建立用户画像首先需要建立数据仓库，用于存储用户标签数据。Hive是基于Hadoop的数据仓库工具，依赖于HDFS存储数据，提供的SQL语言可以查询存储在HDFS中的数据。开发时一般使用Hive作为数据仓库，存储标签和用户特征库等相关数据。

"数据仓库之父" W.H.Inmon 在《Building the Data Warehouse》（中文版《数据仓库（原书第4版）》）一书中定义数据仓库是"一个面向主题的、集成的、非易失的、随时间变化的、用来支持管理人员决策的数据集合"。

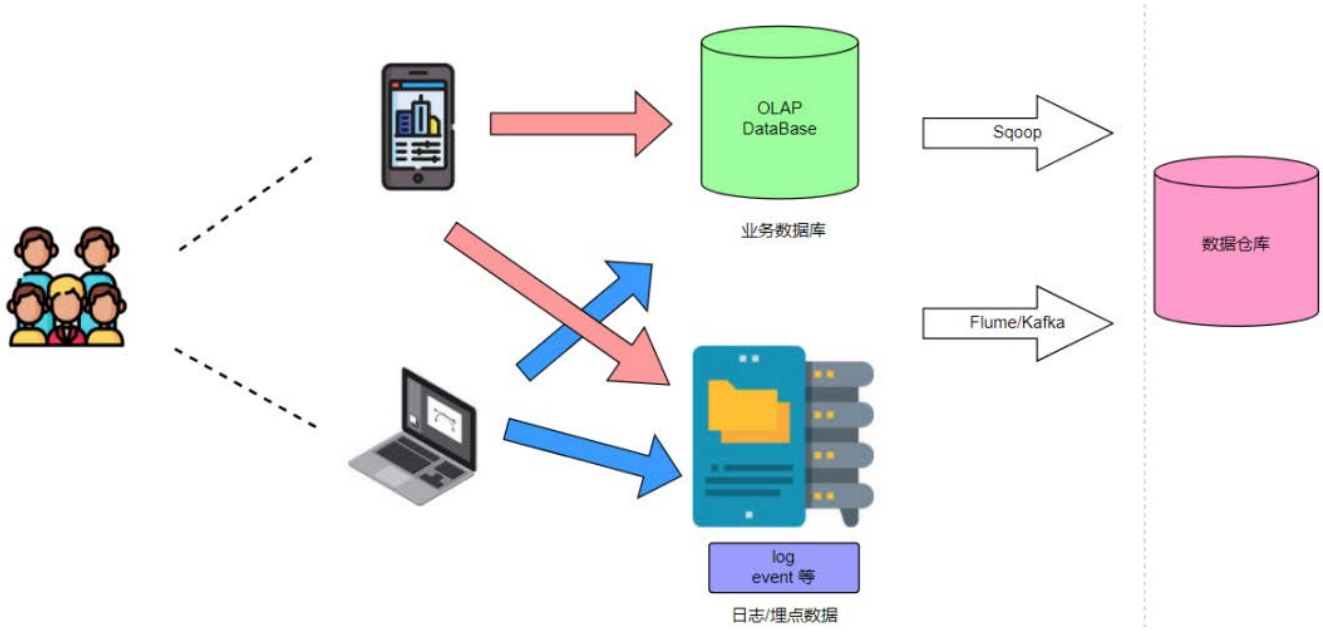
- 面向主题：业务数据库中的数据主要针对事务处理，各个业务系统之间是相互分离的，而数据仓库中的数据是按照一定主题进行组织的。

- 集成：数据仓库中存储的数据是从业务数据库中提取出来的，但并不是对原有数据的简单复制，而是经过了抽取、清理、转换（ETL）等工作。业务数据库记录的是每一项业务处理的流水账。这些数据不适合进行分析处理，进入数据仓库之前需要经过一系列计算，同时抛弃一些无关分析处理的数据。

- 非易失：业务数据库中一般只存储短期数据，因此其数据是不稳定的，记录的是系统中数据变化的瞬态。数据仓库中的数据大多表示过去某一时刻的数据，主要用于查询、分析，不像业务系统中的数据库一样经常修改，一般数据仓库构建完成后主要用于访问，不进行修改和删除。

