大数据项目之企业级数据仓库实战

# 第一章 数据仓库概述

## 数据仓库简介

* + 1. **什么是数据库？**

**数据库**(Database)是按照数据结构来组织、存储和管理数据的建立在计算机存储设备上的仓库。

数据库是长期储存在计算机内、有组织的、可共享的数据集合。数据库中的数据指的是以一定的数据模型组织、描述和储存在一起、具有尽可能小的冗余度、较高的数据独立性和易扩展性的特点并可在一定范围内为多个用户共享。

常用的数据库有 MySQL、ORACLE、SQL Server 等。

## 什么是数据仓库？

数据仓库是**决策支持系统**（dss）和**联机分析应用数据源**的结构化数据环境。数据仓库研究和解决从数据库中获取信息的问题。

#### 数据仓库的特征在于面向主题、集成性、稳定性和时变性，用于支持管理决策。

**数据仓库存在的意义在于对企业的所有数据进行汇总，为企业各个部门提供统一的， 规范的数据出口**。

**面向主题**：数据仓库中的数据是按照一定的主题域进行组织的，每一个主题对应一个宏观的分析领域。数据仓库排除对于决策无用的数据，提供特定主题的简明视图。

**集成的**：企业内不同业务部门数据的完整集成。

对于企业内所有数据的集成要注意一致性（假设财务系统中对于性别使用 F/M，而 OA 系统对性别使用 A/B，这就是数据不一致，如果想搭建企业级的数据仓库，需要数据具有一致性）。

**稳定的**：数仓里不存在数据的更新和删除操作。

**变化的**：数仓里会完整的记录某个对象在一段时期内的变化情况。

#### 数据仓库的目标是实现集成、稳定、反映历史变化有组织有结构的存储数据的集合。

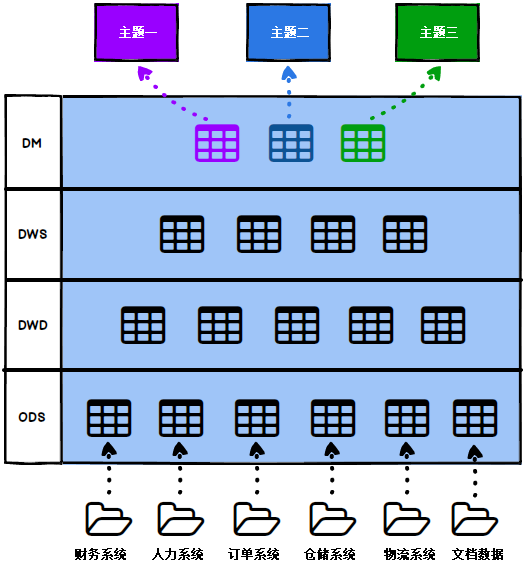


图 1-1 数据仓库基本架构

如上图所示，一个公司可能有多个业务系统，而数据仓库就是将所有的业务系统按照某种组织架构整合起来，形成一个仓储平台，也就是数仓。

ODS—脱敏/清洗—DWD—汇总—DWS—汇总/宽表—DM

三范式（关系型数据库）——目的：让字段拆分开，尽可能实现数据库没有冗余

而数仓会利用冗余换取查询的便利——宽表

## OLTP 与 OLAP

**操作型处理**，叫**联机事务处理 OLTP**（On-Line Transaction Processing，），也可以称面向交易的处理系统，它是针对具体业务在数据库联机的日常操作，通常对少数记录进行查询、修改。用户较为关心操作的响应时间、数据的安全性、完整性和并发支持的用户数等问题。传统的数据库系统作为数据管理的主要手段，主要用于操作型处理。

**分析型处理**，叫**联机分析处理 OLAP**（On-Line Analytical Processing）一般针对某些主题的历史数据进行分析，支持管理决策。

OLTP 与 OLAP 的异同如表 1.1 所示：

表 1-1 OLTP与OLAP的异同

|  |  |
| --- | --- |
| 操作型处理 | 分析型处理 |
| 细节的 | 综合的或提炼的 |
| 实体-关系（ER）模型 | 星型或雪花模型 |
| 存取瞬间数据 | 存储历史数据，不包含最近的数据 |

|  |  |
| --- | --- |
| 可更新的 | 只读，只追加 |
| 一次操作一个单元 | 一次操作一个集合 |
| 性能要求高，响应时间短 | 性能要求宽松 |
| 面向事务 | 面向分析 |
| 一次操作数据量小 | 一次操作数据量大 |
| 支持日常操作 | 支持决策需求 |
| 数据量小 | 数据量大 |
| 客户订单、库存水平和银行账户等 | 客户收益分析、市场细分等 |

## 数据仓库技术架构

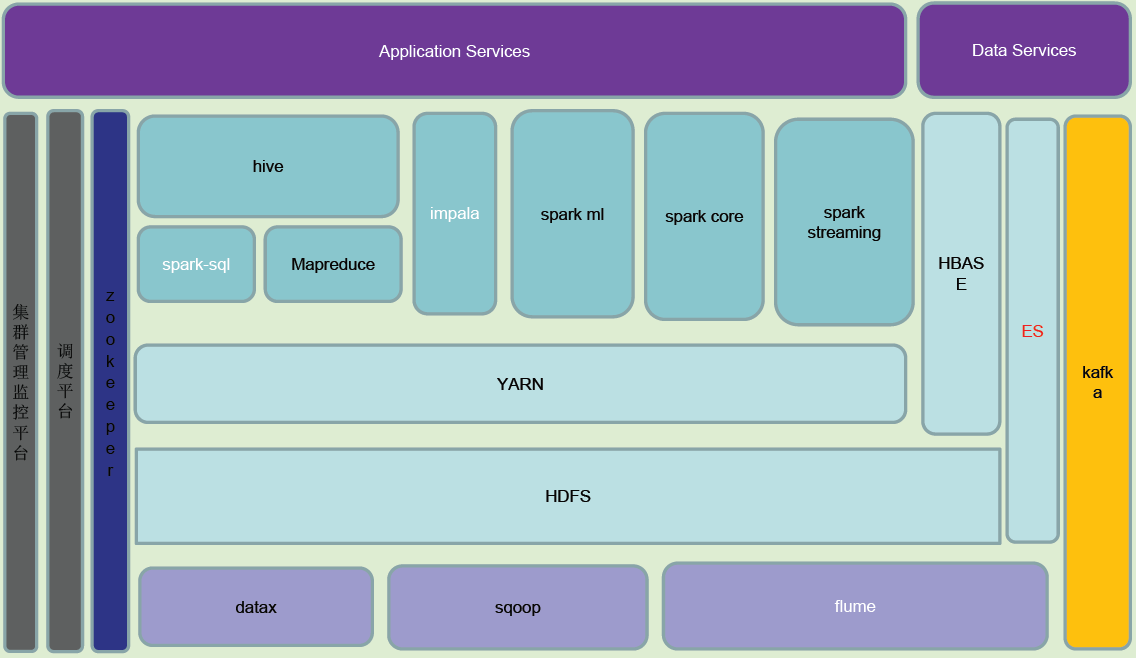


图 1-2 数据仓库技术架构

数据仓库是一门综合性学科，理论与技术并重，不仅有高深的理论概念，而且还涉及大量的大数据相关技术框架，从数据采集到数据通道，从离线处理到实时处理，从集群监控到任务调度，数据仓库无所不含，因此，为了更好地掌握数据仓库技术，我们需要掌握相关的理论概念及技术框架。

sqoop从关系型数据库把数据导入到hdfs（hive hbase）

hbase有事务？ 有 行级事务 一级索引rowkey 二级索引用户自己创建——index表

表级事务，跨表事务，分布式事务。。。都不行。借助协处理器 一条命令需改两张表。 自定义二级索引 凤凰?

企业中的数据——业务数据（用户数据、商品数据、订单数据）——mysql

——用户行为数据 ——hive

## 课程目标

数据仓库对一个企业的所有数据进行整合，为不同业务部门提供统一的数据出口，一个完善合理的数据仓库对于企业整体的数据管理是意义重大的，而数据仓库也是整个大数据系

统中的重要一环，更高层次的数据分析、数据挖掘等工作都会基于数据仓库进行，因此，系

统学习数据仓库的核心概念以及创建方法对于每一个大数据开发工程师都是非常必要的。数据仓库具有非常复杂的理论体系，在数据仓库的创建过程中也会有各种各样的建模思

想以及注意事项，对于数据仓库的整体理论架构和实战思想，我们将会在本课程中进行详细

的剖析和说明。

通过本课程的学习，学生将会更深入的理解数据仓库的概念，并且通过企业实战项目， 具备根据企业实际的业务场景搭建数据仓库的能力。

# 第二章 核心技术框架

## 数据仓库周边技术框架

* + 1. **技术框架概述**

在数据仓库的搭建过程中，会涉及大量技术框架，我们之前已经讨论过数据仓库中的核心技术框架，下面我们会对数据仓库建设过程中的周边技术框架进行说明，常用的周边技术框架入表 2-1 所示：

表 2-1 数据仓库周边技术框架

|  |  |
| --- | --- |
| 功能 | 技术框架 |
| 数据采集 | Datax、Sqoop、Flume、rsyslog、logstash、filebeat 等 |
| 数据通道 | Kafka、RabbitMQ、ZeroMQ 等 |
| 调度 | Oozie、Azkaban、Airflow 等 |

## 数据采集

将目前主要的数据源分为数据库、日志、平面文件（在一些情况下你不能直接取抓取源数据库的数据，只能由源端数据的生成端按照一定格式将数据整理到文件中，然后采集到数据仓库中，总的来说，平面文件就是有一定格式的文件，其实日志也是平面文件的一种）三种，针对每种数据源的特点，采用适当的采集方案，目前常用的主要开源产品如下：

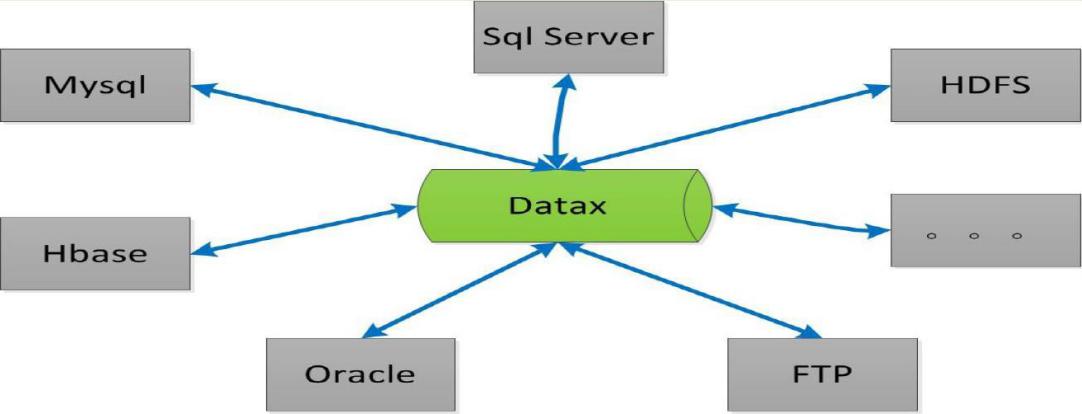
表 2-2 数据采集技术框架

|  |  |
| --- | --- |
| 采集对象 | 框架名称 |
| 数据库 | Datax、Sqoop |

|  |  |
| --- | --- |
| 日志 | Flume、Logstash、Filebeat、Rsyslog |
| 平面文件 | ftp、sftp、scp |

### Datax

Ali 开源的，基于多线程级别的并行实现离线、异构平台同步工具，能够快速实现数据异构数据源的离线同步。



Datax 的架构如下图所示：

图 2-1 DataX 支持的数据类型



Datax 的特点如下：

图 2-2 DataX 架构

* 支持数据实例广泛，灵活的实现各种结构存储之间的数据同步；
* 部署、使用简单；
* 能够实现流量精准控制；
* datax 不是分布式的框架，单节点运行。

### Sqoop

Sqoop，开源的、基于 MapReduce 的、关系型数据库与 HDFS 之间数据同步工具，大数据生态中的关键组件，目前有 Sqoop1（绝大多数公司都使用 Sqoop1）、Sqoop2 两个版本同步更新。

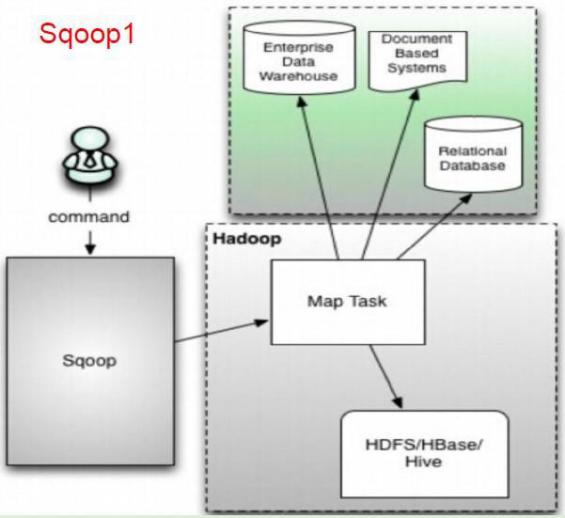


图 2-3 Sqoop 架构

Sqoop1：仅有一个客户端，架构简单明了，部署即用，使用门槛比较低；但是耦合性强， 用户密码暴露导致不安全（通过环境变量可以解决密码暴露的问题）。

Sqoop2：服务端部署、运行，提供 cli、rest api、web ui 等入口，conncetor 集中管理，RDBMS 账号控制更安全，但 Sqoop2 仅负责数据的读、写操作，架构相对复杂。

### Datax 与 Sqoop 的对比

Datax：支持更多的异构同步，多线程而非分布式，易于部署；开源版本仅提供单节点部署方案，任务较多时容易产生性能瓶颈；

Sqoop：支持 RDBMS->HDFS、HDFS->RDBMS 的同步，支持场景有限，基于 MapReduce

的分布式程序，方便弹性扩展；

### 数据采集框架综述

Linux 自带的 scp 是非实时的数据采集。

Rsyslog 性能很高，但是配置很复杂。

Filebeat 轻量级，耗费的资源相对较少，但无法对日志进行解析。

Flume 和 Logstach 都支持实时，但比较耗费资源（相对的，如果服务器资源充足，也没有问题），在 Logstach 中对日志进行解析是非常耗费资源的。

## 数据通道

### Kafka

Kafka 是目前使用最为广泛的消息队列，能够满足绝大部分的生产环境需求，关于 Kafka

的各项特性在之前的基础课程以及项目课程中已经详细讲解过，此处不再赘述。  
1.解耦和 2.消峰

### RabbitMQ

RabbitMQ 是一个由 Erlang 语言开发的 AMQP 的开源实现。

AMQP ：Advanced Message Queue，高级消息队列协议。它是应用层协议的一个开放标

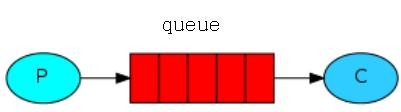
准，为面向消息的中间件设计，基于此协议的客户端与消息中间件可传递消息，并不受产品、

开发语言等条件的限制。

* + - * 1. 消息模型

所有 MQ 产品从模型抽象上来说都是一样的过程：

消费者（consumer）订阅某个队列，生产者（producer）创建消息，然后发布到队列（queue）中，最后将消息发送到监听的消费者。



* + - * 1. RabbitMQ 基本概念

图 2-4 消息模型

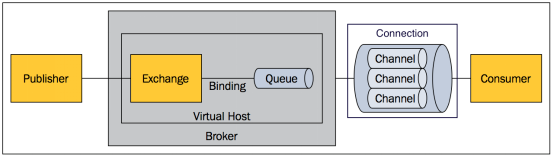


图 2-5 RabbitMQ 内部结构

RabbitMQ 的核心组件如下表所示：

表 2-3 RabbitMQ核心组件

|  |  |
| --- | --- |
| 名称 | 说明 |
| Message | 消息，消息是不具名的，它由消息头和消息体组成。消息体是不透明的，而消息头则由一系列的可选属性组成，这些属性包括routing-key（路由键）、priority（相对于其他消息的优先权）、delivery-mode（指出该消息可能需要持久性存储）等。 |
| Publisher | 消息的生产者，也是一个向交换器发布消息的客户端应用程序。 |
| Exchange | 交换器，用来接收生产者发送的消息并将这些消息路由给服务器中的队列。 |
| Binding | 绑定，用于消息队列和交换器之间的关联。一个绑定就是基于路由键将交换器和消息队列连接起来的路由规则，所以可以将交换器理解成一个由绑定构成的路由表。 |

|  |  |
| --- | --- |
| Queue | 消息队列，用来保存消息直到发送给消费者。它是消息的容器， 也是消息的终点。一个消息可投入一个或多个队列。消息一直在队列里面，等待消费者连接到这个队列将其取走。 |
| Connection | 网络连接，比如一个 TCP 连接。 |
| Channel | 信道，多路复用连接中的一条独立的双向数据流通道。信道是建立在真实的 TCP 连接内的虚拟连接，AMQP 命令都是通过信道发出去的，不管是发布消息、订阅队列还是接收消息，这些动作都是通过信道完成。因为对于操作系统来说建立和销毁 TCP 都是非常昂贵的开销，所以引入了信道的概念，以复用一条 TCP 连接。 |
| Consumer | 消息的消费者，表示一个从消息队列中取得消息的客户端应用程序。 |
| Virtual Host | 虚拟主机，表示一批交换器、消息队列和相关对象。虚拟主机是共享相同的身份认证和加密环境的独立服务器域。每个 vhost 本质上就是一个 mini 版的 RabbitMQ 服务器，拥有自己的队列、交换器、绑定和权限机制。vhost 是 AMQP 概念的基础， 必须在连接时指定，RabbitMQ 默认的 vhost 是 / 。 |
| Broker | 表示消息队列服务器实体。 |

2.2 RabbitMQ 消息路由

生产者把消息发布到 Exchange 上，Binding 决定交换器的消息应该发送到那个队列， 消息最终到达队列并被消费者接收。

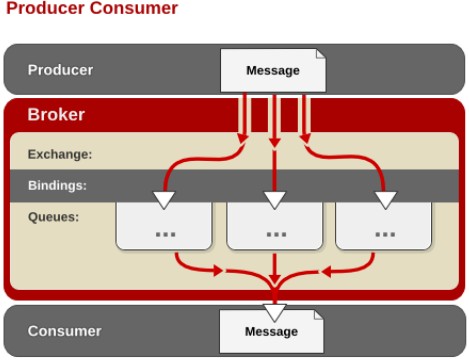


图 2-6 RabbitMQ 消息路由

## 系统调度

### Oozie

Oozie 是一个重量级的任务调度系统，功能全面，但是需要进行大量的基于 XML 的配置，而且代码复杂度比较高。

在实际工厂环境下如果使用 Oozie，复杂的配置会严重降低工作效率，因此往往会在上层封装一层 JavaEE，开发出一套 OM 界面，用户通过填写页面上的选项来完成配置，使得

