



下载APP



25 | Spark 3.0（二）：DPP特性该怎么用？

2021-05-10 吴磊

Spark性能调优实战

[进入课程 >](#)**讲述：吴磊**

时长 13:48 大小 12.64M



你好，我是吴磊。

DPP（Dynamic Partition Pruning，动态分区剪裁）是 Spark 3.0 版本中第二个引人注目的特性，它指的是在星型数仓的数据关联场景中，可以充分利用过滤之后的维度表，大幅削减事实表的数据扫描量，从整体上提升关联计算的执行性能。


今天这一讲，我们就通过一个电商场景下的例子，来说说什么分区剪裁，什么是动态分区剪裁，它的作用、用法和注意事项，让你一次就学会怎么用好 DPP。



分区剪裁


我们先来看这个例子。在星型 (Start Schema) 数仓中，我们有两张表，一张是订单表 orders，另一张是用户表 users。显然，订单表是事实表 (Fact)，而用户表是维度表 (Dimension)。业务需求是统计所有头部用户贡献的营业额，并按照营业额倒序排序。那这个需求该怎么实现呢？

首先，我们来了解一下两张表的关键字段，看看查询语句应该怎么写。

 复制代码

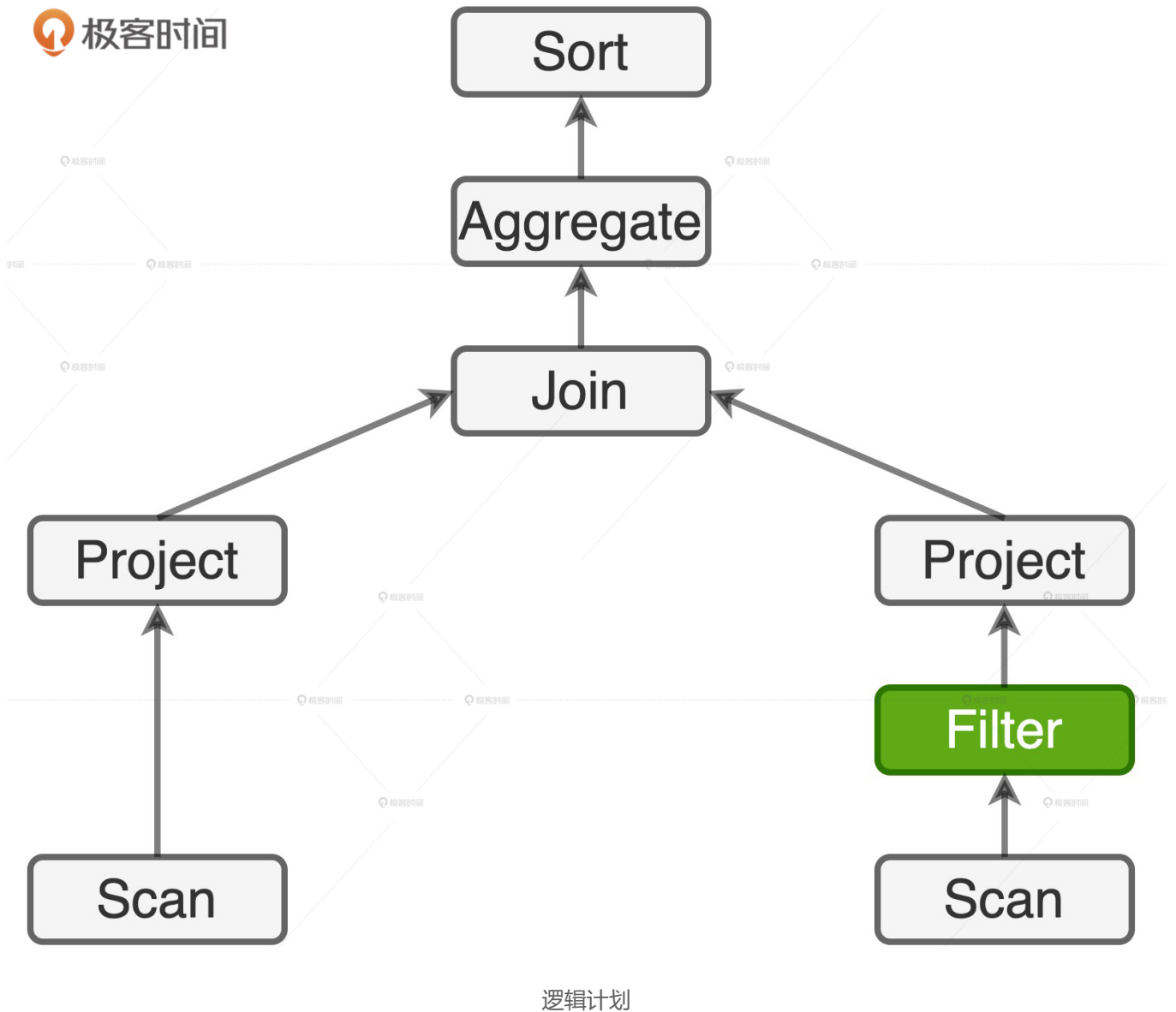
```
1 // 订单表orders关键字段
2 userId, Int
3 itemId, Int
4 price, Float
5 quantity, Int
6
7 // 用户表users关键字段
8 id, Int
9 name, String
10 type, String //枚举值，分为头部用户和长尾用户
11
```

