Spark分区器HashPartitioner和RangePartitioner代码详解

在Spark中分区器直接决定了RDD中分区的个数；也决定了RDD中每条数据经过Shuffle过程属于哪个分区；也决定了Reduce的个数。这三点看起来是不同的方面的，但其深层的含义是一致的。

我们需要注意的是，只有Key-Value类型的RDD才有分区的，非Key-Value类型的RDD分区的值是None的。

在Spark中，存在两类分区函数：HashPartitioner和RangePartitioner，它们都是继承自Partitioner，主要提供了每个RDD有几个分区（numPartitions）以及对于给定的值返回一个分区ID（0~numPartitions-1），也就是决定这个值是属于那个分区的。

HashPartitioner分区

HashPartitioner分区的原理很简单，对于给定的key，计算其hashCode，并除于分区的个数取余，如果余数小于0，则用余数+分区的个数，最后返回的值就是这个key所属的分区ID。实现如下：

```
//////////////////////////////////////////////////////////////////////
 User: 过往记忆
 Date: 2015-11-10
 Time: 06:59
 bolg:
 本文地址：/archives/1522
 过往记忆博客，专注于hadoop、hive、spark、shark、flume的技术博客，大量的干货
 过往记忆博客微信公共帐号：iteblog_hadoop
//////////////////////////////////////////////////////////////////////

class HashPartitioner(partitions: Int) extends Partitioner {
  require(partitions >= 0, s"Number of partitions ($partitions) cannot be negative.")

  def numPartitions: Int = partitions

  def getPartition(key: Any): Int = key match {
    case null => 0
    case _ => Utils.nonNegativeMod(key.hashCode, numPartitions)
  }

  override def equals(other: Any): Boolean = other match {
    case h: HashPartitioner =>
      h.numPartitions == numPartitions
    case _ =>
```

```
    false
  }

  override def hashCode: Int = numPartitions
}
```

RangePartitioner分区

从HashPartitioner分区的实现原理我们可以看出，其结果可能导致每个分区中数据量的不均匀，极端情况下会导致某些分区拥有RDD的全部数据，这显然不是我们需要的。而RangePartitioner分区则尽量保证每个分区中数据量的均匀，而且分区与分区之间是有序的，也就是说一个分区中的元素肯定都是比另一个分区内的元素小或者大；但是分区内的元素是不能保证顺序的。简单的说就是将一定范围内的数映射到某一个分区内。

前面讨论过，RangePartitioner分区器的主要作用就是将一定范围内的数映射到某一个分区内，所以它的实现中分界的算法尤为重要。这个算法对应的函数是rangeBounds。这个函数主要经历了两个过程：以Spark 1.1版本为界，Spark 1.1版本社区对rangeBounds函数进行了一次重大的重构。

因为在Spark 1.1版本之前，RangePartitioner分区对整个数据集进行了2次的扫描：一次是计算RDD中元素的个数；一次是进行采样。具体的代码如下：

```
// An array of upper bounds for the first (partitions - 1) partitions
private val rangeBounds: Array[K] = {
  if (partitions == 1) {
    Array()
  } else {
    val rddSize = rdd.count()
    val maxSampleSize = partitions * 20.0
    val frac = math.min(maxSampleSize / math.max(rddSize, 1), 1.0)
    val rddSample = rdd.sample(false, frac, 1).map(_._1).collect().sorted
    if (rddSample.length == 0) {
      Array()
    } else {
      val bounds = new Array[K](partitions - 1)
      for (i <- 0 until partitions - 1) {
        val index = (rddSample.length - 1) * (i + 1) / partitions
        bounds(i) = rddSample(index)
      }
      bounds
    }
  }
}
```

注意看里面的rddSize的计算和rdd.sample的计算。所以如果你进行一次sortByKey操作就会对RDD扫描三次！而我们都知道，分区函数性能对整个Spark作业的性能是有直接的影响，而且影响很大，直接影响作业运行的总时间，所以社区不得不对RangePartitioner中的rangeBounds算法进行重构。

