Spark SQL小文件问题在OPPO的解决方案

Spark SQL小文件是指文件大小显著小于hdfs block块大小的的文件。过于繁多的小文件会给HDFS带来很严重的性能瓶颈，对任务的稳定和集群的维护会带来极大的挑战。

一般来说，通过Hive调度的MR任务都可以简单设置如下几个小文件合并的参数来解决任务产生的小文件问题：

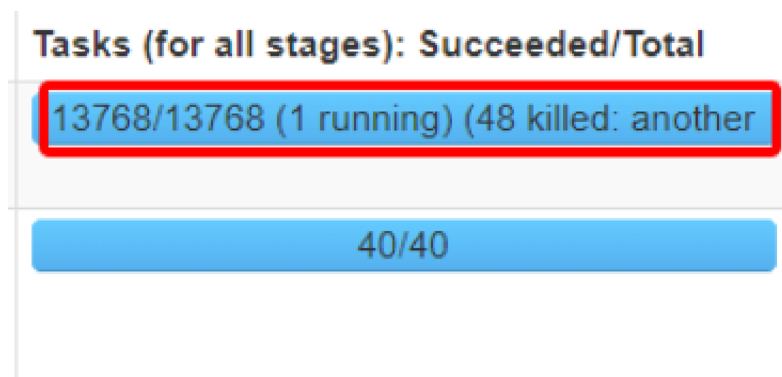
```
set hive.merge.mapfiles=true;
set hive.merge.mapredfiles=true;
set hive.merge.size.per.task=xxxx;
set hive.merge.smallfiles.avgsize=xxx;
```

然而在我们将离线调度任务逐步从Hive迁移到Spark的过程中，由于Spark本身并不支持小文件合并功能，小文件问题日益突出，对集群稳定性造成很大影响，一度阻碍了我们的迁移工作。

为了解决小文件问题，我们经历了从开始的不断调整参数到后期的代码开发等不同阶段，这里给大家做一个简单的分享。

1. Spark为什么会产生小文件

Spark生成的文件数量直接取决于RDD里partition的数量和表分区数量。注意这里的两个分区概念并不相同，RDD的分区与任务并行度相关，而表分区则是Hive的分区数目。生成的文件数目一般是RDD分区数和表分区的乘积。因此，当任务并行度过高或者分区数目很大时，很容易产生很多的小文件。



如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

因此，如果需要从参数调整来减少生成的文件数目，就只能通过减少最后一个阶段RDD的分区数

来达到了（减少分区数目限制于历史数据和上下游关系，难以修改）

2. 基于社区版本的参数进行调整的方案

2.1 不含有Shuffle算子的简单静态分区SQL

这样的SQL比较简单，主要是filter上游表一部分数据写入到下游表，或者是两张表简单UNION起来的任务，这种任务的分区数目主要是由读取文件时Partition数目决定的。

因为从Spark 2.4以来，对Hive
orc表和parquet支持已经很不错了，为了加快运行速率，我们开启了将Hive orc/parquet表自动转为DataSource的参数。对于这种DataSource表的类型，partition数目主要是由如下三个参数控制其关系。

spark.sql.files.maxPartitionBytes；
spark.sql.files.opencostinbytes；
spark.default.parallelism；

其关系如下图所示，因此可以通过调整这三个参数来输入数据的分片进行调整：

```
val defaultMaxSplitBytes =
  fsRelation.sparkSession.sessionState.conf.filesMaxPartitionBytes
val openCostInBytes = fsRelation.sparkSession.sessionState.conf.filesOpenCostInBytes
val defaultParallelism = fsRelation.sparkSession.sparkContext.defaultParallelism
val totalBytes = selectedPartitions.flatMap(_.files.map(_.getLen + openCostInBytes)).sum
val bytesPerCore = totalBytes / defaultParallelism

val maxSplitBytes = Math.min(defaultMaxSplitBytes, Math.max(openCostInBytes, bytesPerCore))
```

如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

而非DataSource表，使用CombineInputFormat来读取数据，因此主要是通过MR参数来进行分片调整：

mapreduce.input.fileinputformat.split.minsize

虽然我们可以通过调整输入数据的分片来对最终文件数量进行调整，但是这样的调整是不稳定的，上游数据大小发生一些轻微的变化，就可能带来参数的重新适配。

为了简单粗暴的解决这个问题，我们对这样的SQL加了repartition的hint，引入了新的shuffle，保

证文件数量是一个固定值。

2.2 带有Shuffle算子的静态分区任务

在ISSUE SPARK-9858中，引入了一个新的参数：

spark.sql.adaptive.shuffle.targetPostShuffleInputSize，

后期基于spark adaptive又对这个参数做了进一步增强，可以动态的调整partition数量，尽可能保证每个task处理targetPostShuffleInputSize大小的数据，因此这个参数我们也可以用来在一定程度上控制生成的文件数量。

```
-- 修改前
INSERT OVERWRITE TABLE a
SELECT imei, name
FROM b
WHERE imei = "xxxx" AND dayno = 20200611;
--修改后
INSERT OVERWRITE TABLE a
SELECT /*+ REPARTITION(10) */
imei, name
FROM b
WHERE imei = "xxxx" AND dayno = 20200611;
```