Oozie 的复杂配置对于用户是透明的。

Oozie 的运行有以下几个级别：

**oozie v1**: workflow，普通工作流，根据作业间的依赖关系构建成有向无环图，和其它的工作流无异；

**oozie v2**：根据 Coordinator 协调工作流的执行，提供时间触发（定时运行）、数据触发

（指定依赖路径，路径中的文件满足条件，任务开始运行）（**可以基于 Hive 元数据触发， 例如 Hive 中的某个表或者某个分区存在了，触发任务**）；

**oozie v3**：根据 Bundle 协调一批 Coordinator 的执行；

#### Oozie 与 hadoop、hive、spark 有较强的版本依赖关系，使用中版本升级等过程需注意

**jar 包冲突**。

* + - 1. Azkaban

Azkaban 是由 Linkedin 开源的一个批量工作流任务调度器。用于在一个工作流内以一个特定的顺序运行一组工作和流程。Azkaban 使用 Properties 文件定义工作流，并提供一个易于使用的 web 用户界面维护和跟踪你的工作流。

Azkaban 是一个轻量级的任务调度器，更适用于中小量任务的调度工作，如果不在意某些功能的缺失，Azkaban 是一个很不错的候选对象。

### Oozie 与 Azkaban 的比较

我们从以下几个方面来对比 Oozie 和 Azkaban：

表 2-4 Oozie与Azkaban的比较

|  |  |  |
| --- | --- | --- |
| 对比项 | Oozie | Azkaban |
| 功能 | 调度 Linux 命令、MapReduce、Spark、Pig、Hive、Java 程序、脚本工作流任务，均可以定时执行工作流任务。 | |
| 工作流定义 | 使用 XML 文件定义工作流 | 使用 Properties 文件定义工作流 |
| 工作流传参 | 支持参数和 EL 表达式，例如  ${fs:dirSize(myInputDir)} | 支持直接传参，例如${input} |
| 定时执行 | 定时执行任务基于时间和输入数据 | 定时执行任务基于时间 |

|  |  |  |
| --- | --- | --- |
| 资源管理 | 暂无严格的权限控制 | 有较严格的权限控制，如用户对工作流进行读/写/执行等操作 |
| 工作流执行 | 作为工作流服务器运行，支持多用户和多工作流 | 有三种运行模式（solo server mode, two server mode, multiple executor  mode），可以根据需要进行配置 |
| 工作流管理 | 支持命令行、HTTP REST、Java API、浏览器操作工作流 | 支持浏览器及 ajax 方式操作工作流 |

在数据仓库的任务调度中，如果表 A 失败，对于 Azkaban，那么所有的任务全部停止， 发送报警消息，对于 Oozie，表 A 失败后立即发送报警消息，只有依赖于这个任务的任务会受到影响，其他的任务不受影响，继续运行，错误修复后，可以再次启动之前报错的任务， 完成任务，单一任务的失败不会影响系统的整体运行。

## Hive 技术框架剖析

**2.2.1 数据存储**

### 行存储

基于 Hadoop 系统行存储结构的优点在于快速数据加载和动态负载的高适应能力，但是， 行存储不支持快速查询，当查询仅仅针对多列表中的少数几列时，它不能跳过不必要的列读取。同时，由于行存储混合着不同数据值的列，行存储不容易获得一个极高的压缩比，即空间利用率不易大幅提高，尽管通过熵编码和利用列相关性能够获得一个较好的压缩比，但是复杂数据存储实现会导致解压开销增大。

### 列存储

列存储在查询时列存储能够避免读不必要的列，并且压缩一个列中的相似数据能够达到较高的压缩比。

### 存储方式的选择

在数据仓库的搭建中，我们选择列存储方式。

对于列式存储范式，有三种存储格式可以选择：rcfile，Orc，Parquet。

#### 若论对 Hive（以 MapReduce 为执行引擎）的支持 Orc 是最好的，但是若论对 Spark 等 Hadoop 生态圈中更多的技术框架，Parquet 的支持是最好的，而 Spark 作为 Hive 的执行引擎时性能非常好，因以我们这里毫无疑问地选择了 Parquet。

* + 1. **数据压缩**

Hive 中可用的压缩格式如下表所示：

表 2-5 Hive压缩格式

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 压缩格式 | 工具 | 算法 | 文件拓展名 | 是否可切分 |
| DEFAULT | 无 | DEFAULT | .delate | 否 |
| gzip | gzip | DEFAULT | .gz | 否 |
| bzip2 | bzip2 | bzip2 | .bz2 | 是 |
| LZO | lzop | LZO | .lzo | 是 |
| Snappy | 无 | Snappy | .nappy | 否 |

由于我们的存储格选用了 Parquet，在压缩格式方面 Parquet 支持 Snappy 和 Gzip，Gzip

的压缩比更高但解压缩的速度较慢，我们选择 Snappy。

## 执行引擎

Hive 的执行引擎包括 MapReduce、Spark [算子]和 Tez。

#### 就性能而言，MapReduce 性能最差， Tez 与 spark 接近，但论社区的活跃程度和技术的成熟度来说 Tez 是不如 Spark 的，因此我们选择 spark。

Spark 目前也有两大分支，即 SparkCore（RDD）和 SparkSQL（DataFrame/DataSet）， 对于这两个分支性能上也有很大差异的。

#### 经过测试，SparkCore 的性能是 MapReduce 的两倍左右，而 SparkSQL 的性能是

**SparkCore 的两倍左右，数据量越大，上述的性能差异会越明显，因此 Hive 数据仓库的执行引擎确定为 SparkSQL**。

* 1. **Hbase 技术框架剖析**

**2.3.1 Hbase 核心架构**

* + 1. Hbase 框架基础

HBase 的数据在存放时会按照Rowkey 进行排序，所以HBase 中的数据都是按照Rowkey

有序的。

HBase 的 Region 采用**先横向拆分再纵向拆分**的方式。

图 2-7 示例表格

所谓先横向，再纵向，就是**先按照行对数据进行划分，再按列对数据进行划分**。

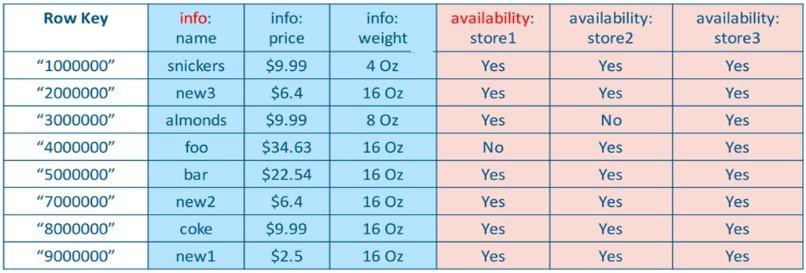


图 2-8 表格横向拆分

按行分隔后，就得到了两个 Region，此时每一个 Region 就需要一个 server 对它进行服务，比如管理和读写服务，也就是 RegionServer（一个 RegionServer 会管理多个 Region）。

图 2-9 Region 与 RegionServer 对应关系

按行拆分后，开始按列拆分，所谓的按列拆分，就是按照列族进行拆分：

图 2-10 表格纵向拆分

HBase 在存储数据的时候，会按照列族去存储数据，每个列族被存到不同的文件中去：

（我们可以将访问频率相近的列放入到同一个列族中）

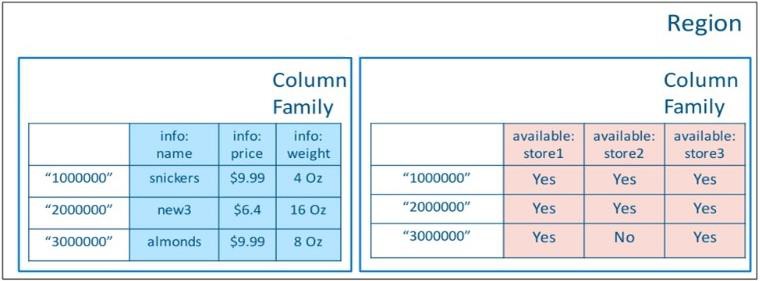


图 2-11 Region 与 Column Family 对应关系

实际的 HDFS 存储文件内容如下图所示：（可以看出 HDFS 中的目录是以列族作为文件夹）

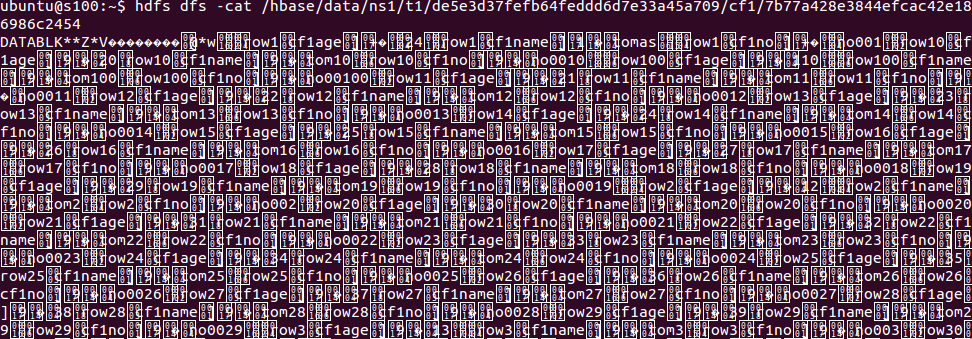


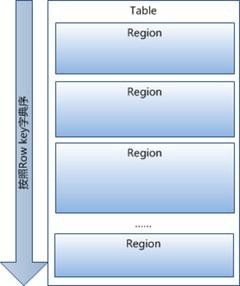
图 2-12 HFile 存储格式

#### 根据上图可知，每一条记录都会记录对应的 RowKey、ColumnFamily、Column，因此， Row Key 设计的长度建议在 16 字节以内，而 ColumnFamily、Column 的长度也应该尽可能的短小，否则会导致存储的数据量剧增，虽然这会使得可读性变差。

* + 1. Hbase 物理存储

① HBase 中，Table 的所有行都按照 RowKey 的字典序排列；

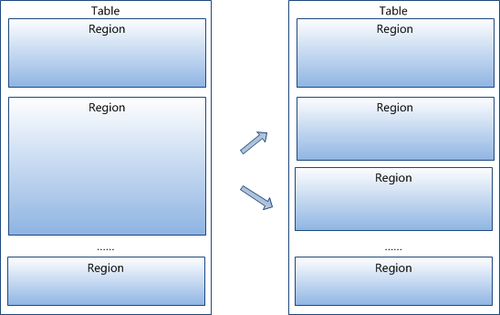
② Table 在行的方向上被分割为多个 Region；

* + - 1. Hbase Region 划分

③ Region 按大小进行分割的，每个表一开始只有一个 Region ，随着数据不断插入表，

Region 不断增大，当增大到一个阀值的时候，Region 会等分会两个新的 Region 。当 table

中的行不断增多，就会有越来越多的 Region 。

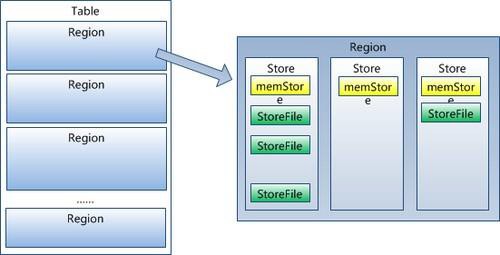


* + - 1. Hbase Region 分裂

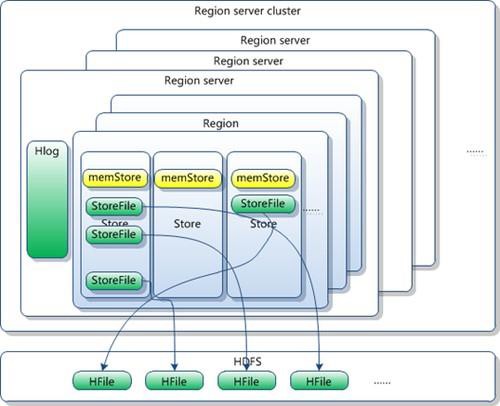
Region 是 HBase 中分布式存储和负载均衡的最小单元。最小单元就表示不同的 Region

可以分布在不同的 HRegionServer 上，但一个 Region 是不会拆分到多个 server 上的。

* HRegion 是 HBase 分布式存储和负载均衡的最小单元（不是存储的最小单元）；
* Region 由一个或者多个 Store （列族）组成，每个 store 保存一个 columns family；
* 每个 Store 又由一个 memStore 和 0 至多个 StoreFile 组成；
* StoreFile 以 HFile 格式保存在 HDFS 上；



* + - 1. Region 内部存储结构



* + - 1. StoreFile 余 Hfile 对应关系
    1. Hbase 用户请求数据流程

HBase 在 0.96 之后去掉了-ROOT- Table，只剩下 Meta Table(hbase:meta)，它存储了集群中所有用户 HRegion 的位置信息。Zookeeper 的节点中（/hbase/meta-region-server）存储的直接是这个 META Table 的位置。

① 从 Zookeeper（/hbase/meta-region-server）中获取 hbase:meta 的位置（HRegionserver 的位置），缓存该位置；

② 从 HRegionserver 中查询用户 Table 对应请求的 RowKey 所在的 HRegionserver，缓存该位置信息；

③ 从查询到 HRegionserver 中读取 Row；

④ 客户会缓存这些位置信息，只是缓存当前 RowKey 对应的 HRegion 的位置；

⑤ 如果下一个要查询的 RowKey 不在同一个 HRegion 中，则需要继续查询 hbase:meta

所在的 HRegion；

⑥ 然而随着时间的推移，客户端缓存的位置信息越来越多，以至于不需要再次查找hbase:meta Table 的信息，除非某个 HRegion 因为宕机或 Split 被移动，此时需要重新查询并更新缓存。

#### META Table 中记录的是那一条数据被放到哪一个Region 中去了，它的存储格式是K-V 的形式，K 是 table，key，region，这里的 key 指的是这个 Region 的第一条数据的 RowKey， 通过两个连续的 Region 的 key，就可以知道前一个 Region 的 RowKey 范围，但是由于只知道范围，你只能确定查找的 RowKey 对应的数据可能在某一个 Region 上，也可能不在， 但是如果不在这个 Region 上也不会在其他的 Region 上。

在设计rowkey的时候，

分区编号：无论谁问你，rowkey的设计，都不能直接回答，你需要问他一个问题：什么场景。用户 rowKey:= （**时间戳年月）.hashcode/6\_a\_列族\_字段\_字段\_**，

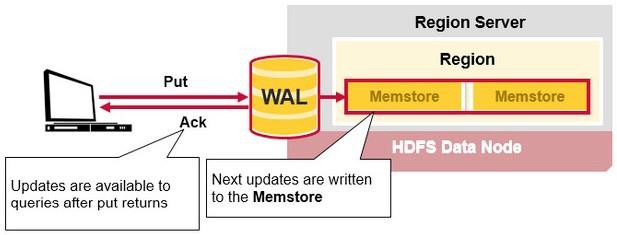
离散：聚合

* + 1. Hbase 数据写入流程

① 当客户端发起 Put 请求时，第一步是将数据写入于写日志，WAL：

* 发布的内容将被添加到存储在磁盘上的 WAL 文件末尾；
* WAL 用于在服务器崩溃的情况下恢复尚未保存的数据。

② 数据写入 WAL 后，被写入 MemStore 中，然后放入 Put 请求确认信息返回客户端。



### Hbase 数据读取流程

2-17 Hbase 数据写入流程

确定数据所在的 Region 后，开始读取指定的数据。

row 对应的 KeyValue cell 可以在多个位置：

① 最近读取的 cell 在 Block Cache 中；

② 最近更新的 cell 在 MemStore 中；

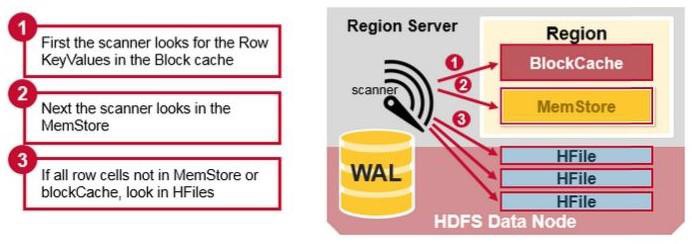
③ Hfile 中；

因此，当读取一行数据时，系统是如何获得相应的 cell 并返回的？读取操作按照以下步骤从 BlockCache，MemStore 和 HFile 合并关键值：

首先，扫描器在 BlockCache（读取缓存）中，查找 Row Cells，最近读取的 Key Values

被缓存在这里，并且当需要内存时，最近最少使用的被清除。

其次，扫描器在 MemStore 中查找，内存写入缓存包含最近的写入。

最后，如果扫描器在 MemStore 和 BlockCache 中没有找到 row cells，然后 HBase 使用Block Cache 的 HFile 索引和布隆过滤器把可能存在目标 row cells 的 HFile 加载到内存中， 并在加载到内存中的 HFile 中进行查找。

2-18 Hbase 数据读取流程

## Hbase Compaction 过程

### minor Compaction

**HBase 会自动选择一些较小的 HFile，并将它们重写成更少且更大的 Hfiles**，这个过程称为 Minor Compaction。Minor Compaction 通过将较小的文件重写为较少但较大的文件来减少存储文件的数量，执行合并排序。

Minor 操作只用来做部分文件的合并操作，不做任何删除数据、多版本数据的清理工作。

Minor 通常会把数个小的相邻的 StoreFile 合并成一个大的 StoreFile，Minor 不会删除标识为删除的数据和过期的数据，只做文件的合并，不做其他事情。

### Major Compaction

#### Major compaction 将 Region 的每一 Store 的所有的 StoreFile 合并，并重写到一个

**StoreFile 中，也就是对 Store 的所有数据进行重写，即每个列族对应这样的一个 StoreFile， 并在此过程中，删除已删除或过期的 Cell（TTL 设置），这样提升了读取性能**。由于 Major

compaction 重写了所有 StoreFile 文件，因此在此过程中可能会发生大量磁盘 I/O 和网络流量， 有较大的性能消耗，这被称为写入放大。

Major compaction 执行计划可以自动运行。由于写入放大，通常计划在周末或晚上进行Major compaction，由于服务器故障或负载平衡，Major compaction 还会使任何远程数据文件成为本地服务器的本地数据文件。

### Compaction 综述

一般情况下都是做 Minor 合并，Major 不少集群都是禁止，然后在集群负载较小时，进行手动 Major 合并。在数据立方公司，也是配置了一个<name>datacube.hregion.majorcompa

ction</name><value>0</value>，这是配置 Major 的合并周期（默认为 7 天），如果配置成 0 即关闭 Major 合并。需要注意，既然 Major 合并是把所有 HFile 都合并成一个文件，可想对集群负载不可小觑。

Minor 则只会选择数个 HFile 文件 compact 为一个 HFile，Minor 的过程一般较快，而且

IO 相对较低。在日常任务时间，都会禁止 Mjaor 操作，只在空闲的时段定时执行。

## Hbase RowKey 设计

HBase 是三维有序存储的，通过 rowkey（行键），column key（column family 和 qualifier） 和 TimeStamp（时间戳）这个三个维度可以对 HBase 中的数据进行快速定位。

HBase 中 rowkey 可以唯一标识一行记录，在 HBase 查询的时候，有以下几种方式：

* + - * 通过 get 方式，指定 rowkey 获取唯一一条记录；
      * 通过 scan 方式，设置 startRow 和 stopRow 参数进行范围匹配；

（另个工具类。）startRowkey endRoekey

* + - * 全表扫描，即直接扫描整张表中所有行记录；

### 1. Rowkey 长度原则

rowkey 是一个二进制码流，可以是任意字符串，最大长度 64kb ，实际应用中一般为

* 1. bytes，以 byte[] 形式保存，一般设计成定长。**建议越短越好，不要超过 16 个字节**，原因如下：
     1. 数据的持久化文件 HFile 中是按照 KeyValue 存储的，如果 rowkey 过长，比如超过

