大厂实时数仓建设案例

本文档来自公众号：**五分钟学大数据**

微信扫码关注

目录

一、实时数仓建设背景

1. 实时需求日趋迫切

目前各大公司的产品需求和内部决策对于数据实时性的要求越来越迫切，需要实时数仓的能力来赋能。传统离线数仓的数据时效性是 T+1，调度频率以天为单位，无法支撑实时场景的数据需求。即使能将调度频率设置成小时，也只能解决部分时效性要求不高的场景，对于实效性要求很高的场景还是无法优雅的支撑。因此实时使用数据的问题必须得到有效解决。

2. 实时技术日趋成熟

实时计算框架已经经历了三代发展，分别是：Storm、SparkStreaming、Flink，计算框架越来越成熟。一方面，实时任务的开发已经能通过编写 SQL 的方式来完成，在技术层面能很好地继承离线数仓的架构设计思想；另一方面，在线数据开发平台所提供的功能对实时任务开发、调试、运维的支持也日渐趋于成熟，开发成本逐步降低，有助于去做这件事。

二、实时数仓建设目的

1. 解决传统数仓的问题

从目前数仓建设的现状来看，实时数仓是一个容易让人产生混淆的概念，根据传统经验分析，数仓有一个重要的功能，即能够记录历史。通常，数仓都是希望从业务上线的第一天开始有数据，然后一直记录到现在。但实时流处理技术，又是强调当前处理状态的一个技术，结合当前一线大厂的建设经验和滴滴在该领域的建设现状，我们尝试把公司内实时数仓建设的目的定位为，以数仓建设理论和实时技术，解决由于当前离线数仓数据时效性低解决不了的问题。

现阶段我们要建设实时数仓的主要原因是：

- 公司业务对于数据的实时性越来越迫切，需要有实时数据来辅助完成决策；
- 实时数据建设没有规范，数据可用性较差，无法形成数仓体系，资源大量浪费；
- 数据平台工具对整体实时开发的支持也日渐趋于成熟，开发成本降低。

2. 实时数仓的应用场景

- 实时 OLAP 分析；
- 实时数据看板；
- 实时业务监控；
- 实时数据接口服务。

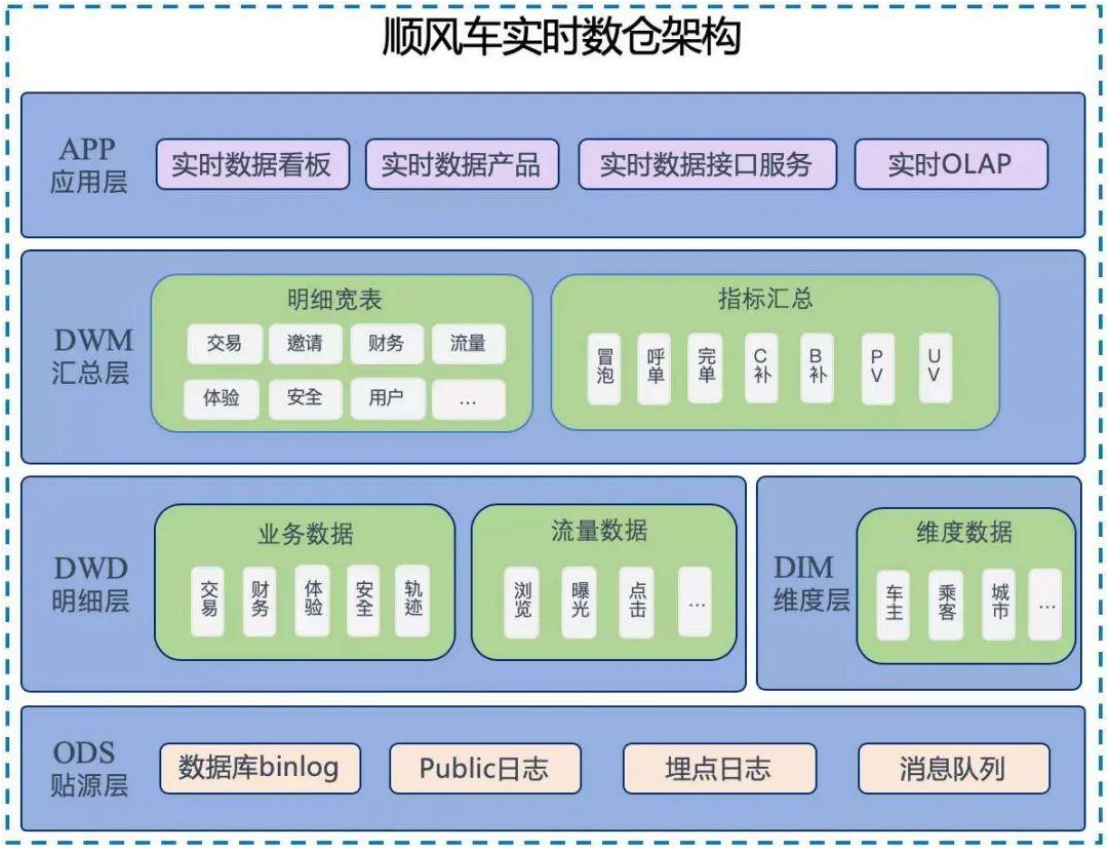
三、实时数仓建设方案

接下来我们分析下目前实时数仓建设比较好的几个案例，希望这些案例能够给大家带来一些启发。

1. 滴滴顺风车实时数仓案例

滴滴数据团队建设的实时数仓，基本满足了顺风车业务方在实时侧的各类业务需求，初步建立起顺风车实时数仓，完成了整体数据分层，包含明细数据和汇总数据，统一了 DWD 层，降低了大数据资源消耗，提高了数据复用性，可对外输出丰富的数据服务。

数仓具体架构如下图所示：



从数据架构图来看，顺风车实时数仓和对应的离线数仓有很多类似的地方。例如分层结构；比如 ODS 层，明细层，汇总层，乃至应用层，他们命名的模式可能都是一样的。但仔细比较不难发现，两者有很多区别：

1. **与离线数仓相比，实时数仓的层次更少一些：**

- 从目前建设离线数仓的经验来看，数仓的数据明细层内容会非常丰富，处理明细数据外一般还会包含轻度汇总层的概念，另外离线数仓中应用层数据在数仓内部，但实时数仓中，app 应用层数据已经落入应用系统的存储介质中，可以把该层与数仓的表分离；
- 应用层少建设的好处：实时处理数据的时候，每建一个层次，数据必然会产生一定的延迟；
- 汇总层少建的好处：在汇总统计的时候，往往为了容忍一部分数据的延迟，可能会人为的制造一些延迟来保证数据的准确。举例，在统计跨天相关的订单事件中的数据时，可能会等到 00:00:05 或者 00:00:10 再统计，确保 00:00 前的数据已经全部接受到位了，再进行统计。所以，汇总层的层次太多的话，就会更大的加重人为造成的数据延迟。

2. **与离线数仓相比，实时数仓的数据源存储不同：**

- 在建设离线数仓的时候，目前滴滴内部整个离线数仓都是建立在 Hive 表之上。但是，在建设实时数仓的时候，同一份表，会使用不同的方式进行存储。比如常见的情况下，明细数据或者汇总数据都会存在 Kafka 里面，但是像城市、渠道等维度信息需要借助 Hbase，mysql 或者其他 KV 存储等数据库来进行存储。

接下来，根据顺风车实时数仓架构图，对每一层建设做具体展开：

1. ODS 贴源层建设

根据顺风车具体场景，目前顺风车数据源主要包括订单相关的 binlog 日志，冒泡和安全相关的 public 日志，流量相关的埋点日志等。这些数据部分已采集写入 kafka 或 ddmq 等数据通道中，部分数据需要借助内部自研同步工具完成采集，最终基于顺风车数仓 ods 层建设规范分主题统一写入 kafka 存储介质中。
命名规范：ODS 层实时数据源主要包括两种。

- 一种是在离线采集时已经自动生产的 DDMQ 或者是 Kafka topic，这类型的数据命名方式为采集系统自动生成规范为：cn-binlog-数据库名-数据库名 eg：cn-binlog-ihap_fangyuan-ihap_fangyuan
- 一种是需要自己进行采集同步到 kafka topic 中，生产的 topic 命名规范同离线类似：ODS 层采用：realtime_ods_binlog_{源系统库/表名}/ods_log_{日志名} eg: realtime_ods_binlog_ihap_fangyuan

2. DWD 明细层建设

根据顺风车业务过程作为建模驱动，基于每个具体的业务过程特点，构建最细粒度的明细层事实表；结合顺风车分析师在离线侧的数据使用特点，将明细事实表的某些重要维度属性字段做适当冗余，完成宽表化处理，之后基于当前顺风车业务方对实时数据的需求重点，重点建设交易、财务、体验、安全、流量等几大模块；该层的数据来源于 ODS 层，通过大数据架构提供的 Stream SQL 完成 ETL 工作，对于 binlog 日志的处理主要进行简单的数据清洗、处理数据漂移和数据乱序，以及可能对多个 ODS 表进行 Stream Join，对于流量日志主要是做通用的 ETL 处理和针对顺风车场景的数据过滤，完成非结构化数据的结构化处理和数据的分流；该层的数据除了存储在消息队列 Kafka 中，通常也会把数据实时写入 Druid 数据库中，供查询明细数据和作为简单汇总数据的加工数据源。
命名规范：DWD 层的表命名使用英文小写字母，单词之间用下划线分开，总长度不能超过 40 个字符，并且应遵循下述规则：realtime_dwd_{业务/pub}_{数据域缩写}_[{业务过程缩写}]_[{自定义表命名标签缩写}]

- {业务/pub}：参考业务命名
- {数据域缩写}：参考数据域划分部分
- {自定义表命名标签缩写}：实体名称可以根据数据仓库转换整合后做一定的业务抽象的名称，该名称应该准确表述实体所代表的业务含义
- 样例：realtime_dwd_trip_trd_order_base

3. DIM 层

- 公共维度层，基于维度建模理念思想，建立整个业务过程的一致性维度，降低数据计算口径和算法不统一风险；
- DIM 层数据来源于两部分：一部分是 Flink 程序实时处理 ODS 层数据得到，另外一部分是通过离线任务出仓得到；
- DIM 层维度数据主要使用 MySQL、Hbase、fusion(滴滴自研 KV 存储) 三种存储引擎，对于维表数据比较少的情况可以使用 MySQL，对于单条数据大小比较小，查询 QPS 比较高的情况，可以使用 fusion 存储，降低机器内存资源占用，对于数据量比较大，对维表数据变化不是特别敏感的场景，可以使用 HBase 存储。

命名规范：DIM 层的表命名使用英文小写字母，单词之间用下划线分开，总长度不能超过 30 个字符，并且应遵循下述规则：dim_{业务/pub}_{维度定义}[_{自定义命名标签}]：

- {业务/pub}：参考业务命名
- {维度定义}：参考维度命名
- {自定义表命名标签缩写}：实体名称可以根据数据仓库转换整合后做一定的业务抽象的名称，该名称应该准确表述实体所代表的业务含义
- 样例：dim_trip_dri_base

---

4. DWM 汇总层建设

在建设顺风车实时数仓的汇总层的时候，跟顺风车离线数仓有很多一样的地方，但其具体技术实现会存在很大不同。

第一：对于一些共性指标的加工，比如 pv，uv，订单业务过程指标等，我们会在汇总层进行统一的运算，确保关于指标的口径是统一在一个固定的模型中完成。对于一些个性指标，从指标复用性的角度出发，确定唯一的时间字段，同时该字段尽可能与其他指标在时间维度上完成拉齐，例如行中异常订单数需要与交易域指标在事件时间上做到拉齐。

第二：在顺风车汇总层建设中，需要进行多维的主题汇总，因为实时数仓本身是面向主题的，可能每个主题会关心的维度都不一样，所以需要在不同的主题下，按照这个主题关心的维度对数据进行汇总，最后来算业务方需要的汇总指标。在具体操作中，对于 pv 类指标使用 Stream SQL 实现 1 分钟汇总指标作为最小

汇总单位指标，在此基础上进行时间维度上的指标累加；对于 uv 类指标直接使用 druid 数据库作为指标汇总容器，根据业务方对汇总指标的及时性和准确性的要求，实现相应的精确去重和非精确去重。

第三：汇总层建设过程中，还会涉及到衍生维度的加工。在顺风车券相关的汇总指标加工中我们使用 Hbase 的版本机制来构建一个衍生维度的拉链表，通过事件流和 Hbase 维表关联的方式得到实时数据当时的准确维度

命名规范：DWM 层的表命名使用英文小写字母，单词之间用下划线分开，总长度不能超过 40 个字符，并且应遵循下述规则：`realtime_dwm_{业务/pub}_{数据域缩写}_{数据主粒度缩写}_[{自定义表命名标签缩写}]_{统计时间周期范围缩写}`：