给定上述数据表，我们只需把两张表做内关联，然后分组、聚合、排序，就可以实现业务逻辑，具体的查询语句如下。

 复制代码

```
1 select (orders.price * order.quantity) as income, users.name
2 from orders inner join users on orders.userId = users.id
3 where users.type = 'Head User'
4 group by users.name
5 order by income desc
```

看到这样的查询语句，再结合 Spark SQL 那几讲学到的知识，我们很快就能画出它的逻辑执行计划。



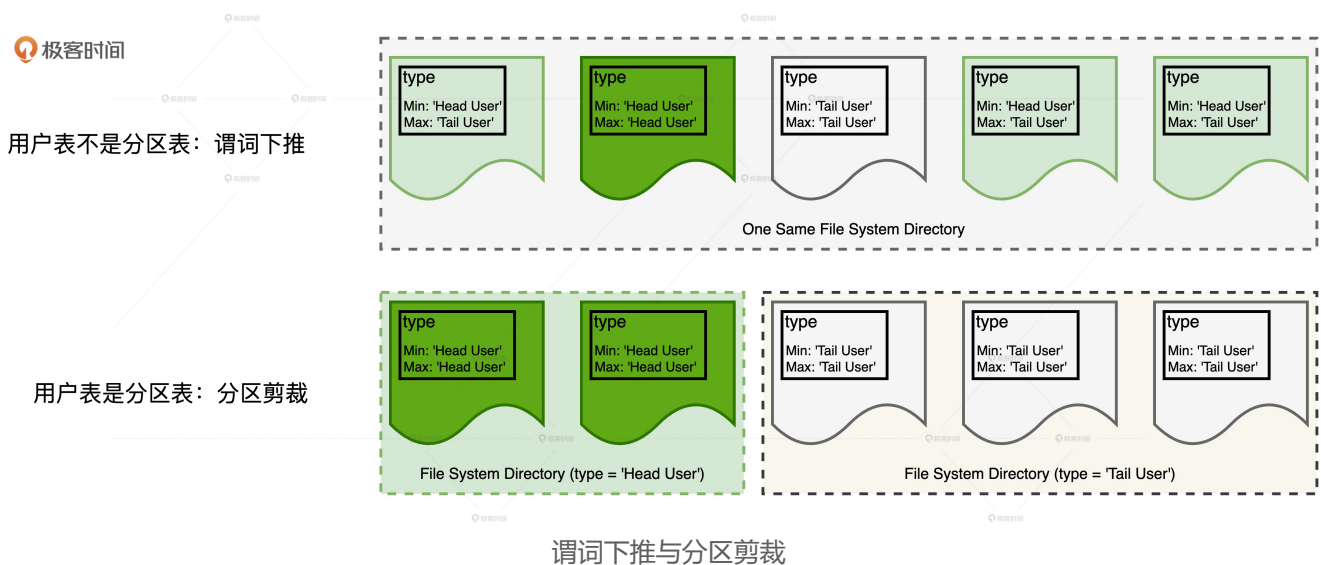
由于查询语句中事实表上没有过滤条件，因此，在执行计划的左侧，Spark SQL 选择全表扫描的方式来投影出 `userId`、`price` 和 `quantity` 这些字段。相反，维度表上有过滤条件 `users.type = 'Head User'`，因此，Spark SQL 可以应用谓词下推规则，把过滤操作下推到数据源之上，来减少必需的磁盘 I/O 开销。

虽然谓词下推已经很给力了，但如果用户表支持分区剪裁 (Partition Pruning)，I/O 效率的提升就会更加显著。那什么是分区剪裁呢？实际上，分区剪裁是谓词下推的一种特例，它指的是在分区表中下推谓词，并以文件系统目录为单位对数据集进行过滤。分区表就是通过指定分区键，然后使用 `partitioned by` 语句创建的数据表，或者是使用 `partitionBy` 语句存储的列存文件 (如 Parquet、ORC 等)。

