数据仓库

什么是数据仓库？

- 数据仓库已被多种方式定义但没有一种严格的定义
  - 为统一的历史数据分析提供坚实平台，对信息处理提供支持
- “数据仓库是一个面向主体的、集成的、时变的、非易失的数据集合，支持管理过程的决策过程”—**W. H. Inmon**
- 建立数据仓库
  - 构造和使用数据仓库的过程

数据仓库——面向主题的

- 围绕一些主题如顾客、供应商、产品和销售等而组织
- 关注于决策者的数据建模和分析，而不是集中于组织机构的日常操作和事务处理
- 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

数据仓库——集成的

- 通过集成多个异种数据源而构成
  - 关系数据库、一般文件和联机事务处理记录
- 使用数据清理和数据集成技术
  - 在不同的数据源中，确保命名约定、编码结构、属性度量等的一致性
    - 例如，旅馆价格：由住宿费、税收、附带的早餐费等等构成
  - 数据被移到数据仓库时就进行了数据转换

数据仓库——时变的

- 数据仓库的时间范围明显长于操作数据库系统
  - 操作数据库：当前的有用信息
  - 数据仓库数据：从历史的角度提供信息（例如：过去的5-10年）
- 数据仓库的每一个关键结构
  - 隐式或显示的包含时间元素
  - 但操作数据的关键结构可以包含也可以不包含“时间元素”

数据挖掘——非易失的

- 数据仓库总是物理地分离存放数据，这些数据源于操作环境下的应用数据
- 操作性的数据更新不会发生在数据仓库的环境下
  - 数据仓库不需要事务处理、恢复和并发控制机制
  - 它只需要两种数据访问：
    - *数据的初始装入和数据访问*

数据仓库和异源DBMS

- 传统的异源数据路的集成:
  - 在异源数据库的顶部建立一个包装程序和集成程序
  - 查询驱动方法
    - 当一个查询提交客户站点，首先使用元数据字典对查询进行转换，将它转换成相应异源站点上的查询，然后，不同站点返回的结果被集成为全局回答
    - 查询驱动方法需要复杂的信息过滤，并且与局部数据源上的处理竞争资源
- 数据仓库：使用更新驱动的方法，为集成的异源数据库系统带来了高性能
  - 将来自多个异源的信息预先集成，并存储在数据仓库中，供直接查询和分析

数据仓库和操作数据库系统

- **联机事务处理OLTP (on-line transaction processing)**
  - **传统的关系DBMS的主要任务**
  - **他们涵盖了一个组织的大部分日常操作：购买、库存、制造、银行、工资、注册、记账等。**
- **联机分析处理OLAP (on-line analytical processing)**
  - **数据仓库系统的主要任务**
  - **数据分析和决策**

数据仓库和操作数据库系统

- **OLTP和OLAP的区别**
  - **用户和系统的面向性:OLTP面向顾客，而OLAP面向市场**
  - **数据内容：OLTP系统管理当前数据，而OLAP管理历史的数据。**
  - **数据库设计：OLTP系统采用实体-联系（ER)模型和面向应用的数据库设计，而OLAP系统通常采用星形和雪花模型**
  - **视图：OLTP系统主要关注一个企业或部门内部的当前数据，而OLAP 系统主要关注汇总的统一的数据。**
  - **访问模式：OLTP访问主要有短的原子事务组成，而OLAP系统的访问大部分是只读操作，尽管许多可能是复杂的查询**

OLTP vs. OLAP

	OLTP	**OLAP**
用户	办事员、数据库专业人员	知识工人
功能	**日常操作**	决策支持
DB 设计	面向用户	面向主题
数据	当前的、最新的、详细的	历史的、汇总的、多维的
操作	读/写 主关键字上的索引/散列	大量扫描
工作单位	短的，简单事务	复杂查询
# 访问记录数量	数十个	数百万
#用户数	数千	数百
DB 规模	100MB-GB	100GB-TB
度量	事务吞吐量	查询吞吐量，响应时间

为什么需要一个分离的数据仓库？

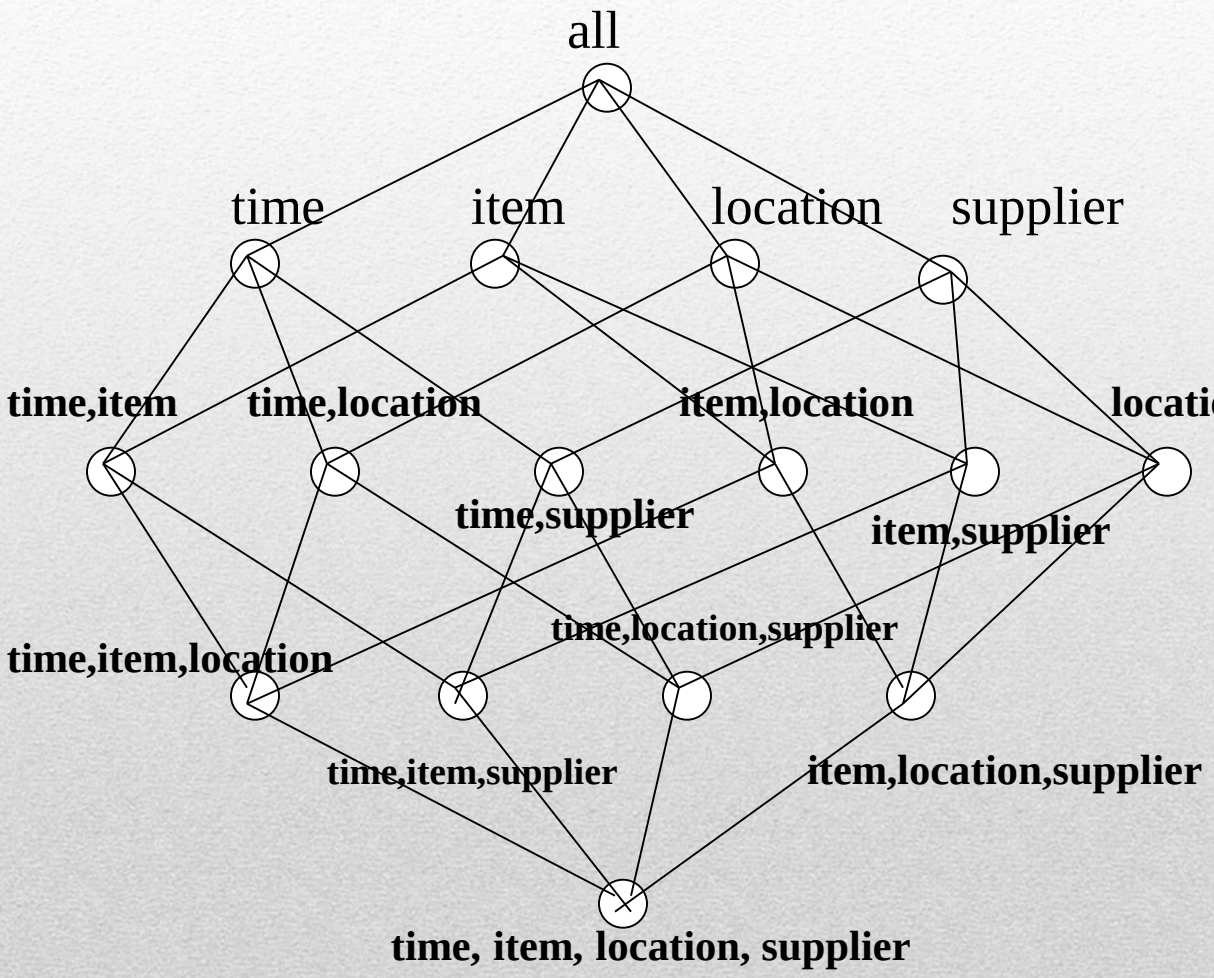
- 提高两个系统的性能
  - 数据库管理系统— OLTP的协调: 存取方法，索引，同步控制，恢复
  - 数据仓库— OLAP的协调: 复杂的OLAP查询，多维视图，合并
- 不同的功能和不同的数据:
  - 数据维护: 决策支持需要历史数据，而操作数据库一般不维护历史数据
  - 数据统一：决策支持需要将来自异源的数据统一（如聚集和汇总）
  - 数据质量: 不同的数据源通常使用不一致的数据表达，代码和形式，这些都需要协调

多维数据模型

由表和电子数据表到数据立方体

- 一个数据仓库建立在多维数据模型上，多维数据模型把数据看成数据立方体的形式
- 一个数据立方体，像**sales**，允许以多维对数据建模和观察
  - 维表，例如**item**的维表包含属性**(item_name, brand, type), time**的维表包含属性**(day, week, month, quarter, year)**
  - 事实表包含度量 (例如**dollars_sold**) 和每个相关维表的关键字
- 数据仓库语义中, 一个 **n-D** 底层方体称为**基本方体**。最高层的**0-D**方体, 存放最高层的汇总, 称为**顶点方体**。所有的方体格组成了数据立方体。

