

**【多易教育】**

**大型离线综合项目**

**<yiee-数据运营系统>**

# 项目整体概述

*项目码云地址： https://gitee.com/hunter\_d/data\_yiee.git*

## 项目背景

假设本公司是一家“电商、泛电商”企业

或者各种各样的： 主营业务在线上(app/网站)进行的公司！（外卖公司、各类app<比如：下厨房；头条；安居客；内涵段子；斗鱼；每日优鲜>、页游、唯品会、拼多多、当当、苏宁易购、国美在线、携程、中华会计网校、多易教育、小象学院、慕课网、尚德教育、有钱花、上哪借 ……）

这类公司，都需要针对用户的线上**访问行为、消费行为**进行统计分析！以支撑公司的业务运营，提高转化率，改善公司运营效果——赚取更大利润！

这个需求，可以通过构建一个**数仓系统**来支撑！

这类公司，往往还需要对每个用户进行个性化推荐/营销运营，改善用户体验，提高用户粘性，增大销售额，——赚取更大利润！

这个需求，可以通过对用户进行精准**用户画像系统（深入分析用户后给用户打上各种标签：年龄，性别，价值指数，行为习惯，消费习惯......）**、**推荐系统**来实现！

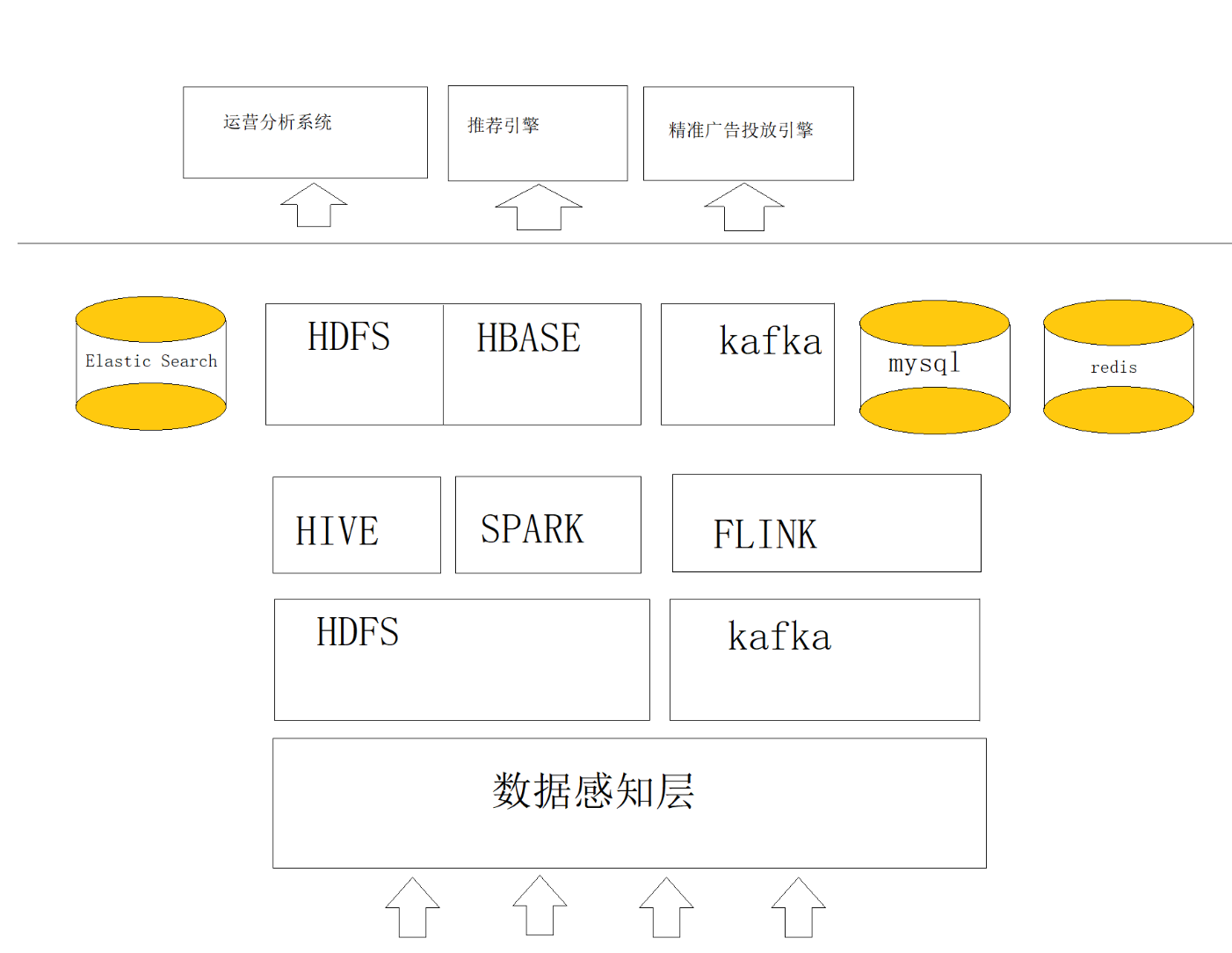
用户画像，本身不光可以为**个性化推荐**服务，还可以为公司**运营提供数据**支撑！比如：减少用户流失率，增加用户粘性，为一些营销活动的设计提供数据依据！

基于上述需求，可以使用离线方式实现，也可以使用实时方式实现，也可以离线和实时结合起来做；

一般公司会既有离线（系统化的，时间跨度长的，运算量大的任务），也有实时；

所以，我们还需要一个**实时数据运营系统**

## 项目流程架构



**其中主要涉及到3大数据处理系统：**

* + 离线数据仓库/实时指标计算
  + 用户画像
  + 个性化推荐

## 技术选型

项目中所涉及到的技术：

采集：

flume ： 分布式日志数据汇聚

sqoop ： 离线批量抽取数据库

cannal ： 实时数据库数据逐条监听、抽取

nginx插件：前端

存储：

hdfs

redis

hbase/elastic search ：用户画像标签数据的存储查询服务

kafka ： 实时计算的缓冲

mysql

运算：

hive

mapreduce

spark core/ sql /streaming

flink/~~storm~~

算法：

通用统计算法sum count max min .......；

图计算： SparkGraphx；

机器学习算法：

KNN k近邻

KMEANS k均值

Naive Bayes朴素贝叶斯

随机森林

协同过滤算法

各类相似度算法：欧几里得距离、余弦相似度、皮尔逊相关系数

NLP算法：

TF-IDF 关键词提取

文本向量化算法

文本相似度算法

文本分类算法

关联规则分析

FP-GROWTH算法

APRIORI算法

# 数仓理论

## 什么数据仓库

概念上：数据仓库，英文名称为Data WareHouse，可简写为DW或DWH。

数据仓库，是为企业所有级别的决策制定过程，提供所有类型数据支持的战略集合。为需要业务智能的企业，提供指导业务流程改进、监视时间、成本、质量以及控制。

*数据仓库（Data Warehouse）是一个****面向主题****的（Subject Oriented）、****集成****的( Integrate）、相****对稳定****的（Non-Volatile）、****反映历史变化****（Time Variant）的数据集合，用于支持管理、运营决策。*

技术上：

不可更新的，反映历史变化的，面向主题的，大容量的数据“仓库”

## 数据库与数据仓库

### 数据库

通常指的是**数据库软件，比如mysql，oracle，sqlserver，db2**

数据库软件的应用领域非常广泛！

数据库软件用的最多的应用场景，就是**联机事务处理OLTP**

比如：电商系统中的数据库，银行系统中的数据库，办公管理系统OA中的数据库，沃尔玛的销售系统中的数据库

**联机事务处理的要求**：

第一，时刻保持联机在线状态，而且要求各种操作（增删改查）请求能得到实时响应

第二，能支撑强大的事务控制要求

像mysql、oracle、db2、sqlserver这些传统的关系型数据库，都是能够用来很好地胜任联机事务处理场景的！

### 数据库VS数据仓库

在业务系统中的联机事务处理数据库，随着业务运行时长的增长，库中的业务数据越来越多，会产生两个问题：1）联机事务处理的速度会越来越慢2）针对这么大量的数据进行报表分析会耗费太多运算和IO资源，导致联机事务处理被阻塞

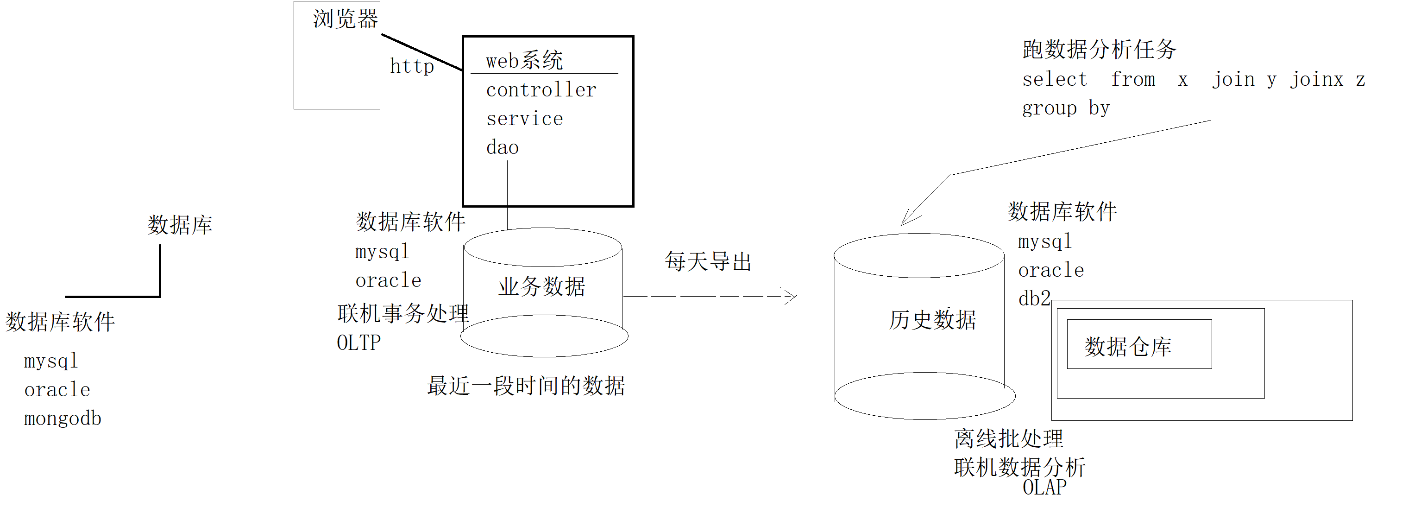
由此，引入数据仓库的概念： 将联机事务处理数据库中的数据不断导入另外一个数据库库系统进行存储，而且也可以在这另外的数据库系统上对各种长时间跨度数据的复杂数据分析、统计！

这“另一个数据库”就是承担了“数据仓库”的角色！

而根据构建“数据仓库”的软件特性不同，数据仓库所支持的数据分析性能也不同：

1.如果是用各类关系型数据库如mysql、oracle等构建的数据仓库，可以支持离线数据分析，也可以支持联机数据分析！

2.如果是hive这种软件构建的数据库，只能支持离线数据分析！



## 数仓主题

### 数仓主题的含义

用户分析主题

今天的日新有多少？今天的日活有多少？今天的日新相对昨天的环比增长？今天的日新相对日活的占比？用户留存比率有多少？

流量分析主题

今天的有多少pv(页面访问量），有多少uv（访问用户数），哪些页面的pv最多，哪些商品的pv最多？

app分析主题

今天的用户中，appv2.0使用占比有多少，appv3.0使用占比有多少，有多少用户做了app升级？

GMV销量分析主题

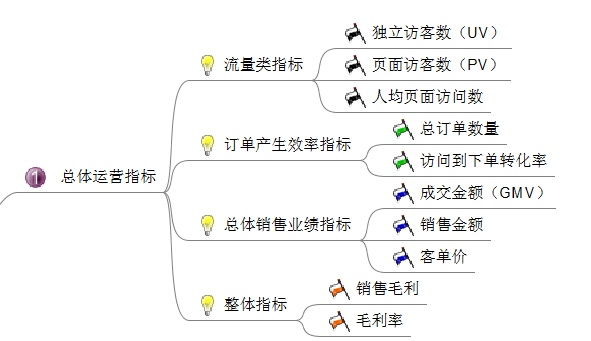
今天的销售总额有多少？今天每类一级品类的商品销售额分别是多少？今天哪个一级品类的销售额最多？今天的销售额环比昨日的增长率是多少？

### 电商行业数仓核心主题

****

#### 1）总体运营指标

从流量、订单、总体销售业绩、整体指标进行把控，起码对运营的电商平台有个大致了解，到底运营的怎么样，是亏是赚。



#### 2）网站流量指标

即对访问你网站的访客进行分析，基于这些数据可以对网页进行改进，以及对访客的行为进行分析等等



#### 3）销售转化指标

分析从下单到支付整个过程的数据，帮助你提升商品转化率。也可以对一些频繁异常的数据展开分析。



#### 4）客户价值指标

这里主要就是分析客户的价值，可以建立RFM价值模型，找出那些有价值的客户，精准营销等等。



#### 5）商品类指标

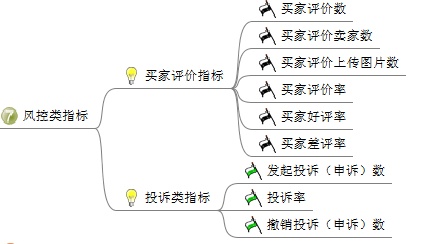
主要分析商品的种类，那些商品卖得好，库存情况，以及可以建立关联模型，分析那些商品同时销售的几率比较高，而进行捆绑销售，有点像啤酒喝尿布的故事。



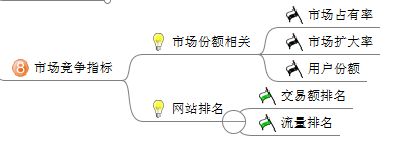
6.市场营销活动指标，主要监控某次活动给电商网站带来的效果，以及监控广告的投放指标



7. 风控类指标：分析卖家评论，以及投诉情况，发现问题，改正问题。

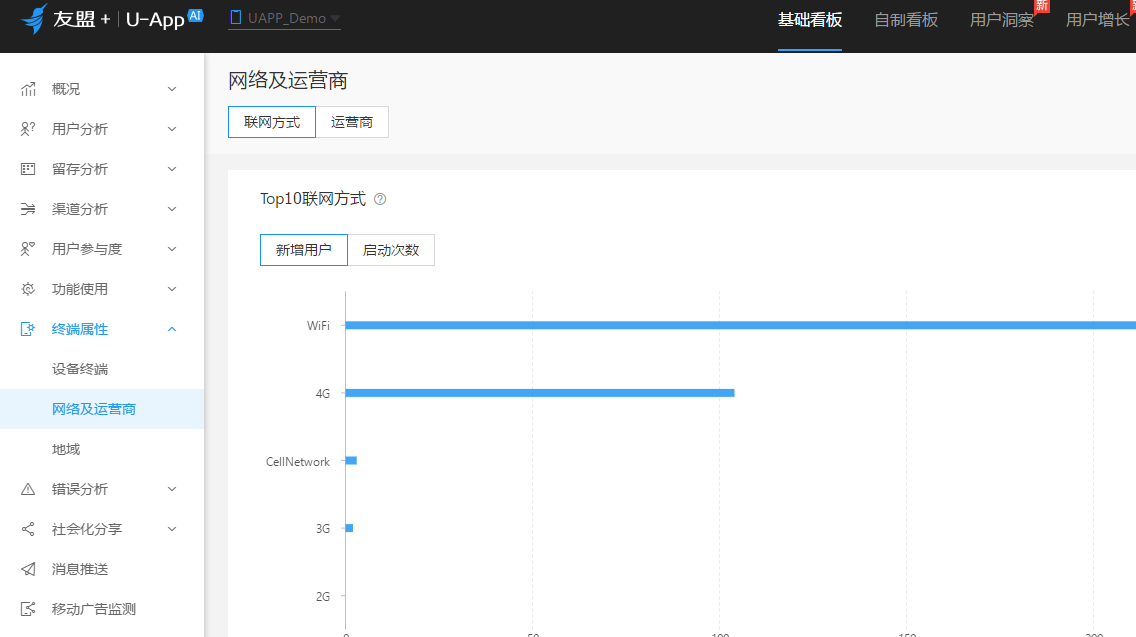


8.市场竞争指标：主要分析市场份额以及网站排名，进一步进行调整



### 本项目要实现的分析主题

如下图所示：



除图中所列主题以外，本项目还将实现以下分析主题：

1. 转化率分析
2. 销售分析
3. 广告效果分析
4. 用户行为规律分析

## 数仓分层

### 数仓概述

仓库中的数据，往往是**分层管理、分层计算**的：

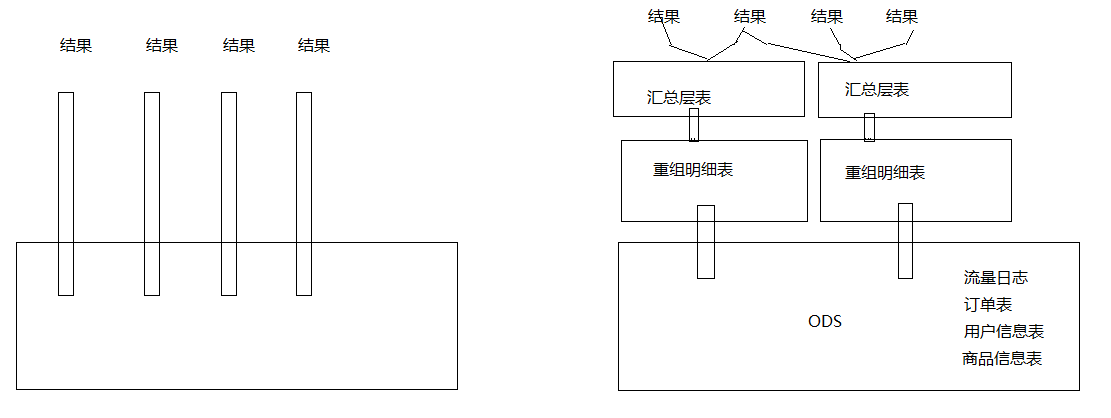
ADS层： 应用服务层

DWS层：数仓汇总层

DWD层：数仓明细层

ODS层：操作数据（最原始的数据）层-- **贴源层**

分层的意义：数据管理更明晰！运算复用度更高！需求开发更快捷！



### 分层的原因

#### ① 空间换时间

通过建设多层次的数据模型供用户使用，避免用户直接使用操作型数据，可以更高效的访问数据。

#### ② 把复杂问题简单化

将一个复杂的任务分解成多个步骤来完成，每一层只处理单一的步骤，比较简单和容易理解。而且便于维护数据的准确性，当数据出现问题之后，可以不用修复所有的数据，只需要从有问题的步骤开始修复。

#### ③ 便于解耦 “底层业务的变化”

随着业务的变化，只需要调整底层的数据，应用层对业务的调整零感知.

### 分层的价值

#### ① 高效的数据组织形式【易维护】

面向主题的特性决定了数据仓库拥有业务数据库所无法拥有的高效的数据组织形式，更加完整的数据体系，清晰的数据分类和分层机制。因为所有数据在进入数据仓库之前都经过清洗和过滤，使原始数据不再杂乱无章，基于优化查询的组织形式，有效提高数据获取、统计和分析的效率。

#### ② 时间价值【高性能】

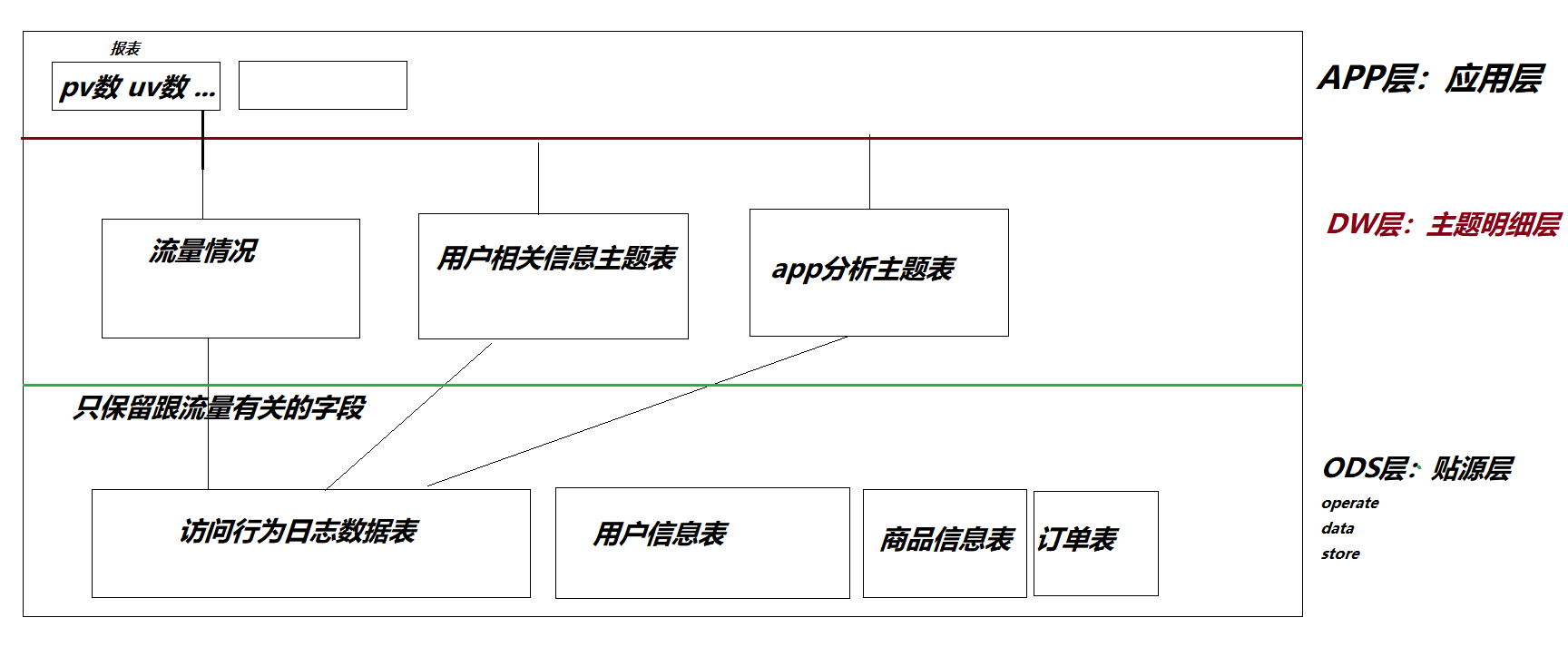
数据仓库的构建将大大缩短获取信息的时间，数据仓库作为数据的集合，所有的信息都可以从数据仓库直接获取，数据仓库的最大优势在于一旦底层从各类数据源到数据仓库的ETL流程构建成型，那么每天就会有来自各方面的信息通过自动任务调度的形式流入数据仓库，从而使一切基于这些底层信息的数据获取的效率达到迅速提升。

从应用来看，使用数据仓库可以大大提高数据的查询效率，尤其对于海量数据的关联查询和复杂查询，所以数据仓库有利于实现复杂的统计需求，提高数据统计的效率。

#### ③ 集成价值【简单化】

数据仓库是所有数据的集合，包括日志信息、数据库数据、文本数据、外部数据等都集成在数据仓库中，对于应用来说，实现各种不同数据的关联并使多维分析更加方便，为从多角度多层次地数据分析和决策制定提供的可能。

数仓分层示意：



## 数仓建模理论

### 什么是数据模型

*数据模型是数据特征的抽象。数据是描述事物的符号记录，模型是现实世界的抽象。数据模型从抽象层次上描述了系统静态特征、动态行为和约束条件为数据库系统信息表示与操作提供了一个抽象的框架。简单讲数据模型就是数据组织和存放的方法，它强调从业务、数据存取和使用角度合理存储数据。*

数据模型所描述的内容有三部分：数据结构、数据操作、数据约束。

* 数据结构：主要描述数据的类型、内容、性质以及数据间的联系等，是目标类型的集合。
* 数据操作：主要描述在相应的数据结构上的操作类型和操作方式。
* 数据约束：主要描述数据结构内数据间的语法词义联系、他们之间的制约和依存关系，以及数据动态变化的规则，以保证数据的正确、有效和相容。它是完整性规则的集合，用以限定符合数据模型的数据库状态，以及状态的变化。*约束条件可以按不同的原则划分为数据值的约束和数据间联系的约束；静态约束和动态约束；实体约束和实体间的参照约束等。*

数据仓库模型是

### 3.2.2为什么要数据模型

从性能、成本、效率、质量的角度看**：**

* 性能：良好的数据模型能帮助我们快速查询所需要的数据，减少数据的I/O吞吐。
* 成本：良好的数据模型能极大减少不必要的数据冗余，也能实现计算结果复用，极大地降低大数据系统中的存储和计算成本。
* 效率：良好的数据模型能极大地改善用户使用数据的体验，提高使用数据的效率。
* 质量：良好的数据模型能改善数据统计口径的不一致，减少数据计算错误的可能性。

因此，毋庸置疑，大数据系统需要数据模型方法来帮助更好地组织和存储数据，以便在性能、成本、效率和质量之间取得最佳平衡。

从业务和数仓建设的角度看**：**

在数据仓库的建设中，我们一再强调需要数据模型，那么数据模型究竟为什么这么重要呢？首先我们需要了解整个数据仓库的建设的发展史。数据仓库的发展大致经历了这样的三个过程：

* 简单的报表阶段：这个阶段，系统的主要目标是解决一些日常工作中业务人员需要的报表，以及生成一些简单的能够帮助领导进行决策所需要的汇总数据。这个阶段的大部分表现形式为数据库和前端报表工具。
* 数据集市阶段：这个阶段主要是根据某个业务部门的需要，进行一定的数据的采集，整理，按照业务人员的需求，进行多维报表的展现，能够提供对特定业务指导的数据，并且能够提供特定的领导决策数据。
* 数据仓库阶段：这个阶段主要是按照一定的数据模型，对整个企业的数据进行采集整理，并且能够按照各个业务部门的需要，提供跨部门的，完全一致的业务报表数据，能够通过数据仓库生成对业务具有指导性的数据，同时为领导决策提供全面的数据支持。

通过对数据仓库建设的发展阶段，我们能够看出，*数据仓库的建设和数据集市的建设的重要区别就在于数据模型的支持*。因此，数据模型的建设，对于我们数据仓库的建设，有着决定性的意义。 一般来说，数据模型的建设主要能够帮助我们解决以下的一些问题：

* *进行全面的业务梳理，改进业务流程*。在业务模型建设的阶段，能够帮助我们的企业或者管理机构对本单位的业务进行全面的梳理。*通过业务模型的建设，我们应该能够全面了解该单位的业务架构图和整个业务的运行情况，能够将业务按照特定的规律进行分门别类和程序化，同时，帮助我们进一步的改进业务的流程，提高业务效率，指导我们业务部门的生产*。
* *建设全方位的数据视角，消灭信息孤岛和数据差异*。通过数据仓库的模型建设，能够为企业提供一个整体的数据视角，不再是各个部门只是关注自己的数据，而且通过模型的建设，勾勒出部门之间的联系，帮助消灭各部门之间的信息孤岛的问题，更为重要的是，通过数据模型的建设，能够保证这个企业的数据一致性，各个部门之间数据的差异将会得到有效解决。
* *解决业务的变动和数据仓库的灵活性*。通过数据模型的建设，能够很好的分离出底层技术的实现和上层业务的展现。当上层业务发生变化时，通过数据模型，底层的技术实现可以非常轻松的完成业务的变动，从而达到整个数据仓库的灵活性。
* *帮助数据仓库系统本身的建设*。通过数据仓库的模型建设，开发人员和业务人员能够很容易的达成系统建设范围的界定，以及长期目标的规划，从而能够使整个项目组明确当前的任务，加快这个系统建设的速度。

### 数据仓库建模阶段划分



从上图我们可以清楚地看出，数据仓库的数据建模大致分为四个阶段：

1. 业务建模，这部分建模工作，主要包含以下几个部分：

* 划分整个单位的业务，一般按照业务部门的划分，进行各个部分之间业务工作的界定，理清各业务部门之间的关系。
* 深入了解各个业务部门内的具体业务流程并将其程序化。
* 提出修改和改进业务部门工作流程的方法并程序化。
* 数据建模的范围界定，整个数据仓库项目的目标和阶段划分。