相比普通数据表，分区表特别的地方就在于它的存储方式。对于分区键中的每一个数据值，分区表都会在文件系统中创建单独的子目录来存储相应的数据分片。拿用户表来举

例，假设用户表是分区表，且以 type 字段作为分区键，那么用户表会有两个子目录，前缀分别是 “Head User” 和 “Tail User”。数据记录被存储于哪个子目录完全取决于记录中 type 字段的值，比如：所有 type 字段值为 “Head User” 的数据记录都被存储到前缀为 “Head User” 的子目录。同理，所有 type 字段值为 “Tail User” 的数据记录，全部被存放到前缀为 “Tail User” 的子目录。

不难发现，**如果过滤谓词中包含分区键，那么 Spark SQL 对分区表做扫描的时候，是完全可以跳过（剪掉）不满足谓词条件的分区目录，这就是分区剪裁。**例如，在我们的查询语句中，用户表的过滤谓词是 “users.type = ‘Head User’ ”。假设用户表是分区表，那么对于用户表的数据扫描，Spark SQL 可以完全跳过前缀为 “Tail User” 的子目录。



通过与谓词下推作对比，我们可以直观地感受分区剪裁的威力。如图所示，上下两行分别表示用户表在不做分区和做分区的情况下，Spark SQL 对于用户表的数据扫描。在不做分区的情况下，用户表所有的数据分片全部存于同一个文件系统目录，尽管 Parquet 格式在注脚 (Footer) 中提供了 type 字段的统计值，Spark SQL 可以利用谓词下推来减少需要扫描的数据分片，但由于很多分片注脚中的 type 字段同时包含 ‘Head User’ 和 ‘Tail User’（第一行 3 个浅绿色的数据分片），因此，用户表的数据扫描仍然会涉及 4 个数据分片。

相反，当用户表本身就是分区表时，由于 type 字段为 ‘Head User’ 的数据记录全部存储到前缀为 ‘Head User’ 的子目录，也就是图中第二行浅绿色的文件系统目录，这个目录中仅包含两个 type 字段全部为 ‘Head User’ 的数据分片。这样一来，Spark SQL 可以完全跳过其他子目录的扫描，从而大幅提升 I/O 效率。

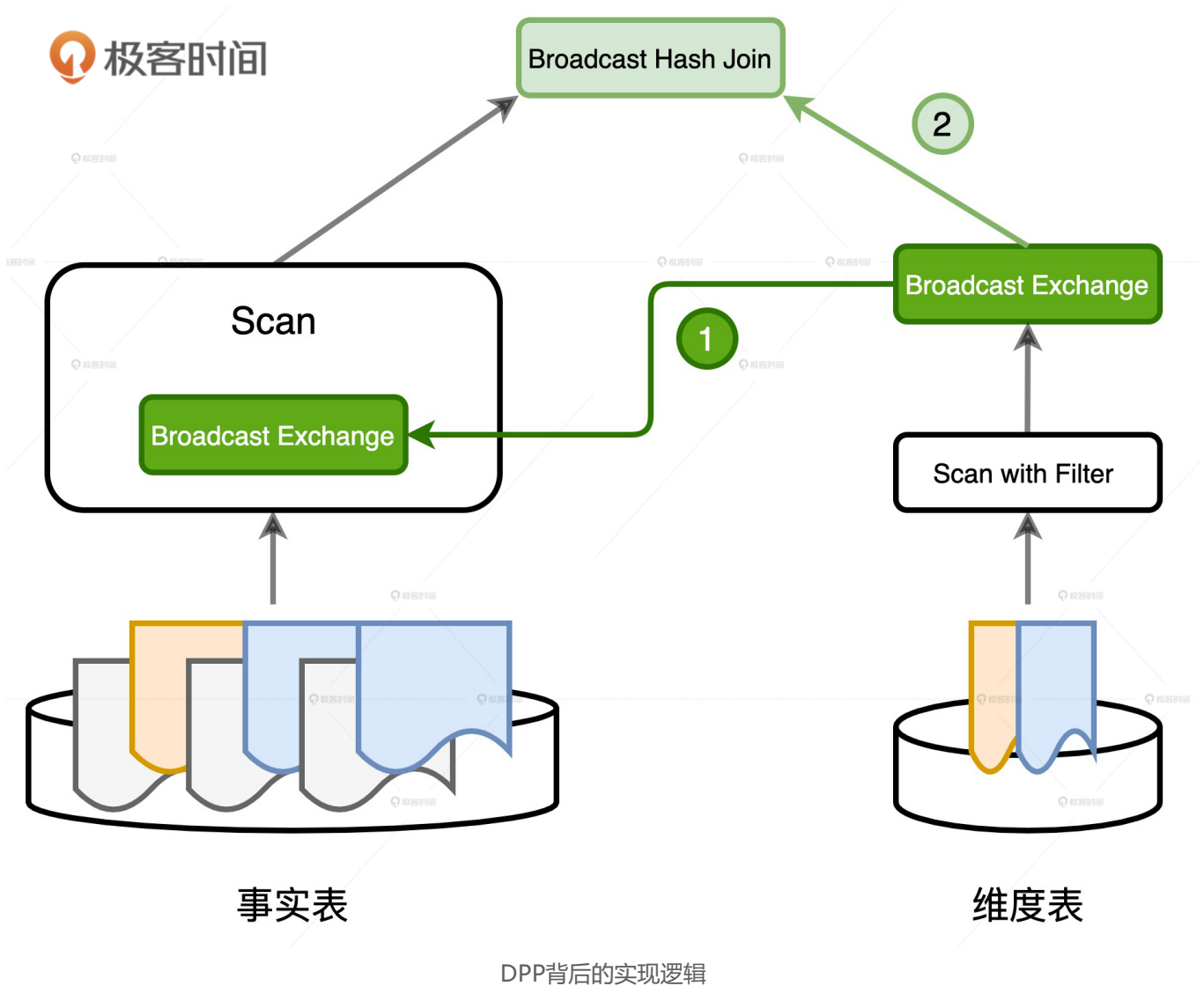
你可能会说：“既然分区剪裁这么厉害，那么我是不是也可以把它应用到事实表上去呢？毕竟事实表的体量更大，相比维度表，事实表上 I/O 效率的提升空间更大。”没错，如果事实表本身就是分区表，且过滤谓词中包含分区键，那么 Spark SQL 同样会利用分区剪裁特性来大幅减少数据扫描量。

