改造mondrian的构想

Mondrian简介

项目中最近需要使用mondrian，于是开始首先阅读以下mondrian的源代码，总体看来mondrian是一个ROLAP引擎，但是它的定位还是一个以库形式 提供的组件，而不是作为服务，例如它将所有的缓存都放在进程内部，当我在一个进程中创建多个cube并且一直保持着他们的connection会导致OOM，所以对mondrian作为服务化的首要改造还是在于如何将mondrian的缓存挪出本进程，放到第三方的存储系统例如redis、hbase等NOSQL存储系统中，这样一方面可以解决上面提到的进程内缓存问题，另一方面可以将数据独立出去可以将服务器做成无状态的，可以水平扩展，保证高可用性。除了缓存这一步之外，另外需要对mondrian做优化的地方还有生成SQL这一部分，mondrian生成的SQL主要包括查看一个level下所有的member和查看一个cell结果这两类，前者无非就是select distinct xxx或者select xxx group by xxx，mondrian一般使用select distinct的方式，这种没有什么可以优化的地方，另外还有一种查询是查询多个维度聚合之后的某一个度量的值，一般是多个维度group by之后计算某一列的sum、avg等值，但是如果在group by之前需要过滤掉一些元素的话，mondrian会将剩余的语句翻译成where xxx IN (xxx1, xxx2,...)（通过执行日志可以看到），这种SQL执行效率是比较低的，即使xxx这一列有主键，改造的方案是尽可能得将IN语句改成多个between或者大于小于这样的SQL，这个的优先级可以低一些。

改进方案

上面提到的是对mondrian本身不足需要进行改造的地方，但是像mondrian这样的一个ROLAP系统的执行性能的确不怎么让人满意，业界做的一些OLAP引擎（例如百度的palo、ebay的kylin等）总是生成可以再查询速度秒级，而mondrian的缓存很渣，每次如果再去数据源执行SQL又会很慢，因此我们对这个性能还需要特别注意，polo自己做了数据仓库，所有需要查询的数据都由该系统自己维护，因此做性能优化也是很容易的，而mondrian使用的数据在数据源（关系型数据库或者大数据系统中）存储，我们并不能改变它存储的形式，因此大部分的性能还是取决于数据源的性能（是否加主键，是否使用SSD存储，hadoop集群节点数等等），基于这个限制我觉得如果想要对性能有质的提升也只有一条路，缓存，缓存，缓存，重要的事情说三遍，但是毕竟缓存一个cube的缓存量还是太大了（下面粗略的计算一下一个cube需要的缓存量），其实如果我们有一个足够大的缓存，我们理论上也就是自己做了数据仓库，这方面还是可以考虑的，既然自己可以使用缓存作为数据仓库，那么就需要做类似ETL操作，预先将用户的数据加载到缓存中，也就是所谓的cube预计算，但是和真正的数据仓库不一样的地方在于，数据仓库存储的全局的需要分析的数据，而预计算的只是针对一个cube所能够使用的数据，目前来看，Kylin的预计算方式是值得我们学习的，但是Kylin主要解决的问题还是基于hadoop系统下的数据分析问题（数据使用hadoop存储，使用hive查询的），由于hive的速度实在是让人难以接受（尤其是负载的OLAP查询产生的SQL），因此做预计算并且将结果缓存到hbase是非常有必要的，但是既然我们使用mondrian这样的引擎就是希望我们的数据源既可以是关系数据库还可以是hive之类的数据库，甚至使用hbase（phoenix提供SQL接口），这里的预计算对于关系数据库是否必要就值得考虑一下了。

既然做了预计算，也就是做缓存，无法避免的问题就是缓存一致性问题了，如何保证在数据量更新的情况下cube的缓存数据也能更新到最新的状态，当然对于OLAP系统来说数据的实时性不是最重要的，一般情况下几个小时更新一次数据或者一天更新一次数据也是可以接受的，而更新数据源之后也不要求cube的数据能够立即更新，能够达到最终一致性的要求也是可以接受的。但是当数据源导入新的数据的时候可能导致整个cube的值都发生改变，如果快速的计算新的cell值也是需要解决的问题，当前看到的一些系统都是用了倒排索引（还不了解）的方式来实现增量计算，具体的实现可以参考一下Kylin的代码。

