

# 数据库系统概念-下

## 数据库系统概念-下

### 第五章 数据库设计和E-R模型

#### 5.1 实体-联系模型

#### 5.2 约束

#### 5.3 实体-联系图

#### 5.4 实体-联系设计问题

#### 5.5 扩展的E-R特性

#### 5.4 转换为关系模式(Reduction to Relation Schema)

### 第六章 关系数据库设计

#### 6.1 Features of Good Relational Design (好的关系设计的特点)

#### 6.2 Atomic Domains and First Normal Form (原子域和第一范式)

#### 6.3 Decomposition Using Functional Dependencies (使用函数依赖进行分解)

#### 6.4 函数依赖理论

#### 6.5 分解算法

#### 6.6 使用多值依赖的分解

#### 6.7 更多的范式

#### 6.8 数据库设计过程

#### 8.9 时态数据库建模 (Modeling Temporal Data)

### 第七章 事务

## 第五章 数据库设计和E-R模型

实体-联系(Entity-Relationship, E-R)数据模型的提出旨在方便数据库的设计, 它是通过允许定义代表数据库全局逻辑结构的企业模式实现的, 其采用的三个基本概念为: 实体集、联系集和属性。

### 5.1 实体-联系模型

- 实体集

- Entity: An **entity** is an object that exists and is distinguishable from other objects.
- Entity set: An **entity set** is a set of entities of the same type that share the same properties.
- An entity is represented by a set of **attributes**, that is descriptive properties possessed by all members of an entity set.
  - Each entity has a **value** for each of its attributes.
    - An attribute has a **null** value when an attribute does not have value for it.
    - **Domain** – the set of permitted values for each attribute.
- 在建模过程中, 通常抽象的使用术语实体集, 而不是指某个个别实体的特别集合; 而使用术语实体集的**外延**来指属于实体集的实体的实际集合

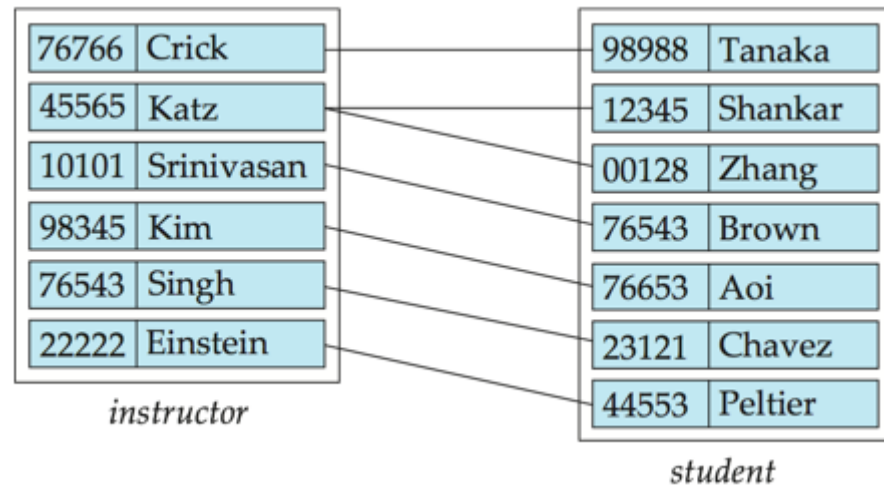
- 联系集

- 联系: 指多个实体间的相互关联;
- 联系集: 相同类型联系的集合;
  - 如果 $E_1, E_2, \dots, E_n$ 为实体集, 那么联系R是:

$$\{(e_1, e_2, \dots, e_n) | e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

的一个子集, 而 $(e_1, e_2, \dots, e_n)$ 是一个联系.

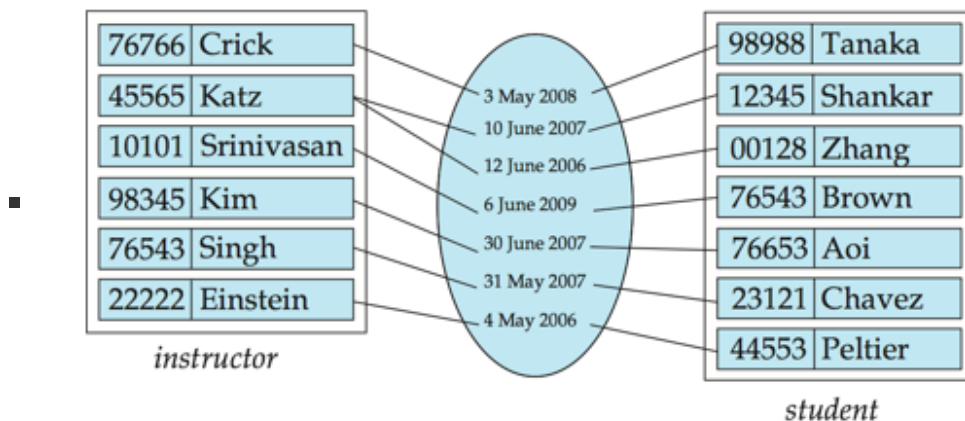
- 二元联系集
- 参与联系集的实体集的数目称为联系集的**度(degree)**
- Example: advisor



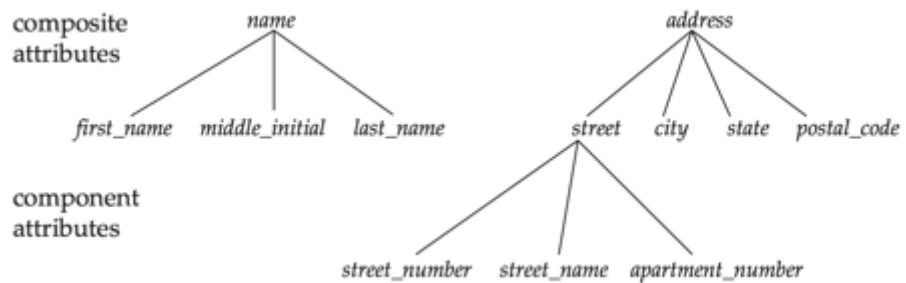
- 实体集之间的关联称为参与 participate：也就是说， $E_1, E_2, \dots, E_n$  参与联系集R;
- 联系实例 Relationship instance:表示在所建模的显示世界企业中命名实体间的一个管理;
- 实体在联系中扮演的功能称为**角色(role)**:

实体在联系中扮演的功能称为实体的**角色 (role)**。由于参与一个联系集的实体集通常是互异的，因此角色是隐含的并且一般并不指定。但是，当联系的含义需要解释时角色是很有用的。当参与联系集的实体集并非互异的时候就是这种情况；也就是说，同样的实体集以不同的角色参与一个联系集多于一次。在这类联系集中，即有时称作**自环的 (recursive)**联系集中，有必要用显式的角色名来指明实体是如何参与联系实例的。例如，考虑记录大学开设的所有课程的信息的实体集 *course*。我们用 *course* 实体的有序对来建模联系集 *prereq*，以描述一门课程 (*C2*) 是另一门课程 (*C1*) 的先修课。每对课程中的第一门课程具有课程 *C1* 的角色，而第二门课程具有先修课 *C2* 的角色。按照这种方式，所有的 *prereq* 联系通过 (*C1*, *C2*) 对来表示，排除了 (*C2*, *C1*) 对。

- 联系可以具有**描述性属性(descriptive attribute)**



- 属性:
  - 每个属性都有一个可取值的集合，称为该属性的域(domain)，或者值集(value set)
  - 类型:
    - 简单和复合属性(Simple and composite attributes): 能否划分为更小的部分



- 单值和多值属性(**Single-valued** and **multivalued** attributes):
  - Example: {phone\_numbers}
- 派生属性(**Derived** attributes): 可以从别的相关属性或实体派生出来
  - 派生属性的值不存储, 而是在需要时计算出来
  - Example: age can be computed by date\_of\_birth(基属性)

## 5.2 约束

- 映射基数(mapping cardinality)约束:
  - 或称基数比率, 表示一个实体通过一个联系集能过关联的实体的个数;
  - For a binary relationship set the mapping cardinality must be one of the following types:
    - One to one
    - One to many
    - Many to one
    - Many to many

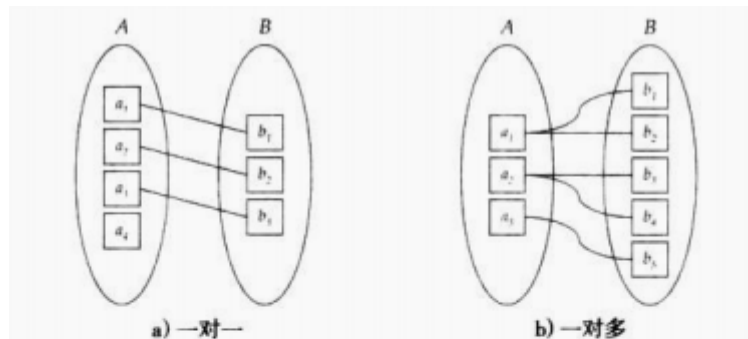


图 7-5 映射基数

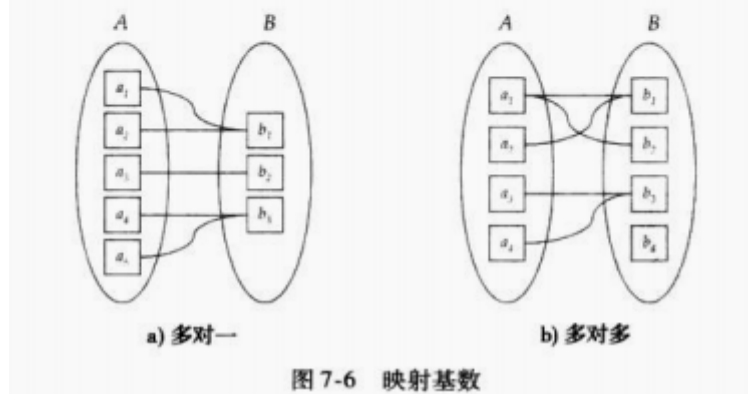
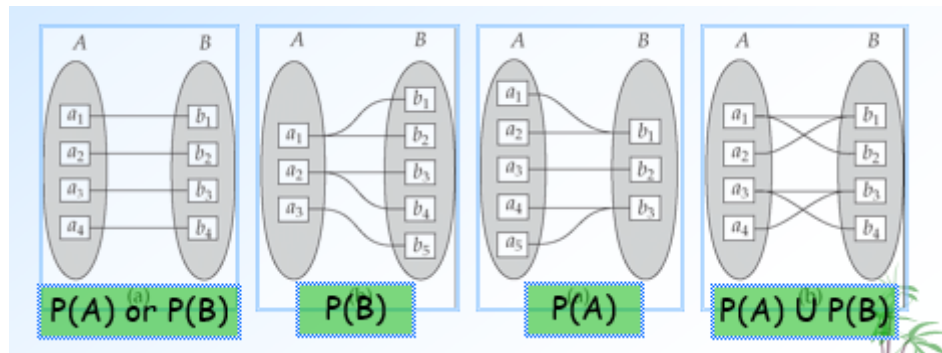