100 字节，1000w 行数据，光 rowkey 就要占用 100\*1000w=10 亿个字节，将近 1G 数据，这样会极大影响 HFile 的存储效率；

* + 1. MemStore 将缓存部分数据到内存，如果 rowkey 字段过长，内存的有效利用率就会降低，系统不能缓存更多的数据，这样会降低检索效率；
    2. 目前操作系统都是 64 位系统，内存 8 字节对齐，控制在 16 个字节，8 字节的整数倍利用了操作系统的最佳特性。

### RowKey 散列原则

必须在设计上保证 RowKey 唯一性，RowKey 是按照字典顺序排序存储的，因此，设计 RowKey 的时候，要充分利用这个排序的特点，将经常读取的数据存储到一块，将最近可能会被访问的数据放到一块。

### RowKey 唯一原则（降维）分区值|年月日|列族|列：10001

### 分区值|年月日|列族|列：10002

### 分区值|年月日|列族|列：19999

必须在设计上保证 RowKey 唯一性，RowKey 是按照字典顺序排序存储的，因此，设计

RowKey 的时候，要充分利用这个排序的特点，将经常读取的数据存储到一块，将最近可能会被访问的数据放到一块。

### 针对热点问题的 RowKey 设计原则

HBase 中的行是按照 RowKey 的字典顺序排序的，这种设计优化了 scan 操作，可以将相关的行以及会被一起读取的行存取在临近位置(分区)，便于 scan。然而糟糕的 Region 设计是热点的源头。 热点发生在大量的 client 直接访问集群的一个或极少数个节点（访问可能是读， 写或者其他操作）。大量访问会使热点 Region 所在的单个机器超出自身承受能力，引起性能下降甚至 Region 不可用，这也会影响同一个 RegionServer 上的其他 Region，由于主机无法服务其他 Region 的请求。 设计良好的数据访问模式以使集群被充分，均衡的利用。

row的离散原理。一台region存储1000万数据。

为了避免写热点，设计 RowKey 使得不同行在同一个 Region，但是在更多数据情况下， 数据应该被写入集群的多个 Region，而不是一个。

下面是一些常见的避免热点的方法以及它们的优缺点：

1. 加盐（rowkey前缀（分区值））

这里所说的加盐不是密码学中的加盐，而是在 rowkey 的前面增加随机数，具体就是给

rowkey 分配一个随机前缀以使得它和之前的 rowkey 的开头不同。分配的前缀种类数量应该和你想使用数据分散到不同的 region 的数量一致。加盐之后的 rowkey 就会根据随机生成的前缀分散到各个 region 上，以避免热点。加随机数可以使数据均匀分布，但是不可重构， 失去 get 快速定位数据的能力。

1. 哈希

哈希会使同一行永远用一个前缀加盐。哈希也可以使负载分散到整个集群，但是读却是可以预测的。使用确定的哈希可以让客户端重构完整的 rowkey，可以使用 get 操作准确获取某一个行数据。

1. 反转

第三种防止热点的方法时反转固定长度或者数字格式的 rowkey。这样可以使得 rowkey 中经常改变的部分（最没有意义的部分）放在前面。这样可以有效的随机 rowkey，但是牺牲了 rowkey 的有序性。

反转 rowkey 的例子以手机号为 rowkey，可以将手机号反转后的字符串作为 rowkey，这样的就避免了以手机号那样比较固定开头导致热点问题。

1. 时间戳反转

一个常见的数据处理问题是快速获取数据的最近版本，使用反转的时间戳作为 RowKey 的一部分对这个问题十分有用，可以用 Long.Max\_Value - timestamp 追加到 key 的末尾（由于 Hbase 数据按照 RowKey 自然排序，timestamp 越大代表数据越新，Long.Max\_Value –

timestamp 就越小，就可以保证最新的数据排在 ），例如 [key][ Long.Max\_Value - timestamp] ,

[key] 的最新值可以通过 scan [key]获得[key]的第一条记录，因为 HBase 中 rowkey 是有序的， 第一条记录是最后录入的数据。

比如需要保存一个用户的操作记录，按照操作时间倒序排序，在设计 RowKey 的时候， 可以这样设计。

[userId 反转][Long.Max\_Value - timestamp]，在查询用户的所有操作记录数据的时候， 直接指定反转后的 userId，startRow 是[userId 反转][000000000000],stopRow 是[userId 反转][Long.Max\_Value - timestamp]。

如果需要查询某段时间的操作记录，startRow 是[user 反转][Long.Max\_Value - 起始时间]，stopRow 是[userId 反转][Long.Max\_Value - 结束时间]。

### 组合 RowKey 设计原则

RowKey 为多个属性拼接而成时，**将具有高标识度的、经常使用检索列的属性放在组合**

#### RowKey 的前面。

1. 应用场景分析
   1. 针对事务数据 RowKey 设计

事务数据是带时间属性的，建议将时间信息存入到 Rowkey 中，这有助于提示查询检索速度。对于事务数据建议缺省就按天为数据建表，这样设计的好处是多方面的。按天分表后， 时间信息就可以去掉日期部分只保留小时分钟毫秒，这样 4 个字节即可搞定。加上散列字段

2 个字节一共 6 个字节即可组成唯一 Rowkey。如下图所示：



图 2-20 事务数据 RowKey 设计

这样的设计从操作系统内存管理层面无法节省开销，因为 64 位操作系统是必须 8 字节对齐，但是对于持久化存储中 Rowkey 部分可以节省 25%的开销。也许有人要问为什么不将时间字段以主机字节序保存，这样它也可以作为散列字段了。这是因为时间范围内的数据还是尽量保证连续，相同时间范围内的数据查找的概率很大，对查询检索有好的效果，因此使用独立的散列字段效果更好，对于某些应用，我们可以考虑利用散列字段全部或者部分来存储某些数据的字段信息，只要保证相同散列值在同一时间（毫秒）唯一。

* 1. 针对统计数据的 RowKey 设计

统计数据也是带时间属性的，统计数据最小单位只会到分钟（到秒预统计就没意义了）。同时对于统计数据我们也缺省采用按天数据分表，这样设计的好处无需多说。按天分表后， 时间信息只需要保留小时分钟，那么 0~1400 只需占用两个字节即可保存时间信息。由于统计数据某些维度数量非常庞大，因此需要 4 个字节作为序列字段，因此将散列字段同时作为序列字段使用也是 6 个字节组成唯一 Rowkey。如下图所示：

图 2-21 统计数据 RowKey 设计

* 1. 针对通用数据的 RowKey 设计

通用数据采用自增序列作为唯一主键，用户可以选择按天建分表也可以选择单表模式。这种模式需要确保同时多个入库加载模块运行时散列字段（序列字段）的唯一性。可以考虑给不同的加载模块赋予唯一因子区别。设计结构如下图所示。



图 2-22 通用数据 RowKey 设计

## Hbase 列族设计

在大多数的工厂环境下，往往只会设计一个列族，以为列族数量过多会导致如下的性能问题：

1. Flush 会产生大量 IO

Flush 的最小单元是 region，也就是说一个 region 中的某个列族做 Flush 操作，其他的列族也会 Flush，对每个列族而言，每次 Flush 都会产生一个文件，频繁 Flush 必然会产生更多的 StoreFile，StoreFile 数量增多又会产生更多的 Compact 操作，Flush 和 Compact 都是很重的 IO 操作。

1. Split 操作可能会导致数据访问性能低下

Split 的最小单元是 region， 如果这个 region 有两个列族 A、B，列族 A 有 100 亿条记

录，列族 B 有 100 条记录，如果最终 Split 成 20 个 region， 那么列族 B 的 100 条记录会分

布到 20 个 region 上， 扫描列族 B 的性能低下。

因此，在设计列族时，过多的列族会导致很多性能问题，列族设计最重要的一点就是减少列族数量。

## Hive on Hbase

HBase 用于在线业务服务，不适合做统计分析。（使用 HBase 进行查询的条件比较苛刻，只能根据 RowKey 去进行查询）

Hive 用于离线分析，适合数据分析，统计。

在 Hbase 的基础课程中，我们在 Hive 中创建表，并将表格关联到 Hbase 中的表格，通过这种方式可以借助 HQL 对 Hbase 中的数据进行分析，但是，在执行分析语句时，会发现

HQL 执行的非常慢。

这是因为虽然我们使用 Hive 创建了一张表，但是这张表并不适用于做统计分析，因为数据都是以 key-value 形式存在的，虽然在 Hive 建立了表，但是本质上数据还是在 HBase 上，你所执行的 Hive 语句（比如 group by 等按列的操作），还是会转化为 HBase 中的操作

（scan 等），其效果与直接在 HBase 里进行相同操作是一样的，都是灾难性的。

因此，**我们一般使用 Hive on HBase 完成数据的加载，也就是关系型数据库、Hive 数据库或者文件中的数据向 HBase 的导入**。我们一般会创建一张映射表，然后往映射表里灌

入数据，后台就会帮我们把数据灌入 HBase 中。

## Hbase 二级索引

目前 HBase 主要应用在结构化和半结构化的大数据存储上，其在插入和读取上都具有极高的性能表现，这与它的数据组织方式有着密切的关系，在逻辑上，**HBase 的表数据按**

#### RowKey 进行字典排序， RowKey 实际上是数据表的一级索引（Primary Index），由于

HBase 本身没有二级索引（Secondary Index）机制，基于索引检索数据只能单纯地依靠

#### RowKey，为了能支持多条件查询，开发者需要将所有可能作为查询条件的字段一一拼接到

**RowKey 中**，这是 HBase 开发中极为常见的做法，但是无论怎样设计，单一 RowKey 固有的局限性决定了它不可能有效地支持多条件查询。

通常来说，**RowKey 只能针对条件中含有其首字段的查询给予令人满意的性能支持**，在查询其他字段时，表现就差强人意了，在极端情况下某些字段的查询性能可能会退化为全表扫描的水平，这是因为**字段在 RowKey 中的地位是不等价的，它们在 RowKey 中的排位决**

**分取值\_列族\_useriD\_zhangsan**

#### 定了它们被检索时的性能表现，排序越靠前的字段在查询中越具有优势，特别是首位字段具有特别的先发优势，如果查询中包含首位字段，检索时就可以通过首位字段的值确定

**RowKey 的前缀部分，从而大幅度地收窄检索区间，如果不包含则只能在全体数据的**

**RowKey 上逐一查找，由此可以想见两者在性能上的差距**。

受限于单一 RowKey 在复杂查询上的局限性，基于二级索引（Secondary Index）的解决

方案成为最受关注的研究方向，并且开源社区已经在这方面已经取得了一定的成果，像

ITHBase、IHBase 以及华为的 hindex 项目，这些产品和框架都按照自己的方式实现了二级索引，各自具有不同的优势，同时也都有一定局限性。

关系型数据库中索引数据与原始数据之间的数据一致性是通过关系型数据库中的组件负责实现的，对于数据库数据的插入和删除都会在索引中展现出来，对于原始数据和索引的操作会是一个原子/事务操作，但是在 HBase 中没有框架自身提供的机制，只能靠开发人员自己去实现。

#### 由于在 HBase 中的二级索引是通过建表的方式实现的，当需要更新时，就是两个表的数据原子更新，也就是跨表的事务功能，而 Hbase 只提供行级事务，没有跨表和跨行的事务功能，这就需要开发者自己去实现，如果对数据一致性要求较高，那么就可能需要自己去实现一套分布式的事务机制，之所以是分布式的事务机制，是因为原始数据可能由一些

HRegionserver 维护，而索引表由另外一些 HRegionserver 维护，这个事务机制就涉及到了多个 HRegionserver，也就是分布式的事务机制。因此，二级索引是 HBase 自身存在的一个短板。

1. 二级索引设计

#### 二级索引的本质就是建立各列值与行键之间的映射关系，以列的值为键，以记录的

**RowKey 为值**。

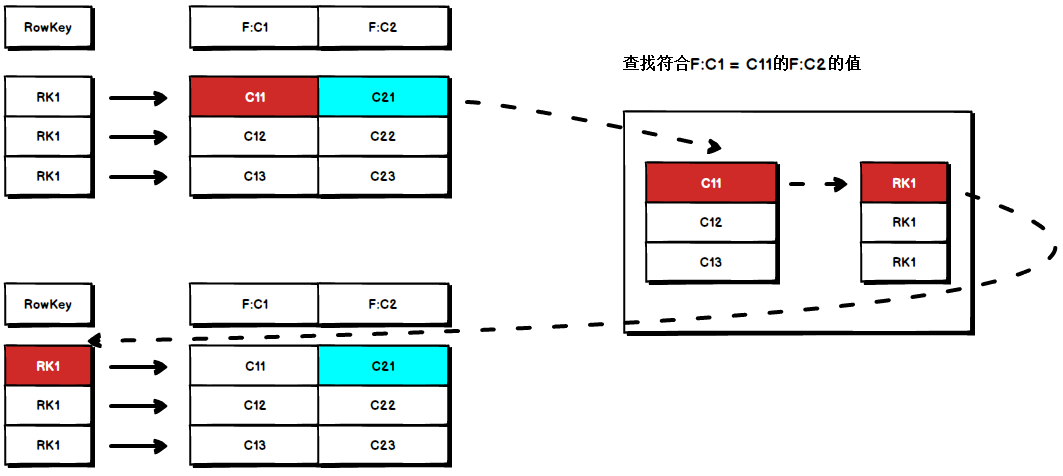


图 2-23 二级索引的建立

如图 2-23 所示，当要对 F:C1 这列建立索引时，只需要建立 F:C1 各列值到其对应行键的映射关系，如 C11->RK1 等，这样就完成了对 F:C1 列值的二级索引的构建，当要查询符合 F:C1=C11 对应的 F:C2 的列值时（即根据 C1=C11 来查询 C2 的值,图 2-23 青色部分）。

其查询步骤如下：

1. 根据 C1=C11 到索引数据中查找其对应的 RK，查询得到其对应的 RK=RK1；
2. 得到 RK1 后就自然能根据 RK1 来查询 C2 的值了 这是构建二级索引大概思路，其他组合查询的联合索引的建立也类似。

### 2. 二级索引设计剖析

“二级多列索引”是针对目标记录的某个或某些列建立的“键-值”数据，**以列的值为**

**键，以记录的 RowKey 为值**，当以这些列为条件进行查询时，引擎可以通过检索相应的“键

-值”数据快速找到目标记录。由于 HBase 本身并没有索引机制，为了确保非侵入性，引擎将索引视为普通数据存放在数据表中，所以，**如何解决索引与主数据的划分存储是引擎第一个需要处理的问题**。

#### 为了能获得最佳的性能表现，我们并没有将主数据和索引分表储存，而是将它们存放在了同一张表里，通过给索引和主数据的 RowKey 添加特别设计的 Hash 前缀，实现了在

**Region 切分时，索引能够跟随其主数据划归到同一 Region 上，即任意 Region 上的主数据其索引也必定驻留在同一 Region 上，这样我们就能把从索引抓取目标主数据的性能损失降低到最小**。

与此同时，**特别设计的 Hash 前缀还在逻辑上把索引与主数据进行了自动的分离，当全体数据按 RowKey 排序时，排在前面的都是索引，我们称之为索引区，排在后面的均为主数据，我们称之为主数据区**。最后，**通过给索引和主数据分配不同的 Column Family，又在物理存储上把它们隔离了起来**。逻辑和物理上的双重隔离避免了将两类数据存放在同一

张表里带来的副作用，防止了它们之间的相互干扰，降低了数据维护的复杂性，可以说这是在性能和可维护性上达到的最佳平衡。

表 2-6 Hbase表格设

startRowKey (0000-a-0102-0000) endRowKey(0000-a-0103-0099)

是不是按照规则查询了100条数据？

10100条数据。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| RowKey | CF:d | | | | CF:i | |
| **索引区** | | | | | | |
| **RegionStartKey-索引名-索引键-索引值** |  | | | | --/-- | |
| 0000-a-0102-0000|25zh78f9（年月） | --/-- | |
| … … | --/-- | |
| |kz23d234 | --/-- | |
| … … | --/-- | |
| 0000-b-0203-0000|25zh78f9 | --/-- | |
| … … | --/-- | |
| 0000-b-0506-0099| kz23d234 | --/-- | |
| **主数据区** | | | | | | |
| **Hash 前缀|原始 ID** | d:q1 | d:q2 | d:q3 | … … | |  | |
| 0000|25zh78f9 | 01 | 02 | 03 | … … | |
| … … | … … | … … | … … | … … | |
| 0099| kz23d234 | 04 | 05 | 06 | … … | |

让我们通过一个示例来详细了解一下二级多列索引表的结构：

假定有一张 Sample 表，使用四位数字构成 Hash 前缀，范围从 0000 到 9999，规划切分

100 个 Region，则 100 个 Region 的 RowKey 区间分别为[0000,0099]，[0100,0199]，……，

[9900,9999]。

以第一个 Region 为例，请看图 2-23，所有数据按 RowKey 进行字典排序，自动分成了索引区和主数据区两段，主数据区的 Column Family 是 d，下辖 q1,q2,q3 等 Qualifier，为了简单起见，我们假定 q1,q2,q3 的值都是由两位数字组成的字符串，索引区的 Column Family 是 i，它不含任何 Qualifier，这是一个典型的“**Dummy Column Family**“，作为区别于 d 的

另一个 Column Family，它的作用就是**让索引独立于主数据单独存储**。

接下来是最重要的部分，即索引和主数据的 RowKey，我们先看主数据的 RowKey，它由四位 Hash 前缀和原始 ID 两部分组成，其中 **Hash 前缀是由引擎分配的一个范围在 0000 到9999 之间的随机值，通过这个随机的Hash 前缀可以让主数据均匀地散列到所有的Region 上**，我们看图 1，因为 Region 1 的 RowKey 区间是[0000,0099]，所以没有任何例外，凡是且

必须是前缀从 0000 到 0099 的主数据都被分配到了 Region 1 上。

接下来看索引的 RowKey，它的结构要相对复杂一些，格式为：**RegionStartKey-索引名**

#### -索引键-索引值，与主数据不同，索引 RowKey 的前缀部分虽然也是由四位数字组成，但却不是随机分配的，而是固定为当前 Region 的 StartKey，这是非常重要而巧妙的设计，一方面，这个值处在 Region 的 RowKey 区间之内，它确保了索引必定跟随其主数据被划分到同一个 Region 里；另一方面，这个值是 RowKey 区间内的最小值，这保证了在同一 Region 里所有索引会集中排在主数据之前。接下来的部分是“索引名”，这是引擎给每类索引添加的一个标识，用于区分不同类型的索引，图 1 中展示了两种索引：a 和 b，索引 a 是为字段 q1 和 q2 设计的两列联合索引，索引 b 是为字段 q2 和 q3 设计的两列联合索引，依次类推，我们可以根据需要设计任意多列的联合索引。再接下来就是索引的键和值了，索引键是由目标记录各对应字段的值组成，而索引值就是这条记录的 RowKey。

现在，假定需要查询满足条件 q1=01 and q2=02 的 Sample 记录，分析查询字段和索引匹配情况可知应使用索引 a，也就是说我们首先确定了索引名，于是在 Region 1 上进行 scan 的区间将从主数据全集收窄至[0000-a, 0000-b)，接着拼接查询字段的值，我们得到了索引键：