- 随时间变化：数据仓库关注的是历史数据，按时间顺序定期从业务库和日志库里面载入新的数据进行追加，带有时间属性。

数据抽取到数据仓库的流程如下图所示。



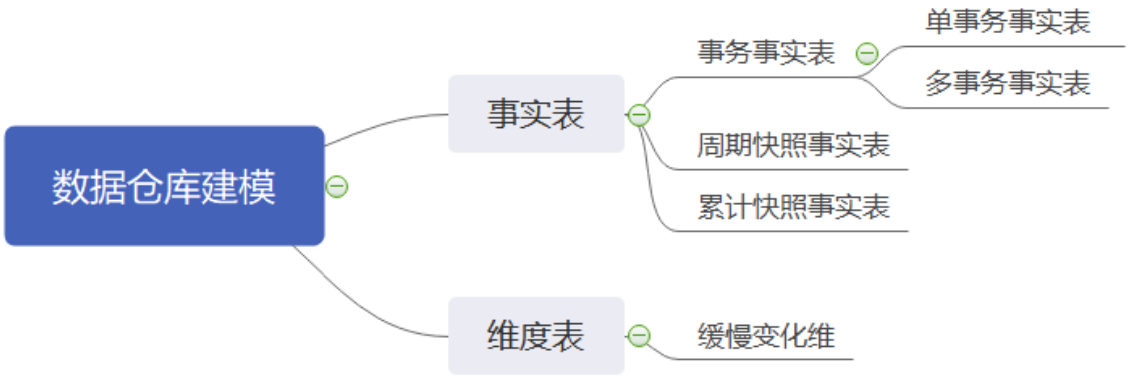
在数据仓库建模的过程中，主要涉及事实表和维度表的建模开发：

事实表主要围绕业务过程设计，就应用场景来看主要包括事务事实表，周期快照事实表和累计快照事实表:

- 事务事实表：用于描述业务过程，按业务过程的单一性或多业务过程可进一步分为单事务事实表和多事务事实表。其中单事务事实表分别记录每个业务过程，如下单业务记入下单事实表，支付业务记入支付事实表。多事务事实表在同一个表中包含了不同业务过程，如下单、支付、签收等业务过程记录在一张表中，通过新增字段来判断属于哪一个业务过程。当不同业务过程有着相似性时可考虑将多业务过程放到多事务事实表中。

- 周期快照事实表:在一个确定的时间间隔内对业务状态进行度量。例如查看一个用户的近1年付款金额、近1年购物次数、近30日登录天数等。

- 累计快照事实表:用于查看不同事件之间的时间间隔，例如分析用户从购买到支付的时长、从下单到订单完结的时长等。一般适用于有明确时间周期的业务过程。



维度表主要用于对事实属性的各个方面描述，例如，商品维度包括商品的价格、折扣、品牌、原厂家、型号等方面信息。维度表开发的过程中，经常会遇到维度缓慢变化的情况，对于缓慢变化维一般会采用：①重写维度值，对历史数据进行覆盖；②保留多条记录，通过插入维度列字段加以区分；③开发日期分区表，每日分区数据记录当日维度的属性；④开发拉链表按时间变化进行全量存储等方式进行处理。