立方体：一个方体格



0-D(顶点）方体

1-D 方体

2-D 方体

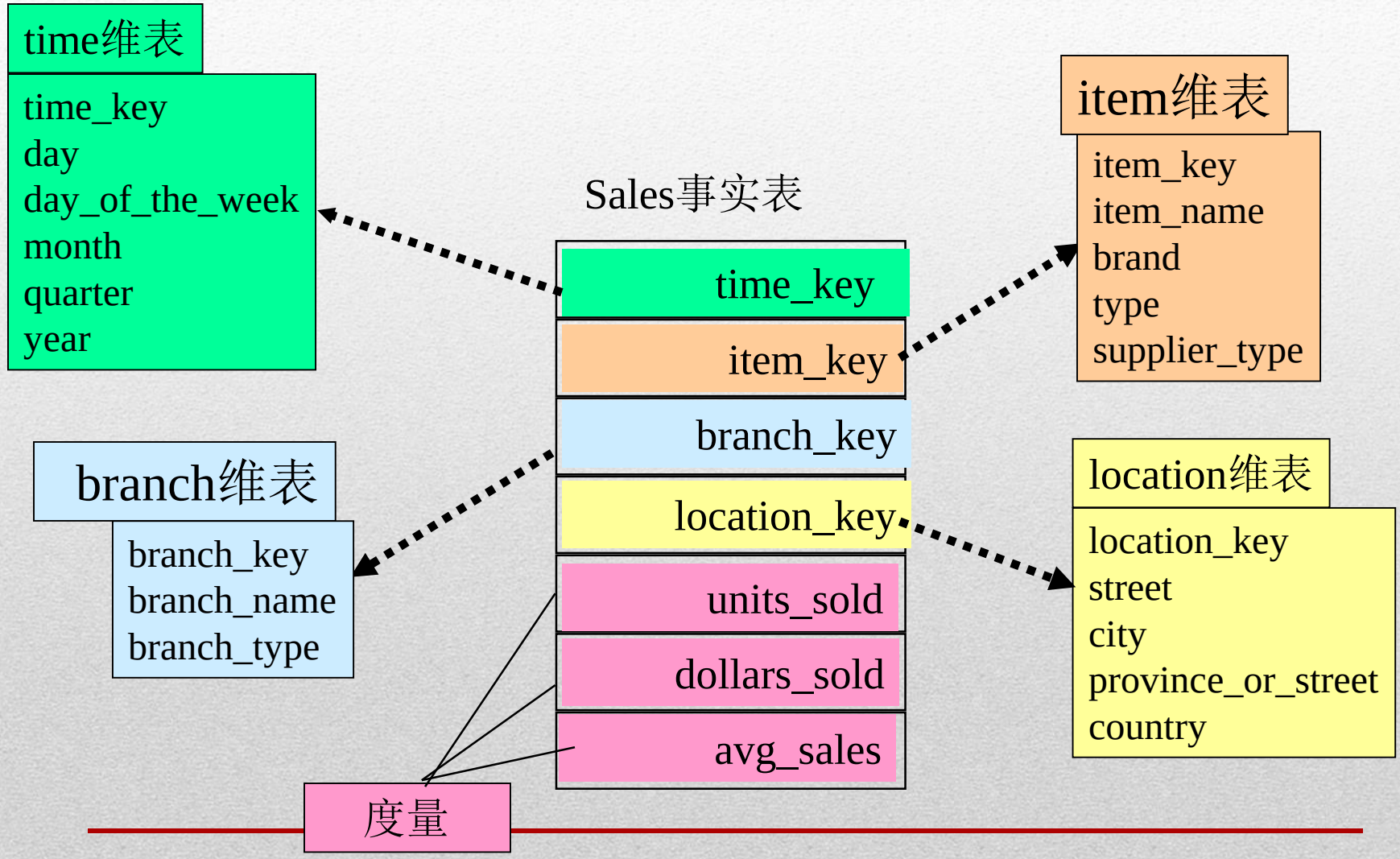
3-D 方体

4-D(基本) 方体

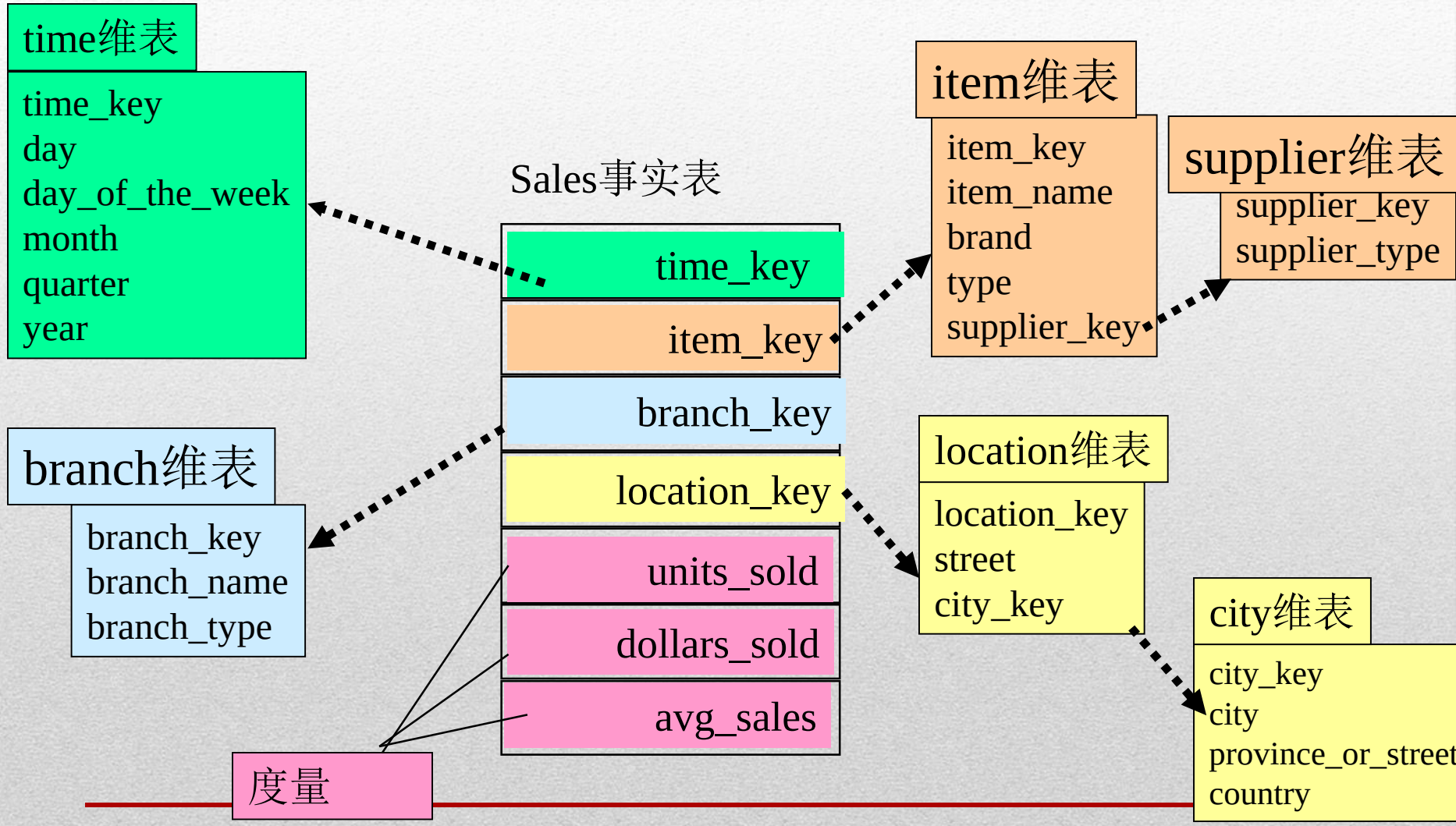
数据仓库的概念性模型

- 建立数据仓库模型：维与度量
  - 星型模型: 中间是事实表，连接一组维表
  - 雪花模式: 雪花模式是星型模式的变种，其中某些维表示规范化的，而数据进一步分解到附加的维表中，它的图形类似于雪花的形状
  - 事实星座表: 多个事实表共享维表，这种模式可以看作星型模式的集合，因此称为星系模式或事实星座

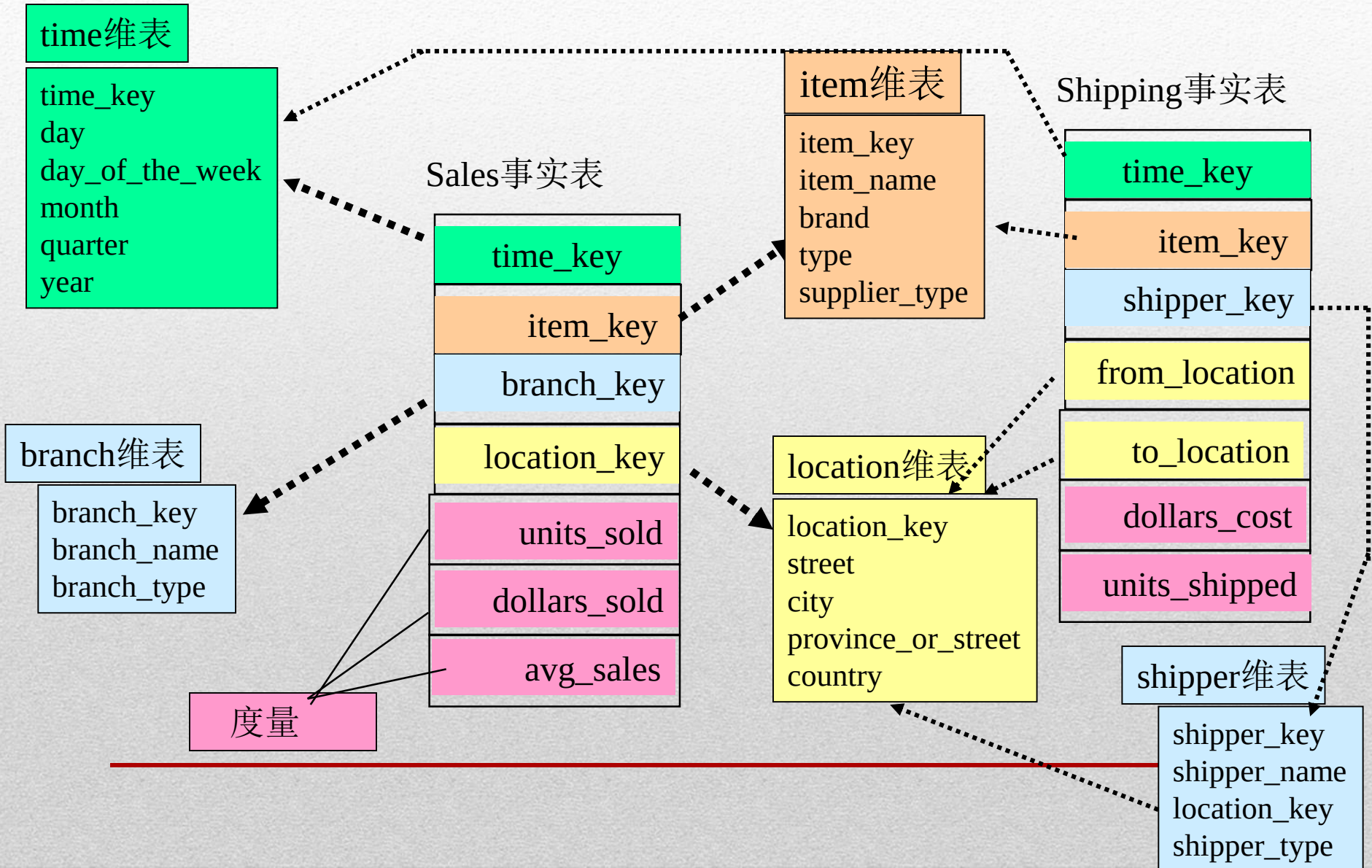
星型模式的例子


time维表
time_key
day
day_of_the_week
month
quarter
year
Sales事实表
time_key
item_key
branch_key
location_key
units_sold
dollars_sold
avg_sales
item维表
item_key
item_name
brand
type
supplier_type
branch维表
branch_key
branch_name
branch_type
location维表
location_key
street
city
province_or_street
country
度量

雪花模式的例子


time维表
time_key
day
day_of_the_week
month
quarter
year
Sales事实表
time_key
item_key
branch_key
location_key
units_sold
dollars_sold
avg_sales
item维表
item_key
item_name
brand
type
supplier_key
supplier维表
supplier_key
supplier_type
branch维表
branch_key
branch_name
branch_type
location维表
location_key
street
city_key
city维表
city_key
city
province_or_street
country
度量

事实星座模式的例子


time维表
time_key
day
day_of_the_week
month
quarter
year
Sales事实表
time_key
item_key
branch_key
location_key
units_sold
dollars_sold
avg_sales
item维表
item_key
item_name
brand
type
supplier_type
Shipping事实表
time_key
item_key
shipper_key
from_location
to_location
dollars_cost
units_shipped
branch维表
branch_key
branch_name
branch_type
location维表
location_key
street
city
province_or_street
country
shipper维表
shipper_key
shipper_name
location_key
shipper_type
度量

数据挖掘查询语言DMQL：语言原语

- 立方体定义（事实表）

  define cube <cube_name> [<dimension_list>]: <measure_list>

- 维定义（维表）

  define dimension <dimension_name> **as** (<attribute_or_subdimension_list>)

- 特殊情况（共享维表)
  - 首先进行“立方体定义”
  - **define dimension** <dimension_name> **as** <dimension_name_first_time> **in cube** <cube_name_first_time>

用DMQL定义星型模式

define cube sales_star [time, item, branch, location]:

dollars_sold = sum(sales_in_dollars), avg_sales = avg(sales_in_dollars),
units_sold = count(*)

define dimension time as (time_key, day, day_of_week, month, quarter, year)

define dimension item as (item_key, item_name, brand, type, supplier_type)

define dimension branch as (branch_key, branch_name, branch_type)

define dimension location as (location_key, street, city, province_or_state, country)

用DMQL定义雪花模式

define cube sales_snowflake [time, item, branch, location]:

dollars_sold = sum(sales_in_dollars), avg_sales = avg(sales_in_dollars),
units_sold = count(*)

define dimension time as (time_key, day, day_of_week, month, quarter, year)

define dimension item as (item_key, item_name, brand, type, supplier(supplier_key, supplier_type))

define dimension branch as (branch_key, branch_name, branch_type)

define dimension location as (location_key, street, city(city_key, province_or_state, country))

