Kylin性能调优记——业务技术两手抓

背景

最近开始使用了新版本的Kylin，在此之前对于新版本的了解只是代码实现和一些简单的新功能测试，但是并没有导入实际场景的数据做分析和查询，线上Hadoop稳定之后，逐渐得将一些老需求往新的环境迁移，基于以前的调研，新版本（V2，版本为1.5.2）的Kylin提供了几个比较显著的功能和优化：

- 新的度量类型，包括TOPN、基于bitmap的精确distinct count和RAW。
- 自定义度量框架，用户可以定义一些特殊的度量需求。
- Fast Cubing算法，减少MR任务，提升build性能。
- 查询优化，Endpoint Coprocessor，并行scan和数据压缩，这部分对于查询性能提升还是比较显著的。
- shard hbase存储，基于一致性哈希的rawkey分布，尽可能的将大的cuboid分散到不同的region上，增加并行扫描度。
- spark计算引擎，in memory准实时计算，这两项目前还处于试验阶段。
- 新的aggregation group分区算法。

有了这么多新的特性和性能提升，经常拿新版本来应付用户的需求：新版本实现了xxx功能，肯定性能会很好，等到新版本稳定了再来搞这个需求吧。等到我们的新版本上线了，业务需求真正上来之后，才发现用起来没有相当的那么简单，对于Kylin这种直接面向用户需求的服务，对于一些特殊的需求更是要不断打磨和调整才能达到更好的性能。本文整理在接入云音乐一个较为复杂的需求的实现和中间调优的过程，一方面开拓一下自己的思路，另外也使得读者能改更好的使用Kylin。

业务需求

数据通过日志导入到Hive中查询，经过一定的聚合运算，整理成如图1中的格式。这个数据源是针对歌曲统计的，每一首歌曲包含歌曲属性信息（歌曲名、歌手ID，专辑ID等）、支付方式、所属圈子ID、标签信息等，需要统计的指标包括每一首歌曲的各种操作的PV和UV，操作包括播放、收藏、下载等10种。因此一共20左右个指标，然后需要按照这20个指标的任意一个指标计算TOP N歌曲和其他统计信息（N可能是1000,1W等），除此之外，在计算TOP N的时候还需要支持字段的过滤，可以执行过滤的字段包括支付方式、圈子ID、标签等。歌曲ID全部的成员数大概在千万级别，该表的数据每天导入到hive，经过初步的聚合计算生成图1中的格式，基本上保证了每天的每一个用户对于每一首歌曲只保留一条记录，每天的记录数大概在几亿条（聚合之后），查询通常只需要查询不同条件下的TOPN的song_id。查询性能尽可能的高（秒级），需要提供给分析人员使用。

基于kylin 1.x版本的实现

Kylin 1.3.0之前的版本对于这种需求的确有点无能为力，我们使用了最简单的方式实现，创建一个包含所有维度、所有度量的cube，其中10个度量为count distinct，使用hyperloglog实现的近似去重计数算法，这种做法能够可以说是最原始的，每次查询时如果不进行任何过滤则需要从hbase中扫描几百万条记录才能满足TOPN的查询，每一条记录还都包含hyperloglog序列化的值（读取到内存之后还需要对hyperloglog进行反序列化），TOP的查询速度可想而知，正常情况（没有并发）下能够在1、2分钟出结果，偶尔还会有一个查询把服务搞挂（顺便吐槽一下我们使用对的虚拟机）。查询以天的数据就这种情况了，对于跨天的数据更不敢尝试了。

这种状况肯定无法满足需求方，只能等着新版本落地之后使用TOPN度量。

需求简化

面对这样的现实，感觉对于去重计数无能为力了，尤其像这种包含10个distinct count度量的cube，需求方也意识到这种困境，对需求做出了一定的让步：不需要跨天计算TOPN的song_id了，只需要根据每一天的PV或者UV值计算TOPN，这样就不需要distinct count了，而且还能保证每天的统计值是精确的。如果希望计算大的时间周期的TOPN，则通过创建一个新的cube实现每个自然周、自然月的统计。对于这样的需求简化，可以大大提升cube

的创建，首先需要对数据源做一些改变（这种改变通常是view来完成），将原始表按照歌曲ID、用户ID和其它的维度进行聚合（group by，保证转换后的新表每一个用户对于每一首歌曲只保存一条记录），PV则直接根据计算SUM（操作次数），UV通过表达式if(SUM(操作次数) > 0, 1, 0)确定每一个userid对每一个songid在当天是否进行了某种操作，这样我们就得到如图2中格式的表：

歌曲ID	用户ID	歌曲属性	支付类型	圈子ID	标签	播放次数	收藏次数	下载次数	xxx次数	是否播放	是否收藏	是否下载	是否xxx
1234	1356	...	0	135	流行	3	1	1	...	1	1	1	...
2345	1356	...	1	135	摇滚	4	2	0	...	1	1	0	...
2345	2452	...	1	135	摇滚	2	0	1	...	1	0	1	...
5678	9999	...	2	140	流行	5	0	0	...	1	0	0	...