- {业务/pub}：参考业务命名
- {数据域缩写}：参考数据域划分部分
- {数据主粒度缩写}：指数据主要粒度或数据域的缩写，也是联合主键中的主要维度
- {自定义表命名标签缩写}：实体名称可以根据数据仓库转换整合后做一定的业务抽象的名称，该名称应该准确表述实体所代表的业务含义
- {统计时间周期范围缩写}：1d:天增量；td:天累计(全量)；1h:小时增量；th:小时累计(全量)；1min:分钟增量；tmin:分钟累计(全量)
- 样例：`realtime_dwm_trip_trd_pas_bus_accum_1min`

---

5. APP 应用层

该层主要的工作是把实时汇总数据写入应用系统的数据库中，包括用于大屏显示和实时 OLAP 的 Druid 数据库(该数据库除了写入应用数据，也可以写入明细数据完成汇总指标的计算)中，用于实时数据接口服务的 Hbase 数据库，用于实时数据产品的 mysql 或者 redis 数据库中。

命名规范：基于实时数仓的特殊性不做硬性要求。

2. 快手实时数仓场景化案例

1) 目标及难点

1. 目标

首先由于是做数仓，因此希望所有的实时指标都有离线指标去对应，要求实时指标和离线指标整体的数据差异在 1% 以内，这是最低标准。

其次是数据延迟，其 SLA 标准是活动期间所有核心报表场景的数据延迟不能超过 5 分钟，这 5 分钟包括作业挂掉之后和恢复时间，如果超过则意味着 SLA 不达标。

最后是稳定性，针对一些场景，比如作业重启后，我们的曲线是正常的，不会因为作业重启导致指标产出一些明显的异常。

2. 难点

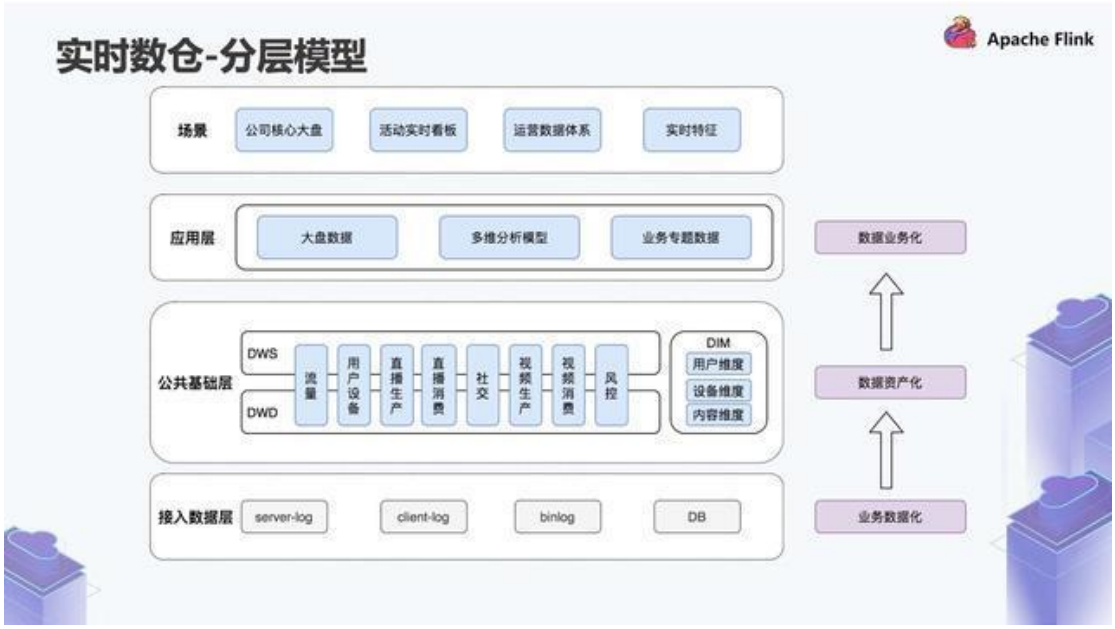
第一个难点是数据量大。每天整体的入口流量数据量级大概在万亿级。在活动如春晚的场景，QPS 峰值能达到亿 / 秒。

第二个难点是组件依赖比较复杂。可能这条链路里有的依赖于 Kafka，有的依赖 Flink，还有一些依赖 KV 存储、RPC 接口、OLAP 引擎等，我们需要思考在这条链路里如何分布，才能让这些组件都能正常工作。

第三个难点是链路复杂。目前我们有 200+ 核心业务作业，50+ 核心数据源，整体作业超过 1000。

2）实时数仓 - 分层模型

基于上面三个难点，来看一下数仓架构：



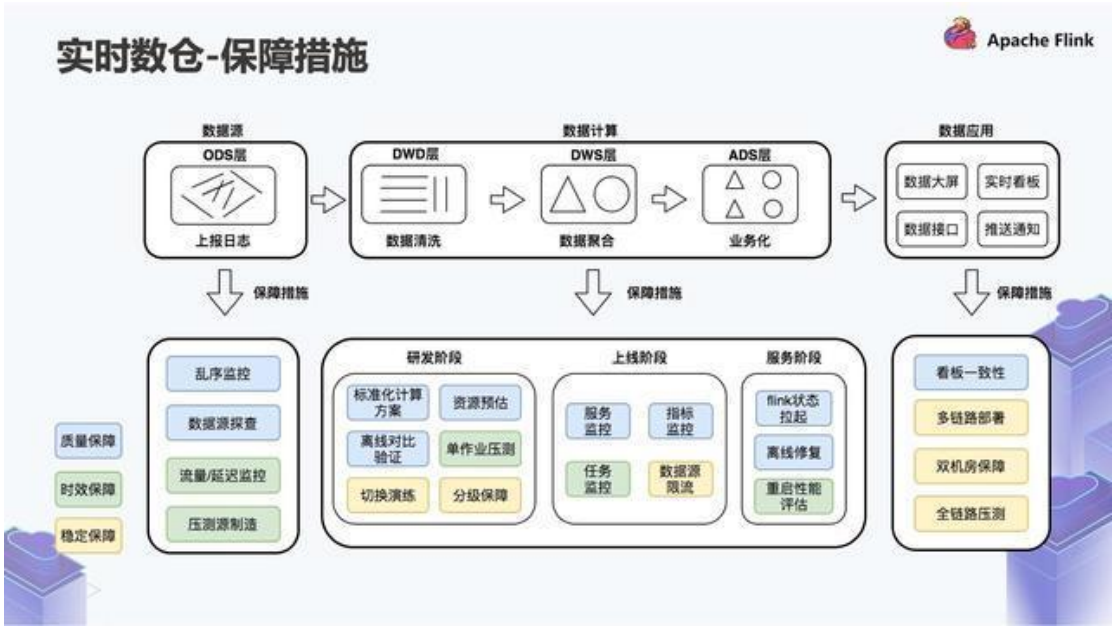
如上所示：

最下层有三个不同的数据源，分别是客户端日志、服务端日志以及 Binlog 日志；在公共基础层分为两个不同的层次，一个是 DWD 层，做明细数据，另一个是 DWS 层，做公共聚合数据，DIM 是我们常说的维度。我们有一个基于离线数仓的主题预分层，这个主题预分层可能包括流量、用户、设备、视频的生产消费、风控、社交等。DWD 层的核心工作是标准化的清洗；DWS 层是把维度的数据和 DWD 层进行关联，关联之后生成一些通用粒度的聚合层次。 再往上是应用层，包括一些大盘的数据，多维分析的模型以及业务专题数据； 最上面是场景。 整体过程可以分为三步：

第一步是做业务数据化，相当于把业务的数据接进来； 第二步是数据资产化，意思是对数据做很多的清洗，然后形成一些规则有序的数据； 第三步是数据业务化，可以理解数据在实时数据层面可以反哺业务，为业务数据价值建设提供一些赋能。

3）实时数仓 - 保障措施

基于上面的分层模型，来看一下整体的保障措施：



保障层面分为三个不同的部分，分别是质量保障，时效保障以及稳定保障。

我们先看蓝色部分的质量保障。针对质量保障，可以看到在数据源阶段，做了如数据源的乱序监控，这是我们基于自己的 SDK 的采集做的，以及数据源和离线的一致性校准。研发阶段的计算过程有三个阶段，分别是研发阶段、上线阶段和服务阶段。研发阶段可能会提供一个标准化的模型，基于这个模型会有一些 Benchmark，并且做离线的比对验证，保证质量是一致的；上线阶段更多的是服务监控和指标监控；在服务阶段，如果出现一些异常情况，先做 Flink 状态拉起，如果出现了一些不符合预期的场景，我们会做离线的整体数据修复。

第二个是时效性保障。针对数据源，我们把数据源的延迟情况也纳入监控。在研发阶段其实还有两个事情：首先是压测，常规的任务会拿最近 7 天或者最近 14 天的峰值流量去看它是否存在任务延迟的情况；通过压测之后，会有一些任务上线和重启性能评估，相当于按照 CP 恢复之后，重启的性能是什么样子。

最后一个是稳定保障，这在大型活动中会做得比较多，比如切换演练和分级保障。我们会基于之前的压测结果做限流，目的是保障作业在超过极限的情况下，仍然是稳定的，不会出现很多的不稳定或者 CP 失败的情况。之后我们会有两种不同的标准，一种是冷备双机房，另外一种是热备双机房。冷备双机房是：当一个单机房挂掉，我们会从另一个机房去拉起；热备双机房：相当于同样一份逻辑在两个机房各部署一次。 以上就是我们整体的保障措施。

3）快手场景问题及解决方案

1. PV/UV 标准化

1.1 场景

第一个问题是 PV/UV 标准化，这里有三个截图：

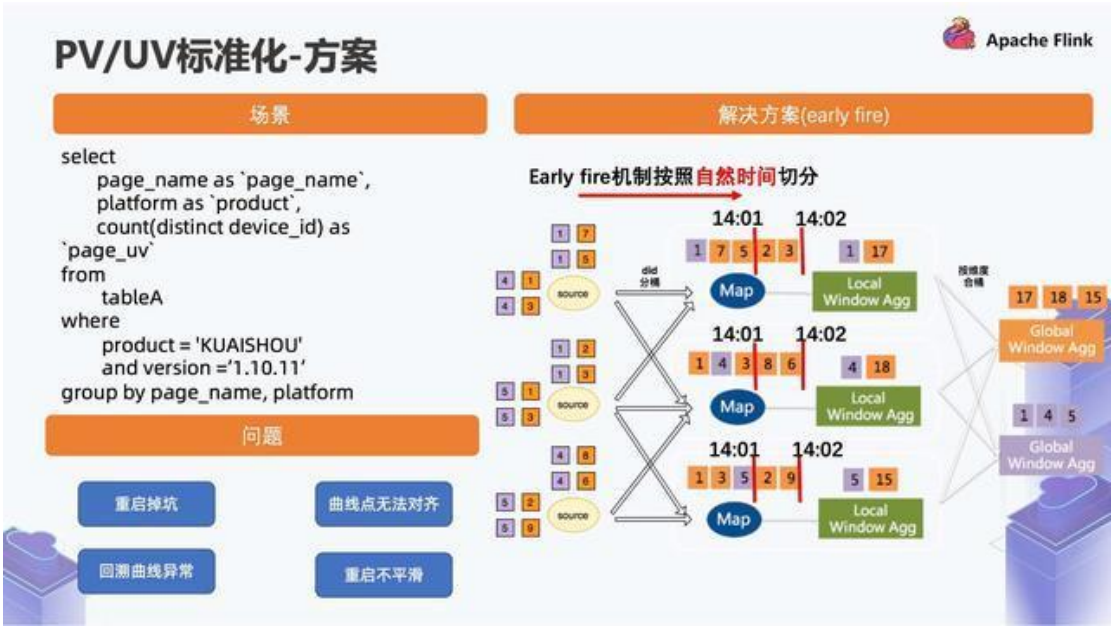
第一张图是春晚活动的预热场景，相当于是一种玩法，第二和第三张图是春晚当天的发红包活动和直播间截图。