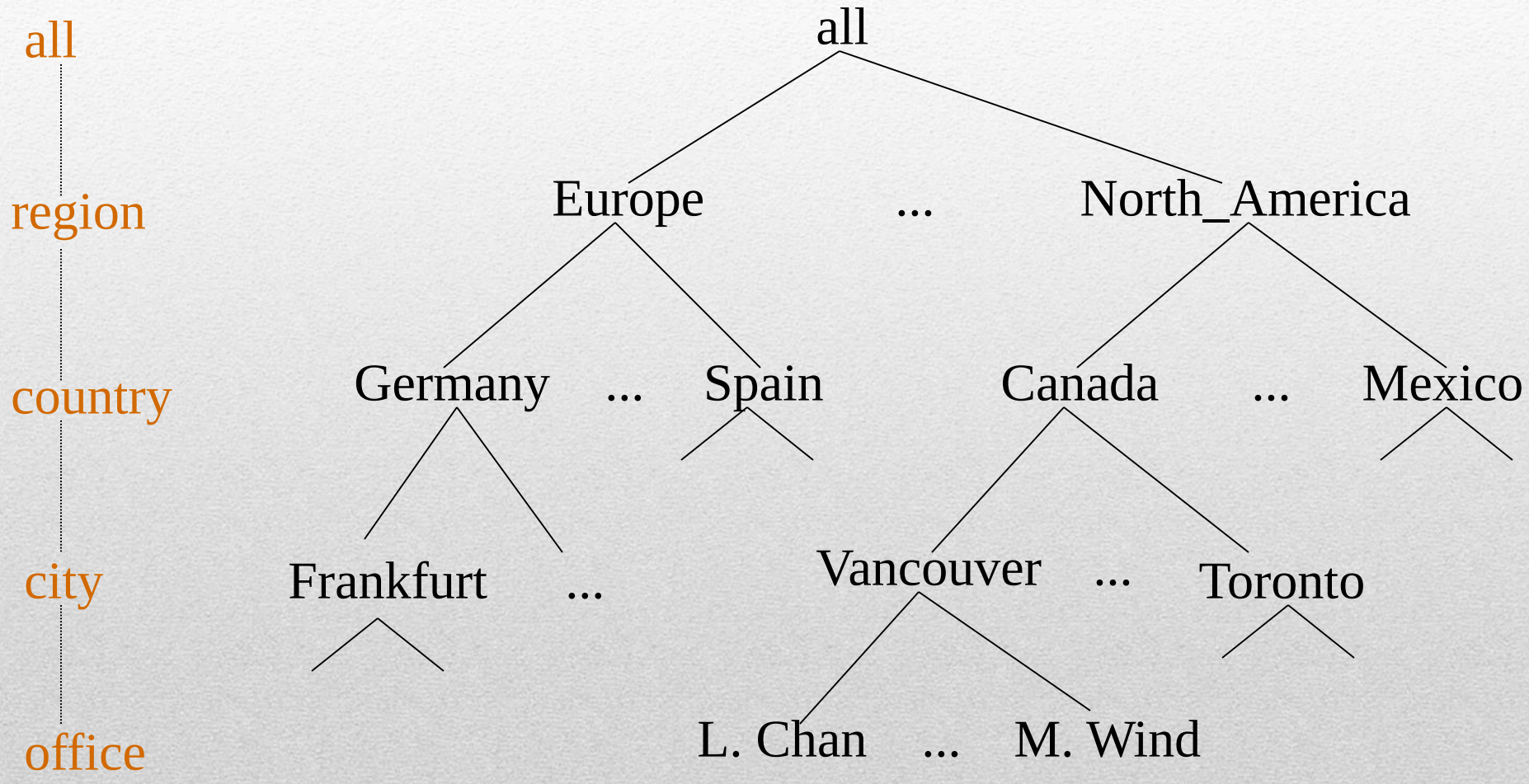
用DMQL定义星系模式

```
define cube sales [time, item, branch, location]:
        dollars_sold = sum(sales_in_dollars), avg_sales = avg(sales_in_dollars),
            units_sold = count(*)
define dimension time as (time_key, day, day_of_week, month, quarter, year)
define dimension item as (item_key, item_name, brand, type, supplier_type)
define dimension branch as (branch_key, branch_name, branch_type)
define dimension location as (location_key, street, city, province_or_state, country)
define cube shipping [time, item, shipper, from_location, to_location]:
        dollar_cost = sum(cost_in_dollars), unit_shipped = count(*)
define dimension time as time in cube sales
define dimension item as item in cube sales
define dimension shipper as (shipper_key, shipper_name, location as location in cube
    sales, shipper_type)
define dimension from_location as location in cube sales
define dimension to_location as location in cube sales
```

度量的三种分类

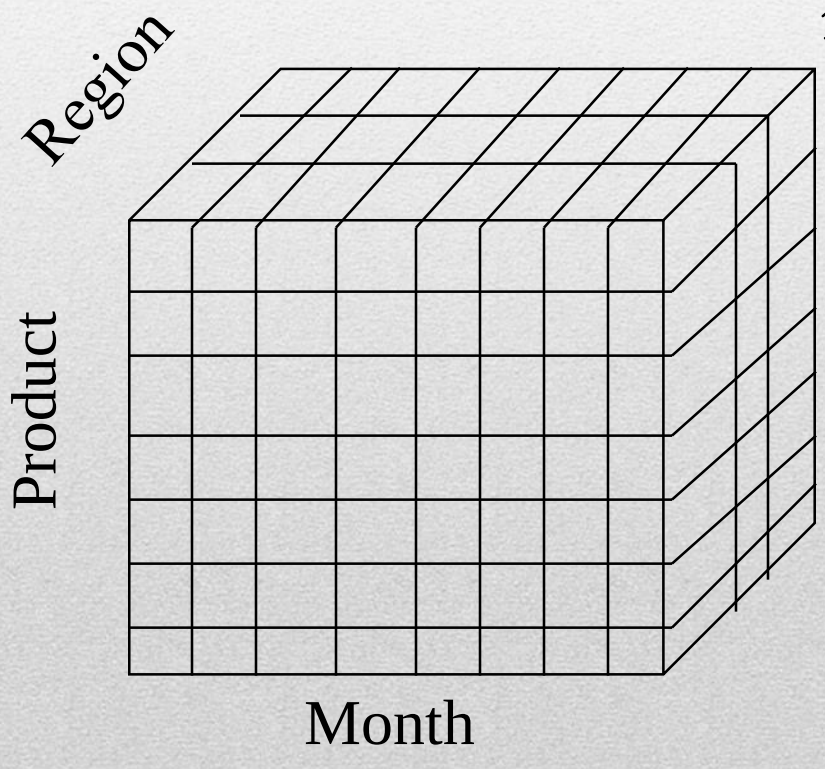
- 分布的（distributive)：可以将数据集分成较小的子集，然后计算每一个子集的度量，最后合并计算结果，得到整个数据集的度量。
  - 例如, count(), sum(), min(), max().
- 代数的（algebraic）：通过一个代数函数或者分布式、多个分布式计算的度量。
  - 例如，avg()=sum/count, min_N(), standard_deviation().
- 整体的（holistic）：对整个数据集度量，不能通过划分子集并合并子集来度量。
  - 例如, median(), mode(), rank().

Location维的概念分层

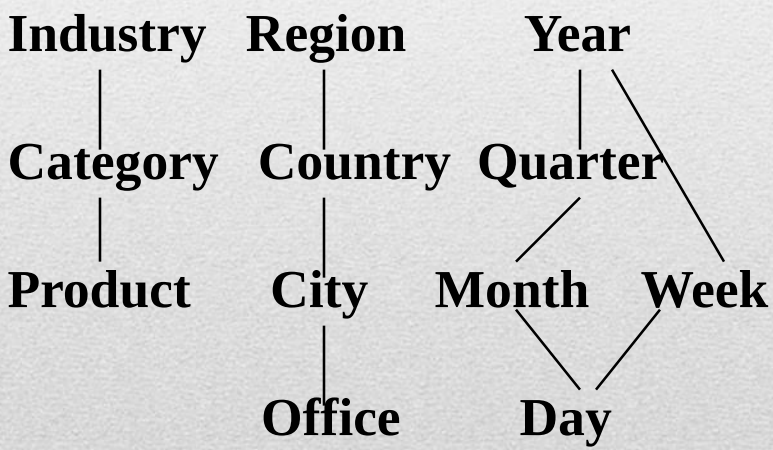


多维数据

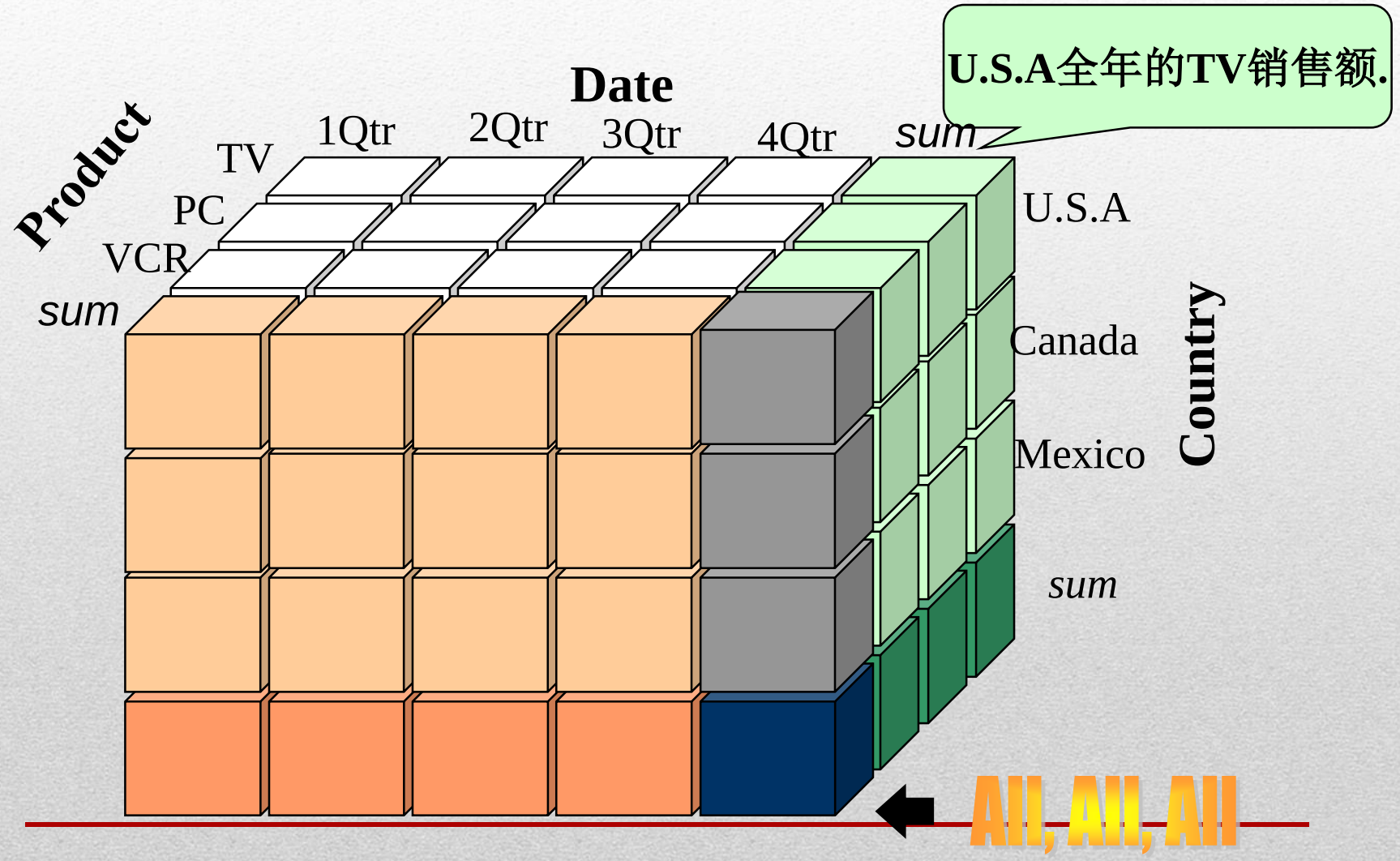
- 具有 **product, month, and region**三个维的销售立方体



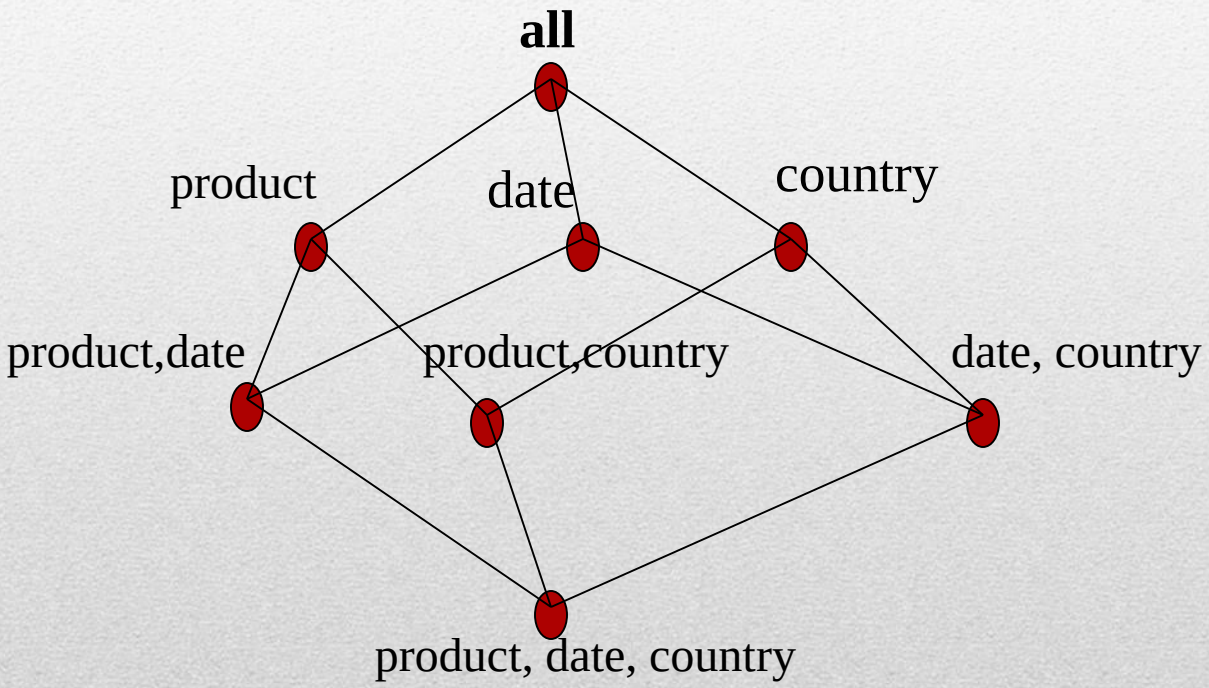
维: Product, Location, Time
分层的汇总路径



数据立方体的例子


U.S.A全年的TV销售额.
Date
1Qtr
2Qtr
3Qtr
4Qtr
sum
Product
TV
PC
VCR
sum
U.S.A
Canada
Mexico
sum
Country
All, All, All

