

Spark性能优化指南——高级篇

本文转自：http://tech.meituan.com/spark-tuning-pro.html

感谢原作者

前言

继基础篇讲解了每个Spark开发人员都必须熟知的开发调优与资源调优之后，本文作为《Spark性能优化指南》的高级篇，将深入分析数据倾斜调优与shuffle调优，以解决更加棘手的性能问题。

数据倾斜调优

调优概述

有的时候，我们可能会遇到大数据计算中一个最棘手的问题——数据倾斜，此时Spark作业的性能会比期望差很多。数据倾斜调优，就是使用各种技术方案解决不同类型的数据倾斜问题，以保证Spark作业的性能。

数据倾斜发生时的现象

- 绝大多数task执行得都非常快，但个别task执行极慢。比如，总共有1000个task，997个task都在1分钟之内执行完了，但是剩余两三个task却要一两个小时。这种情况很常见。
- 原本能够正常执行的Spark作业，某天突然报出OOM（内存溢出）异常，观察异常栈，是我们写的业务代码造成的。这种情况比较少见。

数据倾斜发生的原理

数据倾斜的原理很简单：在进行shuffle的时候，必须将各个节点上相同的key拉取到某个节点上的一个task来进行处理，比如按照key进行聚合或join等操作。此时如果某个key对应的数据量特别大的话，就会发生数据倾斜。比如大部分key对应10条数据，但是个别key却对应了100万条数据，那么大部分task可能就只会分配到10条数据，然后1秒钟就运行完了；但是个别task可能分配到了100万数据，要运行一两个小时。因此，整个Spark作业的运行进度是由运行时间最长的那个task决定的。

因此出现数据倾斜的时候，Spark作业看起来会运行得非常缓慢，甚至可能因为某个task处理的数据量过大导致内存溢出。

下图就是一个很清晰的例子：hello这个key，在三个节点上对应了总共7条数据，这些数据都会被拉取到同一个task中进行处理；而world和you这两个key分别才对应1条数据，所以另外两个task只要分别处理1条数据即可。此时第一个task的运行时间可能是另外两个task的7倍，而整个stage的运行速度也由运行最慢的那个task所决定。

公告

昵称：XGogo
 园龄：1年9个月
 粉丝：25
 关注：2
 +加关注

2017年12月						
日	一	二	三	四	五	六
26	27	28	29	30	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

搜索

找找看

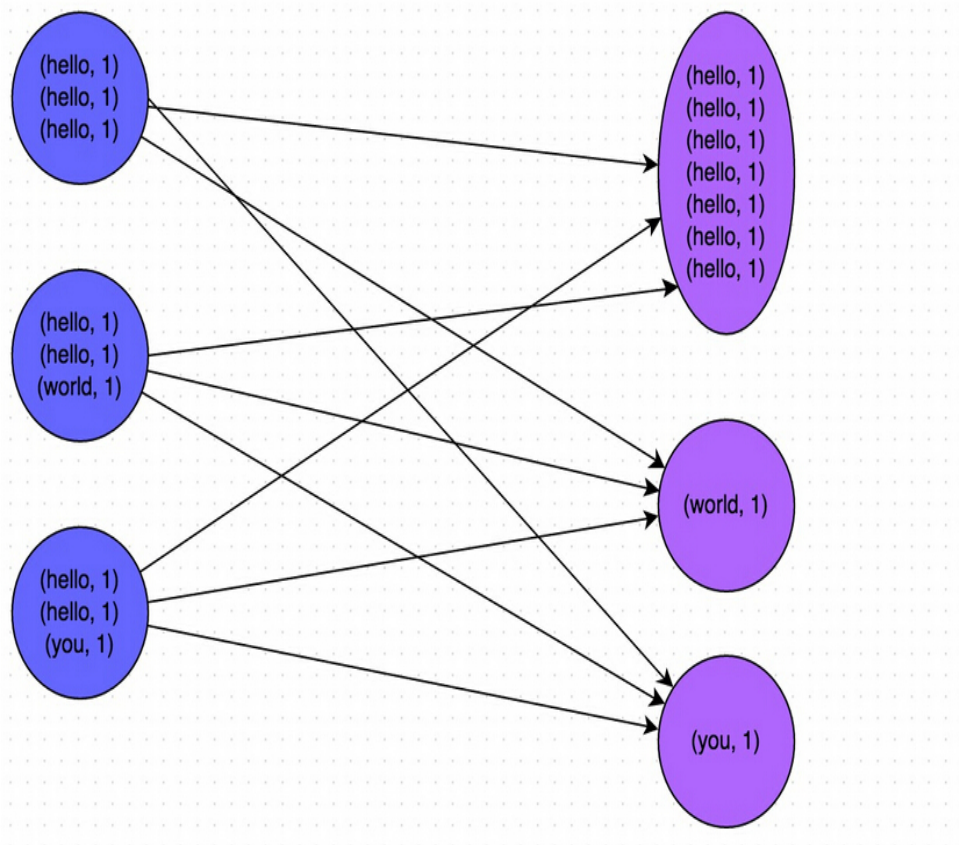
谷歌搜索

常用链接

[我的随笔](#)
[我的评论](#)
[我的参与](#)
[最新评论](#)
[我的标签](#)

随笔分类(117)

Alluxio(3)
 Apache Kudu(2)
 C++(1)
 Flume(3)
 HBASE(18)
 HDFS(7)
 Java(20)
 Kafka(8)
 Linux
 OpenTSDB(1)
 Python
 Solr(9)
 spark(12)
 tensorflow(4)
 Zookeeper(11)
 编程概述(10)
 机器学习(1)
 文件格式(2)
 修行(5)



如何定位导致数据倾斜的代码

数据倾斜只会发生在shuffle过程中。这里给大家罗列一些常用的并且可能会触发shuffle操作的算子：distinct、groupByKey、reduceByKey、aggregateByKey、join、cogroup、repartition等。出现数据倾斜时，可能就是你的代码中使用了这些算子中的某一个所导致的。

某个task执行特别慢的情况

首先要看的，就是数据倾斜发生在第几个stage中。

如果是用yarn-client模式提交，那么本地是直接可以看到log的，可以在log中找到当前运行到了第几个stage；如果是用yarn-cluster模式提交，则可以通过Spark Web UI来查看当前运行到了第几个stage。此外，无论是使用yarn-client模式还是yarn-cluster模式，我们都可以在Spark Web UI上深入看一下当前这个stage各个task分配的数据量，从而进一步确定是不是task分配的数据不均匀导致了数据倾斜。

比如下图中，倒数第三列显示了每个task的运行时间。明显可以看到，有的task运行特别快，只需要几秒钟就可以运行完；而有的task运行特别慢，需要几分钟才能运行完，此时单从运行时间上看就已经能够确定发生数据倾斜了。此外，倒数第一列显示了每个task处理的数据量，明显可以看到，运行时间特别短的task只需要处理几百KB的数据即可，而运行时间特别长的task需要处理几千KB的数据，处理的数据量差了10倍。此时更加能够确定是发生了数据倾斜。