在活动进行过程中，我们发现 60~70% 的需求是计算页面里的信息，如：

- 这个页面来了多少人，或者有多少人点击进入这个页面；
- 活动一共来了多少人；
- 页面里的某一个挂件，获得了多少点击、产生了多少曝光。

1.2 方案

抽象一下这个场景就是下面这种 SQL：



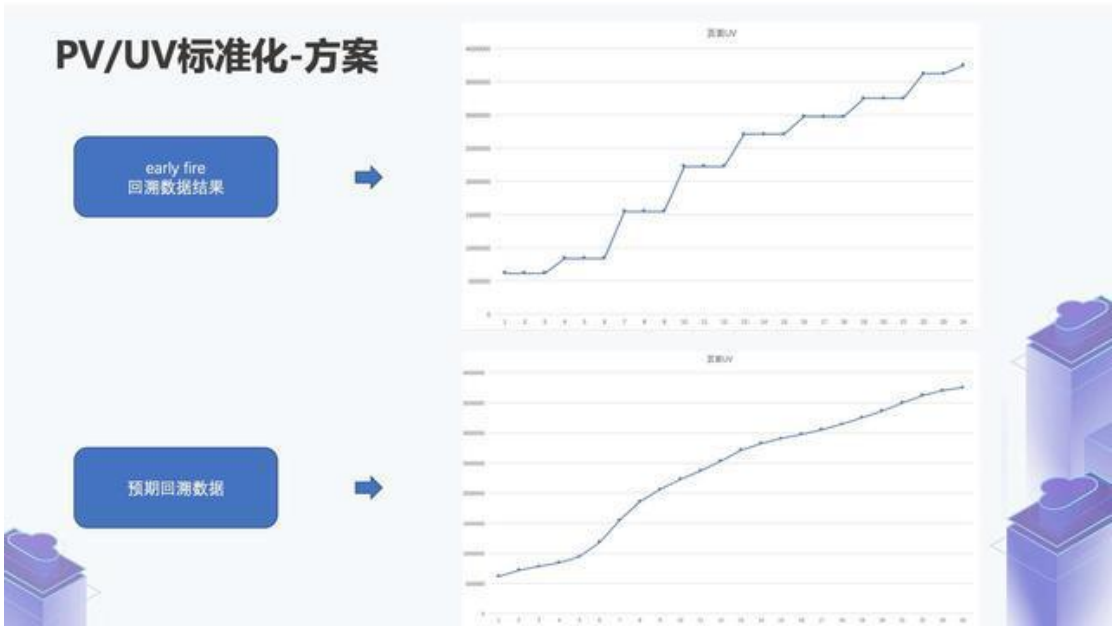
简单来说，就是从一张表做筛选条件，然后按照维度层面做聚合，接着产生一些 Count 或者 Sum 操作。

基于这种场景，我们最开始的解决方案如上图右边所示。

我们用到了 Flink SQL 的 Early Fire 机制，从 Source 数据源取数据，之后做了 DID 的分桶。比如最开始紫色的部分按这个做分桶，先做分桶的原因是防止某一个 DID 存在热点的问题。分桶之后会有一个叫做 Local Window Agg 的东西，相当于数据分完桶之后把相同类型的数据相加。Local Window Agg 之后再按照维度进行 Global Window Agg 的合桶，合桶的概念相当于按照维度计算出最终的结果。Early Fire 机制相当于在 Local Window Agg 开一个天级的窗口，然后每分钟去对外输出一次。

这个过程中我们遇到了一些问题，如上图左下角所示。

在代码正常运行的情况下是没有问题的，但如果整体数据存在延迟或者追溯历史数据的情况，比如一分钟 Early Fire 一次，因为追溯历史的时候数据量会比较大，所以可能导致 14:00 追溯历史，直接读到了 14:02 的数据，而 14:01 的那个点就被丢掉了，丢掉了以后会发生什么？



在这种场景下，图中上方的曲线为 Early Fire 回溯历史数据的结果。横坐标是分钟，纵坐标是截止到当前时刻的页面 UV，我们发现有些点是横着的，意味着没有数据结果，然后一个陡增，然后又横着的，接着又一个陡增，而这个曲线的预期结果其实是图中下方那种平滑的曲线。

为了解决这个问题，我们用到了 Cumulate Window 的解决方案，这个解决方案在 Flink 1.13 版本里也有涉及，其原理是一样的。

数据开一个大的天级窗口，大窗口下又开了一个小的分钟级窗口，数据按数据本身的 Row Time 落到分钟级窗口。

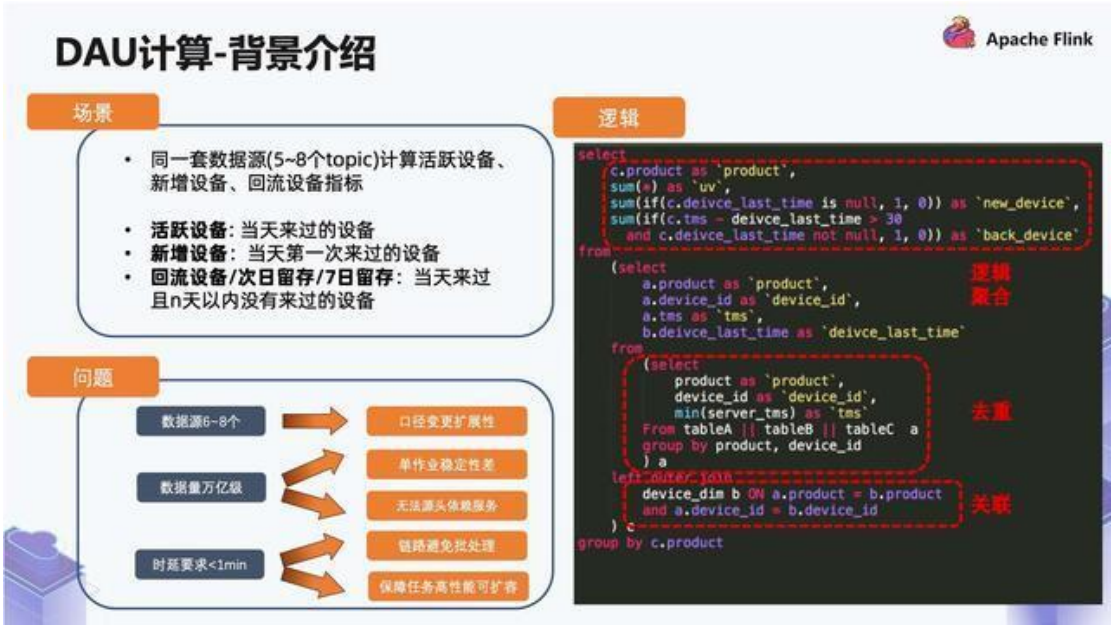
Watermark 推进过了窗口的 event_time，它会进行一次下发的触发，通过这种方式可以解决回溯的问题，数据本身落在真实的窗口， Watermark 推进，在窗

口结束后触发。 此外，这种方式在一定程度上能够解决乱序的问题。比如它的乱序数据本身是一个不丢弃的状态，会记录到最新的累计数据。 最后是语义一致性，它会基于事件时间，在乱序不严重的情况下，和离线计算出来的结果一致性是相当高的。 以上是 PV/UV 一个标准化的解决方案。

2. DAU 计算

2.1 背景介绍

下面介绍一下 DAU 计算：



我们对于整个大盘的活跃设备、新增设备和回流设备有比较多的监控。

活跃设备指的是当天来过的设备； 新增设备指的是当天来过且历史没有来过的设备； 回流设备指的是当天来过且 N 天内没有来过的设备。 但是我们计算过程之中可能需要 5~8 个这样不同的 Topic 去计算这几个指标。

我们看一下离线过程中，逻辑应该怎么算。

首先我们先算活跃设备，把这些合并到一起，然后做一个维度下的天级别去重，接着再去关联维度表，这个维度表包括设备的首末次时间，就是截止到昨天设备首次访问和末次访问的时间。

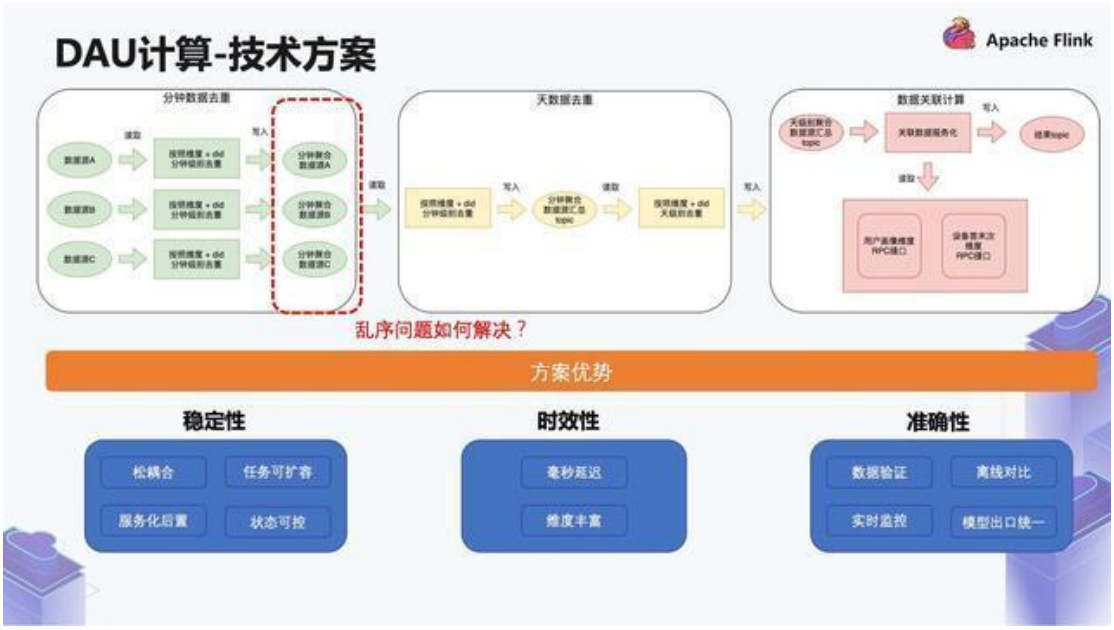
得到这个信息之后，我们就可以进行逻辑计算，然后我们会发现新增和回流的设备其实是活跃设备里打的一个子标签。新增设备就是做了一个逻辑处理，回流设备是做了 30 天的逻辑处理，基于这样的解决方案，我们能否简单地写一个 SQL 去解决这个问题？

其实我们最开始是这么做的，但遇到了一些问题：

第一个问题是：数据源是 6~8 个，而且我们大盘的口径经常会做微调，如果是单作业的话，每次微调的过程之中都要改，单作业的稳定性会非常差； 第二个问题是：数据量是万亿级，这会导致两个情况，首先是这个量级的单作业稳定性非常差，其次是实时关联维表的时候用的 KV 存储，任何一个这样的 RPC 服务接口，都不可能在万亿级数据量的场景下保证服务稳定性； 第三个问题是：我们对于时延要求比较高，要求时延小于一分钟。整个链路要避免批处理，如果出现了一些任务性能的单点问题，我们还要保证高性能和可扩容。

2.2 技术方案

针对以上问题，介绍一下我们是怎么做的：



如上图的例子，第一步是对 A B C 这三个数据源，先按照维度和 DID 做分钟级别去重，分别去重之后得到三个分钟级别去重的数据源，接着把它们 Union 到一起，然后再进行同样的逻辑操作。

这相当于我们数据源的入口从万亿变到了百亿的级别，分钟级别去重之后再进行一个天级别的去重，产生的数据源就可以从百亿变成了几十亿的级别。

在几十亿级别数据量的情况下，我们再去关联数据服务化，这就是一种比较可行的状态，相当于去关联用户画像的 RPC 接口，得到 RPC 接口之后，最终写入到了目标 Topic。这个目标 Topic 会导入到 OLAP 引擎，供给多个不同的服务，包括移动版服务，大屏服务，指标看板服务等。

这个方案有三个方面的优势，分别是稳定性、时效性和准确性。

首先是稳定性。松耦合可以简单理解为当数据源 A 的逻辑和数据源 B 的逻辑需要修改时，可以单独修改。第二是任务可扩容，因为我们把所有逻辑拆分得非常

细粒度，当一些地方出现了如流量问题，不会影响后面的部分，所以它扩容比较简单，除此之外还有服务化后置和状态可控。 其次是时效性，我们做到毫秒延迟，并且维度丰富，整体上有 20+ 的维度做多维聚合。 最后是准确性，我们支持数据验证、实时监控、模型出口统一等。 此时我们遇到了另外一个问题 - 乱序。对于上方三个不同的作业，每一个作业重启至少会有两分钟左右的延迟，延迟会导致下游的数据源 Union 到一起就会有乱序。

2.3 延迟计算方案

遇到上面这种有乱序的情况下，我们要怎么处理？

我们总共有三种处理方案：

第一种解决方案是用 “did + 维度 + 分钟” 进行去重，Value 设为 “是否来过”。比如同一个 did，04:01 来了一条，它会进行结果输出。同样的，04:02 和 04:04 也会进行结果输出。但如果 04:01 再来，它就会丢弃，但如果 04:00 来，依旧会进行结果输出。