0102，scan 区间又进一步收窄为[0000-a-0102, 0000-a-0103)，于是我们可以很快地找到

0000-a-0102-0000|63af51b2 这条索引，进而得到了索引值，也就是目标数据的 RowKey：

0000|63af51b2，通过在 Region 内执行 Get 操作，最终得到了目标数据。需要特别说明的是**这个 Get 操作是在本 Region 上执行的，这和通过 HTable 发出的 Get 有很大的不同，它专门用于获取 Region 的本地数据，其执行效率是非常高的，这也是为什么我们一定要将索引和它的主数据放在同一张表的同一个 Region 上的原因**。

# 第三章 数据仓库基本理论

## 关系模式范式

**核心思想： 你在学习mysql的时候的建表思想：**

**三大范式：**

* + 1. **范式理论概述**

关系型数据库设计时，遵照一定的规范要求，目的在于降低数据的冗余性和数据的一致性，目前业界范式有：第一范式(1NF)、第二范式(2NF)、第三范式(3NF)、巴斯-科德范式

(BCNF)、第四范式(4NF)、第五范式(5NF)。

范式的标准定义是：符合某一种级别的关系模式的集合，表示一个关系内部各属性之间的联系的合理化程度。通俗地讲，**范式可以理解为一张数据表的表结构所符合的某种设计标准的级别**。

使用范式的根本目的是：**减少数据冗余，尽量让每个数据只出现一次，获取数据时通过 join 拼接出最后的数据**。

## 范式基本概念

表 3-1 学生成绩表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **学号** | **姓名** | **系名** | **系主任** | **课名** | **分数** |
| 20170901176 | 王小强 | 计算机系 | 马小腾 | C 语言 | 95 |
| 20170901176 | 王小强 | 计算机系 | 马小腾 | 计算机基础 | 99 |
| 20170901176 | 王小强 | 计算机系 | 马小腾 | 高等数学 | 80 |
| 20170901179 | 李阳 | 经管系 | 王小石 | 经济学 | 95 |
| 20170901179 | 李阳 | 经管系 | 王小石 | 管理学 | 92 |
| 20170901186 | 张小俊 | 数学系 | 钱小森 | 高等数学 | 89 |
| 20170901186 | 张小俊 | 数学系 | 钱小森 | 线性代数 | 96 |

### 函数依赖

#### 若在一张表中，在属性（或属性组）X 的值确定的情况下，必定能确定属性 Y 的值， 那么就可以说 Y 函数依赖于 X，写作 X → Y。

也就是说，在数据表中，如果符合函数依赖，那么不存在任意两条记录，它们在 X 属性（或属性组）上的值相同，而在 Y 属性上的值不同。这也就是“函数依赖”名字的由来，类

似于函数关系 y = f(x)，在 x 的值确定的情况下，y 的值一定是确定的。

例如，对于表 3 中的数据，找不到任何一条记录，它们的学号相同而对应的姓名不同。所以我们可以说姓名函数依赖于学号，写作 学号 → 姓名。但是反过来，因为可能出现同名的学生，所以有可能不同的两条学生记录，它们在姓名上的值相同，但对应的学号不同， 所以我们不能说学号函数依赖于姓名。

表中其他的函数依赖关系还有如： 系名 → 系主任

学号 → 系主任

（学号，课名） → 分数

以下函数依赖关系则不成立： 学号 → 课名

学号 → 分数课名 → 系主任

（学号，课名） → 姓名

### 完全函数依赖

#### 在一张表中，若 X → Y，且对于 X 的任何一个真子集（假如属性组 X 包含超过一个属性的话），X ' → Y 不成立，那么我们称 Y 对于 X 完全函数依赖，记做：



例如：

学号 F→ 姓名

（学号，课名） F→ 分数 （注：因为同一个的学号对应的分数不确定，同一个课名 对应的分数也不确定）

### 部分函数依赖

#### 假如 Y 函数依赖于 X，但同时 Y 并不完全函数依赖于 X，那么我们就称 Y 部分函数依赖于 X，记做：



简单来说，（学号，课名）→ 系名，学号 → 系名，那么（学号，课名）p→ 系名。

### 传递函数依赖

#### 假如 Z 函数依赖于 Y，且 Y 函数依赖于 X ，且 Y 不包含于 X，X 不函数依赖于 Y，那么我们就称 Z 传递函数依赖于 X，记做：

简单来说，系名 → 系主任，学号 → 系名，那么学号 T→ 系主任。

## 一范式

一范式（1NF）：**域应该是原子性的，即数据库表的每一列都是不可分割的原子数据项**。  
域：域就是列的取值范围，比如性别的域就是（男，女）

表 3-2 不符合一范式的表格设计

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **商品** | **商家ID** | **用户ID** |
| **001** | 5 台电脑 | XXX旗舰店 | 00001 |

很明显如图 3-1 所示的表格设计是不符合第一范式的，商品列中的数据不是原子数据项，是可以进行分割的，因此对表格进行修改，让表格符合第一范式的要求，修改结果如图3-2 所示：

表 3-3 符合一范式的表格设计

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **商品** | **数量** | **商家ID** | **用户ID** |
| **001** | 电脑 | 5 | XXX旗舰店 | 00001 |

实际上， **1NF 是所有关系型数据库的最基本要求**， 你在关系型数据库管理系统

（RDBMS），例如 SQL Server，Oracle，MySQL 中创建数据表的时候，**如果数据表的设计**

**不符合这个最基本的要求，那么操作一定是不能成功的**。也就是说，只要在 RDBMS 中已经存在的数据表，一定是符合 1NF 的。

## 二范式

#### 二范式（2NF）：在 1NF 的基础上，实体的属性完全函数依赖于主关键字（混合主键）， 不能存在部分函数依赖于主关键字（混合主键）。

**如果存在某些属性只依赖混合主键中的部分属性，那么不符合二范式**。

表 3-4 不符合二范式的表格设计

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **学生 ID** | **姓名** | **所属系** | **系主任** | **所修课程** | **分数** |
| 20170901176 | 王小强 | 计算机系 | 马小腾 | 000001 | 95 |
| 20170901176 | 王小强 | 计算机系 | 马小腾 | 000002 | 99 |

上述表格中是混合主键（学生 ID + 所修课程），但是所属系和系主任这两个属性只依赖于混合主键中的学生 ID 这一个属性，因此，不符合第二范式。

如果有一天学生的所属系要调整，那么所属系和系主任这两列都需要修改，如果这个学生修了多门课程，那么表中的多行数据都要修改，这是非常麻烦的，不符合第二范式。

为了消除这种部分依赖，只有一个办法，就是**将大数据表拆分成两个或者更多个更小**

#### 的数据表。

表 3-5 符合二范式的表格设计（1）

|  |  |  |
| --- | --- | --- |
| **学生 ID** | **所修课程** | **分数** |
| 20170901176 | 000001 | 95 |
| 20170901176 | 000002 | 99 |

|  |  |  |
| --- | --- | --- |
| **系 ID** | **所属系** | **系主任** |
| 000001 | 计算机系 | 马小腾 |
| 000002 | 计算机系 | 马小腾 |

|  |  |
| --- | --- |
| **学生 ID** | **姓名** |
| 20170901176 | 王小强 |
| 20170901176 | 王小强 |

通过上述的修改，当一个学生的所属系需要调整时，不管学生修了多少门课程，都只需要改变表 3-5 中的一行数据即可。

## 三范式

#### 3NF 在 2NF 的基础之上，消除了非主属性对于主键（复合主键）的传递依赖。

表 3-6 不符合三范式的表格设计

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **订单ID** | **商品ID** | **商品颜色** | **商品尺寸** | **商家ID** | **用户ID** |
| **001** | 0001 | 深空灰 | 300\*270\*40 | XXX旗舰店 | 00001 |

很明显，表 3-7 中，商品颜色依赖于商品 ID，商品 ID 依赖于订单 ID，那么非主属性商品颜色就传递依赖于订单 ID，因此不符合三范式，解决方案是**将大数据表拆分成两个或者更多个更小的数据表**。

表 3-7 符合三范式的表格设计（1）

|  |  |  |  |
| --- | --- | --- | --- |
| **订单ID** | **商品ID** | **商家ID** | **用户ID** |
| **001** | 0001 | XXX旗舰店 | 00001 |

|  |  |  |
| --- | --- | --- |
| **商品ID** | **商品颜色** | **商品尺寸** |
| **0001** | 深空灰 | 300\*270\*40 |

## 数据仓库建模基本理论

* + 1. **数据仓库建模目标**

#### 数据仓库建模的目标是通过建模的方法更好的组织、存储数据，以便在性能、成本、效率和数据质量之间找到最佳平衡点。

**访问性能**：能够快速查询所需的数据，减少数据 I/O；DM集市（hbase ES）

**数据成本**：减少不必要的数据冗余，实现计算结果数据复用，降低大数据系统中的数据成本和计算成本；

**使用效率**：改善用户应用体验，提高使用数据的效率；

**数据质量**：整合所有数据源的数据，改善数据统计口径的不一致性，减少数据计算错误的可能性，提供高质量的、一致的数据访问平台。

上述的四点之间是存在冲突的，为了提高访问性能，可能会提高数据冗余（减少 Join）， 这样会降低计算成本，但是会导致数据存储成本很高，并且由于数据的冗余，会提高数据统计口径不一致的风险，我们的目的是通过合理的设计在性能、成本、效率和数据质量之间找到平衡点。

## ER 实体模型

### 基本理论

ER 模型是数据库设计的理论基础，当前几乎所有的 OLTP 系统设计都采用 ER 模型建模的方式。

#### 在信息系统中，将事物抽象为“实体”、“属性”、“关系”来表示数据关联和事物描述；其中，实体：Entity，关系：Relationship，这种对数据的抽象建模通常被称为 ER 实体关系模型。

**实体**：**通常为参与到过程中的主体，客观存在的**，比如商品、仓库、货位、汽车，此实体非数据库的实体表；

**属性**：**对主体的描述、修饰即为属性**，比如商品的属性有商品名称、颜色、尺寸、重量、产地等；

**关系**：**现实的物理事件是依附于实体的**，比如商品入库事件，依附实体商品、货位， 就会有“库存”的属性产生；用户购买商品，依附实体用户、商品，就会有“购买数量”、

“金额”的属性产品。

### 实体之间的对照关系

实体之间建立关系时，存在对照关系：

1:1，即 1 对 1 的关系，比如实体人、身份证，一个人有且仅有一个身份证号；（A->B： 相互完全依赖，知道 A 一定确定 B，知道 B 一定确定 A）。

#### （动静分离：在数据库设计时，会将动态属性（年龄、地址、偏好、...）和静态属性（姓名、性别、身份证号、...）进行分离，剥离为两张表，一张父表，一张子表，从而提高性能）。

1:n，即 1 对多的关系，比如实体学生、班级，对于某 1 个学生，仅属于 1 个班级，而在 1 个班级中，可以有多个学生；（一张学生表，一张班级表，通过班级 ID 这个外键进行关联）。

n:m，即多对多的关系，比如实体学生、课程，每个学生可以选修多门课程，同样每个课程也可以被多门学生选修；（一张学生表，一张课程表，一张选课表）。

### ER 建模的图形表示

在日常建模过程中：

“实体”：使用矩形表示；

“关系”：使用菱形表示；

“属性”：使用椭圆形表示；

所以 ER 实体关系模型也称作 E-R 关系图。

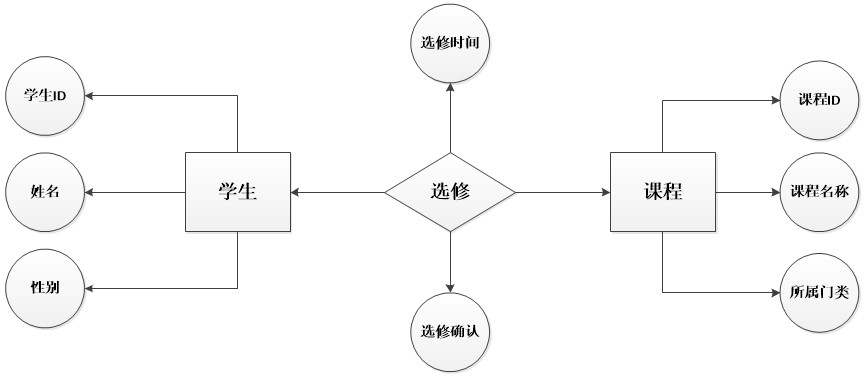
## ER 实体建模实例

### 场景

学生选课系统，该系统主要用来管理学生和选修课程，其中包括课程选修、学生管理功能，现需要完成数据库逻辑模型设计。

### 实现步骤

1. 抽象出主体 —— 学生，课程；
2. 梳理主体之间的关系 —— 选修；（学生与选修课程是一个多对多的关系）
3. 梳理主体的属性；
4. 画出 E-R 关系图；

图 3-4 ER 关系图

#### 使用 ER 模型构建数据仓库的成功率是比较低的，因为涉及到“抽象出实体”这个过程， 这就涉及到将企业所有业务系统中的所有实体都抽象出来，这需要先梳理出所有的业务系 统实体，再梳理实体之间的关系，这是一个非常复杂的过程，可能你把公司所有的实体梳 理清楚了，可能业务又要调整了。

* + 1. **维度模型**

维度建模的理论由 Ralph Kimball 提出，他提出将数据仓库中的表划分为**事实表**和**维度表**两种类型。

#### 维度建模源自数据集市，主要面向分析场景。

“事实表”，**用来存储事实的度量（measure）及指向各个维的外键值**。“维度表”， **用来保存该维的元数据，即维的描述信息，包括维的层次及成员类别等**。

简单的说，维度表就是你观察该事物的角度（维度)，事实表就是你要关注的内容。例如用户使用滴滴打车，那么打车这件事就可以转化为一个事实表，即打车订单事实表，然后用户对应一张用户维度表，司机对应一张司机维度表。



### 1. 事实表

图 3-5 维度模型

#### 在现实世界中，每一个操作型事件，基本都是发生在实体之间的，伴随着这种操作事件的发生，会产生可度量的值，而这个过程就产生了一个事实表，存储了每一个可度量的事件。

**发生在现实世界中的操作性事件所产生的可度量数值，存储在事实表中**。从最低的粒

度级别来看，事实表行对应一个度量事件，反之亦然。因此，事实表的设计完全依赖于物理活动，不受可能产生的最终报表的影响。**除数字度量外，事实表总是包含外键，用于关联与之相关的维度，也包含可选的退化维度键和日期/时间戳**。查询请求的主要目标是基于事

实表展开计算和聚集操作。

事实表往往包含三个重要元素：

#### 维度表外键

1. **度量数据**
2. **事件描述信息**

例如在电商场景中的一次购买事件，涉及主体包括客户、商品、商家，产生的可度量值包括商品数量、金额、件数等。



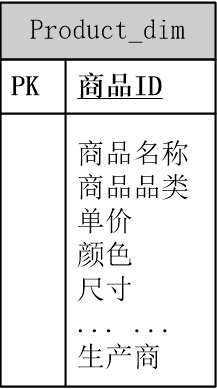
### 2. 维度表

图 3-6 订单事实表

**每个维度表都包含单一的主键列**。**维度表的主键可以作为与之关联的任何事实表的外键，当然，维度表行的描述环境应与事实表行完全对应**。维度表通常比较宽，是扁平型非

规范表，包含大量的低粒度的文本属性。

比如商品，单一主键为商品 ID，属性包括产地、颜色、材质、尺寸、单价等，但并非属性一定是文本，比如单价、尺寸，均为数值型描述性的，日常主要的维度抽象包括：时间维度表、地理区域维度表等。

图 3-7 商品维度表

综上所述，如果针对用户的下单行为（单一商品）进行维度建模，我们可以得到如下模型：

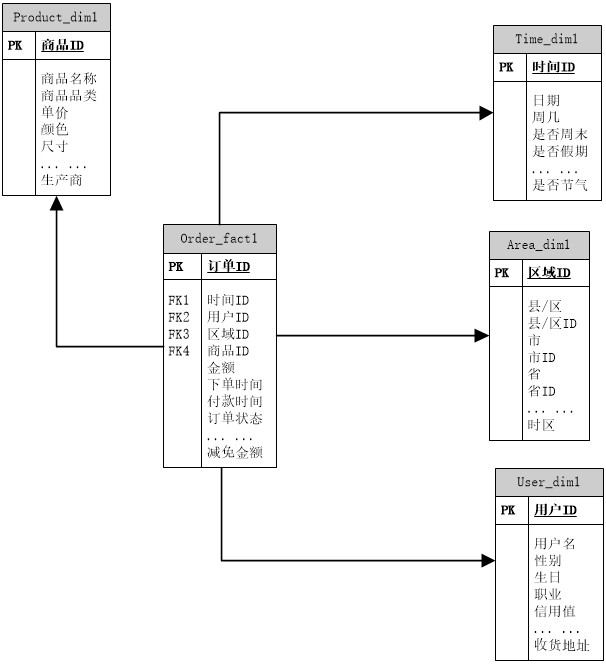


图 3-8 维度建模实例

## 维度建模实例

### 背景

某电商平台，经常需要对订单进行分析，以某宝的购物订单为例，以维度建模的方式设计该模型。

### 事实表与维度表的划分

事实表为订单表、子订单表，维度包括商品维度、用户维度、商家维度、区域维度、时

间维度。

### 事实表分析

**订单中包含的度量**：商品件数、总金额、总减免金额； **描述性属性**：下单时间、付款时间、订单状态等；

**子订单包含度量**：商品 ID、单价、减免金额；

**描述性属性**：加入购物车时间、下单时间、付款时间、状态；

### 维度表分析

**商品维度**：商品 ID、商品名称、商品品类、单价、颜色、尺寸、生产商等； **用户维度**：用户 ID、姓名、性别、生日、职业、信用值、收货地址等；

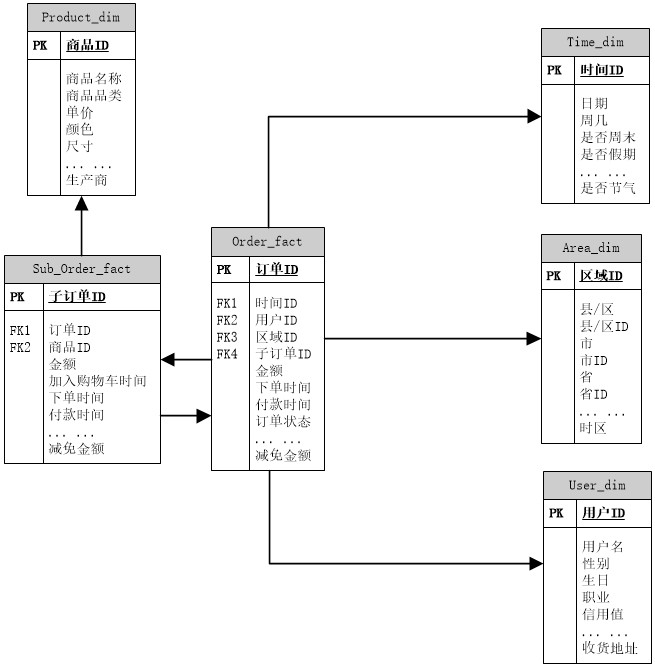
**时间维度**：日期 ID、日期、周几、是否周末、是否假期等； **区域维度**：区域 ID，县/区、县/区 ID、市、市 ID、省、省 ID；