85	154	0	SUCCESS	PROCESS_LOCAL	3 / rz-data-hdp-dn0912.rz.sankuai.com	2016/01/29 13:42:02	3.2 min	0.9 s	807.7 KB / 8691
86	155	0	SUCCESS	PROCESS_LOCAL	46 / rz-data-hdp-dn0890.rz.sankuai.com	2016/01/29 13:42:02	49 s	0.5 s	531.4 KB / 5309
87	156	0	SUCCESS	PROCESS_LOCAL	92 / rz-data-hdp-dn1275.rz.sankuai.com	2016/01/29 13:42:02	31 s	0.6 s	360.7 KB / 3696
88	157	0	SUCCESS	PROCESS_LOCAL	64 / rz-data-hdp-dn0121.rz.sankuai.com	2016/01/29 13:42:02	27 s	0.4 s	406.1 KB / 4104
89	158	0	SUCCESS	PROCESS_LOCAL	13 / rz-data-hdp-dn1184.rz.sankuai.com	2016/01/29 13:42:02	14 s	0.4 s	347.3 KB / 3561
90	159	0	SUCCESS	PROCESS_LOCAL	5 / rz-data-hdp-dn0912.rz.sankuai.com	2016/01/29 13:42:02	13 s	0.3 s	351.4 KB / 3622
91	160	0	RUNNING	PROCESS_LOCAL	90 / rz-data-hdp-dn0059.rz.sankuai.com	2016/01/29 13:42:02	3.9 min	0.8 s	1617.0 KB / 18545
92	161	0	SUCCESS	PROCESS_LOCAL	87 / rz-data-hdp-dn0879.rz.sankuai.com	2016/01/29 13:42:02	26 s	0.4 s	318.1 KB / 3081
93	162	0	SUCCESS	PROCESS_LOCAL	55 / rz-data-hdp-dn0875.rz.sankuai.com	2016/01/29 13:42:02	19 s	0.5 s	359.6 KB / 3574
94	163	0	RUNNING	PROCESS_LOCAL	82 / rz-data-hdp-dn0430.rz.sankuai.com	2016/01/29 13:42:02	3.9 min	0.8 s	2023.4 KB / 22812
95	164	0	SUCCESS	PROCESS_LOCAL	99 / rz-data-hdp-dn0817.rz.sankuai.com	2016/01/29 13:42:02	5 s	0.2 s	188.1 KB / 1426
96	165	0	SUCCESS	PROCESS_LOCAL	56 / rz-data-hdp-dn0875.rz.sankuai.com	2016/01/29 13:42:02	10 s	0.3 s	214.5 KB / 1683
97	166	0	SUCCESS	PROCESS_LOCAL	71 / rz-data-hdp-dn0576.rz.sankuai.com	2016/01/29 13:42:02	2.9 min	0.4 s	673.8 KB / 6932
98	167	0	SUCCESS	PROCESS_LOCAL	77 / rz-data-hdp-dn0242.rz.sankuai.com	2016/01/29 13:42:02	13 s	0.3 s	276.3 KB / 2349
99	168	0	RUNNING	PROCESS_LOCAL	58 / rz-data-hdp-dn0491.rz.sankuai.com	2016/01/29 13:42:02	3.9 min	1 s	1321.0 KB / 14508

知道数据倾斜发生在哪一个stage之后，接着我们就需要根据stage划分原理，推算出来发生倾斜的那个stage对应代码中的哪一部分，这部分代码中肯定会有一个shuffle类算子。精准推算stage与代码的对应关系，需要对Spark的源码有深入的理解，这里我们可以介绍一个相对简单实用的推算方法：只要看到Spark代码中出现了shuffle类算子或者是Spark SQL的SQL语句中出现了会导致shuffle的语句（比如group by语句），那么就可以判定，以那个地方为界限划分出了前后两个stage。

文章分类(56)

HBASE(13)
HDFS(7)
java(5)
lucene(4)
Parquet(4)
solr(9)
spark(10)
计算机基础知识(4)

优秀博客

JAVA高性能
kafka高手文章
阿里技术团队
虾皮工作室

最新评论

1. Re:NXLog中文文档(1)：简介
博主，可否进一步说下这个nxlog的使用和配置？

--谷仁儿

2. Re:NXLog中文文档(3):配置
亲，能稍微讲下运行逻辑和原理吗，这个nxlog整体的适用场景和常见配置，从实用的角度出发，谢谢啦！

--谷仁儿

3. Re:关于Snmp的Trap代码开发之坑
V3版本遇到过，v2版本没有遇到。

--XGogo

4. Re:关于Snmp的Trap代码开发之坑
2c版本，在高频率发送情况下，接收端会出现部分包接收无响应的情况，是监听性能问题吗？

--funying

5. Re:solr中时区处理
没看懂你的逻辑，solr.in.sh起什么作用？安装文件没有也不影响数据导入和查询 配置它根本不起作用 tomcat/bin/setenv.sh确定tomcat/bin下有setenv.sh吗.....

--前世张公子

阅读排行榜

1. IDEA15 下运行Scala遇到问题以及解决办法(17895)
2. Apache Curator入门实战(10052)
3. Spark性能优化指南——高级篇(8929)
4. 关于Solr6.0中solrj使用简单例子(8011)
5. hbase RowFilter如何根据rowkey查询以及实例实现代码 habase模糊查询【转】(5361)

评论排行榜

1. Alluxio 内存存储系统部署(3)
2. 关于Snmp的Trap代码开发之坑(2)
3. Spark的基本说明(1)
4. spark结构化数据处理：Spark SQL、DataFrame和Dataset(1)
5. solr中时区处理(1)

推荐排行榜

1. Apache Curator入门实战(5)
2. Spark性能优化指南——高级篇(2)
3. Spark性能优化指南——基础篇转(1)

这里我们就以Spark最基础的入门程序——单词计数来举例，如何用最简单的方法大致推算出一个stage对应的代码。如下示例，在整个代码中，只有一个reduceByKey是会发生shuffle的算子，因此就可以认为，以这个算子为界限，会划分出前后两个stage。

- stage0，主要是执行从textFile到map操作，以及执行shuffle write操作。shuffle write操作，我们可以简单理解为对pairs RDD中的数据进行分区操作，每个task处理的数据中，相同的key会写入同一个磁盘文件内。
- stage1，主要是执行从reduceByKey到collect操作，stage1的各个task一开始运行，就会首先执行shuffle read操作。执行shuffle read操作的task，会从stage0的各个task所在节点拉取属于自己处理的那些key，然后对同一个key进行全局性的聚合或join等操作，在这里就是对key的value值进行累加。stage1在执行完reduceByKey算子之后，就计算出了最终的wordCounts RDD，然后会执行collect算子，将所有数据拉取到Driver上，供我们遍历和打印输出。

```
1. val conf = new SparkConf()
2. val sc = new SparkContext(conf)
3.
4. val lines = sc.textFile("hdfs://...")
5. val words = lines.flatMap(_.split(" "))
6. val pairs = words.map(_._1)
7. val wordCounts = pairs.reduceByKey(_ + _)
8.
9. wordCounts.collect().foreach(println(_))
```

通过对单词计数程序的分析，希望能够让大家了解最基本的stage划分的原理，以及stage划分后shuffle操作是如何在两个stage的边界处执行的。然后我们就知道如何快速定位出发生数据倾斜的stage对应代码的哪一个部分了。比如我们在Spark Web UI或者本地log中发现，stage1的某几个task执行得特别慢，判定stage1出现了数据倾斜，那么就可以回到代码中定位出stage1主要包 括了reduceByKey这个shuffle类算子，此时基本就可以确定是由reduceByKey算子导致的数据倾斜问题。比如某个单词出现了100万次，其他单词才出现10次，那么stage1的某个task就要处理100万数据，整个stage的速度就会被这个task拖慢。

某个task莫名其妙内存溢出的情况

这种情况下去定位出问题的代码就比较容易了。我们建议直接看yarn-client模式下本地log的异常栈，或者通过YARN查看yarn-cluster模式下的log中的异常栈。一般来说，通过异常栈信息就可以定位到你的代码中哪一行发生了内存溢出。然后在那行代码附近找找，一般也会有 shuffle类算子，此时很可能就是这个算子导致了数据倾斜。

但是大家要注意的是，不能单纯靠偶然的内存溢出就判定发生了数据倾斜。因为自己编写的代码的bug，以及偶然出现的数据异常，也可能导致内存溢出。因此还是要按照上面所讲的方法，通过Spark Web UI查看报错的那个stage的各个task的运行时间以及分配的数据量，才能确定是否是由于数据倾斜才导致了这次内存溢出。

查看导致数据倾斜的key的数据分布情况

知道了数据倾斜发生在哪里之后，通常需要分析一下那个执行了shuffle操作并且导致了数据倾斜的RDD/Hive表，查看一下其中key的分布情况。这主要是为之后选择哪一种技术方案提供依据。针对不同的key分布与不同的shuffle算子组合起来的各种情况，可能需要选择不同的技术方案来解决。

此时根据你执行操作的情况不同，可以有很多种查看key分布的方式：

1. 如果是Spark SQL中的group by、join语句导致的数据倾斜，那么就查询一下SQL中使用的表的key分布情况。
2. 如果是对Spark RDD执行shuffle算子导致的数据倾斜，那么可以在Spark作业中加入查看key分布的代码，比如RDD.countByKey()。然后对统计出来的各个key出现的次数，collect/take到客户端打印一下，就可以看到key的分布情况。

举例来说，对于上面所说的单词计数程序，如果确定是stage1的reduceByKey算子导致了数据倾斜，那么就应该看看进行reduceByKey操作的RDD中的key分布情况，在这个例子中指的就是pairs RDD。如下示例，我们可以先对pairs采样10%的样本数据，然后使用countByKey算子统计出每个key出现的次数，最后在客户端遍历和打印样本数据中各个key的出现次数。

```
1. val sampledPairs = pairs.sample(false, 0.1)
2. val sampledWordCounts = sampledPairs.countByKey()
3. sampledWordCounts.foreach(println(_))
```