这个解决方案存在一些问题，因为我们按分钟存，存 20 分钟的状态大小是存 10 分钟的两倍，到后面这个状态大小有点不太可控，因此我们又换了解决方案 2。

第二种解决方案，我们的做法会涉及到一个假设前提，就是假设不存在数据源乱序的情况。在这种情况下，key 存的是 “did + 维度”，Value 为 “时间戳”，它的更新方式如上图所示。04:01 来了一条数据，进行结果输出。04:02 来了一条数据，如果是同一个 did，那么它会更新时间戳，然后仍然做结果输出。04:04 也是同样的逻辑，然后将时间戳更新到 04:04，如果后面来了一条 04:01 的数据，它发现时间戳已经更新到 04:04，它会丢弃这条数据。这样的做法大幅度

减少了本身所需要的一些状态，但是对乱序是零容忍，不允许发生任何乱序的情况，由于我们不好解决这个问题，因此我们又想出了解决方案 3。

方案 3 是在方案 2 时间戳的基础之上，加了一个类似于环形缓冲区，在缓冲区之内允许乱序。

比如 04:01 来了一条数据，进行结果输出；04:02 来了一条数据，它会把时间戳更新到 04:02，并且会记录同一个设备在 04:01 也来过。如果 04:04 再来了一条数据，就按照相应的时间差做一个位移，最后通过这样的逻辑去保障它能够容忍一定的乱序。

综合来看这三个方案：

方案 1 在容忍 16 分钟乱序的情况下，单作业的状态大小在 480G 左右。这种情况虽然保证了准确性，但是作业的恢复和稳定性是完全不可控的状态，因此我们还是放弃了这个方案；

方案 2 是 30G 左右的状态大小，对于乱序 0 容忍，但是数据不准确，由于我们对准确性的要求非常高，因此也放弃了这个方案；

方案 3 的状态跟方案 1 相比，它的状态虽然变化了但是增加的不多，而且整体能达到跟方案 1 同样的效果。方案 3 容忍乱序的时间是 16 分钟，我们正常更新一个作业的话，10 分钟完全足够重启，因此最终选择了方案 3。

3. 运营场景

3.1 背景介绍

运营场景可分为四个部分：

第一个是数据大屏支持，包括单直播间的分析数据和大盘的分析数据，需要做到分钟级延迟，更新要求比较高；

第二个是直播看板支持，直播看板的数据会有特定维度的分析，特定人群支持，对维度丰富性要求比较高；

第三个是数据策略榜单，这个榜单主要是预测热门作品、爆款，要求的是小时级别的数据，更新要求比较低；

第四个是 C 端实时指标展示，查询量比较大，但是查询模式比较固定。

下面进行分析这 4 种不同的状态产生的一些不同的场景。

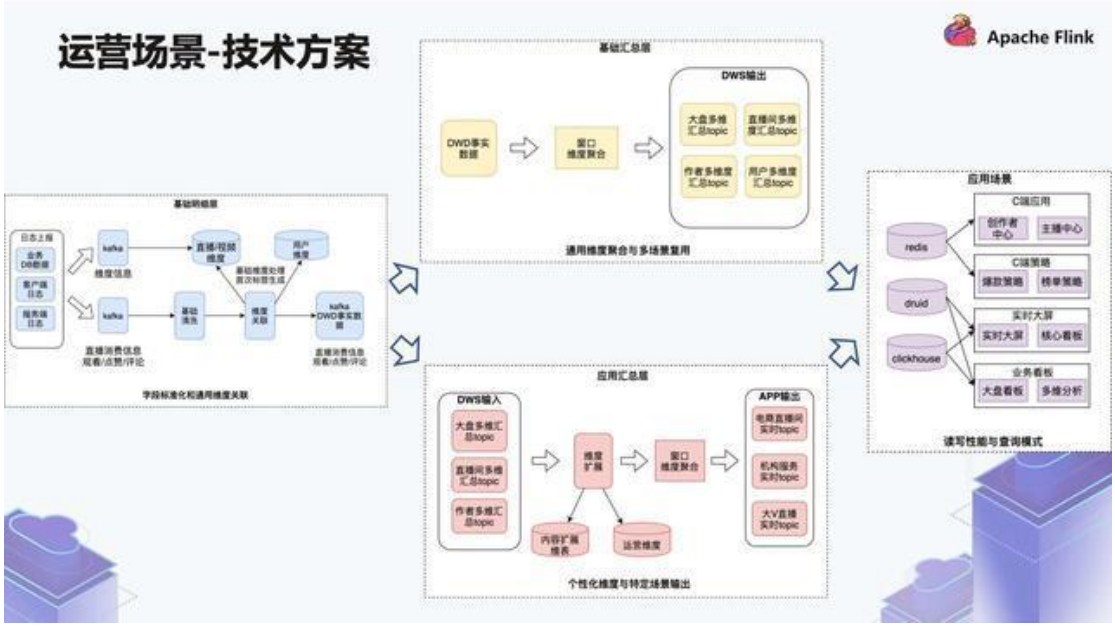
场景	时效性	计算逻辑	稳定性	查询模式	QPS
数据大屏	秒~分钟更新	复杂	分钟曲线一致 最终结果一致	多维度查询 通用业务场景	100+
实时数据看板	分钟级	复杂	分钟曲线一致 最终结果一致	多维度查询 特定业务场景	100+
数据策略	小时级	简单	最终一致	多维度查询 特定业务逻辑	10+
C端实时指标	15分钟级	简单	最终一致	单用户指标查询 基本是user_id查询	1000+

前 3 种基本没有什么差别，只是在查询模式上，有的是特定业务场景，有的是通用业务场景。

针对第 3 种和第 4 种，它对于更新的要求比较低，对于吞吐的要求比较高，过程之中的曲线也不要求有一致性。第 4 种查询模式更多的是单实体的一些查询，比如去查询内容，会有哪些指标，而且对 QPS 要求比较高。

3.2 技术方案

针对上方 4 种不同的场景，我们是如何去做的？



首先看一下基础明细层（图中左方），数据源有两条链路，其中一条链路是消费的流，比如直播的消费信息，还有观看 / 点赞 / 评论。经过一轮基础清洗，然后做维度管理。上游的这些维度信息来源于 Kafka，Kafka 写入了一些内容的维度，放到了 KV 存储里边，包括一些用户的维度。

这些维度关联了之后，最终写入 Kafka 的 DWD 事实层，这里为了做性能的提升，我们做了二级缓存的操作。

如图中上方，我们读取 DWD 层的数据然后做基础汇总，核心是窗口维度聚合生成 4 种不同粒度的数据，分别是大盘多维汇总 topic、直播间多维汇总 topic、作者多维汇总 topic、用户多维汇总 topic，这些都是通用维度的数据。

如图中下方，基于这些通用维度数据，我们再去加工个性化维度的数据，也就是 ADS 层。拿到了这些数据之后会有维度扩展，包括内容扩展和运营维度的拓展，然后再去做聚合，比如会有电商实时 topic，机构服务实时 topic 和大 V 直播实时 topic。

分成这样的两个链路会有一个好处：一个地方处理的是通用维度，另一个地方处理的是个性化的维度。通用维度保障的要求会比较高一些，个性化维度则会做很

多个性化的逻辑。如果这两个耦合在一起的话，会发现任务经常出问题，并且分不清楚哪个任务的职责是什么，构建不出这样的一个稳定层。

如图中右方，最终我们用到了三种不同的引擎。简单来说就是 Redis 查询用到了 C 端的场景，OLAP 查询用到了大屏、业务看板的场景。

3. 腾讯看点实时数仓案例

腾讯看点业务为什么要构建实时数仓，因为原始的上报数据量非常大，一天上报峰值就有上万亿条。而且上报格式混乱。缺乏内容维度信息、用户画像信息，下游没办法直接使用。而我们提供的实时数仓，是根据腾讯看点信息流的业务场景，进行了内容维度的关联，用户画像的关联，各种粒度的聚合，下游可以非常方便的使用实时数据。

1）方案选型

➢ **实时数仓：**

大规模数据处理架构	业界使用	灵活性	容错性	成熟度	迁移成本	批/流处理代码
Lambda架构	**高**	**高**	**高**	**高**	**低**	**2套**
Kappa架构	低	低	低	低	高	1套

➢ **实时计算引擎：**

实时计算引擎	准确性	容错机制	延时	吞吐量	易用性	业界使用
Flink	**Exactly-once**	**Checkpoint轻量级快照**	**低**	**高**	**高**	**高**
SparkStreaming	Exactly-once	RDD Checkpoint	中	高	中	中
Storm	At-least-once	Acker重试	低	低	低	低

➢ **实时存储引擎：**

存储引擎	维度索引	高并发	预聚合	高性能查询	特性
ClickHouse	**有**	**高**	**有**	**快**	**海量数据批量写入、实时多维聚合OLAP查询**
Ckv/HBase	无	高	无	慢	KV读写、多维聚合查询效率极低
Tdsql/Tidb	有	低	无	慢	适合OLTP
ES	有	快	无	慢	全文检索适合日志查询，运营分析，实时聚合查询慢
Druid	有	高	单粒度	中	计算全局TopN只能是近似值

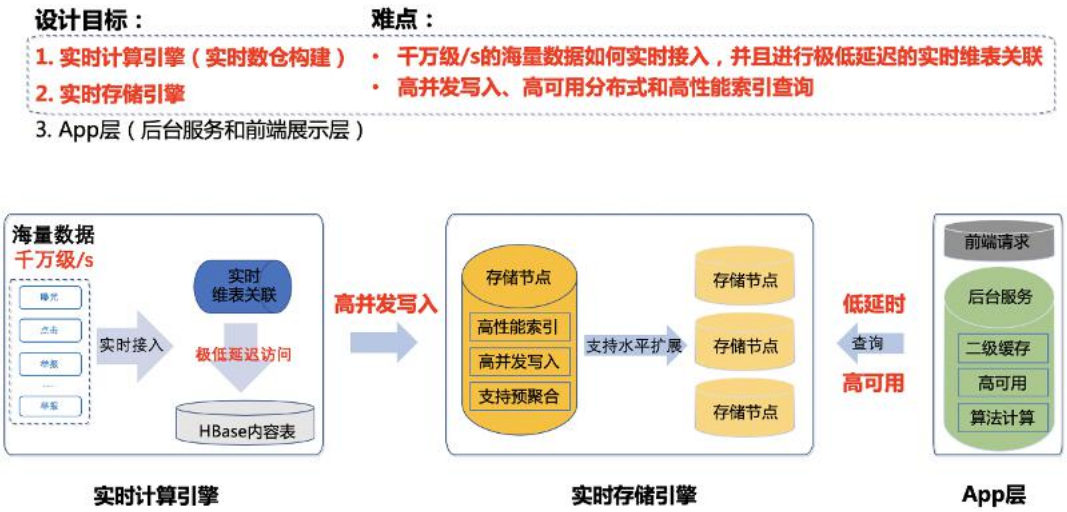
那就看下我们多维实时数据分析系统的方案选型，选型我们对比了行业内的领先方案，选择了最符合我们业务场景的方案。

- 第一块是实时数仓的选型，我们选择的是业界比较成熟的 Lambda 架构，他的优点是灵活性高、容错性高、成熟度高和迁移成本低；缺点是实时、离线数据用两套代码，可能会存在一个口径修改了，另一个没改的问题，我们每天都有做数据对账的工作，如果有异常会进行告警。
- 第二块是实时计算引擎选型，因为 Flink 设计之初就是为了流处理，SparkStreaming 严格来说还是微批处理，Strom 用的已经不多了。再看

Flink 具有 Exactly-once 的准确性、轻量级 Checkpoint 容错机制、低延时高吞吐和易用性高的特点，我们选择了 Flink 作为实时计算引擎。

- 第三块是实时存储引擎，我们的要求就是需要有维度索引、支持高并发、预聚合、高性能实时多维 OLAP 查询。可以看到，Hbase、Tdsql 和 ES 都不能满足要求，Druid 有一个缺陷，它是按照时序划分 Segment，无法将同一个内容，存放在同一个 Segment 上，计算全局 TopN 只能是近似值，所以我们选择了最近两年大火的 MPP 数据库引擎 ClickHouse。