不过，对于实际工作中的绝大多数关联查询来说，事实表都不满足分区剪裁所需的前提条件。比如说，要么事实表不是分区表，要么事实表上没有过滤谓词，或者就是过滤谓词不包含分区键。就拿电商场景的例子来说，查询中压根就没有与订单表相关的过滤谓词。因此，即便订单表本身就是分区表，Spark SQL 也没办法利用分区剪裁特性。

对于这样的关联查询，我们是不是只能任由 Spark SQL 去全量扫描事实表呢？要是在以前，我们还没什么办法。不过，有了 Spark 3.0 推出的 DPP 特性之后，情况就大不一样了。

动态分区剪裁

我们刚才说了，DPP 指的是在数据关联的场景中，Spark SQL 利用维度表提供的过滤信息，减少事实表中数据的扫描量、降低 I/O 开销，从而提升执行性能。那么，DPP 是怎么做到这一点的呢？它背后的逻辑是什么？为了方便你理解，我们还用刚刚的例子来解释。



DPP背后的实现逻辑

首先, 过滤条件 `users.type = 'Head User'` 会帮助维度表过滤一部分数据。与此同时, 维度表的 ID 字段也顺带着经过一轮筛选, 如图中的步骤 1 所示。经过这一轮筛选之后, 保留下来的 ID 值, 仅仅是维度表 ID 全集的一个子集。

然后, 在关联关系也就是 `orders.userId = users.id` 的作用下, 过滤效果会通过 `users` 的 ID 字段传导到事实表的 `userId` 字段, 也就是图中的步骤 2。这样一来, 满足关联关系的 `userId` 值, 也是事实表 `userId` 全集中中的一个子集。把满足条件的 `userId` 作为过滤条件, 应用 (Apply) 到事实表的数据源, 就可以做到减少数据扫描量, 提升 I/O 效率, 如图中的步骤 3 所示。

DPP 正是基于上述逻辑, 把维度表中的过滤条件, 通过关联关系传导到事实表, 从而完成事实表的优化。虽然 DPP 的运作逻辑非常清晰, 但并不是所有的数据关联场景都可以享受到 DPP 的优化机制, 想要利用 DPP 来加速事实表数据的读取和访问, 数据关联场景还要满足三个额外的条件。