2. 领域概念建模，这部分的建模工作，主要包含以下几个部分：

* 抽取关键业务概念，并将之抽象化。
* 将业务概念分组，按照业务主线聚合类似的分组概念。
* 细化分组概念，理清分组概念内的业务流程并抽象化。
* 理清分组概念之间的关联，形成完整的领域概念模型。

3. 逻辑建模，这部分的建模工作，主要包含以下几个部分：

* 业务概念实体化，并考虑其具体的属性
* 事件实体化，并考虑其属性内容
* 说明实体化，并考虑其属性内容

4. 物理建模，这部分得建模工作，主要包含以下几个部分：

* 针对特定物理化平台，做出相应的技术调整
* 针对模型的性能考虑，对特定平台作出相应的调整
* 针对管理的需要，结合特定的平台，做出相应的调整
* 生成最后的执行脚本，并完善之。

从我们上面对数据仓库的数据建模阶段的各个阶段的划分，我们能够了解到整个数据仓库建模的主要工作和工作量，希望能够对我们在实际的项目建设有所帮助

### 数据仓库建模方法论

#### 范式建模法

范式建模法其实是我们在构建数据模型常用的一个方法，该方法的主要由 Inmon 所提倡，主要解决关系型数据库得数据存储，利用的一种技术层面上的方法。目前，*我们在关系型数据库中的建模方法，大部分采用的是三范式建模法*。

范式是数据库逻辑模型设计的基本理论，一个关系模型可以从第一范式到第五范式进行无损分解，这个过程也可称为规范化。在数据仓库的模型设计中目前一般采用第三范式，它有着严格的数学定义。从其表达的含义来看，*一个符合第三范式的关系必须具有以下三个条件* :

* 每个属性值唯一，不具有多义性 ;
* 每个非主属性必须完全依赖于整个主键，而非主键的一部分 ;
* 每个非主属性不能依赖于其他关系中的属性，因为这样的话，这种属性应该归到其他关系中去。

由于范式是基于整个关系型数据库的理论基础之上发展而来的，因此，本人在这里不多做介绍，有兴趣的读者可以通过阅读相应的材料来获得这方面的知识。

根据 Inmon 的观点，数据仓库模型的建设方法和业务系统的企业数据模型类似。在业务系统中，企业数据模型决定了数据的来源，而企业数据模型也分为两个层次，即主题域模型和逻辑模型。同样，*主题域模型可以看成是业务模型的概念模型，而逻辑模型则是域模型在关系型数据库上的实例*。

从业务数据模型转向数据仓库模型时，同样也需要有数据仓库的域模型，即概念模型，同时也存在域模型的逻辑模型。这里，业务模型中的数据模型和数据仓库的模型稍微有一些不同。主要区别在于：

* 数据仓库的域模型应该包含企业数据模型的域模型之间的关系，以及各主题域定义。数据仓库的域模型的概念应该比业务系统的主题域模型范围更加广。
* 在数据仓库的逻辑模型需要从业务系统的数据模型中的逻辑模型中抽象实体，实体的属性，实体的子类，以及实体的关系等。

以笔者的观点来看，Inmon 的范式建模法的最大优点就是从关系型数据库的角度出发，结合了业务系统的数据模型，能够比较方便的实现数据仓库的建模。但其缺点也是明显的，由于建模方法限定在关系型数据库之上，在某些时候反而限制了整个数据仓库模型的灵活性，性能等，特别是考虑到数据仓库的底层数据向数据集市的数据进行汇总时，需要进行一定的变通才能满足相应的需求。因此，笔者建议读者们在实际的使用中，参考使用这一建模方式。

#### 维度建模法

*维度建模法简单描述就是按照事实表、维度表来构建数仓、集市*。维度建模从分析决策的需求出发构建模型，为分析需求服务，因此它重点关注用户如何更快速地完成需求分析，同时具有较好的大规模复杂查询的相应性能。其典型的代表是星型模型，以及在一些特殊场景下使用的雪花模型。

维度建模的优缺点：

* 优点，以星型模型为例子，其针对各个维度做了大量的预处理，如按照维度进行预先的统计、分类、排序等，从而极大的提升数据仓库的处理能力；另一个优点是维度建模非常直观，紧紧围绕着业务模型，可以直观的反映出业务模型中的业务问题。不需要经过特别的抽象处理，即可以完成维度建模。
* 缺点，由于在构建星型模式之前需要进行大量的数据预处理，因此会导致大量的数据处理工作。而且，当业务发生变化，需要重新进行维度的定义时，往往需要重新进行维度数据的预处理。而在这些预处理过程中，往往会导致大量的*数据冗余*；另外一个维度建模法的缺点就是，*如果只是依靠单纯的维度建模，不能保证数据来源的一致性和准确性，而且在数据仓库的底层，不是特别适用于维度建模的方法*。

所以维度建模的领域主要适用于数据集市层，它的最大的作用其实是为了*解决数据仓库建模中的性能问题*。维度建模很难能够提供一个完整地描述真实业务实体之间的复杂关系的抽象方法

维度建模大致分为一下几个步骤**：**

1. 选择需要进行分析决策的业务过程。业务过程可以是单个业务时间，比如交易的支付、退款等；也可以是某个事件的状态，比如当前的账户余额等；还可以是一系列相关业务时间组成的业务流程，具体需要看我们分析的是某些时间发生情况，还是当前状态或是时间流转效率
2. 选择粒度。在事件分析中，我们需要预判所有分析需求细分的程度，从而决定选择的粒度，粒度是维度的一个组合。
3. 识别维表。选择好粒度之后，就需要基于粒度设计维表，包括维度属性，用于分析时进行分组和筛选。
4. 选择事实。确定分析需要衡量的指标。

#### Data Vault建模

Data Vault 是ER模型的衍生，其设计的出发点也是*为了实现数据的整合，但不能直接用于数据分析决策*，*它强调建立一个可审计的基础数据层，也就是强调数据的历史性、可追溯性和原子性，而不要求对数据进行过度的一致性处理和整合*；同时它基于主题概念将企业数据进行结果化组织，并引入了更进一步的范式处理来优化模型以应对源系统变更的扩展性。Data Vault模型由以下几部分组成。

* Hub：是企业的核心业务实体，由实体key、数据仓库序列代理键、装载时间、数据来源组成。
* Link：代表Hub之间的关系。这里与ER模型最大的区别是将关系作为一个独立的单元抽象，可以提升模型的扩展性。它可以直接描述1:1、1:n、n:n的关系，而不需要做任何变更。它由Hub的代理键、装载时间、数据来源组成。
* Satellite：是Hub的详细描述内容，一个Hub可以有多个Satellite。它由Hub代理键、装载时间、数据来源、详细的Hub描述信息组成。

Data Vault 模型比ER模型更容易设计和产出，它的ETL加工可实现配置化。Hub可以想像成人的骨架，那么Link就是链接骨架的韧带，Satellite是骨架上的血和肉。

#### Anchor 建模

Anchor 对Data Vault模型做了进一步规范化处理，初衷是设计一个高度可扩展的模型，其*核心思想是所有的扩展只是添加而不是修改，一次将模型规范到6NF，基本变成了K-V结构化模型*。Anchor模型的组成。

* Anchors：类似于Data Vault的Hub，代表业务实体，且只有主键。
* Attributes：功能类似于Data Vault的Satellite，但是它更加规范化，将其全部K-V结构化，一个表只有一个Anchors的属性描述。
* Ties：将是Anchors之间的关系，单独用表来描述，类似于Data Vault的Link，可以提升整体模型关系的扩展能力。
* Knots：代表哪些可能会在多个Anchors中公用的属性的提炼，比如性别、状态等这种枚举类型且被公用的属性。

在上述四个基本对象的基础上，又可以细化分为历史的和非历史的，其中*历史的会以时间戳加多条记录的方式记录数据的变迁历史*。Anchor模型的创建者以此方式来获取极大的可扩展性，但是也会增加非常多的查询join操作，创建者的观点是，*数据仓库中的分析查询只是基于一小部分字段进行的，类似于列存储结构，可以大大减少数据扫描，从而对查询性能影响较小*。一些有数据表裁剪特性的数据库如MariaDB 的出现，会大量减少join操作。但实际情况是不是如此还有带商榷。

## 数据仓库建设流程

### 数据调研

### 业务调研

数据仓库是要涵盖所有业务领域，还是各个业务领域独自建设，业务领域内的业务线也同样面临着这个问题。所以要构建大数据数据仓库，就需要了解各个业务领域、业务线的业务有什么共同点和不同点，以及各个业务线可以细分为哪几个业务模块，每个业务模块具体的业务流程又是怎样的。业务调研是否充分，将会直接决定数据仓库建设是否成功。

#### 需求调研

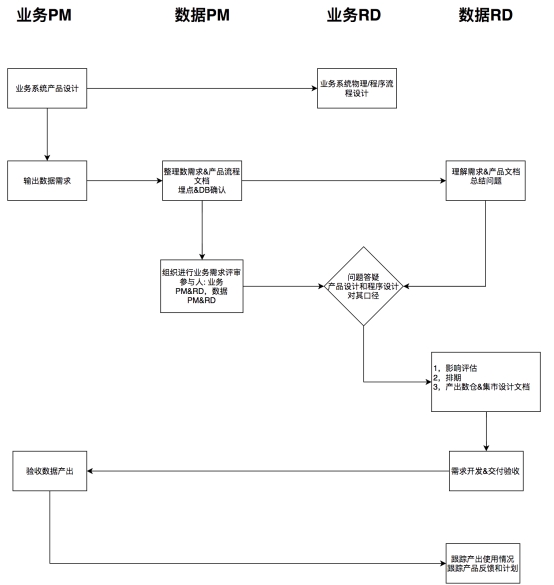
了解业务系统的业务后不等于说就可以实施数仓建设了，还需要收集数据使用者的需求，及找分析师、运营人员、产品人员等了解他们对数据的诉求。通常需求调研分下面两种途径：

1. 根据与分析师、运营人员、产品人员的沟通获取需求。
2. 对现有报表、数据进行研究分析获取数据建设需求。

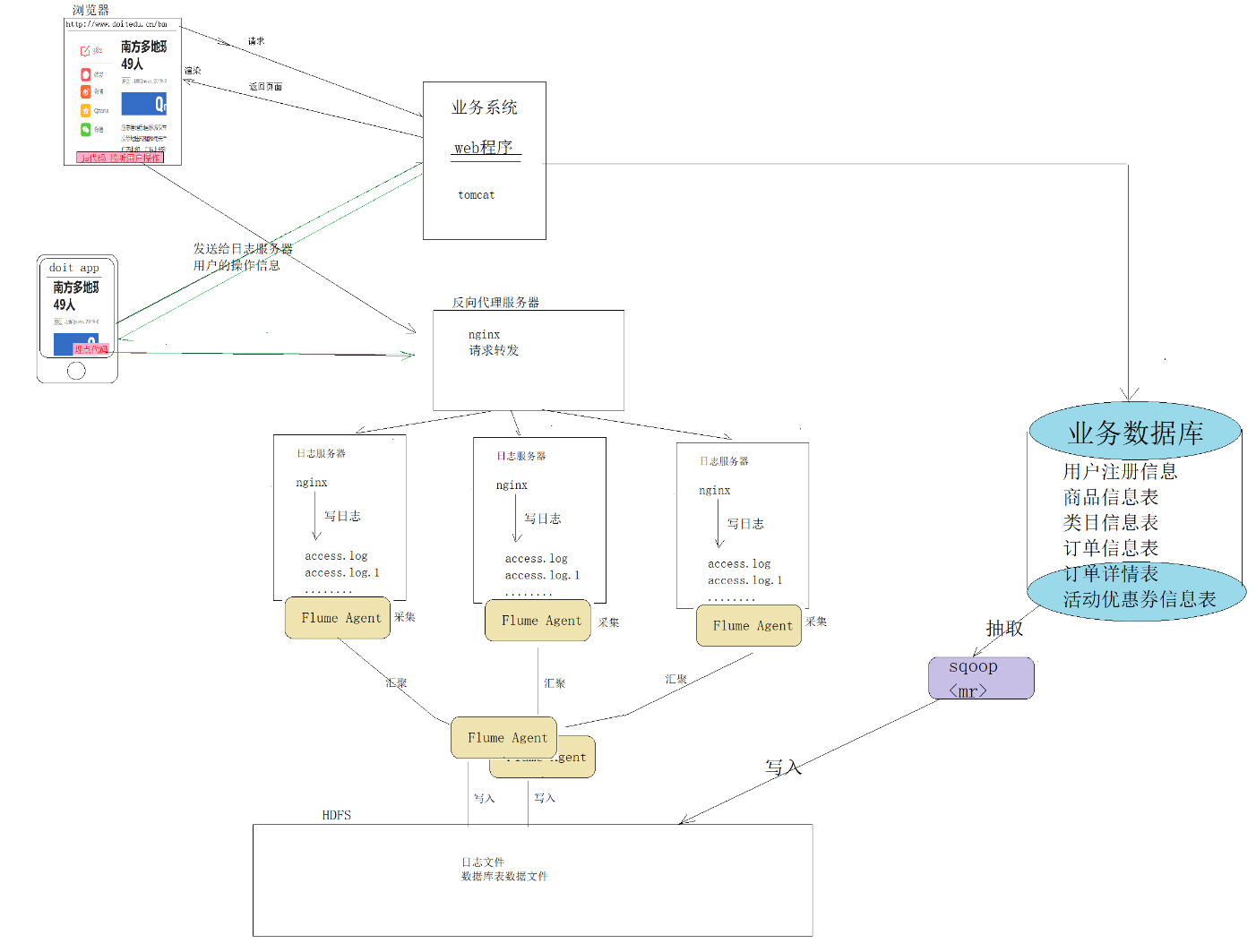
例：保险数仓需求指标：



#### 调研方法论



# 数据采集



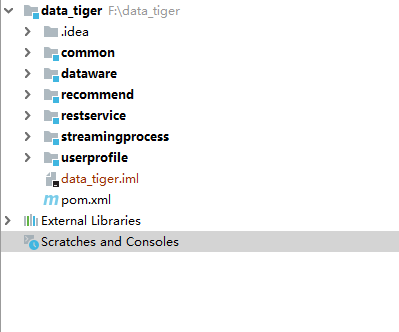
# 项目工程搭建

创建一个父工程；

父工程中引入公共的依赖和插件；

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  <**modelVersion**>4.0.0</**modelVersion**>   <**groupId**>cn.doitedu</**groupId**>  <**artifactId**>data\_tiger</**artifactId**>  <**packaging**>pom</**packaging**>  <**version**>1.0-SNAPSHOT</**version**>  <**modules**>  <**module**>dataware</**module**>  <**module**>userprofile</**module**>  <**module**>recommend</**module**>  <**module**>streamingprocess</**module**>  <**module**>common</**module**>  </**modules**>   <**dependencies**>  <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-sql\_2.11</**artifactId**>  <**version**>2.2.0</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.scala-lang</**groupId**>  <**artifactId**>scala-library</**artifactId**>  <**version**>2.11.12</**version**>  </**dependency**>   </**dependencies**>   <**pluginRepositories**>  <**pluginRepository**>  <**id**>ali-plugin</**id**>  <**url**>http://maven.aliyun.com/nexus/content/groups/public/</**url**>  <**snapshots**>  <**enabled**>true</**enabled**>  </**snapshots**>  </**pluginRepository**>  </**pluginRepositories**>    <**build**>  <**plugins**>  *<!-- 指定编译java的插件 -->* <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-compiler-plugin</**artifactId**>  <**version**>3.5.1</**version**>  <**configuration**>  <**source**>1.8</**source**>  <**target**>1.8</**target**>  </**configuration**>  </**plugin**>   *<!-- 指定编译scala的插件 -->* <**plugin**>  <**groupId**>net.alchim31.maven</**groupId**>  <**artifactId**>scala-maven-plugin</**artifactId**>  <**version**>3.2.2</**version**>  <**executions**>  <**execution**>  <**goals**>  <**goal**>compile</**goal**>  <**goal**>testCompile</**goal**>  </**goals**>  <**configuration**>  <**args**>  <**arg**>-dependencyfile</**arg**>  <**arg**>${project.build.directory}/.scala\_dependencies</**arg**>  </**args**>  </**configuration**>  </**execution**>  </**executions**>  </**plugin**>   </**plugins**>  </**build**> </**project**> |

然后在父工程中创建各个子工程；



编写demo代码，测试工程；

项目的码云git地址：

*https://gitee.com/hunter\_d/data\_tiger.git*

# 知识字典构建

## 地理位置字典构建

### 需求说明

公司通过某渠道，收集到一批地理位置信息数据，只是数据格式需要加工：



加工的结果格式要求为：

*GEOHASH码， 省，市，区，镇*

具体实现方式不限！

### 新技能：GeoHash编码

直接去匹配两个哪怕距离很近的gps坐标，很可能都匹配不上！

gps坐标的匹配，不应该做这种精确匹配，应该做范围匹配；

在geo领域，gps坐标，可以转化成一种编码：GeoHash编码（如wx3y），这种码有一个特点：

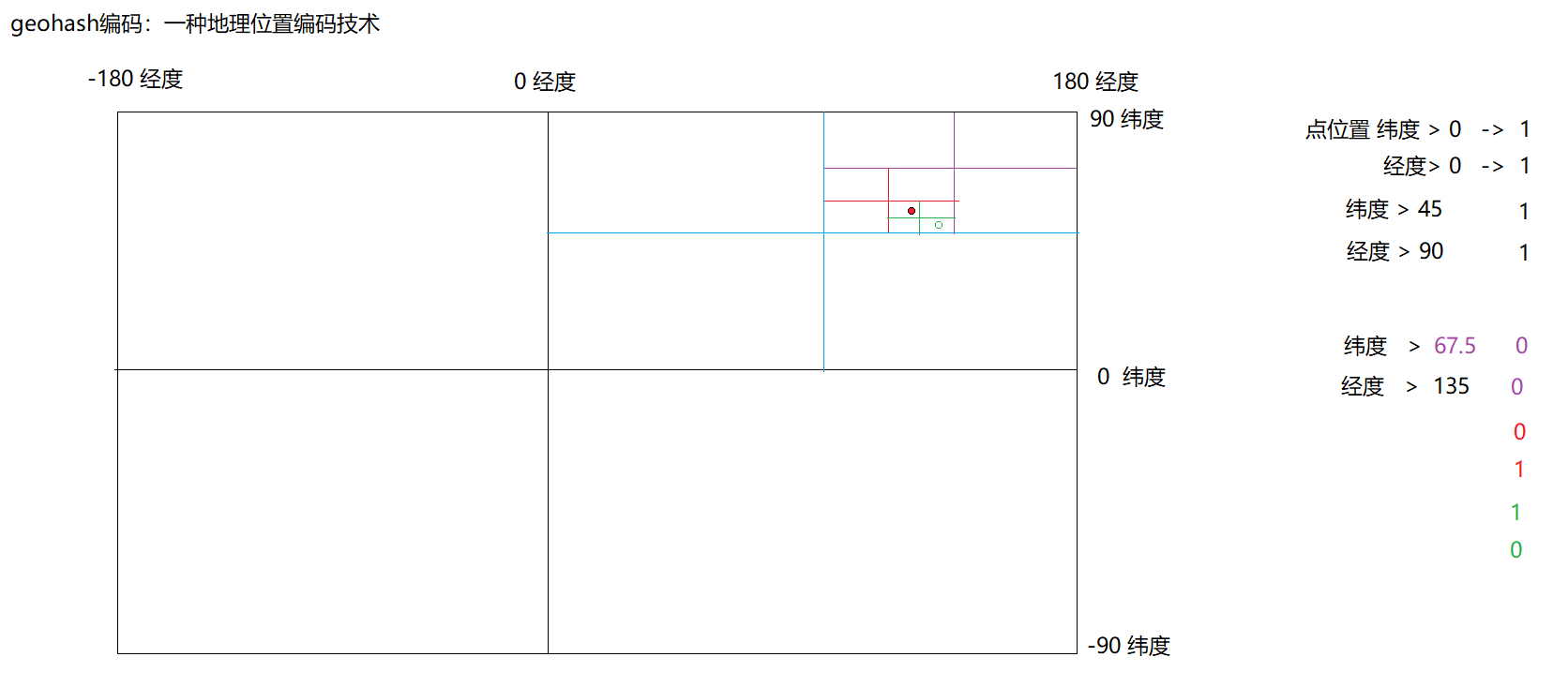
两个gps坐标的位置越近，这这两个坐标的geohash码的相同的位数更多，只有更少的几位尾数不同！如下所示：

wx3y569

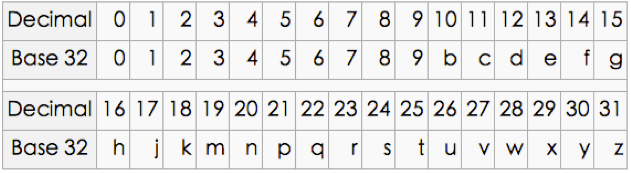
wx3y434

GeoHash编码的基本原理：

不断地将地球的经度、纬度范围，**进行二分查找的方法**，输出1/0比特，形成一串二进制码（二分查找的次数越多，输出的bit串越长），

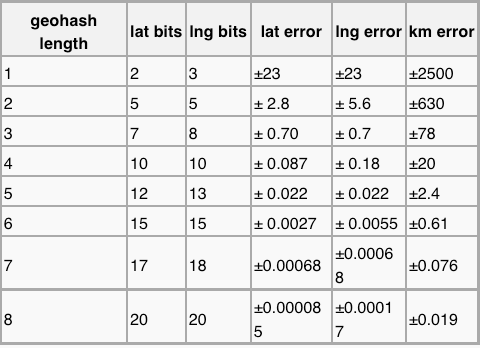


然后将这一串二进制码，按照5bit一组合查**base32码表**，



输出最终结果！

geohash码的精确度对应表格：



*gps 转码成 geohash编码，这个算法不需要自己手写，有现成的工具包*

maven依赖坐标：

|  |
| --- |
| <**dependency**>  <**groupId**>ch.hsr</**groupId**>  <**artifactId**>geohash</**artifactId**>  <**version**>1.3.0</**version**> </**dependency**> |

api调用示例：

|  |
| --- |
| String geohashcode = GeoHash.*withCharacterPrecision*(45.667, 160.876547, 8).toBase32(); |

### 地理位置字典构建代码实现

#### 需求说明

原材料是一种关系型表的层级结构

*id 地点名称 父id 短名称 行政级别*



实现方式有两种

#### 实现方式（一）

直接用spark读原始文件，进行扁平化运算及geohash编码

#### 实现方式（二）

在mysql上用sql语句查询来实现扁平化 ；

将地理位置数据导入mysql，然后在mysql中形成扁平化结构的结果

地理位置原始数据导入mysql，建表语句：

|  |
| --- |
| *SET FOREIGN\_KEY\_CHECKS=0;*  *-- ----------------------------*  *-- Table structure for area*  *-- ----------------------------*  *DROP TABLE IF EXISTS `area`;*  *CREATE TABLE `area` (*  *`lng` double DEFAULT NULL,*  *`lat` double DEFAULT NULL,*  *`province` mediumtext NOT NULL,*  *`city` mediumtext NOT NULL,*  *`district` mediumtext NOT NULL,*  *`county` mediumtext NOT NULL*  *) ENGINE=InnoDB DEFAULT CHARSET=utf8;* |

查询sql语句：

|  |
| --- |
| *SELECT*  *a.BD09\_LNG,*  *a.BD09\_LAT,*  *d.areaname AS province,*  *c.areaname AS city,*  *b.areaname AS district*  *FROM*  *t\_md\_areas a*  *JOIN t\_md\_areas b ON a. LEVEL = 4 AND a.PARENTID = b.id*  *JOIN t\_md\_areas c ON b.parentid = c.id*  *JOIN t\_md\_areas d ON c.parentid = d.id* |

得到扁平化的表结构：



然后用spark去读mysql中的这个扁平化结果，并将其中的经纬度**换成geohash码**

## 商圈字典构建代码实现

|  |
| --- |
| **package** cn.doitedu.dicts  **import** ch.hsr.geohash.GeoHash **import** cn.doitedu.common.util.SparkUtil **import** org.apache.spark.sql.SparkSession  */\*\*  \** **@date:** *2019/7/23  \** **@site:** *www.doitedu.cn  \** **@author:** *hunter.d 涛哥  \** **@qq:** *657270652  \** **@description:** *商圈位置信息字典构建程序  \*/* **object** BizDict {    **def** main(args: Array[String]): Unit = {   **val** spark = SparkUtil.*getSparkSession*(**"biz"**)  **import** spark.implicits.\_    **val** ds = spark.read.textFile(**"data\_warehouse/data/biz\_origin"**)   ds.map(\_.split(**","**))  .filter(\_.size > 5)  .map(arr => {  **var** province = arr(1)  **var** city = arr(2)  **var** district = arr(3)  **var** biz = arr(4)  **var** lng = arr(5)  **var** lat = arr(6)   province = province.trim.substring(1, province.length - 2)  city = city.trim.substring(1, city.length - 2)  district = district.trim.substring(1, district.length - 2)  biz = biz.trim.substring(1, biz.length - 2)  lng = lng.trim.substring(1, lng.length - 2)  lat = lat.trim.substring(1, lat.length - 4)    **var** geo: String = **""  try** {  geo = GeoHash.*withCharacterPrecision*(lat.toDouble, lng.toDouble, 6).toBase32  } **catch** {  **case** e: Exception =>  }   (geo, province, city, district, biz)  })  .filter(!\_.\_1.equals(**""**))  .toDF(**"geo"**, **"province"**, **"city"**, **"district"**, **"biz"**)  .write  .parquet(**"data\_warehouse/data/biz\_dict"**)  spark.close()  } } |