2）设计目标与设计难点



我们多维实时数据分析系统分为三大模块

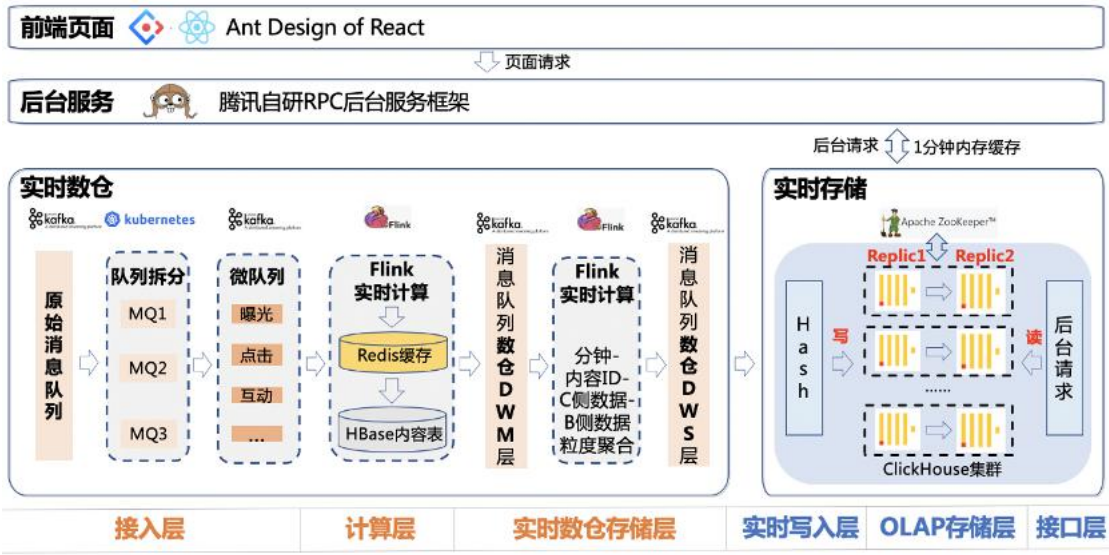
1. 实时计算引擎
2. 实时存储引擎
3. App 层

难点主要在前两个模块：实时计算引擎和实时存储引擎。

1. 千万级/s 的海量数据如何实时接入，并且进行极低延迟维表关联。
2. 实时存储引擎如何支持高并发写入、高可用分布式和高性能索引查询，是比较难的。

这几个模块的具体实现，看一下我们系统的架构设计。

3）架构设计



前端采用的是开源组件 Ant Design，利用了 Nginx 服务器，部署静态页面，并反向代理了浏览器的请求到后台服务器上。

后台服务是基于腾讯自研的 RPC 后台服务框架写的，并且会进行一些二级缓存。

实时数仓部分，分为了接入层、实时计算层和实时数仓存储层。

- 接入层主要是从千万级/s 的原始消息队列中，拆分出不同行为数据的微队列，拿看点的视频来说，拆分过后，数据就只有百万级/s 了；
- 实时计算层主要负责，多行行为流水数据进行行转列，实时关联用户画像数据和内容维度数据；
- 实时数仓存储层主要是设计出符合看点业务的，下游好用的实时消息队列。我们暂时提供了两个消息队列，作为实时数仓的两层。一层 DWM 层是内容 ID-用户 ID 粒度聚合的，就是一条数据包含内容 ID-用户 ID 还有 B 侧内容数据、C 侧用户数据和用户画像数据；另一层是 DWS 层，是内容 ID 粒度聚合的，一条数据包含内容 ID，B 侧数据和 C 侧数据。可以看到内容 ID-用户 ID 粒度的消息队列流量进一步减小到十万级/s，内容 ID 粒度的更是万级/s，并且格式更加清晰，维度信息更加丰富。

实时存储部分分为实时写入层、OLAP 存储层和后台接口层。

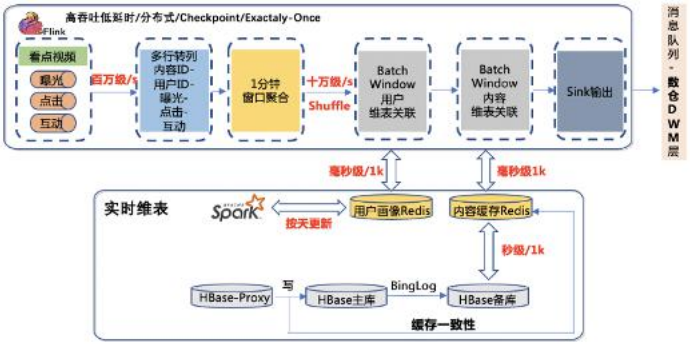
- 实时写入层主要是负责 Hash 路由将数据写入；
- OLAP 存储层利用 MPP 存储引擎，设计符合业务的索引和物化视图，高效存储海量数据；
- 后台接口层提供高效的多维实时查询接口。

4）实时计算

这个系统最复杂的两块，实时计算和实时存储。

先介绍实时计算部分：分为实时关联和实时数仓。

1. 实时高性能维表关联



难点：1分钟的实时数据流(百万级/s)直接关联HBase需要小时级耗时	
解决方案：	耗时：
✓ Flink行转列/聚合	小时级->十几分钟
✓ Batch+Redis缓存	十几分钟->秒级
✓ 上游过滤一些不存在的赛道内容ID防止缓存穿透	秒级->秒级
✓ 消峰填谷，防止雪崩	秒级->数十秒

指标	优化前	优化后	节省
时间	小时级	数十秒	90%+
数据量	百亿级	十亿级	

实时维表关联这一块难度在于 百万级/s 的实时数据流，如果直接去关联 HBase，1 分钟的数据，关联完 HBase 耗时是小时级的，会导致数据延迟严重。

我们提出了几个解决方案：

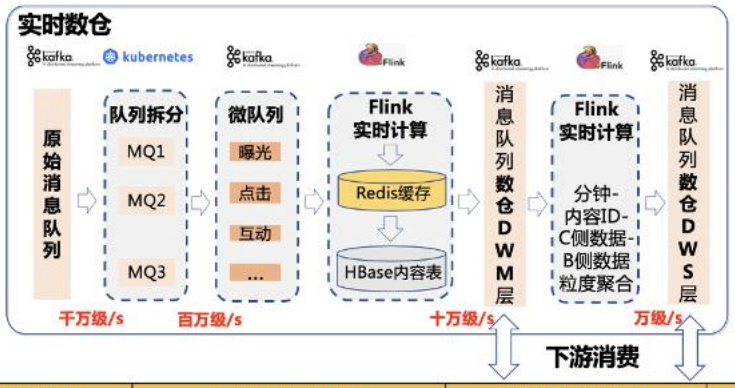
- 第一个是，在 Flink 实时计算环节，先按照 1 分钟进行了窗口聚合，将窗口内多行行为数据转一行多列的数据格式，经过这一步操作，原本小时级的关联耗时下降到了十几分钟，但是还是不够的。
- 第二个是，在访问 HBase 内容之前设置一层 Redis 缓存，因为 1000 条数据访问 HBase 是秒级的，而访问 Redis 是毫秒级的，访问 Redis 的速度基本是访问 HBase 的 1000 倍。为了防止过期的数据浪费缓存，缓

存过期时间设置成 24 小时，同时通过监听写 HBase Proxy 来保证缓存的一致性。这样将访问时间从十几分钟变成了秒级。

- 第三个是，上报过程中会上报不少非常规内容 ID，这些内容 ID 在内容 HBase 中是不存储的，会造成缓存穿透的问题。所以在实时计算的时候，我们直接过滤掉这些内容 ID，防止缓存穿透，又减少一些时间。
- 第四个是，因为设置了定时缓存，会引入一个缓存雪崩的问题。为了防止雪崩，我们在实时计算中，进行了削峰填谷的操作，错开设置缓存的时间。

可以看到，优化前后，数据量从百亿级减少到了十亿级，耗时从小时级减少到了数十秒，减少 99%。

2. 下游提供服务



50个消费者	建设实时数仓前	建设实时数仓后	节省
Flink(核心)	X1*50	X1	1个月节省上百万
Redis(内存)	X2*50	X2	减少98%
接入人力成本（天）	X3*50	X3	
接入实时数据门槛	高(涉及模块多、有门槛)	低（Flink SQL）	

实时数仓的难度在于：它处于比较新的领域，并且各个公司各个业务差距比较大，怎么能设计出方便，好用，符合看点业务场景的实时数仓是有难度的。

先看一下实时数仓做了什么，实时数仓对外就是几个消息队列，不同的消息队列里面存放的就是不同聚合粒度的实时数据，包括内容 ID、用户 ID、C 侧行为数据、B 侧内容维度数据和用户画像数据等。

我们是怎么搭建实时数仓的，就是上面介绍的实时计算引擎的输出，放到消息队列中保存，可以提供给下游多用户复用。

我们可以看下，在我们建设实时数据仓库前后，开发一个实时应用的区别。没有数仓的时候，我们需要消费千万级/s 的原始队列，进行复杂的数据清洗，然后再进行用户画像关联、内容维度关联，才能拿到符合要求格式的实时数据，开发和扩展的成本都会比较高，如果想开发一个新的应用，又要走一遍这个流程。有

了数仓之后，如果想开发内容 ID 粒度的实时应用，就直接申请 TPS 万级/s 的 DWS 层的消息队列。开发成本变低很多，资源消耗小很多，可扩展性也强很多。看个实际例子，开发我们系统的实时数据大屏，原本需要进行如上所有操作，才能拿到数据。现在只需要消费 DWS 层消息队列，写一条 Flink SQL 即可，仅消耗 2 个 CPU 核心，1G 内存。

可以看到，以 50 个消费者为例，建立实时数仓前后，下游开发一个实时应用，可以减少 98%的资源消耗。包括计算资源，存储资源，人力成本和开发人员学习接入成本等等。并且消费者越多，节省越多。就拿 Redis 存储这一部分来说，一个月就能省下上百万人民币。

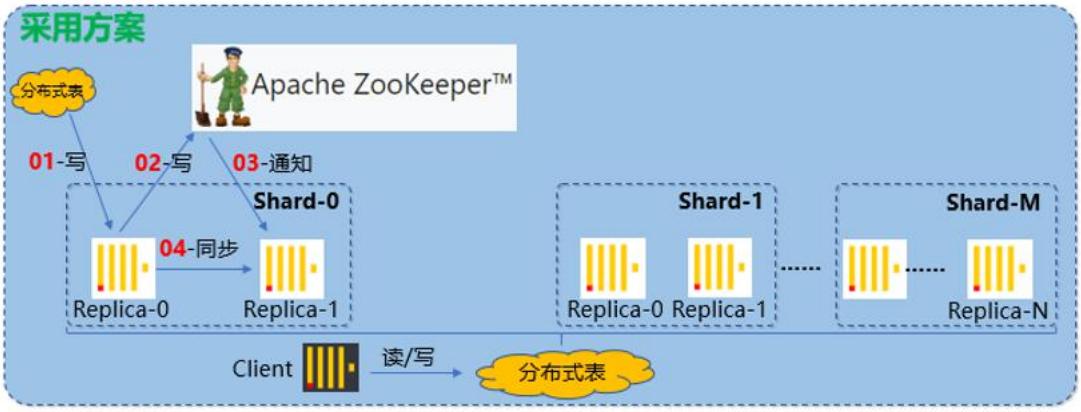
5）实时存储

介绍完实时计算，再来介绍实时存储。

这块分为三个部分来介绍

- 第一是 分布式-高可用
- 第二是 海量数据-写入
- 第三是 高性能-查询

1. 分布式-高可用

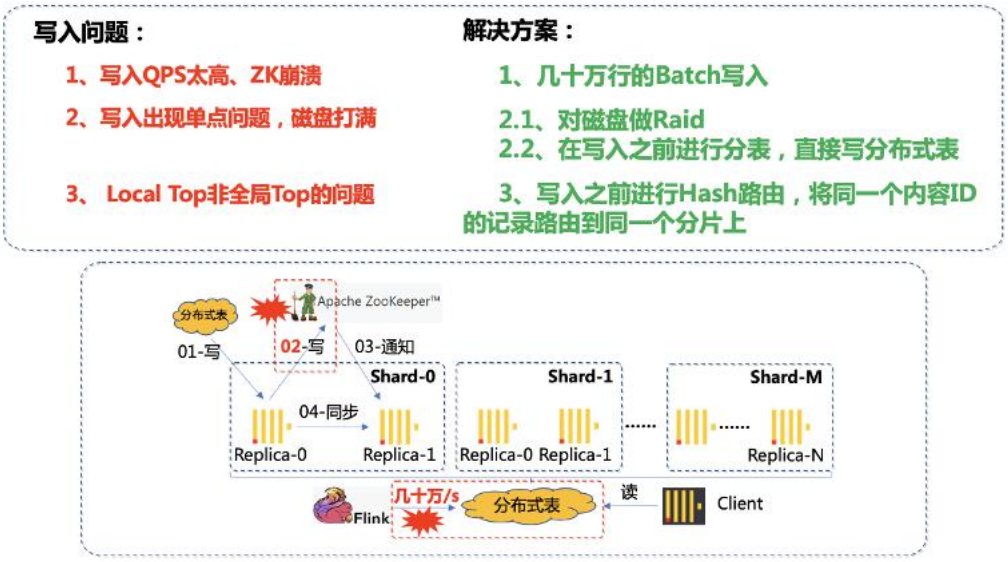


我们这里听取的是 Clickhouse 官方的建议，借助 ZK 实现高可用的方案。数据写入一个分片，仅写入一个副本，然后再写 ZK，通过 ZK 告诉同一个分片的其他副本，其他副本再过来拉取数据，保证数据一致性。