首先, DPP 是一种分区剪裁机制, 它是以分区为单位对事实表进行过滤。结合刚才的逻辑, 维度表上的过滤条件会转化为事实表上 Join Key 的过滤条件。具体到我们的例子中, 就是 `orders.userId` 这个字段。显然, DPP 生效的前提是事实表按照 `orders.userId` 这一列预先做好了分区。因此, **事实表必须是分区表, 而且分区字段 (可以是多个) 必须包含 Join Key。**

其次, 过滤效果的传导, 依赖的是等值的关联关系, 比如 `orders.userId = users.id`。因此, **DPP 仅支持等值 Joins, 不支持大于、小于这种不等值关联关系。**

此外, DPP 机制得以实施还有一个隐含的条件: **维度表过滤之后的数据集要小于广播阈值。**

拿维度表 `users` 来说, 满足过滤条件 `users.type = 'Head User'` 的数据集, 要能够放进广播变量, DPP 优化机制才能生效。为什么会这样呢? 这就要提到 DPP 机制的实现原理了。

结合刚才对于 DPP 实现逻辑的分析和推导, 我们不难发现, 实现 DPP 机制的关键在于, 我们要让处理事实表的计算分支, 能够拿到满足过滤条件的 Join Key 列表, 然后用这个列表来对事实表做分区剪裁。那么问题来了, 用什么办法才能拿到这个列表呢?

Spark SQL 选择了一种 “一箭双雕” 的做法: **使用广播变量封装过滤之后的维度表数据。** 具体来说, 在维度表做完过滤之后, Spark SQL 在其上构建哈希表 (Hash Table), 这个哈希表的 Key 就是用于关联的 Join Key。在我们的例子中, Key 就是满足过滤 `users.type = 'Head User'` 条件的 `users.id`; Value 是投影中需要引用的数据列, 在之前订单表与用户表的查询中, 这里的引用列就是 `users.name`。



哈希表构建完毕之后，Spark SQL 将其封装到广播变量中，这个广播变量的作用有二。第一个作用就是给事实表用来做分区剪裁，如图中的步骤 1 所示，哈希表中的 Key Set 刚好可以用来给事实表过滤符合条件的数据分区。

第二个作用就是参与后续的 Broadcast Join 数据关联，如图中的步骤 2 所示。这里的哈希表，本质上就是 Hash Join 中的 Build Table，其中的 Key、Value，记录着数据关联中所需的所有字段，如 users.id、users.name，刚好拿来和事实表做 Broadcast Hash Join。

因此你看，鉴于 Spark SQL 选择了广播变量的实现方式，要想有效利用 DPP 优化机制，我们就必须要确保，过滤后的维度表刚好能放到广播变量中去。也因此，我们必须谨慎对待配置项 spark.sql.autoBroadcastJoinThreshold。

小结

这一讲，我们围绕动态分区剪裁，学习了谓词下推和分区剪裁的联系和区别，以及动态分区剪裁的定义、特点和使用方法。

相比于谓词下推，分区剪裁往往能更好地提升磁盘访问的 I/O 效率。

这是因为，谓词下推操作往往是根据文件注脚中的统计信息完成对文件的过滤，过滤效果取决于文件中内容的“纯度”。分区剪裁则不同，它的分区表可以把包含不同内容的文件，隔离到不同的文件系统目录下。这样一来，包含分区键的过滤条件能够以文件系统目录为粒度对磁盘文件进行过滤，从而大幅提升磁盘访问的 I/O 效率。

而动态分区剪裁这个功能主要用在星型模型数仓的数据关联场景中，它指的是在运行的时候，Spark SQL 利用维度表提供的过滤信息，来减少事实表中数据的扫描量、降低 I/O 开销，从而提升执行性能。

动态分区剪裁运作的背后逻辑，是把维度表中的过滤条件，通过关联关系传导到事实表，来完成事实表的优化。在数据关联的场景中，开发者要想利用好动态分区剪裁特性，需要注意 3 点：

事实表必须是分区表，并且分区字段必须包含 Join Key

动态分区剪裁只支持等值 Joins, 不支持大于、小于这种不等值关联关系

维度表过滤之后的数据集, 必须要小于广播阈值, 因此, 开发者要注意调整配置项 `spark.sql.autoBroadcastJoinThreshold`

每日一练

1. 如果让你重写 DPP 实现机制, 你有可能把广播阈值的限制去掉吗? (提示: 放弃使用 Broadcast Hash Join 的关联方式, 但仍然用广播变量来做分区剪裁。)
2. 要让事实表拿到满足条件的 Join Key 列表, 除了使用广播变量之外, 你觉得还有其他的途径吗?

期待在留言区看到你的思考和答案, 我们下一讲见!

