Kylin Cube构建的流程

前言

在使用Kylin的时候，最重要的一步就是创建cube的模型定义，即指定度量和维度以及一些附加信息，然后对cube进行build，当然我们也可以根据原始表中的某一个string字段（这个字段的格式必须是日期格式，表示日期的含义）设定分区字段，这样一个cube就可以进行多次build，每一次的build会生成一个segment，每一个segment对应着一个时间区间的cube，这些segment的时间区间是连续并且不重合的，对于拥有多个segment的cube可以执行merge，相当于将一个时间区间内部的segment合并成一个。下面从源码开始分析cube的build和merge过程。本文基于Kylin-1.0-incubating版本，对于Kylin的介绍可以参见：http://blog.csdn.net/yu616568/article/details/48103415

入口介绍

在kylin的web页面上创建完成一个cube之后可以点击action下拉框执行build或者merge操作，这两个操作都会调用cube的rebuild接口，调用的参数包括：1、cube名，用于唯一标识一个cube，在当前的kylin版本中cube名是全局唯一的，而不是每一个project下唯一的；2、本次构建的startTime和endTime，这两个时间区间标识本次构建的segment的数据源只选择这个时间范围内的数据；对于BUILD操作而言，startTime是不需要的，因为它总是会选择最后一个segment的结束时间作为当前segment的起始时间。3、buildType标识着操作的类型，可以是"BUILD"、"MERGE"和"REFRESH"。

这些操作的统一入口就是JobService.submitJob函数，该函数首先取出该cube所有关联的构建cube的job，并且判断这些job是否有处于READY、RUNNING、ERROR状态，如果处于该状态意味着这个job正在执行或者可以之后被resume执行，做这种限制的原因不得而知（可能是构建的区间是基于时间吧，需要对一个cube并行的构建多个segment（时间区间的数据）的需求并不明显）。所以如果希望build或者merge cube，必须将未完成的cube的操作执行discard操作。然后根据操作类型执行具体的操作： 1. 如果是BUILD，如果这个cube中包含distinct count聚合方式的度量并且这个cube中已经存在其他segment，则执行appendAndMergeSegments函数，否则执行buildJob函数。 2. 如果是MERGE操作则执行mergeSegments函数。 3. 如果是REFRESH，则同样执行buildJob函数。为这个时间区间的segment重新构建。 buildJob函数构建一个新的segment，mergeSegments函数合并一个时间区间内的所有segments，appendAndMergeSegments函数则首先根据最后一个segment的时间区间的end值build一个新的segment然后再将所有的时间区间的segments进行合并（为什么包含distinct count的聚合函数的cube的构建一定要进行合并呢？这应该是有distinct-count使用的hyperloglog算法决定的，下次可以专门分析一下这个算法）。

BUILD操作

Build操作是构建一个cube指定时间区间的数据，由于kylin基于预计算的方式提供数据查询，构建操作是指将原始数据（存储在Hadoop中，通过Hive获取）转换成目标数据（存储在Hbase中）的过程。主要的步骤可以按照顺序分为四个阶段：1、根据用户的cube信息计算出多个cuboid文件，2、根据cuboid文件生成htable，3、更新cube信息，4、回收临时文件。每一个阶段操作的输入都需要依赖于上一步的输出，所以这些操作全是顺序执行的。

1. 计算cuboid文件

在kylin的CUBE模型中，每一个cube是由多个cuboid组成的，理论上有N个普通维度的cube可以是由2的N次方个cuboid组成的，那么我们可以计算出最底层的cuboid，也就是包含全部维度的cuboid（相当于执行一个group by全部维度列的查询），然后在根据最底层的cuboid一层一层的向上计算，直到计算出最顶层的cuboid（相当于执行了一个不带group by的查询），其实这个阶段kylin的执行原理就是这个样子的，不过它需要将这些抽象成mapreduce模型，提交mapreduce作业执行。

1.1 生成原始数据（Create Intermediate Flat Hive Table）

这一步的操作是根据cube的定义生成原始数据，这里会新创建一个hive外部表，然后再根据cube中定义的星状模型，查询出维度（对于DERIVED类型的维度使用的是外键列）和度量的值插入到新创建的表中，这个表是一个外部表，表的数据文件（存储在HDFS）作为下一个子任务的输入，它首先根据维度中的列和度量中作为参数的列得到需要出现在该表中的列，然后执行三步hive操作，这三步hive操作是通过hive -e的方式执行的shell命令。

