OLAP引擎——Kylin介绍

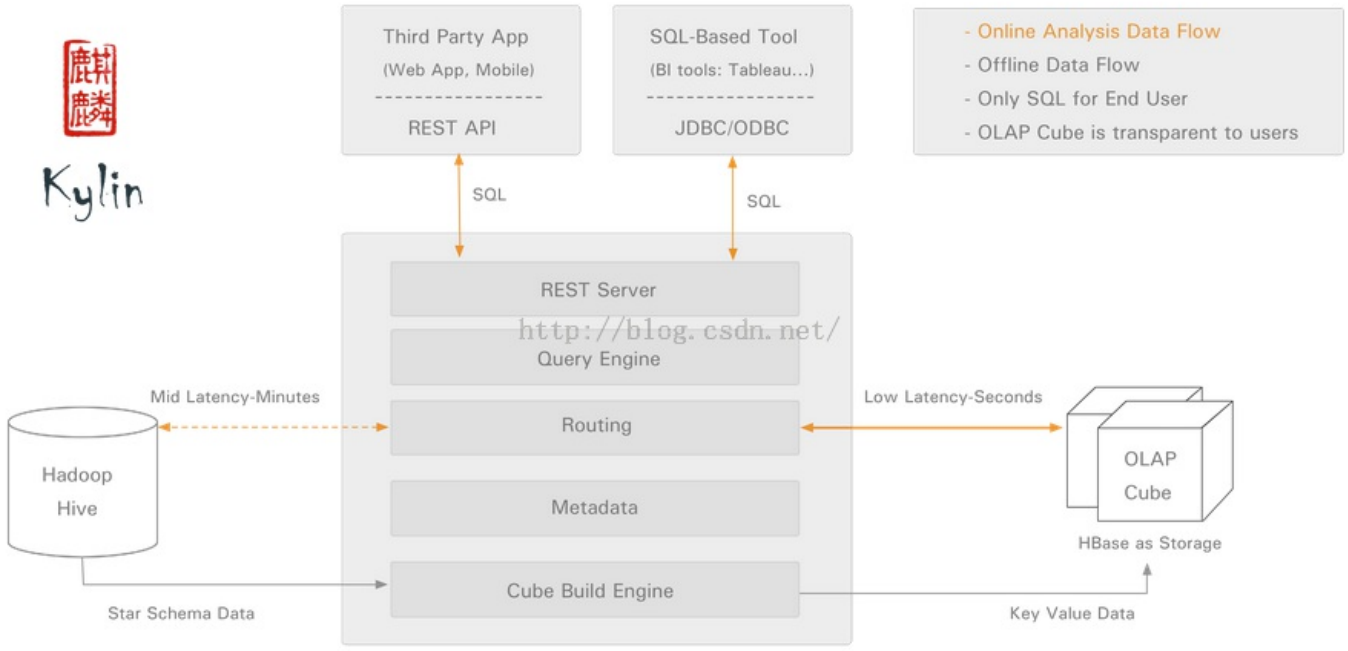
Kylin简介

Kylin是ebay开发的一套OLAP系统，与Mondrian不同的是，它是一个MOLAP系统，主要用于支持大数据生态圈的数据分析业务，它主要是通过预计算的方式将用户设定的多维立方体缓存到HBase中（目前还仅支持hbase），这段时间对mondrian和kylin都进行了使用，发现这两个系统是时间和空间的一个权衡吧，mondrian是一个ROLAP系统，所有的查询可以通过实时的数据库查询完成，而不会有任何的预计算，大大节约了存储空间的要求（但是会有查询结果的缓存，目前是缓存在程序内存中，很容易导致OOM），而kylin是一个MOLAP系统，通过预计算的方式缓存了所有需要查询的的数据结果，需要大量的存储空间（原数据量的10+倍）。一般我们要分析的数据可能存储在关系数据库（mysql、oracle，一般是程序内部写入的一些业务数据，可能存在分表甚至分库的需求）、HDFS上数据（结构化数据，一般是业务的日志信息，通过hive查询）、文本文件、excel等。

kylin主要是对hive中的数据进行预计算，利用hadoop的mapreduce框架实现。而mondrian理论上可以支持任意的提供SQL接口数据，由于关系数据库一般会存在索引，所以即使使用mondrian去查询性能还是可以接受的，当前我们使用的oracle数据库，千万条级别的记录，查询可以在分钟级别完成，但是对于hive、 这样的数据源查询就太慢了，慢得不可以接受。