提建议

学习推荐

大数据开发「面试必备 100 题 + 视频课程」免费领

立即领取

仅限 99 人



© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 24 | Spark 3.0 (一) : AQE的3个特性怎么才能用好?

下一篇 26 | Join Hints指南: 不同场景下, 如何选择Join策略?

精选留言 (9)

写留言



辰

2021-05-13

这个dpp限制其实蛮多的, 就比如拿课件中的例子, 通过user.id关联, 而且这个关联应该也是大部分场景都用到的, 但是的事实表居然要按userid进行分区, 不管是电商项目还是其他项目, userid肯定是比较多的, 比如10万个user就对应10万userid,也就对应了10万个分区, 这怕不是要玩爆系统哦, 不知道理解的对不对

展开 ∨

作者回复: 理解得非常的对~

分区键和Join Keys的选取, 确实是个博弈, 咱们课件中的例子, 为了方便说明DPP的机制和原理, 所以没有深究。但是你说的非常对, 就是这个Join Keys, 确实本身就是个限制。因为就像你说的, 平时的数据关联, 都是ID类的Join Keys比较多, 而这样的数据列, 往往是不适合做分区键的。

所以要想充分利用DPP, 确实是得在数仓规划的最开始, 就把以后要用的常用查询都考虑到, 在做分区表设计的时候, 尽量做到分区处理本身与查询执行性能之间的平衡。



2



kingcall

2021-05-11

回答:

问题1 Broadcast Hash Join 本身也是依赖Broadcast 来完成的, 所以Broadcast 肯定是可以完成这个需求的, 但是有一个问题 Broadcast Hash Join 有spark.sql.autoBroadcastJoinThreshold 这个限制条件, 但是这个条件也仅仅是限制了是否自动发生Broadcast Join, 所以手动Broadcast话就没这个限制了或者使用hints, 所以想怎么玩就怎么玩了, 不...

展开 ∨

作者回复: 问题1可以参考下面几个同学的思路~ 强制广播和依靠广播阈值 (也就是spark.sql.autoBroadcastJoinThreshold) 来广播, 其实本质上区别不大。

DPP的核心机制, 还是用维表的Key set来过滤事实表数据。这个Key set全集是必需的, Spark的默认实现, 是采用了广播变量, 这个广播的哈希表还能用于后续的BHJ, 一箭双雕, 但是也因此

而受限于广播阈值和8G的限制。

问题2答得不错，就是用其他方式获取Key set全集，分布式缓存、存储，其实都行，虽然都会引入数据分发，但去掉了广播的限制~ 在DPP的第二个环节，也就是两表过滤之后的Join，其实采用SMJ或者是Shuffle Hash Join，就可以再次去掉广播阈值的限制。

所以总结下来，要想去掉广播的限制，需要对两个环节进行改进，一个是Key set的全集存储、分发；一个是过滤之后两表的Join。



Fendora范东_

2021-05-10

有几个疑问，麻烦磊哥指点下

- 1.文中所写SQL，最终有3个stage,读数据的两个stage按理说是没有依赖关系的，资源足够情况下是可以并发运行的，这就和DPP运行逻辑相违背了，这块spark是怎么做的呢？
- 2.有没有这样一种可能:DPP生效，即维度表做了过滤查询，事实表基于维度表广播过来的hashtable再做过滤查询，这个过程比DPP不生效，即两张表并发进行过滤查询，过程还...
展开 ∨

作者回复: 好问题~

先说前两个问题，其实这个问题是，Spark是先扫描事实表，还是先做DPP，答案肯定是后者。

说说DPP的意义，这里有个关键，就是Spark “一上来”、在最开始，是怎么知道谁是事实表、维度表的？其实，Spark SQL的静态优化，关于数据统计，比如针对Parquet、CSV等等不同的数据源，它其实不是什么都不做，它至少会统计“表大小”这些最基本的信息，根据这些size类型的信息，它其实可以判断哪个是事实表，哪个是维度表。

如此一来，它先去判断DPP的前提条件是否成立，成立的话，就走一波DPP。

再来说第3题，确实是好问题，不过这里其实本质上是和DPP的实现机制有关，也就是说，它用“一箭双雕”的方式，一方面利用广播来过滤，一方面顺水推舟，用ReuseExchange（内存中的reuse，跟之前的磁盘reuse不同）再次利用广播来完成Broadcast Hash Join，一石二鸟。

再者，DPP实际上是先于AQE Join策略调整的，因为这里还没有Shuffle呢，还记得AQE的触发机制是Shuffle吗？因此DPP相当于截了AQE的“胡儿”（打麻将的截胡儿），不过其实谁先谁后的，没那么重要，反正我们享受到了Broadcast Hash Join的性能提升。

