关于数据倾斜

什么是数据倾斜

对于 spark / hadoop 这样分布式的计算系统而言，只要机器够，根本不担心数据量大，最怕的是不均。举个例子，如果有一个亿数据，平均分配到200台机器上，每台机器也就50万数据，处理起来还不是小菜一碟。但是如果出现1000万的数据分配在了190台机器上，另外的9000万数据分配到剩下的10台机器上，那对于这10台机器而言，就悲催了，这就是数据倾斜。这种分配不均是分布式计算最大的问题，一旦数据倾斜，整个计算过程就存在木桶效应，即使99.99的计算任务都完成了，也必须等剩下的0.01%完成才算成功。

数据倾斜的表现

Hadoop 中的数据倾斜（ hive 本质上也是 map reduce 来执行，表现类似，常出现在 distinct 和 join 过程）主要表现在ruduce 阶段卡在99.99%，一直99.99%不能结束。详细看日志可能会发现这样的问题

- 有一个或多几个 reduce 卡住
- 各种 container 报错 OOM
- 个别读写的数据量极大，至少远远超过其它正常的 reduce

Spark中的数据倾斜常表现为

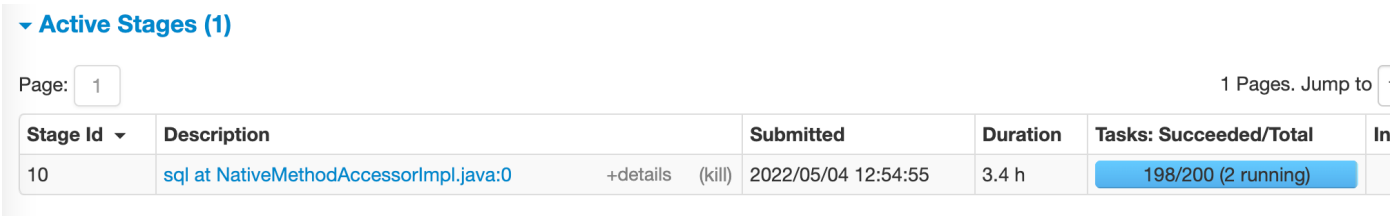
- 单个 Executor 执行时间特别久，整体任务卡在某个阶段不能结束
- Executor lost，OOM，Shuffle 过程出错
- Driver OOM
- 曾经正常运行的任务突然失败

此外，需要注意的是，在 Spark streaming 程序中，数据倾斜更容易出现，特别是在程序中包含一些类似 sql 的 join、group 这种操作的时候。 因为 Spark Streaming 程序在运行的时候，我们一般不会分配特别多的内存，因此一旦在这个过程中出现一些数据倾斜，就十分容易造成 OOM 。

数据倾斜问题定位

简而言之，多数 task 都执行得非常快，但个别 task 执行非常慢，导致整个任务不能结束。甚至用 SparkStreaming 做实时算法时候，可能一直会有 Executor 出现 OOM 的错误，但是其余的 Executor 内存使用率却很低。

如何定位，可以 Spark 的 web UI 查看，如果看到类似这种，那就基本上可以确定是数据倾斜了。

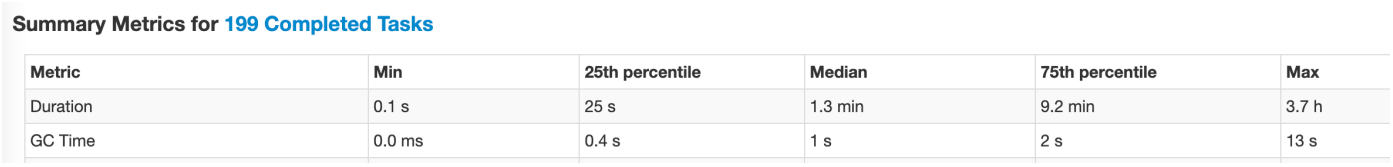


Active Stages (1)

Page: 1 1 Pages. Jump to

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	In
10	sql at NativeMethodAccessorImpl.java:0 +details (kill)	2022/05/04 12:54:55	3.4 h	198/200 (2 running)	

也可以详细的看到如下的计算4分位数



Summary Metrics for 199 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.1 s	25 s	1.3 min	9.2 min	3.7 h
GC Time	0.0 ms	0.4 s	1 s	2 s	13 s