## 高德地理位置加强

## [用户画像]app信息字典

## [用户画像]统一身份识别id映射字典

## [用户画像]url页面内容信息字典

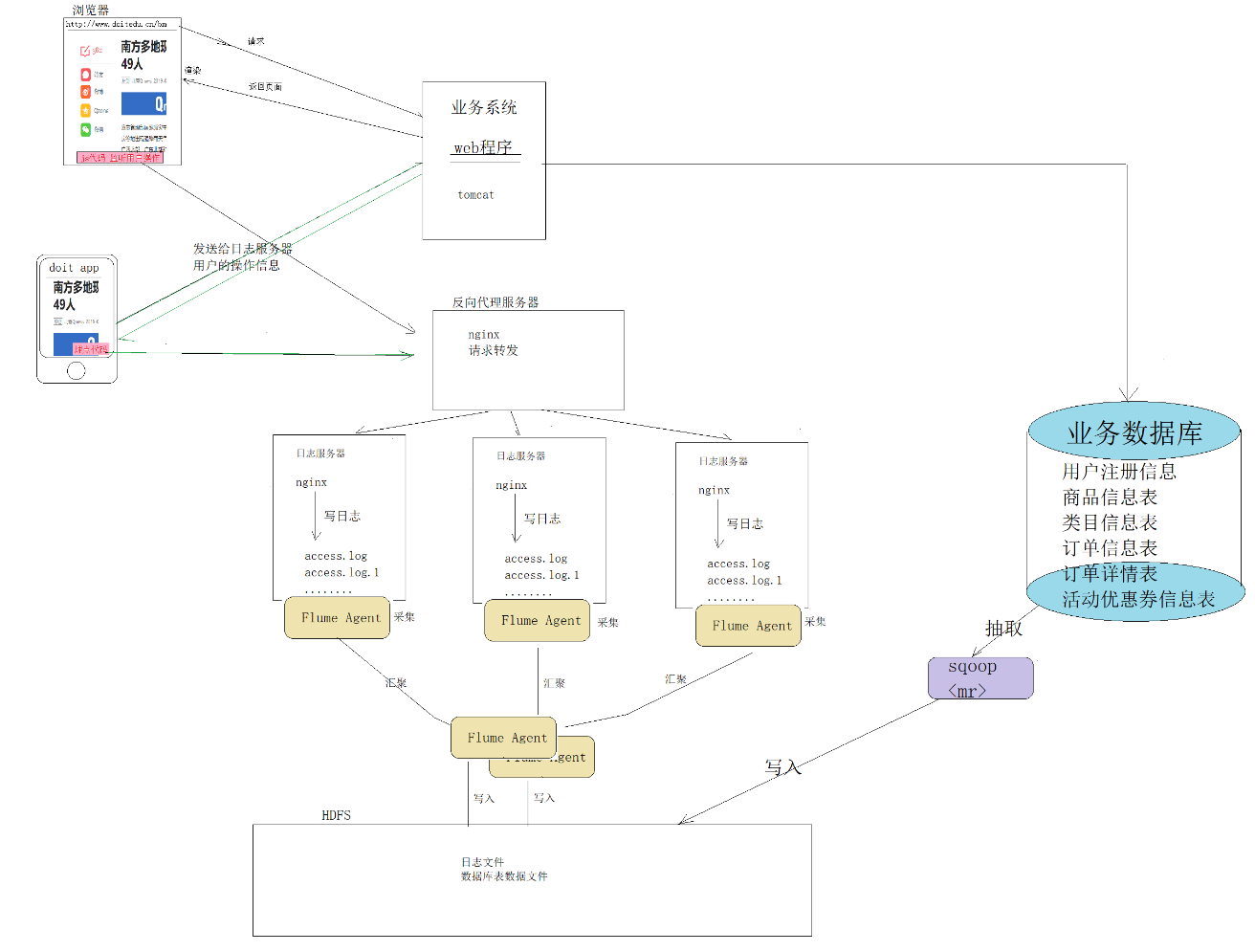
# 数据采集

## 数据来源

1） 用户访问行为日志（流量数据）

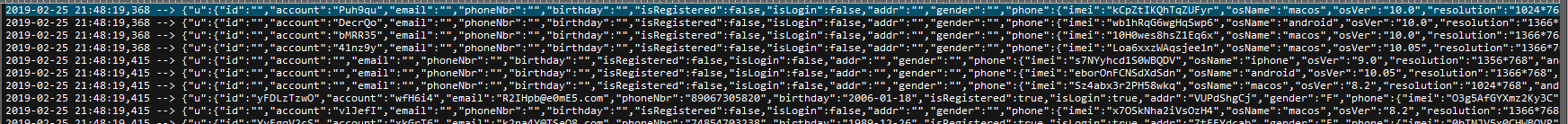
2） 业务系统数据库数据

## 采集系统工作流程



# 公司流量日志说明

## 整体结构



其中主要包含两大块信息：

* 用户终端属性信息；
* 事件信息；

其中主体结构为一个json字串，样例如下：

|  |
| --- |
| *{*  *"u": { // 用户信息*  *"id": "mwbqapodT", // 用户的cookieid*  *"account": "aq0BkE", // 用户账号*  *"email": ["26XWSp@8creA.com",](mailto:%2226XWSp@8creA.com%22,) // 用户email*  *"phoneNbr": "28147302391", // 用户手机号*  *"birthday": "1994-06-5", // 用户生日*  *"isRegistered": true, // 是否注册用户*  *"isLogin": true, // 是否登录用户*  *"addr": "vsmukPFiT", // 用户住址*  *"gender": "F", // 性别*  *"phone": { // 用户的手机信息*  *"imei": "EJl3FhXUgUjspUL9", // 手机IMEI码*  *"osName": "android", // 手机操作系统名称*  *"osVer": "10.0", // 手机操作系统版本*  *"resolution": "1024\*768", // 手机分辨率*  *"androidId": "cFJx5Q7NuDTI", // 安卓系统id*  *"manufacture": "华为", // 安卓品牌制造商*  *"deviceId": "" //设备id*  *},*  *"app": { // 用户的app信息*  *"appid": "com.51doit.mall", // app id*  *"appVer": "1.9.0", // app 版本*  *"release\_ch": "天翼空间", // app下载渠道*  *"promotion\_ch": "抖音" // app推广渠道*  *},*  *"loc": { // 用户的位置信息*  *"areacode": 370784111, // 位置地域编码*  *"longtitude": 119.08123855438232, // 经度*  *"latitude": 36.15410449093023, // 纬度*  *"carrier": "爱奇艺卡", // 手机sim卡运营商*  *"netType": "4G", // 联网类型*  *"cid\_sn": "YClBRcbhXDoH", // 基站编码*  *"ip": "64.89.246.13" // ip 地址*  *},*  *"sessionId": "uw9f8uMzEfme" // 本次连接的会话id*  *},*  *"eventType": "favor", // 日志类型： favor是收藏*  *"commit\_time": 1551102499420, //日志提交时间*  *"event": { // 用户行为事件详细信息*  *"skuid": "50847610" // 所收藏的商品id*  *}*  *}* |

## 事件类型说明

关于事件类型和event字段补充说明如下

|  |  |  |
| --- | --- | --- |
| 编号 | 事件名称 | 说明 |
| 1 | startup | 启动事件 |
| 2 | favor | 收藏事件 |
| 3 | rate | 评分事件 |
| 4 | add\_cart | 添加item到购物车事件 |
| 5 | thumbup | 点赞事件 |
| 6 | ad\_click | 点击广告事件 |
| 7 | ad\_show | 广告展示曝光事件 |
| 8 | search | 站内搜索事件 |
| 9 | pg\_view | 页面浏览事件 |
| 10 | rec\_click | 推荐项点击事件 “推荐栏目id，商品id，事件所发生的页面” |
| 11 | sub\_order | 提交订单事件 |

事件详情信息说明：

|  |
| --- |
| **pg\_view**事件 "event":{  "pgid":"30-03-258413" // 所浏览的页面的id  "url":"/abc/edh/30-03-258413.html?a=x&b=y",  "title":"Xiaomi/小米 小米8SE 前置2000万柔光自拍 超感光双摄",  "skuid":"3985910262736", // 商品id sku->spu  "trace\_id":"uuid092304ksdl0wei", // 广告跟踪码  }  **ad\_show** 事件  "event":{  "adid":"10-03-01" // ad\_id  "url":"/abc/edh/a.html?a=x&b=y",  "skuid":"3985910262736", // 商品id  }  **ad\_click** 事件  "event":{  "adid":"10-03-01" // ad\_id  "url":"/abc/edh/a.html?a=x&b=y",  "trace\_id":"uuid092304ksdl0wei", // 广告跟踪码  "skuid":"3985910262736", // 商品id  } |

*注意： 不同事件日志的event字段，内容和结构均有不同！*

## 字段类型说明

commit\_time 为Long

Longtitude： 为Double

Latitude：为Double

其他字段原始类型皆为String

# 【流量数据】数据预处理

## 需求说明

### 清洗过滤

去除日志头部的时间戳，只保留json体

去除json数据体中的以下字段（这是前端开发人员在埋点设计方案变更后遗留的无用字段）：

|  |
| --- |
| *"email"*  *"phoneNbr"*  *"birthday"*  *"isRegistered"*  *"isLogin"*  *"addr"*  *"gender"* |

过滤掉日志中： account | cookieid | sessionid | imei |deviceId |androidId | IP全为空的记录！

过滤掉日志中缺少关键字段（event）的记录！

过滤掉json格式不完整的（脏数据）！

### 数据解析

将json打平： 解析成扁平格式(CSV格式）；

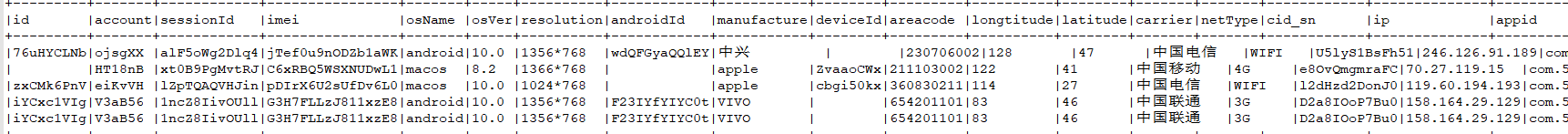


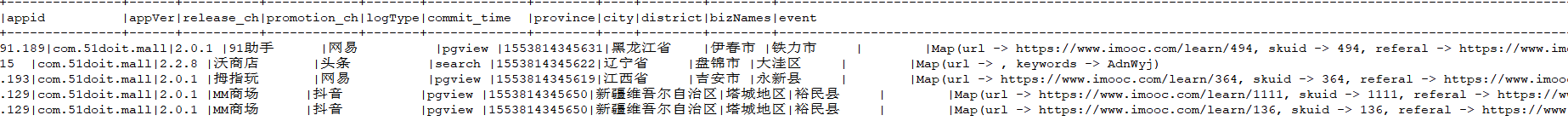
event还是保留原来结构（HashMap<String,String>），不用打平

### 数据集成

* 将日志中的**经纬度**解析成省、市、区（县）信息；（为了方便后续的地域维度分析）
* 集成商圈信息；（为了方便后续的地域维度分析）
* 将页面标题进行分词，并过滤单字词；（为了方便后续的用户兴趣标签分析）

最后，将数据输出为**parquet格式（因为要放入hive）**

---->

----> 

**补充需求：**

如果某条日志中的gps地理位置无法从公司内部字典中匹配信息，则将这些gps坐标，输出到一个专门的文件中；

## 新技能：高德地图服务

### 导论

由于公司自己的地理位置知识库是不完备的，每日的流量数据预处理中，有一些gps坐标是解析不出地理位置信息的；

我们可以将这些解析不出来的gps坐标，单独过滤出来，然后通过请求高德地图服务来解析，并将解析结果追加到我们的地理位置知识库中，知识库就可以日益充实和完善！

高德地图，提供各类地理位置处理功能的web服务接口！

我们的流量数据中，有一些gps坐标无法从公司内部的地理位置字典中查询到位置信息，就可以把这些gps拿去请求高德的web服务，获取地理位置信息，并丰富公司内部的地理位置知识字典！

### 高德地图服务申请

高德开发者开放平台网址webAPI：<https://lbs.amap.com/>



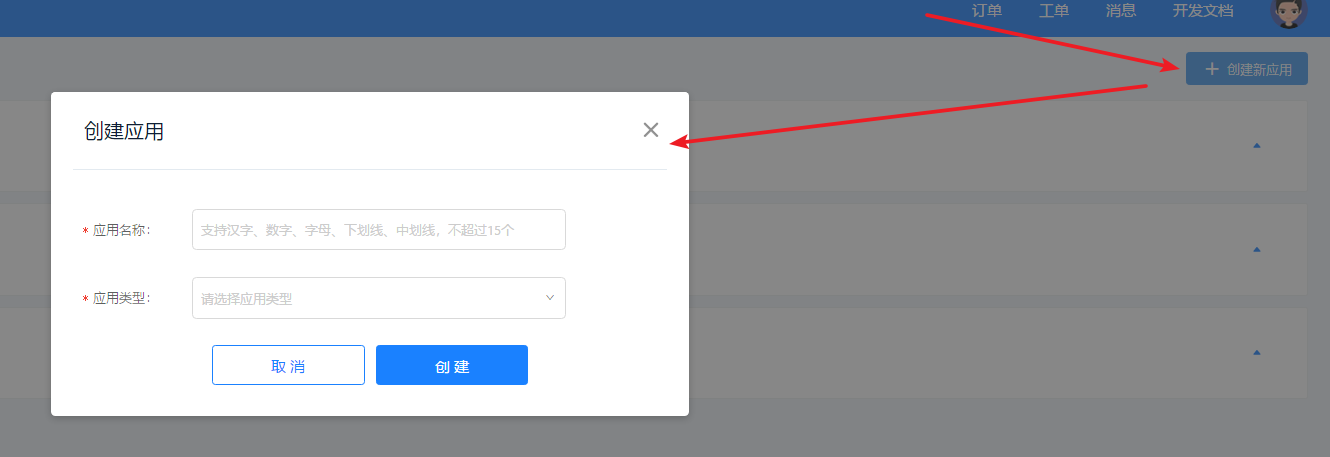
开发者接口文档入口：选Web服务API



申请应用Key



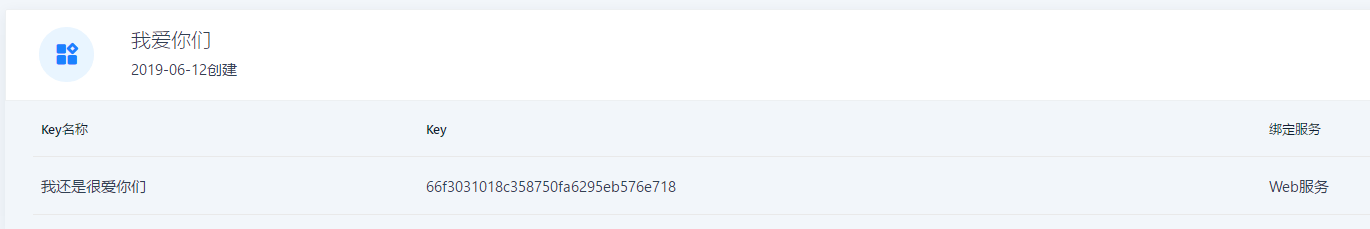
从控制台，新建应用



添加key给应用







### 高德地图服务API

API文档地址： <https://lbs.amap.com/api/webservice/summary/>



返回结果解析说明



|  |
| --- |
| {  "status": "1",  "regeocode": {  "addressComponent": {  "city": [],  "province": "北京市",  "adcode": "110105",  "district": "朝阳区",  "towncode": "110105026000",  "streetNumber": {  "number": "6号",  "location": "116.481977,39.9900489",  "direction": "东南",  "distance": "62.165",  "street": "阜通东大街"  },  "country": "中国",  "township": "望京街道",  "businessAreas": [  {  "location": "116.470293,39.996171",  "name": "望京",  "id": "110105"  },  {  "location": "116.494356,39.971563",  "name": "酒仙桥",  "id": "110105"  },  {  "location": "116.492891,39.981321",  "name": "大山子",  "id": "110105"  }  ],  "building": {  "name": "方恒国际中心B座",  "type": "商务住宅;楼宇;商务写字楼"  },  "neighborhood": {  "name": "方恒国际中心",  "type": "商务住宅;楼宇;商住两用楼宇"  },  "citycode": "010"  },  "formatted\_address": "北京市朝阳区望京街道方恒国际中心B座方恒国际中心"  },  "info": "OK",  "infocode": "10000"  } |

### 高德API请求demo

|  |
| --- |
| **object** GaodeLBSDemo {  **def** main(args: Array[String]): Unit = {  **import** scala.collection.JavaConversions.\_  *// 构造一个http客户端* **val** client = HttpClientBuilder.*create*().build()  *// 构造也给get协议请求* **val** get = **new** HttpGet(**"https://restapi.amap.com/v3/geocode/regeo?key=66f3031018c358750fa6295eb576e718&location=116.481488,39.990464"**) *// 用客户端去执行请求* **val** response = client.execute(get)  **val** entity = response.getEntity  **val** in = entity.getContent  **val** lines: util.List[\_] = IOUtils.*readLines*(in)  *// 后续* ***TODO 解析响应回来的json，取出省、市、区、街道、商圈*** } } |

## 预处理开发实现

### 整体流程

1）json解析，解析成功的返回LogBean对象，解析失败的返回null

（这样一来，json格式不对、不完整的脏数据就被识别出来了）

2）对上一步结果RDD[LogBean]进行过滤（清掉json不完整的脏数据，清掉不符合规则的数据）

3）对数据进行字典知识集成

4）从集成后的结果中跳出无法解析的gps，写入一个待解析目录

5）输出最终结果保存为parquet文件

### 完整代码

case class定义：

|  |
| --- |
| **package cn.doitedu.preprocess case class EventBean(  cookieid: String,  account: String,  imei: String,  osName: String,  osVer: String,  resolution: String,  androidId: String,  manufacture: String,  deviceId: String,  appid: String,  appVer: String,  release\_ch: String,  promotion\_ch: String,  areacode: String,  longtitude: Double,  latitude: Double,  carrier: String,  netType: String,  sessionId: String,  eventType: String,  commit\_time: Long,  var event:Map[String, String],  var province:String = "",  var city:String = "",  var district:String = "",  var biz:String = ""  )** |

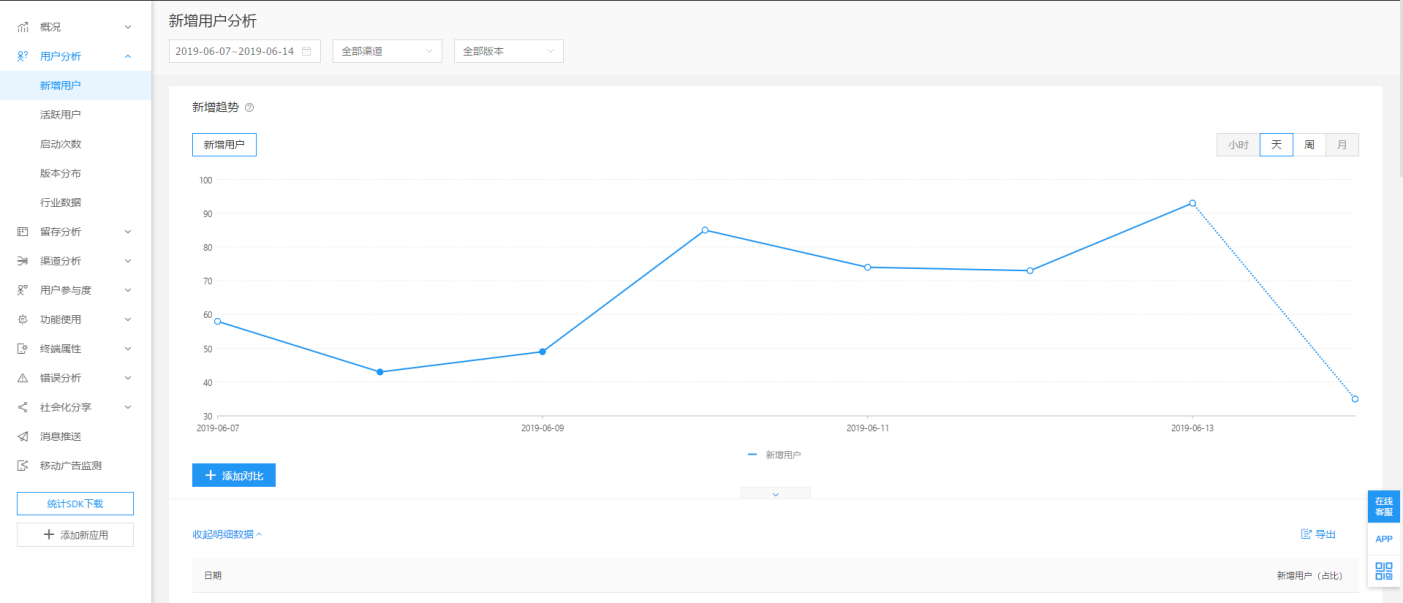
预处理流程如下：

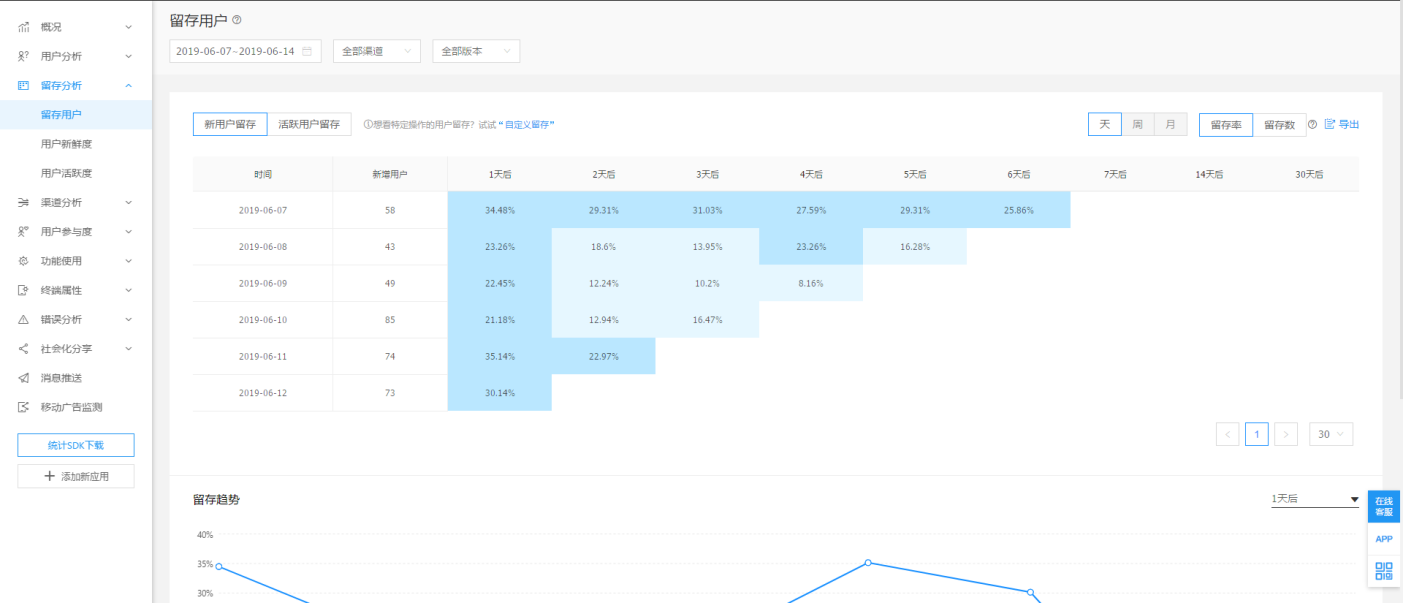
|  |
| --- |
| **package** cn.doitedu.preprocess  **import** java.util  **import** ch.hsr.geohash.GeoHash **import** cn.doitedu.common.util.SparkUtil **import** com.alibaba.fastjson.{JSON, TypeReference} **import** com.hankcs.hanlp.HanLP **import** com.hankcs.hanlp.seg.common.Term **import** org.apache.commons.lang3.StringUtils **import** org.apache.spark.sql.Dataset  **import** scala.collection.{JavaConverters, mutable}   */\*\*  \** **@date:** *2019/7/23  \** **@site:** *www.doitedu.cn  \** **@author:** *hunter.d 涛哥  \** **@qq:** *657270652  \** **@description:** *流量日志预处理程序  \*/* **object** EventLogPre {  **def** main(args: Array[String]): Unit = {   **val** spark = SparkUtil.*getSparkSession*(**"event\_preprocess"**)  **import** spark.implicits.\_   *// 读取原始日志  // spark、mapreduce都能直接读取一些常见类型的压缩文件/gz/bz2/deflate/snappy....* **val** ds = spark.read.textFile(**"data\_warehouse/data/event\_logs/2019-06-15"**)    *// 切割，提取json* **val** json: Dataset[String] = ds.map(x => x.split(**" --> "**)(1))    *// 解析、过滤* **val** filteredDs: Dataset[EventBean] = json.map(x => {   **var** bean: EventBean = **null**;   **try** {  *// 解析json* **val** jSONObject = JSON.*parseObject*(x)  **val** uobj = jSONObject.getJSONObject(**"u"**)  **val** cookieid = uobj.getString(**"cookieid"**)  **val** account = uobj.getString(**"account"**)   **val** phoneObj = uobj.getJSONObject(**"phone"**)   **val** imei = phoneObj.getString(**"imei"**)  **val** osName = phoneObj.getString(**"osName"**)  **val** osVer = phoneObj.getString(**"osVer"**)  **val** resolution = phoneObj.getString(**"resolution"**)  **val** androidId = phoneObj.getString(**"androidId"**)  **val** manufacture = phoneObj.getString(**"manufacture"**)  **val** deviceId = phoneObj.getString(**"deviceId"**)   **val** appObj = uobj.getJSONObject(**"app"**)  **val** appid = appObj.getString(**"appid"**)  **val** appVer = appObj.getString(**"appVer"**)  **val** release\_ch = appObj.getString(**"release\_ch"**)  **val** promotion\_ch = appObj.getString(**"promotion\_ch"**)   **val** locObj = uobj.getJSONObject(**"loc"**)  **val** areacode = locObj.getString(**"areacode"**)  **val** longtitude = locObj.getDouble(**"longtitude"**)  **val** latitude = locObj.getDouble(**"latitude"**)  **val** carrier = locObj.getString(**"carrier"**)  **val** netType = locObj.getString(**"netType"**)   **val** sessionId = uobj.getString(**"sessionId"**)   **val** eventType = jSONObject.getString(**"logType"**)  **val** commit\_time = jSONObject.getLong(**"commit\_time"**)   **val** event = jSONObject.getJSONObject(**"event"**)   **import** scala.collection.JavaConversions.\_  **val** eventMap: util.Map[String, String] = event.getInnerMap  **val** eventMapScala = eventMap.toMap    */\*val keys = event.keySet()  val eventMap: Map[String, String] = keys.map(key => (key, event.getString(key))).toMap\*/* bean = *EventBean*(  cookieid,  account,  imei,  osName,  osVer,  resolution,  androidId,  manufacture,  deviceId,  appid,  appVer,  release\_ch,  promotion\_ch,  areacode,  longtitude,  latitude,  carrier,  netType,  sessionId,  eventType,  commit\_time,  eventMapScala  )   } **catch** {  **case** e: Exception => e.printStackTrace()  }  bean  })  .filter(bean => {  *// 这几个标识字段不能全为空* **val** account = bean.account  **val** androidId = bean.androidId  **val** cookieid = bean.cookieid  **val** deviceId = bean.deviceId  **val** imei = bean.imei  **val** sessionId = bean.sessionId   **val** sb = **new** StringBuilder  **val** pinjie = sb.append(account).append(androidId).append(cookieid).append(imei).append(sessionId).append(deviceId)  **val** str = pinjie.replaceAllLiterally(**"null"**, **""**)   StringUtils.*isNotBlank*(str)  })    *// 读取地理位置字典、商圈字典* **val** area = spark.read.parquet(**"data\_warehouse/data/area\_dict"**)   *// 将地理位置数据收集到driver端* **val** areaMap: collection.Map[String, Array[String]] = area.map(row => {  *// "geo","province","city","district"* **val** geo = row.getAs[String](**"geo"**)  **val** province = row.getAs[String](**"province"**)  **val** city = row.getAs[String](**"city"**)  **val** district = row.getAs[String](**"district"**)  (geo, *Array*(province, city, district))  }).*rdd*.collectAsMap()   **val** bc1 = spark.sparkContext.broadcast(areaMap)    **val** biz = spark.read.parquet(**"data\_warehouse/data/biz\_dict"**)  *// 将商圈信息数据收集到driver端* **val** bizMap: collection.Map[String, Array[String]] = biz.map(row => {  *// "geo","province","city","district","biz"* **val** geo = row.getAs[String](**"geo"**)  **val** province = row.getAs[String](**"province"**)  **val** city = row.getAs[String](**"city"**)  **val** district = row.getAs[String](**"district"**)  **val** biz = row.getAs[String](**"biz"**)  (geo, *Array*(province, city, district, biz))  }).*rdd*.collectAsMap()  **val** bc2 = spark.sparkContext.broadcast(bizMap)   *// 集成地理位置信息* **val** integrated: Dataset[EventBean] = filteredDs.map(bean => {  **val** lng = bean.longtitude  **val** lat = bean.latitude  **val** geohash = GeoHash.*withCharacterPrecision*(lat, lng, 6).toBase32   *// 从广播变量中获取字典* **val** areaDict = bc1.value  **val** bizDict = bc2.value   **var** province: String = **""  var** city: String = **""  var** district: String = **""  var** biz: String = **""   var** bizInfo: Array[String] = **null** bizInfo = bizDict.getOrElse(geohash, *Array*(**""**, **""**, **""**, **""**))  province = bizInfo(0)  city = bizInfo(1)  district = bizInfo(2)  biz = bizInfo(3)   **var** areaInfo: Array[String] = **null  if** (bizInfo(0).equals(**""**)) {  areaInfo = areaDict.getOrElse(geohash.substring(0, 5), Array(**""**, **""**, **""**))   province = areaInfo(0)  city = areaInfo(1)  district = areaInfo(2)  }   bean.province = province  bean.city = city  bean.district = district  bean.biz = biz   bean  })   *// 没有匹配上地理位置的那些gps坐标，需要另行输出* integrated  .filter(bean => StringUtils.isBlank(bean.province))  .map(bean => (bean.longtitude, bean.latitude))  .toDF(**"lng"**, **"lat"**)  .selectExpr(**"concat\_ws(',',lng,lat)"**)  .write  .text(**"data\_warehouse/data/gps\_to\_parse"**)   *// 加载停止词词典 TODO* **val** stpwdSet = spark.read.textFile(**"data\_warehouse/data/stopwords"**).rdd.collect().toSet  **val** bc3 = spark.sparkContext.broadcast(stpwdSet)   *// 标题分词* **val** result = integrated.map(bean => {  **val** maybeTitle = bean.event.getOrElse(**"title"**, **null**)   **var** keywords: String = **""  if** (maybeTitle != **null**) {  **val** terms: util.List[Term] = HanLP.segment(maybeTitle)  *// TODO 过滤停止词的具体位置* **import** scala.collection.JavaConversions.\_  **val** stpwd = bc3.value  keywords = terms.map(term => term.word).filter(word => {  word.size > 1 && (!stpwd.contains(word))  }).mkString(**" "**)   **val** kwEventMap = bean.event.+(**"title\_kwds"** -> keywords)  bean.event = kwEventMap  }   bean  }).toDF()   result  .coalesce(1)  .write  .parquet(**"data\_warehouse/data/out\_eventlog\_preprocess/2019-06-15"**)   spark.close()  } } |

