

架构师
ARCHITECT

特刊

Apache Kylin实践

(第二期)

SPECIAL ISSUE
December, 2017

架构师特刊



序言

Apache Kylin 自 2014 年 10 月开源以来，已经走入了第四个年头，在过去的几年中，Kylin 项目和社区的发展，离不开每一位朋友的贡献，离不开每一个用户的信任、支持和反馈，甚至更多是贡献。今天非常高兴看到 InfoQ 即将出版第二本 Apache Kylin 的电子书，很荣幸能够在此分享一些这一年的感悟。

这一年，是 Apache Kylin 高速发展的一年，2017 年 4 月 30 日，Apache Kylin 的下一代 v2.0 正式发布，这是 Kylin 自诞生以来最重要的版本之一，引入了 Spark Cubing，雪花模型支持，支持云存储框架等，为 Kylin 的后续发展打下了坚实的基础。也使得 Kylin 的应用面不断扩开，特别是 Spark 的引入，大大加快了整个项目的演进过程。8 月 17 日，v2.1 发布，修复了超过 100 个 Bug，并扩大了数据输入源及 SQL 的支持，从而使得更多的数据可以接入 Kylin 以加速分析过程。11 月 3 日，最新的 v2.2

发布，在稳定性、安全性等方面做了重大改进。这几次 release 中，越来越多的贡献者踊跃出来，社区吸收了多为来自中国、欧洲等的贡献者，也非常欣喜，有越来越多的核心功能由不同的公司和团队贡献进来。

这一年，也是社区高速发展的一年，我们非常欣喜的发现，在各个线下活动、Meetup 及会议上，越来越多的用户告诉我们在使用 Kylin，越来越多的朋友在分享他们使用 Kylin 的经验、教训及建议等。这本电子书，汇聚了来自百度外卖、美团、唯品会、链家、Strikingly 等多个案例，非常值得每位朋友参考。目前，据不完全统计，Kylin 在全球的用户已经超过 500 多家，包括来自中国的大量中大型互联网公司、金融企业，来自美国、欧洲、日本等多个不同的国家和地区，包括 Yahoo! Japan，Adobe 等。

这一年，也是中国开源力量不断崛起的一年，作为中国第一个贡献到 Apache 软件基金会的顶级开源项目，第一个来自中国的 Apache 顶级项目副总裁，我及我的团队受邀在各个公司举行了多次分享，将我们在孵化、毕业以及运营社区的经验不断分享给各个开源爱好者。我们深刻感受到了来自中国开发者社区的强烈的开源热情和能力。

今天，有来自 eBay 中国研发中心的的 Eagle 和 Griffin、来自华为的 CarbonData、来自阿里的 RocketMQ 及 Weex 等都相继贡献到 Apache 软件基金会并孵化，有几个已经或正在毕业中。有更多的来自其他公司和团队的项目目前正在积极准备中。个人非常荣幸能够在各个项目的初期等各个阶段有所贡献，目前也是多个 Apache 孵化项目的导师（RocketMQ，Weex，Superset 等）。期待更多来自中国的贡献，期待在国际舞台上，我们的技术和实力，进一步得以展现。

—— 韩卿 (Luke Han)
联合创始人 &CEO, Kyligence Inc.
联合创始人 &VP of Apache Kylin
Apache 软件基金会 (ASF) 会员

北京 | 伦敦 | 纽约 | 旧金山 | 圣保罗 | 上海 | 东京



全球软件开发大会2018

主办方 **Geekbang**. InfoQ
极客邦科技

[北京站]

北京·国际会议中心

会议：2018年4月20–22日 培训：2018年4月18–19日

纵览20大热门专题

8折进行时

现在报名每张立减1360元

团购享受更多优惠 截至2017年3月11日

大会官网：www.qconbeijing.com
访问官网获取更多前沿技术趋势

如有任何问题，欢迎咨询
电话：151 1001 9061
微信：qcon-0410



ArchSummit

全球架构师峰会

2018 · 深圳站

从2012年开始算起，InfoQ已经举办了9场ArchSummit全球架构师峰会，有来自Microsoft、Google、Facebook、Twitter、LinkedIn、阿里巴巴、腾讯、百度等技术专家分享过他们的实践经验，至今累计已经为中国技术人奉上了近千场精彩演讲。

● 2017.07.07–08 深圳站

how to use sagas to maintain data consistency in a microservice architecture

--Chris Richardson, *POJOs in Action* 作者，知名微服务专家

● 2017.12.08–11 北京站

《创新是人类的自信》

--王坚博士，阿里巴巴集团技术委员会主席

● 2018.7.06–09 深圳站

限时**7折报名中**，名额有限，快快抢购。

7折报名中

名额有限，快快抢购

华南地区架构领域最有影响力的会议，届时有哪些专题和演讲，敬请扫描右方二维码浏览官网。



目录

- 07 Apache Kylin 在美团点评的应用
- 16 美团 Apache Kylin 精确去重指标优化历程
- 22 唯品会海量实时 OLAP 分析技术升级之路
- 40 Apache Kylin 在百度外卖流量分析平台的应用与实践
- 64 Apache Kylin 在链家 GAIA 大数据平台中的实践
- 72 AWS 上 Apache Kylin 调度系统的设计



AI前线 (ID: ai-front)

关注技术落地
探寻 AI 应用场景

每周一节技术分享公开课
助力你全面拥抱人工智能技术

InfoQ



关注AI前线公众号

Apache Kylin 在美团点评的应用

作者 高大月



背景

美团点评的OLAP需求大体分为两类：

- 即席查询：指用户通过手写SQL来完成一些临时的数据分析需求。这类需求的SQL形式多变、逻辑复杂，对响应时间没有严格的要求。
- 固化查询：指对一些固化下来的取数、看数的需求，通过数据产品的形式提供给用户，从而提高数据分析和运营的效率。这类需求的SQL有固定的模式，对响应时间有比较高的要求。

我们针对即席查询提供了Hive和Presto两个引擎。而固化查询由于需

要秒级响应，很长一段时间都是通过先在数仓对数据做预聚合，再将聚合表导入MySQL提供查询实现的。但是随着公司业务数据量和复杂度的不断提升，从2015年开始，这个方案出现了三个比较突出的问题：

- 随着维度的不断增加，在数仓中维护各种维度组合的聚合表的成本越来越高，数据开发效率明显下降；
- 数据量超过千万行后，MySQL的导入和查询变得非常慢，经常把MySQL搞崩，DBA的抱怨很大；
- 由于大数据平台缺乏更高效率的查询引擎，查询需求都跑在Hive/Presto上，导致集群的计算压力大，跟不上业务需求的增长。

为了解决这些痛点，我们在2015年末开始调研更高效率的OLAP引擎，寻找固化查询场景的解决方案。

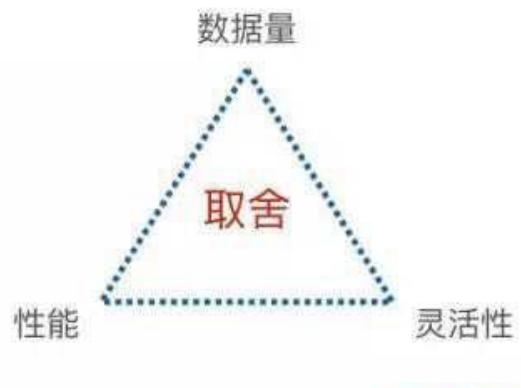
为什么选择 Kylin

在调研了市面上主流的开源OLAP引擎后，我们发现，目前还没有一个系统能够满足各种场景的查询需求。其本质原因是，没有一个系统能同时在数据量、性能、和灵活性三个方面做到完美，每个系统在设计时都需要在这三者间做出取舍。

数据量：要支持多大的数据量？
千万行、上亿行、还是十亿、百亿或更大

性能：需要怎样的性能表现？
毫秒级、秒级、还是分钟或小时级？

灵活性：需要支持什么样的查询需求？
明细、聚合、还是Both？
是否需要支持Join或者子查询？
离线还是实时？



例如

- MPP架构的系统（Presto/Impala/SparkSQL/Drill等）有很好的数据量和灵活性支持，但是对响应时间是没有保证的。当数据量和

计算复杂度增加后，响应时间会变慢，从秒级到分钟级，甚至小时级都有可能。

- 搜索引擎架构的系统（Elasticsearch等）相对比MPP系统，在入库时将数据转换为倒排索引，采用Scatter-Gather计算模型，牺牲了灵活性换取很好的性能，在搜索类查询上能做到亚秒级响应。但是对于扫描聚合为主的查询，随着处理数据量的增加，响应时间也会退化到分钟级。
- 预计算系统（Druid/Kylin等）则在入库时对数据进行预聚合，进一步牺牲灵活性换取性能，以实现对超大数据集的秒级响应。

有了这套框架，我们不难结合美团点评的自身需求特点，选择合适的OLAP引擎。

美团点评的OLAP需求特点

- 数据以离线N+1生产为主，数据量在千万到百亿级别
- 20左右个维度、50左右个指标、需要任意维度组合
- 指标中包含大量的去重指标，结果要求精确
- 面向内部PM/运营/销售的数据产品：响应时间要求3秒内
- 最好提供SQL接口

可以看出，我们对数据量和性能的要求是比较高的。MPP和搜索引擎系统无法满足超大数据集下的性能要求，因此很自然地会考虑預计算系统。而Druid主要面向的是实时Timeseries数据，我们虽然也有类似的场景，但主流的分析还是面向数仓中按天生产的结构化表，因此Kylin的MOLAP Cube方案是最适合我们场景的引擎。

Kylin 的使用现状

2016年初，我们开始向各个业务线推广基于Kylin的解决方案。经过一年的努力，Kylin已经应用到了美团点评的几乎所有主要业务线上，并且

在外卖、酒旅等数个业务线得到了大规模的使用，Kylin已经成为了这些业务的首选OLAP引擎。

截至16年底，生产环境共有214个Cube，包含的数据总行数为2853亿行，Cube在HBase中的存储有59TB。日查询次数超过了50万次，TP50查询时延87ms，TP99时延1266ms，很好地满足了我们对性能的要求。

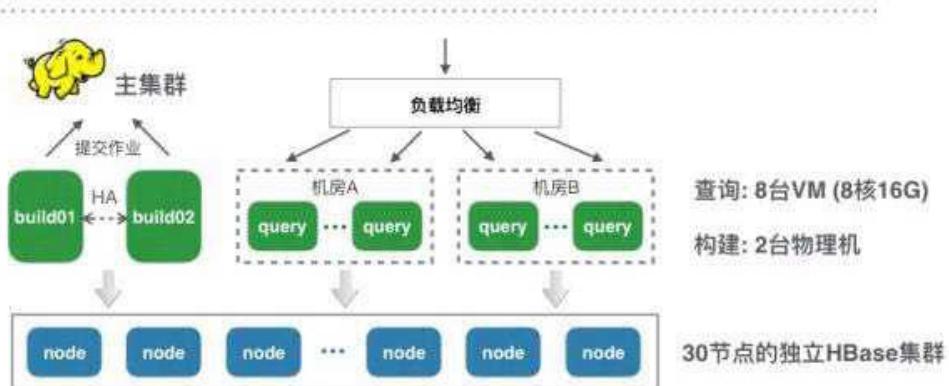
Kylin在美团点评的使用现状



为了支持这些需求，我们的线上环境包含一个30节点的Kylin专属HBase集群，2台用于Cube构建的物理机，和8台8核16G的VM用作Kylin的查询机。Cube的构建是运行在主计算集群的MR作业，各业务线的构建任务拆分到了他们各自的资源队列上。

由于Kylin对外是REST接口，我们接入了公司统一的http服务治理框架来实现负载均衡和平滑重启。

部署结构



“维度爆炸”问题在实践中是可解的

提到MOLAP Cube方案，很多没接触过Kylin的人会担心“维度爆炸”的问题，即每增加一个维度，由于维度组合数翻倍，Cube的计算和存储量也会成倍增长。我们起初其实也有同样的担心，但调研和使用Kylin一阵子后发现，这个问题在实践中并没有想象的严重。这主要是因为

- Kylin支持Partial Cube，不需要对所有维度组合都进行预计算；
- 实际业务中，维度之间往往存在衍生关系，而Kylin可以把衍生维度的计算从预计算推迟到查询处理阶段。

应对“维度爆炸”问题

- 通过「Partial Cube」减少预计算维度组合数
 - Aggregation Group：显式控制哪些维度组合(cuboid)需要预计算
 - 没有预计算的Cuboid可以Runtime计算
- 通过「衍生维度」减少Cube中的维度个数
 - 事实表上的衍生维度：ExtendedColumn
 - 维表上的衍生维度：Derived Dimension

事实表上的衍生维度

- 业务中维度经常成对出现，例如 [CITY_ID, CITY_NAME]
- 通常需要根据ID列过滤，但显示对应的NAME列
- 如果将ID和NAME都作为维度，维度个数会翻倍
- 解决方案：**将NAME作为ID列的Extended Column度量**

The diagram illustrates the optimization of a fact table. On the left, labeled "优化前: 4个维度" (Before Optimization: 4 dimensions), is a table with columns: CityID, CityName, CateID, CateName, Revenue. The data rows are: (10, 北京, 100, 火锅, 150,000) and (20, 上海, 200, 本帮菜, 120,000). An arrow points to the right, labeled "优化后: 2个维度 (CityID, CateID)" (After Optimization: 2 dimensions (CityID, CateID)). The optimized table has columns: CityID, CateID, CityName, CateName, Revenue. The data rows remain the same. Curved arrows above the table columns indicate that CityName and CateName are now represented as extended columns derived from their respective ID columns.

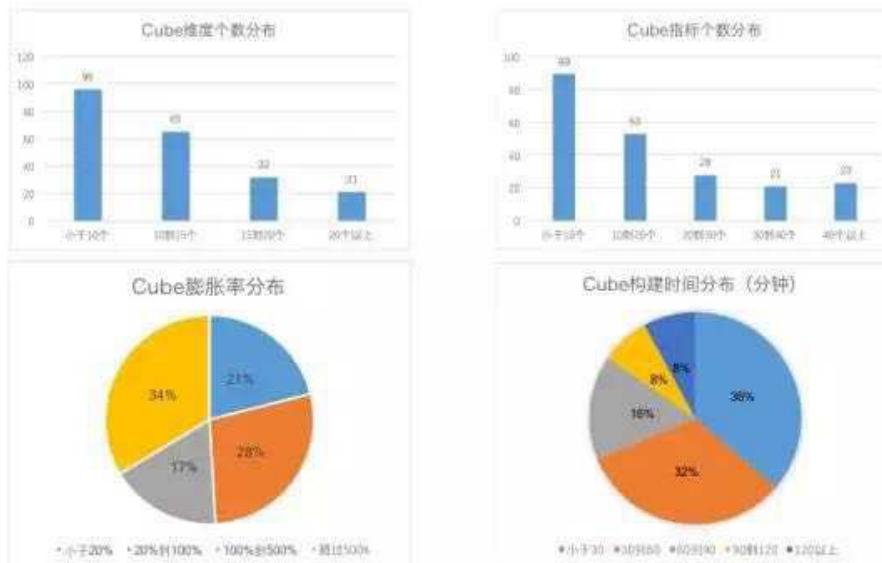
CityID	CityName	CateID	CateName	Revenue
10	北京	100	火锅	150,000
20	上海	200	本帮菜	120,000

CityID	CateID	CityName	CateName	Revenue
10	100	北京	火锅	150,000
20	200	上海	本帮菜	120,000

以事实表上的衍生维度为例，我们业务中的很多维度都是(ID, NAME)成对出现的。查询时需要对ID列进行过滤，但显示时只需要取对应的NAME列。如果把这两列都作为维度，维度个数会翻倍。而在Kylin中，

可以把NAME作为ID列的extendedcolumn指标，这样Cube中的维度个数就减半了。

下面分享一些我们线上Cube的统计数据。



可以看到，采用衍生维度后，90%的场景可以把Cube中的维度个数（Rowkey列数）控制在20个以内。指标个数呈现长尾分布，小于10个指标的Cube是最多的，不过也有近一半的Cube指标数超过20。总共有382个去重指标，占到了总指标数的10%，绝大多数都是精确去重指标。49%的Cube膨胀率小于100%，即Cube存储量不超过上游Hive表。68%的Cube能够在1小时内完成构建，92%在2小时内完成构建。

美团外卖的使用案例

下面分享一下Kylin在美团外卖的使用案例，感谢外卖的同事 靳国卫和惠明 提供材料。

外卖数据业务对交互式的OLAP分析有着很强的需求。在使用Kylin以前，采用的是在Hive中开发聚合表再导入MySQL的方案。随着业务数据量高速增长和需求的不断升级，这套方案遇到了开头提到的查询效率和开发效率的双重问题。

外卖数据开发面临的问题

· 业务层面

- 多维度，多角度精细化分析数据
- 查询速度快，数据精确去重
- 数据应用及时就绪

· 开发层面

- 提前预计算各个维度组合数据，对数据进行精确去重
- ETL开发量大，包含事实层、聚合层、主题层ETL
- 维护成本高，维度变化后ETL改动量大
- 计算资源高，数据应用就绪延迟
- 数据结果集大，存储空间和数据传输遇到困难



在使用Kylin后，除了查询性能的显著提升，外卖的数据开发方式发生了很大的改变。原来需要做繁琐的聚合层和主题层数据，现在只需要把重点放到基础数据的建设上，预计算的工作交给Kylin就行了。在对同一个需求同时采用老方案和Kylin方案实施后发现，使用Kylin后的数据开发效率提升了3倍。

Kylin在外卖数仓架构的位置



下面是一个对流量数据应用Kylin的具体案例。我们在Kylin 1.5.3版本添加了全局字典，实现了上亿基数、任意类型字段（例如设备ID）的精确去重计数，把Kylin的使用场景扩宽到了流量数据。

接入Kylin前后对比

对比项	接入Kylin之前	接入Kylin之后	对比结果
数据开发流程	需要开发事实层、聚合层、主题层和维度表ETL	只需要开发事实层、维度表、配置Kylin	1. ETL开发效率提升3倍 2. 维护成本只需要改Kylin配置，不需要修改ETL
存储方式	Hive中预计算结果推送到MySQL	存储在HBase	1. 支撑结果数据量级增大 2. 都使用JDBC查询数据 3. 结果数据传输提高1倍
多维数据处理	Hive中的cube或者grouping sets	Kylin UI 配置	维度变化只需要改Kylin配置，不需要修改ETL
精确去重	Hive中count distinct	基于bitmap的精确去重	1. 节省计算资源 2. 支持跨天去重
转化率&用户留存	Hive中通过join过滤	通过UDAF处理bitmap	1. 开发效率提升 2. 不需要预计算，节省资源



案例：用户行为分析应用

需求背景

随着外卖业务的不断演进，产品团队需要更细粒度的用户行为分析数据，去评估产品策略的效果，支持产品策略的制订

服务团队

用户产品团队，线上运营团队，数据团队

功能介绍

用户行为明细查询，用户事件分析，事件转化分析，事件轨迹分析



案例：用户行为分析应用



Aggregation Groups

Visit [Aggregation group](#) for more about aggregation group.