这里没有选用消息队列进行数据同步，是因为 ZK 更加轻量级。而且写的时候，任意写一个副本，其它副本都能够通过 ZK 获得一致的数据。而且就算其它节点

第一次来获取数据失败了，后面只要发现它跟 ZK 上记录的数据不一致，就会再次尝试获取数据，保证一致性。

2. 海量数据-写入



数据写入遇到的第一个问题是，海量数据直接写入 Clickhouse 的话，会导致 ZK 的 QPS 太高，解决方案是改用 Batch 方式写入。Batch 设置多大呢，Batch 太小的话缓解不了 ZK 的压力，Batch 也不能太大，不然上游内存压力太大，通过实验，最终我们选用了大小几十万的 Batch。

第二个问题是，随着数据量的增长，单 QQ 看点的视频内容每天可能写入百亿级的数据，默认方案是写一张分布式表，这就会造成单台机器出现磁盘的瓶颈，尤其是 Clickhouse 底层运用的是 Mergetree，原理类似于 HBase、RocketsDB 的底层 LSM-Tree。在合并的过程中会存在写放大的问题，加重磁盘压力。峰值每分钟几千万条数据，写完耗时几十秒，如果正在做 Merge，就会阻塞写入请求，查询也会非常慢。我们做的两个优化方案：一是对磁盘做 Raid，提升磁盘的 IO；二是在写入之前进行分表，直接分开写入到不同的分片上，磁盘压力直接变为 1/N。

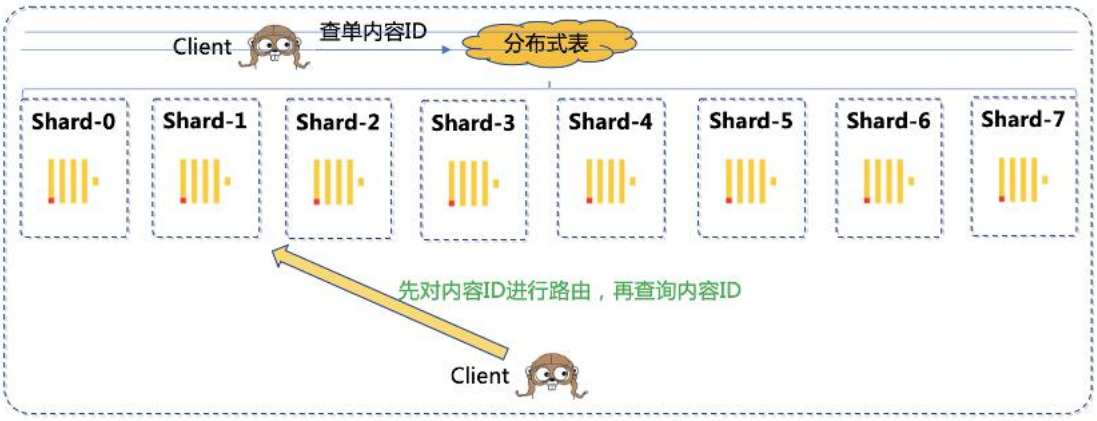
第三个问题是，虽然我们写入按照分片进行了划分，但是这里引入了一个分布式系统常见的问题，就是局部的 Top 并非全局 Top 的问题。比如同一个内容 ID 的数据落在了不同的分片上，计算全局 Top100 阅读的内容 ID，有一个内容 ID 在分片 1 上是 Top100，但是在其它分片上不是 Top100，导致汇总的时候，会丢失一部分数据，影响最终结果。我们做的优化是在写入之前加上一层路由，将同一个内容 ID 的记录，全部路由到同一个分片上，解决了该问题。

介绍完写入，下一步介绍 Clickhouse 的高性能存储和查询。

3. 高性能-存储-查询

Clickhouse 高性能查询的一个关键点是稀疏索引。稀疏索引这个设计就很有讲究，设计得好可以加速查询，设计不好反而会影响查询效率。我根据我们的业务场景，因为我们的查询大部分都是时间和内容 ID 相关的，比如说，某个内容，过去 N 分钟在各个人群表现如何？我按照日期，分钟粒度时间和内容 ID 建立了稀疏索引。针对某个内容的查询，建立稀疏索引之后，可以减少 99%的文件扫描。

还有一个问题就是，我们现在数据量太大，维度太多。拿 QQ 看点的视频内容来说，一天流水有上百亿条，有些维度有几百个类别。如果一次性把所有维度进行预聚合，数据量会指数膨胀，查询反而变慢，并且会占用大量内存空间。我们的优化，针对不同的维度，建立对应的预聚合物化视图，用空间换时间，这样可以缩短查询的时间。



分布式表查询还会有一个问题，查询单个内容 ID 的信息，分布式表会将查询下发到所有的分片上，然后再返回查询结果进行汇总。实际上，因为做过路由，一个内容 ID 只存在于一个分片上，剩下的分片都在空跑。针对这类查询，我们的优化是后台按照同样的规则先进行路由，直接查询目标分片，这样减少了 N-1/N 的负载，可以大量缩短查询时间。而且由于我们是提供的 OLAP 查询，数据满足最终一致性即可，通过主从副本读写分离，可以进一步提升性能。

我们在后台还做了一个 1 分钟的数据缓存，针对相同条件查询，后台就直接返回了。

4. 扩容

这里再介绍一下我们的扩容的方案，调研了业内的一些常见方案。

比如 HBase，原始数据都存放在 HDFS 上，扩容只是 Region Server 扩容，不涉及原始数据的迁移。但是 Clickhouse 的每个分片数据都是在本地，是一个比较底层存储引擎，不能像 HBase 那样方便扩容。

Redis 是哈希槽这种类似一致性哈希的方式，是比较经典分布式缓存的方案。

Redis slot 在 Rehash 的过程中虽然存在短暂的 ask 读不可用，但是总体来说迁移是比较方便的，从原 h[0]迁移到 h[1]，最后再删除 h[0]。但是 Clickhouse 大部分都是 OLAP 批量查询，不是点查，而且由于列式存储，不支持删除的特性，一致性哈希的方案不是很适合。

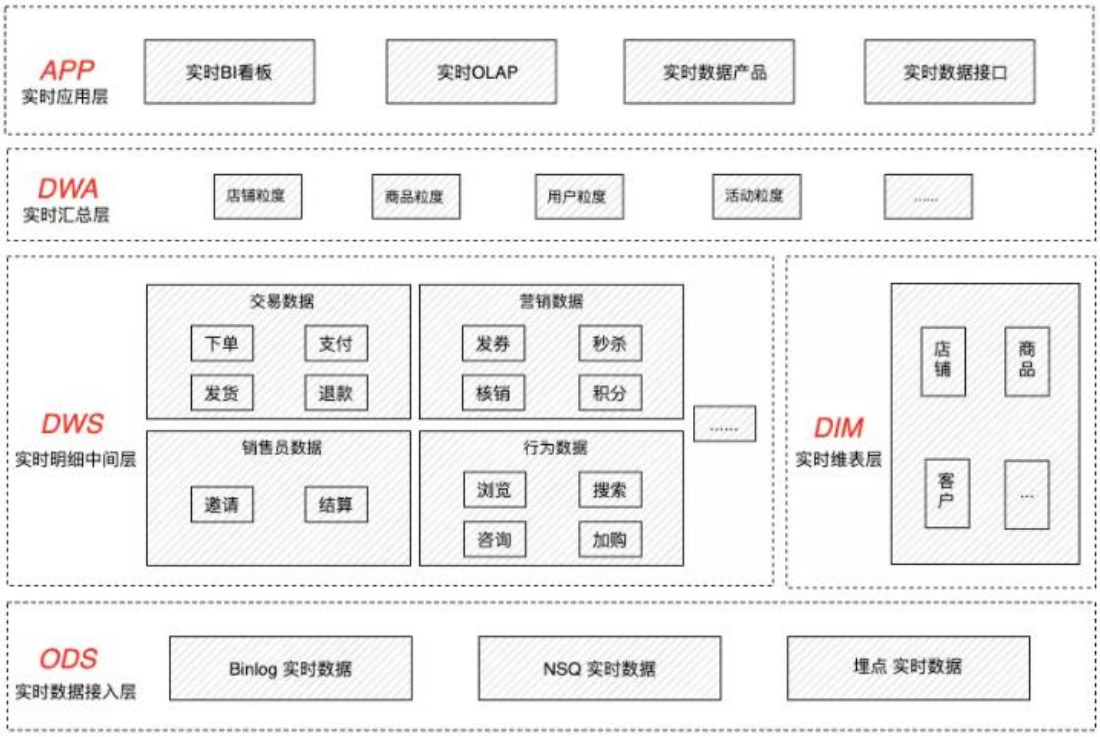
目前扩容的方案是，另外消费一份数据，写入新 Clickhouse 集群，两个集群一起跑一段时间，因为实时数据就保存 3 天，等 3 天之后，后台服务直接访问新集群。

4. 有赞实时数仓案例

1）分层设计

传统离线数仓的分层设计大家都很熟悉，为了规范的组织和管理数据，层级划分会比较多，在一些复杂逻辑处理场景还会引入临时层落地中间结果以方便下游加工处理。实时数仓考虑到时效性问题，分层设计需要尽量精简，降低中间流程出错的可能性，不过总体而言，实时数仓还是会参考离线数仓的分层思想来设计。

实时数仓分层架构如下图所示 ：



- ODS（实时数据接入层）

ODS 层，即实时数据接入层，通过数据采集工具收集各个业务系统的实时数据，对非结构化的数据进行结构化处理，保存原始数据，几乎不过滤数据；该层数据的主要来源有三个部分：第一部分是业务方创建的 NSQ 消息，第二部分是业务数据库的 Binlog 日志，第三部分是埋点日志和应用程序日志，以上三部分的实时数据最终统一写入 Kafka 存储介质中。

ODS 层表命名规范：部门名称.应用名称.数仓层级主题域前缀数据库名/消息名

例如：接入业务库的 Binlog

实时数仓表命名：`deptname.appname.ods_subjectname_tablename`

例如：接入业务方的 NSQ 消息

实时数仓表命名：`deptname.appname.ods_subjectname_msgname`

- DWS（实时明细中间层）

DWS 层，即实时明细中间层，该层以业务过程作为建模驱动，基于每个具体的业务过程事件来构建最细粒度的明细层事实表；比如交易过程，有下单事件、支付事件、发货事件等，我们会基于这些独立的事件来进行明细层的构建。在这层，事实明细数据同样是按照离线数仓的主题域来进行划分，也会采用维度建模的方式组织数据，对于一些重要的维度字段，会做适当冗余。基于有赞实时需求的场景，重点建设交易、营销、客户、店铺、商品等主题域的数据。该层的数据来源

于 ODS 层，通过 FlinkSQL 进行 ETL 处理，主要工作有规范命名、数据清洗、维度补全、多流关联，最终统一写入 Kafka 存储介质中。

DWS 层表命名规范：部门名称.应用名称.数仓层级_主题域前缀_数仓表命名

例如：实时事件 A 的中间层

实时数仓表命名：deptname.appname.dws_subjectname_tablename_eventnameA

例如：实时事件 B 的中间层

实时数仓表命名：deptname.appname.dws_subjectname_tablename_eventnameB

- DIM（实时维表层）

DIM 层，即实时维表层，用来存放维度数据，主要用于实时明细中间层宽化处理时补全维度使用，目前该层的数据主要存储于 HBase 中，后续会基于 QPS 和数据量大小提供更多合适类型的存储介质。

DIM 层表命名规范：应用名称_数仓层级_主题域前缀_数仓表命名

例如：HBase 存储，实时维度表

实时数仓表命名：appname_dim_tablename

- DWA（实时汇总层）

DWA 层，即实时汇总层，该层通过 DWS 层数据进行多维汇总，提供给下游业务方使用，在实际应用过程中，不同业务方使用维度汇总的方式不太一样，根据不同的需求采用不同的技术方案去实现。第一种方式，采用 FlinkSQL 进行实时汇总，将结果指标存入 HBase、MySQL 等数据库，该种方式是我们早期采用的方案，优点是实现业务逻辑比较灵活，缺点是聚合粒度固化，不易扩展；第二种方式，采用实时 OLAP 工具进行汇总，该种方式是我们目前常用的方案，优点是聚合粒度易扩展，缺点是业务逻辑需要在中间层预处理。