对应于立方体的方体格



0-D(顶点) 方体

1-D方体

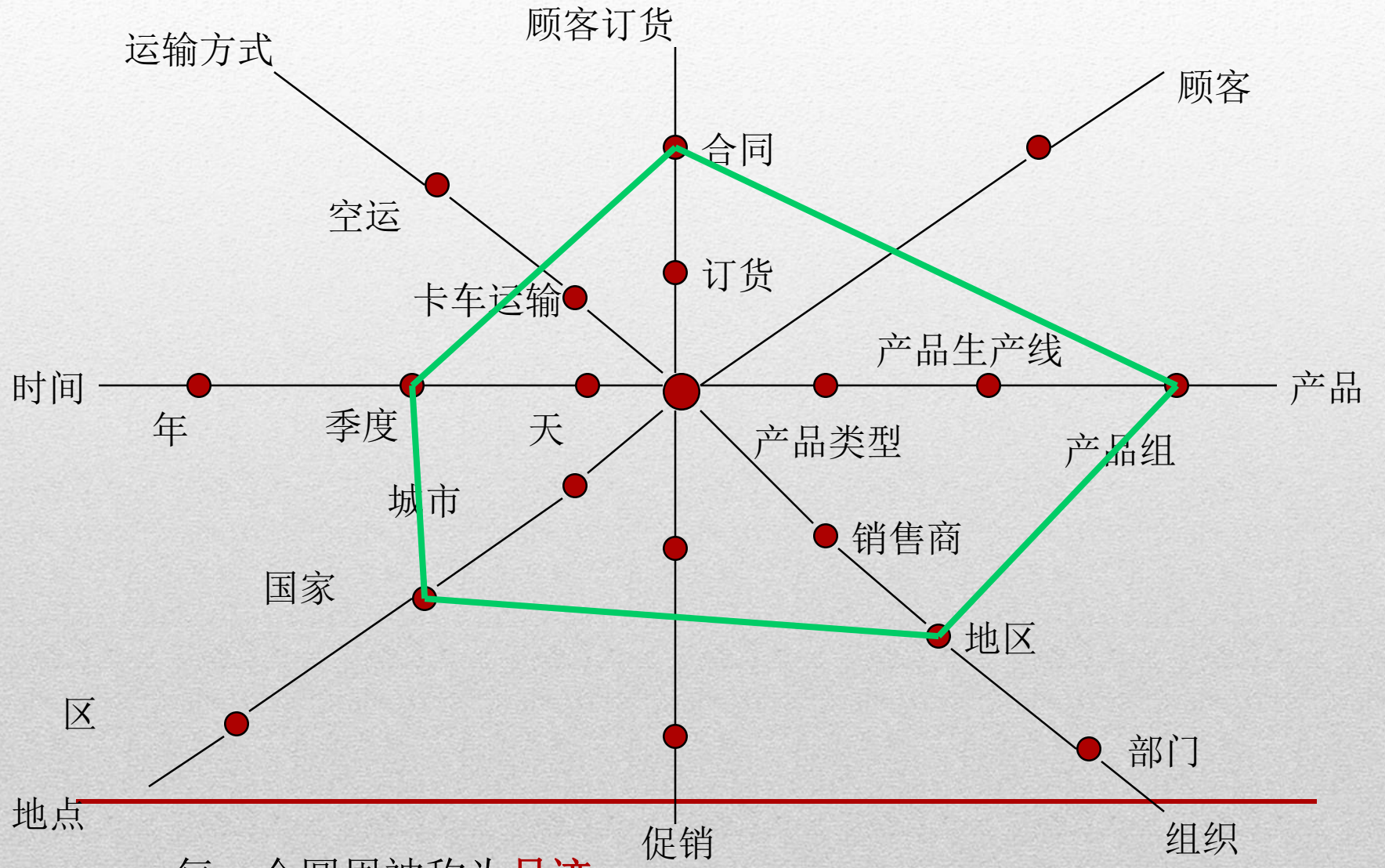
2-D方体

3-D(基本) 方体

典型的OLAP操作

- 上卷（Roll up）：汇总数据
  - 通过维的概念分层向上攀升或者通过维归约来实现
- 下钻（roll down)：上卷的逆操作
  - 从高层的汇总到低层汇总或详细数据，或者引入新的维来实现
- 切片（Slice ）和切块（ dice）：
  - 映射和选择
- 转轴（Pivot）：
  - 是一种目视操作，它转动数据的视角，提供数据的替代表示。如：将一个3-D立方体转换成2-D平面序列.
- 其他的操作：
  - 钻过（drill across)：涉及多个事实表的查询
  - 钻透（drill through）：钻到数据立方体的底层，到后端关系表（使用SQL）

一个星型网的查询模型



每一个圆周被称为**足迹**

数据仓库的系统结构

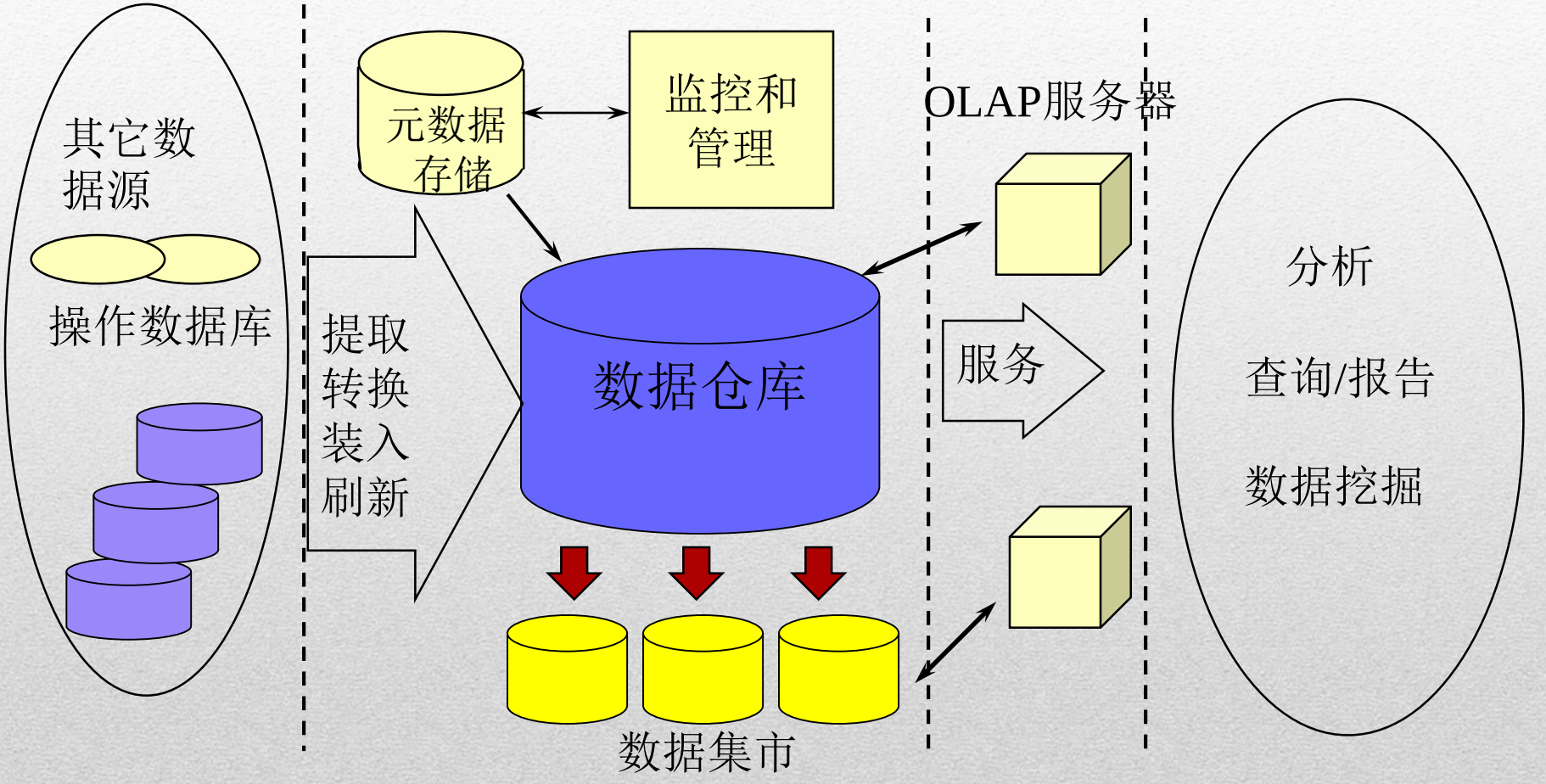
数据仓库的设计：一个商务分析框架

- 关于数据仓库设计的四种视图
  - 自顶向下视图
    - 允许选择数据仓库所需的相关信息
  - 数据源视图
    - 揭示被操作数据库系统捕获、存储和管理的信息。
  - 数据仓库视图
    - 由事实表和维表构成
  - 商务查询视图
    - 从最终用户的角度透视数据仓库的数据

数据仓库的设计过程

- 使用自顶向下方法、自顶向上方法或二者结合的混合方法设计。
  - 自顶向下：由总体设计和规划开始 (当技术成熟并以掌握，这种方法是有用的)
  - 自底向上方法: 以实验和原型开始 (在商务建模和技术开发的早期阶段，这种方法是有用的)
- 从软件工程的观点
  - 瀑布式方法: 在进行下一步前，每一步都进行结构化和系统的分析
  - 螺旋式方法: 涉及功能渐增的系统的快速产生，相继版本的时间间隔很短
- 典型的数据仓库设计过程
  - 选取待建模的商务处理，例如：订单，发表等
  - 选取商务处理的粒度
  - 选取用于每个事实记录的维
  - 选取安放在事实表中的度量

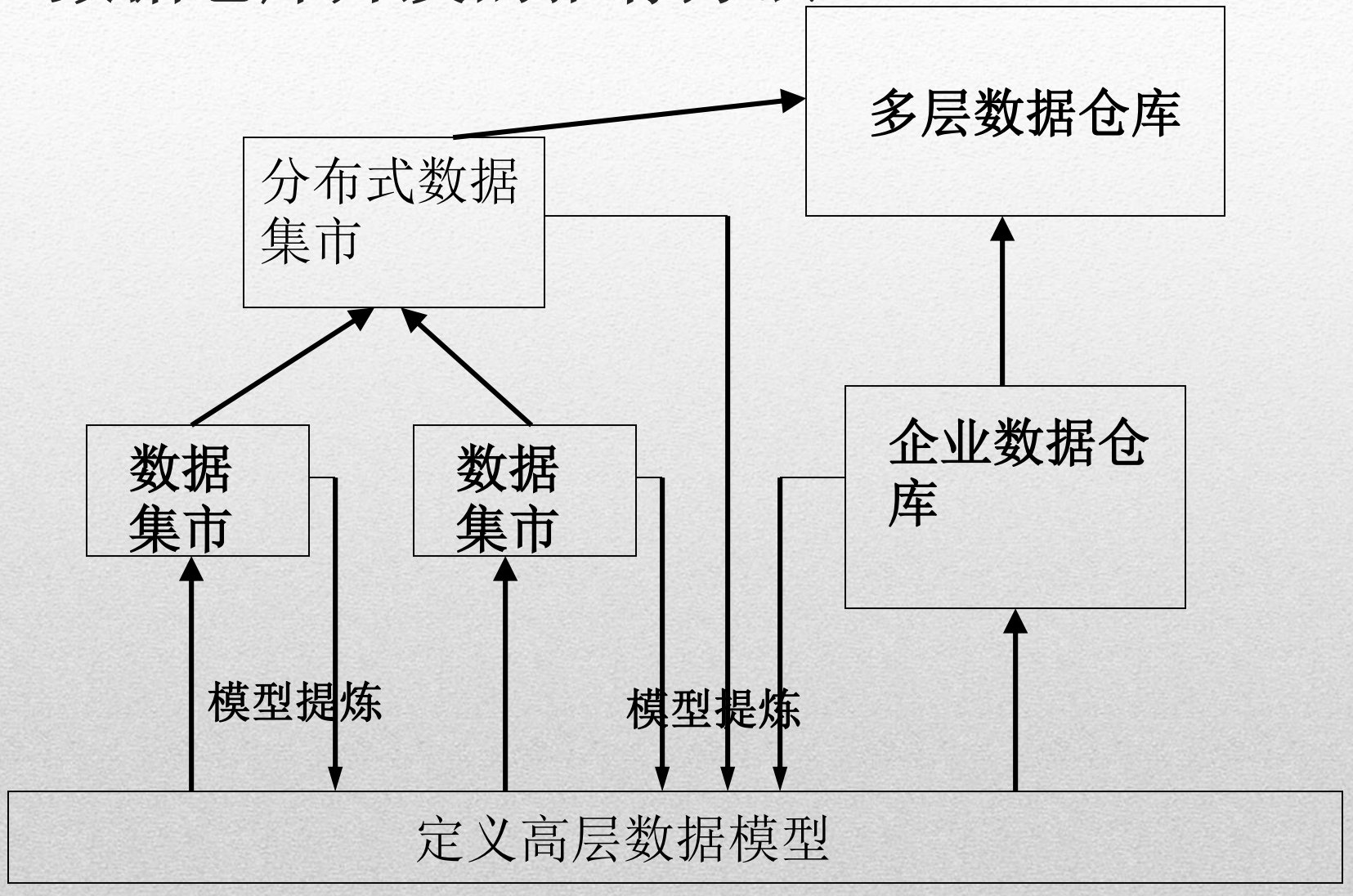
多层体系结构


其它数据源
操作数据库
元数据存储
监控和管理
OLAP服务器
提取
转换
装入
刷新
数据仓库
服务
分析
查询/报告
数据挖掘
数据集市
数据源
数据存储
OLAP引擎
前端工具