ID Aggregation Groups

1	Includes	[EVENT_ID, EVENT_NAME, CITY_ID, CITY_NAME, CLIENT_ID, DEVICE_MODE, APP_VERSION, CHANNEL_NAME, DIM1, DIM2, DT]
	Mandatory Dimensions	[EVENT_ID, EVENT_NAME, DT]
	Hierarchy Dimensions	[CLIENT_ID, APP_VERSION]

Joint Dimension

[CITY_ID, CITY_NAME]



案例：用户行为分析应用

- 每日新增数据量：1.5亿行
- 包含20个业务维度和1个UV精确去重指标
- 膨胀率（Cube大小/Hive表大小）：5.59%
- 最近7天构建时间中位数：78分钟
- 查询效率：TP99 < 3秒

平台化经验与思考

一个开源项目从run起来到真正作为平台化的服务提供出去，中间会遇到很多的挑战和问题需要解决。下面是我们总结的一些经验，在这里分享给大家，也欢迎同行们和我们一起探讨。

平台化经验 1/3

- 从实际业务需求出发，避免闭门造车
 - KYLIN-1186 支持INT类型的精确去重
 - KYLIN-1705 全局字典，支持全类型的精确去重
 - KYLIN-1732 支持窗口函数
 - KYLIN-2088 基于bitmap计算留存的UDAF

平台化经验 2/3

- 推动命名规范、接入流程标准化
- 坚持和社区一起发展
 - 不搞内部版本，所有bugfix和改进全部合入社区
 - 经历3次大版本升级，享受社区最近成果
 - 团队培养了2位Kylin PMC和1位Committer
 - 整理文档、梳理使用场景和案例、组织培训

平台化经验 3/3

- 保障服务稳定性
- 建立监控报警体系，积累问题排查工具 ([KYLIN-1908](#) / [KYLIN-2105](#))
- 系统架构避免单点 ([KYLIN-1910](#) / [KYLIN-2006](#))
- 大查询限制与报警
 - 查询执行50秒超时退出 ([KYLIN-2377](#))
 - HBase协处理器执行20秒超时退出 ([KYLIN-2079](#))
- Hive表结构变更支持 ([KYLIN-2012](#))



未来规划

- **更稳定的服务**: 全局字典构建MR化，查询内存管理
- **更易用的平台**: Cube生命周期管理，ETL版CubeBuilder
- **更丰富的场景**: 明细查询，实时Cube
- **更优异的性能**: 优化精确去重和大结果集查询性能

作者简介

高大月，美团点评工程师，Apache Kylin PMC成员，目前主要在美团点评数据平台负责OLAP查询引擎的建设

美团 Apache Kylin 精确去重指标优化历程

作者 康凯森



问题背景

本文记录了我将Apache Kylin超高基数的精确去重指标查询提速数十倍的过程，大家有任何建议或者疑问欢迎讨论。

某业务方的cube有12个维度，35个指标，其中13个是精确去重指标，并且有一半以上的精确去重指标单天基数在千万级别，cube单天数据量1.5亿行左右。业务方一个结果仅有21行的精确去重查询竟然耗时12秒多，其中HBase端耗时6秒多，Kylin的query server端耗时5秒多：

```
SELECT A, B, count(distinct uuid), FROM table WHERE dt = 17150 GROUP BY A, B
```

精确去重指标已经在美团点评生产环境大规模使用，我印象中精确去

重的查询的确比普通的Sum指标慢一点，但也挺快的。这个查询慢的如此离谱，我就决定分析一下，这个查询到底慢在哪。

优化1 将精确去重指标拆分HBase列族

我首先确认了这个cube的维度设计是合理的，这个查询也精准匹配了cuboid，并且在HBase端也只扫描了21行数据。

那么问题来了，为什么在HBase端只扫描21行数据却需要6秒多？一个显而易见的原因是Kylin的精确去重指标是用bitmap存储的明细数据，而这个cube有13个精确去重指标，并且基数都很大。我从两方面验证了这个猜想：

1. 同样SQL的查询Sum指标只需要120毫秒，并且HBase端Scan仅需2毫秒。
2. 我用HBase HFile命令行工具查看并计算出HFile中单个KeyValue的大小，发现普通指标的列族中每个KeyValue平均大小是29B，精确去重指标列族的每个KeyValue平均大小却有37M。

所以我第一个优化就是将精确去重指标拆分到多个HBase列族，优化后的效果十分明显。查询时间从12秒多减少到5.7秒左右，HBase端耗时从6秒多减少到1.3秒左右，不过query server耗时依旧有4.5秒多。

优化2 移除不必要的toString避免bitmap deserialize

Kylin的query server耗时依旧有4.5秒多，我猜测肯定还是和bitmap比较大有关，但是为什么bitmap大会导致如此耗时呢？为了分析query server端查询处理的时间到底花在了哪，我利用Java Mission Control进行了性能分析。

JMC分析很简单，在Kylin的启动进程中增加以下参数：

```
-XX:+UnlockCommercialFeatures -XX:+FlightRecorder  
-XX:+UnlockDiagnosticVMOptions -XX:+DebugNonSafepoints  
-XX:StartFlightRecording=delay=20s,duration=300s,name=kylin,filename=myrecording.  
jfr,settings=profile
```

获得myrecording.jfr文件后，我们在本机执行jmc命令，然后打开

myrecording.jfr文件就可以进行性能分析。从jmc的热点代码图中我们发现，耗时最多的代码竟然是一个毫无意义的toString。去掉这个toString之后，query server的耗时直接减少了1秒多。

优化3 获取bitmap的字节长度时避免deserialize

在优化2去掉无意义的toString之后，热点代码已经变成了对bitmap的deserialize。不过bitmap的deserialize共有两处，一处是bitmap本身的deserialize，一处是在获取bitmap的字节长度时。于是很自然的想法就是在获取bitmap的字节长度时避免deserialize bitmap，当时有两种思路：

1. 在serialize bitmap时就写入bitmap的字节长度。
2. 在MutableRoaringBitmap序列化的头信息中获取bitmap的字节长度。(Kylin的精确去重使用的bitmap是RoaringBitmap)

我最终确认思路2不可行，采用了思路1。

思路1中一个显然的问题就是如何保证向前兼容，我向前兼容的方法就是根据MutableRoaringBitmap deserialize时的cookie头信息来确认版本，并在新的serialize方式中写入了版本号，便于之后序列化方式的更新和向前兼容。

经过这个优化后，Kylin query server端的耗时再次减少1秒多。

优化4 无需上卷聚合的精确去重查询优化

从精确去重指标在美团点评大规模使用以来，我们发现部分用户的应用场景并没有跨segment上卷聚合的需求，即只需要查询单天的去重值，或是每次全量构建的cube，也无需跨segment上卷聚合。所以我们希望对无需上卷聚合的精确去重查询进行优化，当时我考虑了两种可行的方案：

方案1：精确去重指标新增一种返回类型

一个极端的做法是对无需跨segment上卷聚合的精确去重查询，我们只存储最终的去重值。

优点：

1. 存储成本会极大降低。

2. 查询速度会明显提高。

缺点：

1. 无法支持上卷聚合，与Kylin指标的设计原则不符合。
2. 无法支持segment的merge，因为要进行merge必须要存储明细的bitmap。
3. 新增一种返回类型，对不清楚的用户可能会有误导。
4. 查询需要上卷聚合时直接报错，用户体验不好，尽管使用这种返回类型的前提是无需上聚合卷。

实现难点：

如果能够接受以上缺点，实现成本并不高，目前没有想到明显的难点。

方案 2：serialize bitmap 的同时写入 distinct count 值。

优点：

1. 对用户无影响。
2. 符合现在Kylin指标和查询的设计。

缺点：

1. 存储依然需要存储明细的bitmap。
2. 查询速度提升有限，因为即使不进行任何bitmap serialize，bitmap本身太大也会导致HBase scan，网络传输等过程变慢。

实现难点：

如何根据是否需要上卷聚合来确定是否需要serialize bitmap？

解决过程：

我开始的思路是从查询过程入手，确认在整个查询过程中，哪些地方需要进行上卷聚合。为此，我仔细阅读了Kylin query server端的查询代码，HBase Coprocessor端的查询代码，Calcite的example例子。发现在HBase端，Kylin query server端，cube build时都有可能需要指标的聚合。

此时我又意识到一个问题：即使我清晰的知道了何时需要聚合，我又该如何把是否聚合的标记传递到精确去重的反序列方法中呢？现在精确去

重的deserialize方法参数只有一个ByteBuffer，如果加参数，就要改变整个kylin指标deserialize的接口，这将会影响所有指标类型，并会造成大范围的改动。所以我把这个思路放弃了。

后来我“灵光一闪”，想到既然我的目标是优化无需上卷的精确去重指标，那为什么还要费劲去deserialize出整个bitmap呢，我只要个distinct count值不就完了。所以我的目标就集中在BitmapCounter本身的deserialize上，并联想到我最近提升了Kylin前端加载速度十倍以上的核心思想：延迟加载，就改变了BitmapCounter的deserialize方法，默认只读出distinct count值，不进行bitmap的deserialize，并将那个buffer保留，等到的确需要上卷聚合的时候再根据buffer deserialize出bitmap。

当然，这个思路可行有一个前提，就是buffer内存拷贝的开销是远小于bitmap deserialize的开销，庆幸的是事实的确如此。最终经过这个优化，对于无需上卷聚合的精确去重查询，查询速度也有了较大提升。显然，如你所见，这个优化加速查询的同时加大了需要上卷聚合的精确去重查询的内存开销。我的想法是首先对于超大数据集并且需要上卷的精确去重查询，用户在分析查询时返回的结果行数应该不会太多，其次我们需要做好query server端的内存控制。

总结

我通过总共4个优化，在向前兼容的前提下，后端仅通过100多行的代码改动，对Kylin超高基数的精确去重指标查询有了明显提升，测试中最明显的查询有50倍左右的提升。

作者简介

康凯森，美团点评大数据工程师，Apache Kylin committer，目前主要负责Apache Kylin在美团点评的平台化建设。

唯品会海量实时 OLAP 分析技术升级之路

作者 谢麟炯



海量数据实时 OLAP 场景的困境

大数据

首先来看一下我们在最初几年遇到的问题。第一就是大数据，听起来好像蛮无聊的，但大数据到底是指什么呢？最主要的问题就是数据大，唯品会在这几年快速发展，用户流量数据从刚开始的几百万、几千万发展到现在的几个亿，呈现了100倍以上的增长。

对我们而言，所谓的大数据就是数据量的快速膨胀，带来的问题最主要的就是传统RDBMS无法满足存储的需求，继而是计算的需求，我们的

挑战便是如何克服这个问题。

慢查询

第二个问题是慢查询，有两个方面：一是OLAP查询的速度变慢；二是ETL数据处理效率降低。

分析下这两个问题：首先，用户使用OLAP分析系统时会有这样的预期，当我点击查询按钮时希望所有的数据能够秒出，而不是我抽身去泡个茶，回来一看数据才跑了10%，这是无法接受的。由于数据量大，我们也可以选择预先计算好，当用户查询时直接从计算结果中找到对应的值返回，那么查询就是秒出的。数据量大对预计算而言也有同样的问题，就是ETL的性能也下降了，本来准备这个数据可能只需40分钟或一个小时，现在数据量翻了一百倍，需要三个小时，这时候数据分析师上班时就会抱怨数据没有准备好，得等到中午分析之类的，会听到来自同事不断的抱怨。

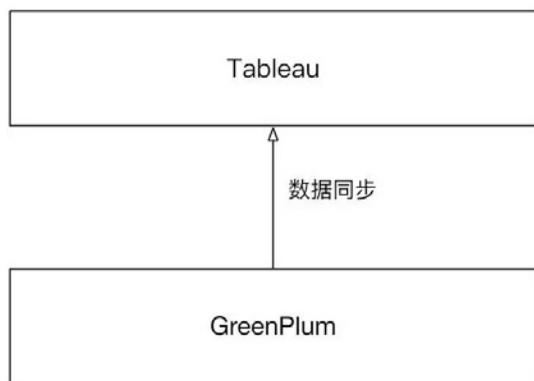
长迭代

数据量变大带来的第三个毛病，就是开发周期变长。两个角度：第一，新业务上线，用户会说我能不能在这个新的角度上线前，看看历史数据，要看一年的，这时就要刷数据了。刷数据这件事情大家知道，每次刷头都很大，花的时间很长。旧业务也一样，加新的指标，没有历史趋势也不行，也要刷数据，开发就不断地刷数据。因为数据量大，刷数据的时间非常长，数据验证也需要花很多的时间，慢慢的，开发周期变慢，业务很急躁，觉得不就是加个字段吗，怎么这么慢。这样一来，数据的迭代长，周期变慢，都让业务部门对大数据业务提出很多的质疑，我们需要改进来解决这些问题。

业务部门的想法是，不管你是什么业务，不管现在用的是什么方法，他们只关心三点：第一，提的需求要很快满足；第二，数据要很快准备好；第三，数据准备好之后，当我来做分析时数据能够很快地返回。业务要的是快快快，但现在的能力是慢慢慢，为此，我们急需解决业务部门的需求和现状之间的冲突。

唯品会大数据实时 OLAP 升级过程

第0阶段



- 优点
 - 使用商业敏捷BI工具，快速满足OLAP报表需求
 - GreenPlum MPP数据库作为数据仓库的存储、计算介质
 - 数据仓库和OLAP分析混用同一个数据库实现。

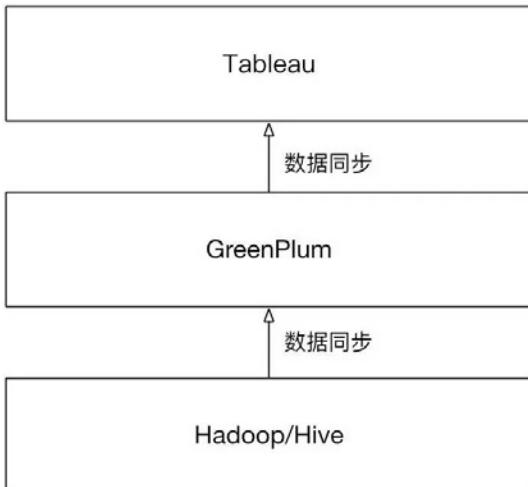
- 缺点：
 - 海量日志数据的接入使得存储和计算资源迅速枯竭，但RDBMS数据库水平扩展能力有限

这是我们的初始状态，架构比较简单。底层的计算、存储和OLAP分析用MDB的数据仓库解决的，上层用Tableau的BI工具，开发速度比较快，同时有数据可视化效果，业务部门非常认可。GreenPlum是MPP的方案，它的高并发查询非常适合我们这种OLAP的查询，性能非常好。所以我们在这个阶段，把GreenPlum作为数据仓库和OLAP混用的实现。

这样一个架构其实是一个通用的架构，像Tableau可以轻易被替换，GreenPlum也可以替换成Oracle之类的，这样一个常用的工具、一个架构，其实满足了部分的需求，但也有个问题，就是像GreenPlum这样的RDBMS数据库，在面对海量的数据写入时存储和计算的资源快速地枯竭了，GreenPlum的水平扩展有限，所以同样碰到了大数据的问题。

第1阶段

所以很快我们就进入了第一阶段。这个阶段，我们引入了Hadoop/Hive，所有的计算结果做预计算之后，会同步到GreenPlum里面，接下去一样，用GreenPlum去做分析。OLAP讲聚合讲的Ad-hoc，继续由



- 优点:
 - 根据服务提供的目标不同，分拆数据仓库和OLAP分析库，Hadoop/Hive负责数据仓库部分，GreenPlum专注负责OLAP部分
- 缺点:
 - 数据需要在两个不同的DB之间同步，数据冗余且可能存在不一致性
 - BI工具没有进化，但用户的需求正在不断进化

GreenPlum承载，数据仓库讲明细数据讲Batch，就交给专为批量而生的Hive来做，这样就能把OLAP和数据仓库这两个场景用两个不一样技术栈分开。这样一个技术方案解决了数据量大的问题，ETL批量就不会说跑不动或者数据没法存储。

但问题是增加了新的同步机制，需要在两个不同的DB之间同步数据。同步数据最显而易见的问题就是除了数据冗余外，如果数据不同步怎么办？比如ETL开发在Hadoop上更新，但没有同步到GreenPlum上，用户会发现数据还是错误的。第二，对用户来说，当他去做OLAP分析时，Tableau是更适合做报表的工具，随着我们业务的扩展和数据驱动不断的深入，业务不管分析师还是商务、运营、市场，他们会越来越多地想组合不同的指标和维度去观察自己的数据，找自己运营的分析点。传统的Tableau报表已经不能满足他们。

我们需要一个新的BI解决方案

对我们来说数据不同步还可以解决，毕竟是偶然发生的，处理一下就可以了。但是BI工具有很大的问题，不能满足业务已经进化的需求。所以我们需要一个新的BI解决方案：

首先它要足够灵活，不能发布之后用户什么都不能做，只能看，我们希望它的维度和指标可以快速整合。

第二，门槛要低，我们不可能希望业务像BI工程师学习它的开发是怎么做的，所以它要入门非常简单。其次，要能够用语言描述自己的需求，而不是用SQL，让商务这种感性思维的人学SQL简直是不可能的，所以要能用语言描述他们自己想要什么。

第三就是开发周期短，业务想看什么，所有的数据都需要提需求，需求分析，排期实施，提变更又要排期实施，这时候虽然说业务发展不是一天一变，但很多业务试错的时间非常快，数据开发出来黄花菜都凉了。所以希望有一个新的BI方案解决这三个问题。

我们看了一下市面上的商业工具并不适合，并且这样灵活的方案需要我们有更强的掌控性，于是我们就开始走向了自研的道路。

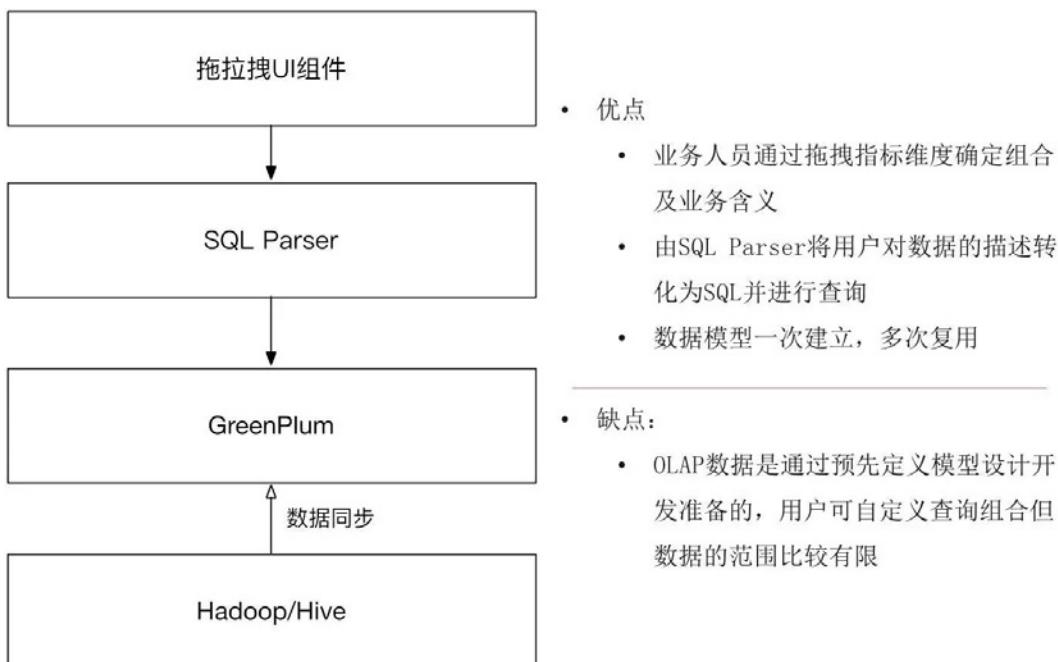
第2阶段

The screenshot shows a user interface for a self-service analysis platform. At the top, there's a title bar with the text '品类UV和PV'. Below it is a search bar and a '维度分析' (Dimension Analysis) section. The main workspace has a '拖放维度到这里, 增加列' (Drag dimensions here, add columns) placeholder. Underneath are sections for 'E 维度' (Dimensions), '访问时间' (Access Time), 'E 度量' (Metrics), and a '查询' (Query) section with dropdowns for '一级品类名称' (Primary Category Name), '二级品类名称' (Secondary Category Name), 'UV(估算)' (Estimated UV), and 'PV'. To the right, there's a '图表类型' (Chart Type) sidebar with options for '数据表' (Data Table), '折线图' (Line Chart), '柱形图' (Bar Chart), and '条形图' (Stacked Bar Chart). At the bottom right, there are buttons for '保存' (Save), '另存为' (Save As), '分享' (Share), and '导出' (Export).