数据倾斜的解决方案

解决方案一：使用Hive ETL预处理数据

方案适用场景：导致数据倾斜的是Hive表。如果该Hive表中的数据本身很不均匀（比如某个key对应了100万数据，其他key才对应了10条数据），而且业务场景需要频繁使用Spark对Hive表执行某个分析操作，那么比较适合使用这种技术方案。

4. Java8初体验（二）Stream语法详解(1)
5. Zookeeper运维的一些经验[转](1)

方案实现思路：此时可以评估一下，是否可以通过Hive来进行数据预处理（即通过Hive ETL预先对数据按照key进行聚合，或者是预先和其他表进行join），然后在Spark作业中针对的数据源就不是原来的Hive表了，而是预处理后的Hive表。此时由于数据已经预先进行过聚合或join操作了，那么在Spark作业中也就不需要使用原先的shuffle类算子执行这类操作了。

方案实现原理：这种方案从根源上解决了数据倾斜，因为彻底避免了在Spark中执行shuffle类算子，那么肯定就不会有数据倾斜的问题了。但是这里也要提醒一下大家，这种方式属于治标不治本。因为毕竟数据本身就存在分布不均匀的问题，所以Hive ETL中进行group by或者join等shuffle操作时，还是会出现数据倾斜，导致Hive ETL的速度很慢。我们只是把数据倾斜的发生提前到了Hive ETL中，避免Spark程序发生数据倾斜而已。

方案优点：实现起来简单便捷，效果还非常好，完全规避掉了数据倾斜，Spark作业的性能会大幅度提升。

方案缺点：治标不治本，Hive ETL中还是会发生数据倾斜。

方案实践经验：在一些Java系统与Spark结合使用的项目中，会出现Java代码频繁调用Spark作业的场景，而且对Spark作业的执行性能要求很高，就比较适合使用这种方案。将数据倾斜提前到上游的Hive ETL，每天仅执行一次，只有那一次是比较慢的，而之后每次Java调用Spark作业时，执行速度都会很快，能够提供更好的用户体验。

项目实践经验：在美团·点评的交互式用户行为分析系统中使用了这种方案，该系统主要是允许用户通过Java Web系统提交数据分析统计任务，后端通过Java提交Spark作业进行数据分析统计。要求Spark作业速度必须要快，尽量在10分钟以内，否则速度太慢，用户体验会很差。所以我们将有些Spark作业的shuffle操作提前到了Hive ETL中，从而让Spark直接使用预处理后的Hive中间表，尽可能地减少Spark的shuffle操作，大幅度提升了性能，将部分作业的性能提升了6倍以上。

解决方案二：过滤少数导致倾斜的key

方案适用场景：如果发现导致倾斜的key就少数几个，而且对计算本身的影响并不大的话，那么很适合使用这种方案。比如99%的key就对应10条数据，但是只有一个key对应了100万数据，从而导致了数据倾斜。

方案实现思路：如果我们判断那少数几个数据量特别多的key，对作业的执行和计算结果不是特别重要的话，那么干脆就直接过滤掉那少数几个key。比如，在Spark SQL中可以使用where子句过滤掉这些key或者在Spark Core中对RDD执行filter算子过滤掉这些key。如果需要每次作业执行时，动态判定哪些key的数据量最多然后再进行过滤，那么可以使用sample算子对RDD进行采样，然后计算出每个key的数量，取数据量最多的key过滤掉即可。

方案实现原理：将导致数据倾斜的key给过滤掉之后，这些key就不会参与计算了，自然不可能产生数据倾斜。

方案优点：实现简单，而且效果也很好，可以完全规避掉数据倾斜。

方案缺点：适用场景不多，大多数情况下，导致倾斜的key还是很多的，并不是只有少数几个。

方案实践经验：在项目中我们也采用过这种方案解决数据倾斜。有一次发现某一天Spark作业在运行的时候突然OOM了，追查之后发现，是Hive表中的某一个key在那天数据异常，导致数据量暴增。因此就采取每次执行前先进行采样，计算出样本中数据量最大的几个key之后，直接在程序中将那些key给过滤掉。

解决方案三：提高shuffle操作的并行度

方案适用场景：如果我们必须要对数据倾斜迎难而上，那么建议优先使用这种方案，因为这是处理数据倾斜最简单的一种方案。

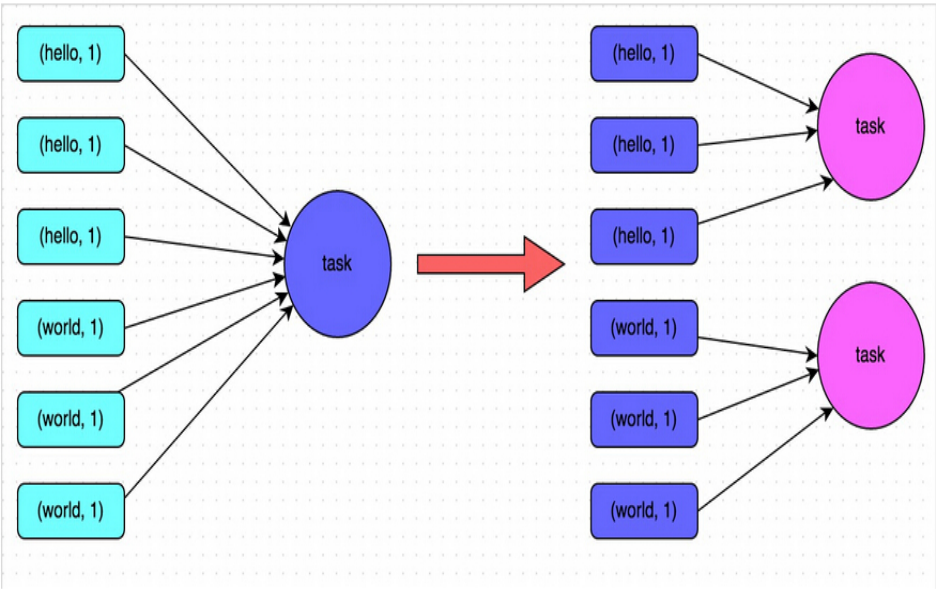
方案实现思路：在对RDD执行shuffle算子时，给shuffle算子传入一个参数，比如reduceByKey(1000)，该参数就设置了这个shuffle算子执行时shuffle read task的数量。对于Spark SQL中的shuffle类语句，比如group by、join等，需要设置一个参数，即spark.sql.shuffle.partitions，该参数代表了shuffle read task的并行度，该值默认是200，对于很多场景来说都有点过小。

方案实现原理：增加shuffle read task的数量，可以让原本分配给一个task的多个key分配给多个task，从而让每个task处理比原来更少的数据。举例来说，如果原本有5个key，每个key对应10条数据，这5个key都是分配给一个task的，那么这个task就要处理50条数据。而增加了shuffle read task以后，每个task就分配到一个key，即每个task就处理10条数据，那么自然每个task的执行时间都会变短了。具体原理如下图所示。

方案优点：实现起来比较简单，可以有效缓解和减轻数据倾斜的影响。

方案缺点：只是缓解了数据倾斜而已，没有彻底根除问题，根据实践经验来看，其效果有限。

方案实践经验：该方案通常无法彻底解决数据倾斜，因为如果出现一些极端情况，比如某个key对应的数据量有100万，那么无论你的task数量增加到多少，这个对应着100万数据的key肯定还是会分配到一个task中去处理，因此注定还是会发生数据倾斜的。所以这种方案只能说是在发现数据倾斜时尝试使用的第一种手段，尝试去用嘴简单的方法缓解数据倾斜而已，或者是和其他方案结合起来使用。



