



下载APP



27 | 大表Join小表：广播变量容不下小表怎么办？

2021-05-14 吴磊

Spark性能调优实战

[进入课程 >](#)**讲述：吴磊**

时长 21:00 大小 19.24M



你好，我是吴磊。

在数据分析领域，大表 Join 小表的场景非常普遍。不过，大小是个相对的概念，通常来说，大表与小表尺寸相差 3 倍以上，我们就将其归类为“大表 Join 小表”的计算场景。因此，大表 Join 小表，仅仅意味着参与关联的两张表尺寸悬殊。

对于大表 Join 小表这种场景，我们应该优先考虑 BHJ，它是 Spark 支持的 5 种 Join 策略中执行效率最高的。**BHJ 处理大表 Join 小表时的前提条件是，广播变量能够容纳小表的全量数据。**但是，如果小表的数据量超过广播阈值，我们又该怎么办呢？



今天这一讲，我们就结合 3 个真实的业务案例，来聊一聊这种情况的解决办法。虽然这 3 个案例不可能覆盖“大表 Join 小表”场景中的所有情况，但是，分析并汇总案例的应对策

略和解决办法，有利于我们在调优的过程中开阔思路、发散思维，从而避免陷入“面对问题无所适从”的窘境。

案例 1：Join Key 远大于 Payload

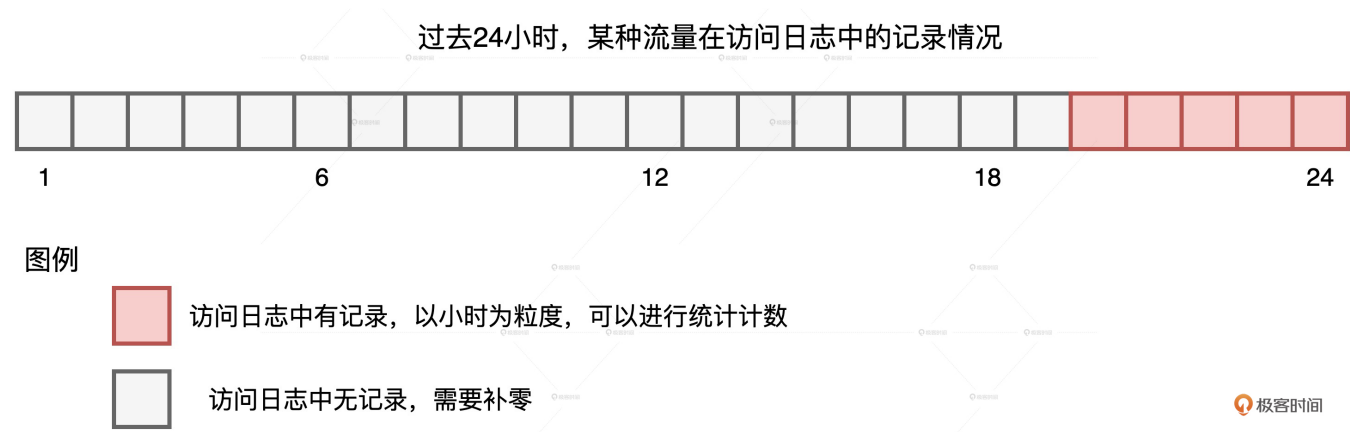
在第一个案例中，大表 100GB、小表 10GB，它们全都远超广播变量阈值（默认 10MB）。因为小表的尺寸已经超过 8GB，在大于 8GB 的数据集上创建广播变量，Spark 会直接抛出异常，中断任务执行，所以 Spark 是没有办法应用 BHJ 机制的。那我们该怎么办呢？先别急，我们来看看这个案例的业务需求。

这个案例来源于计算广告业务中的流量预测，流量指的是系统中一段时间内不同类型用户的访问量。这里有三个关键词，第一个是“系统”，第二个是“一段时间”，第三个是“用户类型”。时间粒度好理解，就是以小时为单位去统计流量。用户类型指的是采用不同的维度来刻画用户，比如性别、年龄、教育程度、职业、地理位置。系统指的是流量来源，比如平台、站点、频道、媒体域名。