DWA 层表命名规范：应用名称_数仓层级_主题域前缀_聚合粒度_数据范围

例如：HBase 存储，某域当日某粒度实时汇总表

实时数仓表命名：appname_dwa_subjectname_aggname_daily

- APP（实时应用层）

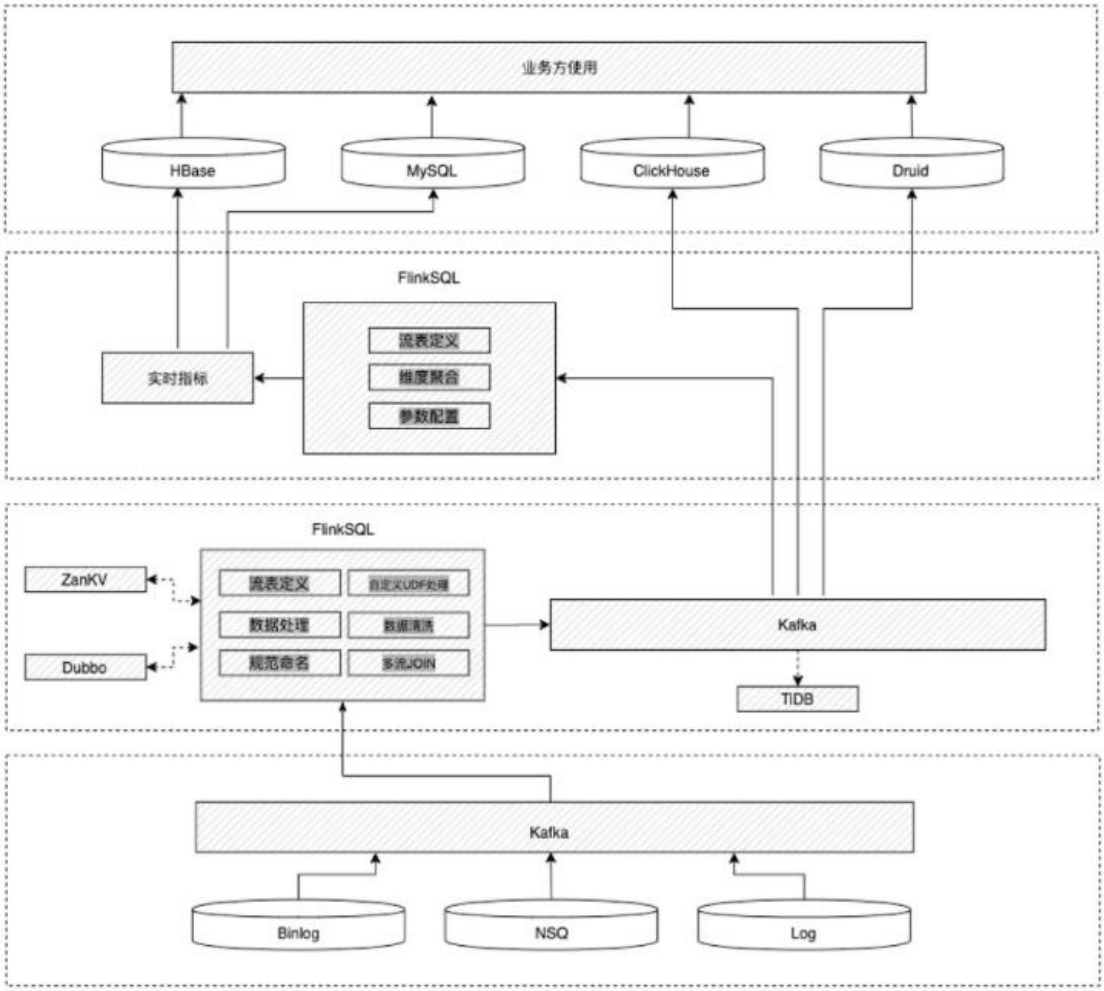
APP 层，即实时应用层，该层数据已经写入应用系统的存储中，例如写入 Druid 作为 BI 看板的实时数据集；写入 HBase、MySQL 用于提供统一数据服务接口；写入 ClickHouse 用于提供实时 OLAP 服务。因为该层非常贴近业务，在命名规范上实时数仓不做统一要求。

2）实时 ETL

实时数仓 ETL 处理过程所涉及的组件比较多，接下来盘点构建实时数仓所需要的组件以及每个组件的应用场景。如下图所示：

组件名称	应用场景
Kafka	实时数仓采用的消息数据存储组件，默认保留近3天的数据，数据接入，明细中间层都会存入该组件中
FlinkSQL	实时数仓采用的ETL处理组件，地位等同于离线数仓的HiveSQL、SparkSQL
ZanKV	实时数仓采用的处理消息时确保消息幂等的组件
Dubbo	在维度补全场景给实时明细中间层补全维度使用
TiDB	实时数仓采用的持久化Kafka消息的组件，用来进行数据验证
HBase、MySQL	作为实时数仓实时维度表的存储、实时汇总数据的存储
Druid、ClickHouse	实时数仓提供OLAP服务所采用的存储组件

具体实时 ETL 处理流程如下图所示：



1. 维度补全

创建调用 Duboo 接口的 UDF 函数在实时流里补全维度是最便捷的使用方式，但如果请求量过大，对 Duboo 接口压力会过大。在实际应用场景补全维度首选还是关联维度表，但关联也存在一定概率的丢失问题，为了弥补这种丢失，可以采用 Duboo 接口调用兜底的方式来补全。伪代码如下：

```
create function call_dubbo as 'XXXXXXX';
create function get_json_object as 'XXXXXXX';

case
    when cast( b.column as bigint) is not null
        then cast( b.column as bigint)
            else cast(coalesce(cast(get_json_object(call_dubbo('clusterUrl'
                                                               ,'serviceName'
                                                               ,'methodName'
                                                               ,cast(concat('[',cast
(a.column as varchar),']') as varchar)
                                                               ,'key'
                                                               )
                                                    ,'rootId')
                                           as bigint)
                                  ,a.column)
                           as bigint)  end
```

2. 幂等处理

实时任务在运行过程中难免会遇到执行异常的情况，当任务异常重启的时候会导致部分消息重新发送和消费，从而引发下游实时统计数据不准确，为了有效避免这种情况，可以选择对实时消息流做幂等处理，当消费完一条消息，将这条消息的 Key 存入 KV，如果任务异常重启导致消息重新发送的时候，先从 KV 判断该消息是否已被消费，如果已消费就不再往下发送。伪代码如下：

```
create function idempotenc as 'XXXXXXX';

insert into table
select
    order_no
from
    (
        select
            a.orderNo                                          as  order_no
          , idempotenc('XXXXXXX', coalesce( order_no, '') )  as  rid
        from
            table1
    ) t
```

```
where
    t.rid = 0;
```

3. 数据验证

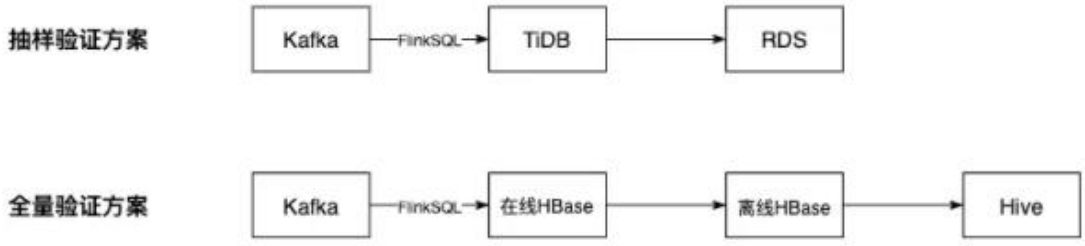
由于实时数仓的数据是无边界的流，相比于离线数仓固定不变的数据更难验收。基于不同的场景，我们提供了 2 种验证方式，分别是：抽样验证与全量验证。如图 3.3 所示

- 抽样验证方案

该方案主要应用在数据准确性验证上，实时汇总结果是基于存储在 Kafka 的实时明细中间层计算而来，但 Kafka 本身不支持按照特定条件检索，不支持写查询语句，再加上消息的无边界性，统计结果是在不断变化的，很难寻找参照物进行比对。鉴于此，我们采用了持久化消息的方法，将消息落盘到 TiDB 存储，基于 TiDB 的能力对落盘的消息进行检索、查询、汇总。编写固定时间边界的测试用例与相同时间边界的业务库数据或者离线数仓数据进行比对。通过以上方式，抽样核心店铺的数据进行指标准确性验证，确保测试用例全部通过。

- 全量验证方案

该方案主要应用在数据完整性和一致性验证上，在实时维度表验证的场景使用最多。大体思路：将存储实时维度表的在线 HBase 集群中的数据同步到离线 HBase 集群中，再将离线 HBase 集群中的数据导入到 Hive 中，在限定实时维度表的时间边界后，通过数据平台提供的数据校验功能，比对实时维度表与离线维度表是否存在差异，最终确保两张表的数据完全一致。

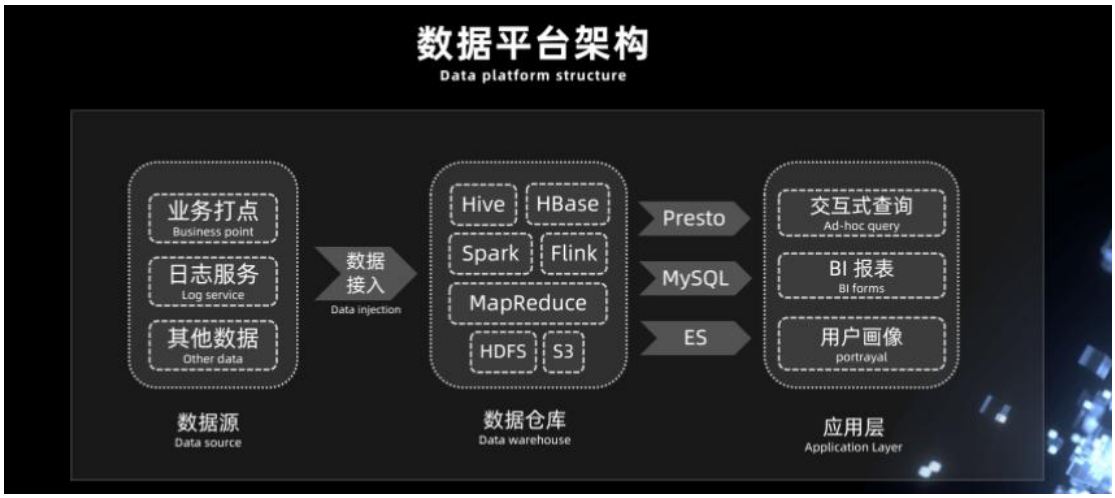


4. 数据恢复

实时任务一旦上线就要求持续不断的提供准确、稳定的服务。区别于离线任务按天调度，如果离线任务出现 Bug，会有充足的时间去修复。如果实时任务出现 Bug，必须按照提前制定好的流程，严格按照步骤执行，否则极易出现问题。造成 Bug 的情况有非常多，比如代码 Bug、异常数据 Bug、实时集群 Bug，如下图展示了修复实时任务 Bug 并恢复数据的流程。

5. 腾讯全场景实时数仓建设案例

在数仓体系中会有各种各样的大数据组件，譬如 Hive/HBase/HDFS/S3，计算引擎如 MapReduce、Spark、Flink，根据不同的需求，用户会构建大数据存储和处理平台，数据在平台经过处理和分析，结果数据会保存到 MySQL、Elasticsearch 等支持快速查询的关系型、非关系型数据库中，接下来应用层就可以基于这些数据进行 BI 报表开发、用户画像，或基于 Presto 这种 OLAP 工具进行交互式查询等。



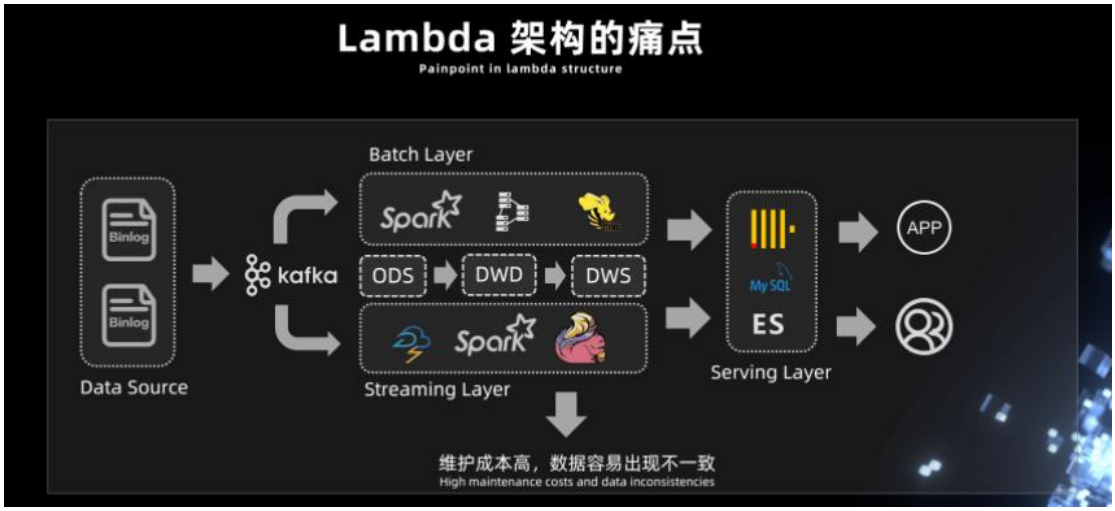
1）Lambda 架构的痛点

在整个过程中我们常常会用一些离线的调度系统，定期的（T+1 或者每隔几小时）去执行一些 Spark 分析任务，做一些数据的输入、输出或是 ETL 工作。离线数据处理的整个过程中必然存在数据延迟的现象，不管是数据接入还是中间的分析，数据的延迟都是比较大的，可能是小时级也有可能是天级别的。另外一些场景中

