**Druid虚拟表设计方案**

目录

[1、需求背景 1](#_Toc534908247)

[2、概要设计思路 2](#_Toc534908248)

[3、常用处理方案： 3](#_Toc534908249)

[1、SQL-on-Hadoop方案（Hive、Spark SQL、Impala、drill） 3](#_Toc534908250)

[2、流计算方案（Storm、Spark Streaming） 4](#_Toc534908251)

[3、全文检索方案（Solr、ElasticSearch） 4](#_Toc534908252)

[4、大厂方案： 5](#_Toc534908253)

[4、详细设计： 8](#_Toc534908254)

[1、业务表的关联关系： 8](#_Toc534908255)

[2、业务中主要使用的维度： 10](#_Toc534908256)

[3、交叉业务字段： 11](#_Toc534908257)

[4、各个主要维度基数 11](#_Toc534908258)

[5、详细设计子表划分： 12](#_Toc534908259)

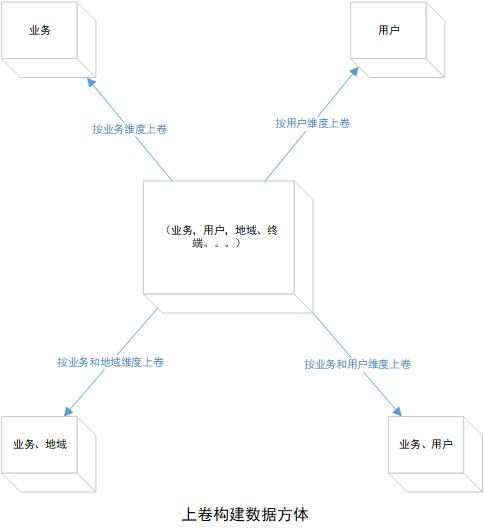
[6、数据验证 14](#_Toc534908260)

[7、代码修改方案： 14](#_Toc534908261)

# 1、需求背景

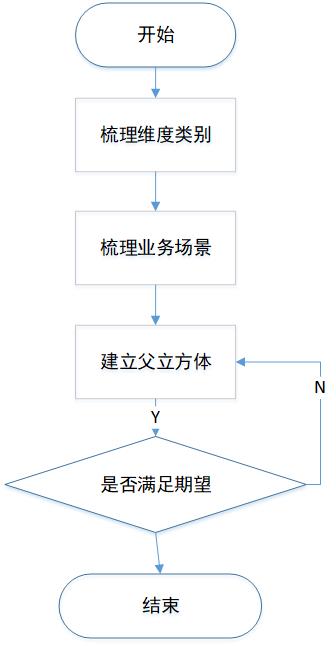
友云音采用druid作为OLAP工具进行查询，Druid目前是宽表设计构建的为基本方体，目前的表可以满足数据查询的需求，但在效率上较低。目前，业务基本立方体包含64个基本维度，以按天聚集粒度计算，一个月生成的基本聚集单元为5435w多个，而一个月的业务总量为6600w个，聚合比例不足1.2。这样的预聚和比例，无法保证按月及年的查询效率及更无法满足复杂查询计算的要求。这样的基本方体设计，无法满足不断上涨的业务需求，故需要对基本方体进行泛化构建高级方体。再基于泛化的数据进行查询，以提高预聚合比例。