图 7-6 映射基数

- Note: Some elements in A and B may not be mapped to any elements in the other set
- 参与约束(Participation Constraints):
  - 如果实体集E的每个实体都参与到联系集R的至少一个联系中, 实体集E在联系集R中的参与称为**全部(total)**的;

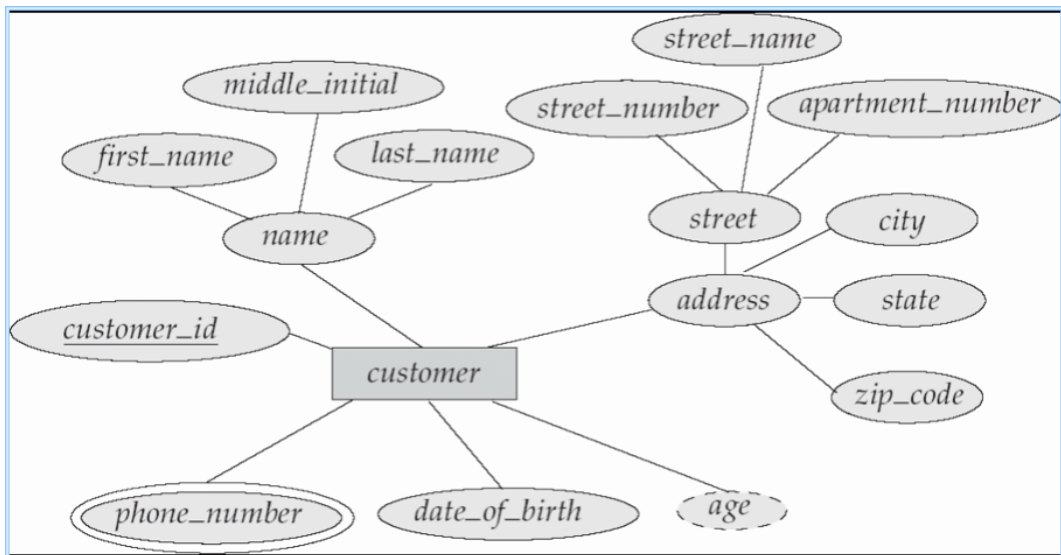
- 如果实体集E的只有部分实体都参与到联系集R的联系中，实体集E在联系集R中的参与称为**部分(partial)**的
- 码——Keys for Relationship Sets
  - 设R是一个涉及实体集 $E_1, E_2, \dots, E_n$ 的联系集，联系集主码的构成依赖于同联系集R相关联的属性集合；
  - 属性集合：
 
$$Primary - key(E_1) \cup Primary - key(E_2) \cup \dots \cup Primary - key(E_n) \cup \{a_1, a_2, \dots, a_n\}$$
 描述了集合R中的一个联系，
   
其中 $Primary - key(E_1) \cup Primary - key(E_2) \cup \dots \cup Primary - key(E_n)$ 构成联系集的一个超码；
  - 联系集的主码结构依赖于联系集的映射基数：



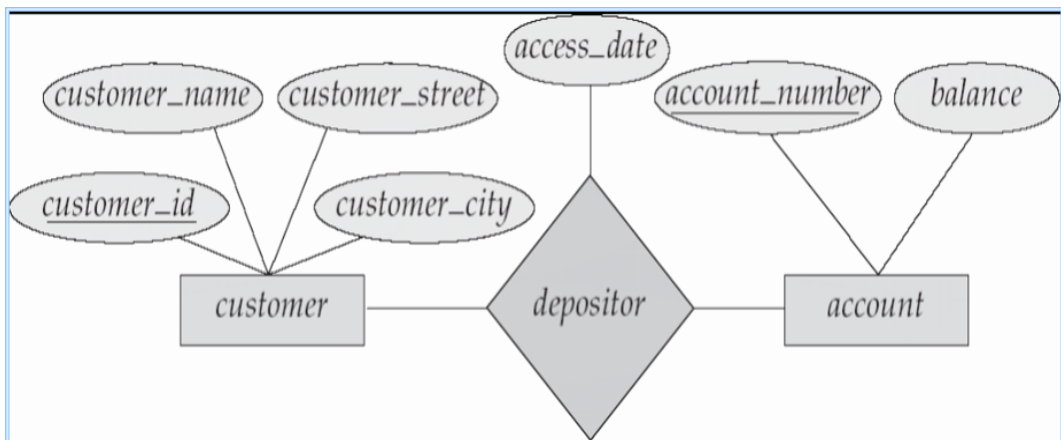
- Need to consider semantics of relationship set in selecting the *primary key* in case of more than one candidate key.

### 5.3 实体-联系图

- 基本结构：
  - Rectangles** represent entity sets：矩形代表实体集；
  - Diamonds** represent relationship sets：菱形代表联系集；
  - Lines** link attributes to entity sets and entity sets to relationship sets：
    - 线段把属性连接到实体集和把实体集连接到联系集；
  - Ellipses** represent attributes: 椭圆代表属性
    - Double ellipses represent multivalued attributes：双椭圆代表多值属性
    - Dashed ellipses denote derived attributes：虚线椭圆代表派生属性
  - Underline** indicates primary key attributes：下划线代表主码
  - Example：
    - E-R Diagram with composite, Multivalued, and Derived Attributes



Relationship sets with Attributes:

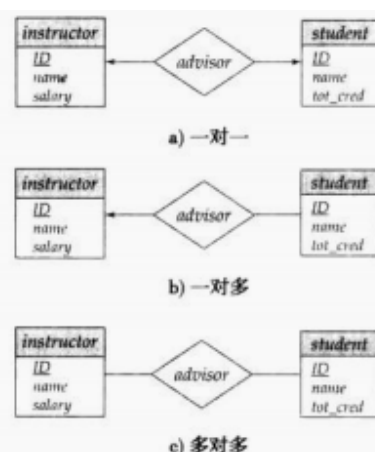


- 映射基数

- We express cardinality constraints by drawing either a directed line ( $\rightarrow$ )(signifying “one” )

or an undirected line ( $\text{---}$ )(signifying “many”) between the relationship set and the entity set.

- 一对一：我们从联系集 *advisor* 向实体集 *instructor* 和 *student* 各画一个箭头 (见图 7-9a)。这表示一名教师可以指导至多一名学生，并且一名学生可以有至多一位导师。
- 一对多：我们从联系集 *advisor* 画一个箭头到实体集 *instructor*，以及一条线段到实体集 *student* (见图 7-9b)。这表示一名教师可以指导多名学生，但一名学生可以有至多一位导师。
- 多对一：我们从联系集 *advisor* 画一条线段到实体集 *instructor*，以及一个箭头到实体集 *student*。这表示一名教师可以指导至多一名学生，但一名学生可以有多位导师。
- 多对多：我们从联系集 *advisor* 向实体集 *instructor* 和 *student* 各画一条线段 (见图 7-9c)。这表示一名教师可以指导多名学生，并且一名学生可以有多位导师。



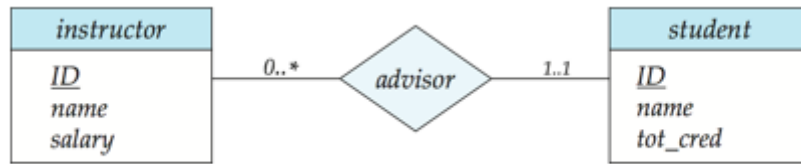
- Total and Partial Participation



- Alternative Notation for Cardinality Limits:

Cardinality limits can also express participation constraints

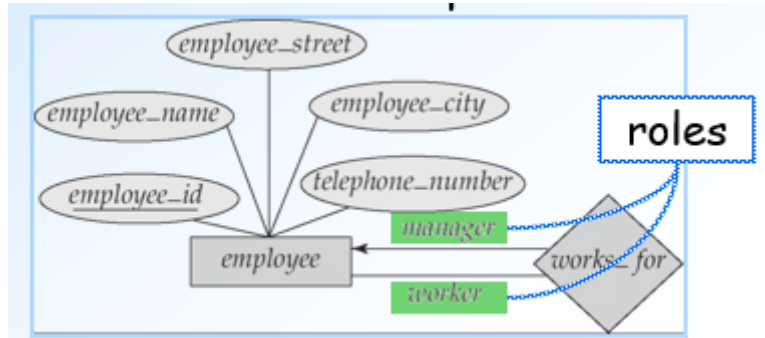
- 实体集与二元联系集之间的一条边可以有一个最大的最小的映射基数，用l...h的形式表示，其中l表示最小映射基数，h表示最大映射基数(\*代表无限制)



- 1...1 表示一个学生必须有且仅有一个导师(全部的);
- 0...\* 表示一个教室可以有零个或多个学生;
- 故: **advisor**联系是从instructor到student的一对多联系
- Another notation:
  - 一条从student到advisor的双线, 以及一个从advisor到instructor的箭头

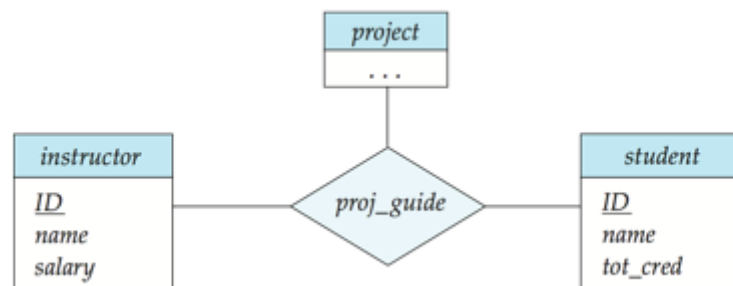
- 角色!!!!!!:

- In E-R diagram, we use roles to specify how entities interact via relationship set



- 非二元联系集:

- 



- We allow at most one arrow out of a ternary (or greater degree) relationship to indicate a cardinality constraint.
- If there is more than one arrow, there are two ways of defining the meaning:

在一个联系集外我们至多允许一个箭头, 因为在一个非二元的联系集外包含两个或更多箭头的 E-R 图可以用两种方法解释。假设实体集  $A_1, A_2, \dots, A_n$  之间有联系集  $R$ , 并且只有指向实体集  $A_{i+1}, A_{i+2}, \dots, A_n$  的边是箭头。那么, 两种可能的解释为:

- 来自  $A_1, A_2, \dots, A_i$  的实体的一个特定组合可以和至多一个来自  $A_{i+1}, A_{i+2}, \dots, A_n$  的实体组合相关联。因而联系  $R$  的主码可以用  $A_1, A_2, \dots, A_i$  的主码的并集来构造。
- 对每个实体集  $A_k, i < k \leq n$ , 来自其他实体集的实体的每个组合可以和来自  $A_k$  的至多一个实体相关联。于是对于  $i < k \leq n$ , 每个集合  $\{A_1, A_2, \dots, A_{k-1}, A_{k+1}, \dots, A_n\}$  构成一个候选码。