# 【流量数据】数仓需求分析与建模

## 数据可视化平台原型页面

公司很可能在一开始的时候，按照领导、产品经理的想法，设计了一套数据可视化的原型页面，如下：





然后，就把这一套原型页面交给某个数据开发人员，让其建设公司的数仓！

## 框架性思路（主题建模）

面对这样一个情况，该从何处下手？

1）深刻理解公司的业务背景

2）梳理报表需求（梳理出主题）

3）梳理原始数据

然后从某个主题出发，开始设计模型，并做ETL开发

|  |  |
| --- | --- |
| 小  提  示 | ETL： extract transform load  数据的抽取、转换、加载 |

## 分层模型

更具体的，则是开始设计分层模型，可详细参悟如下案例：

1) 考虑下面两个报表：

|  |  |  |  |
| --- | --- | --- | --- |
| PV总数 | 访客数UV | 总访问次数 | 平均每次访问时长 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 访客数UV | 人均访问次数 | 人均访问深度 | 人均访问时长 | 回头客占比 |

2) 分层思想

可以发现，上述报表中的各个数据指标，都可以归类为“流量”类指标；

而其中，诸如总访问次数、平均每次访问时长、人均访问次数，人均访问时长，回头客占比等指标，可以想象的出来，在计算过程中，拥有一些共同的中间状态；

比如，如果有一个这样的中间表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 日期 | 用户id | 会话id | 会话时长 | 会话访问深度 |

则，上述两个报表中的指标，都可以比较方便地计算出来；

# 【流量数据】ODS层流量数据加载

## 建模

ODS层：操作数据层（原始数据层）

这一层的**建模原则**是：模型结构应该**跟导入数据的结构保持一致**

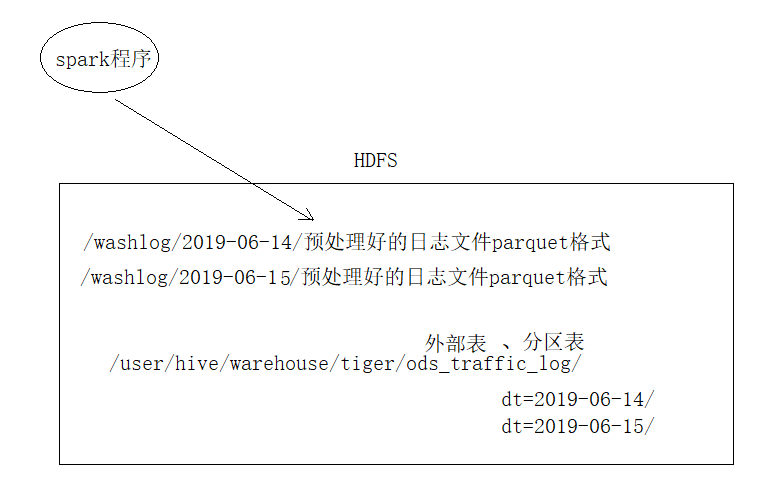
而在具体流程中，我们的“原始数据”指的就是经过spark程序预处理完成之后的数据

但是，

1）考虑到ods层对接的数据是hive数仓外部的数据（spark预处理程序所输出的数据）

我们应该把ods层的表定义为**外部表**！

2）考虑到流量数据是会每天新增的，为了将来的计算效率，ods层的流量数据表还应该定义为**分区表**！



## 模型，表结构

|  |
| --- |
| create external table ods\_traffic\_log(  cookieid string,  account string,  imei string,  osName string,  osVer string,  resolution string,  androidId string,  manufacture string,  deviceId string,  appid string,  appVer string,  release\_ch string,  promotion\_ch string,  areacode string,  longtitude double,  latitude double,  carrier string,  netType string,  cid\_sn string,  ip string,  sessionId string,  logType string,  commit\_time bigint,  province string,  city string,  district string,  county string,  biz array<string>,  eventMap map<string,string>  )  partitioned by (dt string)  stored as parquet  location '/washlog/'  ; |

## 加载数据

|  |  |
| --- | --- |
| 小提示 | HiveServer启动时可以选择启动为前台模式，也可以启动为后台模式：  如何将一个程序启动为后台模式？  示例如下：  nohup bin/hiveserver2 1>/dev/null 2>&1 & |

* **步骤：**

1） 启动hiveserver2

2） 启动hiveserver的客户端程序beeline，来连接hive服务

*[root@c701 hive-3.1.1]# bin/beeline -u jdbc:hive2://localhost:10000 -n root*

3） 新建一个数仓的库tiger

*0: jdbc:hive2://localhost:10000> create database tiger;*

*0: jdbc:hive2://localhost:10000> use tiger;*

4) 创建ods层的流量数据表（调用上一节建模中的建表语句）

5）加载预处理后的日志到 表： *ods\_traffic\_log*

a) 在hdfs上创建预处理结果数据目录

*[root@c703 ~]# hdfs dfs -mkdir -p /washlog/2019-06-13/*

b）从windows上将预处理测试结果文件上传到linux（alt+p打开sftp，或者rz，上传即可）

c）将测试结果文件上传到hdfs

*[root@c703 ~]# hdfs dfs -put part-00000-3afd3170-0129-401e-8351-b5144306a73c-c000.snappy.parquet /washlog/2019-06-13/*

d） load数据到hive表

此处，*ods\_traffic\_log表已经是一个外部表，而且已经指向了hdfs的/washlog目录，而要将/washlog下的数据导入ods\_traffic\_log表，其实是将 /washlog下的一个****天目录****映射到ods\_traffic\_log表的一个****天分区****！*

*有两种方式都可以：*

load data inpath '/washlog/2019-06-13' into table ods\_traffic\_log partition(dt='2019-06-13');

或者：

alter table ods\_traffic\_log add partition(dt='2019-06-13') location '/washlog/2019-06-13';

e）加载进去之后，可以用如下语法来查询表中是否有新增的分区：

show partitions ods\_traffic\_log;

select \* from ods\_traffic\_log limit 10;

# 【流量数据】流量概况分析

## DWD层建模及开发

### 建模

**设计1：**

由于本需求是流量主题指标计算，并不需要具体的事件详情信息，所以在DWD层只**留下流量主题计算所需要的明细字段**即可，当然，也可以把一些“维度信息”综合进来（也可以专门放在维度表）：

1.所访问的页面所属的栏目

2.将时间戳提取出 年，月，日，季度，促销季，某活动期间，双十一期间，周年庆，某节日

**举例1：事实表和维度表分离：**

日志数据： imei001,pg\_view,commit\_time,url2

时间维度表：

2019-06-18,10年店庆

2019-11-11,双十一促销

url栏目维度表：

url1，栏目A

url2，栏目B

**举例2：事实表维度表结合：**

imei001,pg\_view,commit\_time,url2,双十一促销,栏目B

优点：降低计算复杂度，减少了join

缺点：灵活性不够！

**设计2：**

由于在后续的报表统计需求中，需要区分每一个访客，所以，需要在DWD层的流量表中用一个规则来指定每个访客的**独特标识**（优选account，次之imei码，再次之androidid，再次之deviceid，再次之ip）

建模语句：

|  |
| --- |
| *drop table if exists yiee.dwd\_traffic\_dtl;*  *create external table if not exists yiee.dwd\_traffic\_dtl;*  *(*  *uid string COMMENT '用户唯一标识',*  *osName string ,*  *osVer string ,*  *resolution string ,*  *manufacture string ,*  *appid string ,*  *appVer string ,*  *release\_ch string ,*  *promotion\_ch string ,*  *carrier string , -- 运营商*  *netType string , -- 网络类型*  *sessionId string , -- 会话id*  *commit\_time bigint ,*  *province string ,*  *city string ,*  *district string ,*  *url string*  *)*  *partitioned by (dt string)*  *stored as parquet*  *;* |

### ETL计算逻辑

计算过程：

1. 过滤出所有pv事件
2. 将pv事件中的event中title去匹配url维度
3. 将事件戳转换成年、月、日

### Spark任务实现

## DWS层建模及ETL开发

### 流量**会话聚合**表建模

按前面章节：《8.3 分层模型》中举的例子，我们需要构建用来做各类流量指标计算的中间模型：

DWS:流量会话聚合表**（一次会话聚合成一条数据）**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 日期 | uid | sessionid | 起始时间 | 结束时间 | pv数 | 省 | 市 | 区 | 手机型号 | osname | osversion | 下载渠道 |

### ETL计算逻辑

计算流程：

源表：dwd\_traffic\_dtl

目标表：dws\_traffic\_session\_agg

计算逻辑：

分组聚合！

uid ： 作为分组key

sessionid： 作为分组key

起始时间： min(commit\_time)

结束时间： max(commit\_time)

pv个数： count(1)

省、市、区、手机型号、操作系统、系统版本都是维度，直接在每组中取一个值 max()

### 流量用户聚合表建模

DWS:流量用户聚合表（**一个用户一条数据**）

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 日期 | uid | 访问次数 | 访问总时长 | pv总数 | 省 | 市 | 区 | 手机型号 | osname | osversion |

### ETL计算逻辑

计算流程：

源表：dws\_traffic\_session\_agg

目标表：dws\_traffic\_user\_agg

计算逻辑：

分组聚合！

uid ： 作为分组key

访问次数： count

访问总时长：sum(end\_time-start\_time)

pv总数： sum(pv\_cnts)

省、市、区、手机型号、操作系统、系统版本都是维度，直接在每组中取一个值 max()

## ADS层报表开发

### 流量概况统计报表建模

**事实：**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 总PV数 | 总访客数UV | 总访问次数 | 平均每次访问时长 | 人均访问次数 | 人均访问深度 | 人均访问时长 | 回头客占比 |

**可能需要的维度：**

日期

地域（省市区）

设备类型

操作系统

App版本

每小时时段

全局汇总，很容易求！—— 全局聚合即可

如果要得到各种维度报表呢？

省 pv总数 uv总数 回头客总数 ......

市 pv总数 uv总数 回头客总数 ......

省 市 pv总数 uv总数 回头客总数 ......

省 市 区 pv总数 uv总数 回头客总数 ......

为避免创建n多的维度组合汇总结果表，可以创建一张全维度表：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 流量概况多维数据立方体cube | | | | | | | | |
| 省 | 市 | 区 | 手机型号 | 操作系统 | 系统版本 | PV总数 | UV总数 | ...... |
| 河北省 | null | null | null | null | null | 8787 | 6543 |  |
| 河南省 | null | null | null | null | null | 5200 | 4800 |  |
| 河北省 | 石家庄 | null | null | null | null | 2200 | 1800 |  |
|  |  |  |  |  |  |  |  |  |

此表，将涵盖所有维度组合下的流量聚合指标！

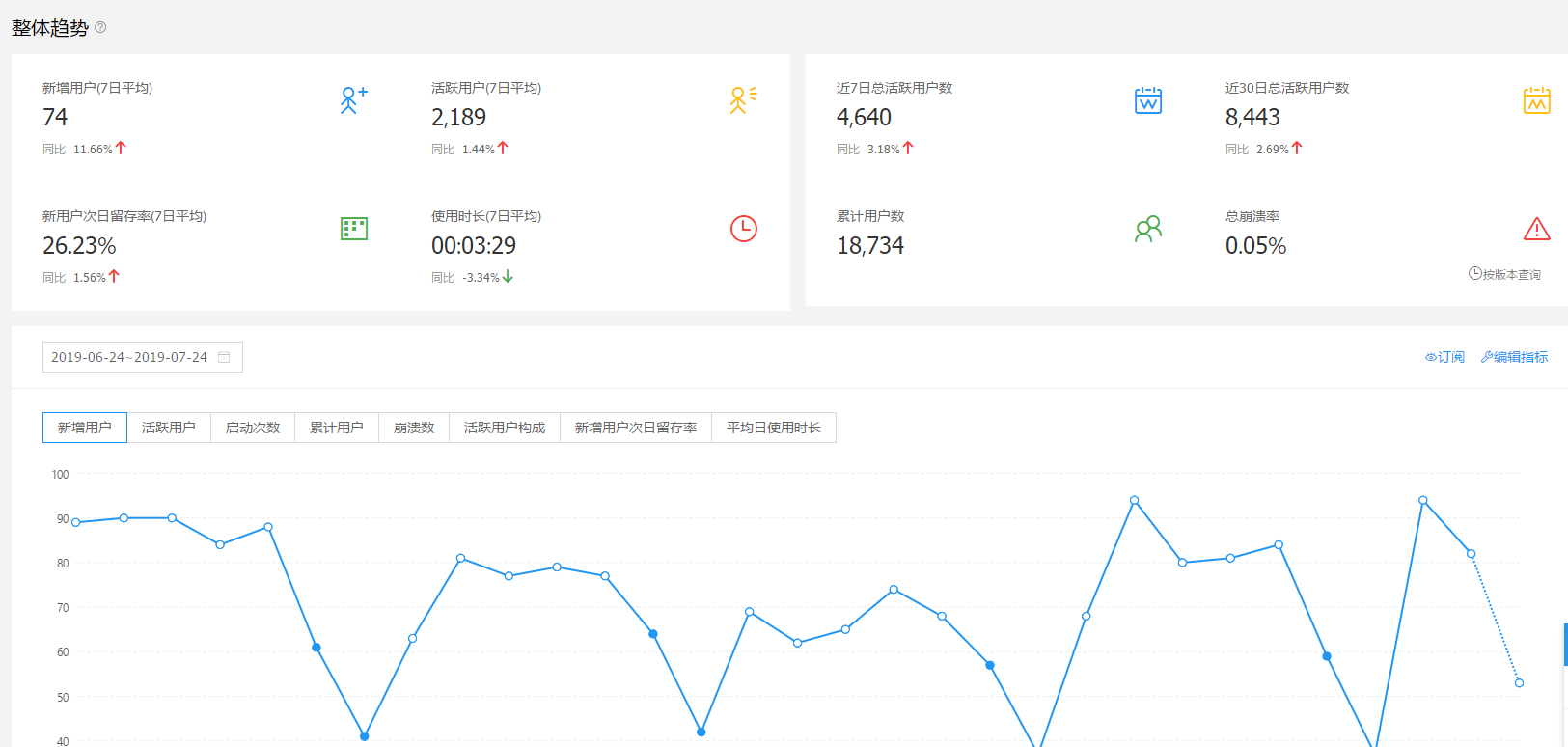
### ETL计算逻辑

# 【流量数据】用户分析-日新日活分析

## DWD层建模思考

### 建模思考过程

* **报表需求1**



发现，报表中有事实指标：新增用户数 活跃用户数 累计总用户数

进而思考：新增用户是什么？怎么判定一个用户是新增用户？

新增用户定义： 计算日之前从没访问过，计算日出现了！

怎么去判别？应该有一个历史用户记录表！

还发现一个事实指标：n日留存数/率

次日留存率定义：

T日新增的用户，在T+1日再次出现的人，就叫次日留存的人

（次日留存人数/T日新用户数）就叫次日留存率

如何去计算次日留存人数？

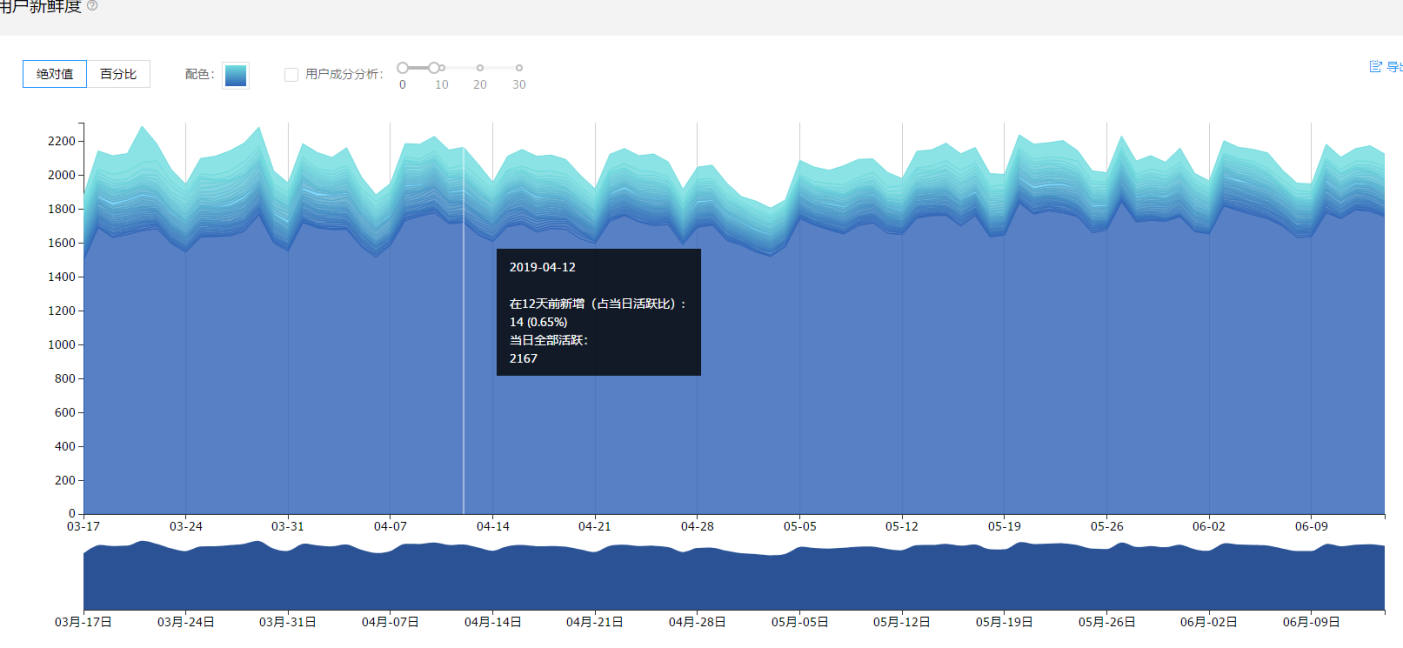
需要一个表：每日新用户记录表 （日新表）

隐含了一个表：每日活跃用户表（日活表）

* **报表需求2**



* **报表需求3**



经过分析，发现，新鲜度计算其实本质上就是留存计算！只不过这个报表中，要求算：

1日后留存 2日后留存 3日后留存 … 30日后留存

* **总结：**

前台产品提出的用户分析相关需求，主要涉及3大事实：

1）新用户数

2）活跃用户数

3）用户留存日

### 日活表建模dwd

报表中，关于日活的事实指标，主要就是一个日活**数**

所以，本日活表中，主要记录：什么时间，什么人出现过，加上各种维度！

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **日活表** |  |  |  |  |  |  |
| 日期 | uid | 省 | 市 | 区 | 下载渠道 | 手机型号 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

### 日新表建模dwd

表结构与日活表完全相同，只是表中的数据不同（日新中只有新增用户）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **日新表** |  |  |  |  |  |  |
| 日期 | uid | 省 | 市 | 区 | 下载渠道 | 手机型号 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

### 历史用户记录表：建模

如果光考虑日新、日活的计算，那么，历史用户记录表只需要记用户id即可

但考虑到留存、用户画像首次登录标签等的计算，可以设计如下模型（留存明细表）：

|  |  |  |
| --- | --- | --- |
| 历史用户记录表 | | |
| Uid | 首次登录日期 | 最近登录日期 |
|  |  |  |



有了这个留存明细表：

既可以作为历史累计用户记录表用！

又方便地计算各种时间段的留存数据！

## DWD层ETL开发

### 日活表

/\*

用户主题dwd层：活跃用户明细表：dwd\_user\_active\_d

日期 uid 省 市 区 手机型号 osname osver

@Author HUNTER

@Date 2019-07-26

@源表：dws\_traffic\_user\_agg

@目标：dwd\_user\_active\_d

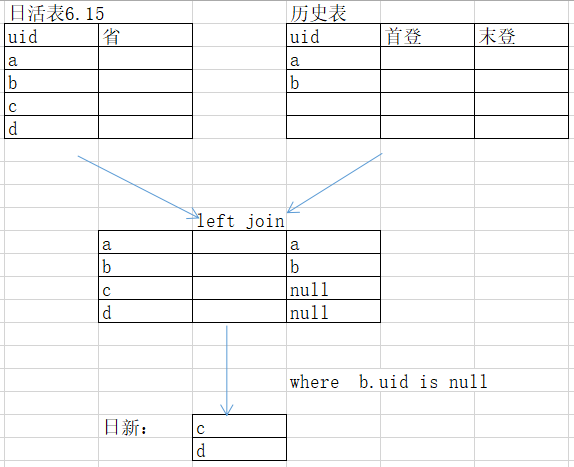
@计算逻辑：

抽数据

\*/

抽所需要的字段即可

### 日新表



### 历史累计用户表



代码实现：

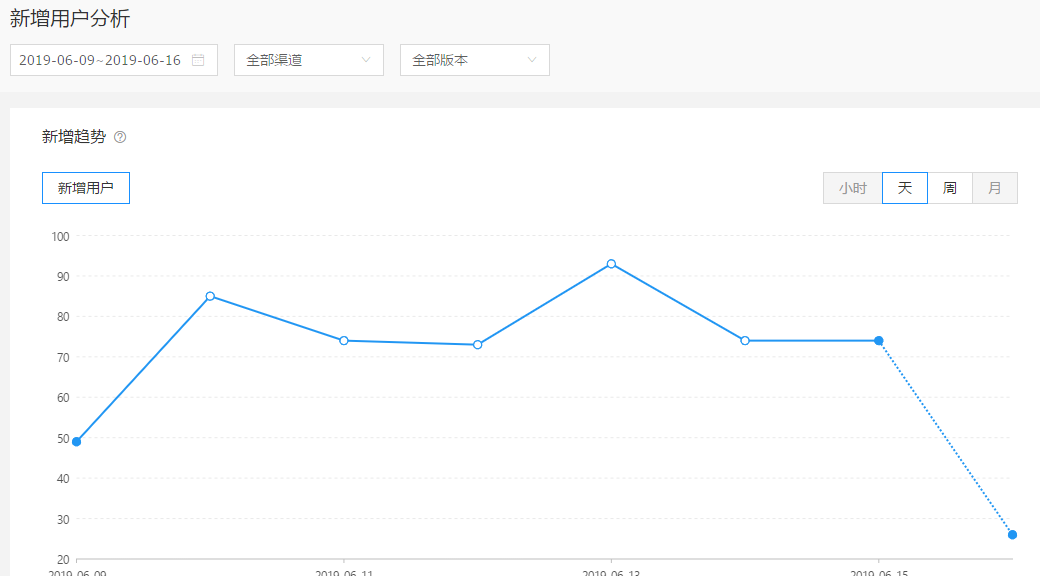
|  |
| --- |
| /\*  用户主题dwd层：历史累计用户记录表：dwd\_user\_his  uid 首次登陆日期 最近登陆日期  @Author HUNTER  @Date 2019-07-26  @源表：dwd\_user\_active\_d  @目标：dwd\_user\_his  @计算逻辑：  1. 将前日的历史记录结果表 left join 当日的日活  join到的， select 历史表的uid/首次登录日期,日活表的日期  join不到的， select 历史表的uid/首次登录日期/最近登录日期  2. 将日新表的uid,日期，日期 加入历史记录表  \*/  -- 建表  drop table if exists dwd\_user\_his;  create table dwd\_user\_his(  uid string,  first\_login\_day string,  last\_active\_day string  )  stored as parquet;  -- etl 开发 -- 要在日新表计算完毕，才进行下面的计算  insert into table dwd\_user\_his  select  a.uid,  a.first\_login\_day,  if(b.uid is null,a.last\_active\_day,b.dt) as last\_active\_day  from  dwd\_user\_his a  left join  dwd\_user\_active\_d b  on b.dt='2019-06-15' and a.uid = b.uid  union all  select  uid,  dt as first\_login\_day,  dt as last\_active\_day  from dwd\_user\_new\_d  where dt='2019-06-15'  ; |

## DWS层汇总表设计和开发

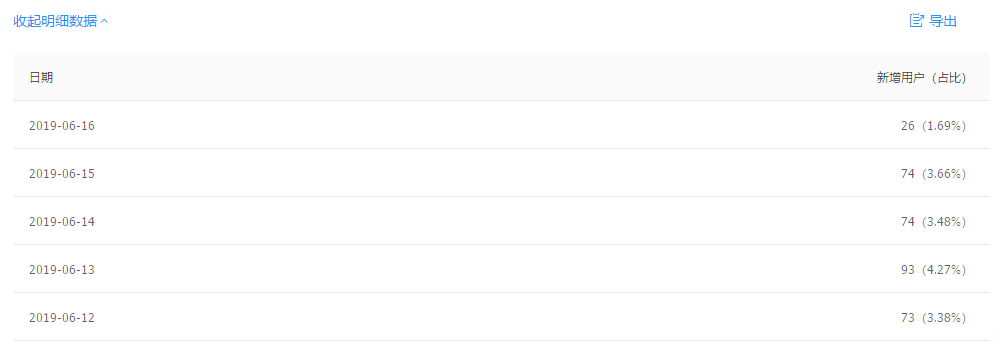
### ADS层需求分析

产品端有如下数据报表可视化需求：

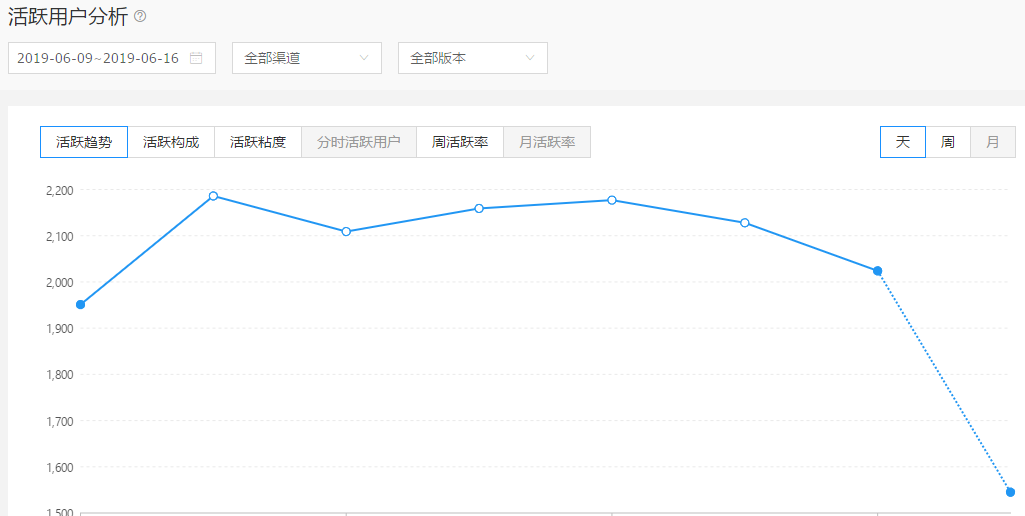
#### 需求1



#### 需求2



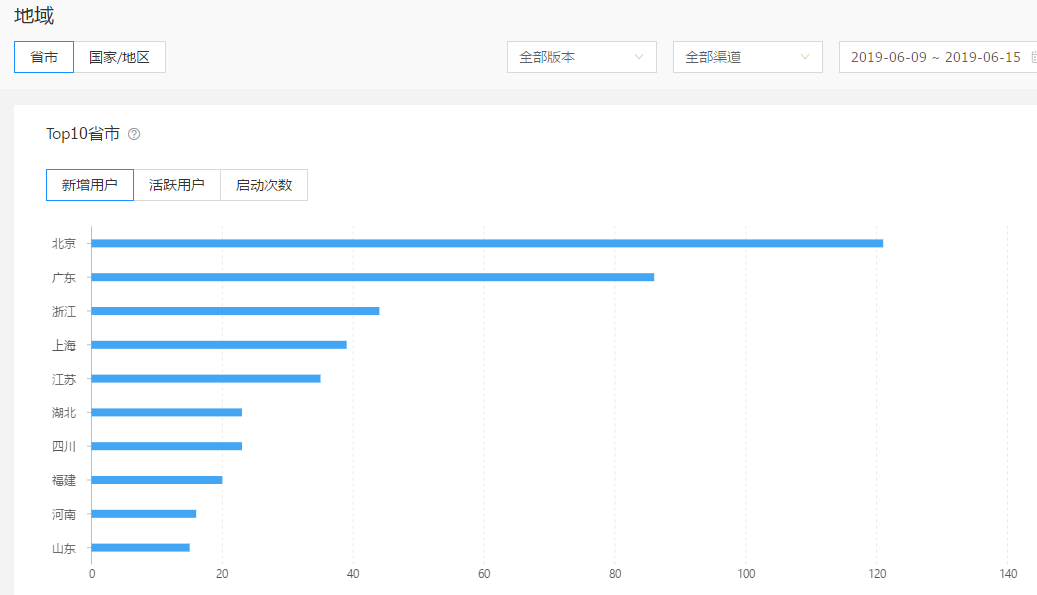
#### 需求3



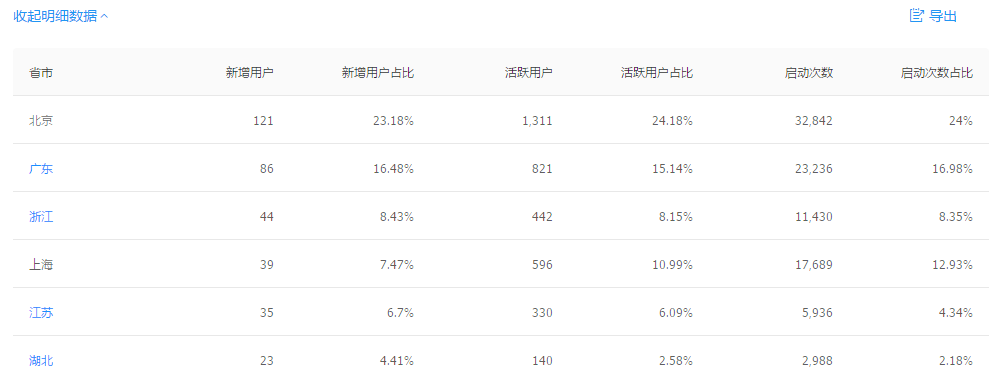
#### 需求4



#### 需求5



#### 需求6



### DWS层设计分析

以上需求中，经过提炼，发现，所涉及的事实指标有：

新用户数

活跃用户数

以及占比

剩下的，就是通过各种维度来进行聚合分析

所以，这一对的报表可视化需求，都可以通过底层的两个数据立方体来支撑：

DWS日新数cube

DWS日活数cube

### DWS层日新cube计算

|  |
| --- |
| /\*  用户主题dws层：日新数cube表：dws\_user\_new\_cube  省 市 区 手机型号 osname osver user\_new\_cnts  @Author HUNTER  @Date 2019-07-26  @源表：dwd\_user\_new\_d  @目标：dws\_user\_new\_cube  @计算逻辑：  根据各种维度来构建数据立方体  \*/  -- 建表  create table dws\_user\_new\_cube(  province string,  city string,  district string,  manufacture string,  osname string,  osver string,  user\_new\_cnts int  )  partitioned by (dt string,dim string)  stored as parquet  ;  -- etl计算  insert into table dws\_user\_new\_cube partition(dt='2019-06-15',dim)  select  province,  city,  district,  manufacture,  osname,  osver,  count(1) as user\_new\_cnts,  grouping\_\_id as dim  from dwd\_user\_new\_d  where dt='2019-06-15'  group by province,city,district,manufacture,osname,osver  with cube  ; |