在画像系统中主要使用Hive作为数据仓库，开发相应的维度表和事实表来存储标签、人群、应用到服务层的相关数据。HiveSQL高级进阶技巧。

分区存储

如果将用户标签开发成一张大的宽表，在这张宽表下放几十种类型标签，那么每天该画像宽表的ETL作业将会花费很长时间，而且不便于向这张宽表中新增标签类型。

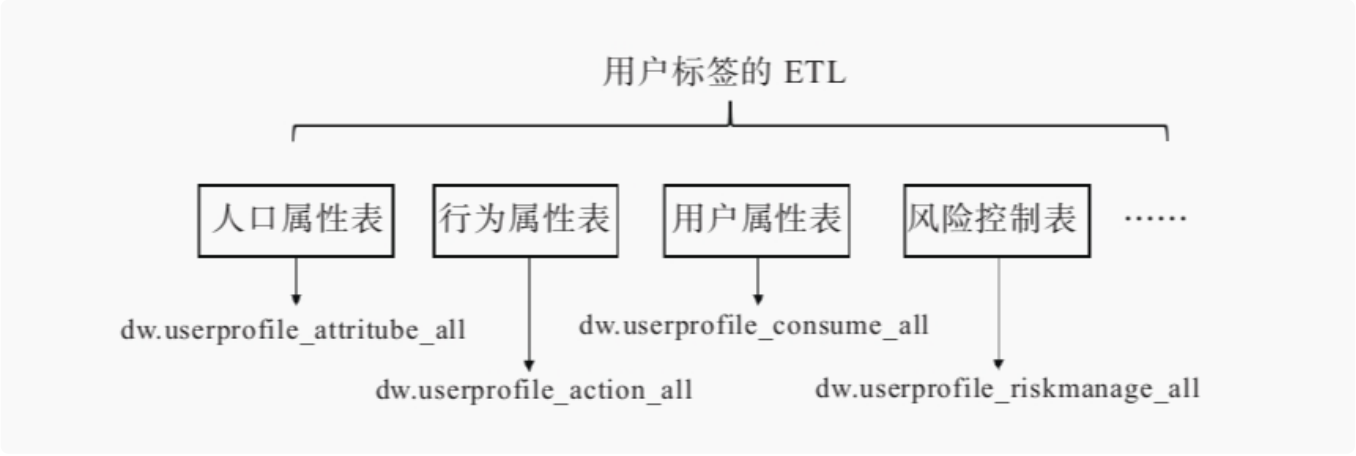
要解决这种ETL花费时间较长的问题，可以从以下几个方面着手：

- 将数据分区存储，分别执行作业;
- 标签脚本性能调优;
- 基于一些标签共同的数据来源开发中间表。

下面介绍一种用户标签分表、分区存储的解决方案。

根据标签指标体系的人口属性、行为属性、用户消费、风险控制、社交属性等维度分别建立对应的标签表进行分表存储对应的标签数据。如下图所示。

- 人口属性表：dw.userprofile_attritube_all
- 行为属性表：dw.userprofile_action_all
- 用户消费表：dw.userprofile_consume_all
- 风险控制表：dw.userprofile_riskmanage_all
- 社交属性表：dw.userprofile_social_all



例如创建用户的人口属性宽表：

字段	字段定义	示例	备注
userid	用户 id	44463729	用户没被打上该标签，数值为0，如果有确定数值的该字段存储具体数值（如年龄标签存放具体年龄），如果没有具体数值则放数值1（如性别标签存储 1）
ATTRITUBE_U_01_001	男性	1	
ATTRITUBE_U_01_002	女性	0	
ATTRITUBE_U_02_001	高活跃用户	1	
ATTRITUBE_U_02_002	中等活跃用户	0	
字段	字段定义	示例	备注
ATTRITUBE_U_02_003	非活跃用户	0	
ATTRITUBE_U_03_001	年龄	28	
ATTRITUBE_U_04_001	重要价值 (RFM 价制度)	0	
ATTRITUBE_U_04_002	重要发展 (RFM 价制度)	0	
ATTRITUBE_U_04_003	重要保持 (RFM 价制度)	0	
ATTRITUBE_U_04_004	重要挽留 (RFM 价制度)	1	
……	……	……	
data_date	日期	20190101	数据分区日期