解决方案四：两阶段聚合（局部聚合+全局聚合）

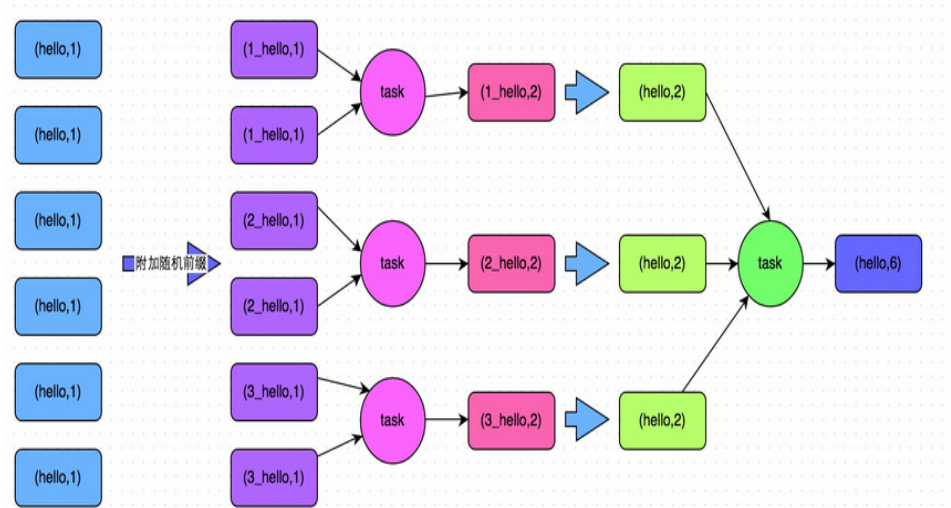
方案适用场景：对RDD执行reduceByKey等聚合类shuffle算子或者在Spark SQL中使用group by语句进行分组聚合时，比较适用这种方案。

方案实现思路：这个方案的核心实现思路就是进行两阶段聚合。第一次是局部聚合，先给每个key都打上一个随机数，比如10以内的随机数，此时原先一样的key就变成不一样的了，比如(hello, 1) (hello, 1) (hello, 1) (hello, 1)，就会变成(1_hello, 1) (1_hello, 1) (2_hello, 1) (2_hello, 1)。接着对打上随机数后的数据，执行reduceByKey等聚合操作，进行局部聚合，那么局部聚合结果，就会变成了(1_hello, 2) (2_hello, 2)。然后将各个key的前缀给去掉，就会变成(hello,2) (hello,2)，再次进行全局聚合操作，就可以得到最终结果了，比如(hello, 4)。

方案实现原理：将原本相同的key通过附加随机前缀的方式，变成多个不同的key，就可以让原本被一个task处理的数据分散到多个task上去做局部聚合，进而解决单个task处理数据量过多的问题。接着去除掉随机前缀，再次进行全局聚合，就可以得到最终的结果。具体原理见下图。

方案优点：对于聚合类的shuffle操作导致的数据倾斜，效果是非常不错的。通常都可以解决掉数据倾斜，或者至少是大幅度缓解数据倾斜，将Spark作业的性能提升数倍以上。

方案缺点：仅仅适用于聚合类的shuffle操作，适用范围相对较窄。如果是join类的shuffle操作，还得用其他的解决方案。



```
1. // 第一步，给RDD中的每个key都打上一个随机前缀。
2. JavaPairRDD<String, Long> randomPrefixRdd = rdd.mapToPair(
3.     new PairFunction<Tuple2<Long,Long>, String, Long>() {
4.         private static final long serialVersionUID = 1L;
5.         @Override
6.         public Tuple2<String, Long> call(Tuple2<Long, Long> tuple)
7.         throws Exception {
```



```
8. Random random = new Random();

9. int prefix = random.nextInt(10);

10. return new Tuple2<String, Long>(prefix + "_" + tuple._1, tuple._2);

11. }

12. });

13.

14. // 第二步，对打上随机前缀的key进行局部聚合。

15. JavaPairRDD<String, Long> localAggrRdd = randomPrefixRdd.reduceByKey(

16. new Function2<Long, Long, Long>() {

17. private static final long serialVersionUID = 1L;

18. @Override

19. public Long call(Long v1, Long v2) throws Exception {

20. return v1 + v2;

21. }

22. });

23.

24. // 第三步，去除RDD中每个key的随机前缀。

25. JavaPairRDD<Long, Long> removedRandomPrefixRdd = localAggrRdd.mapToPair(

26. new PairFunction<Tuple2<String,Long>, Long, Long>() {

27. private static final long serialVersionUID = 1L;

28. @Override

29. public Tuple2<Long, Long> call(Tuple2<String, Long> tuple)

30. throws Exception {

31. long originalKey = Long.valueOf(tuple._1.split("_")[1]);

32. return new Tuple2<Long, Long>(originalKey, tuple._2);

33. }

34. });

35.

36. // 第四步，对去除了随机前缀的RDD进行全局聚合。

37. JavaPairRDD<Long, Long> globalAggrRdd = removedRandomPrefixRdd.reduceByKey(

38. new Function2<Long, Long, Long>() {

39. private static final long serialVersionUID = 1L;

40. @Override

41. public Long call(Long v1, Long v2) throws Exception {

42. return v1 + v2;

43. }

44. });
```

解决方案五：将reduce join转为map join

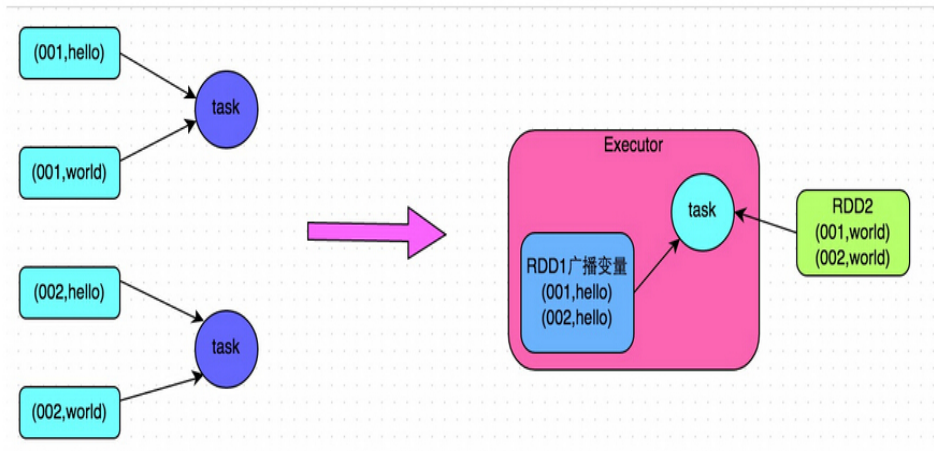
方案适用场景：在对RDD使用join类操作，或者是在Spark SQL中使用join语句时，而且join操作中的一个RDD或表的数据量比较小（比如几百M或者一两G），比较适用此方案。

方案实现思路：不使用join算子进行连接操作，而使用Broadcast变量与map类算子实现join操作，进而完全规避掉shuffle类的操作，彻底避免数据倾斜的发生和出现。将较小RDD中的数据直接通过collect算子拉取到Driver端的内存中来，然后对其创建一个Broadcast变量；接着对另外一个RDD执行map类算子，在算子函数内，从Broadcast变量中获取较小RDD的全量数据，与当前RDD的每一条数据按照连接key进行比对，如果连接key相同的话，那么就将两个RDD的数据用你需要的方式连接起来。

方案实现原理：普通的join是会走shuffle过程的，而一旦shuffle，就相当于会将相同key的数据拉取到一个shuffle read task中再进行join，此时就是reduce join。但是如果一个RDD是比较小的，则可以采用广播小RDD全量数据+map算子来实现与join同样的效果，也就是map join，此时就不会发生shuffle操作，也就不会发生数据倾斜。具体原理如下图所示。

方案优点：对join操作导致的数据倾斜，效果非常好，因为根本就不会发生shuffle，也就根本不会发生数据倾斜。

方案缺点：适用场景较少，因为这个方案只适用于一个大表和小表的情况。毕竟我们需要将小表进行广播，此时会比较消耗内存资源，driver和每个Executor内存中都会驻留一份小RDD的全量数据。如果我们广播出去的RDD数据比较大，比如10G以上，那么就可能发生内存溢出了。因此并不适合两个都是大表的情况。



```

1. // 首先将数据量比较小的RDD的数据，collect到Driver中来。
2. List<Tuple2<Long, Row>> rdd1Data = rdd1.collect()
3. // 然后使用Spark的广播功能，将小RDD的数据转换成广播变量，这样每个Executor就只有一份RDD的数据。
4. // 可以尽可能节省内存空间，并且减少网络传输性能开销。
5. final Broadcast<List<Tuple2<Long, Row>>> rdd1DataBroadcast = sc.broadcast(rdd1Data);
6.
7. // 对另外一个RDD执行map类操作，而不再是join类操作。
8. JavaPairRDD<String, Tuple2<String, Row>> joinedRdd = rdd2.mapToPair(
9.     new PairFunction<Tuple2<Long, String>, String, Tuple2<String, Row>>() {
10.         private static final long serialVersionUID = 1L;
11.         @Override
12.         public Tuple2<String, Tuple2<String, Row>> call(Tuple2<Long, String> tuple)
13.             throws Exception {
14.             // 在算子函数中，通过广播变量，获取到本地Executor中的rdd1数据。
15.             List<Tuple2<Long, Row>> rdd1Data = rdd1DataBroadcast.value();
16.             // 可以将rdd1的数据转换为一个Map，便于后面进行join操作。
17.             Map<Long, Row> rdd1DataMap = new HashMap<Long, Row>();
18.             for(Tuple2<Long, Row> data : rdd1Data) {
19.                 rdd1DataMap.put(data._1, data._2);
20.             }
21.             // 获取当前RDD数据的key以及value。
22.             String key = tuple._1;
23.             String value = tuple._2;
24.             // 从rdd1数据Map中，根据key获取到可以join到的数据。
25.             Row rdd1Value = rdd1DataMap.get(key);
26.             return new Tuple2<String, String>(key, new Tuple2<String, Row>(value, rdd1Value));
27.         }

```

28. });

29.