我们进入了OLAP分析的第二个阶段，这时前端开发了一个产品叫自助分析平台，这个平台上用户可以通过拖拉拽把左边的维度指标自己组合拖到上面，组成自己想看的结果。结果查询出来后可以用表格也可以图形进行展示，包括折线、柱状、条形图，这里面所有的分析结果都是可保存、可分享、可下载的。

利用这样的工具可以帮助分析师或者业务人员更好地自由的组合刚才

我们所说的一切，并且灵活性、门槛低的问题其实也都迎刃而解了。而且像这样拖拉拽是非常容易学习的，只要去学习怎么把业务逻辑转化成一个数据的逻辑描述，搞懂要怎么转化成什么形式，行里面显示什么，列显示什么，度量是什么就可以了，虽然有一点的学习曲线，但比起学习完整的BI工具，门槛降低了很多。



前端是这样的产品，后端也要跟着它一起变。首先前端是一个拖拉拽的UI组件，这个组件意味着用传统的选择SQL，直接形成报表的方式已经不可行了，因为所有的一切不管是维度指标都是用户自己组合的，所以我们需要一个SQL Parser帮助用户把它的数据的描述转化成SQL，还要进行性能的调优，保证以一个比较高的性能反馈数据。

所以我们就开发了一个SQL Parser用来承接组件生成的数据结构，同时用SQL Parser直接去OLAP数据。还是通过预计算的方式，把我们需要的指标维度算好同步到SQL Parser。这样的模型一旦建立，可以多次复用。

但我们知道这个计算方案有几个明显的缺点：第一，所有的数据必须经过计算，计算范围之外的不能组合；第二，还是有数据同步的问题，第

三是什么？其实预算算的时候大家会经常发现我们认为这些组合是有效的，用户可能不会查，但不去查这次计算就浪费掉了。是不是有更好的办法去解决这种现状？

我们需要一个新的 OLAP 计算引擎

从这个角度来看GreenPlum已经不能满足我们了，就算预先计算好也不能满足，需要一个新的OLAP计算引擎。这个新的引擎需要满足三个条件：

目标	解决方案
查询速度要快	<ol style="list-style-type: none"> SQL内高并发优于顺序读 纯内存计算优于内存+磁盘的计算
数据模型是维度建模	<ol style="list-style-type: none"> 方案必须支持Join
数据不需要同步	<ol style="list-style-type: none"> 数据要存储在HDFS 计算引擎要能够感知到Hive的Metadata
可通过扩容提高计算能力	<ol style="list-style-type: none"> 计算引擎要无状态 计算节点之间互相无依赖 存储和计算分离
方案成熟稳定	<ol style="list-style-type: none"> 在大型互联网公司有成功经验

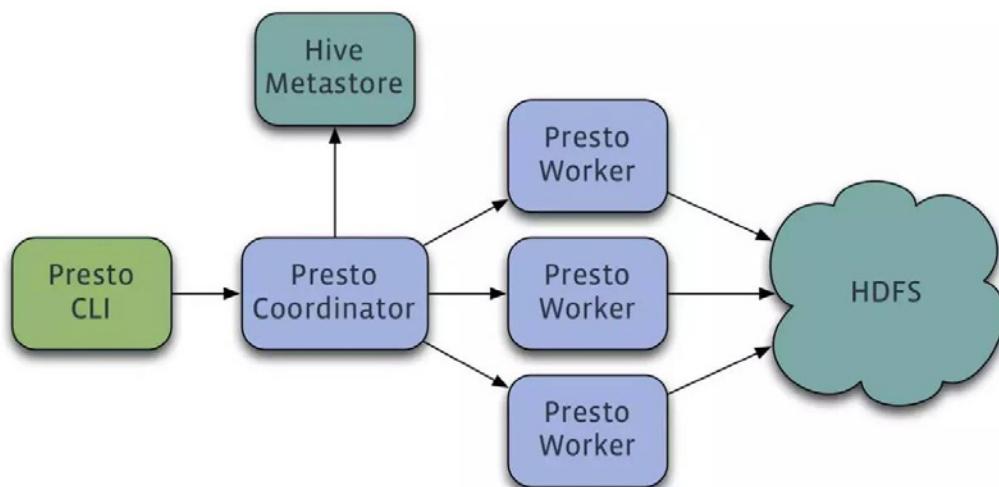
1. 没有预算算的模型。因为预算算的缺点是没有传统意义上的数据汇总层，直接从DW层明细数据上的直接计算。而且我们所有的业务场景化，只要DW层有这个数据就不用再开发了，直接拿来用就可以了。之前我们讲到数据先汇总，有些缓慢变化是需要刷数据的，这个头很疼，也要解决。
2. 速度要足够快。数据平均10秒返回，看上去挺慢的，不是秒出，为什么当时定这样的目标？因为刚才讲到之前的开发方式业务要排期等，这个周期非常长，如果现在通过一个可以随意组合的方式去满足它90%以上的需求，其实它在真正做的时候对性能的要求并没有那么严苛。我们也不希望这边查询的时候因为等待数据

把自己分析的思路或者日程打乱了，10秒可能是比较合适的。然后，因为我们的数据仓库DW层用维度建模，所以这个OLAP引擎必须支持Join。

3. 最后是支持横向扩展，计算能力可通过计算节点扩容获得提高，同时没有DB同步的问题。这里面东西还是挺多的，怎么解决这个问题呢？我们把需求分解了一下。

首先查询速度要快，我们需要一个SQL内在的高并发。其次用纯内存计算代替内存+硬盘的计算，内存+硬盘的计算讲的就是Hive，Hive一个SQL启动一下，包括实际计算过程都是很慢的。第二个是数据模型，刚才讲到数据仓库才是维度建模的，必须支持Join，像外面比较流行的Druid或者ES的方案其实不适用了。第三个就是数据不需要同步，意味着需要数据存在HDFS上，计算引擎要能够感知到Hive的Metadata。第四个是通过扩容提高计算能力，如果想做到完全没有服务降级的扩容，一个计算引擎没有状态是最好的，同时计算的节点互相无依赖。最后一点是方案成熟稳定，因为这是在尝试新的OLAP方案，如果这个OLAP方案不稳定，直接影响到了用户体验，我们希望线上出问题时我们不至于手忙脚乱到没办法快速解决。

Presto : Facebook 贡献的开源 MPP OLAP 引擎



这时候Presto进入我们的视野，它是Facebook贡献的开源MPP OLAP引擎。这是一个红酒的名字，因为开发组所有的人都喜欢喝这个牌子的红酒，所以把它命名为这个名字。作为MPP引擎，它的处理方式是把所有的数据Scan出来，通过Hash的方法把数据变成更小的块，让不同的节点并发，处理完结果后快速地返回给用户。我们看到它的逻辑架构也是这样，发起一个SQL，然后找这些数据在哪些HDFS节点上，然后分配后做具体的处理，最后再把数据返回。

为什么是 Presto

解决方案	Presto
SQL内高并发查询	<input checked="" type="checkbox"/>
纯内存计算	<input checked="" type="checkbox"/>
支持Join	<input checked="" type="checkbox"/>
数据存储在HDFS	<input checked="" type="checkbox"/>
感知Hive的Metadata	<input checked="" type="checkbox"/>
计算引擎要无状态	<input checked="" type="checkbox"/>
计算节点之间互相无依赖	<input checked="" type="checkbox"/>
成熟方案	<input checked="" type="checkbox"/>

从原理上来看，高并发查询因为是MPP引擎的支持。纯内存计算，它是纯内存的，跟硬盘没有任何交互。第三，因为它是一个SQL引擎，所以支持Join。另外数据没有存储，数据直接存储在HDFS上。计算引擎没有状态，计算节点互相无依赖都是满足的。另外它也是一个成熟方案，Facebook本身也是大厂，国外有谷歌在用，国内京东也有自己的版本，所以这个东西其实还是满足我们需求的。

Presto 性能测试

我们在用之前做了POC。我们做了一个尝试，把在我们平台上常用的SQL（不用TPCH的原因是我们平台的SQL更适合我们的场景），在GP和Presto两个计算引擎上，用相同的机器配置和节点数同时做了一次基准性

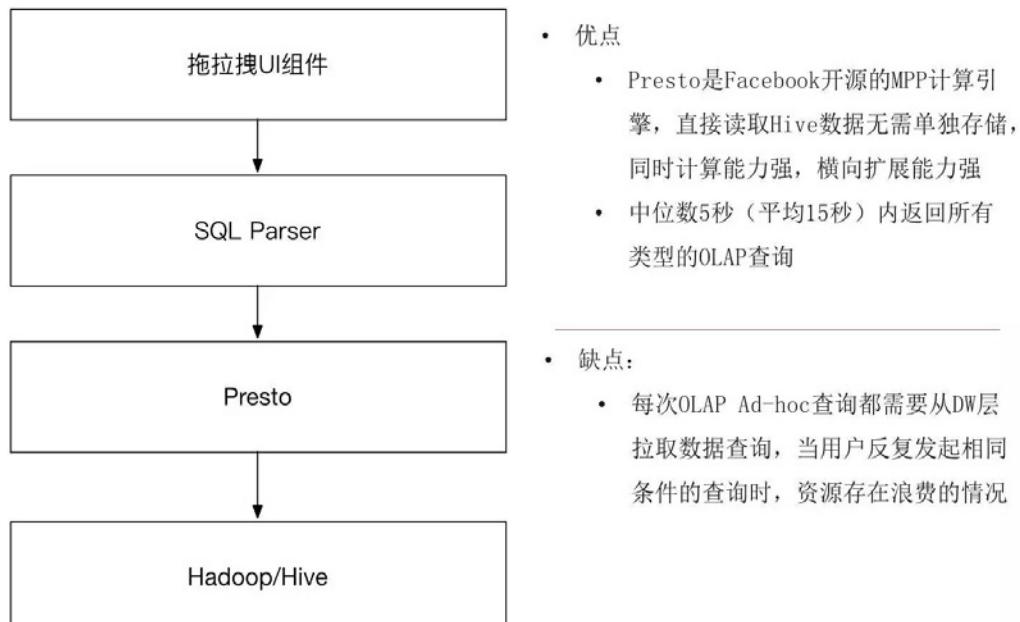
能测试，可以看到结果是非常令人欢喜的。



整体而言相同节点的Presto比GreenPlum的性能提升70%，而且SQL9到SQL16都从100多秒下降到10秒，可见它的提升是非常明显的。

当我们做完性能测试时，我们一个专门做引擎开发的同学叫了起来，说就你了，用Presto替代GreenPlum。

第3阶段



在Presto引进来之后，我们发现整个数据架构变得非常顺畅，上层用拖拉拽的UI组件生成传给SQL到Parser，然后传给Presto执行。不管是流量数据，还是埋点，还是曝光数据返回非常快，同时我们也把场景扩展到包括订单、销售之类的事务型分析上。用了之后中位数返回时间5秒钟，平均返回时间15秒，基本上这段时间可以返回所有的OLAP查询。因为用了DW数据，维度更丰富，大多数的需求问题被解决。

用了Presto之后用户的第一反应是为什么会这么快，到底用了什么黑科技。但是运行了一段时间后我们观察了用户的行为是什么样的，到底在查询什么样的SQL，什么维度和指标的组合，希望还能再做一些优化。

最快的计算方法是不计算

在这个时候我们突然发现，即使是用户自由组合的指标也会发现不同业务在相同业务场景下会去查完全相同的数据组合。比如很多用户会查同一渠道的销售流量转化，现在的方案会有什么问题呢？完全相同的查询也需要到上面真正执行一遍，实际上如果完全相同的可以直接返回结果不用计算了。所以我们就在想怎么解决这个问题？实际这里有一个所谓的理论——就是最快的计算就是不计算，怎么做呢？如果我们能够模仿Oracle的BGA，把计算结果存储下来，用户查询相同时可以把数据取出来给用户直接返回就好了。

于是这里就讲到了缓存复用。第一个阶段完全相同的直接返回，第二个阶段更进一步，相对来说更复杂一些，如果说提出一个新的SQL，结果是上一个的，我们也不结算，从上一个结果里面直接做二次处理，把缓存的数据拿出来反馈给用户。除了这个亮点之外，其实缓存很重要的就是生命周期管理，非常复杂，因为数据不断地更新，缓存如果不更新可能查出来的数据不对，在数据库会说这是脏读或者幻影读，我们希望缓存的生命周期可以自己管理，不希望是通过超时来管理缓存，我们更希望缓存可以管理自己的生命周期，跟源数据同步生命周期，这样缓存使用效率会是最好的。

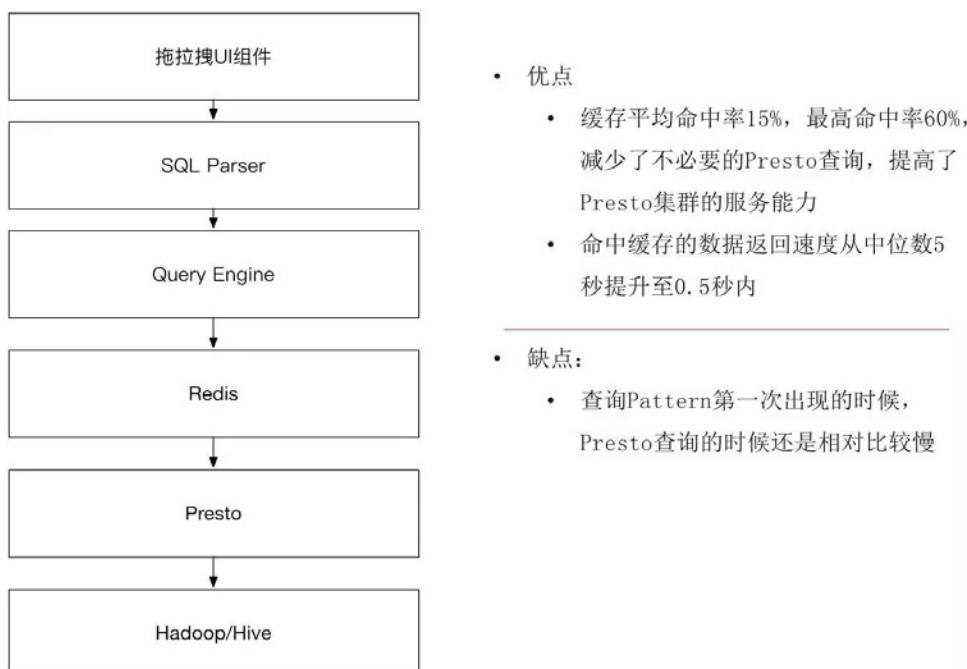
Redis：成熟的缓存方案

目标	解决方案
SQL和结果集要可以1对1匹配	Redis key-value存储符合需求
结果可以快速匹配	内存存储, Key匹配快速
生命周期管理	Redis提供API, 可二次开发与ETL调度系统联动自动清理
复用缓存结果	Redis不支持, 但可以自己二次开发

说到缓存要提到Redis，这是很多生产系统上大量使用的，它也非常适合OLAP。

首先我们想要的是SQL跟结果一对一匹配，它是非常符合这个需求的。其次我们希望缓存更快的返回，Redis是纯内存的存储，返回速度非常快，一般是毫秒级。第三个生命周期管理，它提供API，我们做二次开发，跟我们ETL调度系统打通，处理更新时就可以通知什么样的数据可以被用到。而缓存复用是不支持的，我们可以自己来做。

第3.5阶段



于是这时就把Redis的方案引入进来。

引入Redis之后带来一个新的挑战，我们不是只有一个计算引擎，我们暂时先把Redis称为一个计算引擎，因为数据可能在Redis，也可能需要通过Presto去把数据读出来，这时我们在刚才生成SQL之后还加入了新的一个组件，Query Engine的目的就是在不同的引擎之间做路由，找到最快返回数据的匹配。比如说我们一个SQL发下来，它会先去找Redis，看在Redis找有没有这个SQL缓存的记录，有就直接返回给用户，没有再到Presto上面查询。上线了之后，我们观察了结果，结果也是非常不错的，发现平均的缓存命中率达到15%，意味着这15%的查询都是秒出。

因为我们有不同的主题，流量主题、销售、收藏、客户，类似不同的主题，用户查询的组合不一样，特殊的场景下，命中率达到60%，这样除去缓存的返回速度非常快之外，缓存也有好处，就是释放了Presto的计算能力，原先需要跑一次的查询便不需要了。释放出来的内存和CPU就可以给其它的查询提供计算能力了。

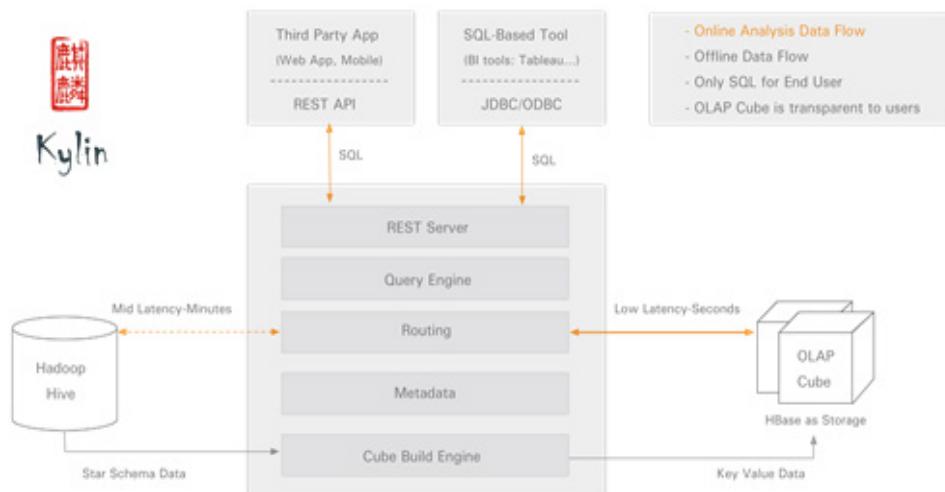
空间换时间：OLAP 分析的另一条途径

缓存的方案实施之后，看上去很不错了，这时候我们又引起了另一次思考，缓存现在是在做第二次查询的提速，但我想让第一次的速度也可以更快一些。说到第一次的查询，我们要走回老路了，我们说空间换时间，是提升第一次查询一个最显而易见的办法。

空间换时间，如果说OLAP ad-hoc查询从事先计算好的结果里查询，那是不是返回速度也会很快？其次，空间换时间要支持维度建模，它也要支持Join。最后希望数据管理简单一些，之前讲到为什么淘汰了GreenPlum，是因为数据同步复杂，数据的预计算不好控制，所以希望数据管理可以更简单一些。预计算的过程和结果的同步不需要二次开发，最好由OLAP计算引擎自己管理。同时之前讲到我们会有一个预先设计存在过度设计的问题，这个问题怎么解决？我们目前的场景是有Presto来兜底的，如果没有命中CUBE，那兜底的好处就是说我们还可以用Presto来承载计算，这让设计预计算查询的时候它的选择余地会更多。它不需要完全