在系统和用户的维度组合之下，流量被细分为数以百万计的不同“种类”。比如，来自 XX 平台 XX 站点的在校大学生的访问量，或是来自 XX 媒体 XX 频道 25-45 岁女性的访问量等等。

我们知道，流量预测本身是个时序问题，它和股价预测类似，都是基于历史、去预测未来。在我们的案例中，为了预测上百万种不同的流量，咱们得先为每种流量生成时序序列，然后再把这些时序序列喂给机器学习算法进行模型训练。

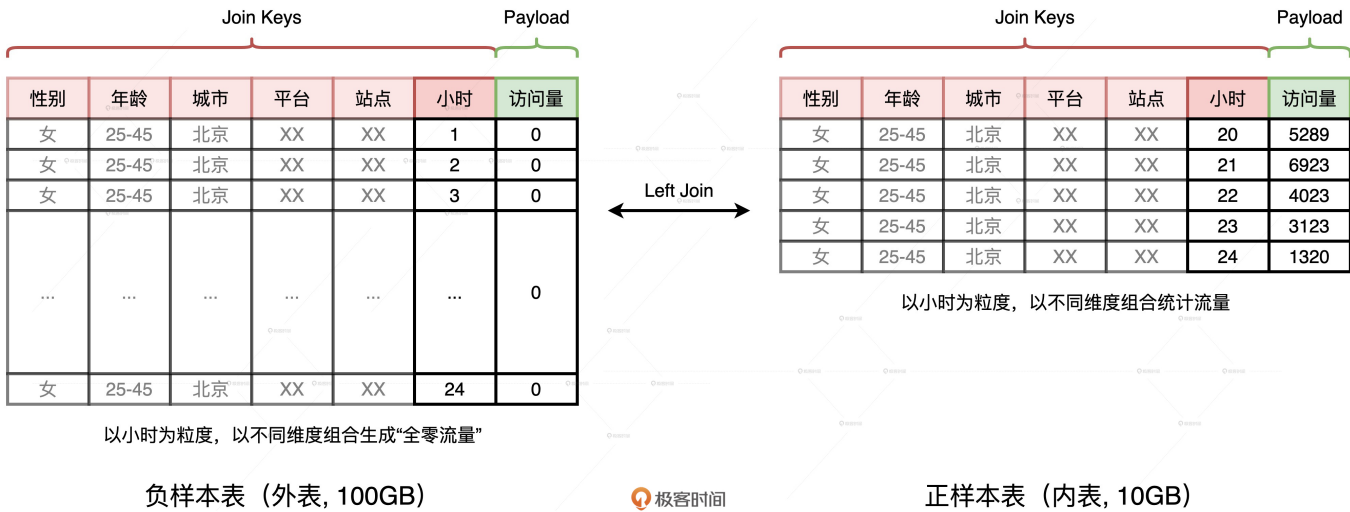
统计流量的数据源是线上的访问日志，它记录了哪类用户在什么时间访问了哪些站点。要知道，我们要构建的，是以小时为单位的时序序列，但由于流量的切割粒度非常细致，因此有些种类的流量不是每个小时都有访问量的，如下图所示。



某种流量在过去24小时的记录情况

我们可以看到，在过去的 24 小时中，某种流量仅在 20-24 点这 5 个时段有数据记录，其他时段无记录，也就是流量为零。在这种情况下，我们就需要用“零”去补齐缺失时段的序列值。那么我们该怎么补呢？

因为业务需求是填补缺失值，所以在实现层面，我们不妨先构建出完整的全零序列，然后以系统、用户和时间这些维度组合为粒度，用统计流量去替换全零序列中相应位置的“零流量”。这个思路描述起来比较复杂，用图来理解会更直观、更轻松一些。



首先，我们生成一张全零流量表，如图中左侧的“负样本表”所示。这张表的主键是划分流量种类的各种维度，如性别、年龄、平台、站点、小时时段等等。表的 Payload 只有一列，也即访问量，在生成“负样本表”的时候，这一列全部置零。