30. // 这里得提示一下。

31. // 上面的做法，仅仅适用于rdd1中的key没有重复，全部是唯一的场景。

32. // 如果rdd1中有多个相同的key，那么就需用flatMap类的操作，在进行join的时候不能用map，而是得遍历rdd1所有数据进行join。

33. // rdd2中每条数据都可能会返回多条join后的数据。

解决方案六：采样倾斜key并分拆join操作

方案适用场景：两个RDD/Hive表进行join的时候，如果数据量都比较大，无法采用“解决方案五”，那么此时可以看一下两个RDD/Hive表中的key分布情况。如果出现数据倾斜，是因为其中某一个RDD/Hive表中的少数几个key的数据量过大，而另一个RDD/Hive表中的所有key都分布比较均匀，那么采用这个解决方案是比较合适的。

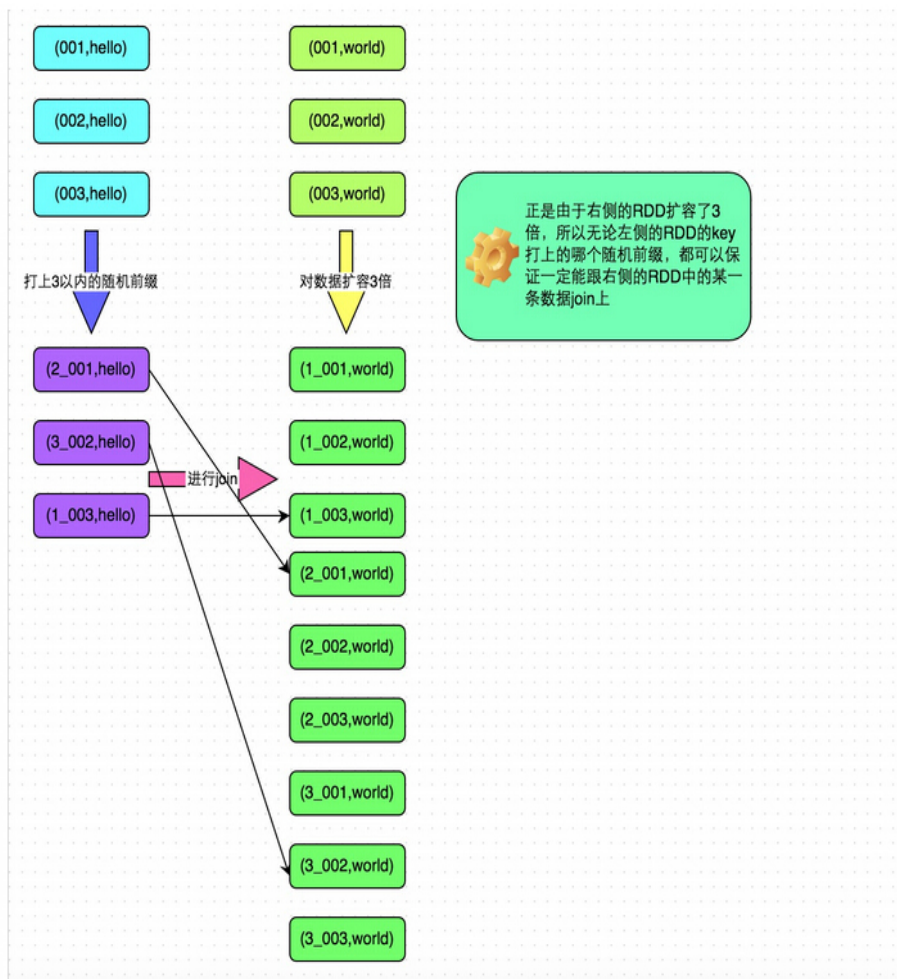
方案实现思路：

- 对包含少数几个数据量过大的key的那个RDD，通过sample算子采样出一份样本来，然后统计一下每个key的数量，计算出来数据量最大的是哪几个key。
- 然后将这几个key对应的数据从原来的RDD中拆分出来，形成一个单独的RDD，并给每个key都打上n以内的随机数作为前缀，而不会导致倾斜的大部分key形成另外一个RDD。
- 接着将需要join的另一个RDD，也过滤出来那几个倾斜key对应的数据并形成一个新的RDD，将每条数据膨胀成n条数据，这n条数据都按顺序附加一个0~n的前缀，不会导致倾斜的大部分key也形成另外一个RDD。
- 再将附加了随机前缀的独立RDD与另一个膨胀n倍的独立RDD进行join，此时就可以将原先相同的key打散成n份，分散到多个task中去进行join了。
- 而另外两个普通的RDD就照常join即可。
- 最后将两次join的结果使用union算子合并起来即可，就是最终的join结果。

方案实现原理：对于join导致的数据倾斜，如果只是某几个key导致了倾斜，可以将少数几个key分拆成独立RDD，并附加随机前缀打散成n份去进行join，此时这几个key对应的数据就不会集中在少数几个task上，而是分散到多个task进行join了。具体原理见下图。

方案优点：对于join导致的数据倾斜，如果只是某几个key导致了倾斜，采用该方式可以用最有效的方式打散key进行join。而且只需要针对少数倾斜key对应的数据进行扩容n倍，不需要对全量数据进行扩容。避免了占用过多内存。

方案缺点：如果导致倾斜的key特别多的话，比如成千上万个key都导致数据倾斜，那么这种方式也不适合。



```

1. // 首先从包含了少数几个导致数据倾斜key的rdd1中，采样10%的样本数据。
2. JavaPairRDD<Long, String> sampledRDD = rdd1.sample(false, 0.1);
3.
4. // 对样本数据RDD统计出每个key的出现次数，并按出现次数降序排序。
5. // 对降序排序后的数据，取出top 1或者top 100的数据，也就是key最多的前n个数据。
6. // 具体取出多少个数据量最多的key，由大家自己决定，我们这里就取1个作为示范。
7. JavaPairRDD<Long, Long> mappedSampledRDD = sampledRDD.mapToPair(
8. new PairFunction<Tuple2<Long,String>, Long, Long>() {
9. private static final long serialVersionUID = 1L;
10. @Override
11. public Tuple2<Long, Long> call(Tuple2<Long, String> tuple)
12. throws Exception {
13. return new Tuple2<Long, Long>(tuple._1, 1L);
14. }
15. });
16. JavaPairRDD<Long, Long> countedSampledRDD = mappedSampledRDD.reduceByKey(
17. new Function2<Long, Long, Long>() {
18. private static final long serialVersionUID = 1L;
19. @Override
20. public Long call(Long v1, Long v2) throws Exception {
21. return v1 + v2;
22. }