通过UI甚至可以直接定位到是哪个过程发生的数据倾斜。数据倾斜只会发生在shuffle过程，可能触发数据倾斜的算子有：distinct、groupByKey、reduceByKey、aggregateByKey、join、cogroup、repartition等。

为了进一步定位数据倾斜是有哪些 key 导致的，可以对key进行统计，如果数据量大，可以进行抽样的形式。如果发现多数数据分布都较为平均，而个别数据比其他数据大上若干个数量级，则说明发生了数据倾斜。计数top就是造成数据倾斜的元凶。

如何解决数据倾斜

基本思路

既然数据倾斜是因为相同 key 的值被分配到少数几个节点上造成的单点问题，那么尽可能的的让 key 平均分配，问题就解决了。所以可以有几个思路：

- 数据预处理：在数据清洗过程中把计数特别多的 key 过滤掉（有损），或单独计算（无损）。
- 业务逻辑优化：从业务逻辑的层面上来优化数据倾斜，比如从数据的源头就尽可能避免单个计量很大的 key 。
- 参数优化：Hadoop 和 Spark 都自带了很多的参数和机制来调节数据倾斜，合理利用它们就能解决大部分问题。
- 程序优化：比如用 group 代替 count(distinct) 等。

过滤异常数据

直接过滤异常数据，对于有的情况下是有损的，但是对于部分情况，该方法也是可用的，比如本该在数据清洗阶段清洗的大量的NULL值、空值未被清洗。

对于这类情况，直接使用 where 条件过滤即可。

另外，如果部分看起来有用的数据，但是预估到实际计算结果中影响不大，也可酌情过滤。

提高shuffle并行度

Spark 在做 Shuffle 时，默认使用 HashPartitioner（非Hash Shuffle）对数据进行分区。如果并行度设置不合理，可能造成大量不相同的 key 对应的数据被分配到了同一个 task 上，从而造成数据倾斜。如果调整 Shuffle 并行度，使得原本被分配到同一 task 的不同 key 发配到不同 key 上处理，则可降低原 key 所需处理的数据量。

- 对于 RDD 算子，可在需要 Shuffle 的操作算子上直接设置并行度或者使用 spark.default.parallelism 设置。
- 对于 Spark SQL，可通过 spark.sql.shuffle.partitions 设置并行度，默认200
- 对于 DataFrame，也可以通过 spark.sql.shuffle.partitions 设置并行度

该方法的使用背景一般是由于并行度过小，导致大量不同的key对应的数据分配到了一个 task 上，一般增大并行度可以解决。该方法只需要设置参数，实现简单，代价相对最小。

不过该方法只是让每个 task 执行更少的 key ，无法解决单个 key 对应数据很大造成的数据倾斜问题，如果某些 key 的数据量非常大，即使一个 task 单独执行它，也会数据倾斜。所以从本质上，该方法只是缓解了数据倾斜，并没有彻底消除倾斜问题。

自定义partitioner

前面有提到，Spark 在做 Shuffle 时，默认使用 HashPartitioner（非Hash Shuffle）对数据进行分区。如果分区不合理，就会出现一定的数据倾斜问题。这时除了提高 shuffle 并行度外，使用自定义的 partitioner 也可以达到类似的目的，将原本被分配到同一个 task 的不同 key 分配到不同 task 上。

该方法与调整并行度效果类似，对于不同 key 被分配到同一个 task 的数据倾斜有效，但是对于单个 key 对应较大数据量时候同样无效。相比于提高 shuffle 并行度，它的优势是不影响原有的并行度设计，因为改变并行度，后续 stage 的并行度也会默认改变，可能会影响后续 stage 。不过由于是自定义，在设置上稍微麻烦，也不够灵活。

reduce端join转化为map端join

通过 Spark 的 Broadcast 机制，将 reduce 端 join 转化为 map 端 join ，避免 Shuffle 从而完全消除 Shuffle 带来的数据倾斜。

如果在 pyspark 中，可以 broadcast 模块，如

```
from pyspark.sql.functions import broadcast
result = broadcast(A).join(B, ["join_col"], "left")
# result = broadcast(A).join(B, A.join_col == B.join_col, "left")
```