- 弱实体集(Weak entity sets)\*\*\*

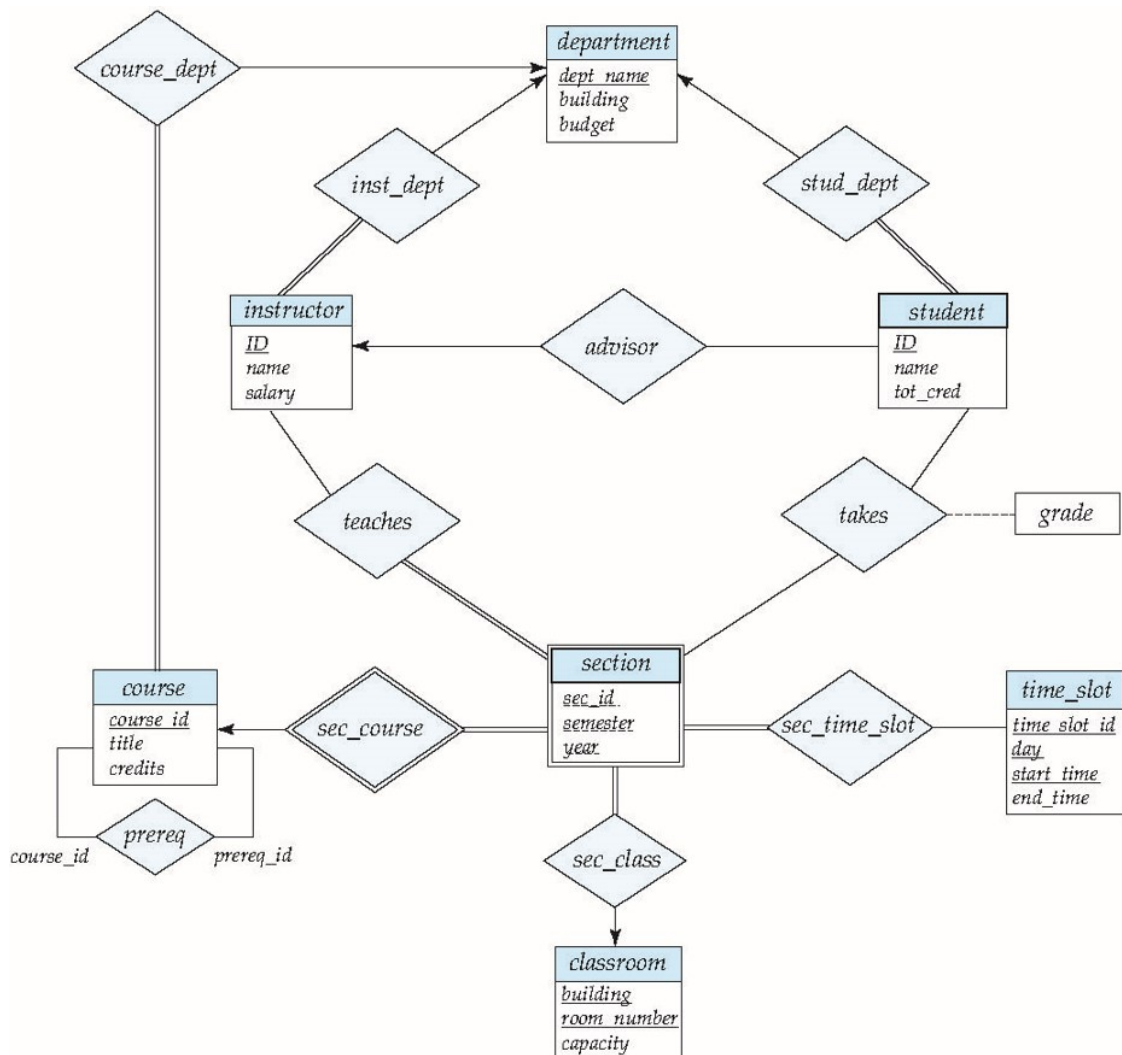
- 定义: 没有足够的属性以形成主码的实体集称作弱实体集(反之为强实体集)
  - 弱实体集必须与另一个称作**标识实体集(identifying entity set)**或**属主实体集(owner entity set)**的实体集关联才有意义(即, 弱实体集存在依赖于标识实体集)



- **标识性联系(identifying relationship)**: 弱实体集与其标识实体集相联的联系;
  - 标识性联系是从弱实体集到标识实体集多对一的, 并且弱实体集在联系中的参与是全部的;
- **分辨符(Discriminator)**: 弱实体集的分辨符是使能够区分依赖于特定强实体集的弱实体集中的实体的属性集合; 弱实体集的分辨符也称为该实体集的部分码;
- **弱实体集的主码**由标识实体集的主码加上该弱实体集的分辨符构成;
- 表示:
  - 矩形表示;
  - 弱实体集的分辨符以虚下划线标明, 而不是实现;
  - 关联弱实体集和标识性实体集的联系集以双菱形表示;



#### • E-R Diagram for a University Enterprise

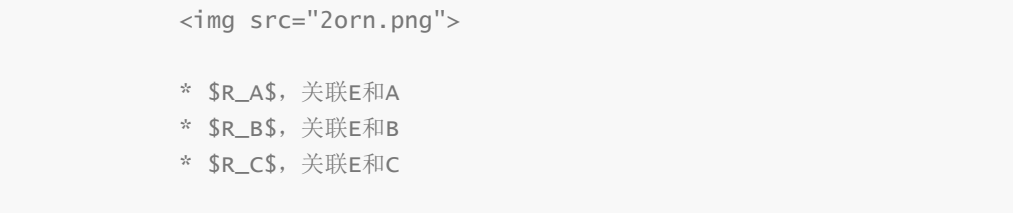


#### 5.4 实体-联系设计问题

- 用实体集还是属性:
  - Choice mainly depends on the structure of the enterprise being modeled, and on the semantics associated with the attribute in question.
- 用实体集还是联系集:

- 在决定用联系集还是实体集时可采用的一个原则是，当描述发生在实体间的行为是采用联系集
- 二元还是n元联系集：
  - Although it is possible to replace any non\_binary ( $n$ -ary, for  $n > 2$ ) relationship set by a number of distinct binary relationship sets, a  $n$ -ary relationship set shows more clearly that several entities participate in a single relationship.
  - Some relationships that appear to be non-binary may be better represented using binary relationships
  - Converting Non-Binary Relationships to Binary Form

事实上，一个非二元( $n$ 元,  $n > 2$ )的联系集总可以用一组不同的二元联系集来替代。简单起见，考虑一个抽象的三元( $n=3$ )联系集R，它将实体集A、B和C联系起来。用实体集E替代联系集R，并创建三个联系集：



- \*  $R_A$ ，关联E和A
- \*  $R_B$ ，关联E和B
- \*  $R_C$ ，关联E和C

如果联系集E有属性，那么将这些属性赋给实体集E；进一步，为E创建一个特殊的标识属性(因为它必须能够通过其属性值来区别实体集中的各个实体)。

针对联系集R中的每个联系 $(a_i, b_i, c_i)$ ，在实体集中创建一个新的实体。然后，在三个新联系集中，分别插入新联系集如下：

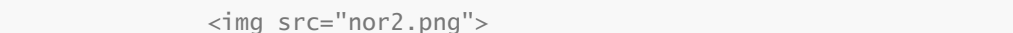
- \* 在 $R_A$ 中插入 $(e_i, a_i)$ ；
- \* 在 $R_B$ 中插入 $(e_i, b_i)$ ；
- \* 在 $R_C$ 中插入 $(e_i, c_i)$ ；

#### Drawbacks of converting:

\* 对于为表示联系集而创建的实体集，可能不得不为其创建一个表示属性(表示属性和额外需要的联系集增加了设计的复杂程度以及对总的存储空间的需求)；

\* A  $n$ -ray relationship set shows more clearly that several entities participate in a single relationship;

\* There may not be alway to translate constraints on the ternary relationship to constraints on the binary relationships;



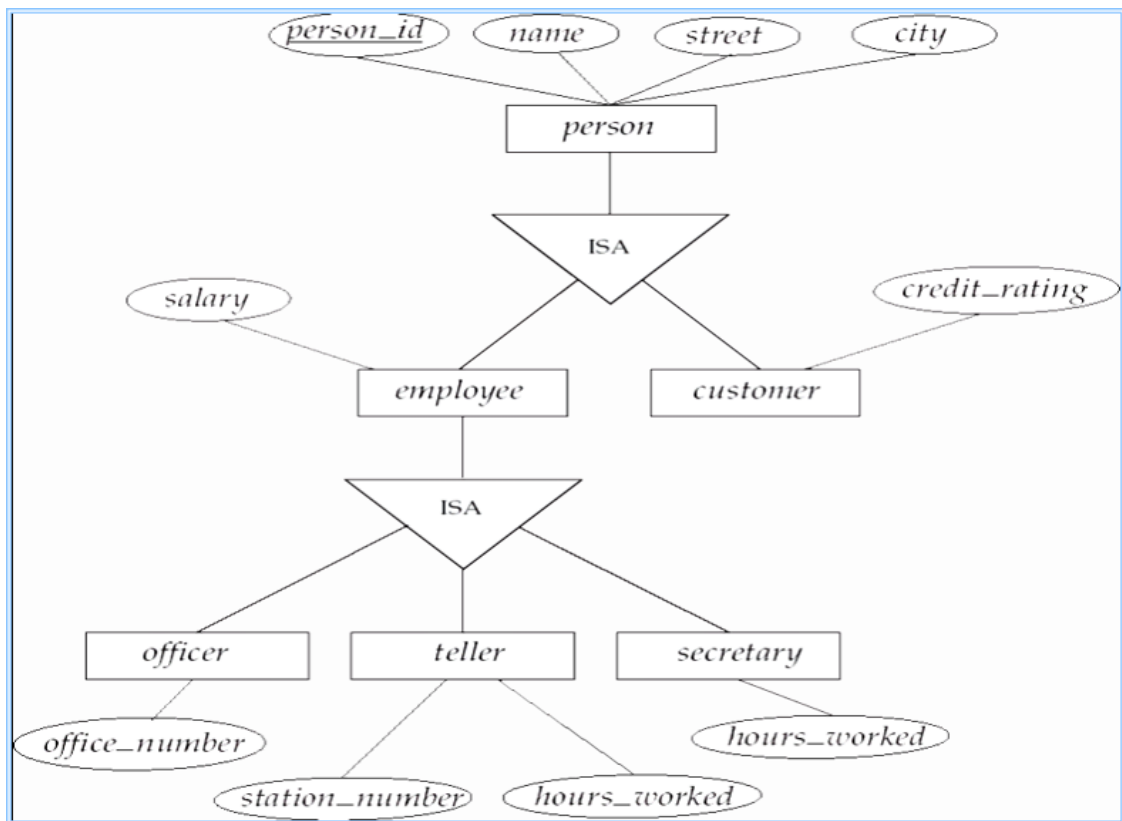
- 联系属性的布局 Placement of relationship attributes :
  - Attributes of one-to-many relationship set can be repositioned to only the entity set on the “many” side.
  - for one-to-one relationship set, On the other hand, can be placed on either sides.
  - For many-to-may relationship, attributes of relationship set can only be placed on relationship set.