这个表中可以保证歌曲ID+用户ID是唯一的（其它的维度是歌曲的属性，一个歌曲只对应相同的属性），对于Kylin来说原始数据量也变小了（还是每天几亿条记录），接下来新建的Cube由于数据源保证每一条记录不是同一个用户对同一首歌曲的操作，因此所有的度量都可以用SUM来实现（去重计数也可以通过对sum(是否xxx)统计精确的计数值），这样对于Kylin的负担减少了一大部分，但是扫描记录数还是这么多（Cube中最大的维度为歌曲，每次查询都需要携带，查询时扫描记录数始终保持在歌曲ID的个数），但是Build性能提升了，查询性能也由于扫描数据量减少（一个SUM度量占用的存储空间只有8字节）而提升，但是还是需要30+秒。

此时，新版本终于在公司内部落地了，仿佛看到了新的曙光。

Kylin新版本（2.x）的实现

新版本有了TOPN度量了，但是目前只支持最大TOP 1000，可以通过直接改cube的json定义改变成最大得到5000，于是尝试了几种使用TOPN的方案。

第一种是将所有维度放在一个Cube中然后对于每一个PV和UV创建一个TOPN度量。这样build速度还是挺快的，但是查询时大约需要25秒+，最坑爹的是，创建TOPN时需要指定一个指标列（做SUM，根据该列的SUM值排序）和一个聚合列（歌曲ID），使用一个聚合列创建的多个TOPN度量居然只有查询第一个度量时才有结果，使用其他指标查询TOPN时返回的聚合值总是null，难道我要为每一个指标都建一个cube，每一个只包含一个topn度量？

第二种方案是不使用全部维度，而只使用在计算TOPN时需要做过滤的列，例如支付类型、圈子ID等维度建包含TOPN的Cube，这样通过抽取部分维度创建一个小Cube用来计算TOPN，然后再使用计算出来的一批批歌曲ID从大Cube（包含全部维度的cube）中取出这个ID对应的其它指标和属性信息。这样做的弊端是对于查询不友好，查询时需要执行两条SQL并且在前端进行分页，如果性能可以接受也就认了，但是现实总是那么残酷，当创建了一个这样的Cube（包含歌曲ID和其他几个成员值很少的维度）进行build数据时，让人意想不到的事情发生了，计算第二层Cuboid的任务居然跑了3个小时才勉强跑完，接下来的任务就更不敢想象了，MR任务的task使用CPU总是100%，通过单机的测试发现多个TOPN的值进行merge的性能非常差，尤其是N比较大的时候（而且Kylin内部会把N放大50倍以减少误差），看来这个方案连测试查询性能的机会都没有了。

尝试了这两个方案之后我开始慢慢的失去信心了，但是想到了Kylin新版本的并行扫描会对性能有所提升，于是就创建了一个小cube（包含歌曲ID和几个过滤维度，和全部的SUM度量）用于计算TOPN，整个cube计算一天的数据只需要不到1G的存储，想必扫描性能会更好吧，查询测试发现在不加任何过滤条件的情况下对任意指标的TOPN查询大约不到15s，但是扫描的记录数仍然是几百W（歌曲ID的数目），可见并行扫描带来的性能提升，但是扫描记录数仍然是一个绕不过去的坎。

在经历这么多打击之后，实在有点不知所措了，感觉歌曲ID的数目始终是限制查询性能的最大障碍，毕竟要想计算TOPN肯定要查看每一个歌曲ID的统计值，然后再排序，避免不了这么多的扫描，唯一的方法也就是减小region的大小，使得一天的数据分布在更多的region中，增大并行度，但这也不是最终的解决方案啊，怎么减小扫描记录数呢？