同样的，用户其他id维度（如cookieid、deviceid、registerid等）的标签数据存储，也可以使用上面案例中的表结构。

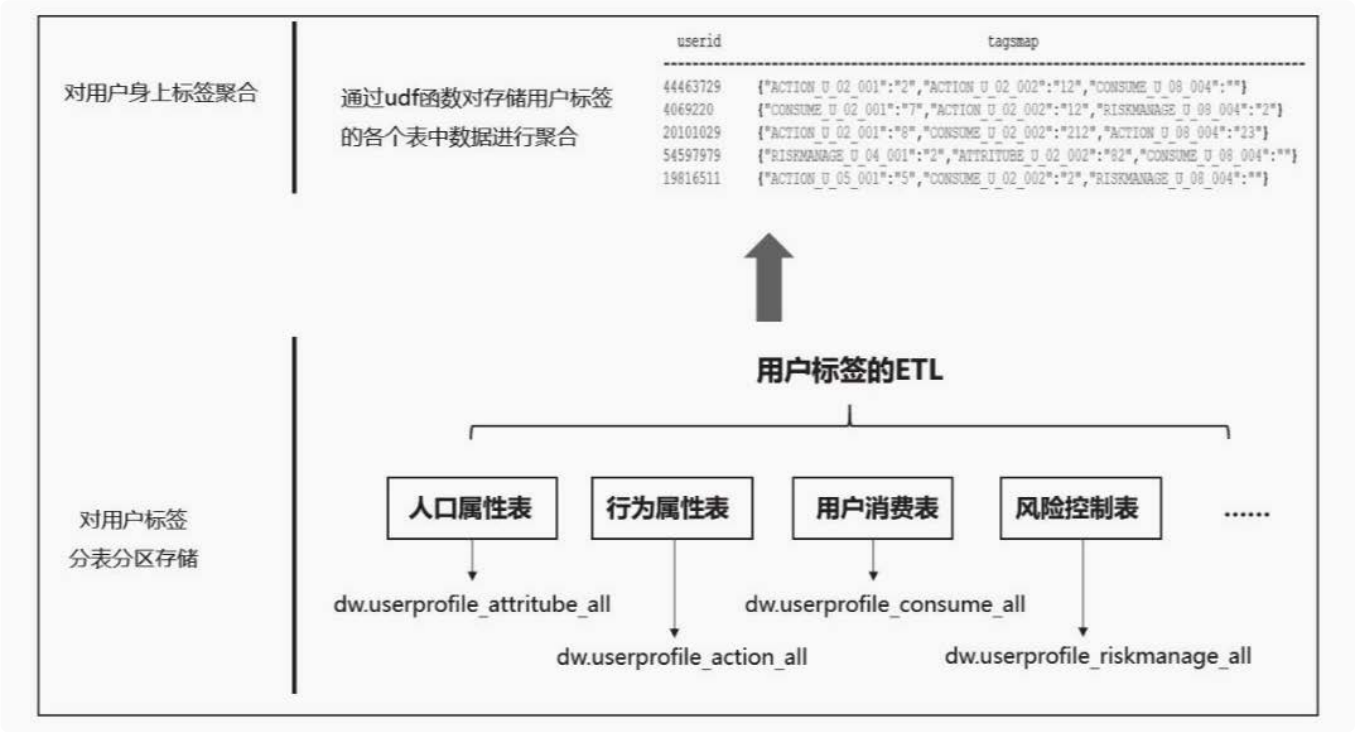
在上面的创建中通过设立人口属性维度的宽表开发相关的用户标签，为了提高数据的插入和查询效率，在Hive中可以使用分区表的方式，将数据存储在不同的目录中。在Hive使用select查询时一般会扫描整个表中所有数据，将会花费很多时间扫描不是当前要查询的数据，为了扫描表中关心的一部分数据，在建表时引入了partition的概念。在查询时，可以通过Hive的分区机制来控制一次遍历的数据量。Hive SQL优化思路。

标签汇聚

在上面一节提到的案例中，用户的每个标签都插入到相应的分区下面，但是对一个用户来说，打在他身上的全部标签存储在不同的分区下面。为了方便分析和查询，需要将用户身上的标签做聚合处理。

标签汇聚后将一个每个用户身上的全量标签汇聚到一个字段中，表结构设计如下：

```
CREATE TABLE `dw.userprofile_userlabel_map_all`
(
    `userid`     string COMMENT 'userid',
    `userlabels` map<string,string> COMMENT 'tagsmap',
)
    COMMENT 'userid 用户标签汇聚'
    PARTITIONED BY ( `data_date` string COMMENT '数据日期')
```



开发udf函数“cast_to_json”将用户身上的标签汇聚成json字符串，执行命令将按分区存储的标签进行汇聚:

```
insert overwrite table dw.userprofile_userlabel_map_all partition(data_date= "data_date")
  select userid,
         cast_to_json(concat_ws(',',collect_set(concat(labelid,':',labelweight)))) as userlabels
      from “用户各维度的标签表”
     where data_date= " data_date "
group by userid
```