图 3-9 维度建模实例

订单表和子订单表是两张事实表，**我们往往会避免事实表之间产生关系**，因此会考虑 让子订单表中有一定的数据冗余，比如订单表和订单明细表都各自记录着用户 ID，区域 ID， 时间 ID 等，这样在查询子订单信息时，就不用通过关联订单表来获取上述数据了，但是子订单表中仍会保存订单 ID 字段。

## 建模方法总结

ER 模型以及维度模型是当前主流的建模方法。

**ER 模型常用于 OLTP 数据库建模，应用到构建数仓时更偏重数据整合**，站在企业整体考虑，将各个系统的数据按相似性、一致性合并处理，为数据分析、决策服务，但并**不便于直接用来支持分析**。

ER 模型的特点如下：

* 需要全面梳理企业所有的业务和数据流；
* 实施周期长；
* 对建模人员要求高；

**维度建模是面向分析场景而生，针对分析场景构建数仓模型**；重点关注快速、灵活的解决分析需求，同时能够提供大规模数据的快速响应性能。**针对性强，主要应用于数据仓库构建和 OLAP 引擎低层数据模型**。

* 不需要完整的梳理企业业务流程和数据；
* 实施周期根据主题边界而定，容易快速实现 demo ；

## 3.2 星型模式、雪花模式与星座模式

* + 1. **星型模式**

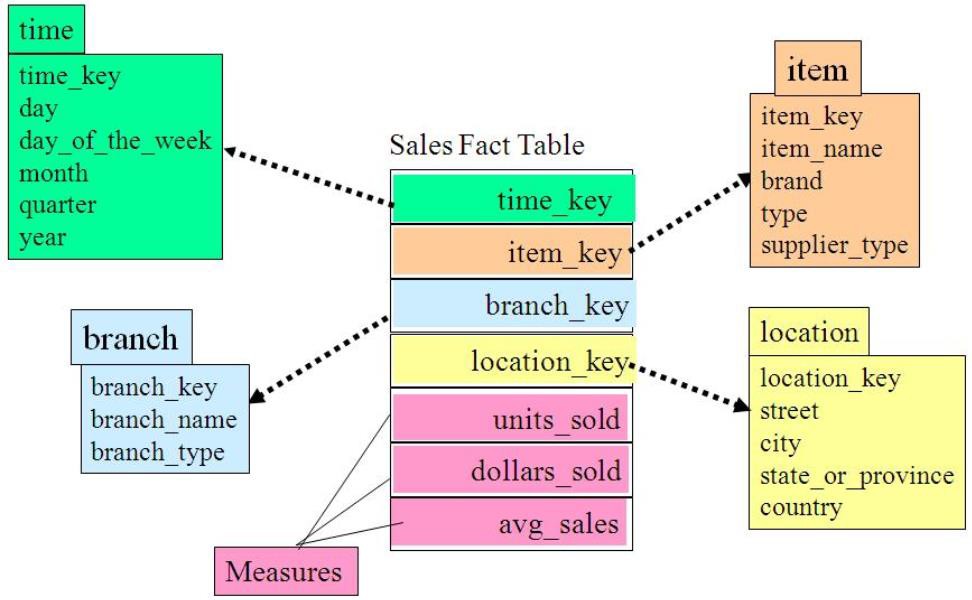


图 3-1 星型模型

星型模式是维度模型中最简单的形式，也是数据仓库以及数据集市开发中使用最广泛的

#### 形式。星型模式由事实表和维度表组成，一个星型模式中可以有一个或多个事实表，每个

**事实表引用任意数量的维度表**。星型模式的物理模型像一颗星星的形状，中心是一个事实表， 围绕在事实表周围的维度表表示星星的放射状分支，这就是星型模式这个名字的由来。

星型模式将业务流程分为事实和维度。事实包含业务的度量，是定量的数据，如销售价格、销售数量、距离、速度、重量等是事实。维度是对事实数据属性的描述，如日期、产品、客户、地理位置等是维度。一个含有很多维度表的星型模式有时被称为蜈蚣模式，显然这个名字也是因其形状而得来的。蜈蚣模式的维度往往只有很少的几个属性，这样可以简化对维度表的维护，但查询数据时会有更多的表连接，严重时会使模型难于使用，因此在设计中应该尽量避免蜈蚣模式。

## 雪花模式

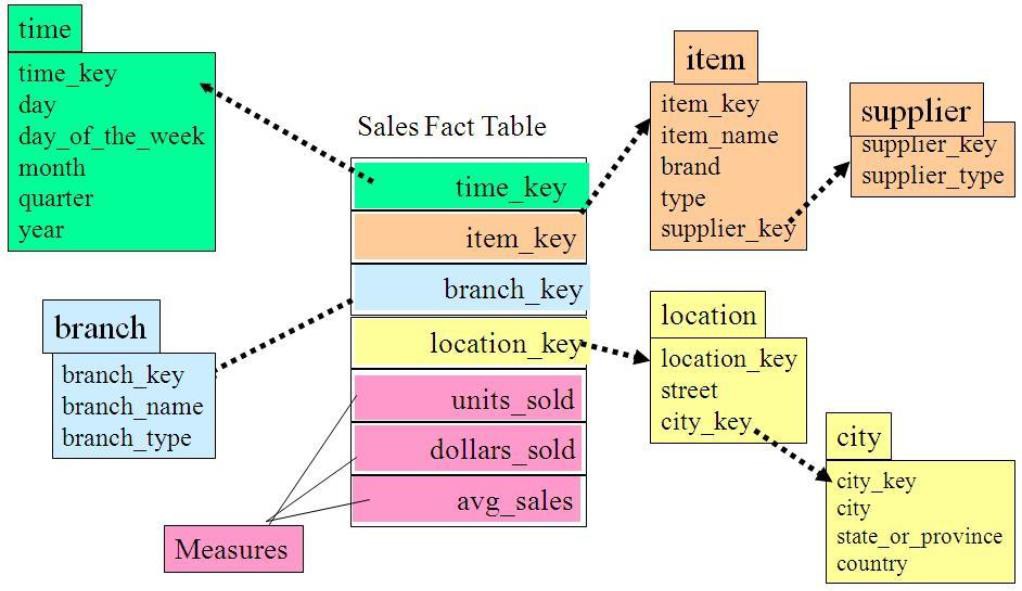


图 3-2 雪花模型

雪花模式是一种多维模型中表的逻辑布局，其实体关系图有类似于雪花的形状，因此得名。

与星型模式相同，雪花模式也是由事实表和维度表所组成。**所谓的“雪花化”就是将行星模型中的维度表进行规范化处理**。当所有的维度表完成规范化后，就形成了以事实表为

中心的雪花型结构，即雪花模式。将维度表进行规范化的具体做法是，把低基数的属性从维度表中移除并形成单独的表。基数指的是一个字段中不同值的个数，如主键列具有唯一值， 所以有最高的基数，而像性别这样的列基数就很低。

**在雪花模式中，一个维度被规范化成多个关联的表，而在星型模式中，每个维度由一个单一的维度表所表示**。一个规范化的维度对应一组具有层次关系的维度表，而事实表作为雪花模式里的字表，存在具有层次关系的多个父表。

## 星座模式

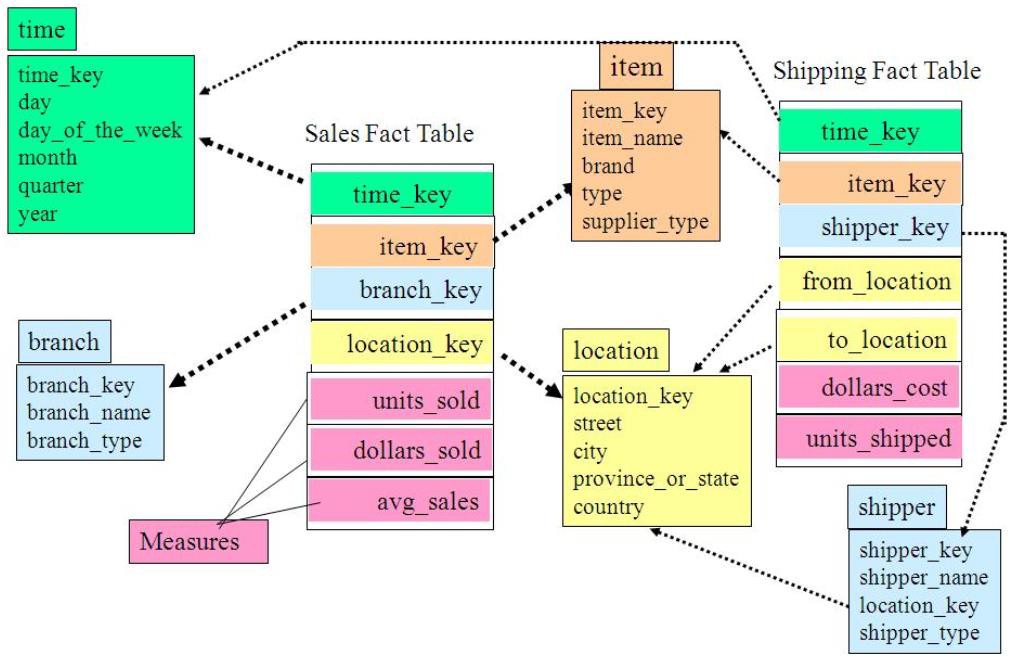


图 3-3 星座模式

数据仓库由多个主题构成，包含多个事实表，而维表是公共的，可以共享，这种模式可以看做星型模式的汇集，因而称作星系模式或者事实星座模式。

## 模型的选择

在数据仓库建模时，会涉及到模式的选择，我们要根据不同模式的特点选择适合具体业务的模式：

**冗余**：雪花模型符合业务逻辑设计，采用 3NF 设计，有效降低数据冗余；**星型模型的维度表设计不符合 3NF（如果是雪花模型改造成了星型模型，那么肯定不符合 3NF，因为一定发生了表的整合，即降维，一定有传递依赖，但是，并不是所有的星型模型都不符合**

**3NF，很多星型模型的表是符合 3NF 的），反规范化**，维度表之间不会直接相关，牺牲部分存储空间。（雪花模型的维度之间是有关联的）

**性能**：雪花模型由于存在维度间的关联，采用 3NF 降低冗余，通常在使用过程中，需要连接更多的维度表，导致性能偏低；星型模型反三范式，采用降维的操作将维度整合，以存储空间为代价有效降低维度表连接数，性能较雪花模型高。（ 星型表的数据冗余大，是用存储空间换取效率 ）（ BI 的一些工具对于星型模型的支持更规范化 ）

**ETL**：雪花模型符合业务 ER 模型设计原则，在 ETL 过程中相对简单，但是由于附属模型的限制，ETL 任务并行化较低（由于雪花模型中有很多的维度依赖，在 ETL 的时候， 需要在保持 3NF 的前提下对数据进行清洗，即对数据一致性/规范化的处理，例如数据来自于多个业务系统，各个系统对于用户的定义不一致，此时要对每个业务定义的用户数据进行

规范化处理，在 3NF 的限制下必然会降低并行度）；星型模型在设计维度表时反范式设计，

所以在 ETL 过程中整合业务数据到维度表有一定难度，但由于避免附属维度，可并行化处理（不用关注太多的关联关系，避免了维度表之间的关联关系，并行度较高，注意，一般场景下星型模型的并行化程度更高，并不是所有场景）。

Hive 的分析通过 MapReduce 实现，每多一个 Join 就会多出一个 MapReduce 过程，对于雪花模型，由于存在着很多维度表之间的关联，这就会导致一次分析对应多个 MapReduce 任务，而星型模型由于不存在维度表的关联，因此一个 MapReduce 就可以实现分析任务。

MapReduce 本身是一个支持高吞吐量的任务，它的每个任务都要申请资源、分配容器、节点通信等待，需要 YARN 的调度，由于相互关联的维度表本身会很小，join 操作用时很少，

YARN 调度的时长可能都大于实际运算的时长，因此我们要尽可能减少任务个数，对于 Hive 来说就是尽可能减少不必要的表的关联。还有一点，雪花模型中拆分出的维度表，每个表对应至少一个文件，这就涉及到 I/O 方面的性能损耗。

#### 因此，我们要采用适当的数据冗余，避免不必要的表之间的关联。

在实际项目中，**不会刻意地去考虑雪花模型，而是刻意地去考虑星型模型，特别是大数据领域的建模，倾斜于使用数据冗余来提高查询效率，倾向于星型模型；雪花模型只会应用在一些我们要求模型的灵活性，要求保证模型本身稳定性的场景下，但是雪花模型并不是首选**。

* 1. **数据仓库分层理论**
     1. **CIF 层次架构**

**CIF 层次架构（信息工厂）通过分层将不同的建模方案引入到不同的层次中**，CIF 将数据仓库分为四层，如图所示：

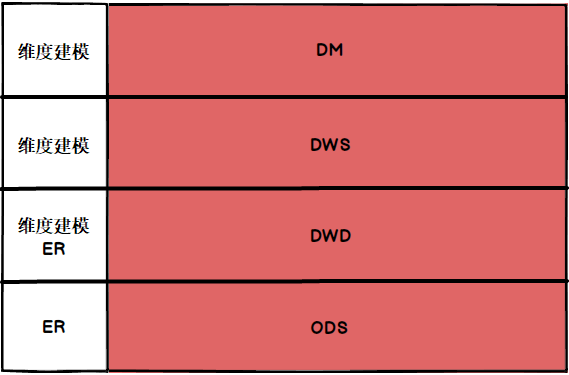


图 3-10 CIF 层次架构

ODS（Operational Data Store）：操作数据存储层，往往是业务数据库表格的一对一映射，将业务数据库中的表格在 ODS 重新建立，数据完全一致；

DWD（Data Warehouse Detail）：数据明细层，在 DWD 进行数据的清洗、脱敏、统一

化等操作，DWD 层的数据是干净并且具有良好一致性的数据；mapreduce

DWS（Data Warehouse Service）：服务数据层（公共汇总层），在 DWS 层进行轻度汇总，为 DM 层中的不同主题提供公用的汇总数据；

DM（Data Market）：数据集市层，DM 层针对不同的主题进行统计报表的生成；

## 层次功能详解

数据采集：

tomcat(J2EE项目) log日志落盘-》flume采集-》kafka->HDFS（随意的目录

/user/zhangsan/2020/12/9）通过linux脚本写入到hive.(ODS，基本数据层)

**mysql mongdb。**

**spark:读取数据：hdfs,hive,mysql,mongodb,文件，**

|  |
| --- |
| **CREATE EXTERNAL TABLE ods\_ykt\_base\_term\_yy\_f (**  **ecode string COMMENT '企业代码'**  **,term\_id int COMMENT '终端编号'**  **,term\_name string COMMENT '终端名称'**  **,term\_addr string COMMENT '终端地址,如果是以太网设备就为实际的IP地址'**  **,dpt\_code string COMMENT '商户部门代码,不可重复'**  **,account\_code int COMMENT '一卡通系统的科目代码'**  **,dscrp string COMMENT '一卡通系统的科目描述'**  **,isuse int COMMENT '是否使用'**  **,extended\_term\_addr int COMMENT '扩展终端编号'**  **,pos\_code string COMMENT '设备运营唯一编号'**  **,type\_id int COMMENT '终端类型编号'**  **,sam\_card\_no bigint COMMENT '卡号,没有sam卡为0'**  **,communication\_mode int COMMENT '通讯方式'**  **,all\_dpt\_code string COMMENT '终端所属部门编号'**  **,update\_flag string COMMENT '更新状态'**  **,update\_time string COMMENT '更新时间'**  **,down\_time string COMMENT '读取时间'**  **,ver string COMMENT '版本号'**  **)**  **COMMENT '终端信息表' PARTITIONED BY(**  **partition\_date string)**  **STORED AS PARQUET**  **LOCATION**  **'hdfs://dse.host.hz.io:8020/jkd\_dw/ods/ods\_ykt\_base\_term\_yy\_f'**  **;** |

### **ODS(原始数据层)**Operational Data Store)

**ODS 层中的数据全部来自于业务数据库，ODS 层的表格与业务数据库中的表格一一对应，就是将业务数据库中的表格在数据仓库的底层重新建立一次，数据与结构完全一致。 ODS\_WEBLOG**

**由于业务数据库（OLTP）基本按照 ER 实体模型建模，因此 ODS 层中的建模方式也是**

**ER 实体模型。**

### DWD（数据ETL层）

DWD 层要做的就是将数据清理、整合、规范化，脏数据、垃圾数据、规范不一致的、状态定义不一致的、命名不规范的数据都会被处理。**DWD 层应该是覆盖所有系统的、完整的、干净的、具有一致性的数据层**。

DWD\_WEBLOG

在 DWD 可能会用到 ER 或者维度模型。在 DWD 层会抽取出公共维度，例如区域等。也就是说 DWD 层是一个非常规范的，高质量的，可信的数据明细层。

### DWS（数据汇总层）（点击数据《--》用户数据）

**DWS 层为公共汇总层，会进行轻度汇总，粒度比明细数据稍粗，会针对度量值进行汇总，目的是避免重复计算**。**往往在 DWS 层建立宽表**，例如订单总金额，可能在原始数据中没有这个数据，进入 DWS 层后可以统计出订单总金额，避免重复地拿订单明细数据去计算。

DWS 层建议使用维度建模，因为数据仓库的主要应用是进行数据分析。

### DM（数据集市）（交给使用数据的各个部门BI）

#### DM 层为数据集市层，面向特定主题，例如订单主题、物流主题等。在 DM 完成报表或者指标的统计，DM 层已经不包含明细数据，是粗粒度的汇总数据，因此 DM 层会被当成 BI 或者 OLAP 的底层模型。

在大数据数据仓库领域内，数据仓库是包括集市的，而且物理上是统一、非隔离的，集市的概念相较与传统数据仓库比较弱化，由于有底层明细数据、通用汇总数据的存在，数据集市一般位于上层。

应用层面存在相应分析主题的概念，甚至很大程度上存在集市交叉的现象，所以如果是在大数据领域构建企业整体数据仓库，并且数据集市也一块规划，建议集市弱化，把它当作是梳理上层数据域的工具。

主题设计思路：

* 按照使用部门划分集市主题
* 按照业务模块划分集市主题

# 第四章 数据仓库数据采集与同步

## 系统埋点设计

* + 1. **数据分类**

在工厂环境中，我们将数据仓库获取的数据划分为业务数据和用户行为数据。

**业务数据**：**业务流程中产生的交易、状态流转、用户等相关的数据**，通常存储在 DB 中， 包括 rdbms、nosql 等，这部分数据是业务相关的，具体哪些数据需要保留一般由业务侧设计，不需要过度关注，按实际需要采集即可。（数据库中的字段，）

**用户行为数据**：**用户在使用产品过程中，与 C 端产品交互过程中产生的数据**，比如页面浏览、点击、停留等，这部分数据由于不影响业务流程本身，所以通常不受关注，但对运营、产品优化至关重要，所以通常也是数据建设的一部分，需要单独设计数据埋点、采集策略。

app类项目，电商类项目。

## 用户行为数据埋点设计。

用户行为数据埋点的设计有三个关键点：

* 用户标识体系建立（酷我听歌，必须需要登录吗？不登录能不能听。）
* 多屏用户标识打通（多屏：手机、PC、Pad）
* 埋点方案设计

### 用户标识体系建立

在电商场景中经常会遇到用户在不登录/注册的情况下访问网站的情况，

（Ip,mac，openId,） 苹果手机中的Id： IP,Mac ,openId();