然后，我们以同样的维度组合统计日志中的访问量，就可以得到图中右侧的“正样本表”。不难发现，两张表的 Schema 完全一致，要想获得完整的时序序列，我们只需要把外表以“左连接（Left Outer Join）”的形式和内表做关联就好了。具体的查询语句如下：

复制代码

```
1 //左连接查询语句
2 select t1.gender, t1.age, t1.city, t1.platform, t1.site, t1.hour, coalesce(t2.
3 from t1 left join t2 on
4 t1.gender = t2.gender and
5 t1.age = t2.age and
6 t1.city = t2.city and
7 t1.platform = t2.platform and
```

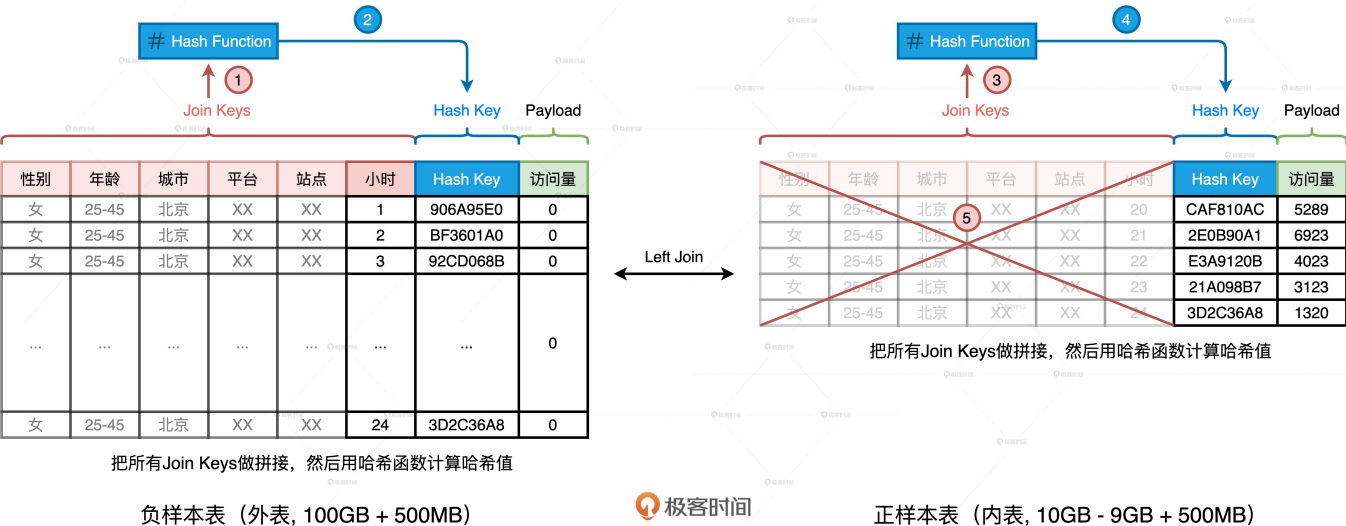
```
8 t1.site = t2.site and  
9 t1.hour = t2.hour
```

使用左连接的方式，我们刚好可以用内表中的访问量替换外表中的零流量，两表关联的结果正是我们想要的时序序列。“正样本表”来自访问日志，只包含那些存在流量的时段，而“负样本表”是生成表，它包含了所有的时段。因此，在数据体量上，负样本表远大于正样本表，这是一个典型的“大表 Join 小表”场景。尽管小表（10GB）与大表（100GB）相比，在尺寸上相差一个数量级，但两者的体量都不满足 BHJ 的先决条件。因此，Spark 只好退而求其次，选择 SMJ（Shuffle Sort Merge Join）的实现方式。

我们知道，SMJ 机制会引入 Shuffle，将上百 GB 的数据在全网分发可不是一个明智的选择。那么，根据“能省则省”的开发原则，我们有没有可能“省去”这里的 Shuffle 呢？要想省去 Shuffle，我们只有一个办法，就是把 SMJ 转化成 BHJ。你可能会说：“都说了好几遍了，小表的尺寸 10GB 远超广播阈值，我们还能怎么转化呢？”