```

```
23. });

24. JavaPairRDD<Long, Long> reversedSampledRDD = countedSampledRDD.mapToPair(

25. new PairFunction<Tuple2<Long,Long>, Long, Long>() {

26. private static final long serialVersionUID = 1L;

27. @Override

28. public Tuple2<Long, Long> call(Tuple2<Long, Long> tuple)

29. throws Exception {

30. return new Tuple2<Long, Long>(tuple._2, tuple._1);

31. }

32. });

33. final Long skewedUserid = reversedSampledRDD.sortByKey(false).take(1).get(0)._2;

34.

35. // 从rdd1中分拆出导致数据倾斜的key，形成独立的RDD。

36. JavaPairRDD<Long, String> skewedRDD = rdd1.filter(

37. new Function<Tuple2<Long,String>, Boolean>() {

38. private static final long serialVersionUID = 1L;

39. @Override

40. public Boolean call(Tuple2<Long, String> tuple) throws Exception {

41. return tuple._1.equals(skewedUserid);

42. }

43. });

44. // 从rdd1中分拆出不导致数据倾斜的普通key，形成独立的RDD。

45. JavaPairRDD<Long, String> commonRDD = rdd1.filter(

46. new Function<Tuple2<Long,String>, Boolean>() {

47. private static final long serialVersionUID = 1L;

48. @Override

49. public Boolean call(Tuple2<Long, String> tuple) throws Exception {

50. return !tuple._1.equals(skewedUserid);

51. }

52. });

53.

54. // rdd2，就是那个所有key的分布相对较为均匀的rdd。

55. // 这里将rdd2中，前面获取到的key对应的数据，过滤出来，分拆成单独的rdd，并对rdd中的数据使用flatMap算子都
    扩容100倍。

56. // 对扩容的每条数据，都打上0~100的前缀。

57. JavaPairRDD<String, Row> skewedRdd2 = rdd2.filter(

58. new Function<Tuple2<Long,Row>, Boolean>() {

59. private static final long serialVersionUID = 1L;

60. @Override

61. public Boolean call(Tuple2<Long, Row> tuple) throws Exception {

62. return tuple._1.equals(skewedUserid);

63. }

64. }).flatMapToPair(new PairFlatMapFunction<Tuple2<Long,Row>, String, Row>() {

65. private static final long serialVersionUID = 1L;
```

```
66. @Override
67. public Iterable<Tuple2<String, Row>> call(
68. Tuple2<Long, Row> tuple) throws Exception {
69. Random random = new Random();
70. List<Tuple2<String, Row>> list = new ArrayList<Tuple2<String, Row>>();
71. for(int i = 0; i < 100; i++) {
72. list.add(new Tuple2<String, Row>(i + "_" + tuple._1, tuple._2));
73. }
74. return list;
75. }
76.
77. });
78.
79. // 将rdd1中分拆出来的导致倾斜的key的独立rdd，每条数据都打上100以内的随机前缀。
80. // 然后将这个rdd1中分拆出来的独立rdd，与上面rdd2中分拆出来的独立rdd，进行join。
81. JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD1 = skewedRDD.mapToPair(
82. new PairFunction<Tuple2<Long,String>, String, String>() {
83. private static final long serialVersionUID = 1L;
84. @Override
85. public Tuple2<String, String> call(Tuple2<Long, String> tuple)
86. throws Exception {
87. Random random = new Random();
88. int prefix = random.nextInt(100);
89. return new Tuple2<String, String>(prefix + "_" + tuple._1, tuple._2);
90. }
91. })
92. .join(skewedUserid2infoRDD)
93. .mapToPair(new PairFunction<Tuple2<String,Tuple2<String,Row>>, Long, Tuple2<String, Row>>() {
94. private static final long serialVersionUID = 1L;
95. @Override
96. public Tuple2<Long, Tuple2<String, Row>> call(
97. Tuple2<String, Tuple2<String, Row>> tuple)
98. throws Exception {
99. long key = Long.valueOf(tuple._1.split("_")[1]);
100. return new Tuple2<Long, Tuple2<String, Row>>(key, tuple._2);
101. }
102. });
103.
104. // 将rdd1中分拆出来的包含普通key的独立rdd，直接与rdd2进行join。
105. JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD2 = commonRDD.join(rdd2);
106.
107. // 将倾斜key join后的结果与普通key join后的结果，union起来。
108. // 就是最终的join结果。
```

```
109. JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD = joinedRDD1.union(joinedRDD2);
```

解决方案七：使用随机前缀和扩容RDD进行join

方案适用场景：如果在进行join操作时，RDD中有大量的key导致数据倾斜，那么进行分拆key也没什么意义，此时就只能使用最后一种方案来解决问题了。

方案实现思路：

- 该方案的实现思路基本和“解决方案六”类似，首先查看RDD/Hive表中的数据分布情况，找到那个造成数据倾斜的RDD/Hive表，比如有多个key都对应了超过1万条数据。
- 然后将该RDD的每条数据都打上一个n以内的随机前缀。
- 同时对另外一个正常的RDD进行扩容，将每条数据都扩容成n条数据，扩容出来的每条数据都依次打上一个0~n的前缀。
- 最后将两个处理后的RDD进行join即可。

方案实现原理：将原先一样的key通过附加随机前缀变成不一样的key，然后就可以将这些处理后的“不同 key”分散到多个task中去处理，而不是让一个task处理大量的相同key。该方案与“解决方案六”的不同之处就在于，上一种方案是尽量只对少数倾斜key对应的数据进行特殊处理，由于处理过程需要扩容RDD，因此上一种方案扩容RDD后对内存的占用并不大；而这一种方案是针对有大量倾斜key的情况，没法将部分key拆分出来进行单独处理，因此只能对整个RDD进行数据扩容，对内存资源要求很高。

方案优点：对join类型的数据倾斜基本都可以处理，而且效果也相对比较显著，性能提升效果非常不错。

方案缺点：该方案更多的是缓解数据倾斜，而不是彻底避免数据倾斜。而且需要对整个RDD进行扩容，对内存资源要求很高。

方案实践经验：曾经开发一个数据需求的时候，发现一个join导致了数据倾斜。优化之前，作业的执行时间大约是60分钟左右；使用该方案优化之后，执行时间缩短到10分钟左右，性能提升了6倍。

```
1. // 首先将其中一个key分布相对较为均匀的RDD膨胀100倍。

2. JavaPairRDD<String, Row> expandedRDD = rdd1.flatMapToPair(

3. new PairFlatMapFunction<Tuple2<Long,Row>, String, Row>() {

4. private static final long serialVersionUID = 1L;

5. @Override

6. public Iterable<Tuple2<String, Row>> call(Tuple2<Long, Row> tuple)

7. throws Exception {

8. List<Tuple2<String, Row>> list = new ArrayList<Tuple2<String, Row>>();

9. for(int i = 0; i < 100; i++) {

10. list.add(new Tuple2<String, Row>(0 + "_" + tuple._1, tuple._2));

11. }

12. return list;

13. }

14. });

15.

16. // 其次，将另一个有数据倾斜key的RDD，每条数据都打上100以内的随机前缀。

17. JavaPairRDD<String, String> mappedRDD = rdd2.mapToPair(

18. new PairFunction<Tuple2<Long,String>, String, String>() {

19. private static final long serialVersionUID = 1L;

20. @Override

21. public Tuple2<String, String> call(Tuple2<Long, String> tuple)

22. throws Exception {

23. Random random = new Random();

24. int prefix = random.nextInt(100);

25. return new Tuple2<String, String>(prefix + "_" + tuple._1, tuple._2);

26. }

27. });
```

28.

29. // 将两个处理后的RDD进行join即可。

30. `JavaPairRDD<String, Tuple2<String, Row>> joinedRDD = mappedRDD.join(expandedRDD);`

解决方案八：多种方案组合使用

在实践中发现，很多情况下，如果只是处理较为简单的数据倾斜场景，那么使用上述方案中的某一种基本就可以解决。但是如果处理一个较为复杂的数据倾斜场景，那么可能需要将多种方案组合起来使用。比如说，我们针对出现了多个数据倾斜环节的Spark作业，可以先运用解决方案一和二，预处理一部分数据，并过滤一部分数据来缓解；其次可以对某些shuffle操作提升并行度，优化其性能；最后还可以针对不同的聚合或join操作，选择一种方案来优化其性能。大家需要对这些方案的思路和原理都透彻理解之后，在实践中根据不同的情况，灵活运用多种方案，来解决自己的数据倾斜问题。

shuffle调优

调优概述

大多数Spark作业的性能主要就是消耗在了shuffle环节，因为该环节包含了大量的磁盘IO、序列化、网络数据传输等操作。因此，如果要让作业的性能更上一层楼，就有必要对shuffle过程进行调优。但是也必须提醒大家的是，影响一个Spark作业性能的因素，主要还是代码开发、资源参数以及数据倾斜，shuffle调优只能在整个Spark的性能调优中占到一小部分而已。因此大家务必把握住调优的基本原则，千万不要舍本逐末。下面我们就给大家详细讲解shuffle的原理，以及相关参数的说明，同时给出各个参数的调优建议。