根据用户的需求或者业务需求把所有的设计在里面，只要挑自己合适的就可以，对于那些没有命中的SQL，虽然慢了一点，但给我们带来的好处就是管理的成本大大降低了。

Kylin : eBay 贡献的开源 MOLAP 引擎



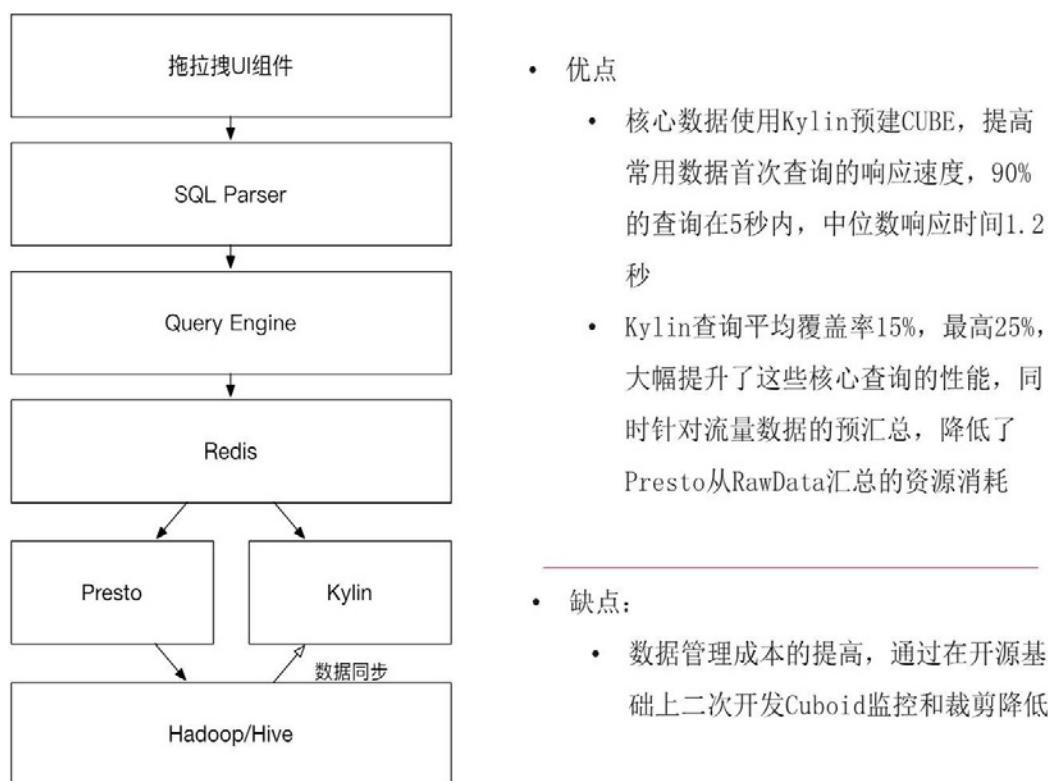
Kylin是由eBay开源的一个引擎，Kylin把数据读出来做计算，结算的结果会被存在HBase里，通过HBase做Ad-hoc的功能。HBase的好处是有索引的，所以做Ad-hoc的性能非常好。

为什么是 Kylin

目标	解决方案
空间换时间	Kylin仿照Oracle CUBE开发，通过预定义/计算CUBE的方式保证Ad-hoc速度
支持维度建模的模型	Kylin支持在定义模型时的Join
数据同步管理简单	<ol style="list-style-type: none"> 1. 预计算调度和数据管理均由Kylin自行管理 2. 有开放的API用于与ETL调度系统打通 3. 有简单的Web Console用户配置和管理CUBE
方案成熟稳定	在大型互联网公司有成功经验

首先空间换时间，我们在刚开始引入Kylin时跟Kylin开发聊过，他们借鉴了Oracle CUBE的概念，对传统数据库开发的人来说很容易理解概念和使用。支持维度建模自然支持也Join。预算算的过程是由Kylin自己管理的，也开放了API，与调度系统打通做数据刷新。另外Kylin是在eBay上很大的、美团也是投入了很大的精力的有成功经验的产品，所以很容易地引进来了。

第4阶段



于是，我们进入了唯品会OLAP分析架构的第四阶段：Hybird：Presto的ROLAP和Kylin的MOLAP结合发挥各自优势，Redis缓存进一步提升效率。

查询在Query Engine根据Redis-> Kylin再到Presto的优先级进行路由探查，在找到第一个可命中的路径之后，转向对应的引擎进行计算并给用户

返回结果。Kylin的引入主要用于提升核心指标的OLAP分析的首次响应速度。这个状态可以看到Kylin的查询覆盖率平均15%，最高25%，大幅提升核心数据的查询。同时流量几十亿、几百亿的数据从Kylin走也大大地减轻了Presto。虽然说这样的架构看起来挺复杂的，但这样的一个架构出来之后，基本上满足了90%的OLAP分析了。

OLAP 分析的技术进化是一个迷宫而不是金字塔

经过这么长一段时间的演进和一些摸索之后，我们觉得OLAP分析的技术是它的进化是一个迷宫，不是一个金字塔。过去说升级，像金字塔往上攀登，但实际上在刚才的过程大家发现实际上它更是像走迷宫，每个转角其实是都碰到了问题，在这个转角，在当时的情况下找最佳的方案。

不是做了大数据之后放弃了计算，也不是做了大数据不再考虑数据同步的问题。其实可以发现很多传统数据仓库或者RDBMS的索引、CUBE之类的概念慢慢重新回到了大数据的视野里面。对我们而言，我们认为更多的时候，我们在做一些大数据的新技术演进时更多的是用更优秀的技术来做落地和实现，而不是去拒绝过去的一些大家感觉好像比较陈旧的逻辑或概念。所以说对每个人来说，适合自己业务的场景方案才是最好的方案。

唯品会在开源计算引擎上所做的改进

接下来讲一下我们在开源计算引擎上做的改进。Presto和Kylin的角度不一样，所以我们在优化的方向上也不同。Presto主要注重提升查询性能，减少计算量，提升实时性。Kylin最主要优化维表查找，提高CUBE的利用率。

Presto上的改进

在提升查询性能上：

- 新增Hint语法，首先做的也是最重要的改动是在Presto中增加了一个Hint的语法，可以在SQL Join级别动态设置策略，通过编译时让join在replica和distribute两者之间设置，提高Join效率；

- 监控告警JOIN数据倾斜，通过减少数据倾斜提高执行效率；
- 增加多集群LOAD BALANCE，可平衡不同集群间计算量。

经过改造，Presto的查询实时性大幅提升。

在减少计算量上：

- 新增Kylin Connector，通过CUBE探嗅自动匹配SQL子查询中可以命中Kylin CUBE的部分，从Kylin提取数据后做进一步的计算，降低查询计算量；
- 经过改造，Presto升级为Hybird OLAP引擎，同时支持ROLAP和MOLAP两种模式。

在提高实时性上：

- 重写Kafka Connector，支持热更新Kafka中Topic、Message 和表/列的映射定义；
- 支持Kafka按offset读取数据，支持PB格式，提高Kafka数据源的读取效率；
- 经过改造，Presto不仅是离线OLAP引擎，准实时数据处理的能力也得到提高。

Kylin上的改进

在优化维表查找上：

- 通过引入Presto解决Kylin亿级维表实时Lookup OOM的问题，通过Presto查询替换了原有复杂的维表映射值查找机制；
- 经过改造，唯品会版的Kylin相比开源版本极大的扩展了对业务场景的支持程度。

在提升CUBE利用率上：

- 开发CUBE Advisor，通过统计分析总结合适的维度和指标组合辅助开发选择判断新建CUBE的策略，减少冗余和经验判断上的误差；
- 提供CUBE命中率监控，形成CUBE新建、使用到总结升级的闭环；

- 经此改造，CUBE命中率大幅提高，减少了资源的浪费提升了响应速度，经过这样的改造，开发不再只是根据自己的经验或者盲目建立，而是有数据可依。

OLAP 方案升级方向

最后我们讲一下OLAP升级方向。

对于Presto，我们将探索如何用RowID级别的索引的存储格式替换现有RowGroup级别索引的ORC File，在数据Scan级别尽可能精确的获取所需的数据，减少数据量，同时提高OLAP查询的并发度，应对大量用户并发OLAP分析场景。

对于Kylin，我们会尝试Streaming Cubing，使得Kylin OLAP分析从离线数据向实时数据迁移，以及探索Lambda Cubing，实现实时离线CUBE无缝融合。

最后，尝试探索下一代的方案，需要有更强的实时离线融合，与更强的原始数据和汇总的数据的融合，以及更强的数据处理能力，短期来讲没有更好的方案，希望跟大家一起讨论。

Apache Kylin 在百度外卖流量分析平台的应用与实践

作者 龚廖安



大家晚上好，我是百度外卖大数据研发中心的研发工程师龚廖安，我2013年11月加入百度，2015年11月加入百度外卖，一直从事大数据相关的研发工作，现负责百度外卖大数据产品方向的整体研发工作。今晚非常荣幸地给大家介绍Apache Kylin在百度外卖流量分析平台的应用与实践。

本次课程的内容包括流量分析的数据问题、技术选型、Apache Kylin解决方案、Kylin在百度外卖的其他应用场景这四个章节。

首先，我们将看看流量分析所面临的数据处理的技术难点有哪些；其次，针对上述技术难点，我们是怎么进行技术方案的尝试和选型，并最终选择Apache Kylin作为数据问题的解决方案；然后，将具体介绍怎么通过

Kylin数据建模，解决流量分析中的数据问题；最后，我们将看看Kylin在外卖其他大数据场景中的应用。

流量分析的数据问题

好，接下来我们开始的第一章节的内容。

在这一章节，我们将直面流量分析的数据问题。主要内容包括两部分：第一，我将给大家介绍什么是百度外卖的流量分析平台，了解流量分析的业务场景。第二，在了解业务场景后，再看看我们将要面对的数据问题有哪些？

什么是流量分析

流量分析是通过对进入百度外卖App的流量从路径、大区、城市、商圈、终端、版本、渠道等多个维度进行分析，帮助活动运营、渠道运营、产品经理、产品运营、大区经理等更好的了解其业务的流量情况，从而进一步优化业务。

■ 路径

首页八大金刚餐饮->频道二级页banner->点菜页选好了->订单页提交订单



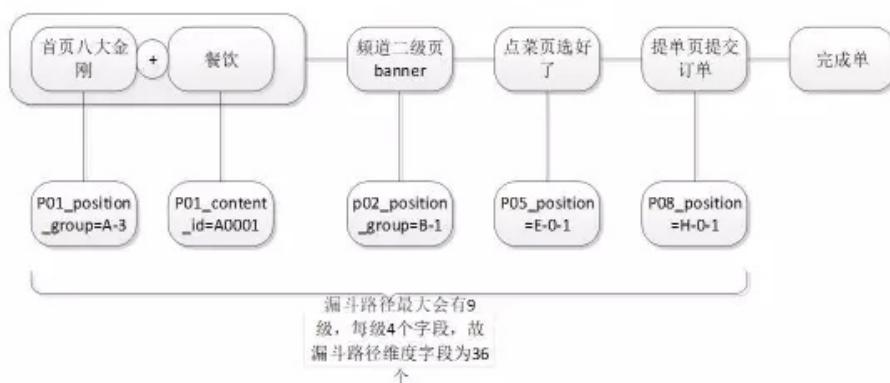
流量分析平台是通过对进入百度外卖App的流量从路径、大区、城市、商圈、终端、版本、渠道等多个维度进行分析，帮助活动运营、渠道运营、产品经理、产品运营、大区经理等角色更好的了解其业务的流量情况，从而进一步优化业务。

在流量分析平台中有一个非常重要的概念，就是路径。什么是路径呢？路径就是用户在百度外卖App里浏览所走过的页面、频道、位置、内容、按钮等。

下面给大家举一个路径的例子：一个用户通过点击百度外卖首页的八大金刚（餐饮）进入了频道二级页面，在频道二级页面的banner中选择了

一个商户进入这个商户的点菜页。点完菜后点击“选好了”按钮，进入订单页，并点击“去支付”按钮提交订单。那么这个用户的路径就是：首页八大金刚餐饮->频道二级页banner->点菜页选好了->订单页提交订单。这是一个用户从进入百度外卖到最后下单的完整路径。

什么是流量分析



用户的路径在我们的后台系统中是怎么记录的呢？首先在用户端，我们会进行日志埋点，用户浏览后会生成相应的日志，然后通过我们的日志流式ETL过程将用户浏览的路径存入数据仓库中。

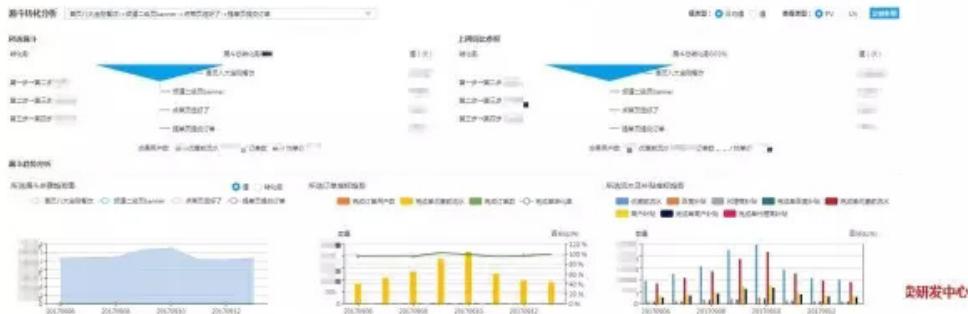
刚才例子中的路径在我们的数据仓库中是怎么记录的呢？首页八大金刚的首页对应P01，八大金刚对应position_group为A-3，餐饮对应的content_id为A0001。频道二级页banner对应的页面为p02，positon_group为B-1，点菜页选好了对应的页面为P05，position为E-0-1，订单页提交订单对应的页面为P08，position为H-0-1。在我们的数据仓库中，会把这些字段按列存储在对应的表中。

我们记录用户路径深度的级数有Me+p01~p08，总9级，Me是记录用户当前所在的页面，且每一级又有position_group、positon_id、content_id和content_type这4种类型，所以路径维度字段有36个。可以说标示路径的维度是非常多的。

了解完路径后，我们看看流量分析平台要做哪些事情呢？

什么是流量分析

- 统计区域细化到商圈
- 时间维度细化到小时
- 进行漏斗分析、趋势分析、数据对比、分布分析
- 全量路径



流量分析平台对流量的分析有下面这些需求。我们分析的粒度在时间和空间上要分别细化到小时和商圈。分析的种类包括漏斗分析、趋势分析、数据对比、分布分析等。路径可以分析到全量路径。

下面这张图是流量分析平台的界面，可以看到进行的漏斗分析和趋势分析。

在了解了业务场景后，我们看看流量分析平台面临哪些挑战。总的来

流量分析的挑战

- 维度多
 - 基础维度9个，路径维度36个
- 数据量大
 - 每天基础数据2.5亿行
 - 常用路径每天生产700万行，全量路径2亿行，并需要保存3个月的数据
- 查询场景多
 - 漏斗转化率分析、漏斗转化率周同比对比、漏斗趋势分析、订单/流水/补贴趋势分析
 - 维度可自由选择
- 在线交互查询
 - 秒级别返回

说，流量分析平台面临的问题主要是数据生产和查询速度的挑战。

首先是数据维度多，在流量分析中，基础维度有9个，路径维度36个。（基础维度主要是时间、城市、商圈、版本、渠道等）。

第二：数据量大，流量分析每天依赖的基础日志数据有2.5亿行。常用路径每天生产的数据是700w行，全量路径每天要生产2亿行，并且需要保留最近3个月的数据，总的数量达到百亿级别。

第三：查询场景多，刚才说到流量分析的场景非常多，每种场景对数据的要求和分析角度都是不一样的。漏斗分析关注的是路径维度的pv/uv，订单、流水的趋势分析关注的是订单、流水在时间上的变化，数据对比关注的是同样的指标和上周同期的对比。

此外，流量分析的维度可以自由选择，这样使得查询场景更难固化。当然每种查询场景都可以通过定制化的SQL来聚合计算，但场景非常多，如果只是在应用层去适配，开发成本太高。

最后是对查询速度的要求，流量分析平台要给用户提供在线查询服务，响应速度需要秒级别，太慢了用户是受不了的。

技术选型

下面我们看到第二章节。这一部分主要介绍我们针对流量分析平台中遇到的数据问题怎么进行尝试和技术选型，最终选择Kylin作为问题的解决方案。这一章节分为两部分，第一部分是我们在中间表方案中进行的尝试，第二部分介绍两种MLOAD查询引擎，Kylin和Druid的对比。

针对数据产品中比较固化且需要秒级响应的查询场景，我们一般的解决方案是每天凌晨对前一天的数据在数据仓库中，通过Impala/Hive对数据进行聚合，并将聚合后的表导入到Mysql或Greenplum，即GP中进行查询。

针对流量分析平台的数据场景，我们最初也是这样进行尝试的。但流量分析平台的数据量大、维度多、查询场景复杂，我们也对中间表方案进行了一些改造，以解决数据生产过程成本高、数据导入慢、数据查询慢的

问题。下面将简单讲述我们尝试的三个中间表方案。

中间表方案

- 在数仓中对数据做预聚合，再将聚合的数据导入

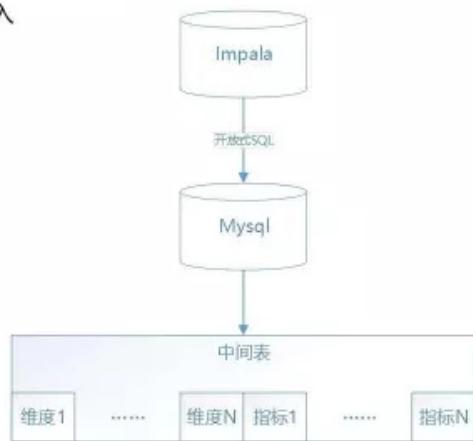
Mysql/Greenplum进行查询。

- 方案1

- 大宽表，所有维度和指标放在一个表
- 数据多次插入，通过联合主键保证维度不重复

- 问题

- 维度组合导致数据量大
- 插入数据慢
- Mysql主键限制



9 / 百度外卖研发中心

方案 1：

通过开放式Sql，利用Impala的计算能力，将基础数据进行聚合并导入到Mysql的一个大宽表中。由于很多指标和维度通过一个Sql计算不了，所以数据会多次插入Mysql，并通过联合主键保证数据不重复。这种方案的优点是数据生产过程简单、查询条件简单、查询速度满足要求。

但这个方案的问题有3个：1、维度组合导致数据量大，数据生产插入

中间表方案

- 方案2

- 每个路径对应一个表，存放路径对应的所
有维度和指标
- 每个路径的数据并行插入
- 通过联合主键保证维度不重复

- 问题

- 中间表数量太多，Mysql空间不够
- 全量路径无法维护

Mysql慢，插入需要半小时以上，当然这个时间还是可以接受的。2、由于Mysql主键数量的限制，导致方案不可行。我们这里有近40个维度需要去重，但我们使用的Mysql只能支持16个主键。

方案2：

下面我们看看方案2。方案一的主要问题是主键太多，为什么联合主键会这么多，主要是路径维度太多，但其实每个路径用的的维度不会这么多。所以我们就考虑是不是可以根据路径对数据生产过程进行拆分，每个路径对应一个数据生产过程，这样维度数就能够满足联合主键数量的限制。

所以我们对方案1进行了改造，每条路径对应一个中间表，各个不同的路径并行插入，提高数据插入效率。但这个方案也有问题，就是如果路径太多，则中间表的数量也会很多，导致数据生产的维护成本太高，如果后面要做全量路径的话，那中间表几乎是不可维护的。由于中间表数量大，数据重复，导致Mysql的空间不够。

中间表方案

■ 方案3

- 中间表存放Impala再同步Greenplum
- 生成中间表分两步，第一步存放所有维度和部分指标，第二步再进行数据合并

■ 问题

- 数据生产过程太复杂，流程长
- 仍无法解决中间表数量的问题

由于流量分析平台前期分析的路径是有限的，所以当时的问题主要是集中在Mysql存储空间和数据写入的效率上，所以我们继续尝试一种直接在集群内部进行数据生产的方案。

方案3去掉了Mysql的存储。中间表用Impala生产后存放在集群上，并通过我们的自动同步工具，将中间表同步到GP中，对外提供查询服务。由于Impala没有主键的概念，所以每次生成的数据都会单独生成一个表，还需要对数据进行二次聚合。这个方案数据生产快了很多，但数据生产过程比较复杂，且仍遗留中间表数量的问题。

中间表方案

■ 问题

- 流量平台维度多，导致数据在数仓中聚合的成本越来越高
- 数据量大导致Mysql的导入和查询变慢
- Impala+GP方案的流程太复杂
- UV计算不准确
- 查询场景相对固定