系统架构

于是，我们开始尝试使用kylin，kylin的出现就是为了解决大数据系统中TB级别数据的数据分析需求，而对于关系数据库中的数据分析进行预计算可能有点不合适了（关系数据库一般存在索引使得即使数据量很大查询也不会慢的离谱，除非SQL写的很烂）。在使用kylin的过程中，也逐渐对kylin有了一定的认识，首先看一下kylin的系统架构：



kylin系统架构

kylin由以下几部分组成：

- REST Server：提供一些restful接口，例如创建cube、构建cube、刷新cube、合并cube等cube的操作，project、table、cube等元数据管理、用户访问权限、系统配置动态修改等。除此之外还可以通过该接口实现SQL的查询，这些接口一方面可以通过第三方程序的调用，另一方也被kylin的web界面使用。
- jdbc/odbc接口：kylin提供了jdbc的驱动，驱动的classname为org.apache.kylin.jdbc.Driver，使用的url的前缀jdbc:kylin:，使用jdbc接口的查询走的流程和使用RESTFul接口查询走的内部流程是相同的。这类接口也使得kylin很好的兼容tebleau甚至mondrian。
- Query引擎：kylin使用一个开源的Calcite框架实现SQL的解析，相当于SQL引擎层。
- Routing：该模块负责将解析SQL生成的执行计划转换成cube缓存的查询，cube是通过预计算缓存在hbase中，这部分查询是可以再秒级甚至毫秒级完成，而还有一些操作使用过查询原始数据（存储在hadoop上通过hive上查询），这部分查询的延迟比较高。
- Metadata：kylin中有大量的元数据信息，包括cube的定义，星状模型的定义、job的信息、job的输出信息、维度的directory信息等等，元数据和cube都存储在hbase中，存储的格式是json字符串，除此之外，还可以选择将元数据存储在本地文件系统。

- Cube构建引擎：这个模块是所有模块的基础，它负责预计算创建cube，创建的过程是通过hive读取原始数据然后通过一些mapreduce计算生成Htable然后load到hbase中。

关键流程

在kylin中，最关键的两个流程是cube的预计算过程和SQL查询转换成cube的过程，cube的构造可以分成cube的构建和cube的合并，首先需要创建一个cube的定义，包括设置cube名、cube的星状模型结构，dimension信息、measure信息、设置where条件、根据hive中事实表定义的partition设置增量cube，设置rowkey等信息，这些设置在mondrian中也是可以看到的，一个cube包含一些dimension和measure，where条件决定了源数据的大小，在mondrian中可以通过view实现。另外，kylin还提供了增量计算的功能，虽然达不到实时计算的需求，但是基本上可以满足数据分析的需求。

查询解析过程主要是使用Calcite框架将用户输入的SQL解析并转换成对hbase的key-value查询操作以获取结果，但是经过使用发现它对SQL的支持是比较差的，所有的SQL不能使用from A,B where xxx之类的join方式，必须使用inner（left、right） join on的方式，否则解析就会出错，这就会导致mondrian生成的SQL压根不能使用kylin查询（因为mondrian生成的SQL是前面一种方式的），另外还有一个局限性就是发现只能对cube相关的表和列进行查询，例如根据维度进行group by查询定义的度量信息，而其他的查询也统统的返回错误，这点倒也不算是很大的问题，毕竟cube的模型已经定义，我们不太可能查询这个模型以外的东西。还有一点有点不能接受的是kylin对于子查询的支持很弱，测试发现查询的结果经常返回空（没有一行），而相同的查询在hive中是有结果的，这对于一些产品方需求支持不是很好，例如产品方可能需要查询年销售额大于xx的地区的每个月的销售总额。我们一般情况下会写出这样的sql：select month, sum(sales) from fact where location in (select location from fact group by year having sum(sales) > 1000) group by month;前一段时间测试发现这种SQL对于关系数据库简直是灾难，因为in语句会导致后面的子查询没有缓存结果，而写成select month, sum(sales) from fact as A inner join (select location from fact group by year having sum(sales) > 1000) as B on A.location = B.location group by month;可以提高性能.