办法总比困难多，我们先来反思，关联这两张表的目的是什么？目的是以维度组合（Join Keys）为基准，用内表的访问量替换掉外表的零值。那么，这两张表有哪些特点呢？**首先，两张表的 Schema 完全一致。其次，无论是在数量、还是尺寸上，两张表的 Join Keys 都远大于 Payload。**那么问题来了，要达到我们的目的，一定要用那么多、那么长的 Join Keys 做关联吗？

答案是否定的。在上一讲，我们介绍过 Hash Join 的实现原理，在 Build 阶段，Hash Join 使用哈希算法生成哈希表。在 Probe 阶段，哈希表一方面可以提供 O(1) 的查找效率，另一方面，在查找过程中，Hash Keys 之间的对比远比 Join Keys 之间的对比要高效得多。受此启发，我们为什么不能计算 Join Keys 的哈希值，然后把生成的哈希值当作是新的 Join Key 呢？



我们完全可以基于现有的 Join Keys 去生成一个全新的数据列，它可以叫 “Hash Key” 。生成的方法分两步：

把所有 Join Keys 拼接在一起，把性别、年龄、一直到小时拼接成一个字符串，如图中步骤 1、3 所示

使用哈希算法（如 MD5 或 SHA256）对拼接后的字符串做哈希运算，得到哈希值即为 “Hash Key” ，如上图步骤 2、4 所示

在两张表上，我们都进行这样的操作。如此一来，在做左连接的时候，为了把主键一致的记录关联在一起，我们不必再使用数量众多、冗长的原始 Join Keys，用单一的生成列 Hash Key 就可以了。相应地，SQL 查询语句也变成了如下的样子。

复制代码

```
1 //调整后的左连接查询语句
2 select t1.gender, t1.age, t1.city, t1.platform, t1.site, t1.hour, coalesce(t2.
3 from t1 left join t2 on
4 t1.hash_key = t2.hash_key
```

添加了这一列之后，我们就可以把内表，也就是“正样本表”中所有的 Join Keys 清除掉，大幅缩减内表的存储空间，上图中的步骤 5 演示了这个过程。当内表缩减到足以放进广播变量的时候，我们就可以把 SMJ 转化为 BHJ，从而把 SMJ 中的 Shuffle 环节彻底省掉。