## 5.5 扩展的E-R特性

- 特化(Specialization):

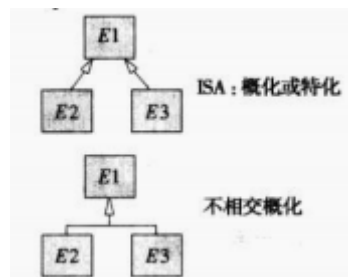


- 定义：在实体集内部进行分组的过程称为特化；
    - 自顶向下
    - Example: person特化为employee和student;
  - 一个实体集可以根据多个可区分的特征进行特化：
    - 当一个实体集上形成了多于一种特化时，某个特定实体可能同时属于多个特化实体集；
  - E-R图表示：用从特化实体指向另一实体的空心箭头表示；
    - 称这种关系为ISA关系，代表“is a”，表示“是一个”；
  - 特化从单一的实体集出发，通过创建不同的低层实体集来强调不同实体间的差异；
    - 低层实体集可以有不适用于高层实体集中所有实体的属性，也可以参与到不适用于高层实体集中所有实体的联系中；
- 高层与低层实体集也可以分别称作超类和子类；
- 概化(generalization):
    - 定义：高层实体集与一个或多个底层实体集间的包含关系
      - 自底向上
      - Example: instructor 与 secretary 可以概化为 employee
    - 对于所有实际应用来说，概化只不过是特化的逆过程；
    - 概化基于这样的认识：一定数量的实体集共享一些共同的特征(即用相同的属性描述它们且它们都参与到相同的联系集中)；
    - 概化是在这些实体集的共性的基础上将它们综合成一个高层实体集；
    - 概化用于强调低层实体集间的相似性并因此它们的差异；
  - 属性继承(Attribute inheritance):
    - 由特化和概化所产生的高层和低层实体的一个重要特性是**属性继承**；
    - 高层实体集的属性被低层实体集继承(inheit)，属性继承适用于所有低层实体集；
    - 低层实体集(或子类)同时还继承地参与其高层实体(或超类)所参与的联系集，参与继承适用于所有低层实体集；
    - Important:
      - 高层实体集所关联的所有属性和联系适用于它的所有低层实体集；
      - 低层实体集特有的性质仅适用于特定的低层实体集
    - 单继承、多继承(产生的结构成为格(lattice))；

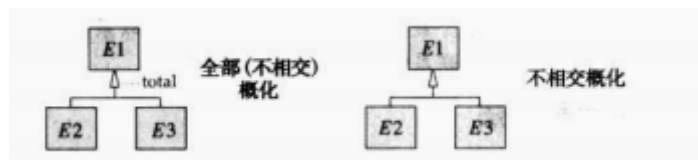


- 概化上的约束:

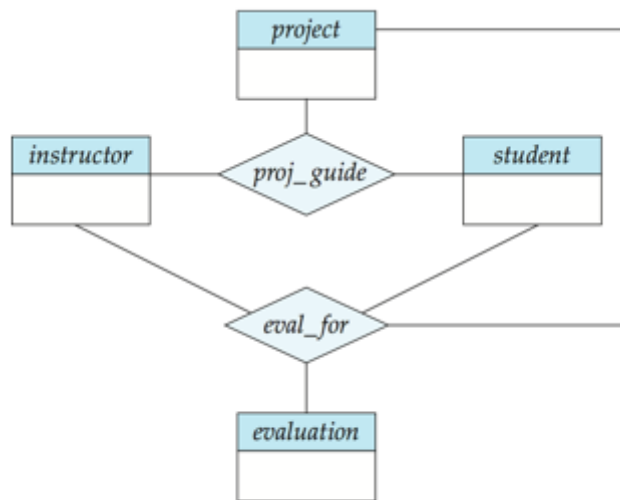
- 关于判定哪些实体能成为给定低层实体集的成员的约束。成员资格可以是下列的一种:
  - 条件定义的(condition-defined):
    - 成员资格的确定基于实体是否满足一个显示的条件或谓词
  - 用户定义的(user-defined):
    - 由数据库用户将实体指派给每个工作组
- 关于在**同一个概化**中一个实体是否可以属于多个低层实体集。低层实体集可能是下述情况之一:
  - 不相交 disjoint:
    - 不相交约束要求一个实体至多属于一个低层实体集;
  - 重叠 overlapping:
    - 同一个实体可以同时属于**同一个概化**中的多个低层实体集;



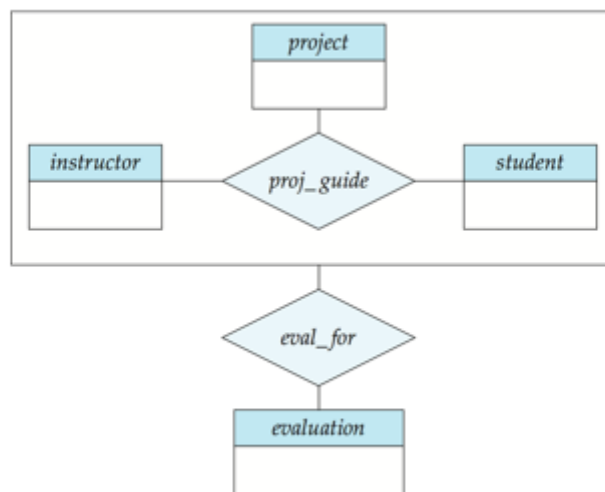
- 对概化的**完全性约束(completeness constraint)**: 定义高层实体集中的一个实体是否必须至少属于该概化/特化的一个低层实体集。约束情况可以为:
  - 全部概化(total generalization):
    - 每个高层实体必须属于一个低层实体集;
  - 部分概化(partial generalization):
    - 允许一些高层实体不属于任何低层实体集;
    - 默认为部分概化, 通过添加关键词“total”改为全部概化



- 聚集:
  - Introduce:



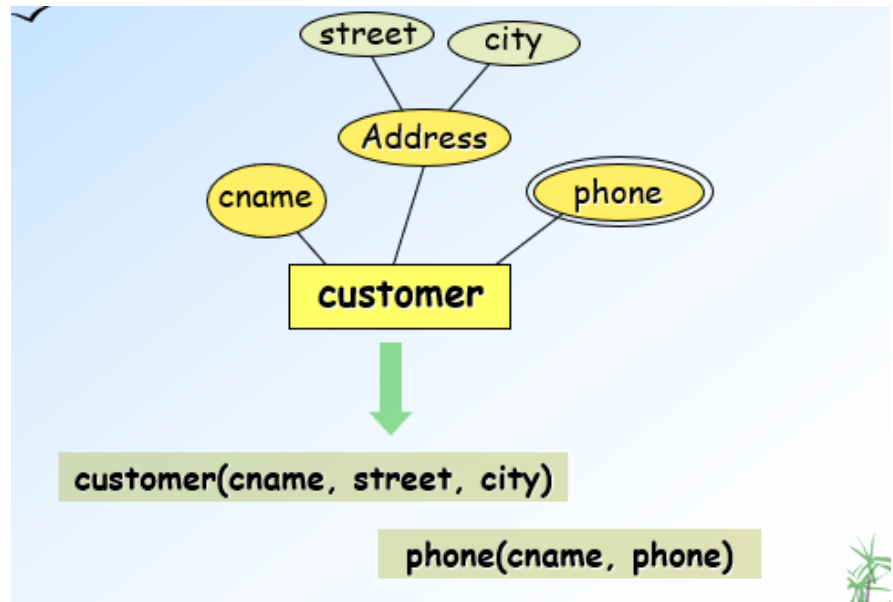
- 冗余: eval\_for 中的每个instructor、student、project组合也在 proj\_guide
    - 上述情况最好的建模办法: 使用聚集(聚集是一种抽象, 通过这种抽象, 联系被视为高层实体)



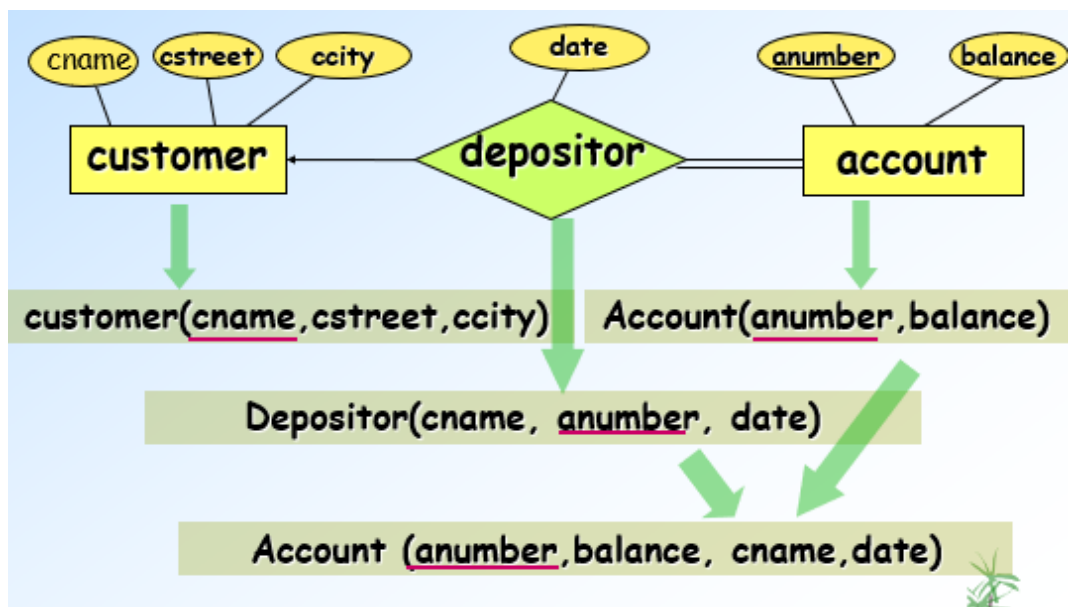
## 5.4 转换为关系模式(Reduction to Relation Schema)

- 实体集表示
  - 强实体集表示
    - 强实体集的主码就是生成的模式的主码;
    - 具有简单属性的强实体集表示:
      - 设E是只具有简单描述性属性 $a_1, a_2, \dots, a_n$ 的强实体集, 则我们使用具有n个不同属性的模式表示这个实体集(该模式的关系的每个元组与同实体集的一个实体相对应);
    - 具有复杂属性的强实体集表示:
      - 复合属性:

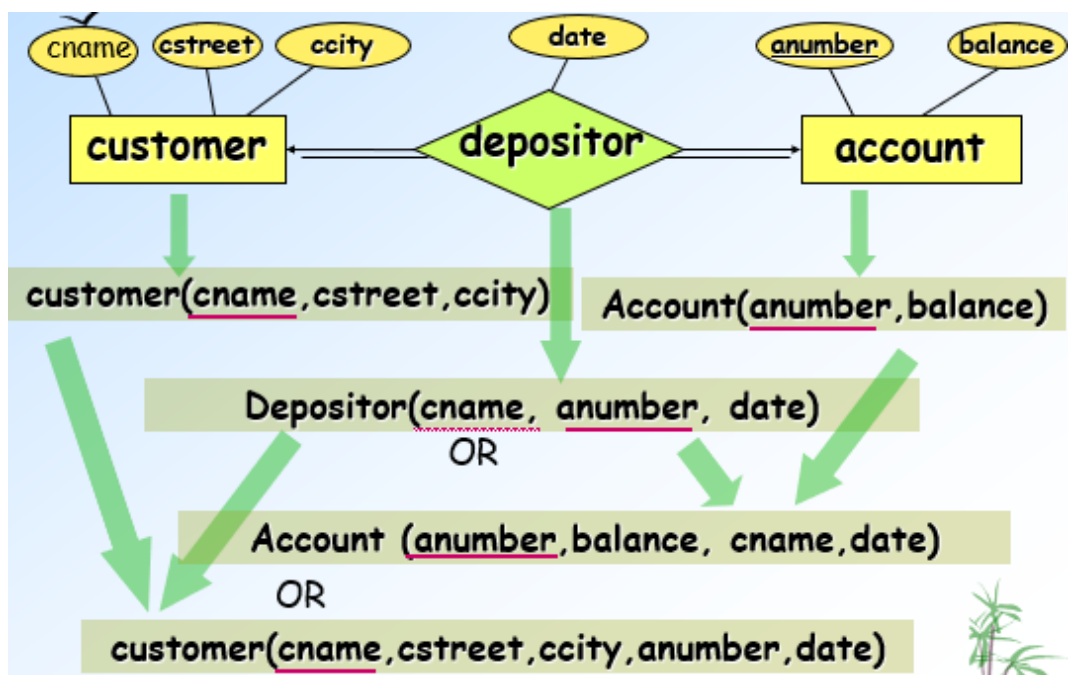
- 为每个子属性创建一个单独的属性，而不为复合属性自身创建一个单独的属性；
- 多值属性：
  - 对于一个多值属性M，构建关系模式R，该模式包含一个对应于M的属性A，以及对应于M所在实体集或联系集的主码的属性：



- Each value of the multivalued attribute maps to a separate tuple of the relation on schema R;
  - 构建的关系模式的主码由模式中的所有属性组成；
  - 在多值属性构建的关系模式上建立外码约束：由实体集的主码所生成党的属性去参照实体集所生成的关系；
- 派生属性：
  - 不在关系数据模型中显式的表示出来；
- 弱实体集表示：
  - 设A是具有属性 $a_1, a_2, \dots, a_n$ 的弱实体集，设B是A所依赖的强实体集，设B的主码包括属性 $b_1, b_2, \dots, b_n$ ，则由弱实体集A转换而来的模式A的属性集合为：
 
$$\{a_1, a_2, \dots, a_n\} \cup \{b_1, b_2, \dots, b_n\}$$
  - 由弱实体集A转换而来的模式的主码由其所依赖的强实体集的主码与弱实体集的分辨符组合而成，外码约束则指明属性 $b_1, b_2, \dots, b_n$ 参照关系B的主码；
- 联系集表示：
  - A many-to-many relationship set is represented as a schema with attributes for the primary keys of the two participating entity sets, and any descriptive attributes of the relationship set.
  - Many-to-one and one-to-many relationship sets that are total on the many-side can be represented by adding an extra attribute to the “many” side, containing the primary key of the “one” side.(联系集“多”的那一方实体集的主码构成主码)



- For one-to-one relationship sets, either side can be chosen to act as the "many" side
  - That is, extra attribute can be added to either of the tables corresponding to the two entity sets.



任何一个实体集的主码都能选作为主码；

- 模式的冗余与合并
- 概化的表示：
  - Method one:
    - 为高层实体集创建一个模式；
    - 为每个低层实体集创建一个模式，模式中的属性包括对应于低层实体集的每个属性以及对应于高层实体集主码的每个属性；

schema	attributes
person	name, street, city
customer	name, credit_rating
employee	name, salary

Drawback: getting information about an *employee* requires accessing two relations, the one corresponding to the low-level schema and the one corresponding to the high-level schema.

- Method two:

- Form a schema for each entity set with all local and inherited attributes
- If specialization is total, the schema for the generalized entity set (*person*) not required to store information (Can be defined as a “view” relation containing union of specialization relations; •But explicit schema may still be needed for foreign key constraints)

schema	attributes
<i>person</i>	<i>name, street, city</i>
<i>customer</i>	<i>name, street, city, credit_rating</i>
<i>employee</i>	<i>name, street, city, salary</i>

Drawback: *street* and *city* may be stored redundantly for people who are both customers and employees

- 聚集的表示:

- To represent aggregation, create a schema containing
  - primary key of the aggregated relationship(*proj\_guide*)
  - the primary key of the associated entity set(*evaluation*)
  - any descriptive attributes (*eval\_for*)

## 第六章 关系数据库设计

一般而言，关系数据库的目标是生成一组关系模式，使我们存储信息时避免不必要的冗余，并且可以让我们方便的获取信息。

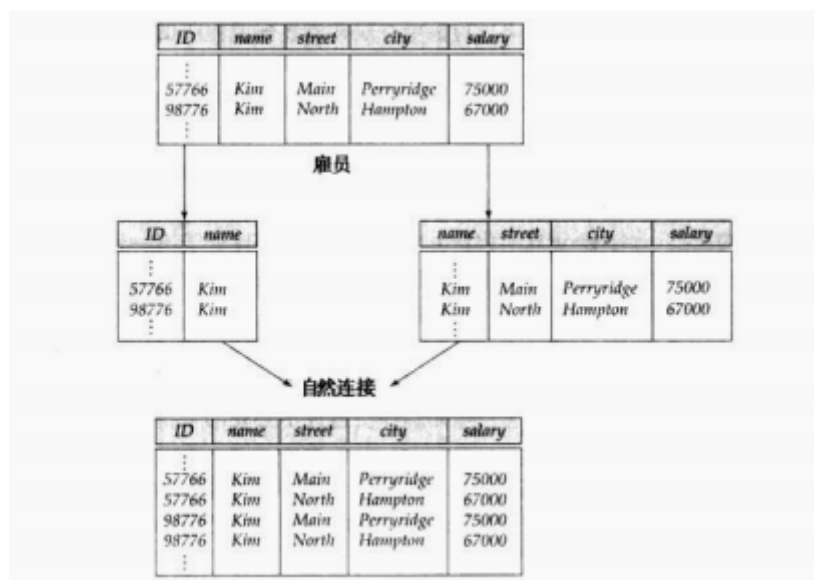
Goal:

- Decide whether a particular relation  $R$  is in “good” form;
- In the case that a relation  $R$  is not in “good” form, decompose it into a set of relations  $R_1, R_2, \dots, R_n$  such that
  - Each relation is in good form;
  - The decomposition is a lossless-join decomposition (无损连接分解);

### 6.1 Features of Good Relational Design (好的关系设计的特点)

- 设计选择: 更大的模式
  - 数据冗余
  - 数据不一致
- 设计选择: 更小的模式
  -





- lossy decomposition (有损分解)

## 6.2 Atomic Domains and First Normal Form (原子域和第一范式)

- Atomic Domains:
  - Domain is atomic if its elements are considered to be indivisible units;
 

域是原子的，如果该域的元素被认为是不可分的单元;
- First Normal Form:
  - A relational schema  $R$  is in *first normal form* if the domains of all attributes of  $R$  are atomic;
 

关系模式 $R$ 是第一范式(1NF)，如果 $R$ 的所有属性的域都是原子的;
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data;
- Atomicity is actually a property of how the elements of the domain are used

## 6.3 Decomposition Using Functional Dependencies (使用函数依赖进行分解)

1. 希腊字母表示属性集(阿尔法)
2.  $r(R)$ 指代关系模式(表示该模式是关系 $r$ 的， $R$ 表示属性集)
3.  $K$ 表示超码

- Functional Dependencies (函数依赖)——A generalization of the notion of a *key*
  - Constraints on the set of **legal relations** (合法关系的集合上的约束);
  - Require that the value for a certain set of attributes determines uniquely the value for another set of attributes;

考虑一个关系模式 $r(R)$ ，令 $\alpha \subseteq R$ 且 $\beta \subseteq R$ ,

- 给定 $r(R)$ 的一个实例，则该实例满足(satisfy)函数依赖 $\alpha \rightarrow \beta$ 的条件是：对实例中的任意二个元组 $t_1$ 和 $t_2$ ，若 $t_1[\alpha] = t_2[\alpha]$ ，则 $t_1[\beta] = t_2[\beta]$ .
- 如果 $r(R)$ 的每个合法实例均满足函数依赖 $\alpha \rightarrow \beta$ ，则该函数依赖在模式 $r(R)$ 上成立(hold);

- $K$  is a **super key** for relation schema  $r(R)$  if and only if  $K \rightarrow R$ .

- **$K$  is a candidate key (候选码) for relation schema  $r(R)$  if and only if  $K \rightarrow R$  and for no  $\alpha \subset K, \alpha \rightarrow R$ .**

- Functional dependencies allow us to express constraints that cannot be expressed using super keys (函数依赖使我们可以表示不能用超码表示的约束);

inst\_dept(ID, name, salary, dept\_name, building, budget)

- 用超码表示的约束:  $(ID, dept\_name) \rightarrow name, salary, building, budget$ ;
- 用函数依赖表示的约束:  $dept\_name \rightarrow budget$ ;

- 我们使用函数依赖来:
  - 判定关系是否满足给定函数依赖集;
    - If relation  $r$  is legal under a set  $F$  of functional dependencies, we say that  $r$  satisfies  $F$
  - 说明合法关系集上的约束;
    - If all legal relations on  $R$  satisfy the set of functional dependencies  $F$ , We say that  $F$  holds on  $R$  ( $F$ 在 $R$ 上成立).

A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.

- 在所有关系都满足的函数依赖是**平凡(trivial)**的:
- **一般地, 如果  $\alpha \subseteq \beta$ , 则形如  $\alpha \rightarrow \beta$  的函数依赖是平凡的;**
- 函数依赖集的闭包:

- Given a set  $F$  of functional dependencies, there are certain other functional dependencies that are logically implied by  $F$ :

If  $A \rightarrow B$  and  $B \rightarrow C$ , then we can infer that  $A \rightarrow C$

- The set of all functional dependencies logically implied by  $F$  is the *closure* of  $F$ , we denote the closure of  $F$  by  $F^+$  ( $F^+$  is a superset of  $F$ );

- **Boyce-Codd Normal Form, BCNF**

- 具有函数依赖集 $F$ 的关系模式 $R$ 属于 **BCNF** 的条件是,  $F^+$ 的所有形如的 $\alpha \rightarrow \beta$ 函数依赖, 下面至少有一项成立:
  - $\alpha \rightarrow \beta$ 是平凡的函数依赖;
  - $\alpha$  是模式 $R$ 的一个超码;

一个数据库设计属于 **BCNF** 的条件是, 构成该设计的关系模式集中的每个模式都属于 **BCNF**;

- 分解不属于 **BCNF** 的模式的一般规则 **important**
  - 设 $R$ 不属于 **BCNF** 的一个模式, 则存在至少一个非平凡的函数依赖 $\alpha \rightarrow \beta$ , 其中 $\alpha$ 不是超码, 我们使用以下两个模式取代 $R$ (BCNF的分解算法):
    - $(\alpha \cup \beta)$
    - $(R - (\beta - \alpha))$

inst\_dept(ID, name, salary, dept\_name, building, budget)

- 超码:  $(ID, dept\_name)$ ;
- $a = dept\_name, b = \{building, budget\}$
- $\alpha \cup \beta = (dept\_name, building, budget)$
- $(R - (\beta - \alpha)) = (ID, name, dept\_name, salary)$

- **BCNF 和保持依赖?**
  - Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation;
  - If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is *dependency preserving*.