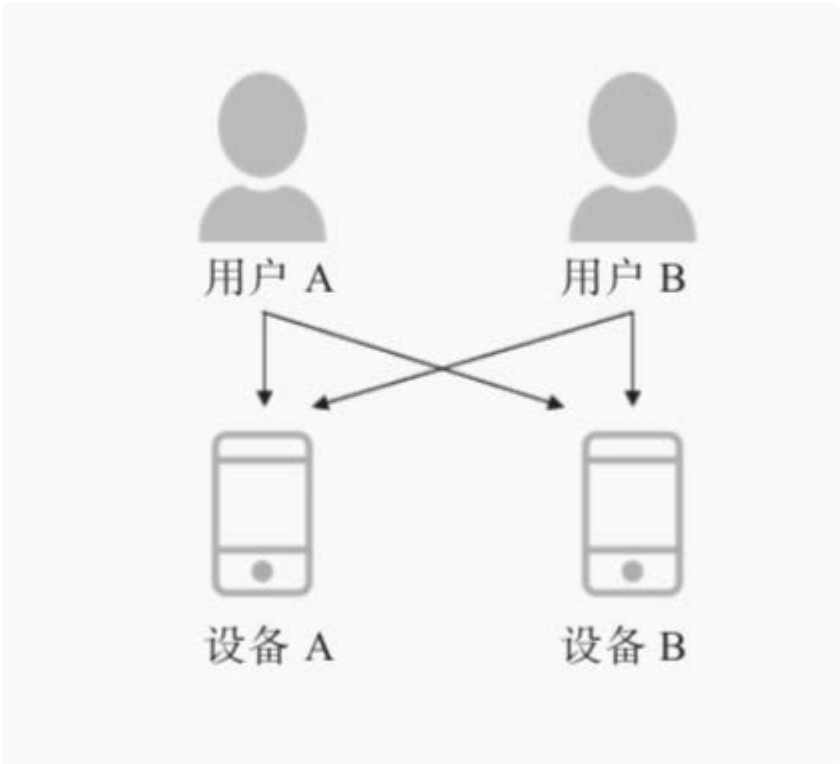
汇聚后用户标签的存储格式如图所示:

userid	userlabels	data_date
44463729	{"ACTION_U_02_001":"2","ACTION_U_02_002":"12","CONSUME_U_08_004":""}	20190101
4069220	{"CONSUME_U_02_001":"7","ACTION_U_02_002":"12","RISKMANAGE_U_08_004":"2"}	20190101
20101029	{"ACTION_U_02_001":"8","CONSUME_U_02_002":"212","ACTION_U_08_004":"23"}	20190101
54597979	{"RISKMANAGE_U_04_001":"2","ATTRITUBE_U_02_002":"82","CONSUME_U_08_004":""}	20190101
19816511	{"ACTION_U_05_001":"5","CONSUME_U_02_002":"2","RISKMANAGE_U_08_004":""}	20190101