如果是在 SQL 中，可以把小表写入 cache 中。

```
CACHE TABLE test_new;
INSERT OVERWRITE TABLE test_join
SELECT test_new.id, test_new.name
FROM test
JOIN test_new
ON test.id = test_new.id;
```

不过将小表写如 cache 中，并没有解决 Shuffle 的问题，只是在读取数据表时不再直接扫描 hive 表，而是扫描内存中缓存的表。要想实现 Broadcast ，还需 spark.sql.autoBroadcastJoinThreshold ，将 Broadcast 的阈值设置得足够大，如：

```
SET spark.sql.autoBroadcastJoinThreshold=104857600;
INSERT OVERWRITE TABLE test_join
SELECT test_new.id, test_new.name
FROM test
JOIN test_new
ON test.id = test_new.id;
```

该方法是通过 Broadcast 机制，将小数据集的数据广播到各 Executor 中，进而将 reduce 侧 join 转化为 map 侧 join 。所以该方法的适用范围是必须要有一个 join 是小数据。

拆分join再union

reduce 端 join 转化为 map 端 join，可以有效解决数据倾斜的问题，但是有一个硬性条件是必须要有一个小表，当两个表都很大时，这种方法就不能直接使用。这种时候就可以把对于数据量较大的为数不多的 key 单独提出来，相当于把将原RDD 拆分成两个 RDD ，一个是大量的但是单个数据量少的 key 形成 leftUnSkewRDD ，一个是少量的但是单个数据量多的 key 形成 leftSkewRDD 。这时候有两种方法。

第一种：

- leftUnSkewRDD 正常 join，因为不会存在数据倾斜。
- leftSkewRDD 则可以 Broadcast机制广播到各 Executor中，将 reduce侧 join 转化为 map 侧 join。
- 将两个 join 结果进行 union

第二种：（左右连接的两个表都存在倾斜 key ）

- 基于统计获取左侧表倾斜 key
- 提取左侧表中倾斜 key 对应的数据形成 leftSkewRDD，不倾斜的为 leftUnSkewRDD
- 提取右侧表中倾斜 key 的数据为 rightSkewRDD ，不倾斜的为 rightUnSkewRDD
- 给 leftSkewRDD 加上 1-n 的随机数形成新的 leftSkewRDD，也给 rightSkewRDD 加上1-n 的随机数形成新的 rightSkewRDD
- leftSkewRDD 与 rightSkewRDD 做笛卡尔积并去掉前缀，筛选key保留即可
- leftUnSkewRDD 与 rightUnSkewRDD 正常join
- 将两个 join 进行 union

相比于 map 侧的 join ，该方法可以适应更大的数据集，如果资源充足，倾斜数据和不倾斜数据还可以并行。不过如果倾斜key非常多，会导致笛卡尔积结果非常大，而且对倾斜 key 与非倾斜 key 分开处理，需要扫描数据集两遍，增加了开销。

大表key加盐，小表扩大N倍

如果出现数据倾斜的 key 比较多，上一种方法将这些大量的倾斜 key 分拆出来，意义不大。此时可直接对存在数据倾斜的数据集全部加上随机前缀，然后对另外一个不存在严重数据倾斜的数据集整体与随机前缀集作笛卡尔乘积。

该方法对大部分场景都适用，不过由于需要将另一个数据集整体扩大 N 倍，资源消耗非常大，尽量在有一个非倾斜表不是很大时才使用。

map端先局部聚合

在 map 端加个 combiner 函数进行局部聚合（使用 reduceByKey 而不是 groupByKey）。相当于提前进行 reduce ，把一个 mapper 中的相同 key 进行聚合，减少 shuffle 过程中数据量 以及 reduce 端的计算量。这种方法可以有效的缓解数据倾斜问题，但是如果导致数据倾斜的 key 大量分布在不同的 mapper 的时候，这种方法就不是很有效了。

两阶段聚合

对存在倾斜的数据进行两阶段聚合，加盐局部聚合再去盐全局聚合。

- 第一阶段：每个 key 都打上一个 1~n 的随机数（少于 key 本身的计数），如如 (hello, 1) (hello, 1) (hello, 1) (hello, 1) (hello, 1)，就会变成 (1_hello, 1) (3_hello, 1) (2_hello, 1) (1_hello, 1) (2_hello, 1)
- 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) (3_hello, 1)