ShuffleManager发展概述

在Spark的源码中，负责shuffle过程的执行、计算和处理的组件主要就是ShuffleManager，也即shuffle管理器。而随着Spark的版本的发展，ShuffleManager也在不断迭代，变得越来越先进。

在Spark 1.2以前，默认的shuffle计算引擎是HashShuffleManager。该ShuffleManager而HashShuffleManager有着一个非常严重的弊端，就是会产生大量的中间磁盘文件，进而由大量的磁盘IO操作影响了性能。

因此在Spark 1.2以后的版本中，默认的ShuffleManager改成了SortShuffleManager。SortShuffleManager相较于HashShuffleManager来说，有了一定的改进。主要就在于，每个Task在进行shuffle操作时，虽然也会产生较多的临时磁盘文件，但最后会将所有的临时文件合并（merge）成一个磁盘文件，因此每个Task就只有一个磁盘文件。在下一个stage的shuffle read task拉取自己的数据时，只要根据索引读取每个磁盘文件中的部分数据即可。

下面我们详细分析一下HashShuffleManager和SortShuffleManager的原理。

HashShuffleManager运行原理

未经优化的HashShuffleManager

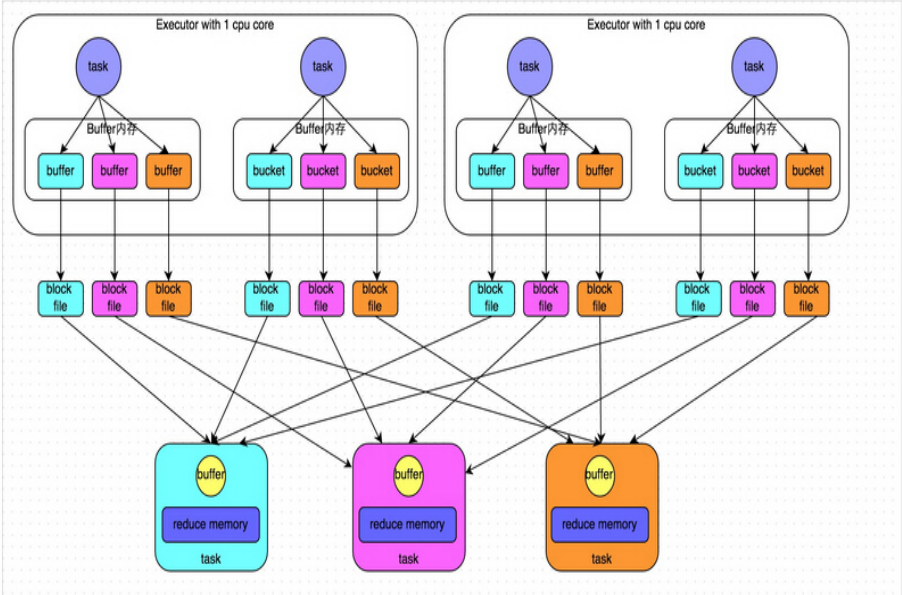
下图说明了未经优化的HashShuffleManager的原理。这里我们先明确一个假设前提：每个Executor只有1个CPU core，也就是说，无论这个Executor上分配多少个task线程，同一时间都只能执行一个task线程。

我们先从shuffle write开始说起。shuffle write阶段，主要就是在在一个stage结束计算之后，为了下一个stage可以执行shuffle类的算子（比如reduceByKey），而将每个task处理的数据按key进行“分类”。所谓“分类”，就是对相同的key执行hash算法，从而将相同key都写入同一个磁盘文件中，而每一个磁盘文件都只属于下游stage的一个task。在将数据写入磁盘之前，会先将数据写入内存缓冲中，当内存缓冲填满之后，才会溢写到磁盘文件中去。

那么每个执行shuffle write的task，要为下一个stage创建多少个磁盘文件呢？很简单，下一个stage的task有多少个，当前stage的每个task就要创建多少份磁盘文件。比如下一个stage总共有100个task，那么当前stage的每个task都要创建100份磁盘文件。如果当前stage有50个task，总共有10个Executor，每个Executor执行5个Task，那么每个Executor上总共就要创建500个磁盘文件，所有Executor上会创建5000个磁盘文件。由此可见，未经优化的shuffle write操作所产生的磁盘文件的数量是极其惊人的。

接着我们来说说shuffle read。shuffle read，通常就是一个stage刚开始时要做的事情。此时该stage的每一个task就需要将上一个stage的计算结果中的所有相同key，从各个节点上通过网络都拉取到自己所在的节点上，然后进行key的聚合或连接等操作。由于shuffle write的过程中，task给下游stage的每个task都创建了一个磁盘文件，因此shuffle read的过程中，每个task只要从上游stage的所有task所在节点上，拉取属于自己的那一个磁盘文件即可。

shuffle read的拉取过程是一边拉取一边进行聚合的。每个shuffle read task都会有一个自己的buffer缓冲，每次都只能拉取与buffer缓冲相同大小的数据，然后通过内存中的一个Map进行聚合等操作。聚合完一批数据后，再拉取下一批数据，并放到buffer缓冲中进行聚合操作。以此类推，直到最后将所有数据都拉取完，并得到最终的结果。



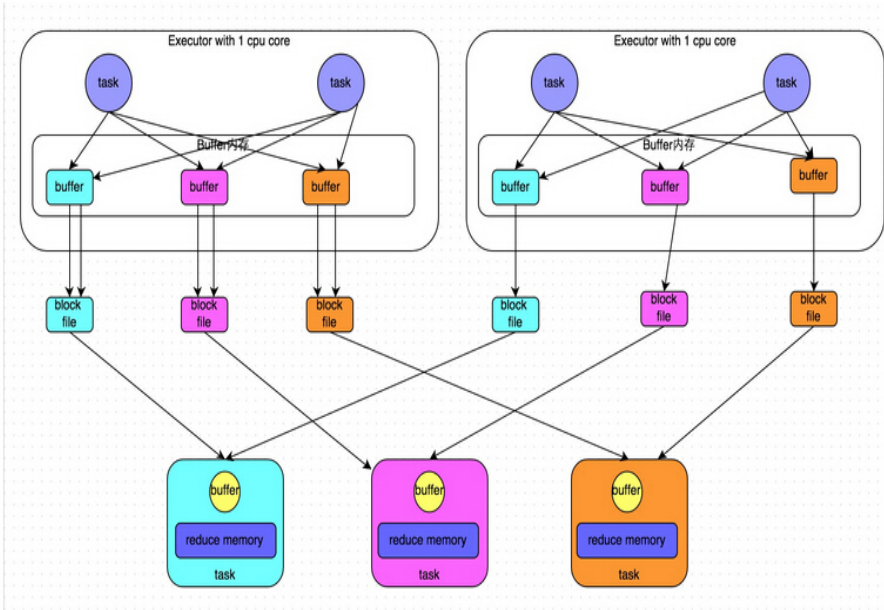
优化后的HashShuffleManager

下图说明了优化后的HashShuffleManager的原理。这里说的优化，是指我们可以设置一个参数，spark.shuffle.consolidateFiles。该参数默认值为false，将其设置为true即可开启优化机制。通常来说，如果我们使用HashShuffleManager，那么都建议开启这个选项。

开启consolidate机制之后，在shuffle write过程中，task就不是为下游stage的每个task创建一个磁盘文件了。此时会出现shuffleFileGroup的概念，每个shuffleFileGroup会对应一批磁盘文件，磁盘文件的数量与下游stage的task数量是相同的。一个Executor上有多少个CPU core，就可以并行执行多少个task。而第一批并行执行的每个task都会创建一个shuffleFileGroup，并将数据写入对应的磁盘文件内。

当Executor的CPU core执行完一批task，接着执行下一批task时，下一批task就会复用之前已有的shuffleFileGroup，包括其中的磁盘文件。也就是说，此时task会将数据写入已有的磁盘文件中，而不会写入新的磁盘文件中。因此，consolidate机制允许不同的task复用同一批磁盘文件，这样就可以有效将多个task的磁盘文件进行一定程度上的合并，从而大幅度减少磁盘文件的数量，进而提升shuffle write的性能。