我们在构建数据方体时，不是通过原始数据建立这样既浪费资源又效率低下，我们通过最小的子女数据方体会更有效。我们的最小子女数据方体是业务宽表。



# 2、概要设计思路

业务宽表是支持业务查询最多的表，目前的查询大部分均的通过业务宽表进行查询。一般存在2-3个维度的交叉查询。对于业务表拆分的流程和步骤如下：



梳理业务宽表维度，按维度进行维度分类，划分出强相关的维度项目为一个分类。

梳理业务场景，按之前划分的维度分类进行统计，分类出使用场景与维度分类之间的关系。

按梳理场景及维度分类的，建立父立方体的。

重导历史数据，测试场景是否满足期望，即聚合比例至少达到10，并测定聚合与基数最佳比例点。

# 3、常用处理方案：

## 1、SQL-on-Hadoop方案（Hive、Spark SQL、Impala、drill）

|  |  |  |
| --- | --- | --- |
| 考核项 | 适配程度 | 适配描述 |
| 数据规模 | V | 储存基于hdfs，存储PB级的数据，数据可无限扩展 |
| 时效性 | X | Hdfs的特性，所有的计算均mapreduce的框架或mpp，常规应用均是T+1数据 |
| 查询性能及并发性 | X | 此类查询并非为即席查询的场景设计，通常说一个sql的执行时间从几分钟几小时不等，如果是百亿规模的查询，采用全表扫描的方式，至少需要数个小时。 |
| 灵活性 | V | sql支持比较全  生态圈比较完善，导入导出比较灵活。  支持JDBC方式，可以与常见的报表系统无缝衔接。 |
| 运维 | V | Hdfs及yarn等组件的高容错性决定了，单点的损坏不会影响整体集群。 |
| 资源占用 |  | 磁盘存储多个副本，磁盘占用较多。  计算的过程需要消耗内存和cpu等资源，但如果内存不足，部分数据被持久化到磁盘的，可能影响任务的性能。 |

## 2、流计算方案（Storm、Spark Streaming）

|  |  |  |
| --- | --- | --- |
| 数据规模 | V | 数据源基于hdfs或在线实时数据，数据可无限扩展。 |
| 时效性 | V | 基于kafka的方式导入数据，数据几秒即可见。 |
| 查询性能与并发 | V | 预先将需要查询的结果处理好，查询时直接访问或将结果暂存在hbase及关系型数据库中，查询性能非常好。 |
| 灵活性 | X | 无法查看明细数据，只能看特定粒度查询，而且需要预先设计数据模型结构，按需查询。 |
| 运维 | V | 容灾性能很好，单间故障任务会自动重算。扩展性较简单。 |
| 资源占用 | X | 对内存的要求很高。 |

## 3、全文检索方案（Solr、ElasticSearch）

|  |  |  |
| --- | --- | --- |
| 考核项 | 适配度 | 适配描述 |
| 数据规模 | X | 使用场景在千万级别，如果到亿级，需要更多的内存资源 |
| 时效性 | V | 千万级别数据量时，导入及查询性能较好，  到亿级以上时，查询性能明显 |
| 查询性能与并发 | V | 采用倒排索引，不需要全表扫描  千万级别一下，内存充足的情况下性能良好。 |
| 灵活性 | X | 主要应用场景为搜索引擎，统计分组功能相对较弱。  对sql支持较弱，复杂查询无法使用sql。 |
| 运维 | V | 数据量一旦过大，需要的资源成倍增加，采用master为单点，index多的情况下，效率下降明显，所有的任务均要经过master，瓶颈明显。一旦有复杂查询，直接会导致整个集群崩溃。副本恢复缓慢。 |
| 资源占用 | X | 需要大量内存，。 |

预计算：

|  |  |  |
| --- | --- | --- |
| 考核项 | 适配度 | 适配描述 |
| 数据规模 | X | 根据聚合粒度，可以到亿级到千亿级 |
| 时效性 | V | 导入及查询性能较好， |
| 查询性能与并发 | V | 采用bitmap，有良好的性能 |
| 灵活性 | X | 目前计算不支持join，对sql支持较弱，需要预先定义好数据立方体。 |
| 运维 | V | 分布式，如果出现问题，可自动调度，但如果数据提交出问题会大量 |
| 资源占用 | X | 需要大量内存。 |

## 4、大厂方案：

1、阿里Mdrill：

1.满足大数据查询需求：adhoc每天的数据量为30亿条，随着日积月累，数据会越来越大，mdrill采用列存储，索引，分布式技术，适当的分区等满足用户对数据的实时在线分析的需求。  
2.支持增量更新：离线形式的mdrill数据支持按照分区方式的增量更新。  
3.支持实时数据导入：在仅有10台机器的情况下，支持每天10亿级别（高峰每小时2亿）的实时导入。  
4.响应时间快：列存储、倒排索引、高效的数据压缩、内存计算，各种缓存、分区、分布式处理等等这些技术，使得mdrill可以仅在几秒到几十秒的时间分析百亿级别的数据。  
5.低成本：目前在阿里adhoc仅仅使用10台48G内存的PC机，但确存储了超过千亿规模的数据。