- **第三范式 3NF**
  - 具有函数依赖集F的关系模式R属于第三范式的条件是,  $F^+$ 的所有形如 $a \rightarrow b$ 的函数依赖, 下面至少有一项成立:
    - $\alpha \rightarrow \beta$ 是平凡的函数依赖;
    - $\alpha$ 是模式R的一个超码;
    - $\beta - \alpha$  的每个属性A都包含于R的一个候选码中;

1.注意: 条件三并未说单个候选码必须包含 $\beta - \alpha$ 的所有属性,  $\beta - \alpha$ 的每个属性A可能包含于不同的候选码中。

2.某种意义上, 条件三代表 **BCNF** 条件的最小放宽, 以确保每一个模式都有保持依赖的 **3NF** 分解。(Third condition is a minimal relaxation of **BCNF** to ensure depend-ency preservation)
  - 任何满足 **BCNF** 的模式也满足 **3NF** (**BCNF** 是比 **3NF** 更严格的范式);

## 6.4 函数依赖理论

- 函数依赖集的闭包
  - 给定关系模式r(R), 如果r(R)的每一个满足F的实例也满足f, 则R上的函数依赖f被r上的函数依赖集F逻辑蕴含(logically imply);
  - F是一个函数依赖集, F的闭包是被F逻辑蕴涵的所有函数依赖的集合, 记作 $F^+$ ;
  - Armstrong 公理(sound and complete, 正确有效和完备的):
    - 自反律 **reflexivity rule**: 若 $\alpha$ 为一属性集且 $\beta \subseteq \alpha$ , 则 $\alpha \rightarrow \beta$ 成立;
    - 增补律 **augmentation rule**: 若 $\alpha \rightarrow \beta$ 且 $\gamma$ 为一属性集, 则 $\gamma\alpha \rightarrow \gamma\beta$ 成立;
    - 传递律 **transitivity rule**: 若 $\alpha \rightarrow \beta$ 和 $\beta \rightarrow \gamma$ 成立, 则 $\alpha \rightarrow \gamma$ 成立;
  - Other rule:
    - 合并律 **union rule**: 若 $\alpha \rightarrow \beta$ 和 $\alpha \rightarrow \gamma$ 成立, 则 $\alpha \rightarrow \beta\gamma$ 成立;
    - 分解律 **decomposition**: 若 $\alpha \rightarrow \beta\gamma$ 成立, 则 $\alpha \rightarrow \beta$ 和 $\alpha \rightarrow \gamma$ 成立;
    - 伪传递律 **pseudotransitivity**: 若 $\alpha \rightarrow \beta$ 和 $\gamma\beta \rightarrow \delta$ 成立, 则 $\alpha\gamma \rightarrow \delta$ 成立;
  - Procedure for computing  $F^+$ :

```

F+=F
repeat
  for each f in F+
    apply reflexivity and augmentation rules on f
    add the resulting functional dependencies to F+
  for each pair of f1 and f2 in F+
    if f1 and f2 can be combined using transitivity
      then add the resulting functional dependency to F+
until F+ does not change any further

```

- 属性集的闭包 important

- $\alpha$  为一个属性集，则函数依赖集  $F$  下被  $\alpha$  确定的所有属性集合称  $F$  下为  $\alpha$  的闭包
- 计算  $F$  下  $\alpha$  的闭包  $\alpha^+$  的算法：

```

result :=  $\alpha$ 
repeat
  for each 函数依赖  $\beta \rightarrow \gamma$  in  $F$  do
    begin
      if  $\beta \subseteq \text{result}$  then result := result  $\cup \gamma$ 
    end
until no changes in results

```

- 用途：
  - 超码判断：
    - 为了判断  $\alpha$  是否为超码，计算  $\alpha^+$ ，检查  $\alpha^+$  是否包含  $R$  的所有属性
  - $F$  的闭包：
    - 对于任意的  $\gamma \subseteq R$ ，找出闭包  $\gamma^+$ ；对任意的  $S \subseteq \gamma^+$ ，输出一个函数依赖  $\gamma \rightarrow S$ ；
  - 测试函数依赖：
    - 通过检查是否  $\beta \subseteq \alpha^+$ ，可以检查函数依赖  $\alpha \rightarrow \beta$  是否成立(或换句话说，是否属于  $F^+$ )；

- **正则覆盖 Canonical cover:**

- **无关属性**(extraneous attribute):
  - 如果去除函数依赖中的一个属性不改变该函数依赖集的闭包，则称该属性是无关的；
  - 形式化定义：考虑函数依赖集  $F$  及  $F$  中的函数依赖  $\alpha \rightarrow \beta$ ，
    - 如果  $A \in \alpha$  并且  $F$  逻辑蕴涵  $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ ，则属性  $A$  在  $\alpha$  中是无关的；
    - 如果  $A \in \beta$  并且函数依赖集  $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$  逻辑蕴涵  $F$ ，则属性  $A$  在  $\beta$  中是无关的；
- 如何有效检验一个属性是否无关？考虑函数依赖集  $F$  的包含属性  $A$  的函数依赖  $\alpha \rightarrow \beta$ 。
  - 如果  $A \in \alpha$ ，为了检验  $A$  是否是无关的，令  $\gamma = \alpha - A$ ，并且检查  $\gamma \rightarrow \beta$  是否可以由  $F$  推出：为此计算  $F$  下的  $\gamma^+$ ，如果  $\gamma$  包含  $\beta$  的所有属性，则  $A$  在  $\alpha$  中是无关的；
  - 如果  $A \in \beta$ ，为了检验  $A$  是否是无关的，考虑集合
 
$$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$$
 并检验  $\alpha \rightarrow A$  是否能够由  $F'$  推出：为此，计算  $F'$  下的  $\alpha^+$ ，如果  $\alpha^+$  包含  $A$ ，则  $A$  在  $\beta$  中是无关的；
- $F$  的**正则覆盖**  $F_c$  是一个依赖集，使得  $F$  逻辑蕴涵  $F_c$  的所有依赖，并且  $F_c$  逻辑蕴涵  $F$  中的所有依赖， $F_c$  还必须具有以下性质：
  - $F_c$  的任何函数依赖都不含无关属性。
  - $F_c$  函数依赖的左半部分都是唯一的：即， $F_c$  中不存在两个依赖  $\alpha_1 \rightarrow \beta_1$  和  $\alpha_2 \rightarrow \beta_2$ ，满足  $\alpha_1 = \alpha_2$ 。
- **计算正则覆盖:**

```

 $F_c = F$ 
repeat
  使用合并律将  $F_c$  中所有形如  $\alpha_1 \rightarrow \beta_1$  和  $\alpha_1 \rightarrow \beta_2$  的依赖替换为  $\alpha_1 \rightarrow \beta_1 \beta_2$ 
  在  $F_c$  中寻找一个函数依赖  $\alpha \rightarrow \beta$ ，它在  $\alpha$  和  $\beta$  中具有一个无关属性
  (注意：使用  $F_c$  而非  $F$  检验无关属性)

```

如果找到一个无关属性, 则将它从 $F_c$ 中的 $\alpha \rightarrow \beta$ 中删除  
until  $F_c$  has no change

◦ 正则覆盖未必是唯一的

• 无损分解(lossless decomposition)

令 $r(R)$ 为一个关系模式,  $F$ 为 $r(R)$ 上的函数依赖集, 而 $R_1$ 和 $R_2$ 为 $R$ 的分解。如果用两个关系模式 $r_1(R_1)$ 和 $r_2(R_2)$ 替代 $r(R)$ 时没有信息损失( $\Pi_{R_1} \bowtie \Pi_{R_2} = r$ ), 则我们称该分解是无损分解。

不是无损分解的分解称为有损分解。无损连接分解和有损连接分解这两个术语有时用来代替无损分解和有损分解。

◦  $R_1$ 和 $R_2$ 是 $R$ 的无损分解, 如果以下函数依赖中至少有一个术语 $F^+$ :

■  $R_1 \cap R_2 \rightarrow R_1$ ;

■  $R_1 \cap R_2 \rightarrow R_2$ ;

换句话说, 如果 $R_1 \cap R_2$ 是 $R_1$ 或 $R_2$ 的超码,  $R$ 上的分解就是无损分解

• 保持依赖(dependency preserving)

令 $F$ 为模式 $R$ 上的一个函数依赖集,  $R_1, R_2, \dots, R_n$ 为 $R$ 的一个分解。 $F$ 在 $R_i$ 上的限定是 $F^+$ 中所有只包含 $R_i$ 中属性的函数依赖的集合 $F_i$ (restriction, 限定)。由于一个限定中的所有函数依赖只涉及一个关系模式的属性, 因此判定这种依赖是否满足可以只检查一个关系。

限定 $F_1, F_2, \dots, F_n$ 的集合是能高效检查的依赖集。令 $F' = F_1 \cup F_2 \cup \dots \cup F_n$ ,  $F'$ 是模式 $R$ 上的一个函数依赖集, 不过通常 $F' \neq F$ , 但是即使 $F' \neq F$ ,  $F'^+ = F^+$ 是可能的。我们称满足 $F'^+ = F^+$ 的分解是保持依赖的分解。

◦ 判定保持依赖性的算法: 输入为分解的关系模式集 $D = R_1, R_2, \dots, R_n$ 和函数依赖集 $F$

计算 $F^+$ :

```
for each  $R_i$  in  $D$  do
  begin
     $F_i := F^+$ 在 $R_i$ 中的限定;
  end
 $F' := \emptyset$ 
计算 $F'^+$ ;
if ( $F'^+ = F^+$ ) then return true
else return false
```

■ 替代方法(避免计算 $F^+$ ):

■ 如果 $F$ 中的每一个函数依赖都可以在分解得到的某一个关系上验证, 那么这个分解就是保持依赖的;

■ 对 $F$ 中的每一个 $\alpha \rightarrow \beta$ , 使用下面的过程:

```
result =  $\alpha$ 
repeat
  for each 分解后的 $R_i$ 
     $t = (result \cap R_i)^+ \cap R_i$ 
     $result = result \cup t$ 
until result has no change
```

如果result包含 $\beta$ 的所有属性, 则函数依赖 $\alpha \rightarrow \beta$ 保持, 分解是保持的当且仅当上述过程中F的所有依赖都保持;

Takes *polynomial time*

## 6.5 分解算法

- BCNF 分解

- BCNF 的判定方法:

- 在某些情况下, 判定一个关系是否属于BCNF可以作如下简化:

- 为了检查非平凡的函数依赖 $\alpha \rightarrow \beta$ 是否违反BCNF, 计算 $\alpha^+$ , 并验证它是否包含R中的所有属性, 即验证它是否是R的超码;
- 检查关系模式R是否属于BCNF, 仅需检查给定结合F中的函数依赖是否违反BCNF就可, 不用检查 $F^+$ 的所有函数依赖(当关系分解后不再适用);

可以证明, 如果F中没有函数依赖违反BCNF, 那么 $F^+$ 中也不会有函数依赖违反BCNF;