三种数据仓库模型

- 企业仓库
  - 搜集了关于主题的所有信息，跨越整个组织
- 数据集市
  - 包含企业范围数据的一个子集，对于特定的用户是有用的，其范围限于特定的、选定的用户，如：商场的数据集市。
    - 独立的数据集市和依赖的数据集市（直接来自数据仓库）
- 虚拟仓库
  - 操作数据库上视图的集合
  - 只有一些可能的汇总视图被物化。

数据仓库开发的推荐方法



OLAP服务器类型

- 关系OLAP (ROLAP)
  - 使用关系和扩充关系DBMS存放并管理数据仓库，而OLAP中间件支持其余部分。
  - 包括每个DBMS后短的优化，聚集导航逻辑的实现，和附加的工具和服务
  - 更大的可伸缩性
- 多维 OLAP (MOLAP)
  - 基于数组的多维存储引擎（稀疏矩阵技术）
  - 对预计算的汇总数据的快速索引
- 混合OLAP (HOLAP)
  - 用户的灵活性，例如，低层次：相关的，高层次：数组
- 特殊的SQL服务器
  - 在星型和雪花模式上支持SQL查询

数据预处理

数据预处理

- 为什么要预处理数据?
- 数据清理
- 数据集成和变换
- 数据归约

为什么进行数据预处理?

- 现实中的数据都是杂乱无章的
  - 不完整的：有些感兴趣的属性缺少属性值，或仅包含聚集数据
  - 含噪声的: 包含错误或孤立点值
  - 不一致的: 在代码或名称上存在差异
- 没有数据的质量就没有挖掘结果的质量!
  - 有质量的决定必须建立在有质量的数据上
  - 数据仓库需要高质量数据的一致集成

数据质量的多方位度量

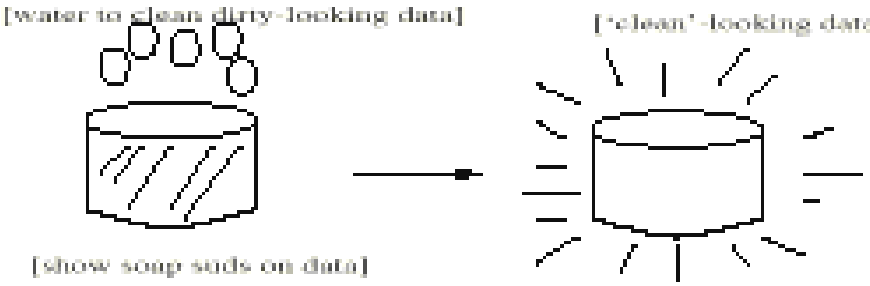
- 一个普遍接受的多方位度量：
  - 准确性
  - 完整性
  - 一致性
  - 合时性
  - 可信性
  - 可解释性
  - 可实现性
- 一个广义范畴：
  - 内在的，承接的，有代表性的和可实现的

数据预处理的主要任务

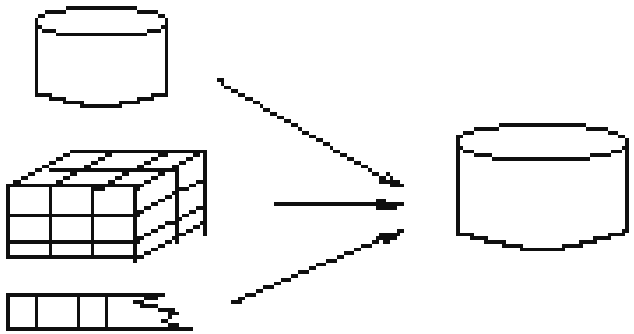
- 数据清理
  - 填写空缺的值，平滑噪声数据，识别、删除孤立点，并解决不一致
- 数据集成
  - 多数据库，数据立方体或文件的集成
- 数据变换
  - 规格化和聚集
- 数据归约
  - 得到数据集的压缩表示，它小的多，但能够产生同样或相似的分析结果
- 数据离散化
  - 数据归约的一部分，但具有特殊的重要性，特别是对数字数据

数据预处理的形式

Data Cleaning



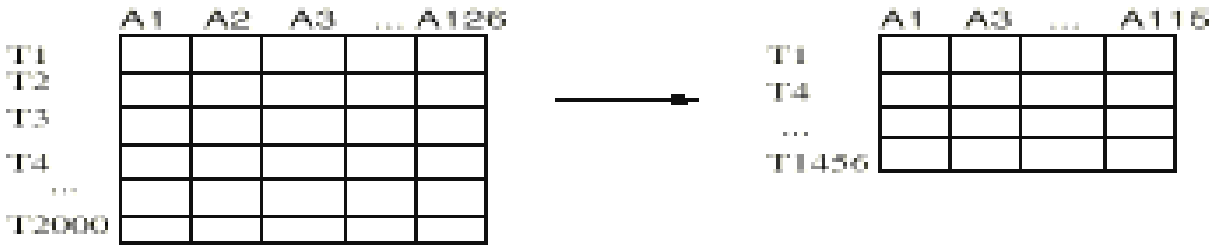
Data Integration



Data Transformation

-2, 32, 100, 59, 48 → -0.02, 0.32, 1.00, 0.59, 0.48

Data Reduction



数据预处理

- 为什麽要预处理数据?
- 数据清理
- 数据集成和变换
- 数据归约

数据清理

- 数据清理的任务
  - 填写空缺值
  - 识别孤立点和平滑噪声数据
  - 纠正不一致数据

空缺值

- 数据不总是可用的
  - 许多元组的一些属性没有记录值。如销售数据中的顾客收入
- 数据缺省的原因
  - 设备故障
  - 与其它记录数据的不一致而导致被删除
  - 数据因为被误解而未被输入
  - 某些数据在输入的时刻被认为是不重要的
  - 没有注册历史或者数据改变了
- 推断空缺的数据

怎样处理空缺数据?

- 忽略元组：当类标号缺少时通常这样做（假定挖掘任务涉及分类）除非元组有多个属性缺少值，否则该方法不是很有效
- 人工填写空缺值：该方法很繁琐，可能行不通
- 使用一个全局常量填写空缺值：例如：每个空缺值都用“unknown”替代?!
- 使用属性的平均值填写空缺值
- 使用与给定元组属同一类的所有样本的平均值
- 使用最有可能的值填充空缺值:基于推导的使用贝叶斯形式化方法和判定树

噪声数据

- 噪声：一个测量变量中的随机错误和偏差
- 造成错误属性值的原因：
  - 错误的数据收集手段
  - 数据输入问题
  - 数据传输问题
  - 技术限制
  - 命名习惯的不一致
- 其它需要数据清理的数据问题
  - 复制的纪录
  - 不完整数据
  - 不一致的数据

怎样处理噪声数据？

- 分箱方法:
  - 首先把数据排序，把排序后数据分到等深的箱中
  - 接着，用按箱中值平滑、按箱平均值平滑、按箱边界平滑等平滑技术平滑数据
- 聚类
  - 探测和删除孤立点
- 计算机和人工检查结合
  - 计算机先探测到可疑值，然后进行人工检查
- 回归
  - 通过让数据来适合一个回归函数来平滑数据

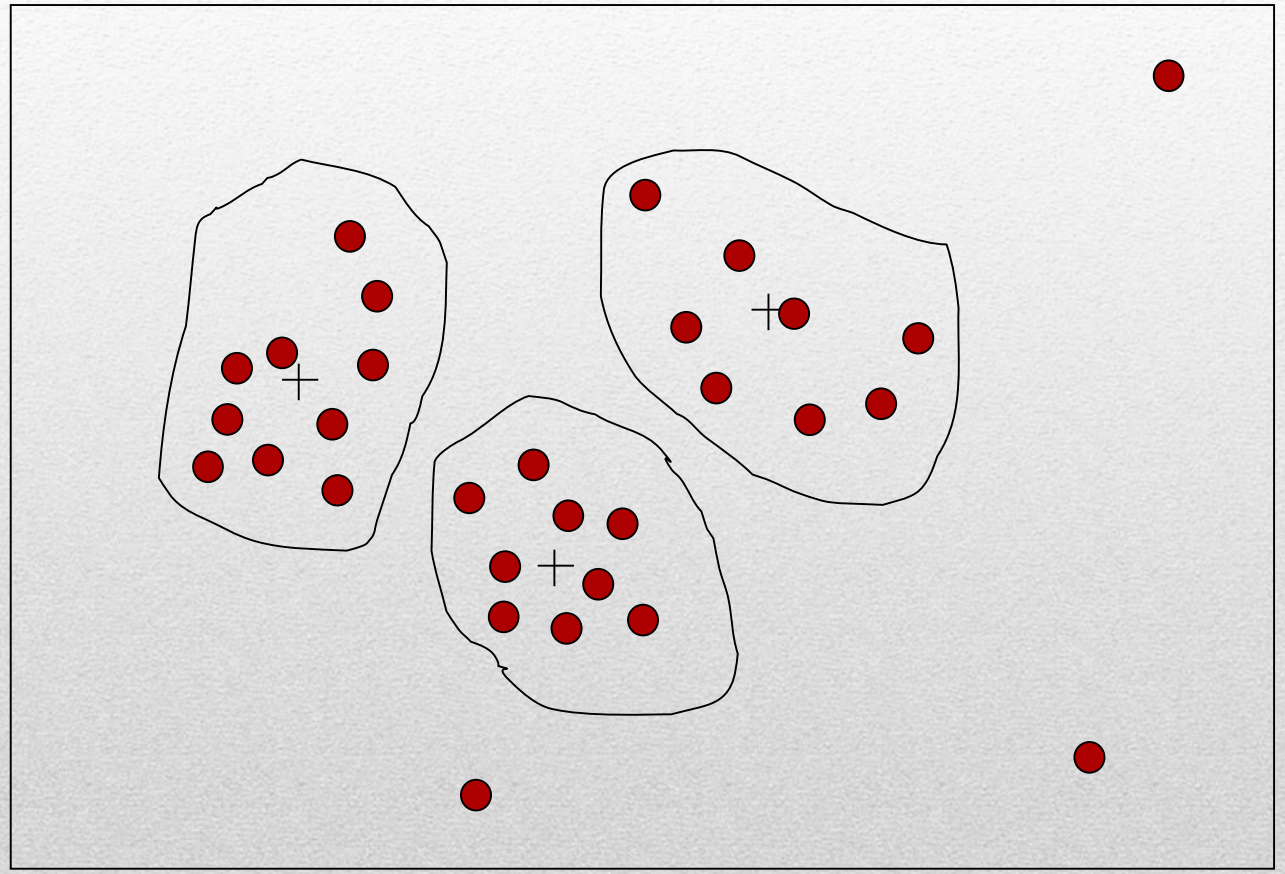
简单离散化方法：分箱

- 等宽（距离）分箱法:
  - 它把区间分成N个相等大小的子区间：统一栅格
  - 如果A和B分别是属性的最小最大值，那么间隔的宽度是: $W = (B-A)/N$.
  - 等宽分箱法是最直接的分箱方法
  - 孤立点可能在表示中占优势地位
  - 倾斜数据不能很好的被处理
- 等深分箱法:
  - 它把区域划分为N个间隔，每个间隔大约包含了等数量的样本
  - 好的数据缩放比例
  - 处理分类属性时可能具有欺骗性