但是测试发现kylin返回的结果为空，而kylin对于in语句的查询时非常高效的（毕竟全部走缓存），那么我们就不得不首先执行子查询得到location集合，然后再写一个SQL使用where location in xxx（kylin对于使用in子句的查询支持还是相当棒的）的方式获得结果，这个应该是需要改进的地方吧。

cube模型

前面介绍了cube在创建过程中需要的设置，这里看一些每一个设置的具体含义吧，首先我们会设置cube名和notification列表，前者需要保证是全局唯一的，后者是一些Email用于通知cube的一些事件的发生。接着我们需要定义一个星状模型，和一般的数据仓库模型一样，需要指定一个事实表和任意多个维度表，如果存在维度表还需要指定事实表和维度表的关联关系，也就是join方式。接下来是定义dimension，在定义dimension的时候可以选择dimension的类型，分为Normal、Hierachy以及Derived，这个后面再进行介绍，dimension的定义决定着cube的大小，也需要用户对原始的表非常了解。

接下来是定义measure，kylin会为每一个cube创建一个聚合函数为count(1)的度量，它不需要关联任何列，用户自定义的度量可以选择SUM、COUNT、DISTINCT COUNT、MIN、MAX，而每一个度量定义时还可以选择这些聚合函数的参数，可以选择常量或者事实表的某一列，一般情况下我们当然选择某一列。这里我们发现kylin并不提供AVG等相对较复杂的聚合函数（方差、平均差更没有了），主要是因为它需要基于缓存的cube做增量计算并且合并成新的cube，而这些复杂的聚合函数并不能简单的对两个值计算之后得到新的值，例如需要增量合并的两个cube中某一个key对应的sum值分别为A和B，那么合并之后的则为A+B,而如果此时的聚合函数是AVG，那么我们必须知道这个key的count和sum之后才能做聚合。这就要求使用者必须自己想办法自己计算了。

定义完measure之后需要设置where条件，这一步是对原始数据进行过滤，例如我们设定销售额小于XXX的地区不在于本次分析范围之内，那么就可以在where条件里设定location in xxx（子查询），那么生成的cube会过滤掉这些location，这一步其实相当于对无效数据的清洗，但是在kylin中这个是会固化的，不容易改变，例如我今天希望将销售额小于XX的地区清洗掉，明天可能有想将年消费小于xxx的用户去除，这就需要每次都创建一个相同的cube，区别仅仅在于where条件，他们之间会有很多的重复缓存数据，也会导致存储空间的浪费，但这也是MOLAP系统不可避免的，因此当过滤条件变化比较多的时候，更好的方案则是创建一个完整的cube（不设置任何where条件），使用子查询的方式过滤掉不希望要的一些维度成员。

接下来的一步是设置增量cube信息，首先需要选择事实表中的某一个时间类型的分区列（貌似只能是按照天进行分区），然后再指定本次构建的cube的时间范围（起始时间点和结束时间点），这一步的结果会作为原始数据查询的where条件，保证本次构建的cube只包含这个闭区间时间内的数据，如果事实表没有时间类型的分区别或者没有选择任何分区则表示数据不会动态更新，也就不可以增量的创建cube了。

最后一步设置rowkey，这一步的建议是看看就可以了，不要进行修改，除非对kylin内部实现有比较深的理解才能知道怎么去修改。当然这里有一个可以修改的是mandatory dimension，如果一个维度需要在每次查询的时候都出现，那么可以设置这个dimension为mandatory，可以省去很多存储空间，另外还可以对所有维度进行划分group，不会组合查询的dimension可以划分在不同的group中，这样也会降低存储空间。

Dimension介绍

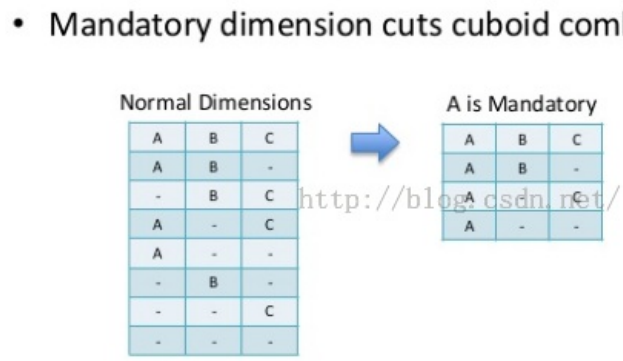
在一个多维数据集合中，维度的个数决定着维度之间可能的组合数，而每一个维度中成员集合的大小决定着每一个可能的组合的个数，例如有三个普通的维度A、B、C，他们的不同成员数分别为10/100/1000，那么一个维度的组合有2的3次方个，分别是{空、A、B、C、AB、BC、AC、ABC}，每一个成员我们称为cuboid（维度的组合），而这些集合的成员组合个数分别为1、10、100、1000、10*100*、*100*1000、10*1000和10*100*1000。我们称每一个dimension中不同成员个数为cardinatily，我们要尽量避免存储cardinatily比较高的维度的组合，在上面的例子中我们可以不缓存BC和C这两个cuboid，可以通过计算的方式通过ABC中成员的值计算出BC或者C中某个成员组合的值，这相当于是时间和空间的一个权衡吧。