1. drop TABLE IF EXISTS xxx. 2. CREATE EXTERNAL TABLE IF NOT EXISTS xxx() ROW FORMAT DELIMITED FIELDS TERMINATED BY '\177' STORED AS SEQUENCEFILE LOCATION xxxx，其中表名是根据当前的cube名和segment的uuid生成的，location是当前job的临时文件，只有当insert插入数据的时候才会创建，注意这里每一行的分隔符指定的是'\177'（目前是写死的，十进制为127）. 3. 插入数据，在执行之前需要首先设置一些配置项，这些配置项通过hive的SET命令设置，是根据这个cube的job的配置文件（一般是在kylin的conf目录下）设置的，最后执行的是INSERT OVERWRITE TABLE xxx SELECT xxxx语句，SELECT子句中选出cube星状模型中事实表与维度表按照设置的方式join之后的出现在维度或者度量参数中的列（特殊处理derived列），然后再加上用户设置的where条件和partition的时间条件（根据输入build的参数）. 需要注意的是这里无论用户设置了多少维度和度量，每次join都会使用事实表和所有的维度表进行join，这可能造成不必要的性能损失（多一个join会影响hive性能，毕竟要多读一些文件）。这一步执行完成之后location指定的目录下就有了原始数据的文件，为接下来的任务提供了输入。

1.2 创建事实表distinct column文件（Extract Fact Table Distinct Columns）

在这一步是根据上一步生成的hive表计算出还表中的每一个出现在事实表中的度量的distinct值，并写入到文件中，它是启动一个MR任务完成

的，MR任务的输入是HCatInputFormat，它关联的表就是上一步创建的临时表，这个MR任务的map阶段首先在setup函数中得到所有度量中出现在事实表的度量在临时表的index，根据每一个index得到该列在临时表中在每一行的值value，然后将<index, value>作为mapper的输出，该任务还启动了一个combiner，它所做的只是对同一个key的值进行去重（同一个mapper的结果），reducer所做的事情也是进行去重（所有mapper的结果），然后将每一个index对应的值一行行的写入到以列名命名的文件中。如果某一个维度列的distinct值比较大，那么可能导致MR任务执行过程中的OOM。
对于这一步我有一个疑问就是既然所有的原始数据都已经通过第一步存入到临时hive表中了，我觉得接下来就不用再区分维度表和事实表了，所有的任务都基于这个临时表，那么这一步就可以根据临时表计算出所有的维度列的distinct column值，但是这里仅仅针对出现在事实表上的维度，不知道这样做的原因是什么？难道是因为在下一步会单独计算维度表的dictionary以及snapshot？

1.3 创建维度词典（Build Dimension Dictionary）

这一步是根据上一步生成的distinct column文件和维度表计算出所有维度的词典信息，词典是为了节约存储而设计的，用于将一个成员值编码成一个整数类型并且可以通过整数值获取到原始成员值，每一个cuboid的成员是一个key-value形式存储在hbase中，key是维度成员的组合，但是一般情况下维度是一些字符串之类的值（例如商品名），所以可以通过将每一个维度值转换成唯一整数而减少内存占用，在从hbase查找出对应的key之后再根据词典获取真正的成员值。　　这一步是在kylin进程内的一个线程中执行的，它会创建所有维度的dictionary，如果是事实表上的维度则可以从上一步生成的文件中读取该列的distinct成员值（FileTable），否则则需要从原始的hive表中读取每一列的信息（HiveTable），根据不同的源（文件或者hive表）获取所有的列去重之后的成员列表，然后根据这个列表生成dictionary，kylin中针对不同类型的列使用不同的实现方式，对于time之类的（date、time、dtaetime和timestamp）使用DateStrDictionary，这里目前还存在着一定的问题，因为这种编码方式会首先将时间转换成'yyyy-MM-dd'的格式，会导致timestamp之类的精确时间失去天以后的精度。针对数值型的使用NumberDictionary，其余的都使用一般的TrieDictionary（字典树）。这些dictionary会作为cube的元数据存储的kylin元数据库里面，执行query的时候进行转换。　　之后还需要计算维度表的snapshotTable，每一个snapshot是和一个hive维度表对应的，生成的过程是：首先从原始的hive维度表中顺序得读取每一行每一列的值，然后使用TrieDictionary方式对这些所有的值进行编码，这样每一行每一列的之都能够得到一个编码之后的id（相同的值id也相同），然后再次读取原始表中每一行的值，将每一列的值使用编码之后的id进行替换，得到了一个只有id的新表，这样同时保存这个新表和dictionary对象（id和值得映射关系）就能够保存整个维度表了，同样，kylin也会将这个数据存储元数据库中。　　针对这一步需要注意的问题：首先，这一步的两个步骤都是在kylin进程的一个线程中执行的，第一步会加载某一个维度的所有distinct成员到内存，如果某一个维度的cardinality比较大，可能会导致内存出现OOM，然后在创建snapshotTable的时候会限制原始表的大小不能超过配置的一个上限值，如果超过则会执行失败。但是应该强调的是这里加载全部的原始维度表更可能出现OOM。另外，比较疑惑的是：1、为什么不在上一步的MR任务中直接根据临时表中的数据生成每一个distinct column值，而是从原始维度表中读取？2、计算全表的dictionary是为了做什么？我目前只了解对于drived维度是必要保存主键和列之间的映射，但是需要保存整个维度表？！