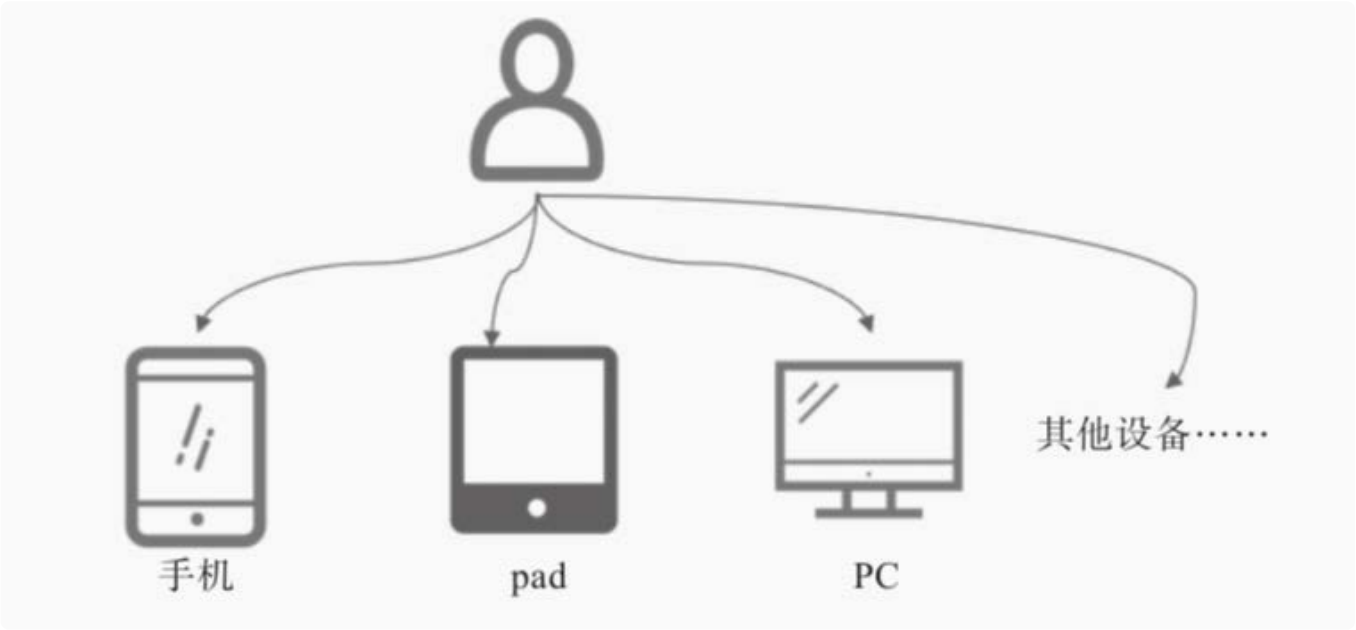
将用户身上的标签进行聚合便于查询和计算。例如，在画像产品中，输入用户id后通过直接查询该表，解析标签id和对应的标签权重后，即可在前端展示该用户的相关信息

ID-MAP

开发用户标签的时候，有项非常重要的内容——ID-MApping，即把用户不同来源的身份标识通过数据手段识别为同一个主体。用户的属性、行为相关数据分散在不同的数据来源中，通过ID-MApping能够把用户在不同场景下的行为串联起来，消除数据孤岛。下图展示了用户与设备间的多对多关系。解密One ID中的核心技术ID-Mapping。



下图展示了同一用户在不同平台间的行为示意图。



举例来说，用户在未登录App的状态下，在App站内访问、搜索相关内容时，记录的是设备id（即cookieid）相关的行为数据。而用户在登录App后，访问、收藏、下单等相关的行为记录的是账号id（即userid）相关行为数据。虽然是同一个用户，但其在登录和未登录设备时记录的行为数据之间是未打通的。通过ID-MApping打通 userid 和 cookieid 的对应关系，可以在用户登录、未登录设备时都能捕获其行为轨迹。

下面通过一个案例介绍如何通过Hive的ETL工作完成ID-Mapping的数据清洗工作。

缓慢变化维是在维表设计中常见的一种方式，维度并不是不变的，随时间也会发生缓慢变化。如用户的手机号、邮箱等信息可能会随用户的状态变化而改变，再如商品的价格也会随时间变化而调整上架的价格。因此在设计用户、商品等维表时会考虑用缓慢变化维来开发。同样，在设计ID-Mapping表时，由于一个用户可以在多个设备上登录，一个设备也能被多个用户登录，所以考虑用缓慢变化维表来记录这种不同时间点的状态变化。

userid	cookieid	start_date	end_date
44463729	07427323-40FB-46B8-8A3D-D67FBC	20190101	20190105
44463729	73C7C634-E5FE-474C-80C0-5B1581	20190106	99991231
54597979	07006ca2-a9bf-463a-828a-132f32	20190113	99991231
20101029	09e40e53-e9b0-424f-837f-c225b2	20190101	20190102
20101029	D0BC25C4-EE11-41AD-94D0-801CA1	20190103	20190105
20101029	e598027d-5412-4e7b-84a2-1d59e5	20190106	99991231

拉链表是针对缓慢变化维表的一种设计方式，记录一个事物从开始到当前状态的全部状态变化信息。

在上图中，通过拉链表记录了userid每一次关联到不同cookieid的情况。如userid为44463729的用户，

在20190101这天登录某设备，在6号那天变换了另一个设备登录。其中start_date表示该记录的开始日期，end_date表示该记录的结束日期，当end_date为99991231时，表示该条记录当前仍然有效。