### DWS层日活cube计算

|  |
| --- |
| /\*  用户主题dws层：日新数cube表：dws\_user\_active\_cube  省 市 区 手机型号 osname osver user\_active\_cnts  @Author HUNTER  @Date 2019-07-26  @源表：dwd\_user\_active\_d  @目标：dws\_user\_active\_cube  @计算逻辑：  根据各种维度来构建数据立方体  \*/  -- 建表  drop table if exists dws\_user\_active\_cube;  create table dws\_user\_active\_cube(  province string,  city string,  district string,  manufacture string,  osname string,  osver string,  user\_active\_cnts int  )  partitioned by (dt string,dim string)  stored as parquet  ;  -- etl 计算  insert into table dws\_user\_active\_cube partition(dt='2019-06-15',dim)  select  province,  city,  district,  manufacture,  osname,  osver,  count(1) as user\_active\_cnts,  grouping\_\_id as dim  from dwd\_user\_active\_d  where dt='2019-06-15'  group by province,city,district,manufacture,osname,osver  with cube  ; |

## ADS层报表开发

以如下报表为例：日新日活地域分析 ads\_user\_new\_actvie\_province



分析：

本表的实质为，查询日新cube中的省维度下的聚合数据

查询日活cube中的省维度下的聚合数据

查询日活、日新cube中的总数据

|  |
| --- |
|  |

# 【流量数据】用户分析-新用户留存



## 留存明细建模概述DWS

留存明细表本身的建模很简单

|  |  |  |  |
| --- | --- | --- | --- |
| **留存明细表** |  |  |  |
| **新增日期** | **新增数** | **留存数** | **留存天数** |
| **6.5** | 4 | 3 | 1 |
| **6.5** | 4 | 3 | 2 |
| **6.6** | 2 | 2 | 1 |
| **6.5** | 4 | 3 | 3 |
| **6.6** | 2 | 1 | 2 |
| **6.7** | 2 | 2 | 1 |
|  |  |  |  |

**但是留存明细表的计算会比较复杂，依赖于底层的建模**

所以，需要详细设计留存明细的底层模型设计

**设计方案1：**

利用历史用户记录明细表来实现留存明细计算

历史明细表的功能：dws\_user\_his

记录每一个用户的新增日期，以及每个用户在最后活跃日！

|  |  |  |
| --- | --- | --- |
| **用户登录历史记录表** | |  |
| **uid** | **首次登录日期** | **最后登录日期** |
| **a** | **5** | **7** |
| **b** | **5** | **7** |
| **c** | **5** | **6** |
| **d** | **5** | **7** |
| **e** | **6** | **6** |
| **f** | **6** | **7** |
| **g** | **7** | **7** |
| **h** | **7** | **7** |

**设计方案2：**

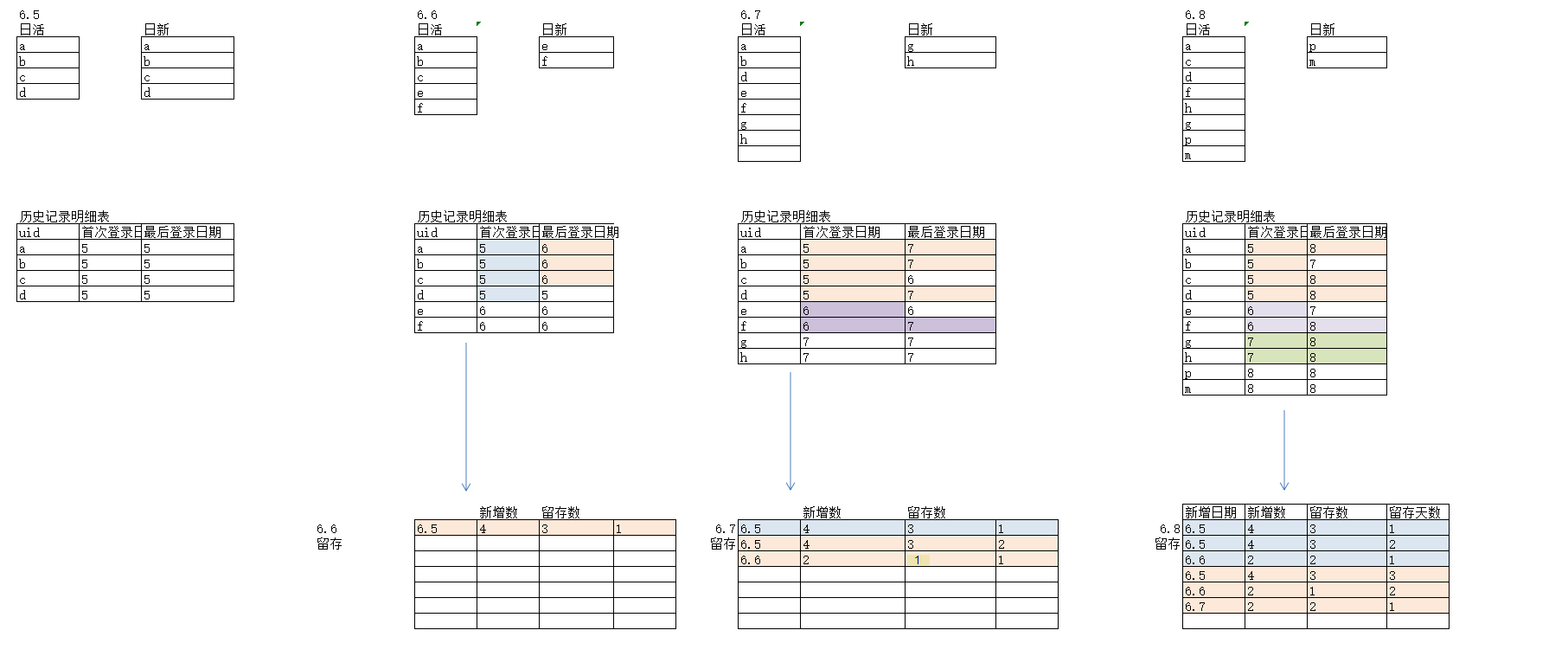
利用用户活跃记录拉链表来实现留存明细计算

拉链表的功能：

记录每一个用户的新增日期，每一个连续活跃区间的起始日和结束日！

|  |  |  |  |
| --- | --- | --- | --- |
| 用户登录历史记录拉链表 | | |  |
| uid | 新增日 | 开始 | 结束 |
| a | 5.20 | 5.20 | 5.26 |
| a | 5.20 | 5.29 | 6.1 |
| a | 5.20 | 6.3 | 9999-12-31 |
| b | 5.22 | 5.22 | 5.30 |
| b | 5.22 | 6.3 | 6.5 |
| c | 6.3 | 6.3 | 9999-12-31 |
|  |  |  |  |
|  |  |  |  |

## DWS留存明细计算方案1



具体计算逻辑（方式1）：

|  |
| --- |
| */\* 用户主题ads层：新用户留存明细表 dws\_newuser\_retention dtstr,new\_cnts,rtn\_days,rtn\_cnts   @Author HUNTER @Date 2019-07-28  @源表：dwd\_user\_his 044MSB 2019-06-14 2019-06-15 05PTAD 2019-06-14 2019-06-18 08HEUV 2019-06-14 2019-06-18 0ANHNZ 2019-06-15 2019-06-18 0BHV4F 2019-06-15 2019-06-18 0IO4RF 2019-06-16 2019-06-17 0KFM55 2019-06-16 2019-06-18 0KUTLG 2019-06-17 2019-06-18 0KWNWM 2019-06-17 2019-06-18 0L87JG 2019-06-18 2019-06-18  @目标：dws\_newuser\_retention  @计算逻辑：  1日后留存： 应该计算 （6-18）-1天  新增总数：这个日期的新增总数  1日后留存数：最后登陆日为计算日的人数   2日后留存： 应该计算 （6-18）-2天  新增总数：这个日期的新增总数  2日后留存数：最后登陆日为计算日的人数  \*/* **select** *date\_sub*(**'2019-06-18'**,1) **as** dtstr, *count*(*if*(**first\_login\_day**=*date\_sub*(**'2019-06-18'**,1),1,**null**)) **as** new\_cnts, 1 **as** rtn\_days, *count*(*if*(**last\_active\_day**=**'2019-06-18'**,1,**null**)) **as** rtn\_cnts **from** dwd\_user\_his **where** *datediff*(**'2019-06-18'**,**first\_login\_day**) **between** 1 **and** 30  **union  select** *date\_sub*(**'2019-06-18'**,2) **as** dtstr,  *count*(*if*(**first\_login\_day**=*date\_sub*(**'2019-06-18'**,2),1,**null**)) **as** new\_cnts,  2 **as** rtn\_days,  *count*(*if*(**last\_active\_day**=**'2019-06-18'**,1,**null**)) **as** rtn\_cnts **from** dwd\_user\_his **where** *datediff*(**'2019-06-18'**,**first\_login\_day**) **between** 1 **and** 30  **union  select** *date\_sub*(**'2019-06-18'**,3) **as** dtstr,  *count*(*if*(**first\_login\_day**=*date\_sub*(**'2019-06-18'**,3),1,**null**)) **as** new\_cnts,  3 **as** rtn\_days,  *count*(*if*(**last\_active\_day**=**'2019-06-18'**,1,**null**)) **as** rtn\_cnts **from** dwd\_user\_his **where** *datediff*(**'2019-06-18'**,**first\_login\_day**) **between** 1 **and** 30  .......... |

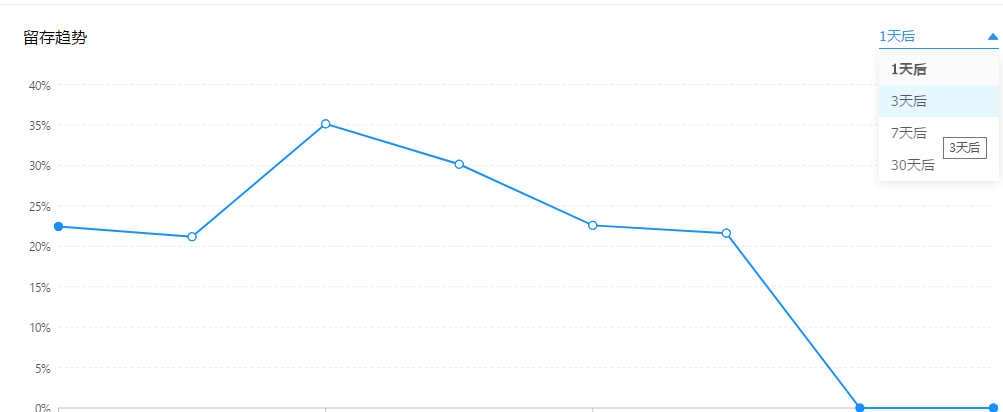
具体计算逻辑（方式2）：

|  |
| --- |
| select  first\_login\_day,  count(1) as new\_cnts,  count(if(last\_active\_day='2019-06-15',1,null)) as rtn\_cnts,  datediff('2019-06-15',first\_login\_day) as rtn\_days  from dwd\_user\_his where datediff('2019-06-15',first\_login\_day) between 1 and 30  group by first\_login\_day  ; |

## ADS留存用户明细报表

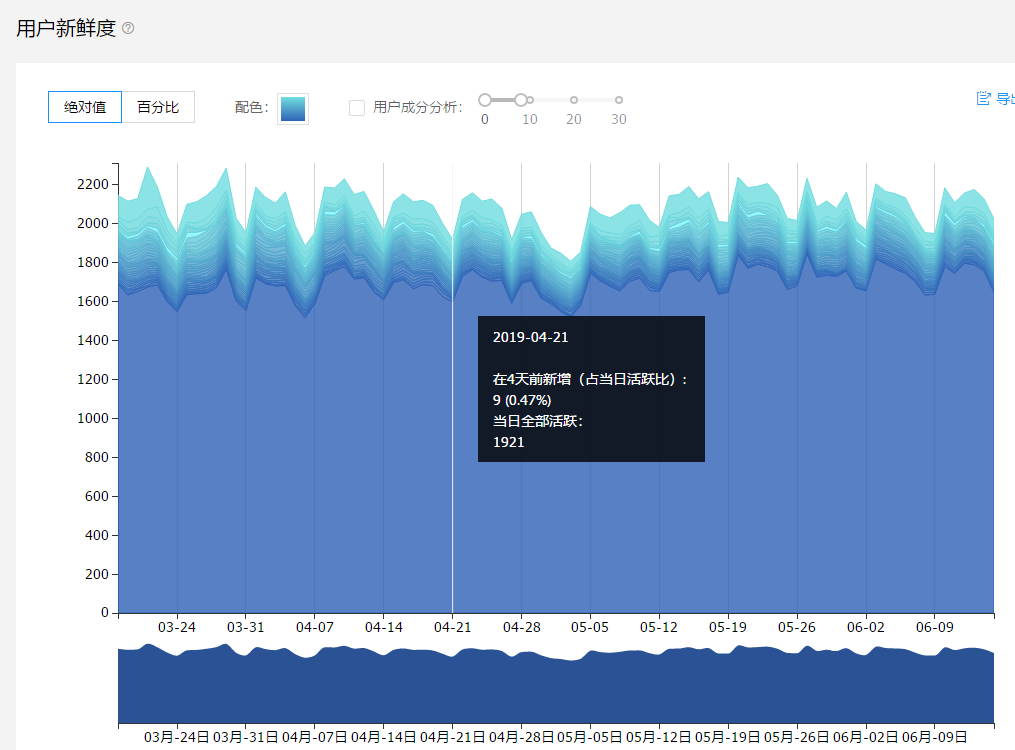


直接查询“留存明细表”即可



查留存明细表即可（只不过将数量换成百分比）！

# 【流量数据】用户分析-用户新鲜度



## 建模

通过分析需求中原型页面，撇去表象，发现本真：

无非就是需要在某个日期上的活跃用户中的“不同日期新增用户”占比

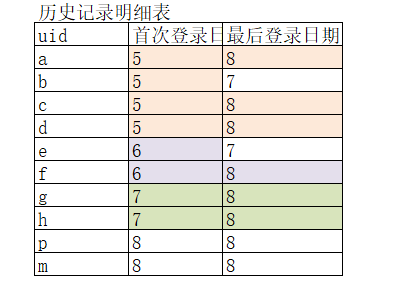
然后，发现可以用两种形式的模型来为页面提供数据支持：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 方式1：横表 | | | | | |
| 计算日期 | 1天前新增占比 | 2天前新增占比 | 3天前新增占比 | 4天前新增占比 | ...... |
| 2019/6/15 | 65% | 70% | 48% | 66% | ...... |

|  |  |  |
| --- | --- | --- |
| **方式2：竖表** | | |
| 计算日期 | 占比类别 | 占比 |
| 2019/6/15 | 1天前新增占比 | 46% |
| 2019/6/15 | 2天前新增占比 |  |
| 2019/6/15 | 3天前新增占比 |  |
| 2019/6/15 | 4天前新增占比 |  |
| 2019/6/15 | 5天前新增占比 |  |
| 2019/6/15 | 6天前新增占比 |  |
| 2019/6/15 | 7天前新增占比 |  |
| 2019/6/15 | ....... | ...... |

## ETL开发计算

本报表，基于**历史用户记录明细表**，可轻松得出！



计算逻辑：

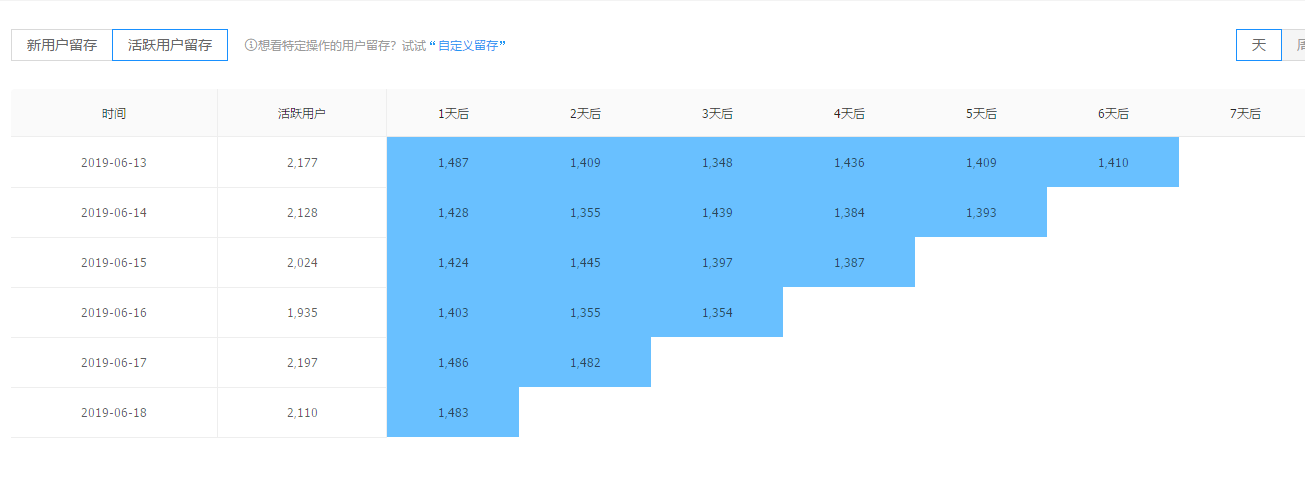
末登日期=计算日，则是计算日的活跃用户

首登日=计算日-1 ，则是1天前的新增用户

首登日=计算日-2 ，则是2天前的新增用户

|  |
| --- |
| *-- elt 计算 --内层得到的结果：* **'2019-06-18'**,0,6 **'2019-06-18'**,1,8 **'2019-06-18'**,2,10 **'2019-06-18'**,3,20  *-- 外层得到总活跃数* **'2019-06-18'**,0,6 ,44 **'2019-06-18'**,1,8 ,44 **'2019-06-18'**,2,10 ,44 **'2019-06-18'**,3,20 ,44   **select** dtstr, fresh\_days, user\_cnts, *sum*(user\_cnts) over(**rows between** unbounded preceding **and** unbounded following) **as** act\_total  **from** ( **select '2019-06-18' as** dtstr, *if*(*datediff*(**last\_active\_day**,**first\_login\_day**)>30,9999,*datediff*(**last\_active\_day**,**first\_login\_day**)) **as** fresh\_days, *count*(1) **as** user\_cnts  **from** eagle.dwd\_user\_his **where last\_active\_day**=**'2019-06-18' group by** if(datediff(last\_active\_day,first\_login\_day)>30,9999,datediff(last\_active\_day,first\_login\_day)) ) o ; |

# 【流量数据】用户分析-活跃用户留存



## 建模

|  |
| --- |
| /\*  活跃用户留存报表  建模：竖表  ads\_user\_act\_retention  \*/  create table ads\_user\_act\_retention(  dt string,  act\_cnts int,  retention\_days string,  retention\_cnts int  )  stored as parquet; |

## 设计计算逻辑

每日都只要以当日为基数

往前推1天、2天、3天、4天、5天、……的留存数计算



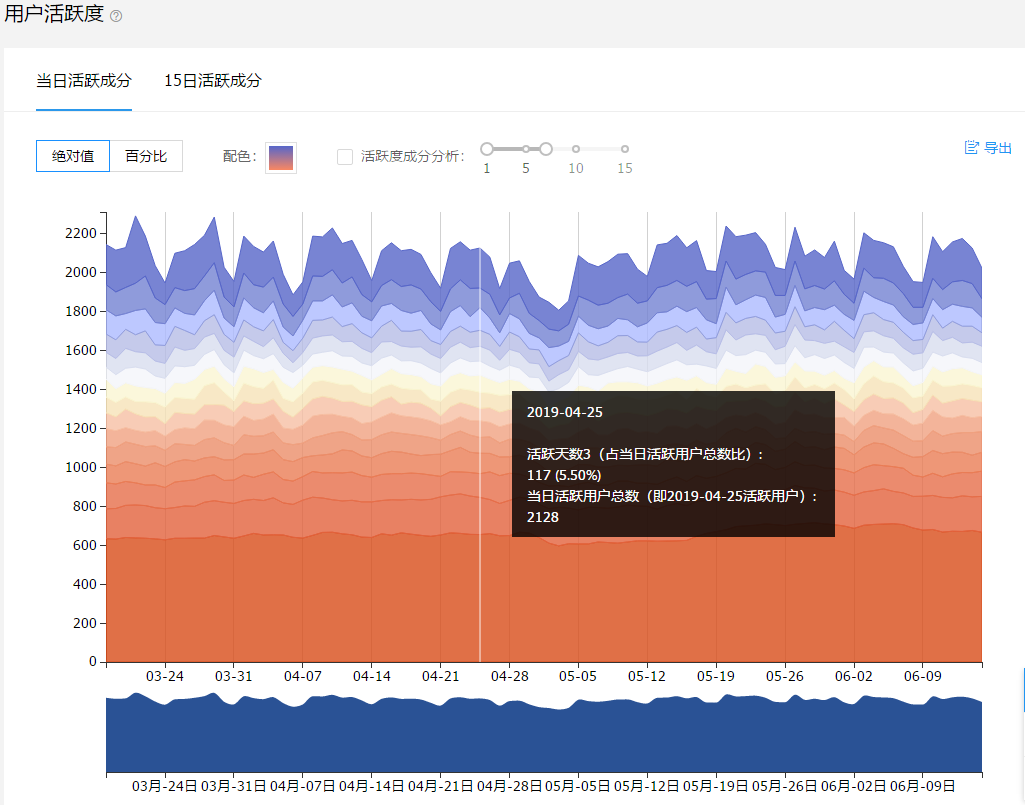
|  |
| --- |
| /\*  方案1：ETL 开发  计算逻辑：  1. 源表： 各天的日活表  2. 逻辑： 滚动计算，计算前N天到当天的，活跃留存数  比如，6.6号，算6.5-6.6的1日留存  6.7号，算6.5-6.7的2日留存，算6.6-6.7的1日留存  6.8号，算6.5-6.8的3日留存，算6.6-6.8的2日留存，算6.7-6.8的1日留存  ......    将 6.5的日活 join 6.8的日活，求count --》 得到6.5号的3日后活跃留存  6.6的日活 join 6.8的日活，求count --》 得到6.6号的2日后活跃留存  6.7的日活 join 6.8的日活，求count --》 得到6.7号的1日后活跃留存  \*/ |

## ETL开发计算

|  |
| --- |
|  |

# 【流量数据】用户分析-活跃度

## 报表需求：



分析可得，本报表需求的实质是统计用户的连续活跃行为

## 建模

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 用户活跃度报表 | | | | |
| 日期 | 当日总活跃数 | 活跃天数类别 | 类别人数 | 人数占比 |
| 16 | 2000 | 1 | 600 |  |
| 16 | 2000 | 2 | 1000 |  |

建表语句：

|  |
| --- |
| drop table if exists ads\_user\_act\_ingredients;  create table ads\_user\_act\_ingredients(  dt string,  act\_cnts int comment '当日活跃总数',  continuos int comment '连续天数',  cnts int comment '人数',  per double comment '人数占比'  )  stored as parquet; |

## 设计计算逻辑

日活表

|  |  |
| --- | --- |
| 2019-06-02 | U001 |
| 2019-06-03 | U001 |
| 2019-06-04 | U001 |
| 2019-06-07 | U001 |
| 2019-06-10 | U001 |
| 2019-06-11 | U001 |
| 2019-06-12 | U001 |
| 2019-06-13 | U001 |
| 2019-06-14 | U001 |
| 2019-06-15 | U001 |

活跃记录:拉链表

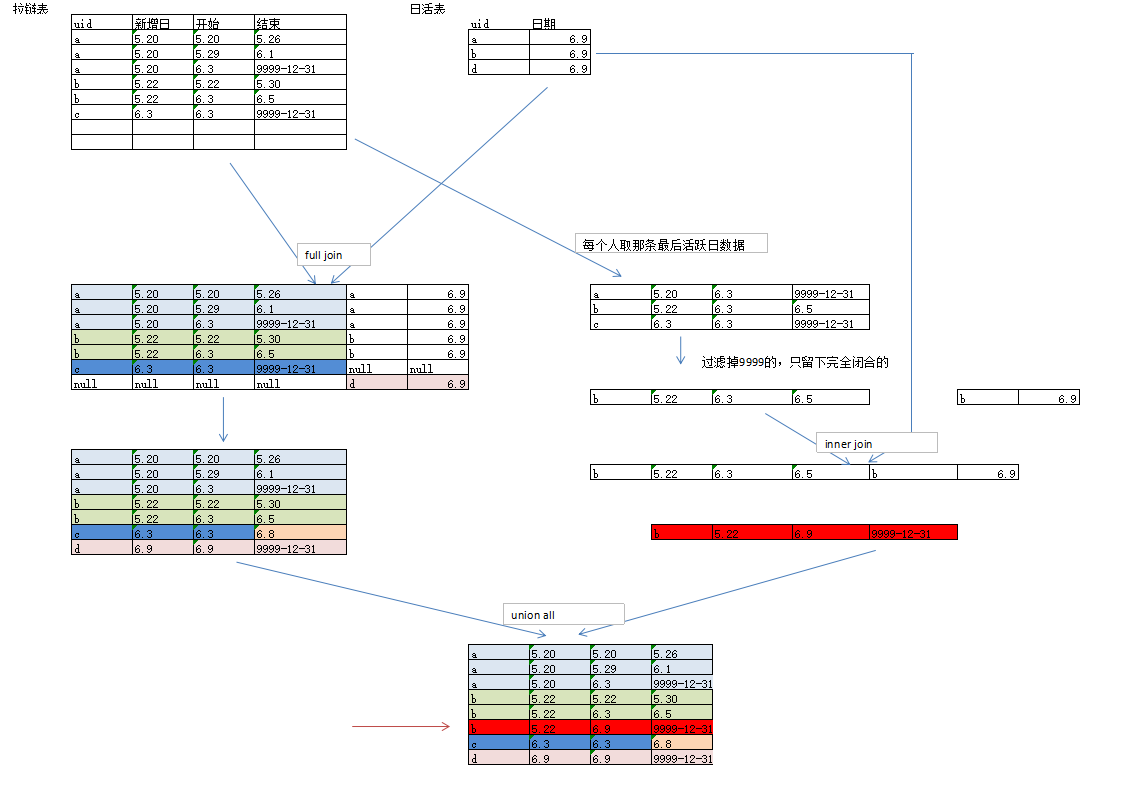
|  |  |  |  |
| --- | --- | --- | --- |
| Uid | 首登日 | 活跃区间起始 | 活跃区间结束 |
| U001 | 2019-06-02 | 2019-06-02 | 2019-06-04 |
| U001 | 2019-06-02 | 2019-06-07 | 2019-06-07 |
| U001 | 2019-06-02 | 2019-06-10 | 9999-12-31 |
| U002 | 1029-06-12 | 2019-06-12 | 9999-12-31 |

|  |
| --- |
| /\*  ETL 开发  计算逻辑：  1.源表：活跃记录拉链表 dws\_user\_active\_zip  2.逻辑：先过滤出当日活跃的记录，然后分别累计（连续1天的，连续2天的....）人数即可  \*/ |