淘宝不登录，将商品放入购物车，登录后，商品已经在购物车了。

对于电商平台来说，希望能够收集到用户所有的访问行为数据，包括用户在未登录/注册之前的行为数据， 这就涉及到了用户标识的问题，如何在用户不登录/注册的情况下对用户进行标识，并且将不登录/注册情况下的用户行为数据追加到用户完整的用户行为数据中，这是用户标识体系需要完成的工作。

同样的，对于运营、产品和研发部门，需要根据用户在注册前后的行为明确是什么吸引了用户最终完成注册，因此需要将用户注册之前的行为与注册之后的行为关联起来，也就是将注册前后的信息打通。

* 1. 用户标识建立方案

考虑到移动端和 PC 端的区别，我们在建立用户标识时划分两个场景，分别是 PC 端和

APP 端：

1. PC 端

PC 端的电商平台在浏览器中进行展示，也就是以 H5 的形式进行展示，采用 cid 和 uid

相结合的方式对用户进行标识。

**cid**：类似于 cookieId，但不是 cookieId，cid 随着不同的用户切换在改变，我们的目的是用 cid 来标识用户，同一个用户在不同的屏幕上是不同的 cid。

**uid**：是用户注册后分配给用户的，可以唯一标识一个用户的序号。

1. APP

APP 端同样采用 cid 和 uid 相结合的方式对用户进行标识。

**cid**：使用设备 imei（入网许可证号）/IDfa 根据某种规则生成唯一设备标识，作为 cid。

**uid**：是用户注册后分配给用户的，可以唯一标识一个用户的序号。

以上的两种方案都引入了 cid 和 uid 的概念，只不过不同的平台 cid 的定义不同。

#### 在用户注册/登录前，使用 cid 作为设备标识，跟踪用户行为数据；用户注册/登录后， 使用 uid 标识用户，跟踪用户行为数据。

① 一个新设备，从来没有访问过电商平台，第一次无登录访问时，电商平台给他分配一个 cid，然后，用户登录了，由于不是切换用户，那么 cid 不变，登录后的数据有一个 uid， 根据登录后的 cid 和登录前的 cid 进行数据的匹配。

② 一个新设备，从来没有访问过电商平台，第一次无登录访问时，电商平台给他分配一个 cid，然后，用户注册了，由于是新注册，cid 不变，用户登录后电商平台分配给他一个新的 uid，根据登录后的 cid 和登录前的 cid 进行数据的匹配。

③ 一个老设备，一个人登录后，cid=xyz，uid=123，退出登录后，cid 不变，uid 不变， 之后所有的未登录访问行为的 cid 和 uid 都是上一次登录的人的 cid 和 uid，因此全部会被划分到上一个登录的人的行为里。

④ 一个老设备，一个人登录后，cid=xyz，uid=123，退出登录后，cid 不变，uid 不变， 之后有人再次登录，如果 cookie 中的 uid 和之前的不一致，那么判定为用户登录切换，是不同用户，cid 改变，uid 改变，如果 cookie 中的 uid 和之前的一致，那么 cid 和 uid 都不变化。

根据③可知，如果一个人登录后再推出登录，在下次登录之前，即使是多个不同的人访问了电商平台，也会被记为上次登录的人的访问行为。**虽然 cid 并不是非常精确，但是他能够让数据找到归属**。

### 多屏用户打通

一个用户可能同时拥有手机、平板、电脑等设备，我们将多种不同的电子设备称为多屏。我们希望能够将同一个用户在多屏上的数据进行打通。

同一用户在不同屏幕上的 cid 是不同的，但是 uid 是相同的，但是如果只使用 uid，那么注册/登录之前的数据就会丢失，因此必须将 cid 和 uid 结合使用。

#### 综上所述，通过 cid 和 uid 相互结合，实现未注册/登录数据与注册/登录数据的整合，

**然后通过 uid 实现多屏数据的打通**。

1. 埋点设计
   1. 埋点对象分类

在生产环境下，我们主要的埋点对象有两类：Native APP、Web H5。

PC 端的用户主要访问的是 H5 页面，移动端用户主要是使用 APP 进行访问，当前我们所接触到的 APP 有多种类型，如 Native APP、Web APP、Hybird APP，在详细讨论埋点设计之前，我们先了解一下几种不同的 APP：

#### Native App

Native App 是一种基于智能手机本地操作系统如 iOS、Android、WP 并使用原生程式编写运行的第三方应用程序，也叫本地 app。一般使用的开发语言为 JAVA、C++、Objective-C。

Native App 因为位于平台层上方，向下访问和兼容的能力会比较好一些，可以支持在线或离线，消息推送或本地资源访问，摄像拨号功能的调取。但是由于设备碎片化，App 的开发成本要高很多，维持多个版本的更新升级比较麻烦，用户的安装门槛也比较高。

#### Web APP

Web App 就是运行于网络和标准浏览器上，基于网页技术开发实现特定功能的应用。

WebApp 是指基于 Web 的系统和应用，其作用是向广大的最终用户发布一组复杂的内容和功能。

当用户登录一个网站（如百度、淘宝），大家很容易理解这是在访问一个 Web App。但是对那些仅仅提供基础服务（如电话查询或是信息查询）的网站，区分用户是否在访问 Web

App 就变得相当困难了。

其实这些服务大多都是 Web App。我常常这样问自己“这个程序是否完成了某个任务？”。即便它只完成了某个非常小的任务，那么它也是一个 Web App。Google 的搜索引擎就是一个 Web App，它本质上和电话查询服务没有什么区别。

#### Hybrid APP

Hybrid App（混合模式移动应用）是指介于 web-app、native-app 这两者之间的 app，兼具“Native App 良好用户交互体验的优势”和“Web App 跨平台开发的优势”。

Hybrid App 是指介于 web-app、native-app 这两者之间的 app，它虽然看上去是一个 Native

App，但只有一个 UI WebView，里面访问的是一个 Web App，比如街旁网最开始的应用就是包了个客户端的壳，其实里面是 HTML5 的网页，后来才推出真正的原生应用。再彻底一点的，如掌上百度和淘宝客户端 Android 版，走的也是 Hybrid App 的路线，不过掌上百度里面封装的不是 WebView，而是自己的浏览内核，所以体验上更像客户端，更高效。

Native APP 无法获取 URL 相关信息，Web APP 无法获取设备相关的信息，因此，市面上现有的 APP 大部分是 Hybird APP，它兼具 Native APP 和 Web APP 的特点，能够同时获得设备信息和 URL 信息。

通常设计埋点时针对 Native App、Web H5 采用不同的规范，但不利于数据整合和使用， **推荐采用事件模型进行埋点设计，即把用户的一切行为动作都看作事件，包括页面浏览、**

#### 按钮悬停、点击等，这样 Native 和 H5 统一数据标准，实际数据仅参数上的区别。

ajax异步请求时间。（nginx）(用户行为数据)

* 1. 埋点分类。

根据埋点位置，可分为**客户端埋点**、**服务端埋点**。

两种埋点方式各有利弊，比如服务端埋点对无后台请求的用户行为无法捕获（比如用户在页面上的悬停，这种事件是有意义的，比如用户会悬停在感兴趣的品类、商品或者广告上），而客户端埋点可能会由于用户的环境问题存在数据包丢失，客户端可能无法获取全部的数据

（有些数据存储在服务端，因为页面不会存储过多不会经常使用的信息，客户端需要一些调用才能够获取到，对性能会有影响）等，所以无特殊情况下，**建议采用服务端埋点方案**。

APP 的更新时是周期性的，你没有办法强制用户去更新 APP，如果在客户端埋点的话就会由于版本更新的问题没有办法改变埋点，但是如果在服务器埋点的话就没有这个问题， 可以在服务器端按照需求设置埋点，不关心 APP 的版本。

如果有一些类似于悬停这种事件，服务端无法捕获到，而这些事件又是非常重要的，那么可以在页面上加入一些代码，当发生这些事件的时候执行一个请求服务器的动作，让服务端进行埋点，负责这部分数据的落地。

## 业务数据埋点设计

业务数据一般存储于 rdbms 中，有业务 RD 根据业务流程设计，具体记录信息无需过度关注，但在可能的情况下，尽可能提出相应的规范。

* 每张业务表必须有自增 id 能够唯一标识一行记录
* 必须包含该记录的写入时间(created\_time)、数据更新时间(updated\_time)，部分表最好能够在 updated\_time 列构建索引（方便后续的数据采集，根据时间进行数据的获取）

#### 独立的业务数据埋点：由业务服务端根据具体场景的需要，独立记录和业务强相关数据日志（不是用户行为数据），通常落地于相应业务应用的服务器磁盘，需要单独采集， 方案可以参考行为数据的采集。

* + 1. **数据埋点案例分析**

1. 背景

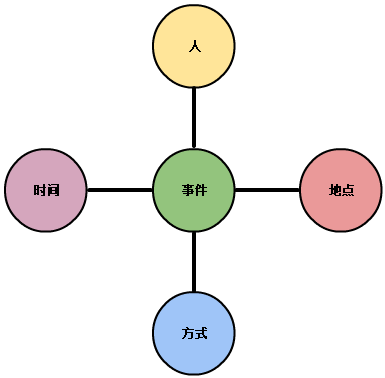
某电商平台，产品覆盖 app、pc 端，其中 app 为 hybrid 架构，业务数存储 mysql 库中；

1. 需求

需要分析用户访问路径、各步转化、流失等进行页面优化、运营效果评估、商品推荐；

### 设计埋点规范

* 一切用户操作行为都看作事件
* 应覆盖事件的核心要素



### 数据格式

图 4-1 时间的核心要素

为确保灵活、可扩展性，上报数据采用 json 格式，不要太深的嵌套（99%的埋点都是一层），将数据分为两部分：公有参数、自定义私有参数。

* 1. 公有参数

#### 公有参数需要覆盖事件核心要素，实际上报数据时不可缺失。

代码清单 4-1 埋点数据公有参数

* 1. 私有参数

{

timestamp cid

uid sessionId ipAddress page ctype type event useragent content{}

version

**时间**

**C 端标识用户 id**

**session**

**ip 地址页面**

**C 端类别(native\h5)**

**类型(pageview\click\start…) 事件**

**手机平台（Android/IOS）等信息私有自定义信息**

**日志版本（日志规范可能会升级，此处标识日志遵循的规范版本）**

}

#### 私有参数即根据事件发生时的场景、需求，定义需要抓取的数据内容。

某种意义上该部分数据也分公共参数与私有参数，比如事件发生在 app 中，设备信息都会伴随当时的设备信息；事件发生在 h5 中，一般会伴随 url。

对于一般的推广 H5，需要抓取基本数据：

currurl（当前页面的 URL）、refferurl（上一级页面的独立 URL）、refferpage（上一级页面的独立名称）（适配 App 页面）、sourceid（跟踪访问的源头，是微信、微博还是其他的推广平台，用以判断哪边的推广效果更好）、自定义参数(比如商品 ID、活动 ID、商家 ID、

posid 等)

对于 App，一般需要附加设备信息：

refferpage（上一级页面的独立名称）、自定义参数( 商品 id、商家 id 等等)、设备信息

(imei、idfa、os、gps、wifi 等信息)

#### 对于 hybrid APP，其中包括 native 和 h5，虽然是 h5，但由于在 app 中，依然能够拿到相关设备信息，因此按照 app 的参数设计进行埋点，同时也要包括 h5 自身的特有数据，

比如 url、refferurl 等。

### 埋点数据内容设计

* 1. APP 启动埋点

本着抓取高价值、符合需求、不影响用户体验和性能的原则，仅在用户启动 app 时触发，在用户授权的情况下抓取用户手机中安装的 app 信息、通讯录信息，频率可以自定义。

代码清单 4-2 埋点数据公有参数

{

timestamp cid

uid sessionId ipAddress page ctype type event useragent

content{

**时间**

**C 端标识用户 id**

**session**

**ip 地址页面**

**C 端类别(native\h5)**

**类型(pageview\click\start…) 事件**

**手机平台（Android/IOS）等信息私有自定义信息**

installapps:[{appname:, intime}, {}, ...], **用户已安装 App 列表**

activeapps:[{appname:, starttime}, {}, ...] **用户当前启动的 App 列表**

phones:[{name:, phone}, {}, ...] **通讯录**

imei **IMEI 号**

idfa **设备唯一标识**

os **操作系统类型**

gps\_lng **GPS 信息**

gps\_lat **GPS 信息**

wifiname **WIFI 名称**

appversion **App 版本**

}

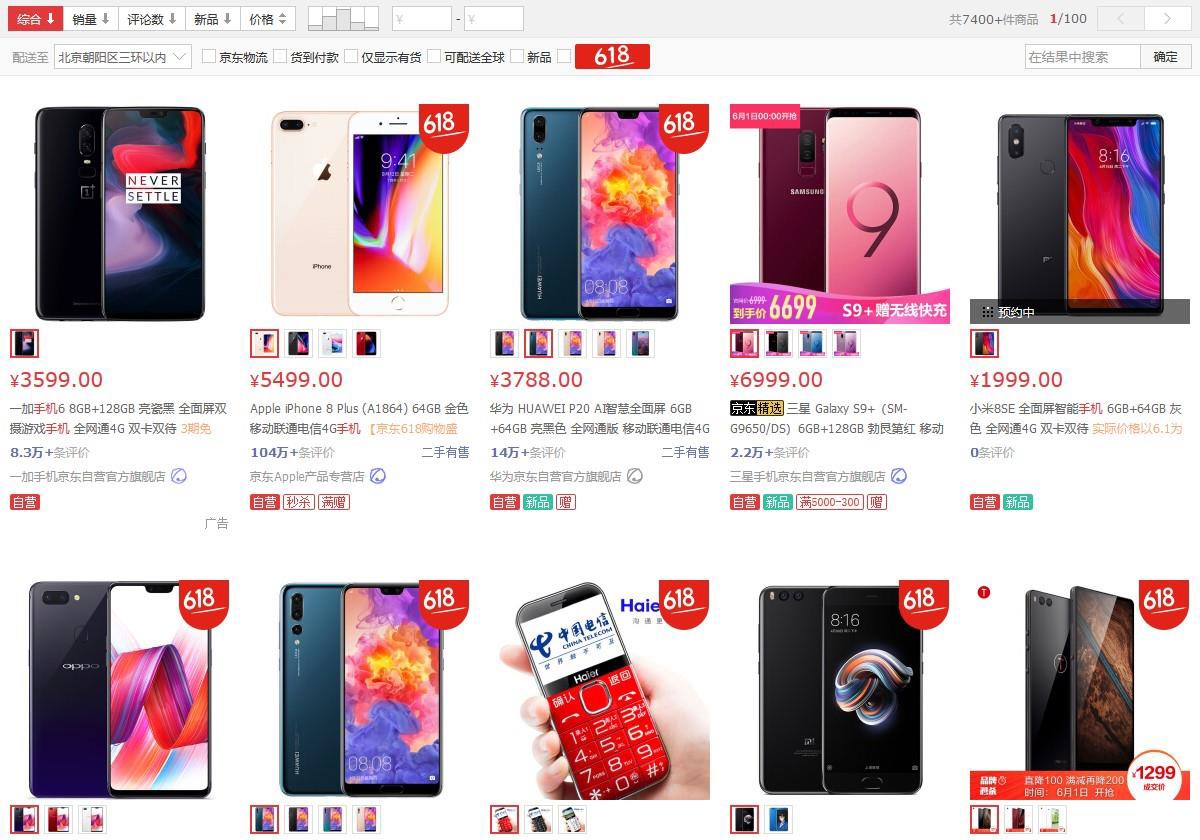
version **日志版本**

}

* 1. 列表页

1. 普通版本

图 4-2 列表页



代码清单 4-3 埋点数据公有参数

{

timestamp cid

uid sessionId ipAddress page ctype type event useragent content{

url refferurl refferpage pagenum imei

idfa os gps

wifiname appversion

}

version

**时间**

**C 端标识用户 id**

**session**

**ip 地址页面**

**C 端类别(native\h5)**

**类型(pageview\click\start…) 事件**

**手机平台（Android/IOS）等信息私有自定义信息**

**当前页面 URL 上 一 页 URL 上一页名称**

**列表页页码（商品有多页，记录当前是第多少页）**

**IMEI 号**

**设备唯一标识操作系统类型**

**GPS 信息**

**WIFI 名称**

**App 版本**

**日志版本**

}

1. 计费版本一

页面采用翻页模式，一页只有一个事件，进入列表页后，将这一页的所有商品进行上报。

代码清单 4-4 埋点数据公有参数

{

timestamp cid

uid sessionId ipAddress page ctype type event useragent content{

url refferurl refferpage pagenum

goodlist[{},{}] imei

idfa os gps

wifiname appversion

}

version

**时间**

**C 端标识用户 id**

**session**

**ip 地址页面**

**C 端类别(native\h5)**

**类型(pageview\click\start…) 事件**

**手机平台（Android/IOS）等信息私有自定义信息**

**当前页面 URL 上 一 页 URL 上 一 页 名 称 列 表 页 页 码 商品列表**

**IMEI 号**

**设备唯一标识操作系统类型**

**GPS 信息**

**WIFI 名称**

**App 版本**

**日志版本**

}

1. 计费版本二

对于精确的曝光计费，特别是滚动刷新的方式（非翻页），用户不真正看到商品是不计入 goodlist 的，之后用户真正滚动到了商品的位置，才会计入 goodlist，上报服务器。

在计费方式二的情况下，会随着每次页面的翻转发送一条列表页日志数据。

代码清单 4-5 埋点数据公有参数

{

timestamp cid

uid sessionId ipAddress page ctype type event useragent content{

url refferurl refferpage

pagenum

**时间**

**C 端标识用户 id**

**session**

**ip 地址页面**

**C 端类别(native\h5)**

**类型(pageview\click\start…) 事件**

**手机平台（Android/IOS）等信息私有自定义信息**

**当前页面 URL 上 一 页 URL 上 一 页 名 称 列表页页码**

goodlist[{goodsId, pos},{}] **商品列表**

imei **IMEI 号**

idfa **设备唯一标识**

os **操作系统类型**

gps **GPS 信息**

wifiname **WIFI 名称**

appversion **App 版本**

version

**日志版本**

}

* 1. 详情页



1. click

图 4-3 商品详情页

代码清单 4-6 埋点数据公有参数

{

timestamp cid

uid sessionId ipAddress page ctype type event useragent content{

url refferurl refferpage goodsid sjid pagenum position prearvtime imei

idfa os gps

wifiname appversion

}

version

**时间**

**C 端标识用户 id**

**session**

**ip 地址页面**

**C 端类别(native\h5)**

**类型(pageview\click\start…)**

**listclick**

**手机平台（Android/IOS）等信息私有自定义信息**

**当前页面 URL 上 一 页 URL 上 一 页 名 称 商品 Id**

**商家 Id**

**列表页编号点击位置**

**预计收货时间**

**IMEI 号**

**设备唯一标识操作系统类型**

**GPS 信息**

**WIFI 名称**

**App 版本**

**日志版本**

1. preview

preview 事件的 PV 往往低于 click 事件，因为有的访问会加载失败。如果 preview 和 click 相差太多（同一个页面），那么就要审视是否是产品出了问题，为什么这么多次没有加载出来，是后台性能太差还是网络一直有问题。

|  |  |  |
| --- | --- | --- |
|  | | 代码清单 4-7 埋点数据公有参数 |
| { | timestamp | **时间** |
|  | cid  uid | **C 端标识**  **用户 id** |
|  | sessionId ipAddress page ctype  type | **session**  **ip 地址页面**  **C 端类别(native\h5)**  **类型(pageview\click\start…)** |
|  | event  useragent | **listclick**  **手机平台（Android/IOS）等信息** |
|  | content{ url  refferurl | **私有自定义信息**  **当前页面 URL**  **上一页 URL** |
|  | refferpage goodsid  sjid | **上一页名称**  **商品 Id**  **商家 Id** |
|  | prearvtime  imei | **预计收货时间**  **IMEI 号** |
|  | idfa os gps  wifiname  appversion | **设备唯一标识**  **操作系统类型**  **GPS 信息**  **WIFI 名称**  **App 版本** |
|  | }  version | **日志版本** |
| } |  |  |