再者，DPP“水到渠成”的BHJ，其实还有一个优势，就是相比AQE，它不需要Shuffle就能触

发, 因此不需要Shuffle Write阶段的计算, 才能做优化, 所以说DPP带来的BHJ, 相比AQE的Join策略调整, 其实是更优的优化~

**zxk**

2021-05-11

问题一: 放弃使用 BHJ 的话, 其中一个表经过裁剪过滤后, 使用广播变量只广播 Join Key 而非整个表的数据, 仍可以实现 DPP, 但个人仍为仍需要针对广播的 Join Key 加上一个 Threshold, 否则可能将 Driver 撑爆。

问题二: 如果维度表的过滤条件正好是 Join key, 同时也是事实表的分区目录, 也可以考虑先将这个过滤条件直接推到事实表作为一个靠近数据源的 filter 条件。...

展开 ∨

作者回复: 满分💯, 答得好, 问得也好~ 思考的很深入

先说第一题, 没错, 只广播Join Keys, 广播阈值的限制会低很多, 不过还是需要利用广播机制, 来传递“过滤条件”, 事实表拿到“过滤条件”来降低扫描量。因此就像你说的, 虽然限制降低, 但是还是有Join Keys超大撑爆广播阈值、甚至是Driver的风险。所以要想完全去掉广播, 那就必须要换一种网络分发方式, 举个例子, 比如分布式文件系统、分布式缓存, 也就是所有Executors都可以通过他们拿到“过滤条件”。当然, 你可能会说, 这样效率就低很多了, 确实, 不过其实这道题的目的, 就是鼓励大家脑洞、多思考、发散思维, 思维越发散, 其实对于Spark为什么采用广播机制的理解就越深刻。

再说第二题, 很不错的思路~ 相当于是Query rewrite, 就是直接把维表Join Key上的过滤条件, 在SQL查询优化阶段就传递给事实表, 不错不错~ 不过, Spark并没有Query rewrite这个阶段, 但是思路我很喜欢~ 其实传统的DBMS都是有Query rewrite这个环节的, Spark偷懒, 这部分没做。

最后再说DPP的意义, 就是你说的, Spark “一上来”、最开始怎么知道谁是事实表、维度表。其实它还真是知道的, Spark SQL的静态优化, 关于数据统计, 比如针对Parquet、CSV等等不同的数据源, 它其实不是什么都不做, 它至少会统计“表大小”这些最基本的信息, 根据这些size类型的信息, 它其实可以判断哪个是事实表, 哪个是维度表。

**张守一**

2021-05-18

老师 如果以userid作为分区字段 相比于日期等分区字段 分区过多 但id等字段又常常作为join key 这个怎么解决呢

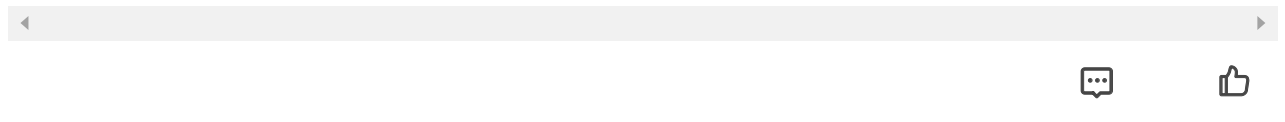
展开 ∨

作者回复: 好问题, 这块确实是DPP的“硬伤”。

就像你说的, Join Keys往往是cardinality比较高的字段, 比如userId; 而分区键往往是要选择那些cardinality比较低的字段, 否则数据的存储就会非常的分散。

一个cardinality高, 一个cardinality低, 两者相互矛盾。这块确实比较“蛋疼”, 如果查询中的Join Keys多属于userId这种cardinality非常高的字段, 坦白说咱们还真没什么好办法。

对于那些cardinality较高的Join Key, 我们就需要做取舍, 也就是在存储效率和DPP之间做权衡。如果查询效率是第一优先级, 那么我们其实还是可以强行对cardinality较高的Join Key做分区键。但如果相反, 存储效率最大的concern, 那么也就只好放弃Join Key做分区键, 放弃DPP优化机制。



aof

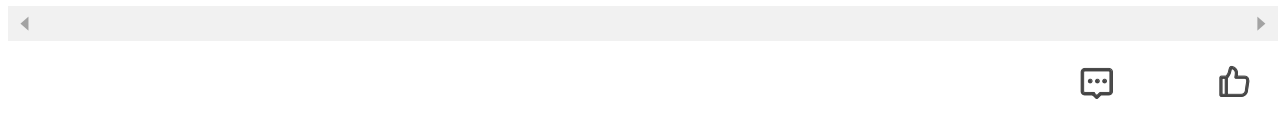
2021-05-16

可以在逻辑计划优化的时候, 就直接将维表的过滤条件通过join条件传导到事实表, 减少数据的扫描

展开 ∨

作者回复: 不错的思路~ 不过这里有个前提条件, 就是对于维表上的过滤字段, 事实表上也要有才行。比如, dim.c1 = "xxx", 那么就要求fact上面也要有c1字段, 否则仅在逻辑计划阶段, 是做不过过滤条件传导的。