1.4 计算生成BaseCuboid文件（Build Base Cuboid Data）

何谓Base cuboid呢？假设一个cube包含了四个维度：A/B/C/D，那么这四个维度成员间的所有可能的组合就是base cuboid，这就类似在查询的时候指定了select count(1) from xxx group by A,B,C,D;这个查询结果的个数就是base cuboid集合的成员数。这一步也是通过一个MR任务完成的，输入是临时表的路径和分隔符，map对于每一行首先进行split，然后获取每一个维度列的值组合作为rowKey，但是rowKey并不是简单的这些维度成员的内容组合，而是首先将这些内容从dictionary中查找出对应的id，然后组合这些id得到rowKey，这样可以大大缩短hbase的存储空间，提升查找性能。然后在查找该行中的度量列，根据cube定义中度量的函数返回对该列计算之后的值。这个MR任务还会执行combiner过程，执行逻辑和reducer相同，在reducer中的key是一个rowKey，value是相同的rowKey的measure组合的数组，reducer回分解出每一个measure的值，然后再根据定义该度量使用的聚合函数计算得到这个rowKey的结果，其实这已经类似于hbase存储的格式了。

1.5 计算第N层cuboid文件（Build N-Dimension Cuboid Data）

这一个流程是由多个步骤的，它是根据维度组合的cuboid的总数决定的，上一层cuboid执行MR任务的输入是下一层cuboid计算的输出，由于最底层的cuboid（base）已经计算完成，所以这几步不需要依赖于任何的hive信息，它的reducer和base cuboid的reducer过程基本一样的（相同rowkey的measure执行聚合运算），mapper的过程只需要根据这一行输入的key（例如A、B、C、D中某四个成员的组合）获取可能的下一层的的组合（例如只有A、B、C和B、C、D），那么只需要将这些可能的组合提取出来作为新的key，value不变进行输出就可以了。 举个例子，假设一共四个维度A/B/C/D，他们的成员分别是（A1、A2、A3），（B1、B2）、(C1)、（D1），有一个measure（对于这列V，计算sum（V）），这里忽略dictionary编码。原始表如下：| A | B | C | D | V | | -----|-----|------|------|------| |A1|B1|C1|D1|2| |A1|B2|C1|D1|3| |A2|B1|C1|D1|5| |A3|B1|C1|D1|6| |A3|B2|C1|D1|8| 那么base cuboid最终的输出如下 （<A1、B1、C1、D1>、2） （<A1、B2、C1、D1>， 3） （<A2、B1、C1、D1>， 5） （<A3、B1、C1、D1>， 6） （<A3、B2、C1、D1>， 8） 那么它作为下面一个cuboid的输入，对于第一行输入 （<A1、B1、C1、D1>， 2），mapper执行完成之后会输出（<A1、B1、C1>， 2）、 （<A1、B1、D1>， 2）、 （<A1、C1、D1>， 2）、 （<B1、C1、D1>， 2）这四项，同样对于其他的内一行也会输出四行，最终他们经过reducer的聚合运算，得到如下的结果： （<A1、B1、C1>， 2） （<A1、B1、D1>， 2） （<A1、C1、D1>， 2 + 3） （<B1、C1、D1>，2 + 5 + 6 ） . . . 这样一次将下一层的结果作为输入计算上一层的cuboid成员，直到最顶层的cuboid，这一个层cuboid只包含一个成员，不按照任何维度进行group by。　　上面的这些步骤用于生成cuboid，假设有N个维度（对于特殊类型的），那么就需要有N + 1 层cuboid，每一层cuboid可能是由多个维度的组合，但是它包含的维度个数相同。

2 准备输出

在上面几步中，我们已经将每一层的cuboid计算完成，每一层的cuboid文件都是一些cuboid的集合，每一层的cuboid的key包含相同的维度个数，下面一步就是将这些cuboid文件导入到hbase中。

2.1 计算分组

这一步的输入是之前计算的全部的cuboid文件，按照cuboid文件的顺序（层次的顺序）一次读取每一个key-value，再按照key-value的形式统计每一个key和value占用的空间大小，然后以GB为单位，mapper阶段的输出是每当统计到1GB的数据，将当前的这个key和当前数据量总和输出，在reducer阶段根据用户创建cube时指定的cube大小（SMALL，MEDIUM和LARGE）和总的大小计算出实际需要划分为多少分区，这时还需要参考最多分区数和最少分区数进行计算，再根据实际数据量大小和分区数计算出每一个分区的边界key，将这个key和对应的分区编号输出到最终文件中，为下一步创建htable做准备。