构建实现一个基于Hadoop分布式架构下的实时的、多维的、交互式的查询、统计、分析引擎，具有万亿数据规模下的万级维度秒级统计分析能力，并具备企业级的稳定可靠表现。 通过细粒度的索引，精确粒度的索引。数据即时导入，索引即时生成，通过索引高效定位到相关数据。能够与Spark深度集成，Spark直接对检索结果集分析计算，同样场景让Spark性能加快百倍。 整体技术架构如下：

2、Hermes平台架构

**Hermes实时多维分析平台架构**

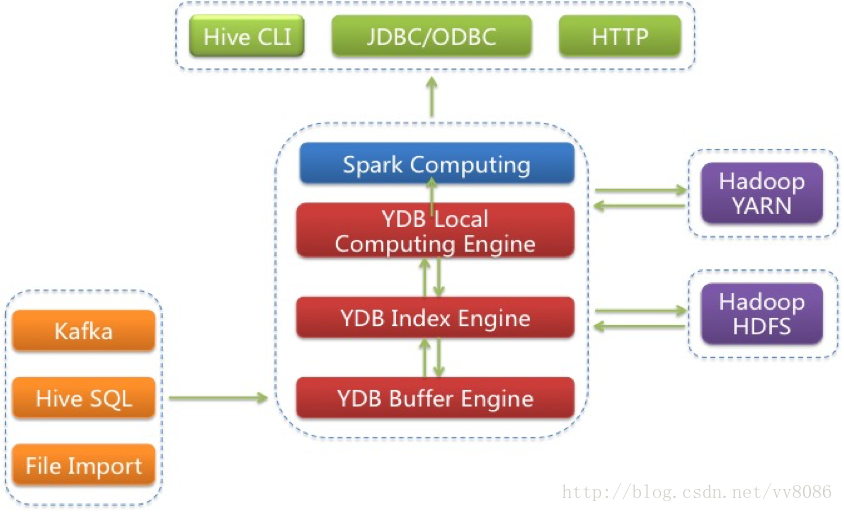
**数据接入**：实时、灵活。支持用户自定义包导入；已入库到TDW的数据，可配置化导出到Hermes，“一次索引、多次使用”；Hermes与TDBank（TDBank是腾讯数据银行，主要负责数据的收集，分发，预处理以及管理工作）做了打通，实现数据从生产环境实时采集入库Hermes。

**数据查询**：SQL接口，使用方便，降低使用门槛。支持查询、统计分析、查询结果导出、导入导出状态查询、导入包分析功能。 其中，导入包分析，指用户定义导入包（用户ID列表），即外部数据与已在Hermes中建好索引的大盘数据做关联，对用户导入包中对应的用户ID特征过滤分析的过程。