首先需要从埋点表和访问日志表里面获取到cookieid和userid同时出现的访问记录。下面案例中，ods.page_event_log是埋点日志表，ods.page_view_log是访问日志表，将获取到的userid和cookieid信息插入cookieid-userid关系表（ods.cookie_user_signin）中。代码执行如下：

```
INSERT OVERWRITE TABLE ods.cookie_user_signin PARTITION (data_date = '${data_date}')
  SELECT t.*
    FROM (
         SELECT userid,
                cookieid,
                from_unixtime(eventtime,'yyyyMMdd') as signdate
           FROM ods.page_event_log      -- 埋点表
           WHERE data_date = '${data_date}'
        UNION ALL
         SELECT userid,
                cookieid,
                from_unixtime(viewtime,'yyyyMMdd') as signdate
           FROM ods.page_view_log   -- 访问日志表
           WHERE data_date = '${data_date}'
           ) t
```

创建ID-Map的拉链表，将每天新增到ods.cookie_user_signin表中的数据与拉链表历史数据做比较，如果有变化或新增数据则进行更新。

```
CREATE TABLE `dw.cookie_user_zippertable`(
`userid` string COMMENT '账号ID',
`cookieid` string COMMENT '设备ID',
`start_date` string COMMENT 'start_date',
`end_date` string COMMENT 'end_date')
COMMENT 'id-map拉链表'
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
```

创建完成后，每天ETL调度将数据更新到ID-Mapping拉链表中，任务执行如下。

```
INSERT OVERWRITE TABLE dw.cookie_user_zippertable
SELECT t.*
  FROM (
      SELECT t1.user_num,
             t1.mobile,
             t1.reg_date,
             t1.start_date,
             CASE WHEN t1.end_date = '99991231' AND t2.userid IS NOT NULL THEN '${data_date}'
```

```
                ELSE t1.end_date
            END AS end_date
       FROM dw.cookie_user_zippertable t1
    LEFT JOIN (  SELECT *
                 FROM ods.cookie_user_signin
                WHERE data_date='${data_date}'
             )t2
          ON t1.userid = t2.userid
UNION
       SELECT userid,
              cookieid,
              '${data_date}' AS start_date,
              '99991231' AS end_date
        FROM ods.cookie_user_signin
       WHERE data_date = '${data_date
       }'
          ) t
```

数据写入表中，如上图所示。

对于该拉链表，可查看某日（如20190801）的快照数据。

```
select  *
from dw.cookie_user_zippertable
where start_date<='20190801' and end_date>='20190801'
```

例如，目前存在一个记录userid和cookieid关联关系的表，但是为多对多的记录（即一个userid对应多条cookieid记录，以及一条cookieid对应多条userid记录）。这里可以通过拉链表的日期来查看某个时间点userid对应的cookieid。查看某个用户（如32101029）在某天（如20190801）关联到的设备id。

```
select cookieid
from dw.cookie_user_zippertable
where userid='32101029' and start_date<='20190801' and end_date>='20190801'
```

userid	cookieid	start_date	end_date
32101029	09e40e53-e9b0-424f-837f-c225b2	20190101	20190102
32101029	D0BC25C4-EE11-41AD-94D0-801CA1	20190103	20190105
32101029	e598027d-5412-4e7b-84a2-1d59e5	20190106	99991231

上图可看出用户'32101029'在历史中曾登录过3个设备，通过限定时间段可找到特定时间下用户的登录设备。

在开发中需要注意关于userid与cookieid的多对多关联，如果不加条件限制就做关联，很可能引起数据膨胀问题：

> 在实际应用中，会遇到许多需要将userid和cookieid做关联的情况。例如，需要在userid维度开发出该用户近30日的购买次数、购买金额、登录时长、登录天数等标签。前两个标签可以很容易地从相应的业务数据表中根据算法加工出来，而登录时长、登录天数的数据存储在相关日志数据中，日志数据表记录的userid与cookieid为多对多关系。因此在结合业务需求开发标签时，要确定好标签口径定义。

小结

本期内容通过案例介绍了将userid 和 cookieid 打通的一种解决方案，实践中还存在需要将用户在不同平台间（如Web端和App端）行为打通的应用场景。