2.2 创建HTable

这一步非常简单，根据上一步计算出的rowKey分布情况（split数组）创建HTable，创建一个HTable的时候还需要考虑一下几个事情：1、列组的设置，2、每一个列组的压缩方式，3、部署coprocessor，4、HTable中每一个region的大小。在这一步中，列组的设置是根据用户创建cube时候设置的，在hbase中存储的数据key是维度成员的组合，value是对应聚合函数的结果，列组针对的是value的，一般情况下在创建cube的时候只会设置一个列组，该列包含所有的聚合函数的结果；在创建HTable时默认使用LZO压缩，如果不支持LZO则不进行压缩，在后面kylin的版本中支持更多的压缩方式；kylin强依赖于hbase的coprocessor，所以需要在创建HTable为该表部署coprocessor，这个文件会首先上传到HBase所在的HDFS上，然后在表的元信息中关联，这一步很容易出现错误，例如coprocessor找不到了就会导致整个regionServer无法启动，所以需要特别小心；region的划分已经在上一步确定了，所以这里不存在动态扩展的情况，所以kylin创建HTable使用的接口如下： public void createTable(final HTableDescriptor desc , byte [][] splitKeys)

2.3 构建hfile文件

创建完了HTable之后一般会通过插入接口将数据插入到表中，但是由于cuboid中的数据量巨大，频繁的插入会对Hbase的性能有非常大的影响，所以kylin采取了首先将cuboid文件转换成HTable格式的Hfile文件，然后在通过bulkLoad的方式将文件和HTable进行关联，这样可以大大降低Hbase的负载，这个过程通过一个MR任务完成。　　这个任务的输入是所有的cuboid文件，在mapper阶段根据每一个cuboid成员的key-value输出，如果cube定义时指定了多个列组，那么同一个key要按照不同列组中的值分别输出，例如在cuboid文件中存在一行cuboid=1，key=1，value=sum(cost),count(1)的数据，而cube中将这两个度量划分到两个列组中，这时候对于这一行数据，mapper的输出为<1, sum(cost)>和<1,count(1)>。reducer使用的是org.apache.hadoop.hbase.mapreduce.KeyValueSortReducer，它会按照行排序输出，如果一行中包含多个值，那么会将这些值进行排序再输出。输出的格式则是根据HTable的文件格式定义的。

2.4 BulkLoad文件

这一步将HFile文件load到HTable中，因为load操作会将原始的文件删除（相当于remove），在操作之前首先将所有列组的Hfile的权限都设置为777，然后再启动LoadIncrementalHFiles任务执行load操作，它的输入为文件的路径和HTable名，这一步完全依赖于HBase的工具。这一步完成之后，数据已经存储到HBase中了，key的格式由cuboid编号+每一个成员在字典树的id组成，value可能保存在多个列组里，包含在原始数据中按照这几个成员进行GROUP BY计算出的度量的值。

3 收尾工作

执行完上一步就已经完成了从输入到输出的计算过程，接下来要做的就是一些kylin内部的工作，分别是更新元数据，更新cube状态，垃圾数据回收。

3.1 更新状态

这一步主要是更新cube的状态，其中需要更新的包括cube是否可用、以及本次构建的数据统计，包括构建完成的时间，输入的record数目，输入数据的大小，保存到Hbase中数据的大小等，并将这些信息持久到元数据库中。

3.2 垃圾文件回收

这一步是否成功对正确性不会有任何影响，因为经过上一步之后这个segment就可以在这个cube中被查找到了，但是在整个执行过程中产生了很多的垃圾文件，其中包括：1、临时的hive表，2、因为hive表是一个外部表，存储该表的文件也需要额外删除，3、fact distinct 这一步将数据写入到HDFS上为建立词典做准备，这时候也可以删除了，4、rowKey统计的时候会生成一个文件，此时可以删除。5、生成HFile时文件存储的路径和hbase真正存储的路径不同，虽然load是一个remove操作，但是上层的目录还是存在的，也需要删除。这一步kylin做的比较简单，并没有完全删除所有的临时文件，其实在整个计算过程中，真正还需要保留的数据只有多个cuboid文件（需要增量build的cube），这个因为在不同segment进行merge的时候是基于cuboid文件的，而不是根据HTable的。

在Kylin-1.x版本中，整个cube的一个build的过程大概就是这样，这样的一个build只不过是生成一虐segment，而当一个cube中存在多个segment时可能需要将它们进行merge，merge的过程和build的流程大致是相同的，不过它不需要从头开始，只需要对字典进行merge，然后在对cuboid文件进行merge，最后生成一个新的HTable。 但是在Kylin-2.x版本中，整个家沟发生了很大的变化，build的引擎也分成了多套，分别是原始的MR引擎，基于Fast Cubing的MR引擎和Spark引擎，这使得build进行的更迅速，大大降低等待时间，后面会持续的再对新的引擎进行分析。