索引文件标记文件中各个分区数据的偏移量和长度。

·reduce任务是如何获取map任务的输出的？通过mapOutputTracker获取到map任务所在Executor的BlockManagerId和Block的大小，然后使用shuffleClient下载。

·发送请求时有哪些性能优化？对请求分批发送，限制分批请求的大小，并行发送请求以及将多个请求数据小的请求合并等。