我们也常常会为了一些实时性的需求去构建一个实时处理过程，比如借助 Flink+Kafka 去构建实时的流处理系统。

整体上，数仓架构中有非常多的组件，大大增加了整个架构的复杂性和运维的成本。

如下图，这是很多公司之前或者现在正在采用的 Lambda 架构，Lambda 架构将数仓分为离线层和实时层，相应的就有批处理和流处理两个相互独立的数据处理流程，同一份数据会被处理两次以上，同一套业务逻辑代码需要适配性的开发两次。Lambda 架构大家应该已经非常熟悉了，下面我就着重介绍一下我们采用 Lambda 架构在数仓建设过程中遇到的一些痛点问题。



例如在实时计算一些用户相关指标的实时场景下，我们想看到当前 pv、uv 时，我们会将这些数据放到实时层去做一些计算，这些指标的值就会实时呈现出来，但同时想了解用户的一个增长趋势，需要把过去一天的数据计算出来。这样就需要通过批处理的调度任务来实现，比如凌晨两三点的时候在调度系统上起一个 Spark 调度任务把当天所有的数据重新跑一遍。

很显然在这个过程中，由于两个过程运行的时间是不一样的，跑的数据却相同，因此可能造成数据的不一致。因为某一条或几条数据的更新，需要重新跑一遍整个离线分析的链路，数据更新成本很大，同时需要维护离线和实时分析两套计算平台，整个上下两层的开发流程和运维成本其实都是非常高的。

为了解决 Lambda 架构带来的各种问题，就诞生了 Kappa 架构，这个架构大家应该也非常的熟悉。

2）Kappa 架构的痛点

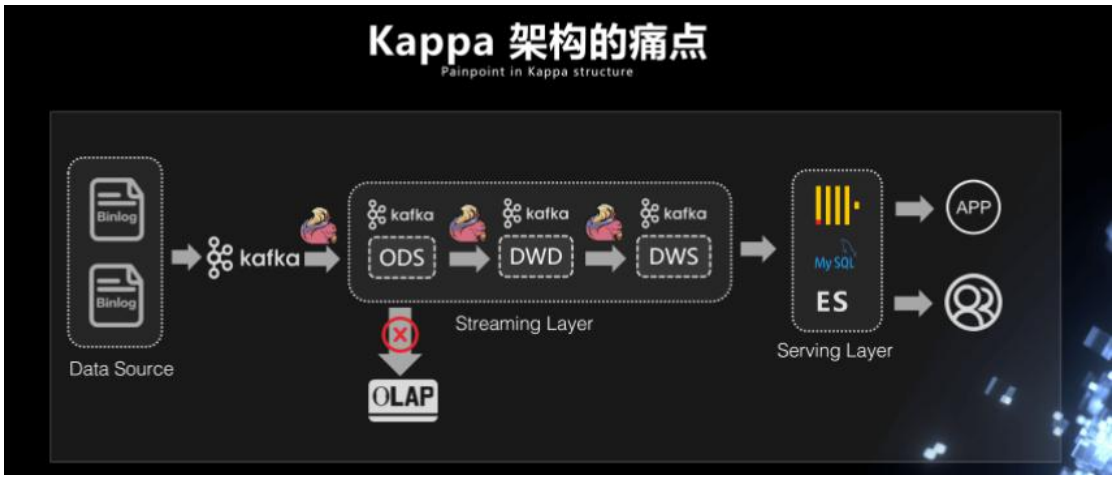
我们来讲一下 Kappa 架构，如下图，它中间其实用的是消息队列，通过用 Flink 将整个链路串联起来。Kappa 架构解决了 Lambda 架构中离线处理层和实时处理层之间由于引擎不一样，导致的运维成本和开发成本高昂的问题，但 Kappa 架构也有其痛点。

首先，在构建实时业务场景时，会用到 Kappa 去构建一个近实时的场景，但如果想对数仓中间层例如 ODS 层做一些简单的 OLAP 分析或者进一步的数据处理时，如将数据写到 DWD 层的 Kafka，则需要另外接入 Flink。同时，当需要从 DWD 层的 Kafka 把数据再导入到 Clickhouse，Elasticsearch，MySQL 或者是 Hive 里面做进一步的分析时，显然就增加了整个架构的复杂性。

其次，Kappa 架构是强烈依赖消息队列的，我们知道消息队列本身在整个链路上数据计算的准确性是严格依赖它上游数据的顺序，消息队列接的越多，发生乱序的可能性就越大。ODS 层数据一般是绝对准确的，把 ODS 层的数据发送到下一个 kafka 的时候就有可能发生乱序，DWD 层再发到 DWS 的时候可能又乱序了，这样数据不一致性就会变得很严重。

第三，Kafka 由于它是一个顺序存储的系统，顺序存储系统是没有办法直接在其上面利用 OLAP 分析的一些优化策略，例如谓词下推这类的优化策略，在顺序存储的 Kafka 上来实现是比较困难的事情。

那么有没有这样一个架构，既能够满足实时性的需求，又能够满足离线计算的要求，而且还能够减轻运维开发的成本，解决通过消息队列构建 Kappa 架构过程中遇到的一些痛点？答案是肯定的，后面的篇幅会详细论述。



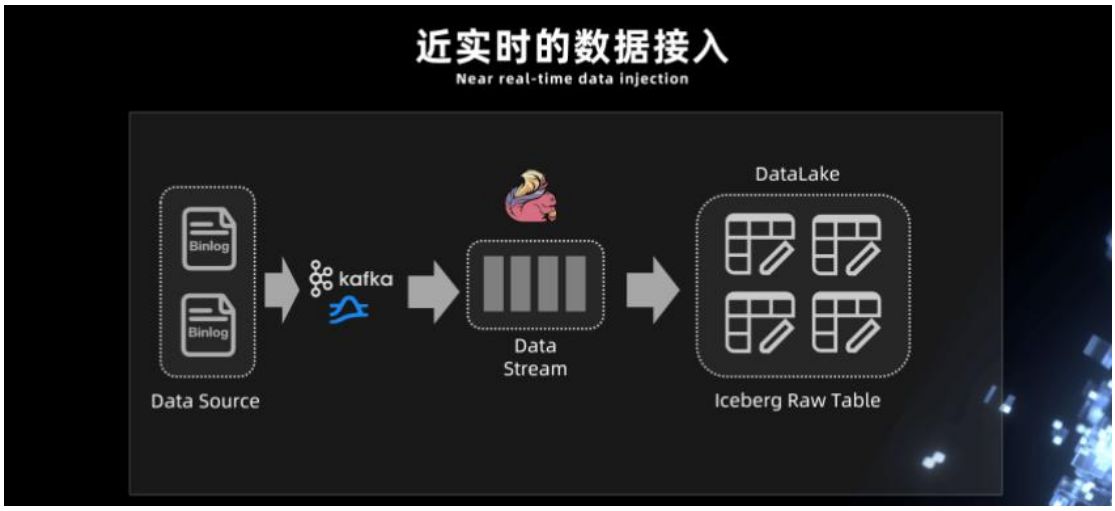
3）痛点总结

4）Flink+Iceberg 构建实时数仓

1. 近实时的数据接入

前面介绍了 Iceberg 既支持读写分离，又支持并发读、增量读、小文件合并，还可以支持秒级到分钟级的延迟，基于这些优势我们尝试采用 Iceberg 这些功能来构建基于 Flink 的实时全链路批流一体化的实时数仓架构。

如下图所示，Iceberg 每次的 commit 操作，都是对数据的可见性的改变，比如说让数据从不可见变成可见，在这个过程中，就可以实现近实时的数据记录。

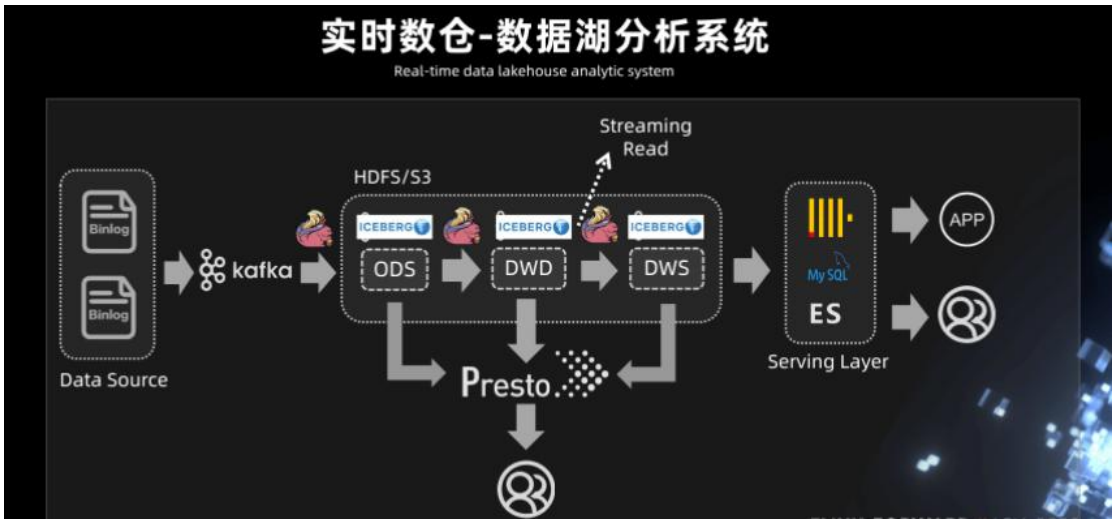


2. 实时数仓 - 数据湖分析系统

此前需要先进行数据接入，比如用 Spark 的离线调度任务去跑一些数据，拉取，抽取最后再写入到 Hive 表里面，这个过程的延时比较大。有了 Iceberg 的表结构，可以中间使用 Flink，或者 spark streaming，完成近实时的数据接入。

基于以上功能，我们再来回顾一下前面讨论的 Kappa 架构，Kappa 架构的痛点上面已经描述过，Iceberg 既然能够作为一个优秀的表格式，既支持 Streaming reader，又可以支持 Streaming sink，是否可以考虑将 Kafka 替换成 Iceberg?

Iceberg 底层依赖的存储是像 HDFS 或 S3 这样的廉价存储，而且 Iceberg 是支持 parquet、orc、Avro 这样的列式存储。有列式存储的支持，就可以对 OLAP 分析进行基本的优化，在中间层直接进行计算。例如谓词下推最基本的 OLAP 优化策略，基于 Iceberg snapshot 的 Streaming reader 功能，可以把离线任务天级别到小时级别的延迟大大的降低，改造成一个近实时的数据湖分析系统。



在中间处理层，可以用 presto 进行一些简单的查询，因为 Iceberg 支持 Streaming read，所以在系统的中间层也可以直接接入 Flink，直接在中间层用 Flink 做一些批处理或者流式计算的任务，把中间结果做进一步计算后输出到下游。

替换 Kafka 的优劣势：

总的来说，Iceberg 替换 Kafka 的优势主要包括：

- 实现存储层的流批统一
- 中间层支持 OLAP 分析
- 完美支持高效回溯
- 存储成本降低

当然，也存在一定的缺陷，如：

- 数据延迟从实时变成近实时
- 对接其他数据系统需要额外开发工作

秒级分析 - 数据湖加速：

由于 Iceberg 本身是将数据文件全部存储在 HDFS 上的，HDFS 读写这块对于秒级分析的场景，还是不能够完全满足我们的需求，所以接下去我们会在 Iceberg 底层支持 Alluxio 这样一个缓存，借助于缓存的能力可以实现数据湖的加速。这块的架构也在我们未来的一个规划和建设中。

最后

微信搜索公众号：五分钟学大数据，学更多大数据技术！

其他大数据技术文档可下方扫码关注获取：