■ 解决方案

- 调研更高效的OLAP引擎，满足流量平台场景的解决方案。

总的来说，中间表方案的问题有以下几个。

1. 流量分析平台维度和指标多，导致数据在数仓中聚合的成本越来越高，开发效率低。
2. Mysql性能达不到要求，主要体现在往Mysql中插入数据和查询数据的时间长。
3. 如果舍弃Mysql，如果用Impala+GP方案生产数据，则流程太复杂。
4. UV计算不准确，这个问题是中间表方案无法解决的，因为计算UV需根据用户进行去重，但我们的中间表的聚合不能做到用户级别的中间表，如果做到用户级别，那就是明细数据了，所以用中间表的话，我们无法对用户去重来计算UV，只能做简单的加和，从而导致UV不准确。
5. 中间表方案的查询场景相对固定，查询的数据只能是中间表预先

设定好的数据，如果有新的数据查询场景在中间表中不能满足需求，则需要开发新的中间表生产流程。

所以，我们需要调研更高效的、满足流量分析平台场景的OLAP引擎。

OLAP引擎选择

■ 数据量

- 百万、千万、亿、百亿？

■ 性能

- 毫秒、秒、分钟、小时

■ 灵活性

- 明细or聚合
- SQL支持？
- 离线or实时

	数据量	性能	灵活性
Mpp架构 (Impala/Presto/Spark)	百亿	响应时间无保障，当数据量和复杂度增加时，响应时间达到分钟甚至小时级别	任意SQL查询，灵活性高
预计算架构 (Druid/Kylin)	百亿	稳定的秒级别响应	固定场景，灵活性差

经过调研，我们发现对OLAP引擎需求主要体现在支持的数据量、查询性能、灵活性三个方面。数据量是指能支持多大的数据量，是百万、千万、亿还是百亿？性能是指对查询请求的响应速度，是毫秒级、秒级、分钟级还是小时级？灵活性是指能够支持什么样的查询需求，明细还是聚合？离线还是实时？是否支持SQL查询？目前业界没有一个查询引擎能在三方面做到完美，所以需要根据业务场景进行取舍，选择合适的查询引擎。

那么对于流量分析平台，我们的业务场景对这三方面的要求是什么样的？在数据量上，流量分析平台需要的是百亿级别的数据量支持；查询性能要求秒级返回；查询场景是聚合数据；数据不要求实时，离线聚合前一天的数据就可以；最好能够支持SQL查询。

目前主流的大数据查询引擎架构有两种：Mpp架构和预计算架构，Mpp架构的查询引擎有Impala、Presto、Spark，预计算架构的查询引擎有Druid和Kylin。我们看看这两种架构在数据量、性能和灵活性三方面的支

持情况。

Mpp架构和预计算架构在数据量上都支持百亿级别的查询。

查询性能上，Mpp架构对查询请求的响应时间没有保证，当数据量和计算复杂度增加后，响应时间会变慢，从秒级到分钟级，甚至到小时级。而预计算架构是在数据入库的时候进行预计算，牺牲了一定的灵活性以换取性能，所以对超大数据集的查询请求响应时间能够稳定在秒级。

在灵活性方面，Mpp架构的灵活性是很强的，支持任意的SQL查询。而预计算架构的场景相对固定，只支持预计算后的数据查询。

从以上分析可以看出，针对流量分析平台的数据查询场景，我们应该选择预计算架构的查询引擎。

Kylin vs Druid



现在业界主流的预计算架构的查询引擎是Kylin和Druid，它们都是MOLAP查询引擎，都满足交互式的查询速度需求，都支持大数据量的查询。这两个查询引擎我们又是怎么选择呢？

下面我们将二者进行对比，看看他们之间的差异，哪个更适合流量分析平台的查询场景？

首先，Kylin的应用场景主要是非实时分析，1.5版本后后，通过对接

Kafka支持实时分析。Druid则以实时分析场景为主，但也支持非实时分析，但不是主流场景。

在计算方式上，Kylin是利用MapReduce、Spark对数据进行聚合计算，而Druid的计算是在内存中进行的，速度会更快，但需要处理复杂的集群资源调度、容灾容错、数据扩容等问题。Kylin只是专注于预计算的优化、查询的优化等问题，将数据计算的问题交给MapReduce、Spark，存储的问题交给HBase、Hive。

Kylin是原生的支持SQL和JDBC接口的，而Druid需要引入第三方的SQL引擎。

Kylin1.5.3开始基于bitmap支持精确count distinct计算方式。Druid不支持精确的count distinct，使用HyperLogLog或datasketches实现近似count distinct，可以调节datasketches算法的参数使值越来越精确，但资源消耗非常严重。

Kylin回溯历史数据非常简单，只需要重新刷新立方体对应的Segment就行，Druid由于支持实时场景，回溯相对复杂一些，在index server出现后，解决了近期数据回溯的问题。

Kylin由于要预计算维度组合并存储，以空间换时间，维度膨胀比较严重，而Druid是通过建索引的方式将数据转化为Segment，所以进行无维度膨胀的问题。

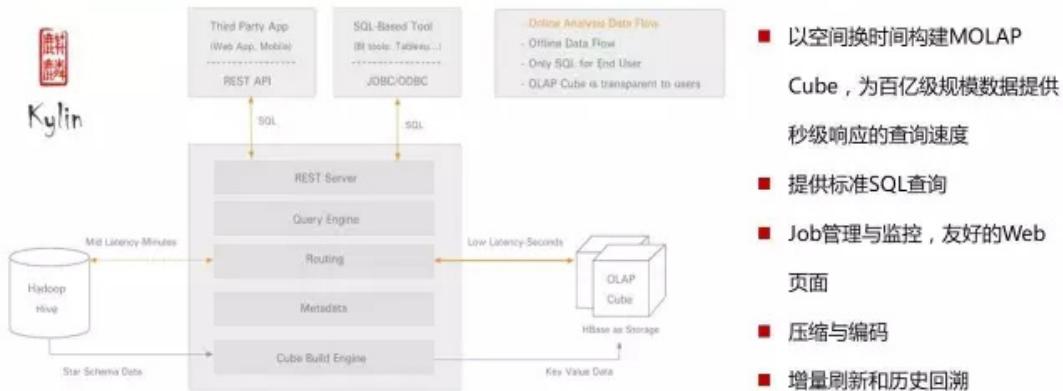
从以上对比可以看出，Druid主要面向的是实时数据的场景，但流量分析平台主要是按天进行数据聚合，且使用更加方便，因此Kylin比Druid更适合作为流量分析平台的查询引擎。

下面给大家简单介绍一下Kylin。

Apache Kylin是一个开源的分布式分析引擎，在Hadoop之上提供超大规模数据的SQL查询接口及多维分析能力，最初由eBay开发并贡献到开源社区，能为巨大Hive表提供亚秒级别的查询响应。

Kylin的架构如图，核心思想是对Hive中的数据预计算，利用Hadoop的MapReduce对多维分析可能用到的维度和指标进行预计算，将计算的结

Kylin的特点



果保存成Cube并存在Hbase中，提供查询时的直接访问。Kylin把高复杂度的聚合计算、多表连接等操作转换成对预计算结果的查询，使其能够拥有很好的快速查询和高并发能力。

此外，Kylin还提供标准SQL查询，友好的web页面进行Job管理和监控，数据压缩编码和方便的增量刷新和数据回溯功能。总的来说就是非常好用。

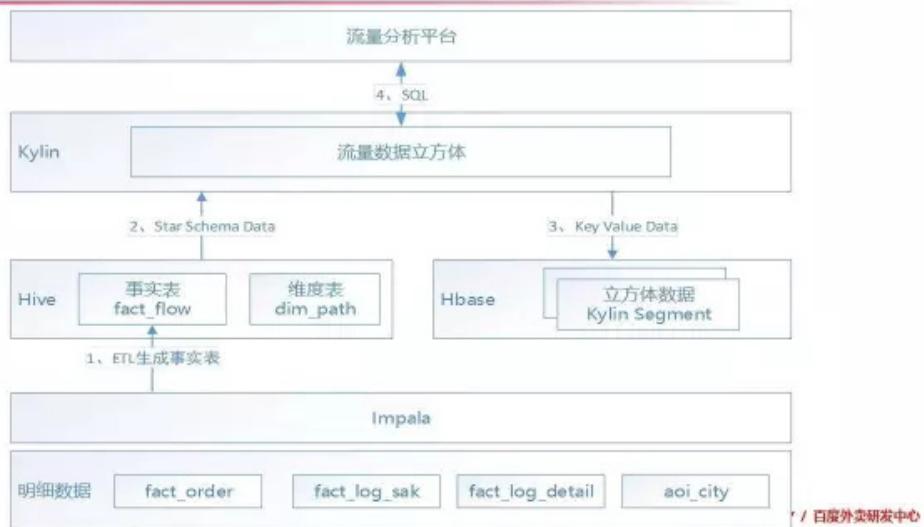
Apache Kylin 解决方案

好，接下来，我们将看看具体的解决方案。本章节将介绍怎么用Kylin对流量分析平台的数据进行建模，并最终满足流量平台的数据需求。该章节分为降维、事实表生产、立方体配置、立方体构建这四个部分。

降维将介绍我们如何对路径维度进行维度聚合，并利用kylin提供的维度优化方案对维度进行优化。事实表生产将介绍用户浏览路径事实表fact_flow的每天例行生产过程。立方体配置将具体介绍流量分析平台的数据立方体是怎么配置的。立方体构建将介绍怎么进行立方体数据的增量更新、Segment合并和数据清理。

在我介绍具体的实践方案之前，我们先看一下利用Kylin进行流量分析平台数据生产的总体架构。

Apache Kylin解决方案



我们从下往上看，在集群上，存放的是流量分析平台依赖的明细数据，包括订单、sak日志、详情日志、商圈、城市信息等。我们每天利用Impala的计算能力，通过ETL例行生产用户浏览路径事实表的数据，并存放在Hive中。我们采用的是星型模型，有一个dim_path的维度表，用于存放路径信息。Kylin根据立方体配置的Schema，每天例行构建立方体数据，并存放在Hbase中。流量分析平台通过Kylin提供的JDBC接口，用SQL查询数据并进行展示。

根据之前的讲述，流量平台的维度有40多个，如果直接用这40多个

Apache Kylin解决方案：降维

■ 路径降维

- 36个路径维度->1个路径ID和dim_path表关联

■ 立方体维度优化

- Aggregation Groups (维度聚合组)
- Mandatory Dimensions (必选维度)
- Hierarchy Dimensions (层级维度)
- Joint Dimensions (联合维度)
- Derived Dimensions (派生维度)

维度进行建模，那维度组合的数目将有 2^{40} 多次方，数据膨胀会非常厉害。所以，用Kylin进行数据建模的第一步就是要进行降维。

流量分析的数据为什么有这么多维度呢？其实主要是路径维度太多，有36个路径维度。经过分析，可以将这36个路径维度可以聚合成一个路径ID，一个路径ID表示一种路径，路径的具体信息存放在路径维度表中，事实表通过路径ID和维度表关联。我们经常关注的路径只有几百个，所以这个维度的基数不会太大。通过路径维度聚合，流量分析平台的维度数量降为11个，这样维度组合就不会太多。

此外，我们还通过Kylin提供的立方体维度优化方案对立方体的维度进行优化，进一步减少维度组合的数目。

Kylin有5种立方体维度优化方案，维度聚合组、必选维度、层级维度、联合维度、派生维度。

- **维度聚合组：**是对维度进行分组，以降低维度组合数目。如果有 $n+m$ 个维度，如果将 n 个维度分为一组， m 个维度分为另一组，则维度组合从 2^{n+m} 次方降为 $2^n \times 2^m$ 次方。
- **必选维度：**指所有的cuboid都包含的维度，每加一个必选维度，维度组合的数目就减少一半。
- **层级维度：**指一系列具有层级关系的维度组成一个层级维度，设置了层级之后，对Cuboid增加了约束，低Level的维度一定要伴随高Level的维度出现，从而使维度组合大大减少。
- **联合维度：**当有些维度始终在一起时，可以将这些维度设为联合维度。则任何有效的Cuboid要么不包含这些维度，要么包含所有维度，这些维度始终在一起，从而降低维度组合的数目。如果有 $n+m$ 个维度，如果将 m 个维度配置成一个组合维度，则维度组合从 2^{n+m} 次方降为 2^{n+1} 次方。
- **派生维度：**派生维度是针对维度表的，如果某张维度表有多个维度，如果该维度表一个或者多个列和维度表的主键是一一对应的，则可以将这些维度设置为派生维度。这样在Kylin内部会将其

统一用维度表的主键来替换，以降低维度组合的数目。但查询效率会降低，因为Kylin在使用维度表主键进行聚合后，查询时需要再将主键转换为真正的维度列返回给用户。

Apache Kylin解决方案：降维

Aggregation Groups

Includes	["FACT_FLOW.PATH_ID","FACT_FLOW.REGION_ID","FACT_FLOW.CITY_ID","FACT_FLOW.AOI_ID","FACT_FLOW.FROM_TYPE","FACT_FLOW.APP_VERSION","FACT_FLOW.APP_CHANNEL","FACT_FLOW.INDEX_HOUR","FACT_FLOW.ORDER_STATS","FACT_FLOW.INDEX_DAY"]
Mandatory Dimensions	["FACT_FLOW.INDEX_DAY","FACT_FLOW.PATH_ID"]
Hierarchy Dimensions	["FACT_FLOW.REGION_ID","FACT_FLOW.CITY_ID","FACT_FLOW.AOI_ID"]

Joint Dimensions

11	NAME	DIM_PATH	derived	["NAME"]
12	PATH	DIM_PATH	derived	["PATH"]

在流量分析平台的立方体模型中，由于查询场景会涉及到所有维度，所以无法将维度分组，我们只建了一个维度聚合组。但采用了必选维度和层级维度这两种优化方案。INDEX_DAY（日期）和PATH_ID（路径ID）是必选维度，REGION_ID（大区ID）、CITY_ID（城市ID）、AOI_ID（商圈ID）是层级维度。同时维度表里的路径名称和路径编码采用的是派生维度。

确定流量平台数据的降维方案后，下一步就是进行流量平台基础数据的生产。流量平台采用的数据模式是星型模型，包括一个事实表fact_flow和一个维度表dim_path。维度表相对固定，不需要每天更新，手动生成一次就可以了。事实表fact_flow需要每天例行更新，该表是调度系统在每天凌晨从基础的订单、sak日志、日志详情表，通过ETL过程例行生产的，并将数据存放在Hive中。

该表包括的维度字段有：path_id、区域id、城市id、商圈id、来源、app版本、app渠道、小时、订单状态、日期。指标字段包括：用户的

cuid、优惠前价格、百度补贴、商户补贴、代理商补贴、订单id。

下面我们看看流量分析平台的立方体是怎么配置的？这里我们只介绍几个关键步骤的配置，包括：模型配置、维度配置、指标配置。首先需要在Model里配置星型模型，通过path_id将事实表和维度表关联起来。

Apache Kylin解决方案：事实表生产

■ 每日例行构建用户浏览路径的事实表fact_flow

■ 维度字段

- PATH_ID、REGION_ID、CITY_ID、AOI_ID、FROM_TYPE、APP_VERSION、APP_CHANNEL、INDEX_HOUR、ORDER_STATUS、INDEX_DAY

■ 指标字段

- CUID、REAL_TOTAL_PRICE、DISCOUNT_BAIDUFEE_PRICE、SHOP_BUTIE、AGENT_BUTIE、ORDER_ID

下一步进行维度配置，维度配置在立方体配置里进行，事实表里的维度的类型都是normal类型，维度表里的维度配置成derived类型，流量分析平台有10个normal维度、2个derived维度。

最后进行指标配置，我们这里配置的指标包括PV、UV、优化前流

Apache Kylin解决方案：立方体配置

■ 星型模型

Model Designer

Fact Table FACT_FLOW

Join Tables					
ID	Table Alias	Table Name	Table Kind	Join Type	Join Condition
1	DIM_PATH	WAIMAI.DIM_PATH	Normal	inner	FACT_FLOW.PATH_ID = DIM_PATH.ID

Apache Kylin解决方案：立方体配置

■ 维度配置

Cube Designer



ID	Name	Table Alias	Type	Column
1	PATH_ID	FACT_FLOW	dimension	PATH_ID
2	REGION_ID	FACT_FLOW	dimension	REGION_ID
3	CITY_ID	FACT_FLOW	dimension	CITY_ID
4	AOI_ID	FACT_FLOW	dimension	AOI_ID
5	FROM_TYPE	FACT_FLOW	normal	FROM_TYPE
6	APP_VERSION	FACT_FLOW	normal	APP_VERSION
7	APP_CHANNEL	FACT_FLOW	normal	APP_CHANNEL
8	INDEX_HOUR	FACT_FLOW	normal	INDEX_HOUR
9	ORDER_STATUS	FACT_FLOW	normal	ORDER_STATUS
10	INDEX_DAY	FACT_FLOW	normal	INDEX_DAY
11	NAME	DIM_PATH	derived	[NAME]
12	PATH	DIM_PATH	derived	[PATH]

22 / 百度外卖研发中心

Apache Kylin解决方案：立方体配置

■ 指标配置

Cube Designer



Name	Expression	Parameters	Return Type
PV	COUNT	Value: 1, Type: constant	bigint
UV	COUNT_DISTINCT	Value: FACT_FLOW.CUID, Type: column	bitmap
REAL_TOTAL_PRICE	SUM	Value: FACT_FLOW.REAL_TOTAL_PRICE, Type: column	bigint
DISCOUNT_BAIDUFEES_PRICE	SUM	Value: FACT_FLOW.DISCOUNT_BAIDUFEES_PRICE, Type: column	bigint
SHOP_BUTIE	SUM	Value: FACT_FLOW.SHOP_BUTIE, Type: column	bigint
AGENT_BUTIE	SUM	Value: FACT_FLOW.AGENT_BUTIE, Type: column	bigint
ORDER_ID	SUM	Value: FACT_FLOW.ORDER_ID, Type: column	bigint

中心

水、百度补贴、商户补贴、代理商补贴、订单量。PV通过count来计算。UV需要做去重，通过对cuid进行count distinct来计算。其他指标都是sum求和来计算。

立方体关键的配置步骤就完成了，后面就绪进行例行增量构建。

由于我们每天都会产出新的基础数据，立方体的数据也需要不断更新，这里我们通过增量构建的方式将每天新生成的数据加入到立方体中提

Apache Kylin解决方案：立方体构建

■ 增量构建

Job Name	Cube	Progress	Last Modified Time	Duration	Actions
flow_cube_new - 20170831000000_20170901000000 - BUILD - GMT+08:00 2017-09-01 07:09:21	flow_cube_new	100%	2017-09-01 07:32:54 GMT+8	23.22 mins	Action
flow_cube_new - 20170830000000_20170831000000 - BUILD - GMT+08:00 2017-08-31 05:57:49	flow_cube_new	100%	2017-08-31 07:28:25 GMT+8	29.65 mins	Action
flow_cube_new - 20170829000000_20170830000000 - BUILD - GMT+08:00 2017-08-30 06:52:44	flow_cube_new	100%	2017-08-30 07:15:49 GMT+8	22.65 mins	Action
flow_cube_new - 20170828000000_20170829000000 - BUILD - GMT+08:00 2017-08-29 06:54:05	flow_cube_new	100%	2017-08-29 07:13:12 GMT+8	16.83 mins	Action
flow_cube_new - 20170821000000_20170828000000 - MERGE - GMT+08:00 2017-08-28 07:37:36	flow_cube_new	100%	2017-08-28 07:45:45 GMT+8	8.03 mins	Action
flow_cube_new - 20170827000000_20170828000000 - BUILD -	flow_cube_new	100%	2017-08-28 07:37:45 GMT+8	21.37 mins	Action