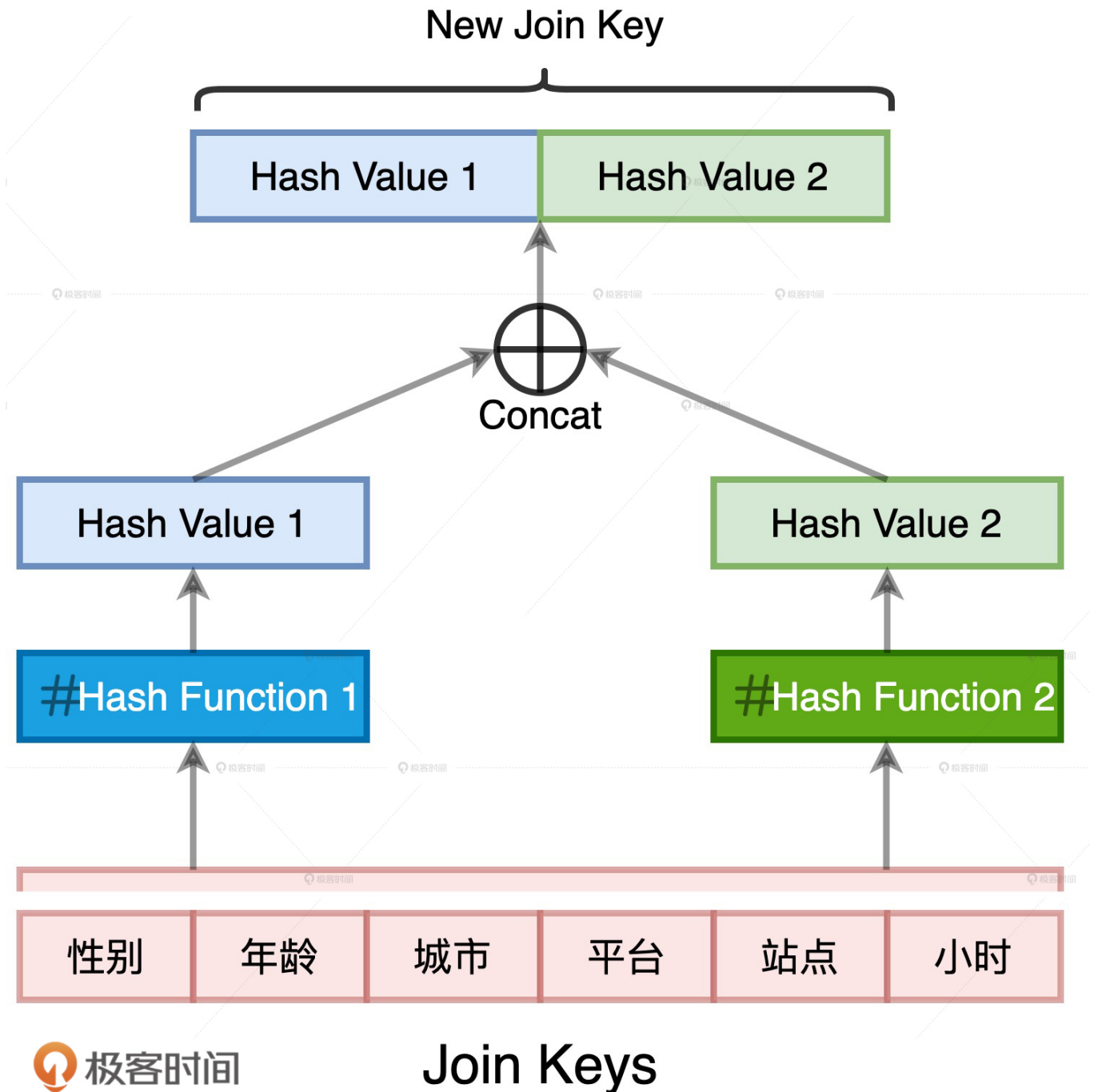
这样一来，清除掉 Join Keys 的内表的存储大小就变成了 1.5GB。对于这样的存储量级，我们完全可以使用配置项或是强制广播的方式，来完成 Shuffle Join 到 Broadcast Join 的

转化，具体的转化方法你可以参考广播变量那一讲（[🔗第 13 讲](#)）。

案例 1 说到这里，其实已经基本解决了，不过这里还有一个小细节需要我们特别注意。**案例 1 优化的关键在于，先用 Hash Key 取代 Join Keys，再清除内表冗余数据。Hash Key 实际上是 Join Keys 拼接之后的哈希值。既然存在哈希运算，我们就必须要考虑哈希冲突的问题。**

哈希冲突我们都很熟悉，它指的就是不同的数据源经过哈希运算之后，得到的哈希值相同。在案例 1 当中，如果我们为了优化引入的哈希计算出现了哈希冲突，就会破坏原有的关联关系。比如，本来两个不相等的 Join Keys，因为哈希值恰巧相同而被关联到了一起。显然，这不是我们想要的结果。


消除哈希冲突隐患的方法其实很多，比如“二次哈希”，也就是我们用两种哈希算法来生成 Hash Key 数据列。两条不同的数据记录在两种不同哈希算法运算下的结果完全相同，这种概率几乎为零。



案例 2：过滤条件的 Selectivity 较高


除了 Join Keys 远大于 Payload 的情况会导致我们无法选择 BHJ，还有一种情况是过滤条件的 Selectivity 较高。这个案例来源于电子商务场景，在星型（Star 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。订单表是事实表（Fact），而用户表是维度表（Dimension）。

这个案例的业务需求很简单，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。订单表 and 用户表的 Schema 如下表所示。

 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所示。

 复制代码

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

在这个案例中，事实表的存储容量在 TB 量级，维度表是 20GB 左右，也都超过了广播阈值。其实，这样的关联场景在电子商务、计算广告以及推荐搜索领域都很常见。

对于两张表都远超广播阈值的关联场景来说，如果我们不做任何调优的，Spark 就会选择 SMJ 策略计算。在 10 台 C5.4xlarge AWS EC2 的分布式环境中，SMJ 要花费将近 5 个小时才完成两张表的关联计算。这样的执行效率，我们肯定无法接受，但我们又能做哪些优化呢？你不妨先花上两分钟去想一想，然后再来一起跟我去分析。