如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

2.3 动态分区任务

动态分区任务因为存在着分区这一变量，单纯调整rdd这边的partition数目很难把控整体的文件数量。

在hive里，我们可以通过设置hive.optimize.sort.dynamic.partition来缓解动态分区产生文件过多导致任务执行时task节点经常oom的状况。这样的参数会引入新的的shuffle，来对数据进行重排序，将相同的partition分给同一个task处理，从而避免了一个task同时持有多个文件句柄。

因此，我们可以借助这样的思想，使用distribute by语句来修改sql，从而控制文件数量。一般而言，假设我们想对于每个分区生成不超过N个文件，则可以在SQL末尾增加DISTRIBUTE BY [动态分区列]，ceil(rand() * N)。

```sql
-- 修改前
INSERT OVERWRITE TABLE a partition (dayno, hour)
SELECT imei, name, dayno, hour
FROM b
WHERE imei = "xxxx";
--修改后,每个分区不超过5个文件
INSERT OVERWRITE TABLE a partition (dayno, hour)
SELECT
imei, name,dayno,hour
FROM b
WHERE imei = "xxxx" DISTRIBUTE BY (dayno, hour, ceil(rand() * 5));
```

如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

3. 自研可合并文件的commitProtocol方案

综上种种，每个方法都存在一定的弊端，众多规则也在实际使用过程中对业务方造成很大困扰。

因此我们产生了想在spark这边实现和hive类似的小文件合并机制。在几个可能的方案选型中，我们最终选择了：重写spark.sql.sources.commitProtocolClass方法。

一方面，该方案对Spark代码无侵入，便于Spark源码的维护，另一方面，该方案对业务方使用友好，可以动态通过set命令设置，如果出现问题回滚也十分方便。业务方在使用过程中，只需要简单设置：

spark.sql.sources.commitProtocolClass，即可控制是否开启小文件合并。

在开启小文件合并参数后，我们会在commit阶段拿到生成的所有文件，引入两个新的job来对这些文件进行处理。首先我们在第一个job获取到所有大小小于spark.compact.smallfile.size的文件，在查找完成后按照spark.compact.size参数值对组合文件，并在第二个job中对这些文件进行合并。

```scala
val result =
  new mutable.HashMap[String, mutable.Set[(String, Array[(String, Long)])]] with mutable.MultiMap[String, (String, Array[(String, Long)

//parallelism in executor
val committedTaskDirRdd = sparkSession.sparkContext.parallelize(committedTaskPaths.map(_.getPath.toString), parallelism)
val serializableHadoopConf = new SerializableConfiguration(fileSystem.getConf)

val ret = new Array[mutable.HashMap[String, mutable.Set[(String, Array[(String, Long)])]]](committedTaskDirRdd.partitions.length)
sparkSession.sparkContext.runJob(committedTaskDirRdd,
  (taskContext: TaskContext, iter: Iterator[String]) => {
    getAllGroupedCommittedLeafDirectoryPathOfTask(serializableHadoopConf.value, iter)
  },
  committedTaskDirRdd.partitions.indices,
  (index, res: mutable.HashMap[String, mutable.Set[(String, Array[(String, Long)])]]) => {
    ret(index) = res
  })
```

组合文件的job

如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

```
val taskIds: Array[String] = sparkSession.sparkContext.runJob(compactTaskSlipRdd,
  (taskContext: TaskContext, iter: Iterator[(Array[String], String)]) => {
    val jobTrackerID = new SimpleDateFormat( pattern = "yyyyMMddHHmmss", Locale.US).format(new Date)
    val jobId = new JobID(jobTrackerID, taskContext.stageId())                     对文件进行合并
    val taskId = new TaskID(jobId, TaskType.MAP, taskContext.partitionId())

    iter.foreach((compactTask: (Array[String], String)) => {
      val compactToTempFileName = compactTask._2 + "." + taskContext.attemptNumber() + ".inprogress"
      val compactToTempFile = new Path(compactWorkingRoot, taskId.toString) + compactToTempFileName
      val compactToFile = new Path(compactWorkingRoot, taskId.toString) + compactTask._2
      Compactor.getCompactor(fileFormat.get).compact(serializableHadoopConf.value, compactTask._1, compactToTempFile, compactToFile)
    })

    taskId.toString
  })
```

如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

本文原文 https://mp.weixin.qq.com/s/P_2DUKEoUvCNRaDk_OAfFQ

本博客文章除特别声明，全部都是原创！
原创文章版权归过往记忆大数据（过往记忆）所有，未经许可不得转载。
本文链接: 【】（）