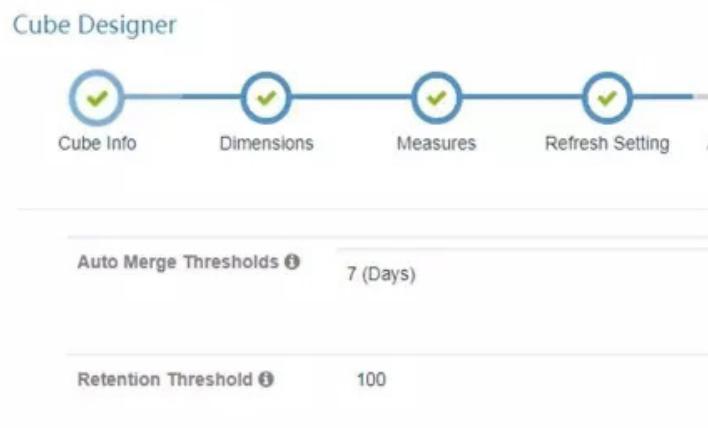
供给流量分析平台查询。

Kylin会将Cube划分为多个Segment，每个Segment用起始时间和结束时间来标志。

Segment代表一段时间内数据的计算结果，一个Segment的起始时间等于之前一个Segment的结束时间。在同一个Cube下，不同Segment的结构定义、构建过程、优化方法、存储方式都是一样的。

Apache Kylin解决方案：立方体构建

■ Segment合并、清理



25 / 百度外卖研发中心

从图中我们看到，每天早上7点多，会进行立方体构建，每次构建前一天产生的数据。通过百度外卖的调度系统，在依赖的fact_flow数据生产

好后，自动调用Kylin的restfulAPI接口进行立方体的构建。

增量构建虽然好用，但也存在一个问题，就是增量构建会使Segment碎片太多，导致Kylin的查询性能下降，我们通过对Segment进行合并和清理来解决。Segment合并是将几个Segment合并到一起，清理是将无用的Segment删除，从而减少Segment的数量。合并和清理我们是通过配置立方体的自动合并门限和Segment保留门限来实现的。

如图所示，在立方体配置的Refresh Setting配置页中，我们将Segment自动合并门限配置成7，则每7天的Segment会合并在一起。Segment保留门限配置成100，则我们只保留100天左右的Segment。如果一个Segment的结束时间距离最晚一个Segment的结束时间大于这个门限，则这个Segment就会自动从立方体中删除。

至此，整个流量分析平台的数据建模的过程都介绍完了。

Apache Kylin解决方案：流量平台进一步应用

■ 全量路径

- 所有路径组合70+万
- 流量桑基图、用户自选路径

■ 分布分析

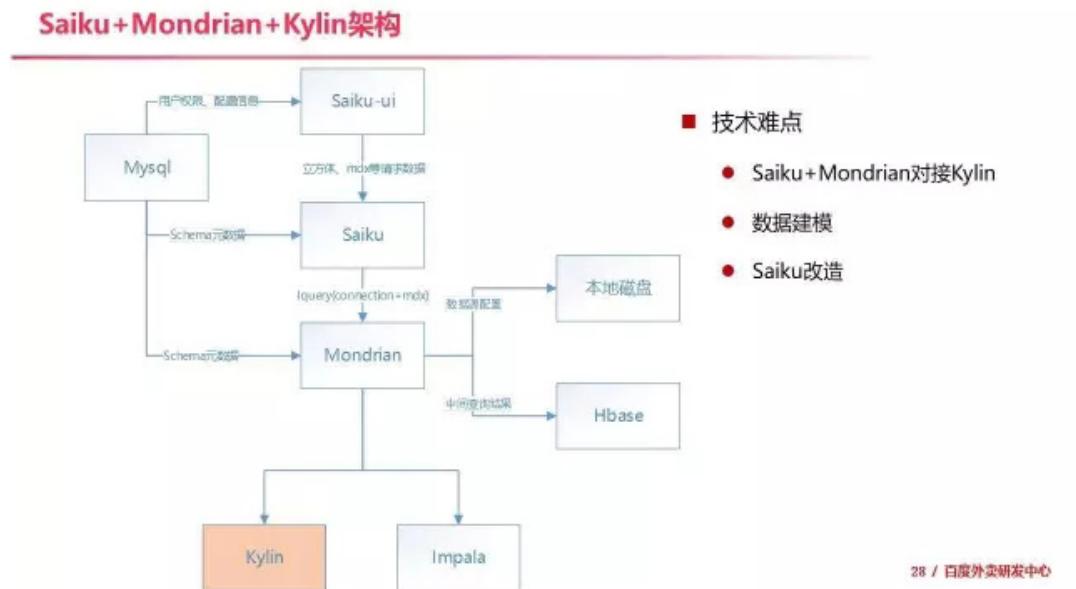
- 从城市、版本、渠道、商圈、入口等多个维度的流量分布进行分析
- 分析指标包括：订单UV、进店率、完成单转化率、订单数等

通过用Kylin进行数据建模，解决了流量分析平台数据的问题，那么下一步在流量平台上，我们还有哪些深入的应用呢？首先是全量路径，目前我们的路径维度里只有常用的几百个路径，后续我们会考虑将全量路径，分析流量的桑基图、用户也可以自定义路径进行分析。要做全量路径的话，另一个挑战就是路径维度的基数会很多，可能会有70+万种组合，需要进一步对路径维度进行优化。

另一个应用就是进行分布分析，目前的分析主要是对路径进行漏斗分析，后续会从城市、版本、渠道、商圈、入口等多个维度的流量分布进行分析，主要分析的指标包括：订单UV、进店率、完成单转化率、订单数、完成订单量、商户补贴等。

其他应用

在第4部分，我们将介绍Kylin在百度外卖大数据部的其他应用。本章将主要介绍怎么将Kylin接入百度外卖的报表系统。



下面我们先来看看整个报表系统的架构。百度外卖大数据部有自己的一个基于Saiku+Mondrian+Impala的报表系统，我们计划增加一个Saiku+Mondrian+Kylin架构的报表系统。通过数据建模，用户可以在Saiku前端拖拽需要的指标和维度，产出自己想要的报表。Saiku将用户拖拽的指标、维度生成mdx语句，Mondrian将mdx转换成Sql提交给查询引擎进行查询，Saiku将查询的结果拼装成用户定义的表格返回给前端进行展示。

之前的报表系统的查询引擎对接的是Impala，优点是灵活，可以支持

各种类型的sql查询，但问题是用户的查询多了就会给Impala造成非常大的压力，影响Impala每天例行的生产任务。且如果Sql比较复杂，Mpp架构的查询引擎返回的时间会比较长甚至出错，数据查询效率较低。所以可以接入Kylin作为查询引擎，为一些场景比较固定的查询提供服务，一方面减少了集群压力，另一方面能提高查询速度。

在对接Kylin的过程中，主要有三个技术难点需要解决：1、Saiku+Mondrian对接Kylin；2、数据建模；3、Saiku的改造。下面我们将主要介绍这三个问题我们是怎么解决的。

报表系统对接Kylin

■ Mondrian4对接Kylin2.0

- 在Mondrian源码中增加对Kylin方言的对接
(<https://github.com/mustangore/kylin-mondrian-interaction>)
- 解决count distinct的sql在Kylin中报错的问题（参考有赞团队的解决方案）
- 解决Kylin2.0的一个数组越界的bug (Kylin2.1版本解决，
<https://issues.apache.org/jira/browse/KYLIN-2670>)
- Mondrian4的Schema兼容View改造
- Mondrian4支持left join的改造(<https://github.com/xiaofanyw/mondrian-join-support>)

29 / 百度外卖研发中心

下面我们先看看Mondrian是怎么和Kylin对接的？我们对接的版本是Mondrian4.4和Kylin2.0，在对接过程中主要解决了以下几个问题。首先是在Mondrian源码中增加对Kylin方言的支持，在git上有一个整合Kylin、Mondrian和Saiku的开源项目，参考这个项目的指引，可以轻松搭建Mondrian+Kylin+Saiku的系统，这里要感谢项目的作者mustangore。

但git上的开源项目有一个bug，当我们需要进行count distinct查询的时候，会报错，Mondrian生成的Sql语法存在问题，在Kylin里查询的时候报错了。有赞技术团队给出了问题的解决方案，可以通过修改Mondrian的Kylin方言补丁和Mondrian的源码搞定。

在对接Mondrian过程中，还发现Kylin2.0存在一个数组越界的bug，该bug已经在kylin2.1版本中解决，如果未升级到2.1版本，可以参考我在kylin社区上提供的解决方案。

我们还对Mondrian4.4进行了一些改造，主要解决了Mondrian4.4的Schema对接Kylin兼容View的问题和Mondrian4对left join的支持。第一个问题是在获取表结构的时候有字段未获取到导致的报错，通过修改Mondrian获取表结构的流程来解决这个问题。第二个问题由于Mondrian只支持inner join，这样的话如果用inner join在Kylin中进行数据建模，当两个表有字段匹配不上时，会导致数据缺失，比如事实表存在空字段而维度表没用空字段。如果要解决这个问题，可以在基础数据中把空字段补上或者在Hive中建视图来补空字段，但由于涉及的表非常多，改造成本太高。如果Mondrian能支持left join，则可以一次性彻底解决这个问题。这可以参考git上的解决方案。

此外我们还进行Mondrian3对接Kylin的改造，主要问题当需要做表关联时，Mondrian3生成的Sql在语法上Kylin不支持，如果要支持的话，改造成本非常高，所以舍弃了这个方案。

报表系统对接Kylin

■ 数据建模

- 已有的Mondrian3的Schema配置升级到Mondrian4
- 通过在Hive中建View兼容Mondrian3中的View配置
- 维度根据场景分组

■ Saiku3.14改造

- 用户访问权限
- 用户指标权限

下面继续看看数据建模和Saiku改造的问题。数据建模需要解决的3个问题。首先需要将我们之前的Mondrian3的Schema升级到Mondrian4的

Schema。Mondrian3和Mondrian4的Schema配置差异是比较大的，其实 Mondrian4在程序内部能自动将Mondrian3的Schema进行升级，但升级过程中容易出错，且升级后的Schema在和Kylin进行兼容的时候存在问题，比如我们在Mondrian3里面使用了很多View的配置，升级后生成的Sql在 Kylin中查询就不了。所以我们只能自己开发工具将Mondrian3的Schema进行升级到Mondrian4的Schema。

第二个问题是兼容View的问题。由于我们之前用Impala作为查询引擎，可以支持非常灵活的查询Sql，所以在Schema里配置了很多View的查询场景。但用Kylin进行建模的话，由于Kylin不能支持那么灵活的Sql，查询的数据只能是已经建模好的数据，所以很多基于View的查询是不能支持的，虽然我们通过对Mondrian4的改造支持了View查询，但这些View在Kylin中还得进行建模，过程非常复杂，所以我们只能进一步降低灵活性，对规范底层的数据进行建模。将Schema中使用到的View先在Hive中创建相应的视图，用这些视图在Kylin中构建立方体。这样能较低成本解决通过View查询的问题。

第三个问题是维度分组。之前用Impala查询，无维度膨胀的问题，所以我们有一些立方体配置了一百多个维度，而这种场景如果在Kylin中建模，则维度膨胀会非常严重，此时需要根据业务场景对维度进行分组，拆

报表系统对接Kylin

The screenshot shows the Mondrian Schema Editor interface. On the left, there's a tree view of the schema structure under 'BMS'. The main area displays the cube definition with various dimensions and measures. A table at the bottom shows historical data for dimensions like 'City' and 'Product' across different dates.

维度	维度名	维度值									
日期	日期	2017-07-01	2017-07-02	2017-07-03	2017-07-04	2017-07-05	2017-07-06	2017-07-07	2017-07-08	2017-07-09	2017-07-10
产品	产品	产品A	产品B	产品C	产品D	产品E	产品F	产品G	产品H	产品I	产品J
城市	城市	上海	北京	广州	深圳	杭州	南京	武汉	长沙	成都	重庆

分成小的立方体进行查询。

Saiku3.14改造，这里主要是解决用户访问权限和用户指标权限的问题。这两个问题都可以通过修改Saiku的源码来解决。第二个问题还需要改造Saiku3.14存放Schema文件的方式，Saiku3.14默认将Schema配置文件和Saiku文件存在本地的一个叫Jackrabbit的文件数据库中，我们需要将其改造成存放在Mysql中，并解析Mondrian4的Schema得到指标，和之前报表系统的权限控制系统打通来进行指标权限控制。

在解决了以上问题后，下面给大家展示一下接入Kylin的报表系统。这是Saiku3.14的前端，通过拖拽左边的指标和维度，能够在右边立刻生成我们需要的报表，这里查看的是两天分城市的月流水，和订单数据，在几秒之内就能够返回，且不会给集群造成压力。

Apache Kylin 在链家 GAIA 大数据平台中的实践

作者 李世昌 张如松



GAIA作为链家大数据服务的一站式平台，承接和管理集团几乎全部的数据资源，提供数据治理、数据任务、数据集群三大方向的管理能力，并且肩负着建设链家指标体系的重任，其中指标系统提供便捷自助的数据获取方式和丰富多彩的报表展示样式，很方便业务方、数据分析师挖掘并发挥链家数据的价值。

在链家GAIA大数据平台中，主要使用Kylin做数据引擎，驱动指标平台的数据呈现，本文将详细介绍Kylin在指标平台中的应用实践，也会分享使用Kylin中遇到的问题和解决方案，希望给自主研发BI系统的技术团队，带去一些思想上的火花和实践中的帮助。

GAIA大数据平台通过半年研发，已于近期上线并已内部推广使用，包含指标报表平台、元数据管理、Adhoc查询、任务调度等多个系统。其中指标报表平台负责沉淀和管理链家网的核心指标体系，并且可以快速创建多维报表，支持上卷下钻、维度对比等灵活分析的报表查询。效果如下图所示：



图1 链家GAIA大数据平台 指标平台数据呈现效果

早期数据报表之痛点

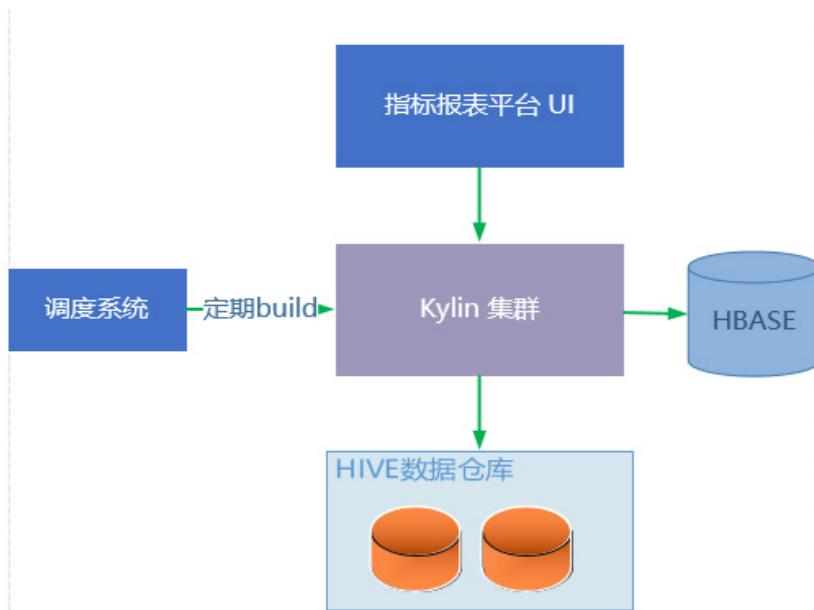
在指标报表平台上线之前，链家网在数据分析方面有三个痛点。第一，是面对公司业务的迅速增长，case by case的报表开发难以快速满足用户需求；第二，是缺乏大数据量快速查询、多维度对比分析的工具，用户获取数据后一般在线下进行深入分析，缺乏数据安全的有效管理；第三，

是数据指标定义混乱，缺乏统一的管理和权威定义，导致一些数据指标计算结果不一致，让用户产生歧义。

为了解决这些问题，更好地发挥数据的价值，大数据工程团队内来自百度、微博、搜狗的同学通过交流和分享业界经验，先后调研了Impala、Drill、Presto等查询引擎，最后决定使用Apache Kylin 作为指标报表平台OLAP查询引擎。如何做此选择？首先是因为Kylin是基于Hadoop生态的开源分布式分析引擎，能够支持TB级别数据量的亚秒级响应，并在2015年12月成为Apache顶级项目，未来想象空间巨大。其次是业内已经有京东、百度、美团等等公司在生产环境使用，从各项反馈看都还不错，并且kylin的核心开发团队是中国团队，在未来深入沟通交流时，也能更加方便与及时。

指标报表平台技术方案

通过自研的指标报表平台，用户可以自助配置报表，报表平台会将用户对报表的维度、指标操作转换成SQL发给Kylin API 进行查询，Kylin内部的查询引擎会从HBase获取数据后，返回给报表平台，然后由前端UI进行渲染展示。组成架构如下图所示：



数据分析师会根据业务的需求，经过深入分析和计算，在Hive仓库中构建OLAP星型模型，然后使用Kylin来配置Model和Cube。Cube创建完成后，通过自研的任务调度系统配置定期build Cube的任务和上游的数据依赖，最终在指标平台页面的报表就可以查询到最新的数据。整个环节流程清晰，不会有太多依赖系统，故而实现成本相对可控，OLAP的数据处理细节，都被封装在Kylin系统中，也让我们更专注精力在建设指标平台上。当然，在上线前的最后冲刺阶段，为了最求更高性能与稳定性，我们也深入到kylin引擎内部，着手调优这套OLAP引擎，下面为大家介绍具体使用Kylin中遇到的一些问题，以及解决办法。

使用 Kylin 过程中遇到的问题及解决方案

我们使用的kylin版本为apache-kylin-1.6.0-hbase1.x-bin，对应的hadoop集群为apache hadoop-2.4.1， hbase 版本是1.1.2。在使用Kylin的过程中，遇到过一些问题，这里分享出来，欢迎大家一起交流探讨。

问题一： Kylin SQL查询中遇到的诡异问题，现象描述， 查询sql如下

```
select
    cast(SUM(pv) as double) as pv,
    cast( count(distinct user_id) as double) as user_id
from olap.olap_log_accts_page_di
inner join DIM.DIM_LOG_USER_LOCATION on DIM.DIM_LOG_USER_LOCATION.user_city_
code=olap.olap_log_accts_page_di.location
where DIM.DIM_LOG_USER_LOCATION.user_region_name in ('华东')
group by DIM.DIM_LOG_USER_LOCATION.user_region_name
```

这个sql能执行成功，但仅仅把 in ('华东') 改成 in ('华东','华南') ,则sql执行报错。错误信息如下所示：

```
Caused by: java.lang.NullPointerException
at com.google.common.base.Preconditions.checkNotNull(Preconditions.java:191)
at org.apache.kylin.storage.hbase.cube.v2.HBaseReadOnlyStore$1$1.
next(HBaseReadOnlyStore.java:131)
at org.apache.kylin.storage.hbase.cube.v2.HBaseReadOnlyStore$1$1.
```

```

next(HBaseReadOnlyStore.java:99)
at org.apache.kylin.gridtable.GTFilterScanner$2.hasNext(GTFilterScanner.java:88)
at org.apache.kylin.gridtable.GTAggregateScanner.iterator(GTAggregateScanner.
java:139)

```

在发现问题后，我们使用官方提供的sample数据，也能稳定复现。经过反复排查和试验，并向Kylin社区发邮件咨询，在Kyligence公司开发同学的技术支持下，最终定位是hbase 1.1.2版本的bug，该问题详细记录在<https://issues.apache.org/jira/browse/HBASE-14269>。定位问题后，我们升级hbase版本后，顺利得到解决。

问题二：对于count distinct 指标，要保证在任意维度和时间范围内的统计准确性，需要配置使用global dictionary。在Kylin1.6版本及以上可以很方便地配置，具体原理参见：<http://kylin.apache.org/blog/2016/08/01/count-distinct-in-kylin/>，但同时也需要注意一些限制：