这个思路本质上其实是Query Re-writer, 就是在逻辑计划阶段把查询进行重写。



little_fly

2021-05-15

老师, 能不能对Spark应用程序中经常遇到的一些报错, 写篇分析文章? 比如, 类似如下的报错信息:

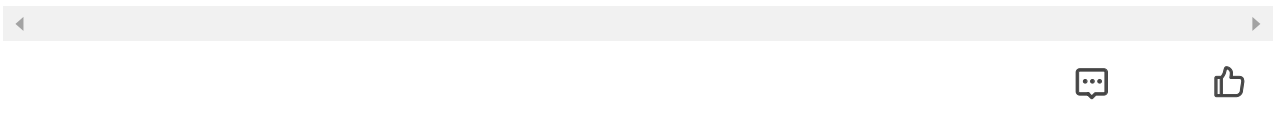
1. org.apache.spark.memory.SparkOutOfMemoryError: Unable to acquire 172 bytes of memory, got 0
2. ERROR cluster.YarnScheduler: Lost executor 4 on node-3: Container killed by Y...

展开 ∨

作者回复: 非常好的建议, 这块我是这么想的, 后续我会整理一个github repo, 把专栏中涉及的代码、数据、结果、以及常见问题等内容汇总到这个项目中去。当然内容不止这些, 结合大家的需要, 我们还会持续不断地向其中添加诸如笔试面试题、工作机会、职业发展等内容, 把potatoes项目打造成我们共有的Spark私塾。看到你的这个问题, 我觉得可以多加一类内容, 就是常见报错, 把这些报错分门别类, 报错原因、应对办法, 等等。

对于这个github repo, 我是打算用开源的思路来做, 比如就这个常见报错, 我是期望大家群策群力, 一起共享工作中遇到的各种报错, 然后可以一起来分析背后的root cause和解决办法~ 然后不断地把它丰富、完善~

等专栏写完, 我就会着手做这件事, 到时候一起弄哈~



Fendora范东_

2021-05-10

问题回答

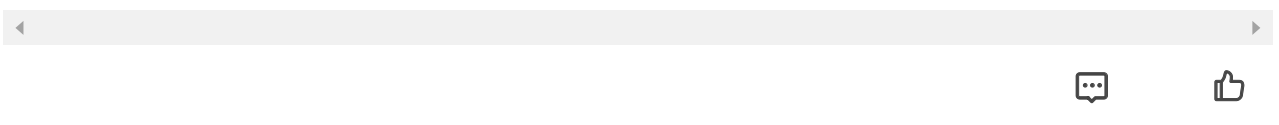
- 1.只广播join key的key set, 这个数据量应该还是比较小的。
- 2.可以把join key的key set缓存到内存中。

展开 ∨

作者回复: 思路不错~ 其实这里面的关键点, 就是让每一个Executors都能拿到过滤之后的Join Keys的全集。

第一个问题, 如果仅仅广播Key Set的话, 就像你说的, 数据集很小, 对于广播阈值的要求会低得多。实际上, 最极端地, 还可以只广播Key Set的哈希值, 如果Join Keys本身很大的话。现在的实现方式之所以附加Payloads, 根本原因是后面打算“一箭双雕”地用BHI来实现两表关联。实际上, 严格来说, 只要拿到Key Set全集, 做完过滤, 采用Shuffle Joins也不是不可以, 毕竟过滤之后的事实表也变小了。

第二个问题, 就像刚刚说的, 只要是能拿到Key Set全集, 就能满足预过滤事实表的这个需求, 所以像是Cache、分布式存储其实都行, 只不过效率上不如广播变量, 因为前两者还会以Task为粒度走网络, 广播仅走一次网络、然后所有Tasks受益。



斯盖丸

2021-05-10

老师, 如果事实表的join字段有好几个, 其中只有一个是分区字段, 那还能享受到DPP

吗？

展开 ✓

作者回复: 可以的哈~