- 然后将各个 key 的前缀给去掉，就会变成 (hello, 2) (hello, 2) (hello, 1)，再次进行全局聚合操作，就可以得到最终结果了，比如 (hello, 5)。

该方法需要进行两次 mapreduce，性能稍差。

hive中的优化

负载均衡处理

Hive 的底层计算模型和框架是 map reduce，如果发生数据倾斜，首先需要调整参数，再进行负载均衡处理。

```
set hive.map.aggr=true; -- (默认为ture) （map端的Combiner ）
set hive.groupby.skewindata=true;  --  (默认为false)
```

解决数据倾斜首先要进行负载均衡操作，将上面两个参数设定为 true，而 map reduce 进程则会生成两个额外的 MR Job，这两个任务的主要操作如下:

- 第一步：MR Job 中Map 输出的结果集合首先会随机分配到 Reduce 中，然后每个 Reduce 做局部聚合操作并输出结果，这样处理的原因是相同的Group By Key有可能被分发到不同的 Reduce Job中，从而达到负载均衡的目的。
- 第二步：MR Job 再根据预处理的数据结果按照 Group By Key 分布到 Reduce 中（这个过程可以保证相同的 Group By Key 被分布到同一个 Reduce 中），最后完成聚合操作。

使用Mapjoin进行关联

本质上就是将 reduce 端 join 转化为 map 端 join 。避免某个过多的key分发到同一个Reduce Job中。这个过程不会经历Shuffle阶段和Reduce阶段，直接由Map端输出结果文件。

以下2个参数控制Mapjoin的使用：

```
set hive.auto.convert.join= ture;  -- 设置自动化开启(默认为false)
hive.mapjoin.smalltable.filesize=25000000;  -- 设置小表大小的上限（默认为25M）
```

执行Mapjoin 操作，具体语句如下：

```
select
/* +mapjoin(a) */
a.*, b.*
from smalltable a
join bigtable b
on a.id = b.id
```

小表放入子查询

小表的查询结果相当于大表的一个where条件（需要注意的是小表的输出结果不能太大）。

```sql
select b.*
from bigtable b
where id in (select id from smalltable)
```

用group by 代替 count distinct

由于SQL中的Distinct操作本身会有一个全局排序的过程，一般情况下，不建议采用Count Distinct方式进行去重计数，除非表的数量比较小。当SQL中不存在分组字段时，Count Distinct操作仅生成一个Reduce 任务，该任务会对全部数据进行去重统计；当SQL中存在分组字段时，可能某些Reduce 任务需要去重统计的数量非常大。如：

```sql
-- 直接distinct
select distinct id as id_cnt
from dattable;
-- 适用group by 替换
select t.id from
(select id
from dattable
group by id) t ;
```

shuffle相关参数调优

spark.shuffle.file.buffer

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

spark.reducer.maxSizeInFlight

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

spark.shuffle.io.maxRetries

- 默认值：3
- 参数说明：shuffle read task从shuffle write task所在节点拉取属于自己的数据时，如果因为网络异常导致拉取失败，是会自动进行重试的。该参数就代表了可以重试的最大次数。如果在指定次数之内拉取还是没有成功，就可能会导致作业执行失败。
- 调优建议：对于那些包含了特别耗时的shuffle操作的作业，建议增加重试最大次数（比如60次），以避免由于JVM的full gc或者网络不稳定等因素导致的数据拉取失败。在实践中发现，对于针对超大数据量（数十亿~

上百亿）的shuffle过程，调节该参数可以大幅度提升稳定性。

spark.shuffle.io.retryWait

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

spark.shuffle.memoryFraction

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

spark.shuffle.manager

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

spark.shuffle.sort.bypassMergeThreshold

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

spark.shuffle.consolidateFiles

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

参考：

漫谈千亿级数据优化实践：数据倾斜（纯干货）

Spark性能优化之道——解决Spark数据倾斜（Data Skew）的N种姿势

Spark 数据倾斜及其解决方案

深入浅出Hive数据倾斜

基于SparkUI Spark Sql 数据倾斜特征及解决方法

Spark性能优化指南——高级篇