仔细观察上面的查询语句，我们发现这是一个典型的星型查询，也就是事实表与维度表关联，且维表带过滤条件。维表上的过滤条件是 `users.type = 'Head User'`，即只选取头部用户。而通常来说，相比普通用户，头部用户的占比很低。换句话说，这个过滤条件的选择性（Selectivity）很高，它可以帮助你过滤掉大部分的维表数据。在我们的案例中，由于头部用户占比不超过千分之一，因此过滤后的维表尺寸很小，放进广播变量绰绰有余。

这个时候我们就要用到 AQE 了，我们知道 AQE 允许 Spark SQL 在运行时动态地调整 Join 策略。我们刚好可以利用这个特性，把最初制定的 SMJ 策略转化为 BHJ 策略（千万别忘了，AQE 默认是关闭的，要想利用它提供的特性，我们得先把 `spark.sql.adaptive.enabled` 配置项打开）。

不过，即便过滤条件的选择性很高，在千分之一左右，过滤之后的维表还是有 20MB 大小，这个尺寸还是超过了默认值广播阈值 10MB。因此，我们还需要把广播阈值 `spark.sql.autoBroadcastJoinThreshold` 调高一些，比如 1GB，AQE 才会把 SMJ 降级为 BHJ。做了这些调优之后，在同样的集群规模下，作业的端到端执行时间从之前的 5 个小时缩减为 30 分钟。

让作业的执行性能提升了一个数量级之后，我们的调优就结束了吗？在调优的本质那一讲，我们一再强调，随着木桶短板的此消彼长，调优是一个不断持续的过程。在这个过程中，我们需要因循瓶颈的变化，动态地切换调优方法，去追求一种所有木板齐平、没有瓶颈的状态。

那么，当我们用动态 Join 策略，把 SMJ 策略中 Shuffle 引入的海量数据分发这块短板补齐之后，还有没有“新晋”的短板需要修理呢？

对于案例中的这种星型关联，我们还可以利用 DPP 机制来减少事实表的扫描量，进一步减少 I/O 开销、提升性能。和 AQE 不同，DPP 并不需要开发者特别设置些什么，只要满足条件，DPP 机制会自动触发。

但是想要使用 DPP 做优化，还有 3 个先决条件需要满足：

DPP 仅支持等值 Joins，不支持大于或者小于这种不等值关联关系


维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`

事实表必须是分区表，且分区字段（可以是多个）必须包含 Join Key

我们可以直接判断出查询满足前两个条件，满足第一个条件很好理解。满足第二个条件是因为，经过第一步 AQE 的优化之后，广播阈值足够大，足以容纳过滤之后的维表。那么，要想利用 DPP 机制，我们必须要让 orders 成为分区表，也就是做两件事情：

创建一张新的订单表 `orders_new`，并指定 `userId` 为分区键

把原订单表 `orders` 的全部数据，灌进这张新的订单表 `orders_new`

 复制代码

```
1 //查询语句
2 select (orders_new.price * orders_new.quantity) as revenue, users.name
3 from orders_new inner join users on orders_new.userId = users.id
4 where users.type = 'Head User'
5 group by users.name
6 order by revenue desc
```

用 `orders_new` 表替换 `orders` 表之后，在同样的分布式环境下，查询时间就从 30 分钟进一步缩短到了 15 分钟。


你可能会说：“为了利用 DPP，重新建表、灌表，这样也需要花费不少时间啊！这不是相当于把运行时间从查询转嫁到建表、灌数了吗？”你说的没错，确实是这么回事。如果为了查询效果，临时再去修改表结构、迁移数据确实划不来，属于“临时抱佛脚”。因此，为了最大限度地利用 DPP，在做数仓规划的时候，开发者就应该结合常用查询与典型场景，提前做好表结构的设计，这至少包括 Schema、分区键、存储格式等等。