在kylin中存在的四种维度是为了减少cuboid的个数，而不是每一个维度是否缓存的，当前kylin是对所有的cuboid中的所有组合都进行计算和存储的，对于普通的dimension，从上面的例子中可以看出N个维度的cuboid个数为2的N次方，而kylin中设置了一些维度可以减少cuboid个数，当然，这需要使用者对自己需要的维度十分了解，知道自己可能根据什么进行group by。

好了，我们先来看一下kylin中的三种特殊的dimension以及它们的作用，这里参考：http://www.slideshare.net/YangLi43/design-cube-in-apache-kylin

1、Mandatory维度

这种维度意味着每次查询的group by中都会携带的，将某一个dimension设置为mandatory可以将cuboid的个数减少一半，如下图：

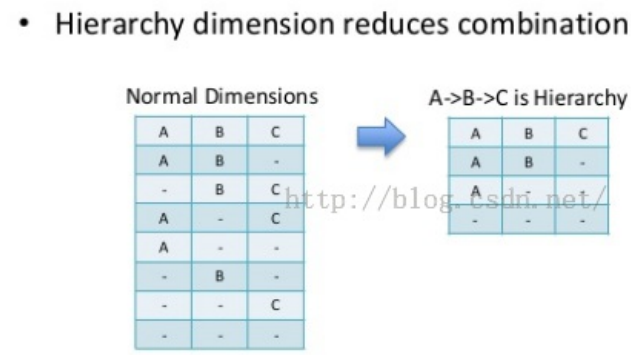


mandatory dimension

这是因为我们确定每一次group by都会携带A，那么就可以省去所有不包含A这个维度的cuboid了。

2、hierarchy维度

这种维度是最常见的，尤其是在mondrian中，我们对于多维数据的操作经常会有上卷下钻之类的操作，这也就需要要求维度之间有层级关系，例如国家、省、城市，年、季度、月等。有层级关系的维度也可以大大减少cuboid的个数。如下图：



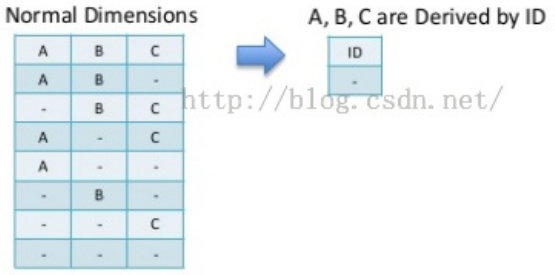
hierarchy dimension

这里仅仅局限于A/B/C是一个层级，例如A是年份，B是季度、C是月份，那么查询的时候可能的组合只有年、xx年的季度、xx年xx季度的xx月，这就意味着我们不能再单独的对季度和月份进行聚合了，例如我们查询的时候不能使用group by month，而必须使用group by year，quart，month。如果需要单独的对month进行聚合，那么还需要再使用month列定义一个单独的普通维度。

3、derived维度

这类维度的意思是可推导的维度，需要该维度对应的一个或者多个列可以和维度表的主键是一对一的，这种维度可以大大减少cuboid个数，如下图：

- Derived dimension reduces combination from 2^N to 2 at the cost of extra runtime aggregation.



derived dimension

例如timeid是时间这个维度表的主键，也就是事实表的外检，时间只精确到天，那么year、month、day三列可以唯一对应着一个time*id*，而事实表的外键，那么我们可以指定year、month、day为一个derived维度，实际存储的时候可以只根据timeid的取值决定维度的组合，但这就要求我们在查询的时候使用的group by必须指定derived维度集合中的所有列。

计算Cuboid

最后，简单介绍一下如何计算cuboid个数的，假设我们存在两个普通维度brand、product，存在一个hierarchy，包含四个维度分别为year、quart、month和day，一个derived维度，指定location信息，包含country、province和city列，这相当于一共9个维度，但是根据上面的分析我们并不需要512分cuboid。