* 1. 首页

我们关注用户是通过什么途径进入主页的，这就涉及到渠道推广，因为很多电商在其他平台进行推广，而推广是需要交纳一定费用的，而其他平台向你提出的收费额度，你怎么确定是否准确，就要通过首页的埋点进行采集。



图 4-4 首页

代码清单 4-7 埋点数据公有参数

{

timestamp cid

uid sessionId ipAddress page ctype type event useragent content{

url refferurl refferpage sourceid imei

idfa os gps

wifiname appversion

}

version

**时间**

**C 端标识用户 id**

**session**

**ip 地址页面**

**C 端类别(native\h5)**

**类型(pageview\click\start…)**

**listclick**

**手机平台（Android/IOS）等信息私有自定义信息**

**当前页面 URL 上 一 页 URL 上一页名称**

**从哪里进入的主页**

**IMEI 号**

**设备唯一标识操作系统类型**

**GPS 信息**

**WIFI 名称**

**App 版本**

**日志版本**

}

* 1. banner

页面上的不同广告位的定价不同，同一广告位会有多个广告（滚动屏，自动切换），这些广告由于前后位置的不同收费不同，除了首页占据大面积的滚动屏，还有固定位置的广告。



图 4-5 banner

代码清单 4-8 埋点数据公有参数

{

timestamp cid

uid sessionId ipAddress page ctype type event useragent content{

url refferurl refferpage position gourl imei

idfa os gps

wifiname appversion

}

version

**时间**

**C 端标识用户 id**

**session**

**ip 地址页面**

**C 端类别(native\h5)**

**类型(pageview\click\start…)**

**bannerclick**

**手机平台（Android/IOS）等信息私有自定义信息**

**当前页面 URL 上 一 页 URL 上 一 页 名 称 点击位置**

**跳转到的 URL**

**IMEI 号**

**设备唯一标识操作系统类型**

**GPS 信息**

**WIFI 名称**

**App 版本**

**日志版本**

}

埋点参数等设计完毕后，提交相关开发组开发。

若存在客户端埋点，需考虑用户网络环境下的数据丢失、数据上报性能：

* 对于 h5，实时、异步上报。
* 对于 App 客户端缓存，小批量压缩上报，尤其是数据比较大时；比如缓存 3s 或者数据大于 1M 主动上报。

## 埋点数据采集系统搭建

埋点数据的采集方案有多种选择，下图为企业常用的埋点数据采集方案之一。

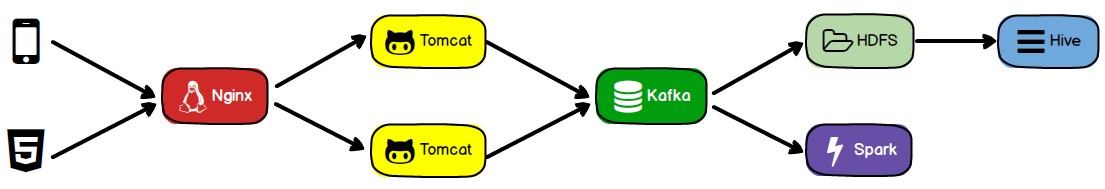


图 4-6 埋点数据采集系统

app->nginx->kafka->hdfs

spark

app->nginx->tomcat->flume->kafka->hdfs->hive

spark->hive

sparkStreaming

app->nginx->tomcat->kafka->hdfs-

spark

如上图所示，APP 以及 H5 端的埋点数据首先被发送到 Nginx 中，由 Nginx 负载均衡到

Tomcat 服务器，Tomcat 服务器中部署的 Servlet 接受埋点数据，处理后调用 Kafka API 发送到 Kafka 集群，离线数据处理和实时处理分别通过 Kafka 消费者消费 Kafka 集群中的数据， 然后进行后续的处理工作。

根据 Kafka 消息队列的特性，消息队列实现了生产者与消费者的隔离，在上述的采集方案中，业务层的 Servlet 为生产者，数据处理层的离线业务或者实时业务为消费者，通过 Kafka 的特性，实现了业务层与数据处理层的隔离。

离线业务消费埋点数据后，会将用户行为数据写入 ODS 层的表格中，之后在数据仓库中进行数据的分析处理。

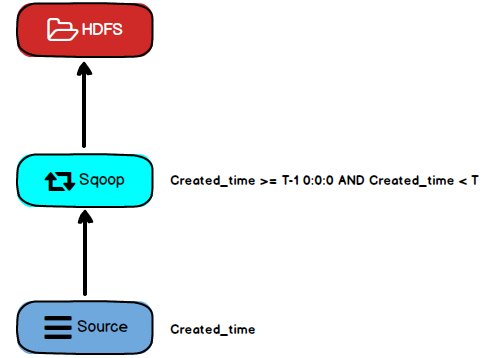
实时业务消费埋点数据后，会对数据进行实时的分析和处理，之后存入数据库。

## 数据同步策略

* + 1. **增量型**

### 无状态变更数据

假设数据源是流水数据，此类数据没有状态变更，写入数据库后基本不再改变，数据中一般包含 Created\_time 信息，可以根据 Created\_time 的值获取增量数据，或者记录上次的获取到的 ID，然后从下一个 ID 开始获取，这是一种纯增量采集。



### 有状态变更数据

图 4-7 无状态变更数据

假设表比较大，比如说一些订单表，这些表的状态变化周期一般偏长，状态变化一直会更新，而且状态变化会跨零点。

此时要求所有的源系统数据库的设计需要加入 Created\_time、updated\_time，在 Source

层把昨天发生变化的数据抽取过来（updated\_time >= T-1 0:0:0），没有设置 updated\_time < T

0:0:0，因为你昨天对数据进行了修改，可能在今天采集数据之前又进行了修改，因此，只 要昨天修改过就符合抽取条件，如果想更更精准地获取数据，或者说让抽取的数据量小一点， 那么可以加入 Created\_time 的限制条件（updated\_time >= T-1 0:0:0 and Created\_time < T

0:0:0），通过这种抽取条件就可以获得昨天更新的数据和昨天新增的数据。



图 4-8 有状态变更数据

获取到更新数据后进行合并，两张表进行比对（根据主键进行比对）：（A 表为 T-2 全量数据，B 表为增量数据）

1. 如果 A 表中包含的数据（根据自增主键判断）在 B 表中也包含，那么就代表数据在昨天发生了变化，那么就用 B 中的数据替换 A 中的数据；
2. 如果 A 表中没有的数据在 B 表中包含，那么就代表数据时昨天新插入的，那么就将

B 中数据插入 A；

1. 如果 A 表中包含的数据在 B 表中没有，那么就代表数据在昨天没有被更新；

经过上述逻辑后，将更新后的全量数据放到 A 表的 T - 1 分区中，成为了 T - 1 的全

量数据，按照这一规则，每一天的全量数据都会被保存一份。完成数据更新后 B 表就可以被丢弃了。

## 全量型

**假设表不是很大，而且数据状态会发生变化，可以进行全量采集，采集所有数据**。此表格每天一个快照，累计时间长了之后，数据量也会很大。

数据仓库中的表格 Dw\_a 按照日期进行分区：



图 4-9 全量数据

每个分区中保存的是全量数据（全量快照）。不论是全量抽取还是增量抽取，每个分区

Dt 中保存的都是截止到当日最后一刻，这张表的快照。

05-09 是 5 月 9 号的全量快照，5 月 10 号时，采集当天的增量数据 A\_incr-20180510， 然后用增量数据 A\_incr-20180510 和 Dt=2018-05-09 中的数据进行合并，写入 Dt=2018-05-10 分区中，这样就能保证 0510 是全量数据。如果没有发生变化，那么 5 月 10 号的快照和 5

月 9 号的快照是一样的。

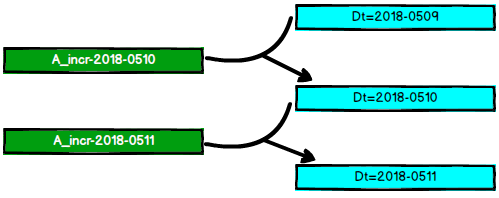


图 4-10 全量采集原理

**所有的表都可以采用这种方式，但是我们考虑，如果表的数据不是很大，那么使用全量采集也没有问题，但是随着数据量的不断增加，每天进行全量采集的代价越来越高，此时可以考虑使用增量采集**。例如订单表，本身数据量很大，变化周期长，那么可以直接采用

增量采集。

# 第五章 数据仓库维度建模剖析

## 概述

* 1. **维度表设计**
     1. **代理键**

**维度表中必须有一个能够唯一标识一行记录的列**（最好是原子性的列，不要是组合键）， 通过该列维护维度表与事实表之间的关系，一般在维度表中业务主键符合条件可以当作维度主键。

但是，数据仓库是整个公司数据的整合，这会涉及到**多个数据源有相同维度**，那么就会出现以下两个问题：

* 当整合多个数据源的维度时，不同数据源的业务主键重复怎么办？
* 涉及维度拉链表时，同一主体多条记录，业务键重复怎么办？

表 5.1 财务部门维度表

|  |  |  |
| --- | --- | --- |
| ID | name | note |
| 1 | Chen | Finc |
| 2 | Zhang | Finc |

表 5.2 研发部门维度表

|  |  |  |
| --- | --- | --- |
| ID | name | note |
| 1 | Li | R&D |
| 2 | Zhou | R&D |

如上图所示，业务键重复，我们可以引入代理键，如下表所示：

表 5.3 引入代理键

|  |  |  |  |
| --- | --- | --- | --- |
| GID | ID | name | note |
| 1 | 1 | Chen | Finc |
| 2 | 2 | Zhang | Finc |
| 3 | 1 | Li | Risk |
| 4 | 2 | Zhou | Risk |

把多个系统的数据复合在一起，同时再维护一个代理键，而且代理键在这个维度表里是唯一标识一条记录的，类似于业务系统的业务键。

#### 代理键是由数据仓库处理过程中产生的、与业务本身无关的、唯一标识维度表中一条记录并充当维度表主键的列，也是描述维度表与事实表关系的纽带。

**在设计有代理键的维度表中，事实表中的关联键是代理键而不是原有的业务主键，即**

**业务关系是靠代理键维护，这样有效避免源系统变化对数仓数据对影响**。

在实际业务中，代理键通常是数值型、自增的值。

## 稳定维度

**部分维度表的维度是在维度表产生后，属性是稳定的、无变化的**。比如时间维度、区 域维度等，针对这种维度，设计维度表的时候，仅需要完整的数据，不需要天的快照数据， 因为当前数据状态就是历史数据状态。



图 5-1 稳定维度

## 缓慢渐变维

**维度数据会随着时间发生变化，变化速度比较缓慢，这种维度数据通常称作缓慢渐变维**，例如电商平台的用户维度表，用户可能会随着时间推移改变收件地址，因此用户维度表

中的收件地址就是一个缓慢变化维。由于数据仓库需要追溯历史变化，尤其是一些重要的数据，所以历史状态也需要采取一定的措施进行保存，保存历史状态的方式有以下三种：

* 每天保存当前数据的全量快照数据（每天一个新增分区），该方案适合数据量较小

（根据公司具体的配置而定）的维度，使用简单的方式保存历史状态。

* 在维表中添加关键属性值的历史字段，仅保留上一个的状态值。可能同时有多个属性都非常重要，而且只能追溯上一个数据，不是所有的历史数据，这种范式应用场景较少。

#### 拉链表：当维度数据发生变化时，将旧数据置为失效，将更改后的数据当作新的记录插入到维度表中，并开始生效，这样能够记录数据在某种粒度上的变化历史。

* + 1. **拉链表详解**

**将数据的变更当做流水记录下来 ，旧的设为失效，新的设为生效，如果粒度为天，那么就可以得到一天的最终状态作为最终状态**。

表 5-4 拉链表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | name | addr | start\_date | end\_date |
| 1 | Zhou | dept2 | 2018-05-01 | 2018-06-09 |
| 1 | Zhou | dept3 | 2018-06-10 | 2018-06-14 |
| 1 | Zhou | dept1 | 2018-06-15 | 9999-12-31 |

表 5-4 中每条记录都有一个 End\_date，当有新的数据产生时，在旧数据的 End\_date 字段中插入日期，然后新插入一条数据，新数据的 End\_date 字段中是一个永久有效的值，如果再发生更新，上一次更新数据的 End\_date 字段设置为当前日期，然后再次插入新数据， 新数据的 End\_date 字段中设置一个永久有效的值。

如果想知道某个员工在 5 月 22 号时在哪个部门，那么可以通过如下 SQL：

Select \* from user where start\_date<= 2018-05-22 and end\_date>= 2018-05-22

#### 根据拉链表的结构，如果对维度表做拉链，那么一个维度实体必然存在多条记录，也就是一个主键 ID 对应多条数据，此时维度表的原子性主键也就没有意义了。

维度表做拉链后会失去原子性主键，那么拉链维度表如何和事实表进行关联呢？ 此时就要用到代理键，也就是在事实表和维度表中同时添加代理键，如下图所示：

表 5-5 用户详情拉链表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| UID | ID | name | addr | start\_date | end\_date |
| 1 | 1 | Zhou | dept2 | 2018-05-01 | 2018-06-09 |
| 2 | 1 | Zhou | dept3 | 2018-06-10 | 2018-06-14 |
| 3 | 1 | Zhou | dept1 | 2018-06-15 | 9999-12-31 |

表 5-6 订单表

|  |  |  |  |
| --- | --- | --- | --- |
| OID | ID -> UID | Tim\_ID | amount |
| 1 | 1 | 5 | 998 |
| 2 | 1 | 9 | 1000 |
| 3 | 2 | 10 | 1499 |

完成代理键的添加后，在之后的统计中，按照代理键进行聚合即可。

事实表来源于业务事务表，代理键和业务本身没有关系，那么怎么在新增数据时在事实表中装载代理键？

#### 当事实表中有新增数据时，新增数据中记录了维度表中原有的原子性主键，可以根据原有的主键匹配维度表中的数据，然后根据新增数据的时间范围找到匹配的代理键，然后在事实表的新增数据中加入代理键。

代理键是维度建模中极力推荐的方式，它的应用能有效的隔离源端变化带来的数仓结构

不稳定问题，同时也能够提高数据检索性能。

但是**代理键维护代价非常高，尤其是数据装载过程中，对事实表带来了较大的影响， 在基于 hive 的数据仓库建设影响更加严重**，比如代理键的生成、事实表中关联键的装载、

不支持非等值关联等问题，带来 ETL 过程更加复杂。

#### 因此，在大数据体系下，谨慎使用代理键，同时对于缓慢渐变维场景，可以考虑用空间换取时间，每天保留维表全量快照，但这样会带来存储成本，根据实际情况衡量。

* 1. **事实表设计**

**5.3.1 事实表设计**

* + 1. 增量存储

当事实表数据无状态变化时，采用增量存储，即每周期仅处理增量部分的数据，纯增量采集。

* + 1. 全量快照

状态有变化，但每天保存当前的快照数据，对于数据量在可控范围内的情况可以采用。保存策略：

* 如果存储空间和成本可接受，完整存储，确保能够追溯到历史每天数据状态；
* 存储空间有限，考虑移动历史快照数据到冷盘，需要使用的时候可恢复；
* 数据历史状态数据无太大价值，可以考虑部分删除，比如近保留每月最后一天的快照数据；

### 拉链

数据量大，但缓慢变化，需要跟踪历史状态，和缓慢渐变维类似。

如果变化非常快，拉链表的数据量会大于快照表数倍，一天变一次，那么一周就保存了

7 份数据，可以考虑把已经失效的数据转移到其他的存储介质或者冷盘上，或者定期（一个月）进行删除。

## 5.3.2 明细事实表

事实表有粒度大小之分，基于数据仓库层次架构，**明细事实表一般存在于 DWD 层**，该层事实表设计不进行聚合、汇总动作，仅做数据规范化、数据降维动作，同时数据保持业务事务粒度，确保数据信息无丢失。

#### DWD 层与业务强相关，DWD 层的表就是业务表经过一系列规范化、降维之后的表。

1. 数据降维

**为了提高模型易用性，将常规维度表中的常用的属性数据冗余到相应的事实表中，从而在使用的时候避免维表关联的方式**，既为数据降维。

例如，在业务中，有比较频繁的应用场景，即分析各商品类别的销量、按区域分析购买力等，如果将商品种类、用户常住地等属性放入事实表中，当分析各商品类别的销量、按区域分析购买力等指标时就无需再关联其他维度表。

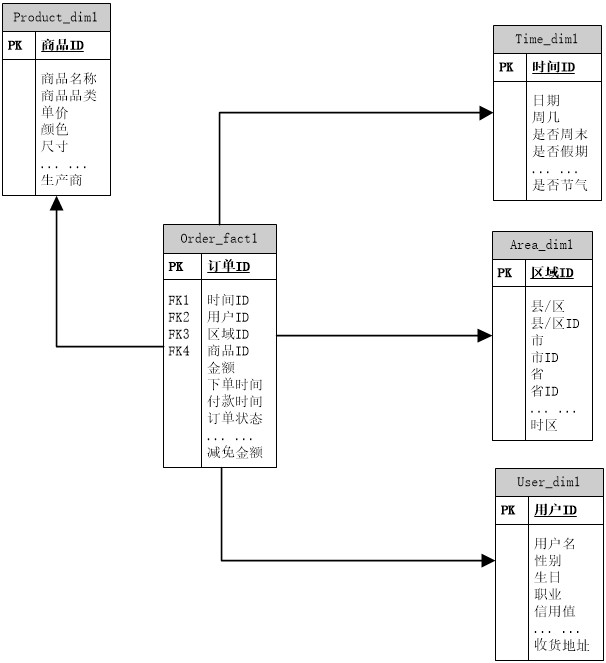
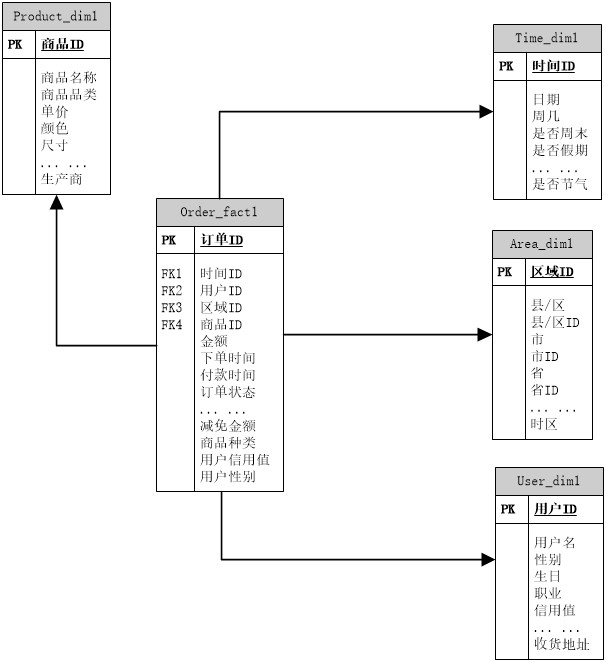