案例 3：小表数据分布均匀

在上面的两个案例中，我们都是遵循“能省则省”的开发原则，想方设法地把 Shuffle Joins 切换为 Broadcast Joins，从而消除 Shuffle。但是，总有那么一些“顽固”的场景，无论我们怎么优化，也没办法做到这一点。那么对于这些“顽固分子”，我们该怎么办呢？


我们知道，如果关联场景不满足 BHJ 条件，Spark SQL 会优先选择 SMJ 策略完成关联计算。但是在上一讲我们说到，**当参与 Join 的两张表尺寸相差悬殊且小表数据分布均匀的时候，SHJ 往往比 SMJ 的执行效率更高**。原因很简单，小表构建哈希表的开销要小于两张表排序的开销。

我们还是以上一个案例的查询为例，不过呢，这次我们把维表的过滤条件去掉，去统计所有用户贡献的营业额。在 10 台 C5.4xlarge AWS EC2 的分布式环境中，去掉过滤条件的 SMJ 花费了将近 7 个小时才完成两张表的关联计算。

 复制代码

```
1 //查询语句
2 select (orders.price * order.quantity) as revenue, users.name
3 from orders inner join users on orders.userId = users.id
4 group by users.name
5 order by revenue desc
```

由于维表的查询条件不复存在，因此案例 2 中的两个优化方法，也就是 AQE Join 策略调整和 DPP 机制，也都失去了生效的前提。**这种情况下，我们不妨使用 Join Hints 来强制 Spark SQL 去选择 SHJ 策略进行关联计算**，调整后的查询语句如下表所示。

 复制代码

```
1 //添加Join hints之后的查询语句
2 select /*+ shuffle_hash(orders) */ (orders.price * order.quantity) as revenue,
3 from orders inner join users on orders.userId = users.id
4 group by users.name
5 order by revenue desc
```

将 Join 策略调整为 SHJ 之后，在同样的集群规模下，作业的端到端执行时间从之前的 7 个小时缩减到 5 个小时，相比调优前，我们获得了将近 30% 的性能提升。

需要注意的是，**SHJ 要想成功地完成计算、不抛 OOM 异常，需要保证小表的每个数据分片都能放进内存**。这也是为什么，我们要求小表的数据分布必须是均匀的。如果小表存在数据倾斜的问题，那么倾斜分区的 OOM 将会是一个大概率事件，SHJ 的计算也会因此而中断。

小结

今天这一讲，我们从 3 个案例出发，探讨并解锁了不同场景下“大表 Join 小表”的优化思路和应对方法。

首先，我们定义了什么是“大表 Join 小表”。一般来说，参与 Join 的两张表在尺寸上相差 3 倍以上，就可以看作是“大表 Join 小表”的计算场景。

其次，我们讲了 3 个大表 Join 小表场景下，无法选择 BHJ 的案例。

第一个案例是 Join Keys 远大于 Payload 的数据关联，我们可以使用映射方法（如哈希运算），用较短的字符串来替换超长的 Join Keys，从而大幅缩减小表的存储空间。如果缩减后的小表，足以放进广播变量，我们就可以将 SMJ 转换为 BHJ，来消除繁重的 Shuffle 计算。需要注意的是，映射方法要能够有效地避免“映射冲突”的问题，避免出现不同的 Join Keys 被映射成同一个数值。

第二个案例是，如果小表携带过滤条件，且过滤条件的选择性很高，我们可以通过开启 AQE 的 Join 策略调整特性，在运行时把 SMJ 转换为 BHJ，从而大幅提升执行性能。这里有两点需要我们特别注意：一是，为了能够成功完成转换，我们需要确保过滤之后的维表尺寸小于广播阈值；二是，如果大表本身是按照 Join Keys 进行分区的话，那么我们还可以充分利用 DPP 机制，来进一步缩减大表扫描的 I/O 开销，从而提升性能。

第三个案例是，如果小表不带过滤条件，且尺寸远超广播阈值。如果小表本身的数据分布比较均匀，我们可以考虑使用 Join hints 强行要求 Spark SQL 在运行时选择 SHJ 关联策略。一般来说，在“大表 Join 小表”的场景中，相比 SMJ，SHJ 的执行效率会更好一些。背后的原因在于，小表构建哈希表的开销，要小于两张表排序的开销。