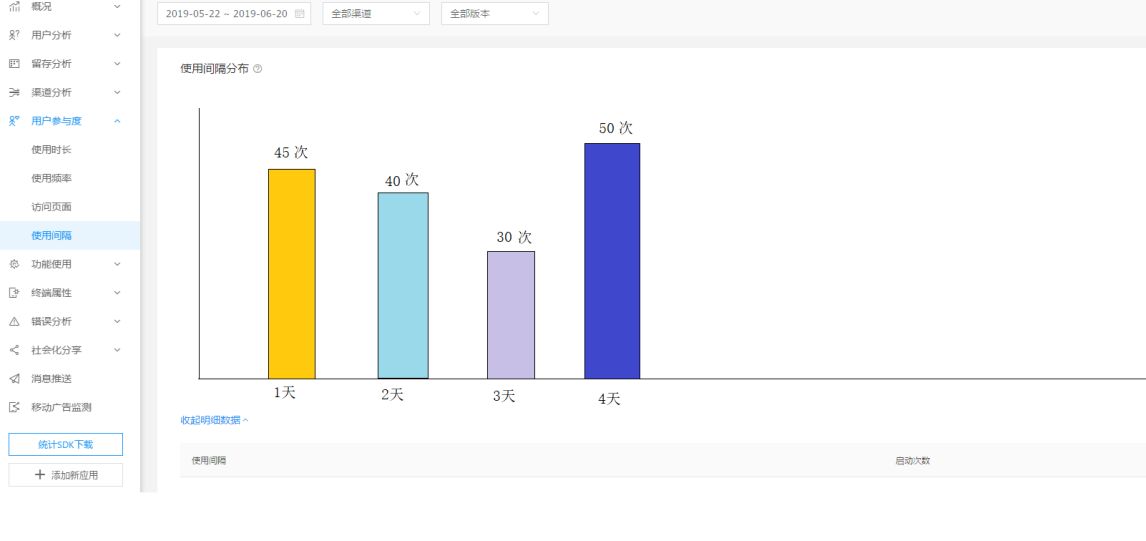
## 用户活跃记录拉链表

拉链表详细介绍请参阅《28.2 数仓知识补充：拉链表介绍》

拉链表本身的计算过程比较复杂，流程逻辑如下图所示：



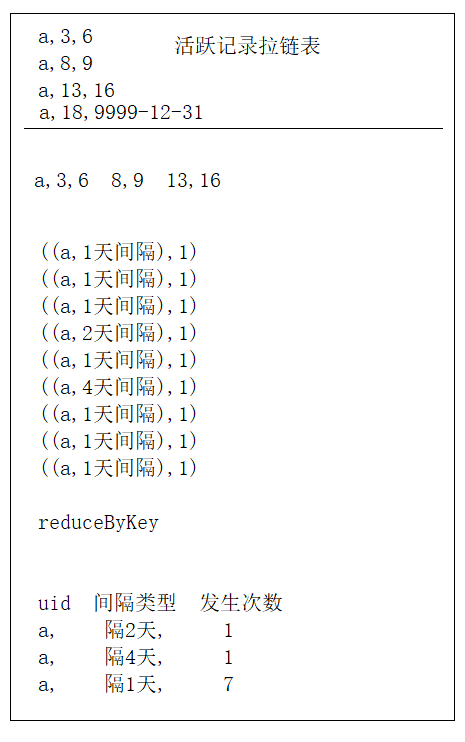
# 【流量数据】用户分析-访问间隔分布



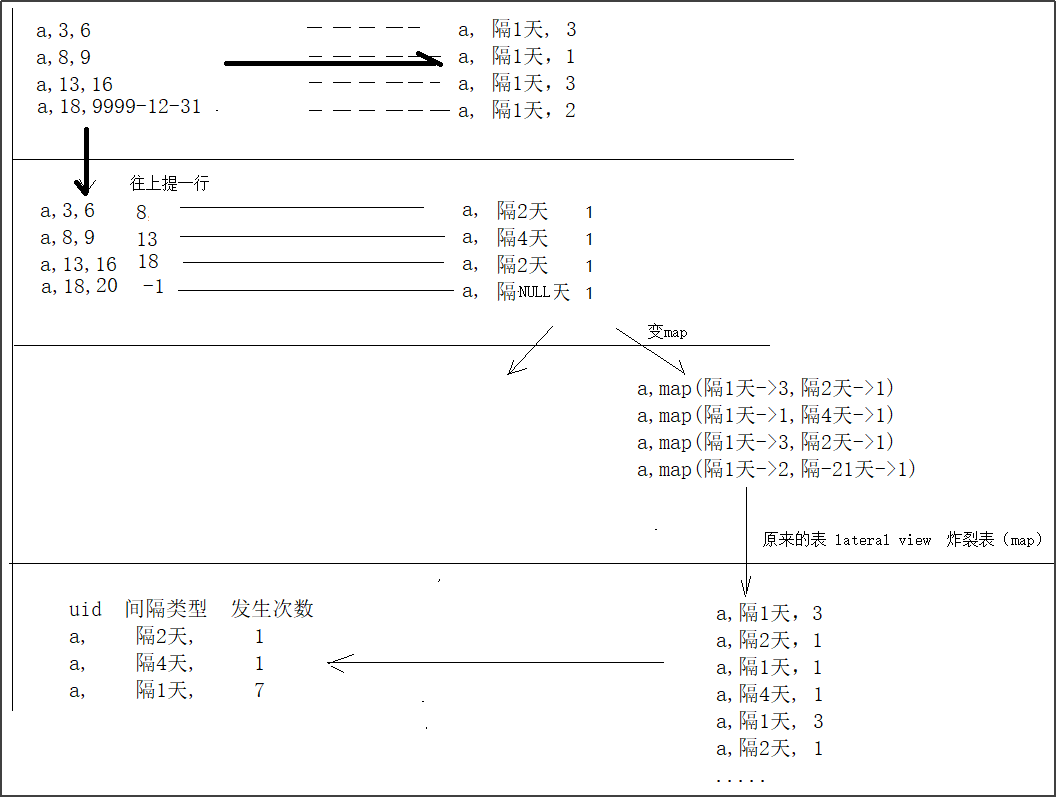
|  |  |  |
| --- | --- | --- |
| 访问间隔分布报表 | | |
| 截止日期 | 隔几天 | 发生过多少人次 |
| 2019-06-16 | 1 | 500 |
| 2019-06-16 | 2 | 400 |
| 2019-06-16 | 3 | 800 |

## 建模

## 实现方案1：spark计算逻辑



## 实现方案2：sql计算逻辑

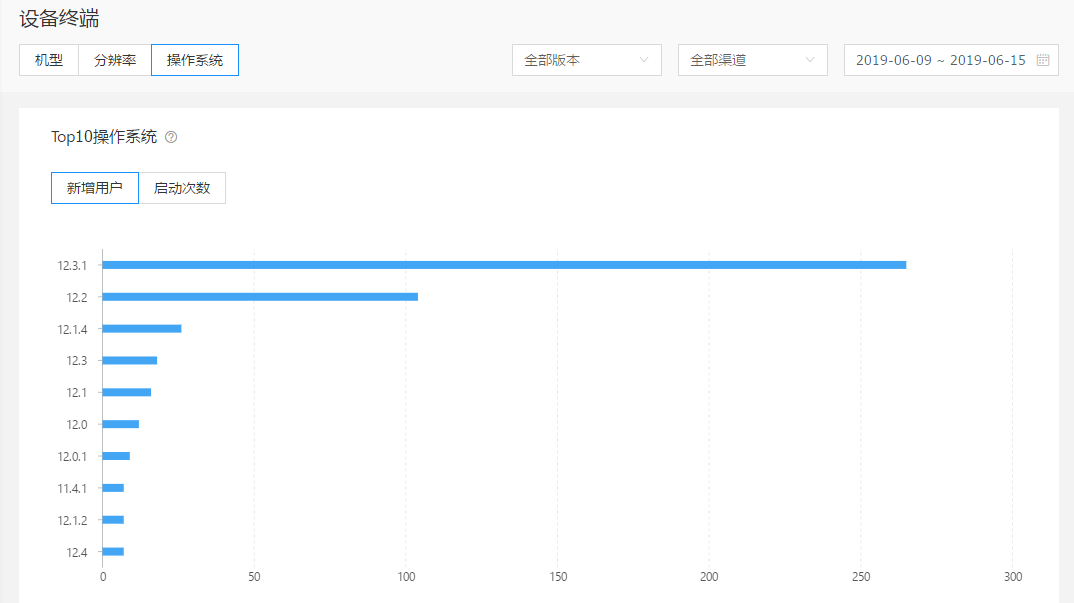


sql语句：

|  |
| --- |
| /\*  方式2：用sql实现  \*/  with tmp as  (select  uid,  datediff(if(end\_dt='9999-12-31','2019-06-19',end\_dt),begin\_dt) as day1\_cnts,  end\_dt,  lead(begin\_dt,1,null) over(partition by uid order by begin\_dt) as jump\_dt  from dws\_user\_active\_zip  where dt='2019-06-19')  select  uid,  interval\_type,  sum(cnts)  from  (  select  uid,  interval\_type,  cnts  from  (  select  uid  ,map('隔1天',day1\_cnts,if(jump\_dt is null,"NA",concat('隔',datediff(jump\_dt,end\_dt),'天')),1) as m  from tmp  )o  lateral view  explode(m) t as interval\_type,cnts  ) o2  where interval\_type!='NA'  group by uid,interval\_type |

# 【流量数据】终端属性分析

## 设备分析报表





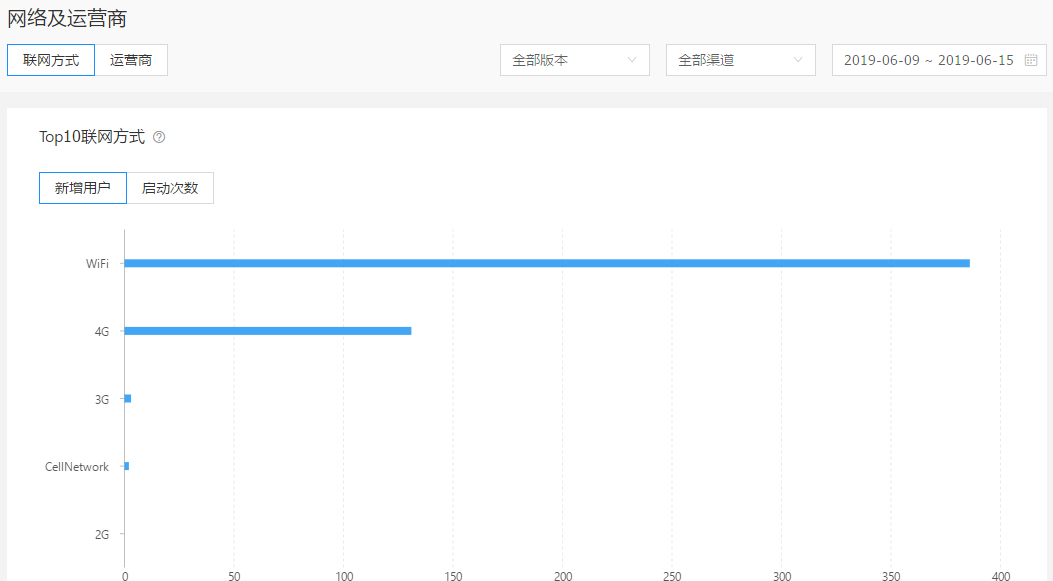
本报表的实质就是：

统计指标为：“新用户数”和“会话数”

维度为：操作系统

因而，可以通过日活表按维度聚合得出

## 网络及运营商分析报表



本报表的实质就是：

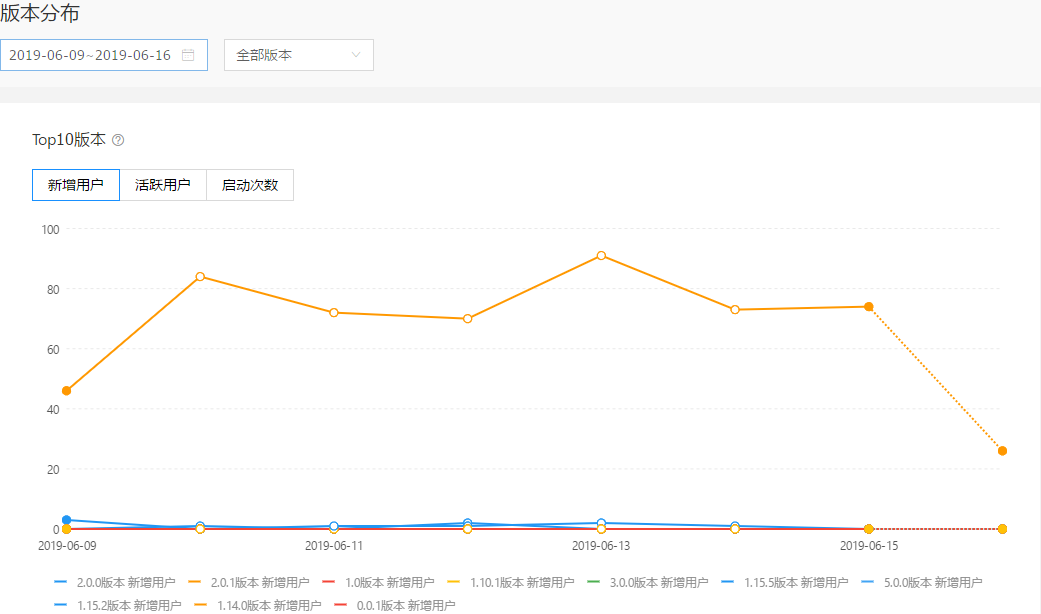
统计指标为：“新用户数”和“会话数”

维度为：联网方式、运营商

因而，可以通过日活表按维度聚合得出

# 【流量数据】app版本分析

## 版本分布报表





## 建模

## ETL开发

## app版本升级明细记录表

### 需求



### 计算逻辑：

将后一行记录中的版本，提升到前一行，来进行版本比较

|  |
| --- |
| create table t2(uid string,day string,ver string)  row format delimited fields terminated by ',';  load data local inpath '/root/t2.dat' into table t2;  uid, day, ver  a,2019-06-20,1.02  a,2019-06-20,1.02  a,2019-06-20,1.02  a,2019-06-20,2.0  a,2019-06-20,2.0  a,2019-06-20,2.5  a,2019-06-20,2.5  b,2019-06-20,1.2  b,2019-06-20,1.2  b,2019-06-20,2.5  c,2019-06-20,2.0  c,2019-06-20,2.0  ==> 期待的结果  uid, 日期 升级前 升级后  a ,2019-06-20 ,1.02 ,2.0  a ,2019-06-20 ,2.0 ,2.5  b ,2019-06-20 ,1.2 ,2.5  with tmp as(  select  uid,  day,  ver,  lead(ver,1,null) over(partition by uid order by ver) as ver2  from t2  )  select \* from tmp where ver2>ver |

# 【流量数据】事件概况分析

## 事件背景介绍

事件： 就是用户在app上或者网页上所做的行为

比如：点赞，收藏，评分，转发，点击，添加购物车，提交订单，去支付…………..



事件如何获取？

通过在app上用sdk**埋点**，或者在网页上用js埋点，监听用户的操作，生成相应的事件描述发送到日志服务器；

网页上嵌入jd-sdk的代码示例：

<a href="#" onclick="\_czc.push(['\_trackEvent', '小说', '打分', '达芬奇密码','5','dafen']);">打分</a>

## 事件分析整体建模思考

建模的思考过程：

1. 通过对各类报表需求的分析，整理出所需要的数据元素：

事件类型 事件操作的标的 产生事件的位置 产生事件的人/会话 产生事件的时间

2. 报表中的统计指标的大略计算逻辑：

是否需要复杂计算！

基于上述思考的结果，我们可以在DWD层设计一个事件明细表：

在DWS层做两个聚合表（一个按会话聚合，一个按用户聚合）

## DWD事件明细表 dwd\_event\_detail

### 建模

功能：记录用户操作所产生的每一个事件的明细信息

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 事件明细表 | | | | | | |
| 事件id | 用户id | 会话id | 事件发生页面 | 事件标的 | 事件value | 时间戳 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

建表语句：

|  |
| --- |
| drop table if exists dwd\_event\_detail;  create table dwd\_event\_detail(  event\_type string comment '事件名称',  uid string,  sessionid string,  location string comment '事件所发生的页面',  event\_dest string comment '事件的标的',  event\_value double comment '事件的值(比如一个商品的价格，一个评分值)',  commit\_time bigint comment '事件发生的时间戳'  )  partitioned by (dt string)  stored as parquet; |

### ETL 开发

|  |
| --- |
| insert into table dwd\_event\_detail partition(dt='2019-06-16')  select  logtype as event\_type,  uid,  sessionid,  eventmap['url'] as location,  case logtype  when 'act\_click' then eventmap['actid']  when 'act\_join' then eventmap['actid']  when 'act\_register' then eventmap['actid']  when 'ad\_click' then eventmap['ad\_id']  when 'ad\_show' then eventmap['ad\_id']  when 'addcart' then eventmap['skuid']  when 'favor' then eventmap['skuid']  when 'order\_pay' then eventmap['order\_id']  when 'order\_sub' then eventmap['order\_id']  when 'pg\_view' then eventmap['skuid']  when 'rate' then eventmap['skuid']  when 'search' then eventmap['keywords']  when 'startup' then eventmap['']  when 'thumbup' then eventmap['skuid']  else null  end as event\_dest,  case logtype  when 'order\_pay' then eventmap['total\_money']  when 'rate' then eventmap['score']  else null  end as event\_value,  commit\_time  from dwd\_traffic\_log  where dt='2019-06-16'; |

## DWS事件汇总表

### 事件会话聚合

#### 建模

|  |  |  |  |
| --- | --- | --- | --- |
| 事件会话聚合表 | | | |
| 事件id | 用户id | 会话id | 事件次数 |
|  |  |  |  |
|  |  |  |  |

建表语句：

|  |
| --- |
| drop table if exists dws\_event\_overall\_agg\_session;  create table dws\_event\_overall\_agg\_session(  event\_type string,  uid string,  sessionid string,  cnts int  )  partitioned by (dt string)  stored as parquet; |

#### ETL

**计算逻辑：**

1.源表，dwd层的事件明细表：dwd\_event\_detail

2.计算逻辑： 分组聚合

|  |
| --- |
| insert into table dws\_event\_overall\_agg\_session partition(dt='2019-06-16')  select  event\_type,uid,sessionid,  count(1) as cnts  from dwd\_event\_detail  where dt='2019-06-16'  group by event\_type,uid,sessionid  ; |

### 事件用户聚合

#### 建模

|  |  |  |
| --- | --- | --- |
| 事件用户聚合表 | | |
| 事件id | 用户id | 事件次数 |
|  |  |  |
|  |  |  |

**建表语句：**

|  |
| --- |
| create table dws\_event\_overall\_agg\_u(  event\_type string,  uid string,  cnts int  )  partitioned by (dt string)  stored as parquet; |

#### ETL

**计算逻辑：**

1. 源表，事件次数会话聚合表：dws\_event\_overall\_agg\_session

2. 逻辑：按用户聚合

|  |
| --- |
| insert into table dws\_event\_overall\_agg\_u partition(dt='2019-06-16')  select  event\_type,  uid,  sum(cnts) as cnts  from  dws\_event\_overall\_agg\_session  where dt='2019-06-16'  group by event\_type,uid  ; |

## ADS事件概况分析报表

### 需求与建模



**建模：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 事件概况统计报表 | | | | | |
| 事件id | 事件名称 | 昨日事件数 | 今日事件数 | 昨日事件人数 | 今日事件人数 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**建表语句：**

|  |
| --- |
| create table ads\_event\_overall(  event\_type string,  cnts\_yesterday int,  cnts\_today int,  users\_yesterday int,  users\_today int  )  stored as parquet; |

### ETL

**计算逻辑：**

1.源表 事件明细用户聚合表：dws\_event\_overall\_agg\_u

2.逻辑： 汇总求和

|  |
| --- |
| insert into table ads\_event\_overall  select  event\_type,  case event\_type  when 'act\_click' then '活动点击'  when 'act\_join' then '活动参与'  when 'act\_register' then '活动注册'  when 'ad\_click' then '广告点击'  when 'ad\_show' then '广告曝光'  when 'addcart' then '添加购物车'  when 'favor' then '收藏'  when 'order\_pay' then '订单支付'  when 'order\_sub' then '订单提交'  when 'pg\_view' then '页面浏览'  when 'rate' then '评分'  when 'search' then '搜索'  when 'startup' then '启动'  when 'thumbup' then '点赞'  else '其他'  end as event\_name,  sum(if(dt='2019-06-15',cnts,0)) as cnts\_yesterday,  sum(if(dt='2019-06-16',cnts,0)) as cnts\_today,  sum(if(dt='2019-06-15',1,0)) as users\_yesterday,  sum(if(dt='2019-06-16',1,0)) as users\_today  from dws\_event\_overall\_agg\_u  where dt='2019-06-16' or dt='2019-06-15'  group by event\_type  ; |

# 【流量数据】路径分析-转化率概念介绍

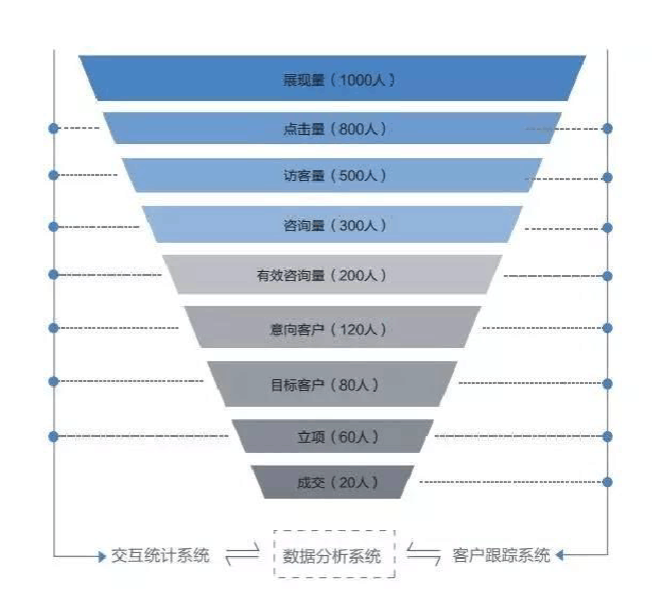
公司有很多很多的各种类型的业务

而每一项业务往往能分成若干个操作环节

用户在业务的各个操作环节上进行操作，一步步走向业务目标（比如买单，比如注册成功，比如充值完成，比如进入充值页）

那么，一个业务的操作环节链条，就叫做这个业务的转化路径！

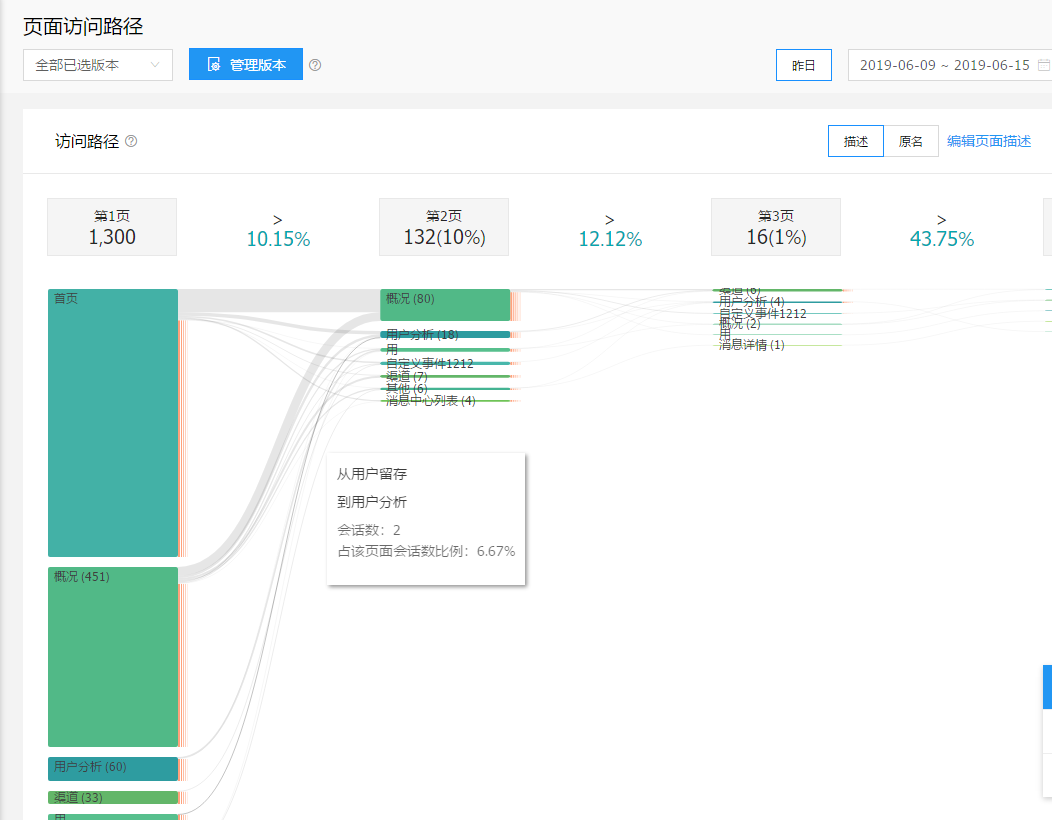
而路径中，每一个环节上的事件发生次数或人数，都会不同，一般是前面的环节上人数多，越往后越少，这样就引出一个概念：**转化率，漏斗模型**





# 【流量数据】路径分析-路径概况统计

## 需求



## 建模

1. 从报表页面原型中，分析出所要计算的数据要素

所访问的页面

第几步访问的

前一步是哪个页面

这是哪个人

这是哪个会话

维度…..