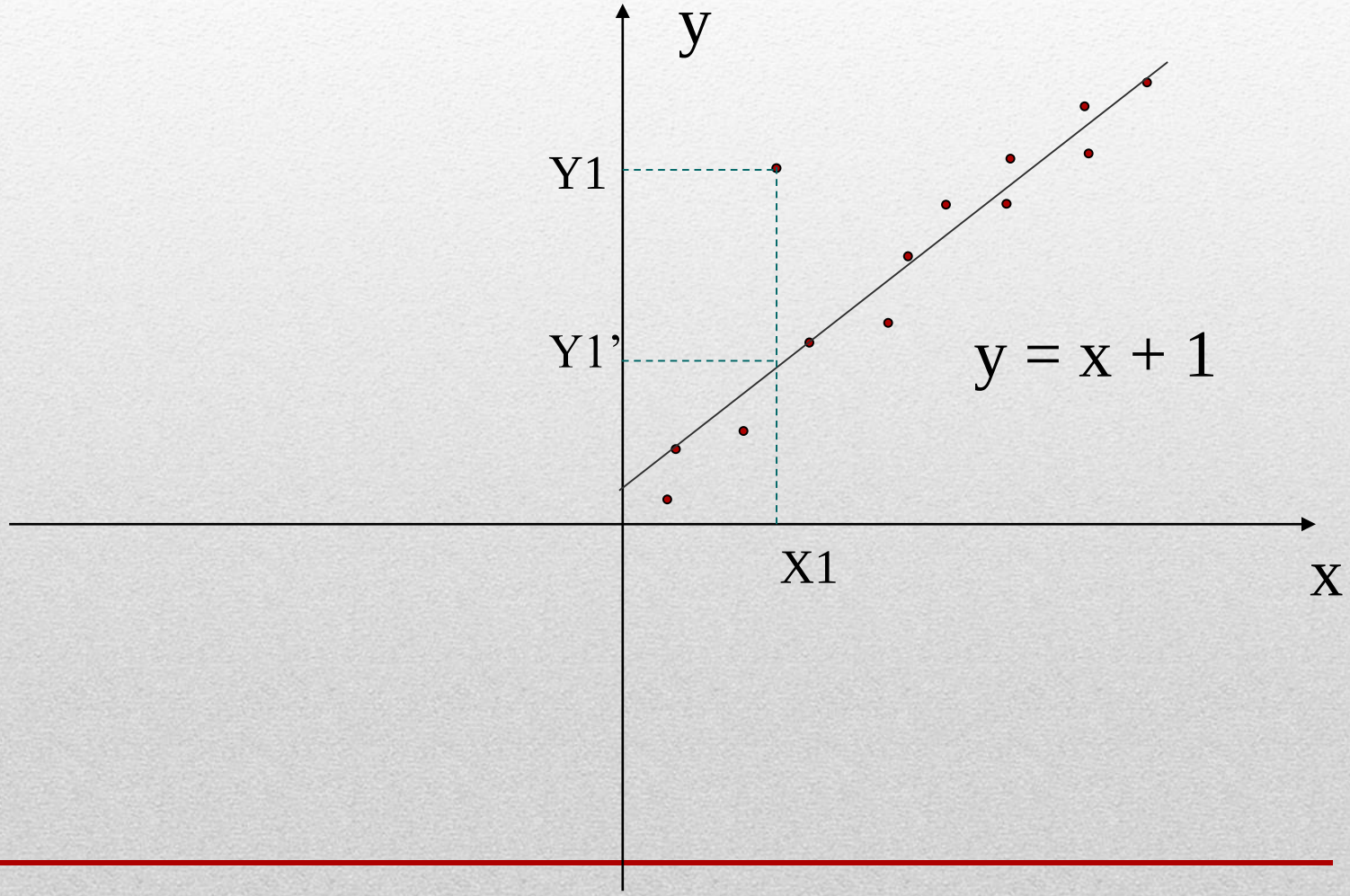
数据平滑的分箱方法

* **price 的排序后数据(美元): 4, 8, 9, 15, 21, 21, 24, 25, 26, 28, 29, 34**
* **划分为（等深的）箱:**
  - **箱 1: 4, 8, 9, 15**
  - **箱2: 21, 21, 24, 25**
  - **箱3: 26, 28, 29, 34**
- **用箱平均值平滑:**
  - **箱 1: 9, 9, 9, 9**
  - **箱 2: 23, 23, 23, 23**
  - **箱 3: 29, 29, 29, 29**
* **用箱边界值平滑:**
  - **Bin 1: 4, 4, 4, 15**
  - **Bin 2: 21, 21, 25, 25**
  - **Bin 3: 26, 26, 26, 34**

聚类分析



回　归



数据预处理

- 为什麽要预处理数据?
- 数据清理
- 数据集成和变换
- 数据归约

数据集成

- 数据集成:
  - 将多个数据源中的数据结合起来存放在一个一致的数据存储中
- 模式集成
  - 从不同的数据源中集成元数据
  - 实体识别问题：从多数据源中识别现实世界实体。例如:确定A数据库中的cust-id和B数据库中的cust-number指的是否为同一实体
- 数据值冲突的检测和处理
  - 对同一现实世界的实体，来自于不同数据源的属性值是不同的
  - 可能的原因：不同的表示，不同的尺度，如公制单位和英制单位

数据集成中的冗余数据的处理

- 冗余数据通常出现在多数据库的集成中
  - 同一个属性在不同的数据库中可能有不同的名字
  - 例如,属性年收入在另外一个表中可能用“derived”表示
- 冗余数据可以通过相关分析进行检测
- 对多数据源中的数据进行仔细的数据集成可以减少/避免冗余和矛盾并且能提高挖掘的速度和质量

数据变换

- 平滑：去掉数据中的噪声
- 聚集：汇总，数据立方体的构造
- 数据概化：概念层的上攀
- 规范化：将属性数据按比例缩放，使之落入一个小的特定区间
  - 最小一最大规范化
  - z-score 规范化
  - 小数定标规范化
- 属性构造
  - 由给定的属性构造新的属性

数据变换：规范化

- 最小－最大规范化

$$v' = \frac{v - min_A}{max_A - min_A}(new\_max_A - new\_min_A) + new\_min_A$$

- **z-score** 规范化

$$v' = \frac{v - mean_A}{stand\_dev_A}$$

- 小数定标规范化

$$v' = \frac{v}{10^j}$$

J是使max(|v’|)<1的最小整数

数据预处理

- 为什么要预处理数据?
- 数据清理
- 数据集成和变换
- 数据归约

数据归约策略

- 数据仓库可以存储数千兆字节的数据：在海量数据上进行复杂数据分析和数据挖掘需要很长时间
- 数据归约
  - 数据归约技术可以用来得到数据集的规约表示，它在规模上要小得多，但能产生同样（或几乎同样的）分析结果
- 数据归约策略
  - 数据立方体聚集
  - 维归约
  - 数值压缩
  - 离散化和概念分层产生

数据立方体聚集

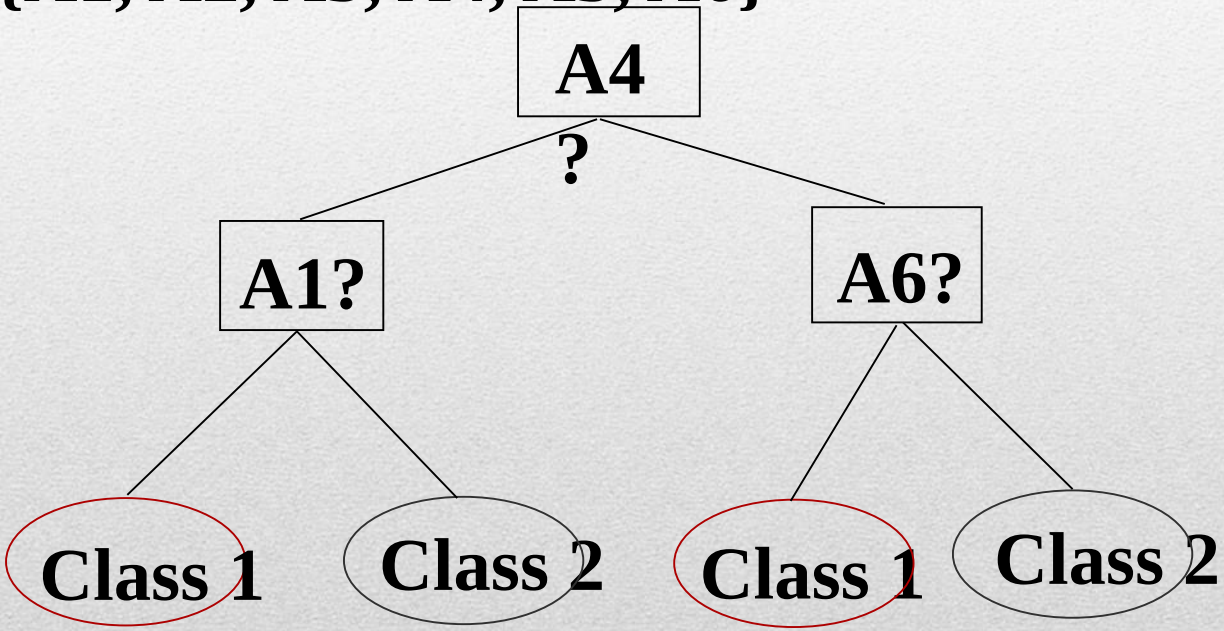
- 数据立方体的最低层(基本方体)
  - 对应于感兴趣实体的聚集数据
  - 例如:电话呼叫数据库中的顾客
- 数据立方体中聚集的多层次
  - 进一步减少了要处理数据的大小
- 涉及适当的层
  - 使用能够完成该任务的最小表示
- 有关聚集信息的查询,如果可能的话,应当使用数据立方体回答

维归约

- 属性选择（如：属性子集选择）：
  - 找出最小的属性集,使得数据类的概念分布尽可能接近使用所有属性的原分布
  - 减少了出现在发现模式上属性的数目,使得模式更易于理解
- 属性子集选择的启发式方法:
  - 逐步向前选择
  - 逐步向后删除
  - 逐步向前选择和逐步向后删除的结合
  - 判定树归纳

判定树归纳的例子

初始属性集:
{A1, A2, A3, A4, A5, A6}

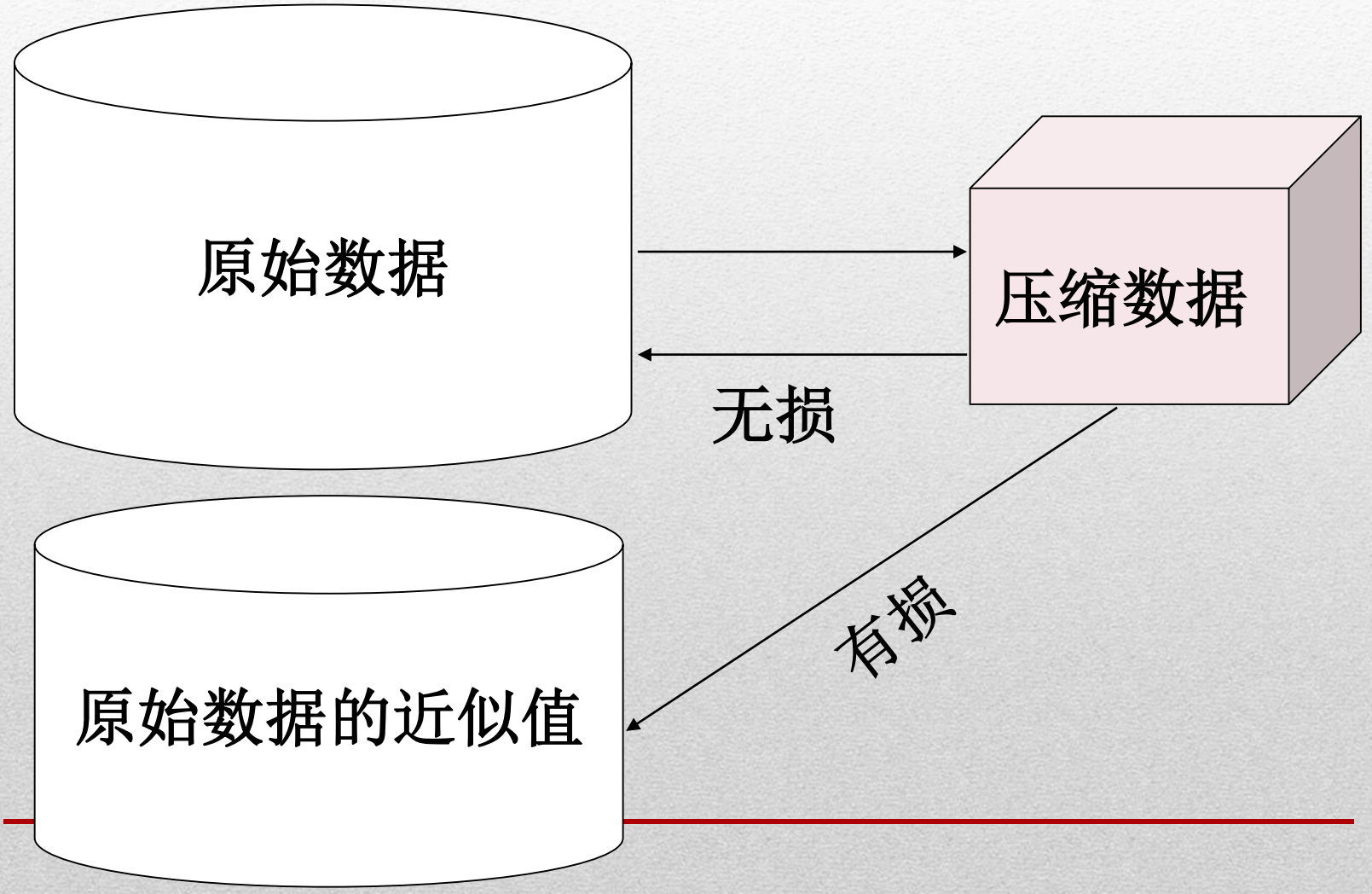


-------> 归约后的属性集: {A1, A4, A6}

数据压缩

- 串压缩
  - 已有广泛的理论和协调的算法
  - 典型的无损压缩
  - 但是只允许有限的数据操作
- 音频/图像压缩
  - 典型的有损压缩，逐步加细
  - 有时可以只重构信号的小片断，而无需重构整个信号
- 时间序列是非音频的
  - 典型的短，并且随时间变化的很慢