图 5-2 原始模型



1. 独立维度的选择

图 5-3 数据降维后的模型

并不是你业务中遇到的每一个实体都要成为一个独立的维度，具体哪些维度可以合并， 要根据实际的业务场景来确定，比如对于出行行业，司机一定是一个独立维度，而汽车这个实体就没有必要称为独立的维度（除非要分析订单的取消与汽车品牌的关系），因此可以汽车信息和司机信息进行合并。

### 事实表不一定有事实

#### 一般将事实表中包含两部分信息：维度、度量，度量即为事实，但有些特殊情况下， 事实表中无度量信息，只是记录一个实际业务动作。

比如，信息审核表：



图 5-4 信息审核表

对于出行行业，用户打车会下一单，后台系统根据下单信息筛选司机，筛选司机是一个事实，筛选司机后指派司机，这个指派过程是一个事实，司机接单是一个事实，这些事实有些伴随着可以度量的值，有些则没有，司机接客成功、司机开始送客、行程结束后乘客支付、乘客评价、乘客投诉都可以归为事实，但是都可能没有度量值。

### 明细事实表设计方案

设计事实表的主要依据是业务过程，之前说过，**每一个业务动作事件，都可以作为一个事实**，那么在一个订单处理过程中，会有多个动作，这个过程中的事实表怎么设计呢？

方案一：单事件事实表

#### 对于每一个业务动作事件，设计一个事实表，仅记录该事件的事实以及状态。（一个业务流程多个单事件事实表）

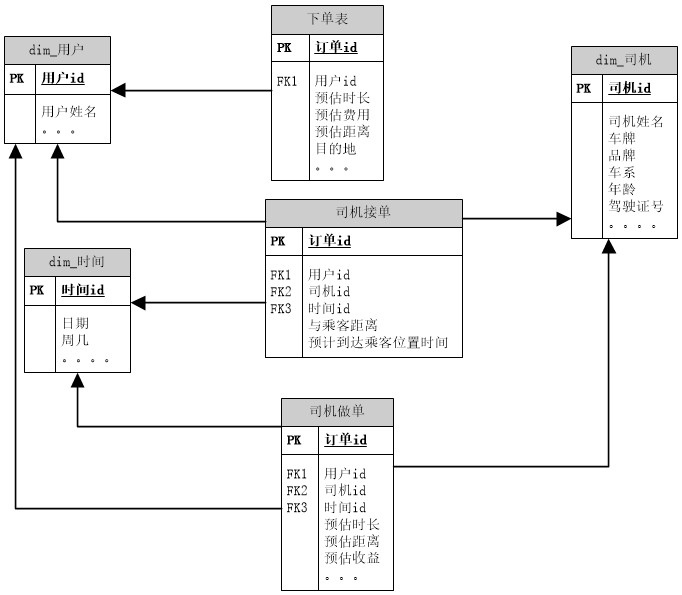
方案二：流程事实表

#### 对于一个业务流程主体，设计一个事实表，跟踪整个流程的事实以及状态流转。

1. 明细事实表设计案例

方案一：单事件事实表

出行领域，用户下单打车，该订单的整个流程包括用户下单、司机接单、司机做单、乘客支付，可能还伴随有评价、投诉等环节，这个场景下的明细事实表怎么设计？

图 5-5 网约车订单单事件事实表

按照上图的方案，针对订单这个模块，有大量的事实表，但是也有好处，这种设计方法会使得在分析时非常明确，这样设计不存在历史订单状态变化的问题，比如 6 月 1 号 23 点

30 下单，23 点 40 司机接单，6 月 2 号 0 点到达目的地，0 点 5 分用户支付，0 点 10 分用户评价，这种跨零点的订单，由于事实表划分明确，单个事件发生在次日，此时的历史订单状态变化的问题就会很少，接近于无状态模型，分析就会相对简单，并且多个事实表中记录的细节更多，把每一个环节的细节信息记录的非常全面。但是，追踪一个订单的当前进度时， 需要去查看不同的事实表的进度，不能非常直观地看到订单状态的流转。

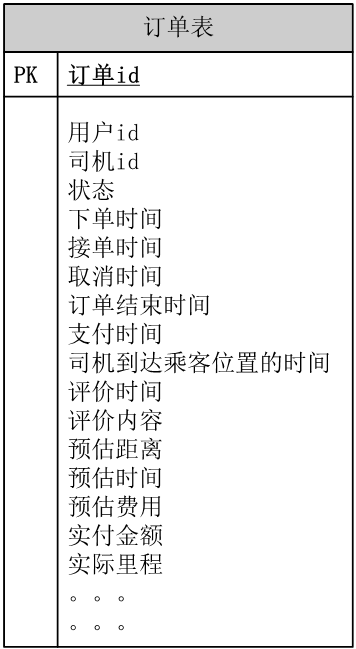
但是，**如果想通过方案一查看订单的整个运转流程，需要 join 很多表，比较复杂**。方案二：流程事实表

图 5-6 网约车订单

上图中的流程信息表不能记录太多细节性的内容，状态记录订单的流转状态，并且记录关键节点的时间，其他时间没有，如果想拿订单流转过程中的一些状态是拿不到的。

#### 这种表的好处是可以很直观的看到订单状态，比较容易跟踪订单的整个运转流程，但是会丢失细节。

为了保证数据的灵活性，可以考虑在业务侧维护方案一的数据内容，在数仓中创建方案二的宽表，也就是两种并存。

实际上，方案一的应用更为广泛，因为它的信息更加全面，分析的时候更灵活，一些特殊的场景更容易满足。

综上所述，我们可以总结出单事件事实表和流程事实表的特点： **单事件事实表**：

* 更方便跟踪业务流程细节数据，针对特殊的业务分析场景比较方便和灵活，数据处理上也更加灵活；
* 不方便的地方就是数仓中需要管理太多的事实表，同时跟踪业务流转不够直观； **流程事实表**：
* 能够更直观的跟踪业务流转和当前状态，流程事实集中，方便大部分的通用分析应用场景，由于和业务侧的数据模型设计思路一致，也是目前最常用的事实表设计；但是细节数据跟踪不到位，特殊场景的分析不够灵活；

两种表的设计区别在于对业务流程的拆分思路不同，具体选择事实表的构建思路，需要根据实际的业务确定，一般建议两者结合。

## 5.3.2 聚合事实表

#### 相对于明细事实表，聚合事实表通常是在明细事实表的基础上，按照一定的粒度粗细进行的汇总、聚合操作，它的粒度较明细数据粒度粗，同时伴随着细节信息的丢失。

在数仓层次结构中，**聚合事实表通常位于 DWS 层**，一般作为通用汇总数据存在，也可以是更高粒度的指标数据。

#### 聚合事实表的数据来源可以是两种明细事实表中的任意一种。

* 日粒度
* 周期性累积（周，月，年）
* 历史累积（累计订单量、累计金额）

1. 可累加事实与不可加事实
   1. 可累加事实

可累加事实是在一定的粒度范围内，可累加的事实度量，比如：订单金额、订单数。

* 1. 不可累加事实

不可累加事实是在更高粒度上不可累加的事实，比如通过率、转化率等。

通常情况下，比率这种不可累积的事实，建议拆分存储，比如通过率拆分为通过数、申请数，由细粒度数据去重计算而得到的事实，正常存储，但是更粗粒度累积是不可直接使用。

### 聚合事实表分类

* 1. 公共维度层/通用汇总层

#### 封装底层计算逻辑，做通用汇总，避免上层直接访问下层明细数据。

应对大部分可预期的、常规的数据需求，通常针对模式相对稳定的分析、BI 指标计算、特征提取等场景，封装部分业务处理、计算逻辑，尽量避免用户直接使用底层明细数据，该层用到的数据范围比较广泛。

通用汇总层需要满足 80%~90%的场景，对数据进行轻度汇总，避面直接访问明细层， 假设明细层有 1 亿条数据，这一层可能只有 1 千万条。

* 1. 日粒度

主要应对模式稳定的分析、BI 日报、特征提取场景，同时日粒度也为后续累积计算提供粗粒度的底层，数据范围一般为上一日的数据。

对可累加指标进行粗粒度的统计，周、月等粒度的统计可以在日粒度基础上计算，假设明细层 1 亿条数据，这一层可能只有 1 百万条。

* 1. 周期性累积

主要应对明确的周期性分析、BI 周期性报表，数据范围一般在某周期（周、月等）内的。底层数据可以来自于公共维度层-通用汇总，也可以来自于日粒度。

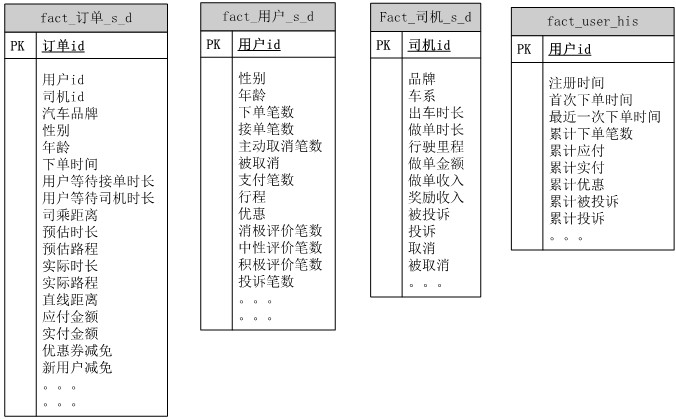
* 1. 历史累积

顾名思义，历史以来某一特定数据的累积，通常在用户画像、经营分析、特征提取方面场景较多，设计数据范围比较广泛，通常是计算耗时较长的一部分，比如某门店累积营业额、某用户累积利润贡献、用户首次下单时间(非可度量、描述性)。

### 聚合事实表案例

公共维度层（订单维度、用户维度、司机维度...）：

在这个粒度，尽量覆盖更多的内容，在很多业务场景偏复杂的情况下，公共维度层能够达到 100 多个字段。

图 5-7 聚合事实表

日粒度：

统计一天的数据，日粒度层与公共维度层并列。

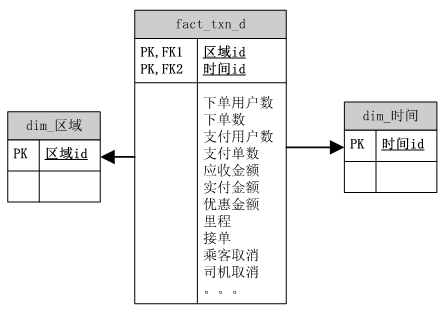


图 5-8 日粒度表

周期性累积：

统计一定周期的数，同时可以处理一些不可累积的数据，例如比率等。

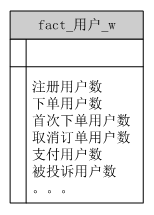


图 5-9 周期性累积表

历史累积：

使用增量数据与历史数据进行累积。

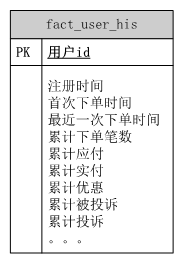


图 5.10 历史累积

# 第六章 数据仓库规范

## 命名规范

* 根据业务过程，抽象基本术语单元；
* 对相应的术语单元做语义翻译，可以采用拼音、英文、含义数字，避免英文、拼音混用；
* 拼音、英文尽量在不失原意的情况下采用缩写形式；
* 避免数字开头；

表 6-1 命名规范示例

|  |  |  |
| --- | --- | --- |
| 语义 | 翻译 | 缩写 |
| 订单 | order | ord |
| 增量 | increment | inc |
| 历史 | history | his |
| 累积 | total | total |
| 天 | day | d |
| 周 | week | w |
| 月 | month | m |
| 年 | year | y |
| 用户 | user | user |

|  |  |  |
| --- | --- | --- |
| 账单/还款计划 | bill | bill |
| 金额 | amount | amt |
| 数量 | count | cnt |
| 手机 | mobile | mob |
| 年龄 | age | age |

## 6.1.1 表命名规范

数据仓库中表格的命名规范如下表所示：

表 6-2 表命名规范

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 数据域 | | 数仓层次 | | 周期/数据范围 | |
| 订单 | ord | 共用维度 | dim | 日快照 | d |
| 用户 | user | 集市 | dm | 增量 | i |
| 财务 | finc | ODS | o | 周 | w |
| 账单 | bill | DWD | d | 拉链表 | l |
| 库存 | sto | DWS | s | 非分区全量表 | a |

在表格的命名中，我们要求能够合理的区分出表所描述的数据域、数据周期等。命名格式：**层次\_数据域\_修饰**/**描述\_范围**/**周期**

根据表的命名规范，我们对以下的数据表进行命名：

* + 1. 订单相关数据表：

DWD 层：d\_ord\_info\_d

DWS 层：s\_ord\_st\_d

* + 1. 维度表：

用户维度：dim\_user\_d

商品缓慢渐变维表：dim\_product\_l

* + 1. ODS 层表：

（**对于 ODS 层表，最好能够区分数据来源，包括来自什么系统、数据源名称**） 业务系统编码：buss

业务系统订单表：loan\_order

Ods 层表命名：o\_buss\_loan\_order\_d

## 字段命名规范

设计模型的时候，按照业务含义、业务术语规范命名字段汉字名称。

* 避免数字开头
* 同一业务含义统一命名，避免不同表达方式
* 统一书写格式，比如用户 id:user\_id，用户姓名：userName 非统一格式
* 统一大小写，建议统一小写
* 避免与关键字、自定义 udf 重名

根据字段命名对字段进行命名，如下表所示：

表 6-3 字段命名规范

|  |  |
| --- | --- |
| 名称 | 字段名称 |
| 用户 ID | user\_id |
| 应还金额 | rep\_amt |
| 实还金额 | act\_amt |
| 逾期利息 | overdue\_int |
| 下单计数 | ord\_cnt |
| 月下单计数 | m\_ord\_cnt |
| 累积下单用户计数 | total\_ord\_user\_cnt |

## 脚本命名规范

数据仓库中会使用到很多自动化脚本，脚本的命名规则如下：

* ETL 脚本名称尽可能和所产出的表同名
* 数据采集、数据推送脚本尽可能标识数据去向
* ETL 脚本若产生多个表，采用对应的数据域和语义描述命名
* Jar 包命名以实际的业务处理逻辑语义描述为主，调度任务命名同样尽量以产出表名命名

根据脚本命名规范，我们对如下的脚本进行命名：

1. 数据采集过程

采集数据到 ODS 层的表 o\_buss\_loan\_order\_d，我们按照输出表 d\_ord\_ino\_d 对脚本和任务进行命名：

数据采集脚本命名：imp\_o\_buss\_loan\_order\_d.sh、imp\_o\_buss\_loan\_order\_d.py

1. 订单 ETL 过程
2. 2020/12/9/order/ODS
3. 2020/12/9/USER/ODS

从 ODS 层的表 o\_buss\_loan\_order\_d 整理数据并且装载到 DWD 层表 d\_ord\_ino\_d 中， 我们按照输出表 d\_ord\_ino\_d 对脚本和任务进行命名：

ETL 脚本命名： d\_ord\_info\_d.sh、d\_ord\_info\_d.py、d\_ord\_info\_d.hql、d\_ord\_info\_d.jar

ETL 任务命名：d\_ord\_info\_d

1. 一个 ETL 脚本产出多个表，比如从商品表中分离出商品维度、厂家维度：

ETL 脚本命名：dim\_product\_mfrs\_d.sh、dim\_product\_mfrs\_d.py、dim\_product\_mfrs\_d.hql、

dim\_product\_mfrs\_d.jar

ETL 任务名称：dim\_product\_mfrs\_d

## 开发规范

* + - 数仓中 MR 程序尽可能统一输入参数、输出参数，单个 jar 程序的功能模块清晰， 避免多种处理逻辑写入一个 jar 包；
    - 每个 ETL 脚本尽可能产出一张数仓表，方便任务排查，同时也减少数仓表的耦合性；
    - ETL 脚本格式、备注清晰，避免大范围、格式杂乱的脚本，合理利用临时表

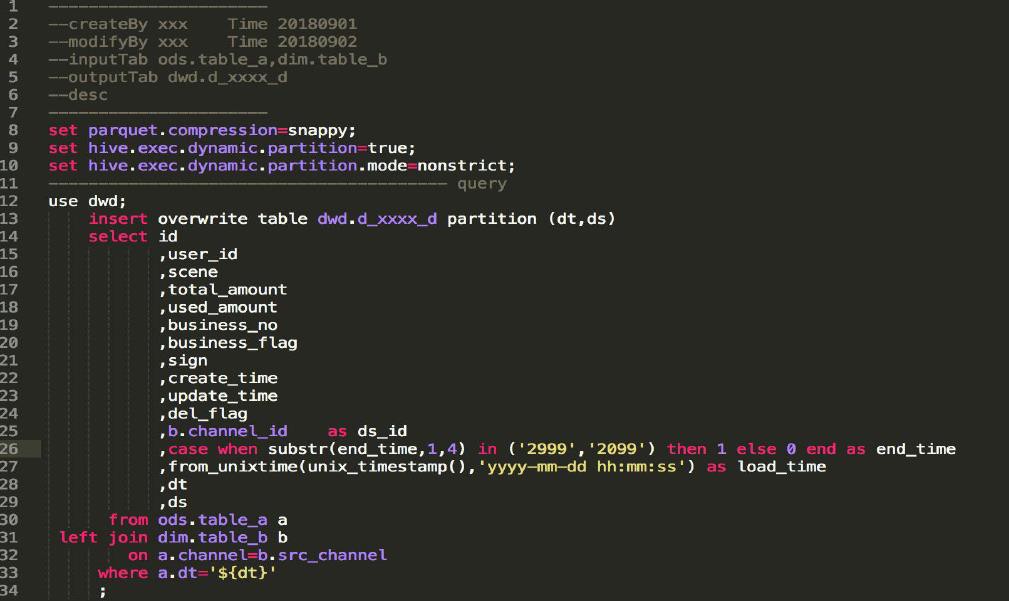
1. 字段列对齐；
2. 关键字列对齐；
3. 禁止使用 Tab，全部使用 4 个空格代替； 标准的 ETL 脚本格式如下图所示：

图 6-1 ETL 脚本格式

# 总结

数据的采集过程，

数仓的分层：ETL过程：python脚本。 hive脚本，sqoop迁移，oozie,azkaban,任务调度系统。

数据仓库是企业管理、维护和分析数据的关键体系，通过数据仓库的建立实现了企业内部数据的整合，并为不同的业务部门提供了同一的数据出口。

本项目深入探讨了数据仓库的基本理论、数据仓库建模基础、数据仓库数据采集与同步、数据仓库维度建模以及数据仓库规范，对数据仓库中的理论知识进行了详细梳理，对数据仓库中涉及到的各类概念进行了详细解析，并通过穿插于概念之间的实例展示了各个概念具体的实现方式。

希望通过对于本课程的学习，让学生掌握数据仓库的基本概念与搭建技巧，能够根据企业实际的业务需求对数据仓库进行维护或者创建新的数据仓库体系。