在阅读新版本的RangePartitioner之前，建议先去了解一下Reservoir sampling（水塘抽样），因为其中的实现用到了Reservoir sampling算法进行采样。

采样总数

在新的rangeBounds算法总，采样总数做了一个限制，也就是最大只采样1e6的样本（也就是1000000）：

```
val sampleSize = math.min(20.0 * partitions, 1e6)
```

所以如果你的分区个数为5，则采样样本数量为100.0

父RDD中每个分区采样样本数

按照我们的思路，正常情况下，父RDD每个分区需要采样的数据量应该是sampleSize/rdd.partitions.size，但是我们看代码的时候发现父RDD每个分区需要采样的数据量是正常数的3倍。

```
val sampleSizePerPartition = math.ceil(3.0 * sampleSize / rdd.partitions.size).toInt
```

这是因为父RDD各分区中的数据量可能会出现倾斜的情况，乘于3的目的就是保证数据量小的分区能够采样到足够的数据，而对于数据量大的分区会进行第二次采样。

采样算法

这个地方就是RangePartitioner分区的核心了，其内部使用的就是水塘抽样，而这个抽样特别适合那种总数很大而且未知，并无法将所有的数据全部存放到主内存中的情况。也就是我们不需要事先知道RDD中元素的个数（不需要调用rdd.count()了！）。其主要实现如下：

```
//////////////////////////////////////////////////////////////////////
 User: 过往记忆
 Date: 2015-11-10
 Time: 06:59
```

```
val (numItems, sketched) = RangePartitioner.sketch(rdd.map(_._1), sampleSizePerPartition)

def sketch[K : ClassTag](
      rdd: RDD[K],
      sampleSizePerPartition: Int): (Long, Array[(Int, Int, Array[K])]) = {
    val shift = rdd.id
    // val classTagK = classTag[K] // to avoid serializing the entire partitioner object
    val sketched = rdd.mapPartitionsWithIndex { (idx, iter) =>
      val seed = byteswap32(idx ^ (shift << 16))
      val (sample, n) = SamplingUtils.reservoirSampleAndCount(
        iter, sampleSizePerPartition, seed)
      Iterator((idx, n, sample))
    }.collect()
    val numItems = sketched.map(_._2.toLong).sum
    (numItems, sketched)
}

def reservoirSampleAndCount[T: ClassTag](
      input: Iterator[T],
      k: Int,
      seed: Long = Random.nextLong())
    : (Array[T], Int) = {
    val reservoir = new Array[T](k)
    // Put the first k elements in the reservoir.
    var i = 0
    while (i < k && input.hasNext) {
      val item = input.next()
      reservoir(i) = item
      i += 1
    }

    // If we have consumed all the elements, return them. Otherwise do the replacement.
    if (i < k) {
      // If input size < k, trim the array to return only an array of input size.
      val trimReservoir = new Array[T](i)
      System.arraycopy(reservoir, 0, trimReservoir, 0, i)
      (trimReservoir, i)
    } else {
      // If input size > k, continue the sampling process.
      val rand = new XORShiftRandom(seed)
```

```
    while (input.hasNext) {
      val item = input.next()
      val replacementIndex = rand.nextInt(i)
      if (replacementIndex < k) {
        reservoir(replacementIndex) = item
      }
      i += 1
    }
    (reservoir, i)
}
}
```

RangePartitioner.sketch的第一个参数是rdd.map(_._1)，也就是把父RDD的key传进来，因为分区只需要对Key进行操作即可。该函数返回值是val (numItems, sketched)，其中numItems相当于记录rdd元素的总数；而sketched的类型是Array[(Int, Int, Array[K])]，记录的是分区的编号、该分区中总元素的个数以及从父RDD中每个分区采样的数据。