**数据计算**：任意纬度组合分析、实时下钻分析，秒极响应。

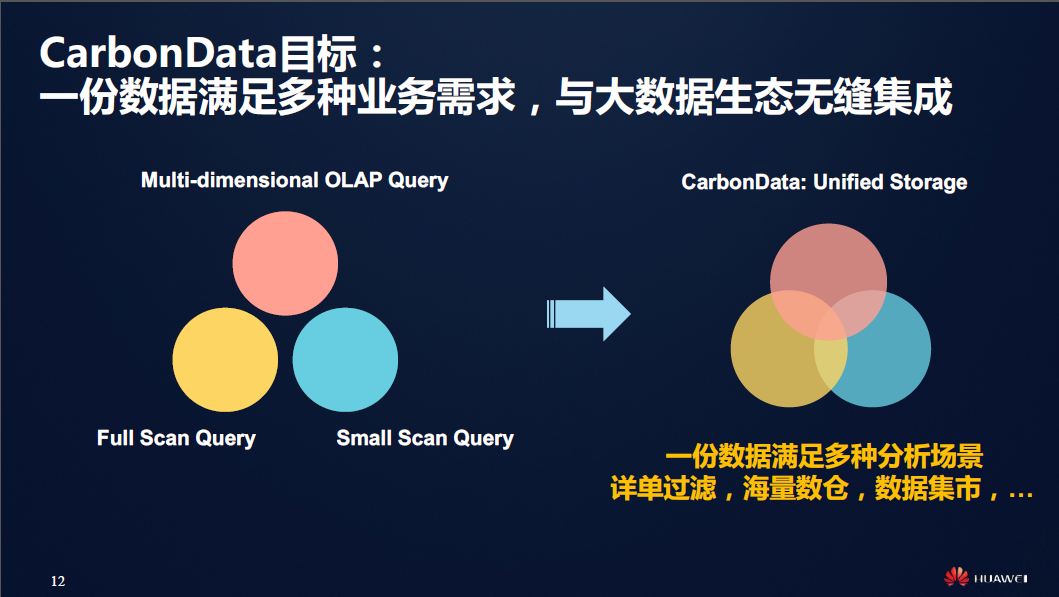
**数据存储**：本地+HDFS存储，高可靠性。

3、延云YDB



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| amtint列筛选 | 筛选后条数 | 排序方式 | YDBBlockSort | Spark |
| 无筛选 | 100亿 | 降序 | 3.3 | 1118 |
| 升序 | 3.6 | 1085 |
| 100 TO 900 | 80亿 | 降序 | 1.5 | 1093 |
| 升序 | 1.3 | 1070 |
| 100 TO 600 | 50亿 | 降序 | 1.53 | 1104 |
| 升序 | 1.38 | 867 |
| 100 TO 200 | 10亿 | 降序 | 7.00 | 1115 |
| 升序 | 1.11 | 1131 |
| 100 TO 110 | 1亿 | 降序 | 2.1 | 1160 |
| 升序 | 3.44 | 1114 |
| 100 TO 101 | 0.1亿 | 降序 | 10.67 | 1089 |
| 升序 | 7.0 | 1110 |

CarbonData：



1、支持海量数据扫描并取其中几列；

2、支持根据主键进行查找，并在秒级响应；

3、支持在海量数据进行类似于OLAP的交互式查询，并且查询中涉及到许多过滤条件，这种类型的workload应该在几秒钟内响应；

4、支持快速地抽取单独的记录，并且从该记录中获取到所有列信息；

5、支持HDFS，无缝对接Hadoop生态圈，天生带有分布式基因。

对于OLAP查询来说，存在多种不同类型的查询，存储结构的不同会影响到不同查询的数据表现。所以CarbonData的定位是作为一种通用的查询存储数据，通过Spark SQL来解决海量查询的问题，并且能够与Hadoop生态圈进行无缝对接。CarbonData最初的应用是与Spark SQL和Spark DataFrame深度结合，后续由携程团队将CarbonData引入了Presto，滴滴团队将CarbonData引入Hive。

其实无论是多维的OLAP查询，还是完整的扫描查询，还是部分范围查询。



# 4、详细设计：

## 1、业务表的关联关系：

首先我们明确目前宽表中的维度分类如下：

在业务数据中，我们的维度可以分为以下几类：

|  |  |  |  |
| --- | --- | --- | --- |
| 维度分类 | 维度字段 |  |  |
| 租户信息 | tid  tname |  |  |
| 应用信息 | appid  appname  app\_ver  app\_supplier  app\_industry  app\_field  app\_type |  |  |
| 业务信息 | back\_error\_d  back\_error\_is  ba  ba\_code  ba\_tag  probe\_type  op\_type  module  bstotal\_d  txtotal\_d  codetime\_d  nettotal\_d  dbtimems\_d  rendertime\_d  tfs\_up\_d  tfs\_down\_d  nodecode  btnname  nodename |  |  |
| 用户信息 | user\_code  user\_name  user\_tag  org\_name  org\_code  industry  role |  |  |
| 地域信息 | ip  city  province  country |  |  |
| 终端信息 | deviceid  terminal\_type  browser\_type  browser\_ver  os\_type  os\_ver  rqua  manufacturer  model  root  screen\_size  jre\_ver  jre\_xmx |  |  |
| 网络供应商信息 | carrier  wan |  |  |
| Agent版本信息 | agent\_ver  jvm\_probe\_ver  js\_probe\_ver  mobile\_probe\_ver  client\_probe\_ver |  |  |