- 由于global dictionary 底层基于bitmap，其最大容量为Integer.MAX_VALUE，即21亿多，如果全局字典中，累计值超过Integer.MAX_VALUE，那么在Build时候便会报错。即使用全局字典还是有容量的限制。
- Count distinct指标字段的字符串长度不能超过255，否则在build cube的第四步Build Dimension Dictionary时会报错，错误信息“at org.apache.kylin.dict.CachedTreeMap.writeValue(CachedTreeMap.java:240)”。根据社区jira上该问题相关issue，在Kylin 2.0版本已解决该问题。
- 如果有多个指标配置global dictionary，那么设置Reuse属性时需要慎重，只有当每次build时第一个指标列的取值都能完全覆盖其他指标列时，才能设置Reuse。否则，在build的时候会报错“org.apache.kylin.dict.AppendTrieDictionary:Not a valid value.”。

问题三：当Cube维度个数超过20后，数据量急剧增长，build时间也很长，需要进行优化。实际测试如下，一张用户访问日志olap事实表，原始hive表一天数据量大小 600M，选取20个维度，build cube 后数据量

达到 20G，且 build 耗时 300 分钟。为了加快 build 速度和优化存储空间，对 cube 进行了一系列优化设置：把 20 个维度按照业务分析组合拆分成 3 个 Aggregation Groups，同时将 cardinality 很小的维度列放在一起配置成 Joint Dimensions。

经过优化后，build 速度降到 200 分钟，数据量也降到 15G 左右。在早期使用指标平台时，新用户常常会提出许多维度上的需求，这块也需要尽量沟通充分，让用户制定有效有限的维度方能让 Kylin 性能发挥到最佳。

在分享了上面的几点问题与解决办法后，这里也分享一下 Kylin 目前版本的限制，帮助刚开始使用 Kylin 的技术朋友，尽量避免因为这些限制导致研发的反复。

Kylin 目前版本存在的一些限制

在应用 kylin 的过程中，逐步熟悉目前版本建立 cube 的规则后，需要对数据仓库中建立 OLAP 模型有一些限制。接下来总结目前我们遇到的问题，供大家在创建事实表与维表时加以借鉴。

1. 事实表中的不同维度字段不能关联同一张维表

在很多情况下同一张 olap 表中，可能会有多个相同类型的维度字段。比如用户所在地和用户访问地两个维度，需要关联同一张城市维度表。但是由于创建 model 时，关联的维表只会出现一次，两个同类型字段不能关联同一张维表。目前有两种解决方案，一是对该维表建立视图，两个字段分别关联维表和维表的视图。二是根据字段的业务属性，建立不同的维度表，不同的字段关联不同的维表。这两种方式同样都会增加建模时的复杂度。

2. 不同维表中的字段名不能相同

在某些情况下不同维表中的字段含义可能相同，比如经纪人表和客源用户表中的名称字段可能都叫 name。在创建 cube 选择维度时，由于 kylin 用字段名进行了去重，导致其中一张维表中的 name 字段会被过滤掉。目前的解决方案是不同维表中的字段根据业务含义命名，譬如经纪人表的

name命名为agent_name，客源用户表的name命名为cust_name。

3. 修改cube 增加维度时会造成cube元数据不同步

当修改一个cube增加新的维度字段后，cube build能成功完成。但是当查询语句中包含该新增加的维度时，会报如下错误：Not a valid ID。该维度并未包含在cube的元数据中。所以在使用kylin的过程中，应尽量避免在cube上做修改，建议新建cube或者clone cube后进行修改。

4. cube并行build问题

Kylin1.6开放了并行了build的功能，但是当对某些字段设置了global dictionary后，在Build Dimension Dictionary时可能会问题。并行build时应避免同时进行这一步。

GAIA 指标平台上线后效果

指标报表平台上线后至今，刚刚一周多时间，目前已构建10个Cube，配置了20多张指标报表，每天新增数据量200G，90%以上的数据查询在1s内返回，UI层面支持用户上卷下钻、多维度多指标对比分析。

目前的指标平台，有效的填充了之前公司内这块的空白，提供了全公司内统一的指标、维度的定义与管理，提供了具有在线分析能力的指标数据查看能力，并且结合GAIA大数据平台权限系统，有效控制了各项数据的查看权限，在保障了数据安全的同时，提供了数据查询的方便，是一剂推进链家进入数据时代的强心针。

GAIA 指标平台与 Kylin 的展望

随着GAIA大数据平台在集团内的推广，使用指标报表平台的用户逐渐增多，未来有望支持到全公司所有职能部门，也有可能在未来将这套系统普及到链家十五万经纪人手中，指标系统必将会是公司内核心的数据系统，Kylin在该系统中的价值绝不一般。

Kylin在大数据量和页面响应效率上的确表现良好，虽然在构建Cube中也存在一些限制，但相信Kylin团队会继续完善，期待新的版本包含更

强大的功能。近期关注到Kylin团队即将发布的2.0版本会大幅提升数据构建速度，我们也将时刻关注2.0的正式版发布。

关于指标报表平台在接下来的迭代中，会尝试支持相同纬度下的跨cube join，让用户查询数据的范围更宽广更灵活，同时也会简化cube的配置，方便分析师快速配置cube并产出报表。在最近几个月中，我们也会将历史查看数据明细的能力，集成到GAIA指标平台中，很多将要实现的功能正在项目组内开展，希望未来有机会与大家做更详细、更新一步的分享交流！

作者介绍

李世昌，链家网资深研发工程师，先后负责链家网Merlin系统、指标系统、数据权限系统等，负责大数据平台应用研发团队，对大数据BI系统、Data Streaming有一定研究。

张如松，链家网高级研发工程师，2015年加入链家网，参与大数据平台多项研发工作，负责GAIA指标平台引擎部分工作。

AWS 上 Apache Kylin 调度系统的设计

作者 张晨

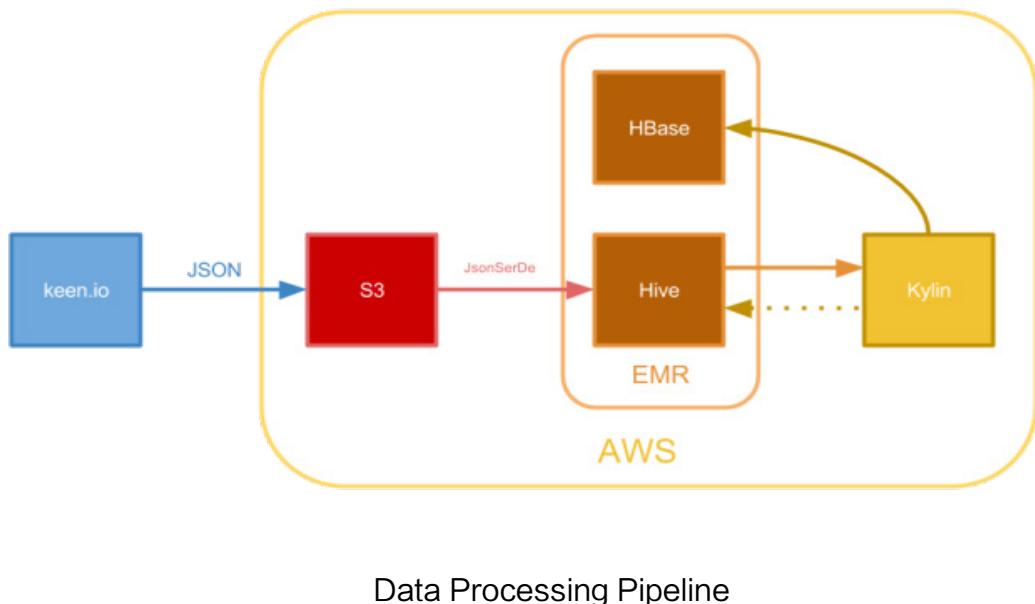


1. 背景介绍

Strikingly 是一家为用户提供建站服务的初创企业，目前的数据平台主要处理的是用户所建立网站的访问者信息统计，可以认为是一套简单的 Google Analytics 服务。这套系统使用 Keen IO 收集访问者信息，使用 Kylin、Hadoop、Hive 等技术处理海量数据，整套系统都部署在 AWS 上，深度使用了 EC2、ECS、ELB、EMR 等 AWS 服务。除了收集访问者的 Page View 信息，这套系统还会处理一些用户付费行为相关的 e-commerce 信息等。

1.1 数据处理流程

下图是目前系统的数据处理流程的一个简化版本：



以用户的 Page View 数据处理为例，主要有以下几个步骤。

1. 数据首先由内嵌在页面当中的 JS 脚本收集到第三方服务 Keen IO 那里。
2. 通过 Keen IO 的数据导出功能，将 Page View 数据以五分钟为单位，使用 JSON 格式打包放置在 AWS S3 上的指定目录。
3. 通过为 Hive 配置 JsonSerDe，我们可以将第二步当中的指定目录直接虚拟成一个 Hive 表，从而可以在 Hive 上使用 SQL 语句进行查询操作。
4. Apache Kylin 控制 EMR 上的各种组件，如 Hive、Hadoop 等进行处理，将数据处理的结果保存在 HBase 表当中，以备之后取用。

当数据被处理好保存在 HBase 上之后，我们就可以调用 Kylin 所提供的 API 对这些数据进行远超过以往的速度的访问，从而支撑每个 Strikingly 或上线了用户对自己网站统计数据的查询要求。

1.2 Apache Kylin 简介

从上面的数据处理流程当中可以看出，Kylin 无疑是整套系统的核心：

一方面它控制了对 Hive 当中原始的数据进行处理并缓存到 HBase 的全过程，另一方面它还要服务用户对处理后的数据大量的查询请求。在这里有必要对 Kylin 及其基本概念进行一个简单的介绍。

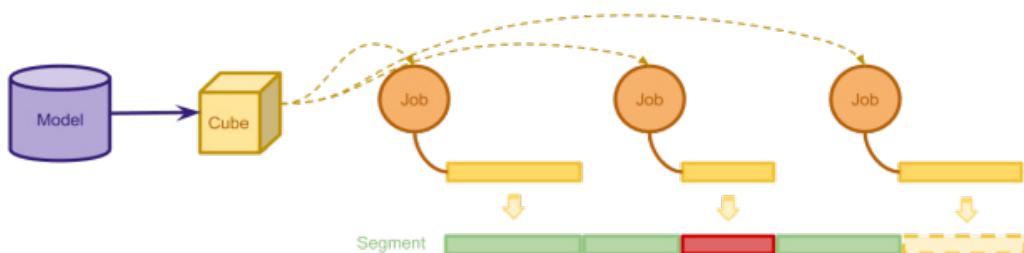
Apache Kylin 是一个开源的分布式数据分析系统，它和 Hadoop、Spark 等一样，都是 Apache 基金会的顶级项目。Kylin 可以被认为是一种 ETL 工具，我把它主要完成的任务概括为以下几点。

根据预先定义好的模型，将保存在 Hive 表或其他来源的数据提取出来。

根据预先定义好的模型关系，将数据按照一定的算法进行变换处理，使得数据以一种易于查询的方法存在。

当用户请求到来时，通过分析用户请求，将查询分解成可以作用于变换后数据格式的任务，从而快速的得到处理结果。

在第 2 点当中 Kylin 使用的技术被称为 Cubiod。这一技术有点像根据模型的定义，预先穷举用户查询的所有可能结果，然后将这些结果以合适的方式组织起来。当然，在实际实现当中，为了更加有效地执行这一任务并将其抽象为可以运行在 Hadoop 集群上的 MapReduce 任务，Kylin 所实现的算法要复杂很多。在此我们不对 Kylin 的技术实现进行深入的介绍，但是为了继续之后的内容，必须要了解 Kylin 当中四个重要的概念：Model、Cube、Job 和 Segment。



Model, Cube, Segment, Job

上图给出了这四个概念的一个关系说明，简单来讲：

- Model 是对数据来源的描述，比如说我们想要进行查询的数据表

的主表是什么，想要和主表 Join 起来的辅助表是什么等等。值得注意的是，在 Model 里，我们要为模型指定一个列(或者称为维度)作为这个模型的 Range Key(或者称为 Partition Key)。所谓 Range Key，就是说根据这个维度，可以比较好地将数据划分为多个不相交的部分。这个 Key 在之后 Segment 的部分也会用到。

- Cube 是对转化和处理数据的方法的描述。在 Kylin 里，我们并不需要手动编写转化的程序，而是通过指定需要进行操作的 Model， 提供一些用户可能进行的查询和统计的指标(比如用户可能在哪些字段上使用 GROUP BY， 在哪些字段上进行 COUNT 或 COUNT_DISTINCT 统计等)， Kylin 会自动根据这些定义和数据模型本身的特点，构造出合理的执行流程。
- Job 是在 Cube 的控制下， Kylin 执行的一次任务处理。它包括了数据处理过程当中，数据抽取、数据变换、数据储存、 垃圾清理等的全过程。Job 的类型包括 BUILD 、 REFRESH 、 MERGE 等。
- Segment 是一个 Job 执行的结果。对于一个 Cube 来说，他的 Segment 可以根据 Range Key 划分为多个区间分别构建，当某一区间的 Segment 构建成功之后， Kylin 就可以实现对这一区间当中数据的快速查询。而构建当中或者构建失败的 Segment 无法提供查询服务。对于横跨多个 Segment 的查询， Kylin 会分别执行对各个 Segment 的查询，再将结果合并起来。我们可以 REFRESH 一个 Segment 以更新其中的数据，或者 MERGE 多个 Segment 以避免跨 Segment 查询带来的性能损失。

当我们根据上述模型构建了 Segment 之后， Kylin 就可以通过 SQL 接口对 Model 指定的数据表进行快速的查询和分析了。值得注意的是，我们还可以以集群的方式部署 Kylin。一个 Kylin 集群可以有一个 Job 节点和若干个 Query 节点组成， 其中 Job 节点单独用来做 Cube 构建的编排控制操作， Query 节点则用来承载用户的查询请求。

1.3 对调度系统的需求

尽管 Kylin 的存在使得我们承载用户 Page View 查询的数据处理平台成为可能，在系统的开发和部署当中仍然遇到一些问题，导致开发一个集中式的调度系统成为必要。在实践当中遇到的需求可以总结为以下几条：

定制化的任务调度，在很多情况下，Kylin 自带的调度系统不能很好的满足我们的需求，这是我们自行实现调度系统最本质的原因。我们遇到的问题主要包括以下几点。

- 由于数据从收集到导入 Kylin 之间的步骤比较多，每个步骤都需要调度控制协调运行。比如说，由于历史设计的原因，在激发 Kylin 构建一段时间区间内的数据时，我们需要先对应的 Hive 表进行一次刷新的操作，这个操作不能由 Kylin 来控制。
- 在构建的时候，我们希望不同 Cube 的任务可以并行运行，但是同一个 Cube 的任务必须串行执行。这是因为，在我们的系统设计中，有一类调度任务需要获取当前 Cube 的各项状态，用于规划之后的构建工作，允许同一 Cube 的任务并行可能导致这类任务获取到的当前状态无效，简便起见我们希望避免这样的情况。
- 有一些特别的任务，如元数据的备份、HBase 表的清理等需要 Block 所有其他的任务。这是因为在这些任务执行的时候需要对系统整体的元信息做一个查询或变更，如果在这一过程中有其他任务正在执行会产生并发一致性问题，不但可能导致所得数据不完整，还可能损坏整个系统。
- 之前提到，横跨多个 Segment 的查询速度可能比较慢，一个可能的优化思路就是根据用户请求的特点，对某一 Cube 的 Segment 实行更有针对性更积极的 Merge 策略。截至目前 Kylin 本身还没有提供对这一需求良好的支持。因此自己实现 Merge 任务的调度也就比较有必要了。

运维自动化，在 Strikingly，我们不但重度使用 AWS 的服务，还使用 Terraform 和 Docker 容器等工具来实现运维自动化，这些自动化工具要求我们系统的每个部分尽可能地无状态和可配置。尽管 Kylin 本身将自己的元数据保存在 HBase 上，但是仍然有一些对运维自动化的挑战。

- 包括 Kylin 在内的 Hadoop 组件深度依赖 XML 或 Java Properties 配置文件进行定义，这对容器镜像的复用造成了一定的阻碍。比如说我们需要将这套系统分别部署在 AWS 的两个不同的 Region 中，理想的情况是可以使用环境变量来定义 IP 地址等的变化，而不用构建不同的容器镜像
- Kylin 目前集群部署的设计对于 Auto Scale 不友好。这主要是由于 Kylin 需要在集群的 Job 节点硬编码 Query 节点的 IP 地址和端口，并在 Cube 构建任务结束或者元信息变化之后通知 Query 节点，以便清理缓存更新数据。这一点大大地限制了 ECS 服务在进行容器编排和 Auto Scale 时的灵活性。
- 在实际中需要使用的一些 Kylin 功能(如元信息备份工具等)只能通过命令行调用，而在在容器中使用简单的 crontab 脚本调度这些命令既笨重又容易出故障。

系统的健壮性，我们的数据平台系统在遇到故障或需要上线新的版本的时候不可避免的需要重启。这样的情况下就可能出现本应被调度的一个事件不幸错过现象。在以前，我们需要人工地辨识这种情况并手动应用这些事件，但是在动态运行的系统当中，手动的工作往往是费事且令人手忙脚乱的。理想状态下我们的调度系统应该具备一定的自我恢复能力，能够在重启之后自主地发现错过的任务从而恢复他们。

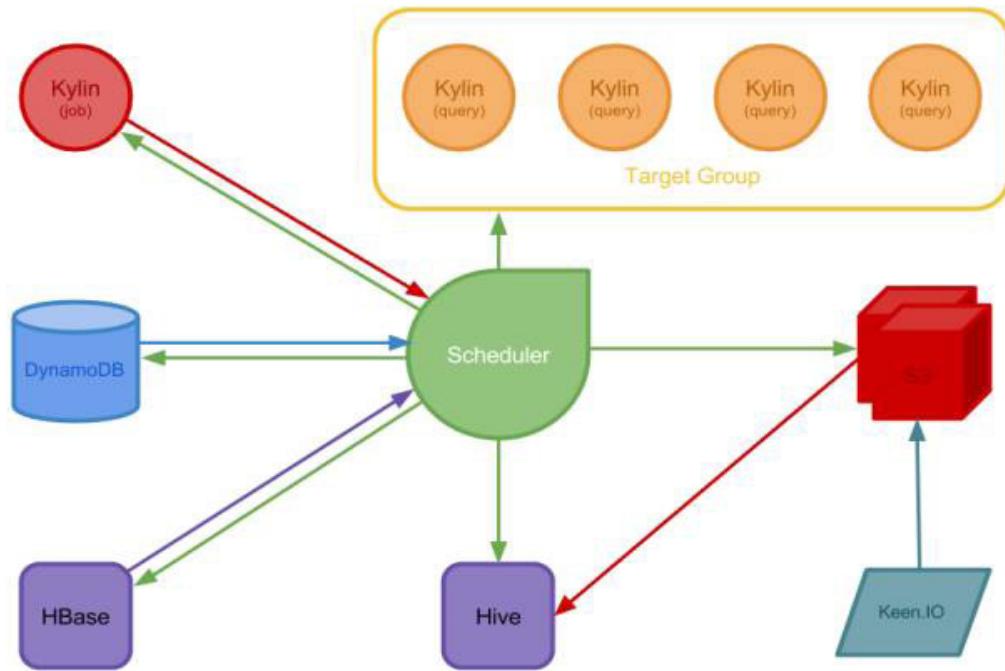
综上所述，实现一个集中式、功能丰富的调度系统势在必行。

2. 调度系统的设计

调度系统的整体设计如下图所示：

图中箭头的方向指示了数据流动的方向和调度器控制组件的方向。为

了满足前文提到的各种需求，我们的调度系统有如下的特点。



Scheduler Overall Design

调度器集中地调度和控制除 Keen IO 之外的各个组件。在激发任务后监控并等待任务完成，以便使所有任务协调有序执行。

Kylin 集群中，Job 节点不再与 Query 节点进行直接的交互，而是由 Scheduler 通过 AWS SDK 获得存在于指定 Target Group 中 Kylin 节点的地址信息，直接控制 Cache 刷新等工作。

调度器使用 DynamoDB 保存运行状态，因故重启之后自动读取记录恢复，从而获得更强的健壮性。保存在 DynamoDB 当中的记录还可以用来弥补任务调度当中出现的缺口。