2. 建模结果

所访问的页面

第几步访问的

前一步是哪个页面

这是哪个人

这是哪个会话

### DWS页面访问路径明细表: dws\_acc\_route

记录的是用户的行为轨迹：

用户id

会话id

步骤

事件 前一事件

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| uid | sessionid | 访问步骤序号 | 页面 | 前一页 |

### ADS页面访问路径分析报表：ads\_acc\_routes\_rpts

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 页面 | 步骤序号 | 前一个页面 | 会话数 | 人数 |

## ETL开发

### dws\_acc\_route计算

源表：

事件明细表：dwd\_event\_detail

计算过程：

假设有如下流量请求记录：

|  |
| --- |
| a,ts,s01  a,ts,s01  a,ts,s01  a,ts,s01  a,ts,s01  a,ts,s01    b,ts,s02  b,ts,s02  b,ts,s02  b,ts,s03  b,ts,s03  b,ts,s03  b,ts,s03  b,ts,s03 |

要得到每个人的每次页面请求的顺序号：

按同一个人的同一次会话中请求时间的顺序打上row number

还要得到该次请求的上一个页面：

用lag over窗口函数将上一次请求的页面，拉到当前行

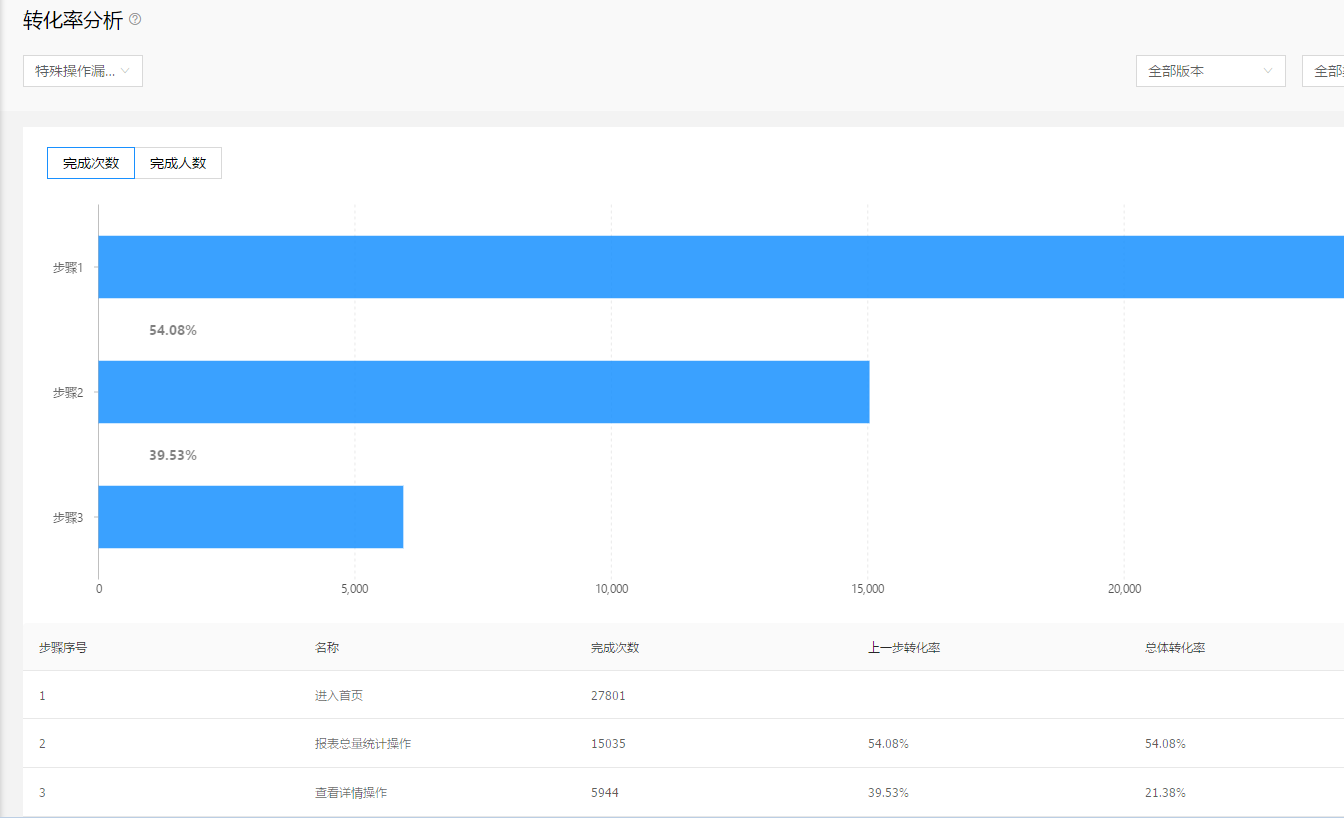
|  |
| --- |
| a,ts,s01,/a ,1 ,null  a,ts,s01,/b ,2 ,/a  a,ts,s01,/c ,3 ,/b  a,ts,s01,/c ,4 ,/c  a,ts,s01,/b ,5 ,/c  a,ts,s01,/d ,6 ,/b    b,ts,s02,/a,1,null  b,ts,s02,/b,2,/a  b,ts,s02,/f,3,/b  b,ts,s03,/a,1,null  b,ts,s03,/b,2,/a  b,ts,s03,/b,3,/b  b,ts,s03,/c,4,/b  b,ts,s03,/d,5,/c |

实现代码：

|  |
| --- |
| */\* 事件分析主题ads层：用户访问路径记录表：dws\_acc\_route  结构：  用户 会话 页面 访问顺序号 前一页面   @Author HUNTER @Date 2019-07-26  @源表：ods\_traffic\_log 0: jdbc:hive2://localhost:10000> select imei,sessionid,event['url'],commit\_time from ods\_traffic\_log t where eventtype='pg\_view' limit 5; +---------+---------------+------------------------+----------------+ | imei | sessionid | \_c2 | commit\_time | +---------+---------------+------------------------+----------------+ | BHGPN7 | hs4NasTIrAAA | http://www.51doit.com/index.html | 1560599141549 | | D9OGFD | OBYsMxtYFcVh | http://www.51doit.com/learn | 1560599141556 | | V6WKTX | ayRyJcjG3e9S | http://www.51doit.com/job | 1560599141573 | | DXJWCX | QCUNyRBAICuQ | http://www.51doit.com/hadoop | 1560599141585 | | VLKQOP | KvpnX3uj72Ao | http://www.51doit.com/spark | 1560599141608 | +---------+---------------+------------------------+----------------+    @目标：dws\_acc\_route  @计算逻辑：  顺序号： 按时间排序标记row nunber  前一页： 按时间排序，取lag over  \*/  -- 建表：* **drop table** if **exists** dws\_acc\_route; **create table** dws\_acc\_route( **uid** string, **sessionid** string, **url** string, **sno int**, **pre\_url** string ) partitioned **by** (dt string) stored **as** parquet ;   *-- etl计算* **insert into table** dws\_acc\_route partition(dt=**'2019-06-16'**) **select imei as** uid, **sessionid**, **event**[**'url'**] **as** url, *row\_number*() over(partition **by** imei,**sessionid order by** commit\_time) **as** sno, *lag*(**event**[**'url'**]) over(partition **by** imei,**sessionid order by** commit\_time) **as** pre\_url  **from** ods\_traffic\_log **where dt**=**'2019-06-16' and eventtype**=**'pg\_view'** ; |

# 【流量数据】路径分析-转化率漏斗

## 需求

 建模和计算思路

基于用户访问路径明细表，可以很容易计算出上述类型的报表：

明细表：

步骤号 所访问的页面 前一个页面 uid sessionid

1 /index null u01 session01

2 /报表总量统计操作 /index u01 session01

1 /index null u02 session02

2 /app分析页 /index u02 session02

核心要点是，理清楚 需求中的每一个业务步骤的定义口径：

什么样的访问（事件）算是完成了业务路径中的什么步骤！

比如，只要访问了 /首页, 则算完成了该业务路径的第一个步骤

比如，访问了 /首页 后，又访问了/报表统计操作 页，则算完成了该业务路径的第二步骤

剩下的工作，就是根据口径来对会话数和人数进行计数！

### 表模型

针对每一种不同的业务，可能会有不同的表结构来支撑：

|  |  |  |  |
| --- | --- | --- | --- |
| 步骤序号 | 事件名称 | 完成次数 | 完成人数 |

### ~~计算思路~~

~~1. 要梳理清楚，需求人员定义的~~**~~业务转化路径~~**~~上的每一个步骤事件~~

~~比如，定一个这样的转化路径：~~

*~~a. 进入首页 （pg\_view: url=”/index.html”）~~*

*~~b. 点击商品类目（pg\_view: url=” /cat1/”）~~*

*~~c. 点击商品详情页（pg\_view: url=” /cat1/xxxxx.html”）~~*

*~~d. 添加购物车（add\_cart）~~*

~~2. 计算逻辑： 统计路径上各种事件的次数和人数~~

~~难点： 如何精确描述路径的操作步骤~~

~~比如，现实中，人们的操作可能有以下两种情况：~~

* ~~（1）不管进入事件的进入顺序，比如：只要发生了b，就算b步骤人数，~~
* ~~（2）先发生a步骤，然后做了一些其他的操作，然后再做了b步骤~~
* ~~（3）先发生了a步骤，然后紧接着就进入b步骤，以此类推~~

~~我们的统计口径中，到底以哪一种为准！~~

~~这个需要跟产品经理或者报表需求方去对接沟通，明确需求！~~

~~我们在此假设，以前（1）种为准！~~

~~这时，就可以设计具体的计算逻辑了：~~

~~判断一个人的一次会话中，~~

~~是否包含a --》 a步骤人数~~

~~是否还包含b --》 b步骤人数~~

~~是否还包含c --》 c步骤人数~~

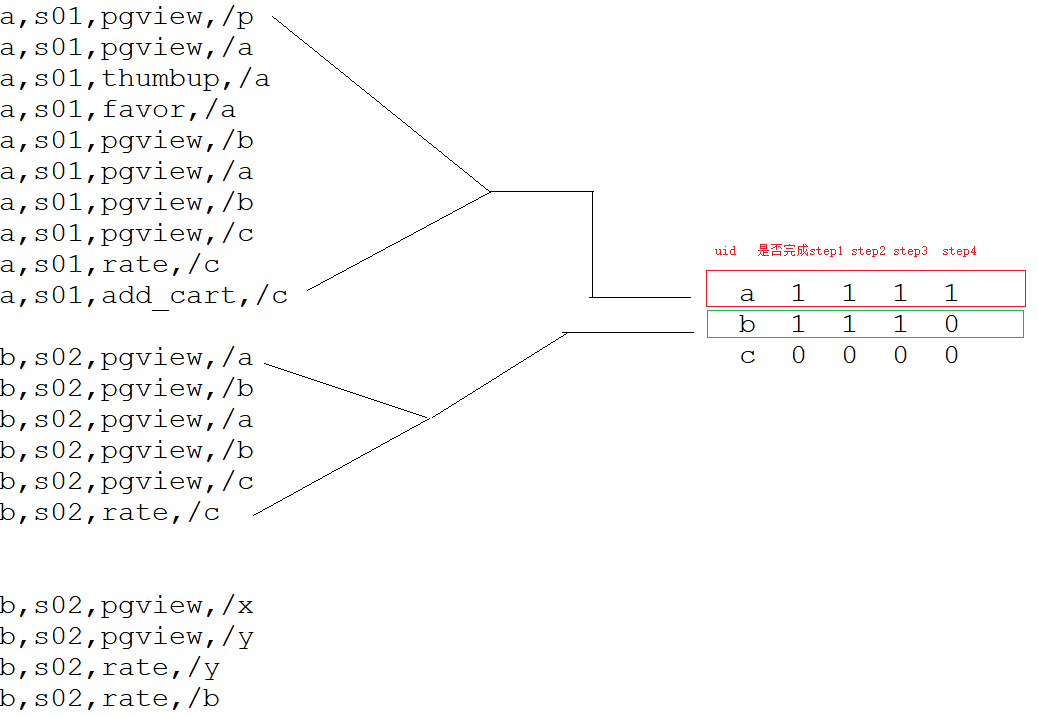
~~是否还包含d --》 d步骤人数~~

~~计算过程：~~

~~1. 过滤掉不包含步骤1的事件~~

~~2. 然后对过滤后的数据，逐步加强约束条件，来计数~~

~~具体来说：~~



## ETL 开发

假设我们要统计的业务路径转化分析需求如下：

|  |
| --- |
| */\* 转化分析主题ads层：会员充值业务路径转化漏斗分析 ads\_member\_encharge\_hopper  结构：  步骤号 步骤名称 完成次数 完成人数   @Author HUNTER @Date 2019-07-26  @源表：dws\_acc\_route 访问路径明细表 @目标：ads\_member\_encharge\_hopper  @计算逻辑：  按路径的定义口径，来统计次数和人数   dws\_acc\_route：访问路径明细记录表 用户 会话 序号 页面 前页 uid sessionid sno url pre\_url u01,s01,1,/index,null u01,s01,2,/member,/index u01,s01,3,/encharge,/member u01,s01,4,/hadoop,/encharge u02,s02,1,/hadoop,null u02,s02,2,/index,/hadoop u02,s02,3,/member,/index u03,s03,1,/index,null u03,s03,2,/member,/index u03,s03,3,/index,/member u03,s03,4,/member,/index u03,s03,5,/spark,/member  drop table if exists dws\_acc\_route\_test; create table dws\_acc\_route\_test( uid string, sessionid string, sno int , url string, pre\_url string ) row format delimited fields terminated by ',' ;  load data local inpath '/root/hivedata/route.dat' into table dws\_acc\_route\_test;   统计口径：  会员充值转化路径  1. 访客进入首页  2. 从任意位置进入/member  3. 从/member进入/encharge  请统计，该业务转化路径上，每个步骤的完成会话数和完成人数 步骤号 步骤名称 完成次数 完成人数 ..... ....... ..... .....   计算步骤： 1, 先根据步骤定义，将每一条记录打上一个是否完成某步骤的标记   u01,s01,1  u01,s01,2  u01,s01,2  u01,s01,3  u01,s02,1  u01,s02,1  u01,s02,2  u02,s03,1  u02,s03,2 2, 将同一个人同一个会话中对某步骤的多次完成，进行去重   u01,s01,1  u01,s01,2  u01,s01,3   u01,s02,1  u01,s02,2   u02,s03,1  u02,s03,2  3, 对上述数据统计会话数和人数即可   如果url是比较真实的情况： http://www.doitedu.cn/encharge?x=98e723&y=124857 可以使用 parse\_url函数来截取其中的页面路径 select parse\_url(url ,'PATH') -- /encharge select parse\_url(url ,'HOST') -- www.doitedu.cn select parse\_url(url ,'QUERY') -- x=98e723&y=124857 \*/*  *-- 建表* **drop table if exists** ads\_member\_encharge\_hopper; **create table** ads\_member\_encharge\_hopper (  **dtstr string**,  **step int**,  **step\_name string**,  **session\_cnts int**,  **user\_cnts int** )  **stored as parquet** ;   *--* **with** tmp **as** (  **select** uid,  sessionid,  **case  when** url = **'/index' then** 1  **when** url = **'/member' then** 2  **when** url = **'/encharge' and** pre\_url = **'/member' then** 3  **else** 0  **end as** step  **from** dws\_acc\_route\_test *-- 访问路径明细记录表* )  **insert into table** ads\_member\_encharge\_hopper **select '2019-06-16' as** dtstr,  step,  **case  when** step = 1 **then '进入首页'  when** step = 2 **then '进入会员页'  when** step = 3 **then '进入充值页'  end as** step\_name,  *count*(1) **as** session\_cnts,  *count*(**distinct** uid) **as** user\_cnts **from** (**select** uid,  sessionid,  step  **from** tmp  **where** step **in** (1, 2, 3)  **group by** uid, sessionid, step  ) o **group by** step ; |

# 【流量数据】广告活动分析-广告概况报表

## 需求

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 广告概况分析报表 | | | | | | | |
| 日期 | 广告id | 曝光次数 | 曝光人数 | 曝光最大页 | 点击次数 | 点击人数 | 点击最大页 |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

## 建模-ETL

### DWD层广告事件明细表建模及ETL

虽然从本报表中，需要的计算元素不包含广告事件的详细内容，但考虑到以后各种各样广告效果统计分析需求，还是决定从长远出发，来建设广告分析主题的分层模型！

首先，建一个DWS层的广告相关事件明细表：dws\_ad\_event\_detail

功能：记录所有跟广告行为有关的事件（广告曝光、广告点击、带广告跟踪码的其他事件）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 广告事件明细表 | | | | |
| uid | 会话id | 事件类型 | 事件时间 | 事件详情 |
|  |  |  |  |  |
|  |  |  |  |  |

建表语句：

|  |
| --- |
| *create table dws\_ad\_event\_detail(*  *uid string,*  *sessionid string,*  *event\_type string,*  *commit\_time bigint,*  *eventmap map<string,string>*  *)*  *partitioned by (dt string)*  *stored as parquet*  *;* |

然后，开发代码，计算出上面的广告相关事件明细表

计算逻辑：

主要是过滤：过滤出ad\_show事件，及其他带ad\_trace\_id码的事件

|  |
| --- |
| insert into table dws\_ad\_event\_detail partition(dt='2019-06-15')  select  uid,  sessionid,  logtype as event\_type,  commit\_time,  eventmap  from dwd\_traffic\_log  where dt='2019-06-15'  and  logtype='ad\_show'  or  (eventmap['ad\_trace\_id'] is not null and trim(eventmap['ad\_trace\_id'])!='')  ; |

### ADS层广告概况报表建模及ETL

根据原型页面的展示需求，梳理出如下表格模型：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 广告概况统计报表 | | | | | | | |
| 日期 | 广告id | 曝光次数 | 曝光人数 | 曝光次数最大页 | 点击次数 | 点击人数 | 点击次数最大页 |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

建表语句：

|  |
| --- |
| create table ads\_ad\_overall(  ad\_id string,  show\_cnts int,  show\_users int,  show\_max\_url string,  click\_cnts int,  click\_users int,  click\_max\_url string  )  partitioned by (dt string)  stored as parquet; |

### 梳理计算要素

从报表中，可以梳理出，在计算过程中所需要的要素如下：

* 日期
* uid
* 页面location
* 广告id
* 事件类型

### 计算逻辑

以页面作为分组条件，来计算，在这个页面上：

页面 某广告 曝光的次数 曝光的人数

/a ad01 100 80

/b ad01 80 60

/b ad02 200 180

那么，要统计某广告的总次数，则按广告分组，把所有页的曝光次数累加！

要得到某广告曝光最多的页面，则按广告分组，求次数最大的页即可

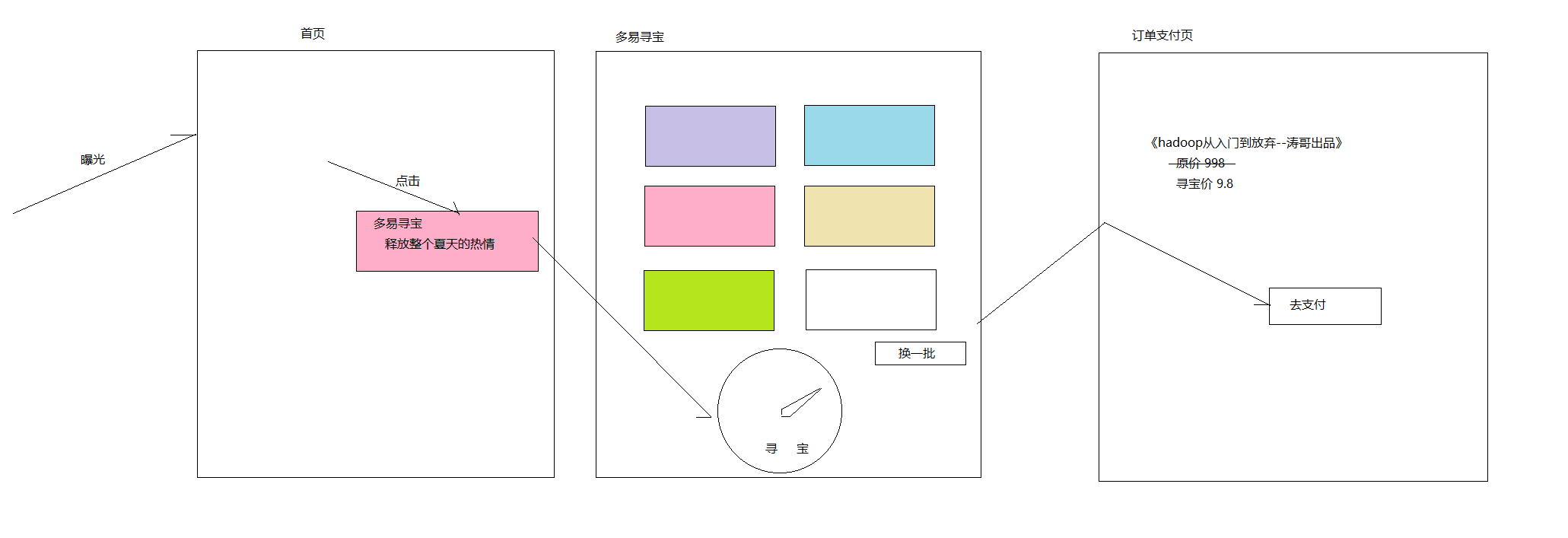
# 【流量数据】广告活动分析-活动概况报表

|  |  |  |  |
| --- | --- | --- | --- |
| 活动概况分析报表 | | | |
| 日期 | 活动id | 活动点击人数 | 活动参与人数 |
|  |  |  |  |
|  |  |  |  |

# 【流量数据】广告活动分析-“DUOYI寻宝”

分析需求：

多易寻宝活动的每一个环节的转化率

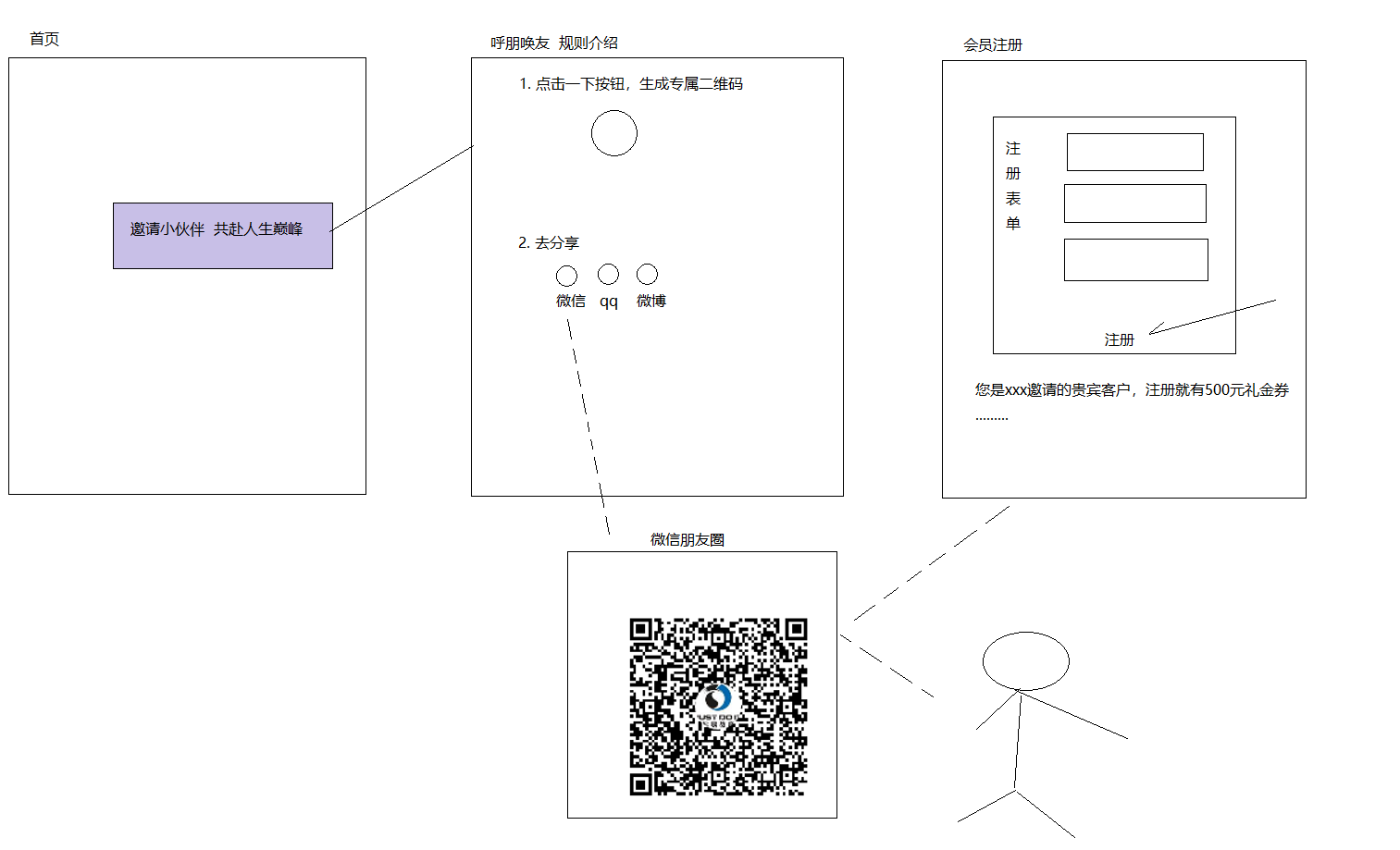


* “多易寻宝”活动，转化路径定义：

路径步骤定义：

1. 第一步 ： 多易寻宝活动广告曝光事件（eventype=’act\_show’, act\_id=’多易寻宝’）
2. 第二步 ： 多易寻宝活动广告点击事件（eventype=’act\_click’, act\_id=’多易寻宝’）
3. 第三步 ： 多易寻宝落地页浏览事件（eventype=’pg\_view’, url=’/act/duoyixunbao’）
4. 第四步 ： 活动参与（eventype=’button\_click’, button\_id=”duoyilunpan”）
5. 第五步 ： 寻宝订单支付页（eventtype=’pg\_view’,url=’/act/duoyixunbao\_pay’）
6. 第六步 ： 去支付点击事件（eventtype=’order\_sub’,url=’/act/duoyixunbao\_pay’）

# 【流量数据】广告活动分析-“呼朋唤友”



* “呼朋唤友”活动，转化路径定义：

1. 用户浏览“呼朋唤友”活动广告（eventtype=act\_show,act\_id=5）
2. 用户点击“呼朋唤友”活动广告（eventtype=act\_click,act\_id=5）
3. 用户点击呼朋唤友二维码生成按钮（eventtype=button\_click,button\_id=hphy\_ewm）
4. 用户点击分享按钮事件（eventtype=button\_click,button\_id=hphy\_fx）
5. 用户打开活动落地页（eventtype=pgview,url=/act/hpyh/register.html）
6. 用户填写表单，点击注册（eventtype=button\_click,button\_id=218349）

# 整体趋势报表



## ADS报表建模

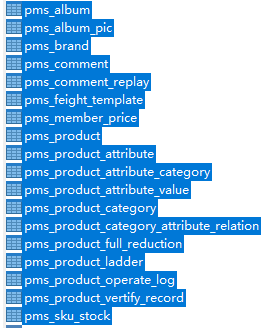
## ADS报表计算

# 【业务数据】业务表模型梳理

## 电商业务库表模型整体架构

业务数据，就是公司的业务系统（javaee系统）在业务数据中产生的数据，对于电商来说，数据主要涉及一下模块：

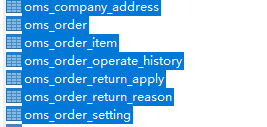
1. 商品信息（主要信息、详情信息、类目信息、属性信息）



1. 用户信息（主要信息、附加信息、会员信息）



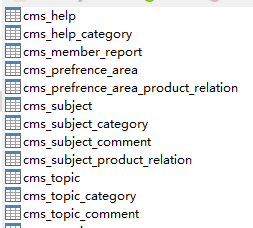
1. 订单信息（主要信息、详情信息、物流信息、评论信息）



1. 购物车信息



1. 内容管理（话题，文章，评论）



1. 营销管理（优惠券、代金券、活动规则、主题推荐）



## 本项目中所需要用到的业务表

### 购物车表

|  |
| --- |
| -- 购物车表ods\_b2c\_cart  drop table if exists ods\_b2c\_cart;  create table ods\_b2c\_cart(  id bigint ,-- ID  session\_id string ,-- session\_id  user\_id bigint ,-- 用户id  goods\_id bigint ,-- 商品ID  number bigint ,-- 商品数量  cancel\_time string ,-- 商品取消时间  submit\_time string ,-- 商品提交时间  dw\_date string ,-- 商品加入时间  )  ----------------------------------------------------------------  1,s01,u001,g013,4,null,null,2019-06-17 17:50:33.768  2,s01,u001,g020,2,null,null,2019-06-17 17:50:50.676  3,s02,u001,g021,2,null,null,2019-06-17 17:50:50.676  4,s12,u002,g018,1,null,null,2019-06-17 18:20:25.005 |