- 为了检查分解后的关系 $R_i$ 是否属于BCNF, 应用如下判定:

- 对于 $R_i$ 中属性的每个子集 $\alpha$ , 确保 $\alpha^+$  (F下 $\alpha$ 的属性闭包)要么不包含 $R_i - \alpha$ 的任何属性, 要么包含 $R_i$ 的所有属性;

- BCNF 分解算法(利用违反BCNF的依赖进行分解):

```
result:={R};
done:=false;
compute  $F^+$ 
where (not done) do
  if (there is a schema  $R_i$  in result that is not in BCNF)
  then
    begin
      令 $\alpha \rightarrow \beta$ 为一个在 $R_i$ 上成立的非平凡函数依赖,
      满足 $\alpha \rightarrow R_i$ 不属于 $F^+$ , 并且 $\alpha \cap \beta = \emptyset$ ;
      result:=(result- $R_i$ ) $\cup$ ( $R_i - \beta$ )  $\cup$  ( $\alpha, \beta$ );
    end
  else done:=true;
# 此算法产生BCNF分解是无损分解;
```

- 3NF 分解

- 分解算法(保持依赖且无损的分解算法)

```
令 $F_c$ 为F的正则覆盖;
i:=0;
for each  $F_c$  中的函数依赖 $\alpha \rightarrow \beta$ 
  i:=i+1;
   $R_i := \alpha\beta$ ;
if 模式 $R_j, j = 1, 2, \dots, i$ 都不包含R的候选码
then
```



```

    i:=i+1
     $R_i := R$  的任意候选码;
    →
    /* 移除冗余关系(可选) */
    repeat
        if 模式  $R_j$  包含于另一个模式  $R_k$  中
        then
            /* 删除  $R_j$  */
             $R_j := R_i$ 
        i:=i-1
    until 不能有可以删除的  $R_j$ 
    return ( $R_1, R_2, \dots, R_i$ )
    # 多项式时间

```

- 3NF算法的正确性:
  - 3NF算法(也称为3NF合成算法)通过保证至少有一个模式包含被分解的模式的候选码, 确保该分解是一个无损分解;
  - 3NF算法通过为正则覆盖中的每个依赖显式地构造一个模式确保依赖的保持;
- BCNF 和 3NF 的比较

## 6.6 使用多值依赖的分解

- 多值依赖
  - Intro

函数依赖规定了某些元组不能出现在关系中(比如, 如果  $A \rightarrow B$  成立, 则不能有两个元组在A上的值相同而在B上的值不同); 从另一个角度出发, 多值依赖并不排除某些元组的存在, 而是要求某种形式的其它元组存在于关系中。由于这个原因, 函数依赖有时称为相等产生依赖(equality-generating dependency), 多值依赖称为元组产生依赖(tuple-generating dependency).

- Definition

令  $r(R)$  为一关系模式, 并另  $\alpha \subseteq R$  且  $\beta \subseteq R$ , 多值依赖(multivalued dependency)

$$\alpha \twoheadrightarrow \beta$$

在  $R$  上成立的条件是, 在关系  $r(R)$  的任意合法实例中, 对于  $r$  中任意一对满足  $t_1[\alpha] = t_2[\alpha]$  的元组对  $t_1$  和  $t_2$ ,  $r$  中都存在元组  $t_3$  和  $t_4$ , 使得

$$(1) \ t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$

$$(2) \ t_1[\beta] = t_3[\beta]$$

$$(3) \ t_2[\beta] = t_4[\beta]$$

$$(4) \ t_2[R - \beta] = t_3[R - \beta]$$

$$(5) \ t_1[R - \beta] = t_4[R - \beta]$$

表格表示:

	$\alpha$	$\beta$	$R - \alpha - \beta$
$t_1$	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
$t_2$	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
$t_3$	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
$t_4$	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$

直观的，多值依赖  $\alpha \twoheadrightarrow \beta$  是说  $\alpha$  和  $\beta$  之间的联系独立于  $\alpha$  和  $R - \alpha - \beta$  之间的联系。若模式  $R$  上所有关系都满足多值依赖  $\alpha \twoheadrightarrow \beta$ ，则  $\alpha \twoheadrightarrow \beta$  是模式  $R$  上的平凡的多值依赖。

◦ Usage: 与函数依赖相同，我们使用两种方式使用多值依赖

- 检验关系以确定它们在给定的函数依赖集和多值依赖集下是否合法；
- 在合法关系集上指定约束；

请注意，若关系  $r$  不满足给定多值依赖，我们可以通过向  $r$  中增加元组构造一个确实满足多值依赖的关系  $r'$ ；

◦ 由多值依赖的定义，存在以下规则：

- 若  $\alpha \rightarrow \beta$ ，则  $\alpha \twoheadrightarrow \beta$  (换句话说，每一个函数依赖也是一个多值依赖)；
- 若  $\alpha \twoheadrightarrow \beta$ ，则  $\alpha \twoheadrightarrow R - \alpha - \beta$ ；

多值依赖有助于我们理解并消除某些形式的信息重复，而这种信息重复用函数依赖是无法理解的；

• 第四范式：

◦ 函数依赖和多值依赖集为  $D$  的关系模式  $r(R)$  属于第四范式(4NF)的条件是，对  $D^+$  中所有形如  $\alpha \twoheadrightarrow \beta$  的多值依赖 ( $\alpha \subseteq R, \beta \subseteq R$ )，至少有以下之一成立：

- $\alpha \twoheadrightarrow \beta$  是一个平凡的函数依赖；(若  $\beta \subseteq \alpha$  或  $\beta \cup \alpha = R$ ，则  $\alpha \twoheadrightarrow \beta$ )；
- $\alpha$  是  $R$  的一个超码；

数据库模式设计属于4NF的条件是构成该设计的关系模式集中的每个模式都属于4NF；

◦ If a relation is in 4NF it is in BCNF;

◦ Restriction of Multivalued Dependencies: 多值依赖的限定

令  $r(R)$  为关系模式， $r_1(R_1), r_2(R_2), \dots, r_n(R_n)$  为  $r(R)$  的分解，要检验中分解的每一个关系模式  $r_i$  是否属于4NF，需要找到  $r_i$  成立的多值依赖。

考虑函数依赖和多值依赖的集合  $D$ ， $D$  在  $R_i$  上的**限定**是集合  $D_i$ ，它包含

- $D^+$  中所有只含  $R_i$  中属性的函数依赖；
- 所有形如：  $\alpha \twoheadrightarrow (\beta \cap R_i)$  的多值依赖，其中  $\alpha \subseteq R_i$  且  $\alpha \twoheadrightarrow \beta$  属于  $D^+$ ；

• 4NF 分解？：

◦ 分解算法(只产生无损分解，保持依赖的问题更加复杂，暂不涉及)：

```

result:={R}
done:=false
计算  $D^+$ ：给定模式  $R_i$ ，令  $D_i$  表示在  $R_i$  上的限定
while(not done) do
    if (result存在  $R_i$  不属于4NF)
    then begin

```

```

    令  $\alpha \twoheadrightarrow \beta$  为  $R_i$  上成立的非平凡多值依赖,
    使得  $\alpha \rightarrow R_i$  不属于  $D_i$ , 并且  $\alpha \cap \beta = \emptyset$ ;
    result := (result -  $R_i$ )  $\cup$  ( $R_i - \beta$ )  $\cup$  ( $\alpha, \beta$ );
end
else done := true;

```

◦ Example:

□  $R = (A, B, C, G, H, I)$   
 $F = \{ A \twoheadrightarrow B$   
 $B \twoheadrightarrow HI$   
 $CG \twoheadrightarrow H \}$

□  $R$  is not in 4NF since  $A \twoheadrightarrow B$  and  $A$  is not a superkey for  $R$

□ Decomposition

- a)  $R_1 = (A, B)$  ( $R_1$  is in 4NF)
- b)  $R_2 = (A, C, G, H, I)$  ( $R_2$  is not in 4NF, decompose into  $R_3$  and  $R_4$ )
- c)  $R_3 = (C, G, H)$  ( $R_3$  is in 4NF)
- d)  $R_4 = (A, C, G, I)$  ( $R_4$  is not in 4NF, decompose into  $R_5$  and  $R_6$ )
  - $A \twoheadrightarrow B$  and  $B \twoheadrightarrow HI \Rightarrow A \twoheadrightarrow HI$ , (MVD transitivity), and
  - and hence  $A \twoheadrightarrow I$  (MVD restriction to  $R_4$ )
- e)  $R_5 = (A, I)$  ( $R_5$  is in 4NF)
- f)  $R_6 = (A, C, G)$  ( $R_6$  is in 4NF)

## 6.7 更多的范式

- 其它约束与范式:
  - 连接依赖(join dependency)约束:
    - Lead to Project-Join Normal Form (投影连接范式, PJNF)
  - 更一般化的约束: Lead to Domain-Key Normal Form (域-码范式, DKNF)
- Problem:
  - 难以推导;
  - 没有形成一套具有正确有效性和完备性的推理规则用于约束的推导;

## 6.8 数据库设计过程

How to get relation schema  $r(R)$ :

- $r(R)$ 可以由E-R图向关系模式集进行转换时生成的;
- $r(R)$ 可以是一个包含所有有意义的属性的单个关系, 然后规范化过程中将 $R$ 分解成一些更小的模式;
- $r(R)$ 可以是关系的即席设计的结果, 然后我们检验它们是否满足一个期望的范式;
- E-R 模型和规范化
  - When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
  - However, in a real (imperfect) design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity.
  - 包含两个实体集以上的联系集有可能或使产生的模式不属于所期望的范式(由于大部分的联系集都是二元的, 故这种情况比较少);

- 规范化可以作为数据建模的一部分规范地进行;
- 属性和联系的命名
- 为了性能去规范化
  - May want to use non-normalized schema for performance;
 

不使用规范化模式的代价是用来保持冗余数据一致性的额外工作(以编码时间和执行时间计算);
  - 把一个规范化的模式变成非规范化的过程称为去规范化, 设计者用它去调整系统的性能以支持响应时间苛刻的操作;
  - For example, displaying *prereqs* along with *course\_id*, and *title* requires join of *course* with *prereq*
    1. Use denormalized relation containing attributes of *course* as well as *prereq* with all above attributes.
      - Faster lookup;
      - Extra space and extra execution time for updates;
      - Extra coding work for programmer and possibility of error in extra code;
    2. Use a materialized view defined as (course natural join prereq): 物化视图存储
      - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors
- 其它设计问题

数据库设计中有一些方面不能用规范化描述, 而它们也会导致不好的数据库设计。与时间或时间段有关的数据就存在这样的问题。这里我们给出一些例子; 显然, 这样的设计应该避免。

考虑一个大学数据库, 我们想存储不同年份中每个系的教师总数。可以用关系 *total\_inst*(*dept\_name*, *year*, *size*) 来存储所需要的信息。这个关系上的唯一的函数依赖是 *dept\_name*, *year*  $\rightarrow$  *size*, 这个关系属于 BCNF。