每日一练

1. 对于案例 1，我们的核心思路是用哈希值来替代超长的 Join Keys，除了用哈希值以外，你觉得还有其他的思路或是办法，去用较短的字符串来取代超长的 Join Keys 吗？
2. 对于案例 2，利用 AQE Join 策略调整和 DDP 机制的关键，是确保过滤后的维表小于广播阈值。你能说说，都有哪些方法可以用来计算过滤后的维表大小吗？
3. 对于案例 3，假设 20GB 的小表存在数据倾斜，强行把 SMJ 转化为 SHJ 会抛 OOM 异常。这个时候，你认为还有可能继续优化吗？

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 26 | Join Hints指南：不同场景下，如何选择Join策略？

下一篇 28 | 大表Join大表（一）：什么是“分而治之”的调优思路？

精选留言 (4)

 写留言



zxk 

2021-05-16

老师我想请问下，在第二个案例中，如果我将 join 条件写成 `on orders.userId = users.id and users.type = 'Head User'`，是否能实现维度表提前过滤并达到 Broadcast 的要求？

展开 

作者回复: 不可以哈，Spark SQL目前还没有那么智能，你这么写的效果，和单独把`users.type = 'Head User'`当作过滤条件的效果是一样的，都是通过Spark SQL AQE来动态触发Join策略调整，还做不到在静态优化阶段就提前决定在运行时把Shuffle Joins转化为Broadcast Joins。



 1



Fendora范东_

2021-05-15

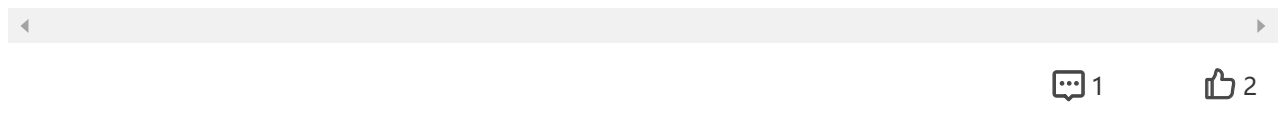
问题回答

- 1.给join keys每个字段维护一个字典，每个字段值在字典内对应一个唯一的整数。拿到每个字段指定的种整数，然后组装起来，作为新的join key。
- 2.把维度表查询sql单独拿出来，缓存其df，查看其屋里计划，可以知道它结果集大小。
- 3.参考AQE的自动倾斜处理特性，把数据倾斜的分区拆分开，然后再进行SHJ

展开 ∨

作者回复: 第一题超赞~ 这个方法非常好，实际上我觉得比较hash的方法还棒！因为它天然能避开冲突的问题，而且存储效率很高，机智如你~ 真的特别好的方法，其实本质上这可以看成是一种编码方式了，老弟你也搞机器学习吗？编码的思路在Machine Learning用的非常普遍~ 当然这里有一个字典维护的成本，不过如果Join Keys的cardinality不是很夸张的话，其实还好~

2、3的思路都没问题~



CycleGAN

2021-05-22

我们生产端用的2.4也没有AQE机制，但是我觉得减小维度表的大小也能模拟出来，要在维度表上做设计，最有效的还是分区做好，比如我们按时间分区，每个分区的数据局小很多，然后摘出来需要的列，列上的长str做编码，再映射，列上的长join key做编码等。感觉把维表做小还是值得的，性价比很高。

然后分析大小的话，一是看cache后看大小，主要还是ui页面里的sql计划里看是不是Bro...

展开 ∨



辰

2021-05-19

老师，这些都是基于spark3.0的新特性，那对于3.0之前的版本又该怎么解决呢，毕竟3.0版本是新出的，对于大多数公司不一定使用了该版本

展开 ∨

作者回复: 咱们举了3个案例，其中第二个案例需要用到3.0的AQE，具体来说是Join策略调整；第三个案例是用join hints把Shuffle SMJ转化为Shuffle HJ。这两个case利用的新特性，确实是3.0才支持。

不过第一个，也就是Join Keys比Payload大很多，这个技巧并不需要3.0哈~