数据压缩



离　散　化

- 属性的三种类型:
  - 标称属性 —来自无序集中的值
  - 序数属性—来自有序集的值
  - 连续属性 —实数
- 离散化:
  - 把连续的属性值区间划分成多个区间
  - 一些分类算法只接受分类属性
  - 通过离散化压缩数据大小
  - 为进一步分析作准备

关联规则

- 关联规则挖掘
- 由事务数据库挖掘单维布尔关联规则

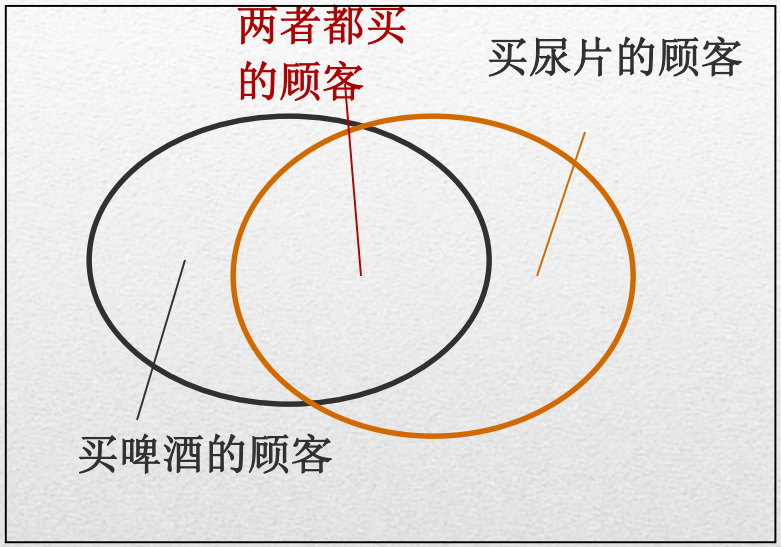
什么是关联挖掘?

- 关联规则挖掘:
  - 从事务数据库,关系数据库和其他信息数据库中找出项目集和对象集中的频繁模式,关连,相关联系
- 应用:
  - 购物篮数据分析,交叉营销等等
- 例子:
  - 规则形式: "Body → Head [支持度,置信度]".
  - buys(x, "diapers") → buys(x, "beers") [0.5%, 60%]
  - major(x, "CS") ^ takes(x, "DB") → grade(x, "A") [1%, 75%]

关联规则:基本概念

- 假设: (1) 事务数据库, (2) 每一个事务是一个项目集列表(顾客一次购买的商品集)
- 找出:所有的使目前的项集与另一个项集相关联的规则
  - 例如, *98% 的人们买了轮胎和汽车附件也会需要汽车的维修服务*
- 应用
  - ** ⇒ 维修协议* (商店应该怎样做才能提升维修协议的销售)
  - *家电 ⇒ ** (商店应该增加其它那些产品的存储量?)

规则度量：支持度和置信度



- 找出所有具有最小支持度和置信度的规则$X \& Y \Rightarrow Z$
  - 支持度s, 一个事务中包含$X \cup Y \cup Z$的可能性
  - 信任度c, *一个事务包含$X \cup Y$,同时包含Z的百分比*

TID	项ID列表
2000	A,B,C
1000	A,C
4000	A,D
5000	B,E,F

设最小支持度为 ***50%,*** *最小置信度为* ***50%,*** *我们有*

- $A \Rightarrow C$ **(50%, 66.6%)**
- $C \Rightarrow A$ **(50%, 100%)**

关联规则挖掘：一个路线图

- 布尔关联和量化关联 (根据规则中所处理的值类型)
  - **buys(x, “SQLServer”) ^ buys(x, “DMBook”) → buys(x, “DBMiner”) [0.2%, 60%]**
  - **age(x, “30..39”) ^ income(x, “42..48K”) → buys(x, “PC”) [1%, 75%]**
- 单层和多层关联
  - 什么牌子的啤酒和什么牌子的尿布相关联？
- 各种扩充
  - 相关性和因果关系分析
    - 关联并不一定必须意味着相关性和因果性
  - 强制的约束
    - **E.g., 小销售 (sum < 100) 引起了大买卖 (sum > 1,000)?**

挖掘大型数据库中的关联规则

- 关联规则挖掘
- 由事务数据库挖掘单维布尔关联规则
- 由事务数据库挖掘多层关联规则
- 由事务数据库和数据仓库挖掘多维关联规则
- 有关联挖掘到相关分析
- 基于约束的关联规则
- 小结

挖掘关联规则的一个例子

TID	项ID列表
2000	A,B,C
1000	A,C
4000	A,D
5000	B,E,F

最小支持度 50%
最小置信度 50%

频繁项集	支持度
{A}	75%
{B}	50%
{C}	50%
{A,C}	50%

对于规则 $A \Rightarrow C$:

支持度 =支持度($\{A \cup C\}$) = 50%

置信度= 支持度($\{A \cup C\}$)/支持度($\{A\}$) = 66.6%

Apriori算法 原理:

任何一个频繁项集的子集必定是频繁项集

挖掘频繁项集:关键步

- 找出频繁项集:具有最小支持度的项目集
  - 一个频繁项集的子集必定是频繁项集
    - 如:如果{AB}是频繁项集,那么{A},{B}都是频繁项集.
  - *使用从1到k的候选集交互式的产生频繁项集*
- 由频繁项集产生关联规则

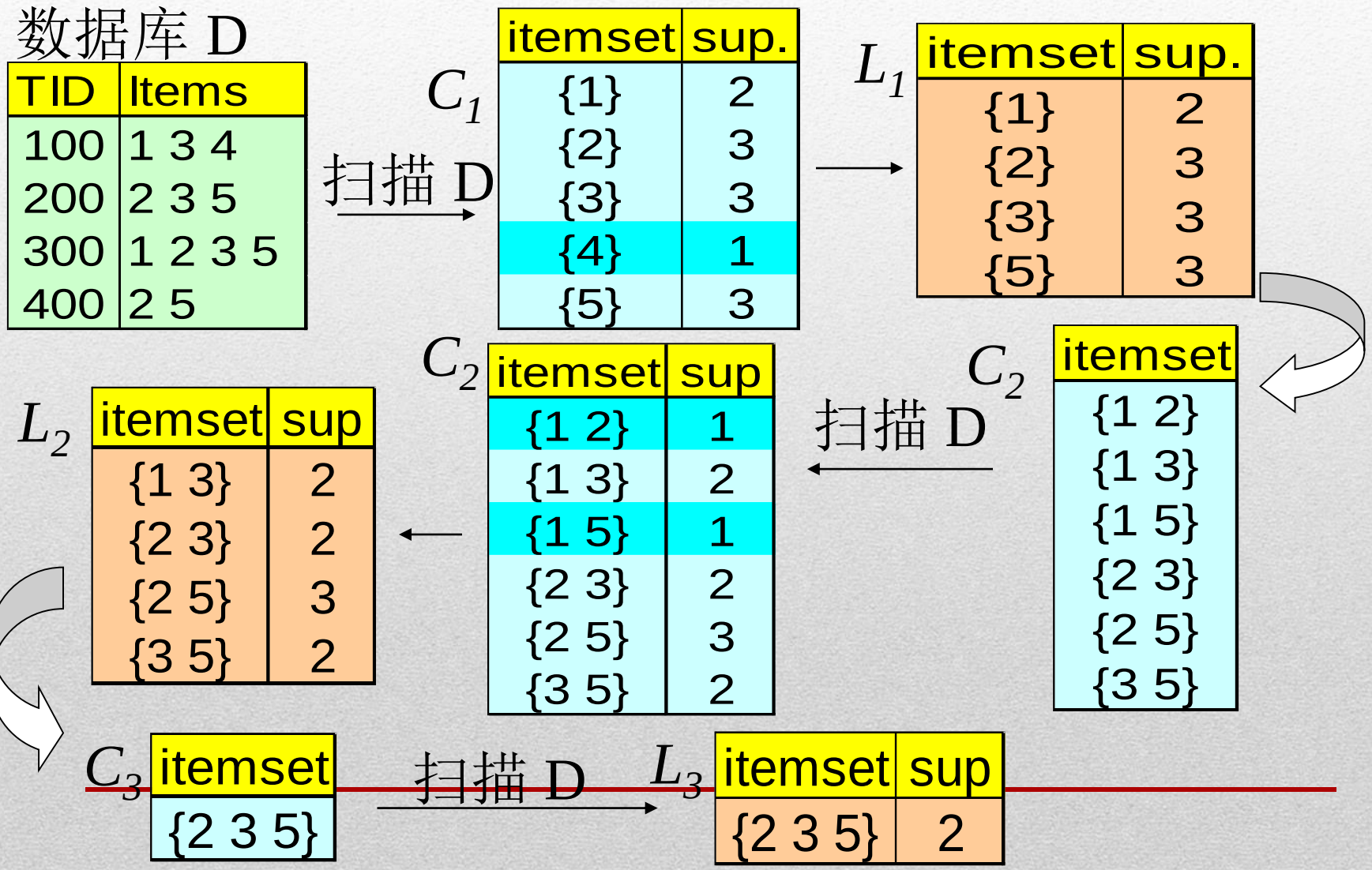
Apriori 算法

- **连接步**: 通过连接产生C_k
- **剪枝步**: 如果一个候选k-项集的(k-1)-子集不在(k-1)-的频繁项集中,则该候选集也不可能是频繁的,从而由C_k删除
- **Pseudo-code**:

  C_k: 大小为k的候选集
  L_k : 大小为k的频繁项集

  L_1 = {频繁项集};
  for (k = 1; L_k !=∅; k++) do begin
  C_{k+1} = 从L_k 中产生的*候选集* ;
  for each transaction t ∈D do
  对于包含在t中的属于C_{k+1} 的所有候选集的计数加一
  L_{k+1} = C_{k+1} 中具有最小支持度的候选集
  end
  return $\cup_k L_k$;

Apriori 算法 — 例子



怎样产生候选集?

- 假设L_{k-1} 中的项是按顺序列出的
- 第一步: 自我连接 L_{k-1}

  Insert into C_k

  select $p.item_1, p.item_2, \ldots, p.item_{k-1}, q.item_{k-1}$

  from $L_{k-1}\ p, L_{k-1}\ q$

  where $p.item_1=q.item_1, \ldots, p.item_{k-2}=q.item_{k-2}, p.item_{k-1} < q.item_{k-1}$
- 第二步: 剪枝步

  对于C_k*中的所有项集c* **do**

  对于c 的所有(k-1)子集s **do**

  if *(s 不在* L_{k-1} *中)* **then** 从C_k*中*删除c

怎样计算候选集的支持度?