由上可知，友云音所有的维度表，租户和应用信息为每个查询均要带入的维度。而业务信息，用户信息，地域信息，终端信息，网络供应商信息。需要根据业务使用情况进行关联，而agent版本信息，在查询中并不涉及。故在数据泛化时可以不考虑。

## 2、业务中主要使用的维度：

根据以上分析，需要明确实际业务中，明确的具体的业务关联关系，并确定

在友云音产品中重要业务页面的使用维度如下：

|  |  |  |
| --- | --- | --- |
| 模块页面 | 主要指标 | 维度关联 |
| 总览 | 用户满意度 | 用户 |
|  | 用户数 | 用户 |
|  | 平均响应时间 | 业务 |
|  | 业务操作数 | all |
|  | 业务数&平均响应时间 | 按业务 |
|  | 用户数按国家或省市 | 用户，地域 |
|  | 平均响应时间按国家或省市 | 业务，地域 |
|  | 业务操作数按国家或省市 | 地域 |
| 业务操作分析 | 关键指标（满意度，平均响应时间，业务数，响应时间95分位值） | All或业务 |
|  | 满意度趋势 | 业务 |
|  | 响应时间分布 | 业务 |
|  | 业务操作数&平均响应时间 | 业务 |
|  | 业务操作类型 | 业务 |
|  | 错误率趋势 | All |
|  | 错误业务操作类型 | 业务 |
|  | 业务操作类型 | 业务 |
|  | 满意度 | 业务 |
|  | 业务操作类型的满意度分布 | 业务 |
|  | 业务操作类型的平均响应时间 | 业务 |
| 慢业务追踪 | 慢业务趋势 |  |
|  | 慢业务操作列表 |  |
|  | 业务操作列表 | 业务、用户 |
| 用户操作分析 | 关键指标分析（用户满意度，用户平均响应时间，用户数，平均响应时间95分为值，） | 用户 |
|  | 用户满意度趋势 | 用户 |
|  | 用户满意度分布 | 用户 |
|  | 用户平均响应时间 | 用户 |
|  | 用户响应时间95分位值分布 | 用户 |
|  | 用户列表 | 用户，业务（慢业务个数，错误数） |
| 地域分析 | 按城市业务操作数，用户数， | 地域、用户 |
|  | 按国家业务操作数，用户数 | 地域、用户 |
|  | 地区列表 | 地域、用户，业务（满意度） |
| 终端分析 | 浏览器分析 | 终端 |
|  | 操作系统 | 终端 |
|  | 运营商 | 运营商 |
|  | 分辨率 | 终端 |
|  | Jre参数 | 终端 |
|  | 内存 | 终端 |
|  | 详细列表 | 终端，业务（慢业务，错误率），用户 |
|  |  |  |

## 3、交叉业务字段：

涉及有交叉的查询指标，涉及到的关联关系具体字段如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 总览 | 用户数按国家或省市 | 用户，地域 |  |
| 总览 | 平均响应时间按国家或省市 | 业务，地域 |  |
| 业务操作分析 | 业务操作列表 | 业务、用户 |  |
| 慢业务追踪 | 业务操作列表 | 业务、用户 |  |
| 用户操作分析 | 用户列表 | 用户，业务（慢业务个数，错误数） |  |
| 地域分析 | 地区列表 | 地域、用户，业务（满意度） |  |
| 终端分析 | 详细列表 | 终端，业务（慢业务，错误率），用户 |  |