调度器直接通过查询 Kylin Job 节点获得 Cube 和 Segment 的完整信息，自行连接 HBase 完成 Kylin 元信息表的备份和数据表的清理工作，从而避免执行命令行工具。备份数据将会被直接放置在 S3 上。

调度器和 Kylin 节点都使用 Docker 容器部署。对于 Kylin 节点来说，

我们使用 Python 脚本自行编写了启动器，这一脚本可以在启动时通过环境变量和预先提供好的配置模板，先进行字符串替换生成由环境变量定制过的配置文件，再启动 Kylin 和其他 Hadoop 组件，从而可以在不同场合使用单个容器镜像。

3. 调度系统的实现

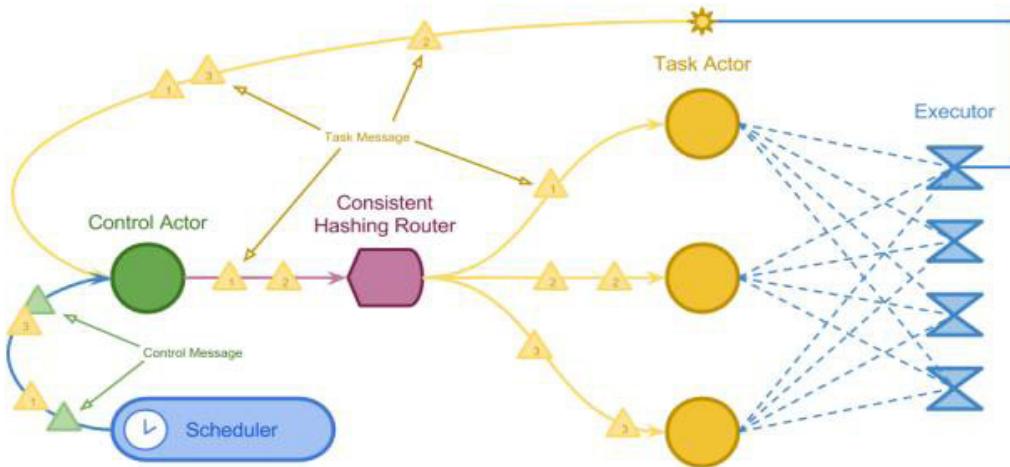
我们决定使用 Scala 编写和实现调度器，以便利用 Java/JVM 上的各种工具和库连接和管理包括 AWS、HBase 和 Hive 在内的各个组件。除了 AWScala、Joda-Time、Spray-JSON 等工具库之外，我们使用的最重要的框架是 Akka。

Akka 是一个 Scala/Java 下 Actor 并发模型的一个开源实现。Akka 实现的 Actor 模型受 Erlang 编程语言的启发，每个 Actor 相当于一个可以执行任务的对象，这个对象执行的具体任务由它接收到的消息(Message)决定。这些对象都包含一个并发安全的队列作为信箱，信箱中的每条消息将会被 Actor 串行地处理，Actor 接受到消息之后也可以再转发给其他的 Actor。我们的调度器将会利用这些特性实现并发任务调度和管理。

Akka 还提供了一种名为 ConsistentHashingRouter 的组件。它本质上是一个专门用来分发消息的 Actor。ConsistentHashingRouter 可以利用消息本身提供的哈希键来分发消息到下游的 Actor，它可以保证具有相同哈希键消息一定会被分发到同一个 Actor 上。我们可以利用这一特点来保证我们对于某一 Cube 的调度操作都是串行的。Akka 的 ConsistentHashingRouter 还支持包括 Auto Scale 在内更多丰富的功能，在此不再赘述，下面是调度器的 Actor 系统的架构图：

调度器的执行模块主要包括 5 个部分：

1. ControlActor 是调度器的重要组成部分，每个任务消息都会首先到达 ControlActor。ControlActor 会对消息进行一些预处理。所有的 ControlMessage 都会由 ControlActor 直接执行，不会被传递给下游。



Actor System Structure

2. **ConsistentHashingRouter** 是一个分发器，它将接收到的 **TaskMessage** 以一致性哈希的方式分发给下游的 **Actor**。这种分发保证同一个 Cube 的消息一定会在同一个 **Actor** 当中串行执行，同时也具备一定的并行性。
3. **TaskActor** 是真正执行调度任务的 **Actor**。所有的 **TaskActor** 都是相同的，它们接收到消息之后也会进行一些预处理，然后根据消息指定的 **Executor** 名称，选择正确的执行器进行执行。如果执行失败，**TaskActor** 也负责捕捉错误并进行一些错误恢复和 **Log** 的处理。
4. **Executor** 是具体执行调度任务的代码逻辑所在的地方。一般来讲所有的 **Executor** 都是单例，接收到对应的消息后，它们对消息的内容进行解析然后调用各种 **Service** 进行执行。**Executor** 根据自己的需要还可以进一步生成新的 **TaskMessage** 传输给 **ControlActor**，从而 **Spawn** 出新的任务。
5. **Scheduler** 是定时器，它定时地生成一些消息传递给 **ControlActor** 从而达到定期执行任务的目的。

除了以上的几个部分，调度器还有一类模块称为 **Service**，用来抽象一些共享的代码逻辑(比如对 AWS 常用操作的整合等)。

首先可以看到，在调度器当中流动的任务消息有两种: ControlMessage 和 TaskMessage。

3.1 ControlMessage

ControlMessage 是一类用来维护和管理调度器状态的消息，它指示 ControlActor 执行一些消息的维护过程。比如说 Recover 消息指示 ControlActor 从 DynamoDB 当中抽取任务消息的状态以查看是否有需要恢复的任务。如果有，它会将这些任务分发给下游执行。ControlMessage 只会由 ControlActor 执行且不会被记录到 DynamoDB 上。

ControlMessage 在 Scala 被定义为一系列的 Case Class。

3.2 TaskMessage

TaskMessage 是描述实际调度任务的消息(如指示 Kylin 进行一次 Cube 构建等)。它的 Scala 类定义如下。

```

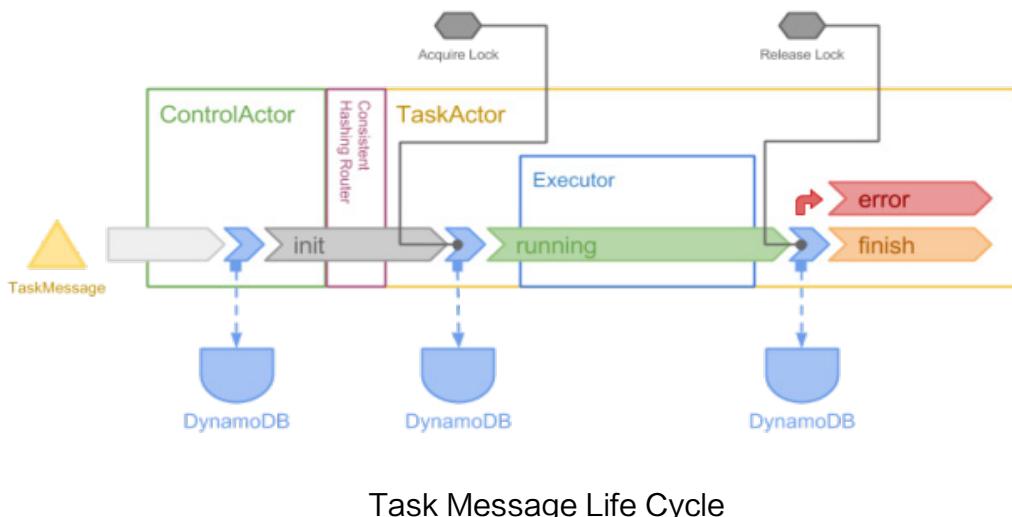
1 case class TaskMessage(
2   uid: String,
3   name: String,
4   hashKey: String,
5   data: Map[String, String]
6 ) extends ConsistentHashable {
7   def blocking = hashKey == null || hashKey.isEmpty
8
9   override def consistentHashKey = if (hashKey == null) "" else hashKey
10 }
```

TaskMessage 包含一个 uid 用来全局唯一地标记一个消息，name 字段指定了 Executor 的名称，hashKey 字段用来提供。

给 ConsistentHashingRouter 作为哈希的参考，一般是相关 Cube 的名称。如果 hashKey 的值是空的，我们就认为这个任务是阻塞的，也就是说在执行它之前，所有其他的任务都要结束，在它执行完成之前，其他任务都需要等待。data 字段是一个 Map[String, String] 类型的字典，用来保存一个任务消息的具体参数。

受函数式编程思想的影响，所有的 TaskMessage 都是 Immutable 的。也就是说一旦被构建出来，一个 TaskMessage 所包含的任何字段都不会改变，因此可以保证在整个处理流程当中任务消息本身不存在副作用。

然而 TaskMessage 在运行过程中为了记录和恢复的需要，存在一个生命周期的概念。也就是说这类消息在生成之后，会经历 init、running、finish 或 error 等几个生命周期。ControlActor 通过查询一个任务消息的生命周期状态来决定在重启恢复时是否需要恢复一个任务。下图展示了一个 TaskMessage 在处理流程各个部分生命周期的变化：



Task Message Life Cycle

简单来说，一个任务消息在进入 ControlActor 后会被转化为 init 状态，经过分发到达 TaskActor 后，会在实际执行前被修改为 running 状态，在执行结束后根据执行状况可能被标记为 finish 或 error 状态。这些状态会和消息定义本身一起被直接更新到 DynamoDB 当中以备之后查询利用。在这里，我们没有使用 Write Ahead Logging 的方式记录这些消息状态的变化，因此在某些情况下仍然可能出现记录丢失等问题。但是考虑到实现的简洁性和实际需要，目前的解决方案应该足够稳定了。

在 TaskMessage 进入执行状态之前会先利用 GlobalLockService 获得执行锁。我们目前使用一个简单的读写锁实现我们的需求：非阻塞任务将会尝试获得读锁，因此非阻塞任务可以并行执行。阻塞任务将会试图获得写

锁，因此阻塞任务和任何其他任务都是互斥的。

3.3 任务类型和Executor

TaskMessage 的类型一一对应于不同的 Executor。接下来我们结合不同的 Executor 来介绍不同调度任务的类型。

任务调度最重要的两个类型是 PlanDataRefresh 和 PlanCubeMaintenance。这两个任务并不具体执行需要完成的调度目标，而是通过请求 Kylin、AWS 等 Service 获得对当前系统状态的了解，通过这些信息决定如何执行真正的调度任务。这两个任务在执行之后将会 Spawn 出新的消息任务导回 ControlActor 等待执行。

PlanDataRefresh 用来规划数据的刷新，它会先生成 HiveTableRefresh 任务用于刷新上一个小时进来数据的 Hive 表，之后会根据当前时间片段覆盖的 Segment 来决定激发什么类型的 Cube 构建任务。

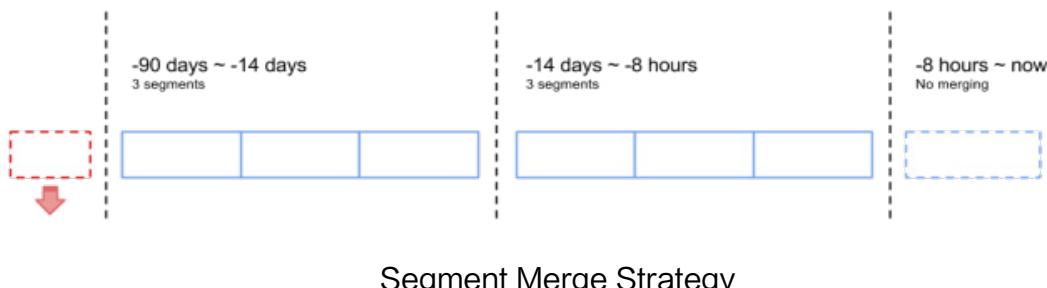
- 当前处理的时间片段没有 Segment 覆盖，将会生成一个长度为 4 个小时的时间片段并在之上激发一个 KylinCubeBuild 任务。
- 当前处理的时间片段已经有 Segment 完全覆盖，将会针对覆盖这一时间片段所有的 Segment 执行 KylinCubeRefresh 任务。
- 在上一种情况之外，如果处理的时间片段有一部分没有被覆盖，将会在没有覆盖的部分激发 KylinCubeBuild 任务。

PlanDataMaintenance 任务用来扫描指定 Cube 的 Segment 和 HiveTableRefresh 任务执行的情况，根据所得信息决定是否进行一系列的维护操作。主要进行的操作有：

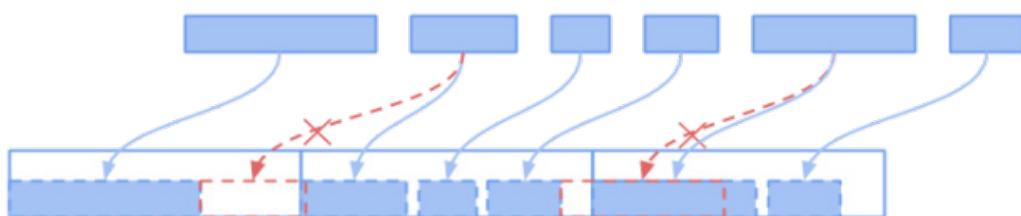
- 通过 Kylin API 获得 Cube 所有 Segment 的覆盖情况。如果存在没有被覆盖到的缺口，则激发 KylinCubeBuild 任务弥补缺口；
- 通过查询 DynamoDB 当中 HiveTableRefresh 任务近期执行的状况。如果发现没有执行过 Hive 表刷新的时间窗口，则在这些窗口上执行 HiveTableRefresh 和 KylinCubeRefresh 等任务；
- 根据 Cube 所有 Segment 的时间片段分布情况，决定是否进

行 Segment 的 MERGE 操作。如果需要进行合并，则激发 KylinCubeMerge 任务。

根据观察，Strikingly 的用户最常查询的数据为最近一周左右的，我们可以根据这一特点，按照过去距今时间的长短设定 Segment 数量的大小，以使得密集查询区间内的 Segment 比较少，又不至于过于频繁地激发合并操作。Cube 的 Segment 数量采用如下的策略进行分配，以使得每一个 Cube 的 Segment 数量维持在 10 个左右。



在每一个时间段内，决定如何合并 Segment 则是通过一个简单的贪心算法实现的。我们首先将时间段均匀划分为三个桶，按照时间顺序尽可能多地向一个桶里放置 Segment，如果一个 Segment 不能被放进前面的桶，再将其放到新的桶中。下图展示了这一过程：



Segment Merge Algorithm

值得注意的是，像 KylinCubeBuild、KylinCubeRefresh 这样的任务会监控对应任务的执行状态，只有当 Kylin 中对应的任务执行完成之后才会结束。由于 Kylin 的 API 是异步的，我们通过循环等待的方法等待任务的结束。这样也可以保证任务在并行状态下的有序协调执行。同时，这些任务在结束时也会通过 AWS SDK 获得 Kylin 的 Query 节点的地址信

息，通过调用 Kylin 有关 Cache 管理的 API 直接广播通知状态的改变，从而取消了在 Kylin 的 Job 节点对 Query 节点地址信息的依赖，使得 Query 节点可以 Auto Scale。

另外两个值得介绍的任务是 KylinMetadataBackup 和 KylinHBaseTableCleanup。这两个任务之前只能通过调用 Kylin 的命令行工具执行。通过查看 Kylin 的源代码，我们发现这两个任务较为简单，可以直接在我们的调度器当中实现。

KylinMetadataBackup 任务用于备份 Kylin 的元信息。Kylin 的元信息除了一些基本配置之外，还包括每一个 Cube、Segment 和 Job 等的定义和构建信息等。默认情况下 Kylin 将所有的元信息保存在 HBase 上的 kylin_metadata 表中，在调度器当中实现备份功能的原理是直接将 HBase 的 Client 集成进来，通过 Scan 这一 HBase 表，将每一条记录 Dump 下来。Kylin 的元信息都是以文件路径和 JSON 内容的形式存在的。我们直接以文件目录的方式将这些数据打包成 GZ 包。最后生成的 GZ 包将会直接被上传到 S3。

KylinHBaseTableCleanup 任务用于清理多余的 HBase 表。这是因为在某些情况下(如 Cube Refresh、系统故障退出等) Kylin 构建任务产生的中间表或过期的表会遗留在 HBase 当中。而 HBase 存在表数过多会拖慢访问速度的问题，因此必须定期清理这些表。Kylin 清理这些表的逻辑也很简单。那就是在没有任务执行的情况下扫描所有 Cube 和 Segment 等，过滤出没有被任何 Cube、Segment 或 Job 引用的 Kylin 数据表，然后将这些表清理掉。我们在调度器当中实现了类似的功能，通过 HBaseAdmin 和 Kylin API，我们找出所有没有用的表，在第一次清理时先将这些表 Disable，第二次再删除。这样可以留出一段时间作为缓冲，防止错误的数据删除。

显而易见，上述两个任务都需要阻塞所有其他任务的执行。因此它们的 hashKey 都为空。

有关任务类型方面最后值得一提的是，只有前面提到的 Plan 类型的

任务和两个 Kylin 相关的 Backup 和 Cleanup 以及 ControlMessage 会被 Scheduler 定期生成。其他的任务都是由这些任务产生的衍生任务。

3.4 其他

除了调度器本身的实现，我们还使用了 New Relic 的 Application Performance Monitor 服务。通过使用 New Relic 提供的 Java Agent，可以在云上对我们运行的 JVM 实例进行监控和 Profiling，大大方便了我们搜集线上服务运行信息的效率。通过 New Relic 收集的数据，可以对各个任务执行的时间和代码执行瓶颈进行分析，从而指导进一步的策略设计和代码优化工作。

4. 总结与展望

上面就是我们为 Kylin 数据处理平台全新设计的集中式任务调度系统。它满足了我们对数据平台调度器定制化、自动化和健壮性的需求，现在已经开始进行局部上线和测试。目前来看，我们的调度系统和数据平台还有以下一些需要提高的地方。

1. 调度器目前无法手动提交任务。虽然新的调度器引入了对任务并发更好的控制，保证了相同和不同 Cube 的任务可以协调执行，但有时手动执行任务仍然无法避免。更好的实践显然是将调度器作为提交任务的唯一入口。一种可能的解决方案是为调度器实现 Web 接口，从而使得人工的操作也可以通过调度器执行。
2. 目前使用 Keen IO 和 Hive 协调工作的解决方案过于复杂且健壮性差。由于网络延迟等原因，数据从 Keen IO 到达数据平台再到构建结束仍然具有很大的延迟和不确定性，晚到的数据本身也增加了系统实现的难度。可能的解决方案包括精简 Hive 表流程、利用 Kafka 或 AWS Kinesis 等消息队列服务传输信息以及自行收集用户访问数据等。
3. 以 Kylin 为基础的数据平台虽然很好的解决了用户海量查询的问

题，但是实时性较差。即使使用 Kylin 的 Kafka 构建模式仍然无法实现亚分钟级别的实时数据查询。为了解决这一问题，一种思路是引进 Netflix Atlas、Facebook Beringei 或 Yandex ClickHouse 这样的时间序列数据库，用于取代 Kylin 满足对短时数据的实时查询需求。

作者介绍

张晨，Strikingly 数据平台工程师，算法、分布式系统、函数式编程爱好者，Shanghai Linux User Group co-Op，上海交大学生技术社团 SJTUG 创始人。

版权声明

InfoQ 中文站出品

架构师特刊：Apache Kylin实践（第二期）

©2017北京极客邦科技有限公司

本书版权为北京极客邦科技有限公司所有，未经出版者预先的书面许可，不得以任何方式复制或者抄袭本书的任何部分，本书任何部分不得用于再印刷，存储于可重复使用的系统，或者以任何方式进行电子、机械、复印和录制等形式传播。

本书提到的公司产品或者使用到的商标为产品公司所有。

如果读者要了解具体的商标和注册信息，应该联系相应的公司。

出版：北京极客邦科技有限公司

北京市朝阳区来广营叶青大厦北园5层

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系 editors@cn.infoq.com。

网 址：www.infoq.com.cn