- 第0层的cuboid（不包含任何维度，不包含group by），cuboid的个数为1，这个cuboid的成员个数也为1；
- 第1层的cuboid包含一个维度，一共有4种组合（分别为brand、product、year、location，因为quart是hierarchy的第二个层级，不能单独group by，而location的三列可以视为一个整体），成员个数则有每一个维度的cardinality；
- 第2层的cuboid有7种，分别为{brand、product}、{brand、year}、{brand、location}、{product、year}、{product、location}、{year、location}和{year、quart}；
- 第3层的cuboid有8种，分别为{brand、product、year}、{brand、product、location}、{product、year、location}、{brand、product、year}、{brand、year、quart}、{product、year、quart}、{location、year、quart}、{year、quart、month}；
- 第4层的cuboid有8种，分别为{brand、product、year、location}、{brand、product、year、quart}、{brand、location、year、quart}、{product、location、year、quart}、{brand、year、quart、month}、{product、year、quart、month}、{location、year、quart、month}、{year、quart、month、day}
- 第5层的cuboid有7种，分别为{brand、product、year、quart、location}、{brand、product、year、quart、momth}、{brand、location、year、quart、month}、{product、location、year、quart、month}、{brand、year、quart、month、day}、{product、year、quart、month、day}、{location、year、quart、month、day}
- 第6层的cuboid有5种，分别为{brand、product、year、quart、month、location}、{brand、product、year、quart、momth、day}、{brand、location、year、quart、month、day}、{product、location、year、quart、month、day}
- 第7层的cuboid有1中，为{brand、product、year、quart、month、day、location}

所以一共40个cuboid（kylin计算的是39个，应该没有把第0层的计算在内）。

增量cube

由于kylin的核心在于预计算缓存数据，那么对于实时的数据查询的支持就不如mondrian好了，但是一般情况下我们数据分析并没有完全实时的要求，数据延迟几个小时甚至一天是可以接受的，kylin提供了增量cube的接口，kylin的实现是一个cube（这里是指逻辑上的cube）中可以包含多个segment，每一个segment对应着一个物理cube，在实际存储上对应着一个hbase的一个表，用户定义根据某一个字段进行增量（目前仅支持时间，并且这个字段必须是hive的一个分区字段），在使用的时候首先需要定义好cube的定义，可以指定一个时间的partition字段作为增量cube的依赖字段，其实这个选择是作为原始数据选择的条件，例如选择起始时间A到B的数据那么创建的cube则会只包含这个时间段的数据聚合值，创建完一个cube之后可以再次基于以前的cube进行build，每次build会生成一个新的segment，只不过原始数据不一样了（根据每次build指定的时间区间），每次查询的时候会查询所有的segment聚合之后的值进行返回，有点类似于tablet的存储方式，但是当segment存在过多的时候查询效率就会下降，因此需要在存在多个segment的时候将它们进行合并，合并的时候其实是指定了一个时间区间，内部会选择这个时间区间内的所有segment进行合并，合并完成之后使用新的segment替换被合并的多个segment，合并的执行时非常迅速的，数据不需要再从HDFS中获取，直接将两个hbase表中相同key的数据进行聚合就可以了。但是有一点需要注意的是当合并完成之后，被合并的几个segment所对应的hbase表并没有被删除。实际的使用过程中对于增量的cube可以写个定时任务每天凌晨进行build，当达到一个数目之后进行merge（其实每次build完成之后都进行merge也应该是可以的）。

cube的词典树

kylin的cube数据是作为key-value结构存储在hbase中的，key是每一个维度成员的组合值，不同的cuboid下面的key的结构是不一样的，例如cuboid={brand，product，year}下面的一个key可能是brand='Nike'，product='shoe'，year=2015，那么这个key就可以写成Nike:shoe:2015，但是如果使用这种方式的话会出现很多重复，所以一般情况下我们会把一个维度下的所有成员取出来，然后保存在一个数组里面，使用数组的下标组合成为一个key，这样可以大大节省key的存储空间，kylin也使用了相同的方法，只不过使用了字典树（Trie树），每一个维度的字典树作为cube的元数据以二进制的方式存储在hbase中，内存中也会一直保持一份。

总结

以上介绍了kylin的整体框架以及部分的模块的流程，由于之前主要是关注cube的一些操作，例如创建、构建、合并等，对于查询这一块了解的较少，当然，这一块是kylin的核心之一。接下来会从源代码的角度去看kylin是如何构建和mergecube的，以及执行查询的流程。