涉及到的关联关系大多集中在用户和地域，用户和业务。

## 4、各个主要维度基数

各个维度主要基数：

|  |  |  |
| --- | --- | --- |
| 各个类型主维度 | 基数 |  |
| 业务类型（ba） | 770,349 |  |
| 用户数（user\_code） | 135,550 |  |
| 城市 | 427 |  |
| 网络供应商 | 100 |  |
| Jre版本 | 83 |  |
| 浏览器类型 | 13 |  |
| jre内存类型 | 21 |  |
| 操作系统 | 189 |  |
| 分辨率 | 893 |  |

由此可见，业务和用户的基数占用的基数最多。其次是城市和分辨，终端占用的基数较少。

## 5、详细设计子表划分：

业务维度主表：

|  |  |  |  |
| --- | --- | --- | --- |
| 维度分类 | 维度字段 |  |  |
| 租户信息 | tid  tname |  |  |
| 应用信息 | appid  appname  app\_ver  app\_supplier  app\_industry  app\_field  app\_type |  |  |
| 业务信息 | back\_error\_d  back\_error\_is  ba  ba\_code  ba\_tag  probe\_type  op\_type  module  bstotal\_d  txtotal\_d  codetime\_d  nettotal\_d  dbtimems\_d  rendertime\_d  tfs\_up\_d  tfs\_down\_d  nodecode  btnname  nodename |  |  |

该表包含所有业务维度的数据，包括业务主维度及基本维度信息。度量保持原有度量。

用户与地域主表：

数据表的拆分：

在业务数据中，我们的维度可以分为以下几类：

|  |  |  |  |
| --- | --- | --- | --- |
| 维度分类 | 维度字段 |  |  |
| 租户信息 | tid  tname |  |  |
| 应用信息 | appid  appname  app\_ver  app\_supplier  app\_industry  app\_field  app\_type |  |  |
| 业务信息 | back\_error\_d  back\_error\_is  bstotal\_d |  |  |
| 用户信息 | user\_code  user\_name  user\_tag |  |  |
| 地域信息 | city  province  country |  |  |

主要考虑用户与地域信息具有相关，故考虑用户与地域是相关的。该数据会与单独用户的统计信息进行对比

用户与终端主表：

|  |  |  |  |
| --- | --- | --- | --- |
| 维度分类 | 维度字段 |  |  |
| 租户信息 | tid  tname |  |  |
| 应用信息 | appid  appname  app\_ver  app\_supplier  app\_industry  app\_field  app\_type |  |  |
| 业务信息 | back\_error\_d  back\_error\_is  bstotal\_d  txtotal\_d |  |  |
| 用户信息 | user\_code  user\_name |  |  |
| 终端信息 | deviceid  terminal\_type  browser\_type  browser\_ver  os\_type  os\_ver  rqua  manufacturer  model  screen\_size  jre\_ver  jre\_xmx |  |  |
| 网络供应商信息 | carrier  wan |  |  |

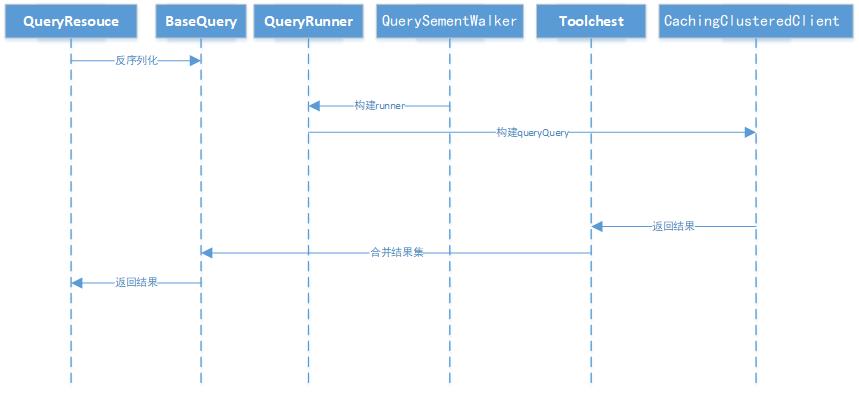
## 6、数据验证

|  |  |  |
| --- | --- | --- |
| 数据 | Ba基数（77w） | 数据对比 |
| 业务数据 | 5435w | 全表 |
| 24,97 w | 业务原始维度 |
| 778w | 只保留ba，rpc\_fla,error\_flg等关键字段 |
| 用户数据 | User基数（13w） |  |
| 786w | 保留用户与业务标识的数据 |
|  |  |  |