### 商品信息表

|  |
| --- |
| -- 商品信息表 ods\_mall\_goods  drop table if exists ods\_b2c\_goods;  create table ods\_b2c\_goods(  sku\_id bigint ,--SKU编号  sku\_name string ,--SKU名称  goods\_id bigint ,--商品编号 -- spu  goods\_no string ,--商品货号  goods\_sn string ,--商品条码  goods\_name string ,--商品名称  size\_id bigint ,--尺码编号  size\_name string ,--尺码名称  colour\_id bigint ,--颜色ID  shop\_id bigint ,--店铺编号  shop\_name string ,--店铺名称  curr\_price double ,--售卖价格  market\_price double ,--市场价格  discount double ,--折扣比例  cost\_price double ,--成本价格  cost\_type string ,--成本类型  warehouse string ,--所在仓库  stock\_cnt bigint ,--进货数量  stock\_amt double ,--进货货值  first\_cat bigint ,--1级分类ID  first\_cat\_name string ,--1级分类名称  second\_cat bigint ,--2级分类ID  second\_cat\_name string ,--2级分类名称  third\_cat bigint ,--3级分类ID  third\_cat\_name string ,--3级分类名称  dw\_date string  ) |

### 订单信息表

|  |
| --- |
| -- 订单主要信息 ods\_b2c\_orders  drop table if exists ods\_b2c\_orders;  create table ods\_b2c\_orders(  order\_id bigint ,--订单ID  order\_no string ,--订单号  order\_date timestamp ,--订单日期  user\_id bigint ,--用户ID  user\_name string ,--登录名  order\_money double ,--订单金额  order\_type string ,--订单类型  order\_status string ,--订单状态  pay\_type string ,--支付类型  pay\_status string ,--支付状态  order\_source string ,--订单来源  last\_update\_time timestamp ,--订单最后修改时间  dw\_date timestamp  )  partitioned by  (dt string)  ; |

### 订单收货人详情信息

|  |
| --- |
| -- 订单的详细信息 ods\_b2c\_orders\_desc  drop table if exists ods\_b2c\_orders\_desc;  create table ods\_b2c\_orders\_desc(  order\_id bigint ,-- 订单ID  order\_no string ,-- 订单号  consignee string ,-- 收货人姓名  area\_id bigint ,-- 收货人地址ID  area\_name string ,-- 地址ID对应地址段（粒度到县区，有些做得好的做到镇）  address string ,-- 收货人地址（收货人手工填的地址）  mobilephone string ,-- 收货人手机号  telphone string ,-- 收货人电话  coupon\_id bigint ,-- 使用代金券ID  coupon\_money double ,-- 使用代金券金额  carriage\_money double ,-- 运费  create\_time string ,-- 创建时间  last\_update\_time string ,-- 最后修改时间  dw\_date string  )  partitioned by (dt string)  location ''  ; |

### 订单商品详情信息

|  |
| --- |
| drop table if exists ods\_b2c\_orders\_goods;  create table ods\_b2c\_orders\_goods(  order\_id bigint ,-- 订单ID  goods\_id bigint ,-- 商品ID  cat\_id bigint ,-- 类目ID  cat\_name string ,-- 类目名称  size\_id bigint ,-- 条码ID  goods\_price double ,-- 商品价格  goods\_amount bigint ,-- 数量  create\_time string ,-- 创建日期  last\_update\_time string ,-- 最后修改日期  dw\_date string  )  partitioned by (dt string)  location ''  ; |

# 【业务数据】ODS业务数据导入

## 导入策略分析

### 要导入的表

通过对报表统计需求的分析（我们要做的报表主要跟订单金额有关），所以，我们要从业务库中导入的表有如下：

* + ~~用户信息表~~
  + 商品信息表
  + 品类信息表
  + 订单信息表
  + 订单详情表
  + 购物车表

### 导入的策略

导入的策略通常有两种：

全量导入

增量导入！

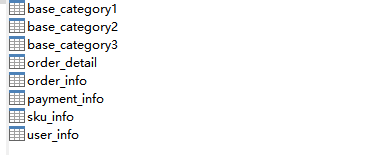
选择的依据：数据变动是否很频繁！数据量是否很大！

变动不频繁，但是还是有变动，而且数据量大，则用增量！

变动频繁，或者数据量不大，干脆全量导入！

* + 用户信息表 -- 全量导入
  + 商品信息表 -- 全量导入
  + 品类信息表 -- 全量导入
  + 订单信息表 -- 增量导入（根据update\_time）
  + 订单详情表 -- 增量导入（根据update\_time）

（增量导入的数据表，为了能够方便查询任何指定日期的业务表状态，可以在数仓中做拉链表）



### 数据导入脚本开发

技术选型：用sqoop来实现！

## 数据迁移知识补充：sqoop

《见sqoop教程》

## ODS用户信息表数据导入

## ODS订单信息表数据导入

# 【业务数据】销量统计分析

GMV：总成交额(数据立方体)

各品类销售额

销售额(品牌、品类、商品)排行榜

## ADS建模

通过对上述报表的分析，发现，其实各个报表都可以统一到一个计算中，如下模型所示：

时间跨度：最近一个月内

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 销售概况统计报表 | | | | | |
| ~~品牌~~ | 1级品类 | 2级品类 | 3级品类 | 商品（spu） | 销售额 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

## DWS建模

对上述ADS层的分析，发现，所需要的维度、事实等各类计算要素，在ODS层中是分散在各个表中的，计算不便。

所以，可以考虑在DW层建一个大宽表，订单记录明细：

# 【业务数据】复购率分析

**主题明细表，数据来源（业务库订单明细表，用户信息表）：**

***用户id，一级品类，二级品类，商品id，成交金额，优惠金额***

**需求：最近30天内的复购率分析**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3级品类 | 购买过该品类的人数 | 购买过2次的人数 | 单次复购率 | 购买过3+次的人数 | 多次复购率 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

# 【业务数据】用户订单画像报表分析（1）

|  |
| --- |
| drop table if exists dm\_user\_order\_tag;  create table dm\_user\_order\_tag(  user\_id bigint ,--用户  first\_order\_time string ,--第一次消费时间  last\_order\_time string ,--最近一次消费时间  first\_order\_ago bigint ,--首单距今时间  last\_order\_ago bigint ,--尾单距今时间  month1\_order\_cnt bigint ,--近30天购买次数  month1\_order\_amt double ,--近30天购买金额  month2\_order\_cnt bigint ,--近60天购买次数  month2\_order\_amt double ,--近60天购买金额  month3\_order\_cnt bigint ,--近90天购买次数  month3\_order\_amt double ,--近90天购买金额  max\_order\_amt double ,--最大消费金额  min\_order\_amt double ,--最小消费金额  total\_order\_cnt bigint ,--累计消费次数（不含退拒）  total\_order\_amt double ,--累计消费金额（不含退拒）  total\_coupon\_amt double ,--累计使用代金券金额  user\_avg\_amt double ,--客单价（含退拒）  month3\_user\_avg\_amt double ,--近90天客单价（含退拒）  common\_address string ,--常用收货地址  common\_paytype string ,--常用支付方式  month1\_cart\_cnt bigint ,--最近30天购物车次数  month1\_cart\_goods\_cnt bigint ,--最近30天购物车商品件数  month1\_cart\_submit\_cnt bigint ,--最近60天购物车次数  month1\_cart\_rate double ,--最近60天购物车商品件数  month1\_cart\_cancel\_cnt bigint ,--最近90天购物车次数  dw\_date string ,--最近90天购物车商品件数  ) partitioned by  (dt string)  ; |

# 【业务数据】用户商品分析（2）

|  |
| --- |
| drop table if exists dm\_user\_goods\_amt;  create table dm\_user\_goods\_amt(  user\_id bigint ,-- 用户  p\_sales\_cnt bigint ,-- 排除退拒销售数量  p\_sales\_amt double ,-- 排除退拒销售金额  p\_sales\_cut\_amt double ,-- 排除退拒销售金额（扣促销减免）  h\_sales\_cnt bigint ,-- 含退拒销售数量  h\_sales\_amt double ,-- 含退拒销售金额  h\_sales\_cut\_amt double ,-- 含退拒销售金额（扣促销减免）  return\_cnt bigint ,-- 退货商品数量  return\_amt double ,-- 退货商品金额  reject\_cnt bigint ,-- 拒收商品数量  reject\_amt double ,-- 拒收商品金额  common\_first\_cat bigint ,-- 最常购买商品一级类目名称  common\_second\_cat bigint ,-- 最常购买商品二级类目名称  common\_third\_cat bigint ,-- 最常购买商品三级类目名称  dw\_date bigint  ) partitioned by (dt string)  stored as parquet  ; |

计算源表： dm\_orders\_goods 订单与商品信息综合宽表

# 附录：知识点汇总

## scala模式匹配（函数）

## sparksql解析复杂json

## 高德地图服务

## gps坐标geohash编码

## HanLP中文分词

HanLP是由一系列模型与算法组成的工具包，目标是普及自然语言处理在生产环境中的应用。HanLP具备功能完善、性能高效、架构清晰、语料时新、可自定义的特点；提供词法分析（中文分词、词性标注、命名实体识别）、句法分析、文本分类和情感分析等功能。

HanLP已经被广泛用于Lucene、Solr、ElasticSearch、Hadoop、Android、Resin等平台，有大量开源作者开发各种插件与拓展，并且被包装或移植到Python、C#、R、JavaScript等语言上去。详见项目主页。

英文的词，是有空格或标点来天然分隔的，很容易实现分词

中文分词，由于中文的语言特点，词和词之间是没有天然分隔的。要想正确分词，需要一些复杂的算法和一些辅助的词典！

比如有一种非常简单的分词算法，就是2字分词法（CJKAnalyzer工具包）：

“我有一头小毛驴我从来也不骑”

=>：我有 有一 一头 头小 小毛 毛驴 驴我 我从 从来

真正在实践中能拿来作为分词工具的有：

庖丁分词

IKAnalyzser

HanLP -- 是一个高级的NLP（自然语言处理）工具套件

HanLp自然语言处理包的使用示例：

导入依赖：

|  |
| --- |
| <**dependency**>  <**groupId**>com.hankcs</**groupId**>  <**artifactId**>hanlp</**artifactId**>  <**version**>portable-1.7.1</**version**> </**dependency**> |

分词api调用：

|  |
| --- |
| **val** terms: util.List[Term] = HanLP.*segment*(**"我有一头小毛驴我从来也不骑，有一天我心血来潮骑着去赶集"**) *println*(terms.map(t => t.*word*).filter(\_.size>1)) |

## hive多重插入语法

从一个表中，写多种查询逻辑，并将结果输出到多个不同表（分区）

语法：  
*from t\_x*

insert into t\_dest partition(p=’p1’)

select ….. where …….

insert into t\_dest partition(p=’p2’)

select ….. where …….

## hive动态分区

指定一个目标分区时，不用写死，而是用查询出来的某个字段的值作为分区值

*注意，有一个分区严格模式开关：关闭严格模式！*

*set hive.exec.dynamic.partition.mode=nonstrict*

*insert into table t\_x partition(dt)*

*select id,name,dt from t\_y;*

*注意，动态分区字段应该作为select中最后一个字段*

## OLAP多维分析

### cube导论

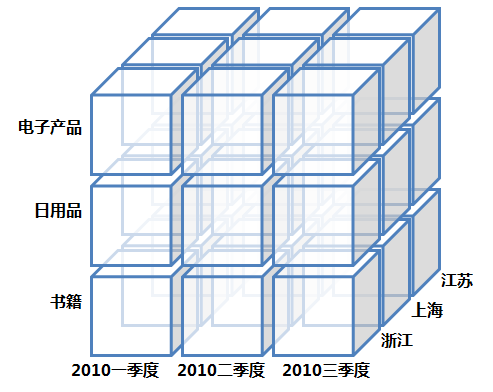
**数据立方体**（Data Cube），是**多维模型**的一个形象的说法。

立方体其本身只有三维，多维模型不仅限于三维模型，可以组合更多的维度

为什么叫数据立方体?

一方面是出于更方便地解释和描述，同时也是给思维成像和想象的空间；

另一方面是为了与传统关系型数据库的二维表区别开来

下图为数据立方体的形象图  


**事实（衡量）：销售额**

其实并不用把cube理解得很高大上,只要理解将衡量指标（事实）**分别**按照不同**维度组合**进行聚合.

hive中也有**cube函数**,可以实现多个任意维度的所有组合情况的统计查询

cube(a,b,c)则：

会对(a,b,c)进行group by，

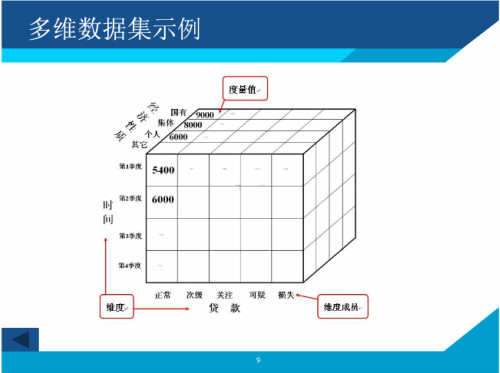
然后依次是(a,b),(a,c),(a),(b,c),(b),(c),

最后在对全表进行group by，

他会统计所选列中值的所有组合的聚合

用cube函数就可以完成所有维度的聚合工作.

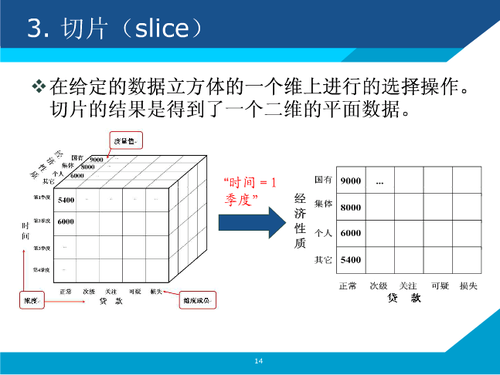
### cube核心操作





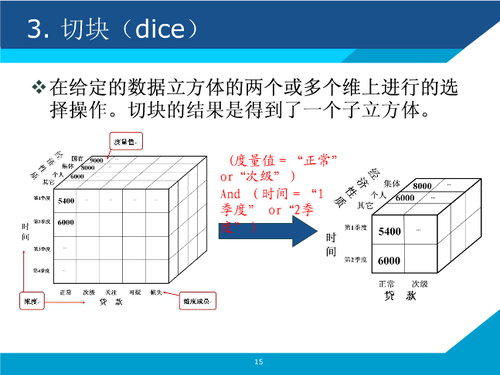
#### SLICE (切片)

将某一个（或多个）维度上的**值锁定**，只观察当这个维度取这个值时的情形，相当于将一个立方体做了一个切片。



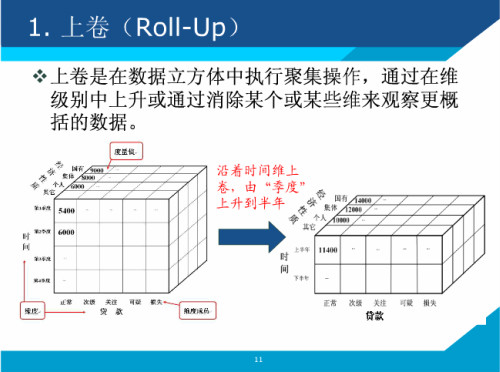
#### DICE (切块）

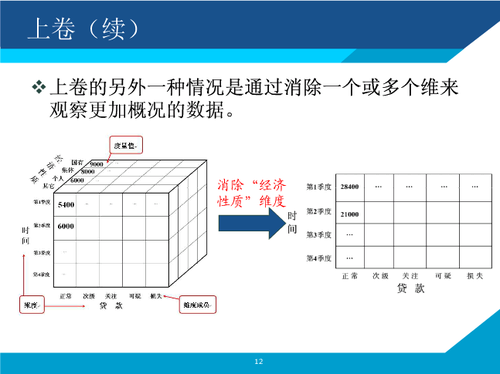
将某一个（或多个）**维度上的值固定在一个区间内**，观察这个取值区间内cube的情形，相当于将一个立方体做了一个切块。



#### ROLL UP (上卷)

沿着某一个（或多个）维度进行聚合，观察聚合后其他维度上的汇总数据，相当于将一个立方体沿着某个维度压缩（聚合）在一起。



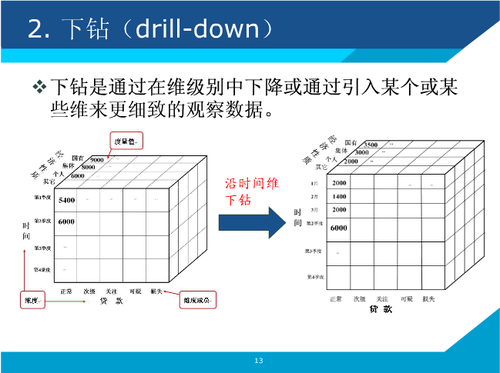


#### DRILL DOWN (下钻）

沿着某一个（或多个）维度在更细粒度层面上进行展开，观察展开后其他维度上的对应数据，相当于将一个立方体沿着某个维度拉伸，拉伸的结果就是粒度变细，比如时间维度从季度拉伸到月。

下钻和上卷是两个相反的操作，取名上并不能很好地顾名思义，简单的解释两个操作就是：在某一个（或多个）维度上是进行更细粒度放大观察还是最粗粒度的聚合观察，就像是图片编辑工具中的

zoom in和zoom out!



#### PIVOT (旋转）

将维度的位置互换。在二维表格中就是行变列，列变行。

## hive 高阶聚合函数

### grouping sets 函数

自由指定要计算的“维度组合”！

|  |  |
| --- | --- |
| **grouping sets语句** | **等价hive** |
| **select device\_id,os\_id,app\_id,count(user\_id)**  **from test\_xinyan\_reg**  **group by device\_id,os\_id,app\_id**  **grouping sets((device\_id))** | SELECT device\_id,null,null,count(user\_id) FROM test\_xinyan\_reg  group by device\_id |
| **select device\_id,os\_id,app\_id,count(user\_id)**  **from test\_xinyan\_reg**  **group by device\_id,os\_id,app\_id**  **grouping sets((device\_id,os\_id))** | SELECT device\_id,os\_id,null,count(user\_id)  FROM test\_xinyan\_reg  group by device\_id,os\_id |
| **select device\_id,os\_id,app\_id,count(user\_id)**  **from test\_xinyan\_reg**  **group by device\_id,os\_id,app\_id**  **grouping sets((device\_id,os\_id),(device\_id))** | SELECT device\_id,os\_id,null,count(user\_id)  FROM test\_xinyan\_reg  group by device\_id,os\_id  UNION ALL  SELECT device\_id,null,null,count(user\_id) FROM test\_xinyan\_reg  group by device\_id |
| **select device\_id,os\_id,app\_id,count(user\_id)**  **from test\_xinyan\_reg**  **group by device\_id,os\_id,app\_id**  **grouping sets((device\_id),(os\_id),(device\_id,os\_id),())** | SELECT device\_id,null,null,count(user\_id) FROM test\_xinyan\_reg  group by device\_id  UNION ALL  SELECT null,os\_id,null,count(user\_id) FROM test\_xinyan\_reg  group by os\_id  UNION ALL  SELECT device\_id,os\_id,null,count(user\_id) FROM test\_xinyan\_reg  group by device\_id,os\_id  UNION ALL  SELECT null,null,null,count(user\_id) FROM test\_xinyan\_reg |

### cube函数

cube简称数据魔方，可以实现hive多个任意维度的查询，cube(a,b,c)则首先会对(a,b,c)进行group by，然后依次是(a,b),(a,c),(a),(b,c),(b),©,最后在对全表进行group by，他会统计所选列中值的所有组合的聚合

**cube即为grouping sets的简化过程函数**

实例：

|  |
| --- |
| select  device\_id,  os\_id,  app\_id,  client\_version, from\_id,  **count(user\_id)**  from test\_xinyan\_reg  group by device\_id,os\_id,app\_id,client\_version,from\_id  with cube; |

等价于：

|  |
| --- |
| SELECT device\_id,null,null,null,null ,count(user\_id) FROM test\_xinyan\_reg group by device\_id  UNION ALL  SELECT null,os\_id,null,null,null ,count(user\_id) FROM test\_xinyan\_reg group by os\_id  UNION ALL  SELECT device\_id,os\_id,null,null,null ,count(user\_id) FROM test\_xinyan\_reg group by device\_id,os\_id  UNION ALL  SELECT null,null,app\_id,null,null ,count(user\_id) FROM test\_xinyan\_reg group by app\_id  UNION ALL  SELECT device\_id,null,app\_id,null,null ,count(user\_id) FROM test\_xinyan\_reg group by device\_id,app\_id  UNION ALL  SELECT null,os\_id,app\_id,null,null ,count(user\_id) FROM test\_xinyan\_reg group by os\_id,app\_id  UNION ALL  SELECT device\_id,os\_id,app\_id,null,null ,count(user\_id) FROM test\_xinyan\_reg group by device\_id,os\_id,app\_id  UNION ALL  SELECT null,null,null,client\_version,null ,count(user\_id) FROM test\_xinyan\_reg group by client\_version  UNION ALL  SELECT device\_id,null,null,client\_version,null ,count(user\_id) FROM test\_xinyan\_reg group by device\_id,client\_version  UNION ALL  SELECT null,os\_id,null,client\_version,null ,count(user\_id) FROM test\_xinyan\_reg group by os\_id,client\_version  UNION ALL  SELECT device\_id,os\_id,null,client\_version,null ,count(user\_id) FROM test\_xinyan\_reg group by device\_id,os\_id,client\_version  UNION ALL  SELECT null,null,app\_id,client\_version,null ,count(user\_id) FROM test\_xinyan\_reg group by app\_id,client\_version  UNION ALL  SELECT device\_id,null,app\_id,client\_version,null ,count(user\_id) FROM test\_xinyan\_reg group by device\_id,app\_id,client\_version  UNION ALL  SELECT null,os\_id,app\_id,client\_version,null ,count(user\_id) FROM test\_xinyan\_reg group by os\_id,app\_id,client\_version  UNION ALL  SELECT device\_id,os\_id,app\_id,client\_version,null ,count(user\_id) FROM test\_xinyan\_reg group by device\_id,os\_id,app\_id,client\_version  UNION ALL  SELECT null,null,null,null,from\_id ,count(user\_id) FROM test\_xinyan\_reg group by from\_id  UNION ALL  SELECT device\_id,null,null,null,from\_id ,count(user\_id) FROM test\_xinyan\_reg group by device\_id,from\_id  UNION ALL  SELECT null,os\_id,null,null,from\_id ,count(user\_id) FROM test\_xinyan\_reg group by os\_id,from\_id  UNION ALL  SELECT device\_id,os\_id,null,null,from\_id ,count(user\_id) FROM test\_xinyan\_reg group by device\_id,os\_id,from\_id  UNION ALL  SELECT null,null,app\_id,null,from\_id ,count(user\_id) FROM test\_xinyan\_reg group by app\_id,from\_id  UNION ALL  SELECT device\_id,null,app\_id,null,from\_id ,count(user\_id) FROM test\_xinyan\_reg group by device\_id,app\_id,from\_id  UNION ALL  SELECT null,os\_id,app\_id,null,from\_id ,count(user\_id) FROM test\_xinyan\_reg group by os\_id,app\_id,from\_id  UNION ALL  SELECT device\_id,os\_id,app\_id,null,from\_id ,count(user\_id) FROM test\_xinyan\_reg group by device\_id,os\_id,app\_id,from\_id  UNION ALL  SELECT null,null,null,client\_version,from\_id ,count(user\_id) FROM test\_xinyan\_reg group by client\_version,from\_id  UNION ALL  SELECT device\_id,null,null,client\_version,from\_id ,count(user\_id) FROM test\_xinyan\_reg group by device\_id,client\_version,from\_id  UNION ALL  SELECT null,os\_id,null,client\_version,from\_id ,count(user\_id) FROM test\_xinyan\_reg group by os\_id,client\_version,from\_id  UNION ALL  SELECT device\_id,os\_id,null,client\_version,from\_id ,count(user\_id) FROM test\_xinyan\_reg group by device\_id,os\_id,client\_version,from\_id  UNION ALL  SELECT null,null,app\_id,client\_version,from\_id ,count(user\_id) FROM test\_xinyan\_reg group by app\_id,client\_version,from\_id  UNION ALL  SELECT device\_id,null,app\_id,client\_version,from\_id ,count(user\_id) FROM test\_xinyan\_reg group by device\_id,app\_id,client\_version,from\_id  UNION ALL  SELECT null,os\_id,app\_id,client\_version,from\_id ,count(user\_id) FROM test\_xinyan\_reg group by os\_id,app\_id,client\_version,from\_id  UNION ALL  SELECT device\_id,os\_id,app\_id,client\_version,from\_id ,count(user\_id) FROM test\_xinyan\_reg group by device\_id,os\_id,app\_id,client\_version,from\_id  UNION ALL  SELECT null,null,null,null,null ,count(user\_id) FROM test\_xinyan\_reg |

### rollup函数

Rollup可以实现将维度从右到左递减多级的统计，显示统计某一层次结构的聚合。

实例：

|  |
| --- |
| select device\_id,os\_id,app\_id,client\_version,from\_id,count(user\_id)  from test\_xinyan\_reg  group by device\_id,os\_id,app\_id,client\_version,from\_id  with rollup; |

等价于：

|  |
| --- |
| select device\_id,os\_id,app\_id,client\_version,from\_id,count(user\_id)  from test\_xinyan\_reg  group by device\_id,os\_id,app\_id,client\_version,from\_id  grouping sets ((device\_id,os\_id,app\_id,client\_version,from\_id),  (device\_id,os\_id,app\_id,client\_version),(device\_id,os\_id,app\_id),(device\_id,os\_id),(device\_id),()); |

## with as 语法

with as语法可以将一个子查询注册为一个临时表，拥有表名

这样，后续的逻辑中，如果需要反复使用这个表数据，就可以通过临时表的表名来指定

with

tmp1 as (select \* from t\_1 where x=8),

tmp2 as (select x,y,f,….. from t\_2 where …… group by …. )

select …… from tmp1 where ……

union

select ……. from tmp2 where ……

## 数仓知识补充：拉链表介绍

拉链表是一种数据模型，需求背景：

1. 数据量比较大;

2. 表中的部分字段会被update,如用户的地址，产品的描述信息，订单的状态等等;

3. 需要查看某一个时间点或者时间段的历史快照信息，比如，查看某一个订单在历史某一个时间点的状态，

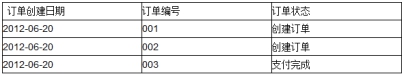
比如，查看某一个用户在过去某一段时间内，更新过几次等等;

4. 变化的比例和频率不是很大，比如，总共有1000万的会员，每天新增和发生变化的有10万左右;

5. 如果对这边表每天都保留一份全量，那么每次全量中会保存很多不变的信息，对存储是极大的浪费;

拉链历史表，既能满足反应数据的历史状态，又可以最大程度的节省存储;

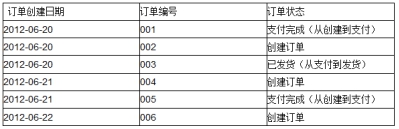
举个简单例子，比如有一张订单表,6月20号有3条记录：



到6月21日，表中有5条记录：



到6月22日，表中有6条记录：

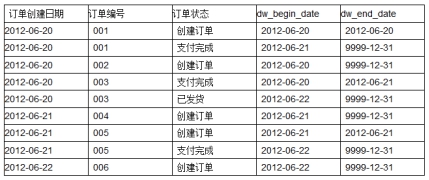


数据仓库中对该表的保留方法：

1. 只保留一份全量，则数据和6月22日的记录一样，如果需要查看6月21日订单001的状态，则无法满足；

2. 每天都保留一份全量，则数据仓库中的该表共有14条记录，但好多记录都是重复保存，没有任务变化，如订单002,004，数据量大了，会造成很大的存储浪费；

如果在数据仓库中设计成历史拉链表保存该表，则会有下面这样一张表：



说明：

1. dw\_begin\_date表示该条记录的生命周期开始时间，dw\_end\_date表示该条记录的生命周期结束时间；

2. dw\_end\_date = '9999-12-31'表示该条记录目前处于有效状态；

3. 如果查询当前所有有效的记录，则select \* from order\_his where dw\_end\_date = '9999-12-31'

4. 如果查询2012-06-21的历史快照，则select \* from order\_his where dw\_begin\_date <= '2012-06-21' and end\_date >= '2012-06-21'，这条语句会查询到以下记录：



和源表在6月21日的记录完全一致：



可以看出，这样的历史拉链表，既能**满足对历史数据的查询需求，又能很大程度的节省存储资源**；

***拉链表的关键意义： 既保存了所有数据的变化过程，又比较节省空间！***

## 数仓知识补充：拉链表实战