Mondrian和Kylin

接下来的任务和方向主要是熟悉mondrian和kylin两个系统的实现，能够基于他们实现出我们自己的面向mdx查询和sql查询的OLAP引擎（目前主要的方向还是mdx），但是mondrian对于mdx的支持还是有一定的缺陷的，最主要的问题就是它目前不支持子查询（难道考虑到中间结果过大问题），因此一些高级的过滤操作目前还是实现不了，例如取出按照年份聚合之后总销量小于1W的所有年份，然后对过滤之后的数据按照其他维度进行mdx查询，这其实相当于对数据源进行一定的过滤，首先使用mdx查出需要过滤的年份，然后在数据源中删除所有年份等于这些的数据（相当于每次查询在源事实表的基础上加了一个where条件），如果在外层实现这样的过滤又太过麻烦，因此对于这样的功能性的缺失目前还不知道如何解决。另外mondrian在定义事实表的时候不仅仅可以支持指定一个table，还可以指定任意的视图，这也就以为这任意的SQL语句的结果都可以作为事实表，使用这种方式可以满足上面的需求但是对于每次查询翻译成的SQL都需要多层的子查询，性能可想而知了。后期我们如果能够缓存整个cube的时候可以直接才cube的基础上满足这样的需求。

另外，mondrian里面的维度是具有层级关系的，当我们定义了一个月级别，它的父级别是年，那么月级别的成员需要携带上具体的年份，这时候如果只按照月份进行聚合统计的时候需要得到的结果是[1997].[1]，[1998].[1]这样的值，而不是[1],[2]这样的结果，如果实现这样的结果还需要将每一个维度的层级进行拆分，这样无疑是非常复杂的，不知道mdx里面有没有什么关键字可以在指定一个level的是打破层架关系。

接下来就要深入看一下mondrian的代码了，其实mondrian的主要接口也就两个，首先是创建connection，其中包括加载cube的过程，另外就是执行一次mdx查询，查询的流程相对比较复杂，涉及到缓存以及如何生成SQL，另外对于mondrian当前缓存的结构以及对缓存的管理也需要重点关注。

空间估算

最后粗略的算一下一个cube的大小，假设场景是这样的一个星型结构，包括一个事实表和3个维度表，分别是时间，地区，产品信息，其中时间维

度分为三个层级，分别是年份，季度和月份，假设年份有10个成员，季度有`4*10`个成员，月份有`10*4*3`个成员，地区有三个级别，分别是国家，省份和城市三个级别，国家有100个成员，省份一共有1000个成员，城市有5000个成员；产品维度包含两个层级，分别为产品分类和产品商标，前者有16个成员，后者一共有500个成员，那么整个cube就是所有维度集合可能的聚合值，其中每一个维度包含一个特殊的成员ALL成员，这个成员属于一个特殊的层级ALL层级（按照mondrian的思想），每一个维度下的ALL层级是最高的层级，时间维度的深度为4，地区维度的深度为4，产品维度的深度为3，那么我们从最底层的层级进行组合，一共包含时间维度的最底层级别包含120个成员，最底层包含5000个成员，产品维度最底层级别包含500个成员，那么可能的组合值就是`120*5000*500`=3亿个组合元素，这是最底层的组合元素，这些所有的组合可以看成一个长为120，宽为500，高为5000的立方体，这3亿个元素就是整个立方体，这个立方体中每一个单元里面包含每一个组合（月=xxx，城市=xxx和产品商标=xxx）的聚合值，一个cell包含所有的度量值，这样整个立方体就建立起来了，这个立方体是全量的值，其他的组合值可以都可以通过将该立方体的某一个子立方作为一个cell进行计算，例如我们要计算年份为1997，国家为中国，产品类型为食品的销售总额，那么就相当于将1997年下的所有月份（12个）、中国的所有城市（假设100个）和产品类型为食品的所有商标（假设为50个）所有组合的聚合值，也就是`12*100*50`=60W个 cell进行组合成的新的cell作为返回的结果。如果我们想要缓存整个cube，最好的情况下我们还是需要缓存所有最低级别所有成员的组合（因为从高层级得不到低层级的信息，除非从数据源获取），这个代价还是相当大的，一般情况下我们需要对成员进行建索引（为每一个成员制定下标），然后通过下标的组合作为key，度量值 的组合作为value进行缓存，但是当数据源哪怕插入一条数据都会改变整个cube中的大量的cell值，这个增量计算还是相当可怕的。

在不考虑增量计算的情况下，其实这个还是缓存量和查询速度的博弈，如果缓存量不够，势必会缓存一些高层级，这样对于底层级的查询就需要走数据源，性能就差，如果查询缓存底层级成员的组合那么所有的查询都不需要再走数据源，而是直接在内存中计算。当然最好的办法还是需要判断一下最常用的层级和每一个层级的成员个数，如果某一层级的成员个数过多则不适合缓存，如果查询频率比较高的层级则更适合缓存。

总结

任务艰巨啊，还是先一步一个脚印的走吧，首先第一步将mondrian的缓存移出程序并且考虑一下缓存结构是否有比较再进行优化，第二步可以分析一下mondrian的执行SQL，看一下有没有优化的空间，第三步是进行预计算保存在缓存中，当然这时候暂时不考虑到动态的增量更新，最后再考虑如何做到增量计算。