sketch函数对父RDD中的每个分区进行采样，并记录下分区的ID和分区中数据总和。

reservoirSampleAndCount函数就是典型的水塘抽样实现，唯一不同的是该算法还记录下i的值，这个就是该分区中元素的总和。

我们之前讨论过，父RDD各分区中的数据量可能不均匀，在极端情况下，有些分区内的数据量会占有整个RDD的绝大多数的数据，如果按照水塘抽样进行采样，会导致该分区所采样的数据量不足，所以我们需要对该分区再一次进行采样，而这次采样使用的就是rdd的sample函数。实现如下：

```
val fraction = math.min(sampleSize / math.max(numItems, 1L), 1.0)
val candidates = ArrayBuffer.empty[(K, Float)]
val imbalancedPartitions = mutable.Set.empty[Int]
sketched.foreach { case (idx, n, sample) =>
  if (fraction * n > sampleSizePerPartition) {
    imbalancedPartitions += idx
  } else {
    // The weight is 1 over the sampling probability.
    val weight = (n.toDouble / sample.size).toFloat
    for (key <- sample) {
      candidates += ((key, weight))
    }
  }
}
if (imbalancedPartitions.nonEmpty) {
  // Re-sample imbalanced partitions with the desired sampling probability.
```

```
  val imbalanced = new PartitionPruningRDD(rdd.map(_._1), imbalancedPartitions.contains)
  val seed = byteswap32(-rdd.id - 1)
  val reSampled = imbalanced.sample(withReplacement = false, fraction, seed).collect()
  val weight = (1.0 / fraction).toFloat
  candidates ++= reSampled.map(x => (x, weight))
}
```

我们可以看到，重新采样的采样因子和Spark 1.1之前的采样因子一致。对于满足于fraction * n > sampleSizePerPartition条件的分区，我们对其再一次采样。所有采样完的数据全部存放在candidates 中。

确认边界

从上面的采样算法可以看出，对于不同的分区weight的值是不一样的，这个值对应的就是每个分区的采样间隔。

```
def determineBounds[K : Ordering : ClassTag](
    candidates: ArrayBuffer[(K, Float)],
    partitions: Int): Array[K] = {
  val ordering = implicitly[Ordering[K]]
  val ordered = candidates.sortBy(_._1)
  val numCandidates = ordered.size
  val sumWeights = ordered.map(_._2.toDouble).sum
  val step = sumWeights / partitions
  var cumWeight = 0.0
  var target = step
  val bounds = ArrayBuffer.empty[K]
  var i = 0
  var j = 0
  var previousBound = Option.empty[K]
  while ((i < numCandidates) && (j < partitions - 1)) {
    val (key, weight) = ordered(i)
    cumWeight += weight
    if (cumWeight > target) {
      // Skip duplicate values.
      if (previousBound.isEmpty || ordering.gt(key, previousBound.get)) {
        bounds += key
        target += step
        j += 1
        previousBound = Some(key)
      }
    }
```

```
    i += 1
  }
  bounds.toArray
}
```

这个函数最后返回的就是分区的划分边界。

注意，按照理想情况，选定的划分边界需要保证划分后的分区中数据量是均匀的，但是这个算法中如果将cumWeight > target修改成cumWeight >= target的时候会保证各分区之间数据量更加均衡。可以看这里https://issues.apache.org/jira/browse/SPARK-10184。

定位分区ID

分区类的一个重要功能就是对给定的值计算其属于哪个分区。这个算法并没有太大的变化。

```
def getPartition(key: Any): Int = {
  val k = key.asInstanceOf[K]
  var partition = 0
  if (rangeBounds.length <= 128) {
    // If we have less than 128 partitions naive search
    while (partition < rangeBounds.length && ordering.gt(k, rangeBounds(partition))) {
      partition += 1
    }
  } else {
    // Determine which binary search method to use only once.
    partition = binarySearch(rangeBounds, k)
    // binarySearch either returns the match location or -[insertion point]-1
    if (partition < 0) {
      partition = -partition-1
    }
    if (partition > rangeBounds.length) {
      partition = rangeBounds.length
    }
  }
  if (ascending) {
    partition
  } else {
    rangeBounds.length - partition
  }
}
```

如果分区边界数组的大小小于或等于128的时候直接变量数组，否则采用二分查找法确定key属于某个分区。