## 7、代码修改方案：

1、druid源码解读查询流程：

Druid的查询过程如下：



1、查询关键组件如下：

1、 QueryResouce：查询入口通过jsckson json的objectMapper把参数反序列化成query对象。

2、QuerySegentWalker：根据查询的interval或者segment构建queryRunner。

3、QueryRunner：采用装饰者和调用链模式实现。

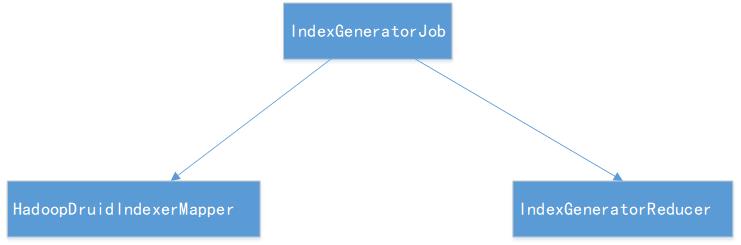
4、QueryToolChest：辅助创建QueryRunner，用户合并结果集。

5、CachingClusteredClient：附加cache信息。

2、储存的关键组件:

Druid采用hadoop-index进行子表合并，实际上执行的是mapreduce任务。

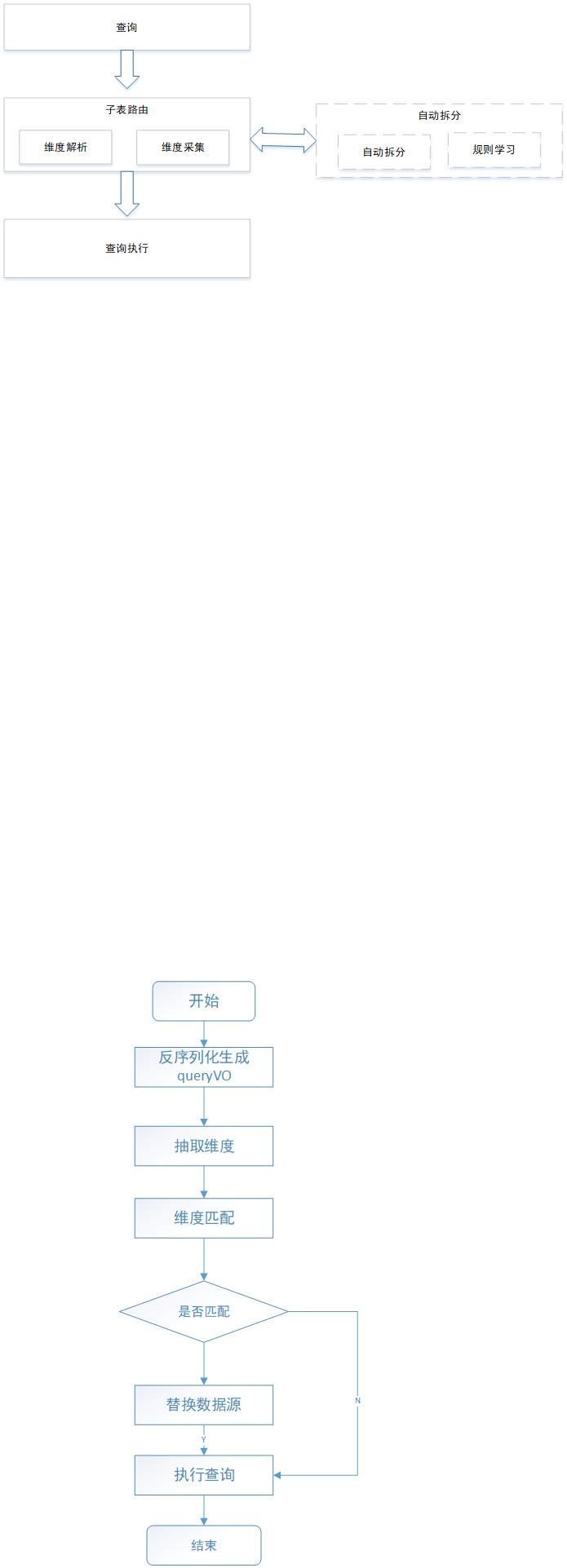
先关的job及map、ruduce的。



设计方案：

概要设计：

该问题主要解决的是采用何种方式来拆分子表及子表拆分的结构。在查询过程中，反序列化的过程时间实际是将查询语句的流转换为queryVO的过程。基本思路为解析反序列化的QueryVO ,通过分析的queryVO 的相关字段确定的匹配规则，确定修改替换database数据源。流程图如下：



该方案面临几个问题：

1、如何确定子表的维度？

2、已何种方式建立子表？

3、如何做到代码无依赖？

1、如何确定子表维度？

子表的维度是通过目前的经验结合业务维度查询建立的。但这种方式存在一定的局限性，同时与分析云沟通，他们也有预聚合这样的需求。在选择kylin等组件时，也担心维度爆炸的问题。所以我们希望建立一种通用的方式，来采集所需要的维度的并通过机器学习，形成自主学习自动创建策略的闭环。