另一个设计是使用多个关系, 每个关系存储一个年份的系规模信息。假设我们感兴趣的年份为 2007、2008 和 2009; 我们将得到形如 *total\_inst\_2007*, *total\_inst\_2008*, *total\_inst\_2009* 这样的关系, 它们都建立在模式(*dept\_name*, *size*) 之上。因为每一个关系上的唯一的函数依赖是 *dept\_name*  $\rightarrow$  *size*, 所以这些关系也属于 BCNF。

但是, 这个设计显然是一个不好的主意——我们必须每年都创建一个新的关系, 每年还不得不写新的查询从而把新的关系考虑进去。由于可能涉及很多关系, 因此查询也将更加复杂。

然而还有一种方法可以表示同样的数据, 即建立一个单独的关系 *dept\_year*(*dept\_name*, *total\_inst\_2007*, *total\_inst\_2008*, *total\_inst\_2009*)。这里唯一的函数依赖是从 *dept\_name* 指向其他属性的依赖, 该关系也属于 BCNF。这种设计也是一个不好的想法, 因为它存在与前面类似的问题, 就是每年我们都不得不修改关系模式并书写新的查询。由于可能涉及很多属性, 因此查询也会变得更加复杂。

像 *dept\_year* 关系中那样的表示方法, 属性的每一个值一列, 叫作交叉表(crosstab)。它们在电子数据表、报告和数据分析工具中广泛使用。虽然这些表示方法显示给用户看是很有用的, 但是由于上面给出的原因, 它们在数据库的设计中并不可取。如 5.6.1 节所述, 为了显示方便, SQL 扩展已经提出了将数据从规范化的关系表示转换到交叉表的方法。

## 8.9 时态数据库建模 (Modeling Temporal Data)

- 一般来说, 时态数据(temporal data)是具有关联的时间段的数据, 在时间段之间数据有效(valid);
- 一个特定时间点上该数据的值称为数据的快照(snapshot);
- $X \rightarrow^t Y$  在某个特定时间点上成立的函数依赖称为时态函数依赖;

形式化的说, 时态函数依赖(temporal functional dependency)  $X \rightarrow^t Y$  在关系模式  $r(R)$  上成立的条件是, 对于  $r(R)$  的所有合法实例,  $r$  的所有快照都满足函数依赖  $X \rightarrow Y$

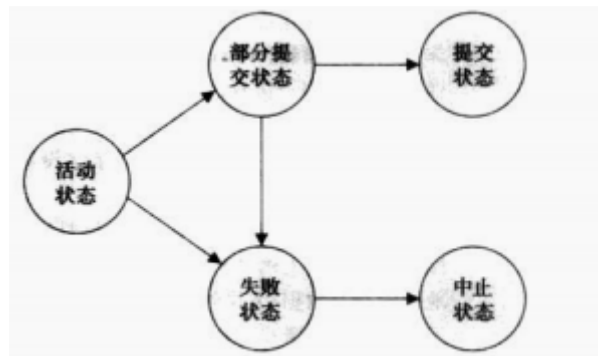
- In practice, database designers may add start and end time attributes to relations;
- 如果一个关系有一个参照时态关系的外码，数据库设计者必须决定参照是针对当前版本的数据还是一个特定时间点数据；
- 一个时态关系中原始的主码无法再唯一标识一条元组；

More detail in book;

## 第七章 事务

- 事务概念
  - A **transaction** is a unit of program execution that accesses and possibly updates various data items.
  - Two main issues to deal with:
    - Failures of various kinds, such as hardware failures and system crashes;
    - Concurrent execution of multiple transactions;
  - To preserve the integrity of data the database system must ensure ACID properties:
    - Atomicity (原子性):
      - Either all operations of the transaction are properly reflected in the database or none are.
      - The system should ensure that updates of a partially executed transaction are not reflected in the database
    - Consistency (一致性):
      - Execution of a transaction in isolation preserves the consistency of the database (隔离执行事务时保持数据库的一致性).
    - Isolation (隔离性):
      - Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
        - 也就是说，对于任何一对事务 $T_i$ 和 $T_j$ ，在 $T_i$ 看来， $T_j$ 或者在 $T_i$ 开始之前已经完成执行，或者在 $T_i$ 完成之后开始执行；
    - Durability (持久性):
      - After a transaction completes successfully, the changes it has made to the data-base persist, even if there are system failures.
- 事务状态
  - Active (活动的):
    - The initial state: the transaction stays in this state while it is executing;
  - Partially committed (部分提交)
    - After the final statement has been executed;
  - Failed (失败的):
    - After the discovery that normal execution can no longer proceed;
  - Aborted (中止的):
    - After the transaction has been rolled back and the database restored to its state prior to the start of the transaction;

- Two options after it has been aborted:
  - Restart the transaction (can be done only if no internal logical error);
  - Kill the transaction;
- Committed (提交的):
  - After successful completion
- 事务状态图:



- 并发执行
  - Multiple transactions are allowed to run concurrently in the system, advantages are:
    - Increased processor and disk utilization, leading to better transaction *throughput*
      - e.g. One transaction can be using the CPU while another is reading from or writing to the disk;
    - Reduced average response time for transactions:
      - short transactions need not wait behind long ones
  - **Concurrency control schemes** (并发控制机制)——Mechanisms to achieve isolation:
    - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
- Schedules
  - **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
    - A schedule for a set of transactions must consist of all instructions of those transactions (一组事务的一个调度必须包含这一组事务的全部指令);
    - Must preserve the order in which the instructions appear in each individual transaction (必须保持指令在各个事务中出现的顺序);
  - 串行的: 每个串行调度来自各事务的指令序列组成, 其中属于同一个事务的指令在调度中紧挨在一起;
  - 当数据库并发的执行多个事务时, 相应的调度不必是串行的;

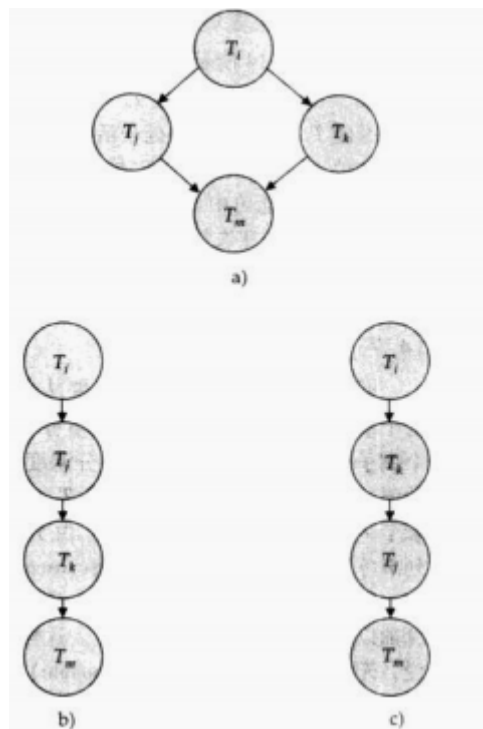
A transaction that successfully completes its execution will have a commit instructions as the last statement;

A transaction that fails to successfully complete its execution will have an abort instruction as the last statement;

- Serializability (可串行化)
  - **Basic Assumption** – Each transaction preserves database consistency.
  - Thus serial execution of a set of transactions preserves database consistency.
  - A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
    - conflict serializability



- view serializability\*
- *Simplified view of transactions:*
  - We ignore operations other than **read** and **write** instructions;
  - We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes;
  - Our simplified schedules consist of only **read** and **write** instructions.
- Conflicting Instructions:
  - Instructions  $I_i$  and  $I_j$  of transactions  $T_i$  and  $T_j$  respectively, **conflict** if and only if there exists some item  $Q$  accessed by both  $I_i$  and  $I_j$ , and at least one of these instructions wrote  $Q$ .
    - $I_i = \text{read}(Q), I_j = \text{read}(Q)$ : They don't conflict.
    - $I_i = \text{read}(Q), I_j = \text{write}(Q)$ : They conflict
    - $I_i = \text{write}(Q), I_j = \text{read}(Q)$ : They conflict
    - $I_i = \text{write}(Q), I_j = \text{write}(Q)$ : They conflict
  - Intuitively, a conflict between  $I_i$  and  $I_j$  forces a (logical) temporal order between them.
    - If  $I_i$  and  $I_j$  are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.
- Conflict Serializability:
  - **If a schedule  $S$  can be transformed into a schedule  $S'$  by a series of swaps of non con-flicting instructions, we say that  $S$  and  $S'$  are conflict equivalent**(如果调度 $S$ 可以经过一系列非冲突指令交换转换成 $S'$ , 我们称 $S$ 与 $S'$ 是冲突等价的).
  - We say that a schedule  $S$  is **conflict serializable** if it is conflict equivalent to a serial schedule.
- Other Notions of Serializability:
  - The schedule that produces same outcome as the serial schedule, yet is not conflict equivalent or view equivalent to it.
  - Determining such equivalence requires analysis of operations other than read and write.
- Test for Conflict Serializability:
  - Precedence graph (优先图):
    - 顶点集由所有参与调度的事务组成;
    - 边集由满足下列三个条件之一的边  $T_i \rightarrow T_j$  组成:
      - 在  $T_j$  执行  $\text{read}(Q)$  之前,  $T_i$  执行  $\text{write}(Q)$ ;
      - 在  $T_j$  执行  $\text{write}(Q)$  之前,  $T_i$  执行  $\text{read}(Q)$ ;
      - 在  $T_j$  执行  $\text{write}(Q)$  之前,  $T_i$  执行  $\text{write}(Q)$ ;
  - A schedule is conflict serializable if and only if its precedence graph is acyclic.
  - If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph:



- Recoverables Schedules

- **Recoverable schedule** — if a transaction  $T_j$  reads a data item previously written by a transaction  $T_i$ , then the commit operation of  $T_i$  appears before the commit operation of  $T_j$ .
  - Example: if  $T_9$  commits immediately after the read

$T_8$	$T_9$
read(A)	
write(A)	
	read(A)
read(B)	

- Cascadeless Schedules

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

$T_{10}$	$T_{11}$	$T_{12}$
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

If  $T_{10}$  fails,  $T_{11}$  and  $T_{12}$  must also be rolled back.

- Can lead to the undoing of a significant amount of work.
  - Cascadeless Schedules (cascading rollbacks cannot occur):
    - For each pair of transactions  $T_i$  and  $T_j$  such that  $T_j$  reads a data item previously written by  $T_i$ , the commit operation of  $T_i$  appears before the read operation of  $T_j$ .
    - Every cascadeless schedule is also recoverable.
    - It is desirable to restrict the schedules to those that are cascadeless
- Concurrency Control:

A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and recoverable and preferably cascadeless.

Testing a schedule for serializability *after* it has executed is a little too late!

- **Goal** – to develop concurrency control protocols that will assure serializability.
- Concurrency control protocols generally do not examine the precedence graph as it is being created:
  - Instead a protocol imposes a discipline that avoids nonserializable schedules;
- Transaction Definition in SQL
  - Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
  - In SQL, a transaction begins implicitly.
  - A transaction in SQL ends by:
    - **Commit work** commits current transaction and begins a new one.
    - **Rollback work** causes current transaction to abort.
  - In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully:
    - Implicit commit can be turned off by a database directive:
      - e.g In JDBC (`connect.setAutoCommit(false)`)