从业务出发

陷入了上面的思考之后，提升查询性能只有通过减少扫描记录数实现，而记录数最少也要等于歌曲ID的成员数，是否可以减少歌曲ID的成员数呢？需求方的一个提醒启发了我们，全部的20个指标最关键的是播放次数，一般播放次数在TOPN的歌曲，也很有可能在其他指标的TOPN中。那么是否可以通过去除长尾数据来减少歌曲ID的个数呢？首先查了一下每天播放次数大于50的个数数量，查询结果对于没接触过业务的我来说还是很吃惊的，居然还剩几十W，也就意味着通过排除每天播放次数小于50的歌曲，可以减少将近80%+的歌曲ID，而这些歌曲ID对于任何指标的TOPN查询都是没有任何作用的，于是通过view把数据源进行在过滤和聚合，创建包含全部维度的cube，查询速度果然杠杠的，只需要2s左右了！毕竟只需要扫描的记录数只有几十万，并且都是SUM的度量。

终于达成目标了，但是这种查询其实是对原始数据的过滤，万一需求方需要查询这些播放次数比较少的歌曲ID的其他指标呢？完全不能从Kylin中获取啊。此时，业务方的哥们根据数据特性想到了一个很好的办法：为播放次数建索引！根据播放次数的数值创建一个新的维度，这个维度的值等于播放次数所在的区间，例如播放次数是几位数这个维度的值就是多少，或者使用case when sum(播放次数)自定义不同的范围。例如某一个歌曲在当天SUM(播放次数)为千万级，那么这个歌曲对应的index维度值为8，百万级则值为7，以此类推。然后查询的时候根据index过滤确定播放次数的范围，这样可以大大减少歌曲ID的数目，进而减少扫描记录数。

此时需要对图2中的记录格式再进行整理，通过group by 歌曲ID,xxx(其他维度)统计出每一首歌曲的操作次数和操作人数，然后再根据操作次数的值确定index的值。得到新的记录格式如图3.

说干就干，整理完数据之后再添加一个index维度创建新的Cube，查询的时候可以先查询select 歌曲ID, sum(指标) as c from table where index >= 7 group by 歌曲ID order by c desc limit 100，使用着这种查询可以直接过滤掉所有播放次数小于百万的歌曲ID，但是经过测试发现，这种查询时间还是15s左右，和上面的方案性能并无改善。整得我们开始怀疑人生了。

维度						度量							
歌曲ID	index	歌曲属性	支付类型	圈子ID	标签	播放次数	收藏次数	下载次数	XXX次数	播放人数	收藏人数	下载人数	Xxx人数
1234	7	...	0	135	流行	3000010	134343	14334	...	123443	13432	13222	...
2345	4	...	1	135	摇滚	1028	334	123	...	134	34	138	...
5678	5	...	2	140	流行	58374	3723	984	...	8742	844	674	...

回归技术

接下来感觉是走投无路之后的各种猜测，例如猜测shard by设置true or false是否对性能有很大的影响，经过查看源码发现shard by只是为了决定该列的值是否参与一致性哈希中桶号的计算，只不过是为了更好的打散rowkey的分布，对于存储和查询性能提升不是太明显，思来想去还是不合理，既然加了过滤不应该还要扫描全部的歌曲ID，是不是Cube中rowkey的顺序错了，查了一下cube的定义，发现index维度被放在rowkey的最后一行，这显然是不了解Kylin是如何确定扫描范围的！Kylin存储在hbase中的记录是rowkey是根据定义Cube时确定的rowkey顺序确定的，把如果查询的时候对某一个维度进行范围过滤，Kylin则会根据过滤的范围确定扫描的区间以减小扫描记录数，如果把这个维度放到rowkey的最后，则将这种过滤带来的减小扫描区间的作用降到了最低。

重新修改Cube把index维度在rowkey中的位置提升到最前面之后重新build，此后再执行相同的查询速度有了质的飞跃，降低到了1s以内。根据index的范围不同扫描记录数也随着改变，但是对于查询TOPN一般只需要使用index >= 6的过滤条件，这种条件下扫描记录数可以降低到几万，并且保证了全部歌曲ID可以被查询。

总结

本文记录了云音乐的一个比较常见的TOPN的需求的优化，在整个优化的过程中在技术和业务上都给与了充分的考虑，在满足查询需求的前提下，查询性能从最初的上百秒提升到1秒以内，这个过程让我充分意识到了解数据特征和业务的重要性，以前遇到问题总是想着是否能够进行算法的优化，是否可以加机器解决，但是当这些都不能再有改进的空间时，转换一下思路，思考一下是否可以从业务数据入手进行优化，可能会取得更好的效果。

随着Kylin 2.x的落地使用，对于业务的支持也更有信心了，尤其是并行查询和Endpoint Coprocessor对性能的提升，但是对于开源技术来说，如何正确的使用才是最困难的问题，我们也会在探索Kylin使用的路上越走越远。

最后，通过本次调优，我发现了一个以前没有注意到的问题：薛之谦最近真的挺火，不相信的话可以查看一下网易云音乐热歌榜。