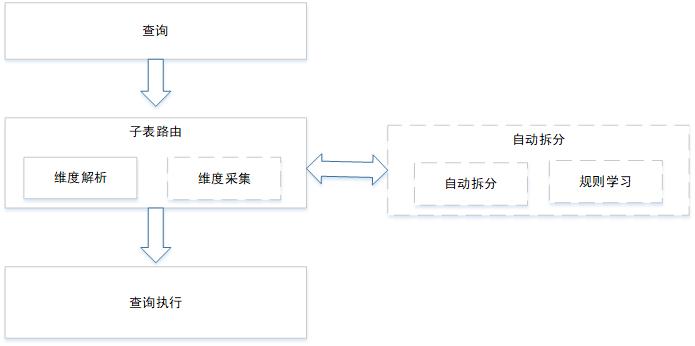
2、已何种方式建立子表？

第一阶段，子表目前需要通过手工的导出，形成新的业务子表。该任务本身为mapreduce任务，故可以通过编写脚本的方式进行执行。之所以采用脚本构建任务的方式，而不是在任务提交时，自动创建是考虑到任务提交需要快速返回结果，而mapreduce任务执行较慢会认为任务执行失败，但可以在提交任务完成后，进行脚本的触发。

3、如何做到代码无依赖？

修改的位置在方法内部，故采用注释的方式无法实现，如果采用字节码增强的方式可以实现，但没太大必要。目前希望采用的是，创建新类，实现需要的方法。在druid源码中加入一行调用代码的方式。

实现的功能框架图如下：



设计的完整流程的分为以下几个部分：

1. 用户提交查询请求。
2. 子表路由进行维度解析、规则匹配，匹配到规则的进行数据源的替换。
3. 将解析的维度，匹配的表及执行时间进行采集，上传进行分析。
4. 当查询语句执行实现超过一定的阈值，则通过机器学习等技术判断子表是否可以在细分，创建新的细分规则，对表进行维度预聚合并创建新表。
5. 按路由子表执行查询。

该问题通过两个阶段完成。

目前阶段只完成子表路由的维护解析部分，其余部分可以之后完成。

维护解析部分方案：

维护抽取：分析queryVO结构与子表路由相关的数据结构：

1、dimFilter结构：

查询的滤器，dimFilter的由filed字段组成，filed中维度字段名称dimension

2、dimensions结构：

需要进行分组统计的维度结构，需要分组统计的最小维度单位。

2、子表匹配规则：

对所需维度求并集，求满足以上条件的维度的最小集合。

3、路由替换：

构建DataSource类，通过名称构建该类型。注意需要与该查询构建的类型一致。