- 为什麽计算候选集的支持度是一个问题?
  - 候选集的总数目可能非常大
  - 一个事务可以包含许多的候选集
- 方法:
  - *候选集存储在哈希树中*
  - 包含项集和计数列表
  - 内部结点 包含了一个哈希表
  - *子集函数*: 找出包含在一个事务中的所有候选集

产生候选集的例子

- L_3={*abc, abd, acd, ace, bcd*}
- 自我连接: L_3*L_3
  - 由 *abc* 和*abd产生abcd*
  - *由 acd* 和 *ace产生acde*
- 剪枝:
  - *删除acde* 因为 *ade* 不在 L_3
- C_4={*abcd*}

提高 Apriori 的有效性的方法

- 散列项集计数: 如果一个k-项集对应的哈希桶计数低于支持度阈值,则它不可能是频繁的
- 事务压缩: 不包含任何k-项集的事务在以后的扫描中是无用的
- 划分: 任何项集如果在DB中是潜在频繁的,那么它至少在DB中的一个划分中是频繁的
- 选样: 在给定数据的一个子集上进行挖掘,低支持阈值+好的策略可以得到完整的频繁项集
- 动态项集计数: 如果一个项集的所有子集已被确定为频繁的,则添加大作为新的候选

Apriori算法足够的快吗?—瓶颈问题的产生

- **Apriori 算法的核心:**
  - 使用频繁(k-1)-项集产生候选k-项集
  - 使用数据库扫描和模式匹配为候选项集收集计数
- ***Apriori算法中的瓶颈问题*: 候选集的产生**
  - 大量的候选集:
    - 10^4 个频繁1-项集将生成 10^7 个候选 2-项集
    - 为了发现一个大小为100的频繁模式, 如, $\{a_1, a_2, \ldots, a_{100}\}$, 需要产生 $2^{100} \approx 10^{30}$ 个候选集
  - 数据库的多遍扫描:
- 需要(n+1)遍扫描,n是最长项集的长度

不产生候选挖掘频繁项集

- 把一个大型的数据库压缩到一棵频繁模式树(FP-树)
  - 高浓缩,但对频繁项集挖掘是完整的
  - 避免了高花销的数据库扫描
- 产生了一个高效的,基于FP-树的频繁模式挖掘方法
  - 分治策略:把挖掘任务分解成小的任务
  - 避免候选集的产生:只检测子数据库

从事务数据库中构造一个FP–树

TID	*Items bought*	*(排序的) 频繁项集*
100	{*f, a, c, d, g, i, m, p*}	{*f, c, a, m, p*}
200	{*a, b, c, f, l, m, o*}	{*f, c, a, b, m*}
300	{*b, f, h, j, o*}	{*f, b*}
400	{*b, c, k, s, p*}	{*c, b, p*}
500	{*a, f, c, e, l, p, m, n*}	{*f, c, a, m, p*}

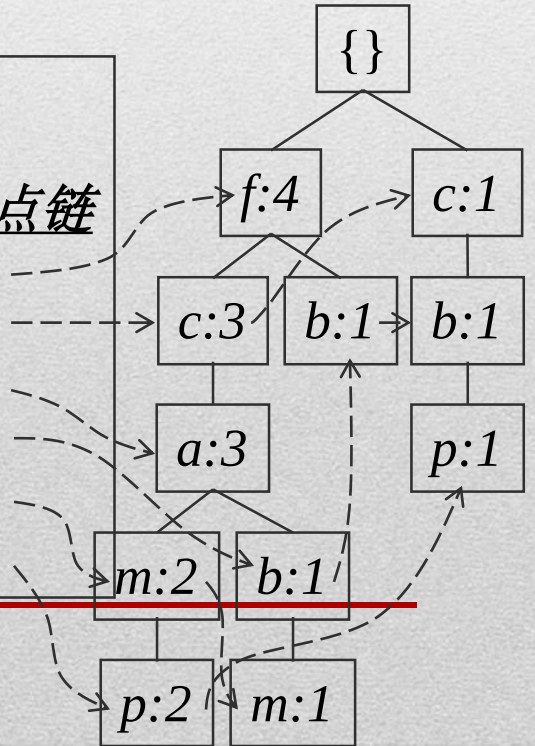
最小支持度 = 0.5

步骤:

1. 扫描DB一次,找出频繁1-项集(单项模式)
2. 频繁项的集合按支持度计数的递减顺序排序
3. 再一次扫描DB,构造FP-树

索引表

项	*支持度计数*	*节点链*
f	*4*	
c	*4*	
a	*3*	
b	*3*	
m	*3*	
p	*3*	



FP-树结构的优点

- 完整性:
  - 不会破坏任何交易的长模式
  - 为频繁模式挖掘保存了完整的信息
- 简洁性
  - 减少了不相关的信息—非频繁项集被删掉
  - 频繁项集按支持度递减顺序排列:越是频繁的项集越有可能被共享
  - 不会比原数据库大(如果不算节点链和计数)
  - 例如:对相联的4个数据库，压缩率将超过100

使用FP–树挖掘频繁模式

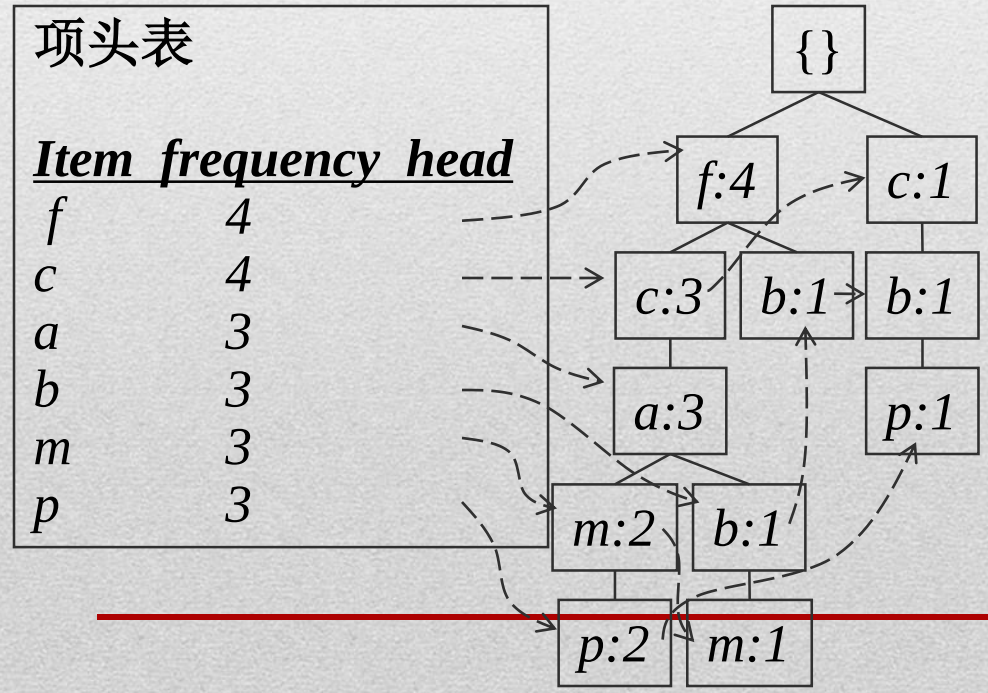
- 基本思想(分治策略）：
  - 使用FP-树循环的产生频繁模式路径
- 方法
  - 对于每一个项,先构造它的条件模式基,然后构造它的条件FP-树
  - 在每一个新创建的条件FP-树上重复此过程
  - 直到结果FP树为空,或它只包含一条路径(单路径将产生所有的它的子路径的结合,每一条子路径都是一个频繁模式)

挖掘FP–树的主要步骤

1) 为FP-树中的每一个节点构造条件模式基
2) 为每一个条件模式基条件FP-树
3) 循环的挖掘条件FP-树，生成至今为止获得的频繁模式
   - 如果条件 FP-树 只包含单条路径，简单的列举所有的模式

第一步：从FP-树到条件模式基

- 从FP-树的频繁项头表开始
- 跟随每一个频繁项的链遍历FP-树
- 聚集项的所有变换的前缀路径来形成条件模式基



条件模式基

item	*条件模式基*
c	***f:3***
a	***fc:3***
b	***fca:1, f:1, c:1***
m	***fca:2, fcab:1***
p	***fcam:2, cb:1***

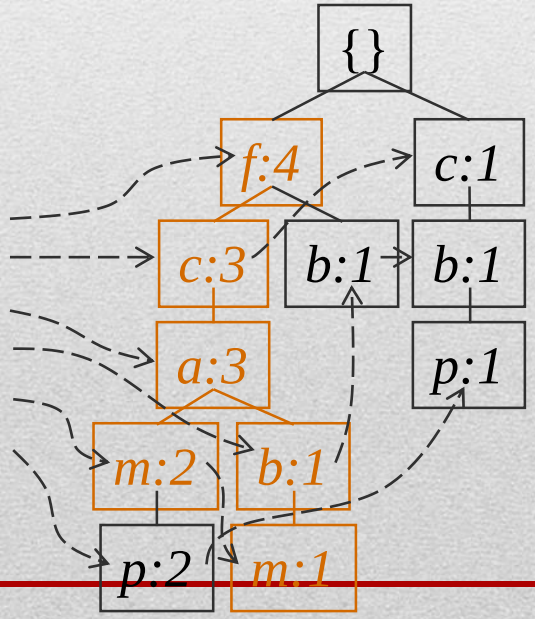
用于条件模式基构造的FP–树的属性

- 节点链属性
  - 对任意的频繁项 a_i, 包含 a_i 的所有可能的频繁模式可以 通过追随a_i的节点链获得, 这个节点链从FP-树项头的a_i头开始
- 前缀路径属性
  - a_i.为路径P, 中的节点a_i 计算频繁模式，只需聚集P中a_i 的前缀子路径，其频繁数与节点 a_i.的相同

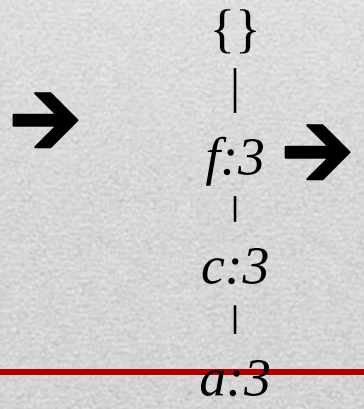
第二步:构造一个条件FP-树

- 对任意的模式基
  - 总计基中的每一个项数目
  - 为模式基中的频繁项构造FP-树

项头表 *频繁项头*	
f	*4*
c	*4*
a	*3*
b	*3*
m	*3*
p	*3*



***m-条件模式* base:**
fca:2, fcab:1



***m-条件* FP-树**

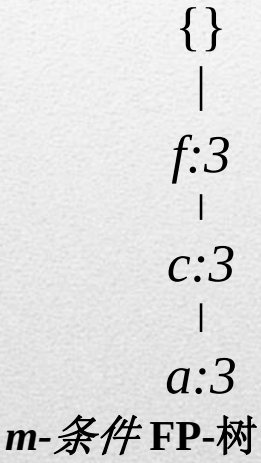
有关 ***m****的所有频繁模式*

m,
fm, cm, am,
fcm, fam, cam,
fcam

通过构造条件模式基挖掘频繁模式

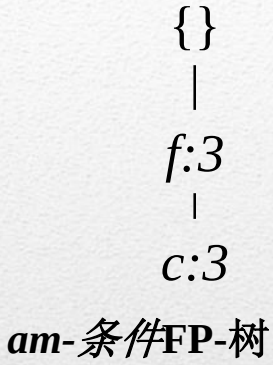
Item	条件模式基	条件 FP-树
p	{(fcam:2), (cb:1)}	{(c:3)}|p
m	{(fca:2), (fcab:1)}	{(f:3, c:3, a:3)}|m
b	{(fca:1), (f:1), (c:1)}	Empty
a	{(fc:3)}	{(f:3, c:3)}|a
c	{(f:3)}	{(f:3)}|c
f	Empty	Empty

第三步:循环挖掘条件FP–树



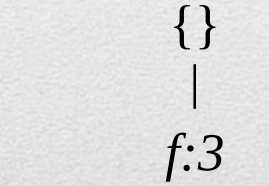
m-条件 **FP-树**

“am”的条件模式基: (fc:3)



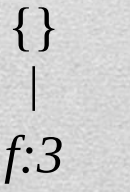
***am-条件*FP-树**

“cm”的条件模式基: (f:3)



***cm-条件* FP-树**

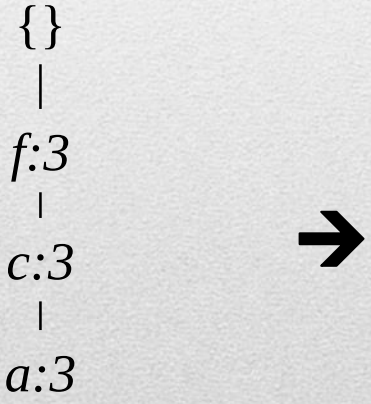
“cam”的条件模式基: (f:3)



***cam-条件*FP-树**

单一 FP–树路径的产生

- 假设一个 **FP-树 T** 有一个单一的路径 **P**
- **T**的完全频繁模式集可以通过列举**P**的子路径的所有组合来产生



***m-条件*FP-树**

→

有关 ***m****的所有频繁模式*

m,

fm, cm, am,

fcm, fam, cam,

fcam

频繁模式增长原理

- 频繁增长属性
  - 假设 α 是DB中的一个频繁项集, B是 α ‘的条件模式基, 且β 是 B中的一个项集. 那么 α ∪ β 是DB中的一个频繁项集 当且仅当 β 在B中是频繁的.
- “*abcdef* ” 是一个频繁模式, 当且仅当
  - “*abcde* ” 是一个频繁模式, 并且
  - “*f* ” 在包含 “*abcde* ”的交易集中是频繁的

为什么 频繁模式增长 是快速的?

- 我们的性能研究表明
  - **FP-树 比 Apriori算法快一个数量级, 也比树-投影算法快**
- 原因
  - 没有候选集的产生，没有候选测试
  - 使用简洁的数据结构
  - 除去了重复的数据库扫描
  - 基本操作是计数和**FP-树构造**