假设第二个stage有100个task，第一个stage有50个task，总共还是有10个Executor，每个Executor执行5个task。那么原本使用未经优化的HashShuffleManager时，每个Executor会产生500个磁盘文件，所有Executor会产生5000个磁盘文件的。但是此时经过优化之后，每个Executor创建的磁盘文件的数量的计算公式为：CPU core的数量 * 下一个stage的task数量。也就是说，每个Executor此时只会创建100个磁盘文件，所有Executor只会创建1000个磁盘文件。



SortShuffleManager运行原理

SortShuffleManager的运行机制主要分成两种，一种是普通运行机制，另一种是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。

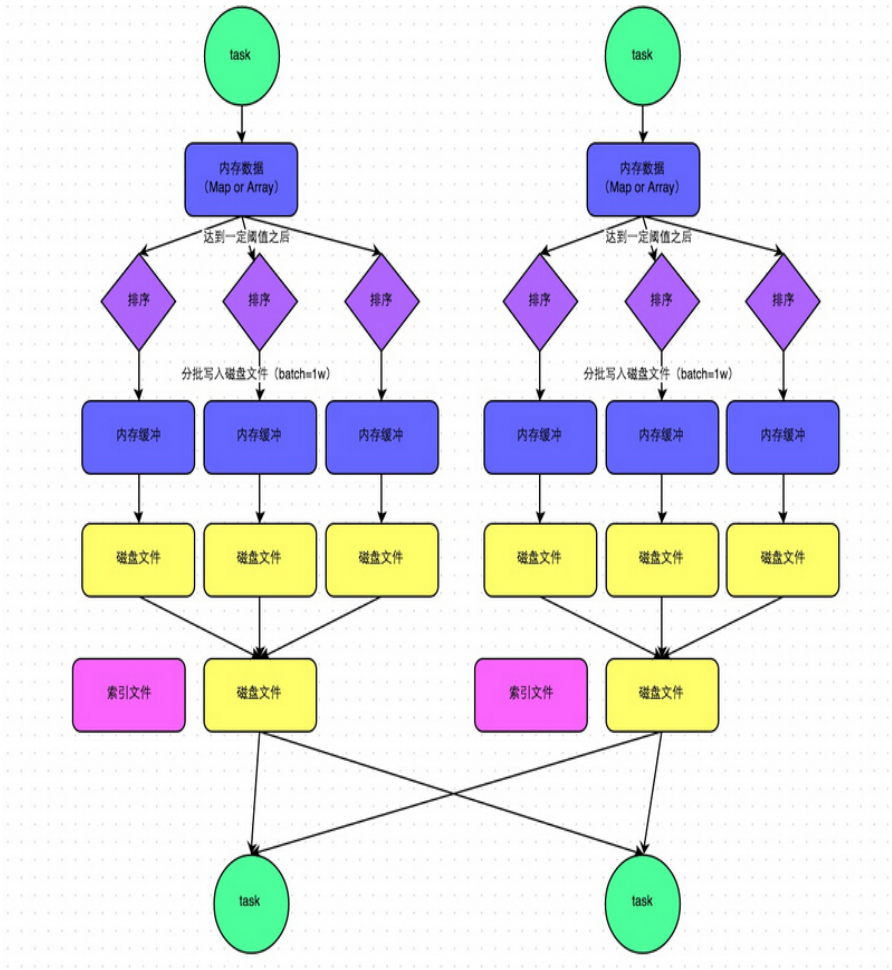
普通运行机制

下图说明了普通的SortShuffleManager的原理。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。



bypass运行机制

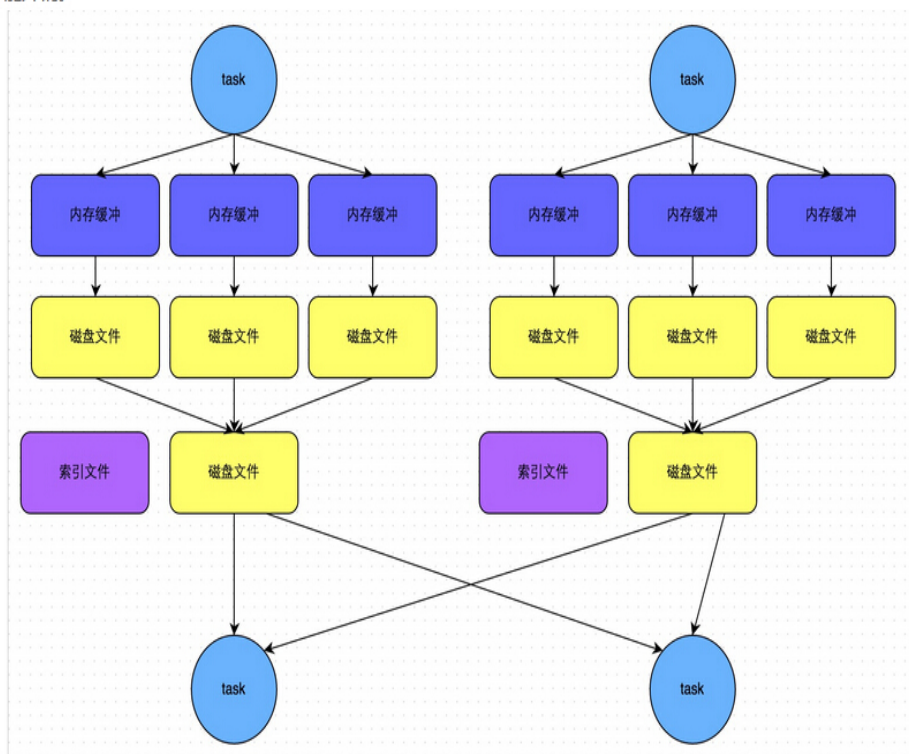
下图说明了bypass SortShuffleManager的原理。bypass运行机制的触发条件如下：

- shuffle map task数量小于spark.shuffle.sort.bypassMergeThreshold参数的值。
- 不是聚合类的shuffle算子（比如reduceByKey）。

此时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，也就省掉了这部分的性能开销。



shuffle相关参数调优

以下是Shuffle过程中的一些主要参数，这里详细讲解了各个参数的功能、默认值以及基于实践经验给出的调优建议。

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.shuffle.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%。

写在最后的话

本文分别讲解了开发过程中的优化原则、运行前的资源参数设置调优、运行中的数据倾斜的解决方案、为了精益求精的shuffle调优。希望大家能够在阅读本文之后，记住这些性能调优的原则以及方案，在Spark作业开发、测试以及运行的过程中多尝试，只有这样，我们才能开发出更优的Spark作业，不断提升其性能。

作者：[盟置 \(XGogo\)](#)

出处：<http://www.cnblogs.com/seaspring/>

本文版权归作者和博客园共有，欢迎转载，但未经作者同意必须保留此段声明，且在文章页面明显位置给出原文连接，否则保留追究法律责任的权利。

不能用于商业用户，若商业使用请联系：

QQ:107463366

微信:shinelife

好文要顶

关注我

收藏该文



XGogo

关注 - 2

粉丝 - 25

+加关注

« 上一篇：[Spark性能优化指南——基础篇转](#)

» 下一篇：[Spark性能优化指南——基础篇](#)

posted @ 2016-07-04 22:44 XGogo 阅读(8930) 评论(0) 编辑 收藏

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

【推荐】50万行VC++源码：大型组态工控、电力仿真CAD与GIS源码库

【推荐】加入腾讯云自媒体扶持计划，免费领取域名&服务器

【推荐】高性能云服务器2折起，0.73元/日节省80%运维成本

【新闻】H3 BPM体验平台全面上线



葡萄城报表
千万种报表 同一种选择

在线设计 报表

数据价值即刻体现

立即了解

最新IT新闻：

- 一箭十星 SpaceX成功完成今年最后一次火箭发射
 - Mozilla Firefox 60敲定为下个ESR延长支持版：明年5月8日上线
 - 蔚来汽车事钱还是生意 “换电”“代工”模式遭质疑
 - 百度索赔5000万为保卫人才？王劲多次回公司挖人
 - 世界卫生组织已确认：将首次把游戏成瘾列为精神疾病
- » 更多新闻...



阿里云 告别高昂运维费用 云计算全面助力

40+款核心产品免费半年 再+8000津贴任意采购

立即申请

最新知识库文章：

- 步入云计算
 - 以操作系统的角度述说线程与进程
 - 软件测试转型之路
 - 门内门外看招聘
 - 大道至简，职场上做人做事做管理
- » 更多知识库文章...

Copyright ©2017 XGogo

有大数据问题，欢迎加入初级讨论QQ群：大数据初级讨论群 156498981