数据库复习笔记

张泽宇 2022012117

1. 介绍

1.1 基本概念

- **对象**（Entity）：数据库关心的客观对象
- **联系**（Relationship）：客观对象之间的联系
- **表**（Table）：对象与对象之间的联系，由表名及其属性来定义

1.2 数据库和文件系统

文件系统的缺点：访问数据难；数据冗余与不一致；数据孤立；数据一致性存在问题；更改原子性问题；多用户并发访问；安全问题

1.3 三维抽象与数据模型

- **模式**（Schema）：数据库的逻辑结构，例如二维表的表头
- **实例**（Instance）：数据库在某一时刻的实际内容，例如二维表的表内容

数据库的三维抽象：

- 物理模式（Physical level）：数据库存储数据的文件格式
- 逻辑模式（Logical level）：数据库中数据的存储和数据之间的关系
- 视图模式/应用模式：数据库向用户展示的视图

数据模型

- **Entity-Relationship model（E-R模型）**：概念设计模型
- **Relational model（关系模型）**：逻辑设计模型
  - 关系模型 = 关系 + 关系代数 + 约束

1.3.1 关系

- 域（Domain）：相同类型数据的集合
  - 空值（null）：每个域都具有的成员，代表未知或不存在
- 关系（Relation）：多个域之间笛卡尔积的子集，即 $r \subset (D_1 \times D_2 \times \cdots \times D_n)$
  - 一条关系是一个n元组，即 $r_t = (a_1, a_2, \ldots, a_n), \forall a_i \in D_i$

1.3.2 关系代数

符号	名称	意义	应用
σ	Selection	返回满足谓词的输入关系的行	$^{\sigma}salary >= 85000^{(instructor)}$
Π	Projection	从输入关系的所有行输出指定的属性。从输出中删除重复的元组。（输出集合）	$^{\Pi}ID, salary^{(instructor)}$
$\times$	Cartesian Product	来自两个输入关系的输出行对，这些行对在具有相同名称的所有属性上具有相同的值。（做笛卡尔积）	$instructor \times department$
$\cup$	Union	输出两个输入关系中的元组并集。	$^{\Pi}name^{(instructor)} \cup ^{\Pi}name^{(student)}$
$-$	Set Difference	输出两个输入关系中设置的元组差值。	$^{\Pi}name^{(instructor)} - ^{\Pi}name^{(student)}$
ρ	Rename	返回名称为 x 的表达式 E 的结果。	$\rho_X(E)$
$\bowtie$	Natural Join	来自两个输入关系的输出行对，这些行对在具有相同名称的所有属性上具有相同的值。	$instructor \bowtie department$

1.3.3 数据库语言

- 数据定义语言（DDL）：定义数据对象的语言，编译后生成存储在数据字典（data dictionary）中的一组表
- 数据操纵语言（DML）：即**查询语言（query language）**，用于获取和操作数据
- **结构化查询语言**（SQL）：DDL+DML，非程序性语言

1.4 数据库管理系统架构

- 存储管理器（Storage manager）：存储在数据库中的低级数据与提交给系统的应用程序和查询之间的接口
- 查询处理器（Query manager）：
  - DDL翻译器
  - DML编译器
  - 查询评估引擎（Query evaluation engine）

1.5 事务

- 事务是在数据库应用程序中执行单个逻辑功能的操作集合
- 事务的特征包括：**原子性**、**一致性**、**隔离性**、**持久性**
  - 例如，在转账的例子中，账户余额的加减和余额的读取显示是分开的事务

1.6 数据库用户和管理员

- 数据库用户
  - 数据库应用程序员（Application programmers）：通过 DML与系统交互
  - 数据库资深用户（Sophisticated users）：使用数据库查询语言组织请求
  - 数据库特别用户（Specialized users）：编写特殊数据库应用
  - 数据库最终用户（Naïve users）：调用之前编写的永久应用程序之一
- 数据库管理员（DBA）：负责数据库管理系统的管理与运营

2. 关系数据模型

2.1 关系数据模型*

- **元组**（tuple）：关系 r 的元素是一个元组，表示表中的一行
  - t[域名]表示t在该域名对应的属性上的值
  - t[数字]表示t第[数字]个属性上的值
- 关系是一组具有相同类型（相同结构）的元组，关系内部的元组是无序的
- **关系模式**（Relation Schema）：$R(\text{域名，域名，域名，}\ldots)$
- **关系实例**（Relation Instance）：$r(R)$代表使用R模式构造的元组
- **超键**（Superkey）：$K(\text{域名，域名，}\ldots) \subset R$，K 的值足以标识每个可能关系 $r(R)$ 的唯一元组（关系对象）
  - 超键可以同时存在不止一个
  - 最小的超键被称为**候选键**，候选键也可以同时存在不止一个
  - 可以选定一个候选键作为**主键**，主键包含的属性被称为主属性
- **外键**（Foreign Key）：存在$r_1(R_1), r_2(R_2)$，α是R_2的外键当且仅当：
  1. K_1是R_1的主键
  2. 对于$\forall t_2, \exists t_1[K_1] = t_2[\alpha]$
  - r_2是施引关系，r_1是被引关系
- 关系模式的集合是数据库模式，关系实例的集合是数据库实例

2.2 关系代数基础*

2.2.1 并运算 Union

$$r \cup s = \{t | t \in r \vee t \in s\}$$

- 输入具有**相同可比较属性**的关系，输出并集
- 例子：

r: (a,b)	s: (a,b)	$r \cup s$
(alice,1),(bob,2)	(alice,3)	(alice,1),(bob,2),(alice,3)

2.2.2 差运算 Set Difference

$$r - s = \{t | t \in r \vee t \notin s\}$$

- 输入具有**相同可比较属性**的关系，输出差集
- 例子：
-

r: (a,b)	s: (a,b)	$r - s$
(alice,1),(bob,2)	(alice,1)	(bob,2)

2.2.3 笛卡尔积运算 Cartesian-Product

$$r \times s = \{tq | t \in r \wedge q \in s\}$$

- 输入**属性不相交**的关系，输出它们对属性的笛卡尔积
- 例子：

r: (a,b)	s: (c,d)	$r \times s$: (a,b,c,d)
(alice,1),(bob,2)	(male,A)	(alice,1,male,A),(bob,2,male,A)

2.2.4 选择运算 Select

$$\sigma_p(r) = \{t | t \in r \wedge p(t)\}$$

- 输入关系，输出**满足选择谓词条件的子集**
- **选择谓词**：$p \rightarrow (OT)^+; O \rightarrow \wedge | \vee | \neg; T \rightarrow APA | APc; A \rightarrow attribute; P \rightarrow = | \neq | \leq | \geq | > | <$
- 例子

r:(A,B)	$p: A = B \wedge B > 5$	$\sigma_p(r)$
(1,2),(6,6),(2,3)		(6,6)

2.2.5 投影运算 Project

$$\Pi_{A_1,A_2,\ldots,A_k}(r) = \{t[A_1, \ldots, A_k] | t \in r\}$$

- 输入关系，将元组映射到**选定的更低维**，并输出一个集合
- 例子

r: (A,B,C)	$\Pi_{A,C}(r)$
(1,2,3),(4,5,6)	(1,3),(4,6)

2.2.6 换名运算 Rename

$$\rho_X(E)$$

- 把关系E重命名为X，属性名不变（创造一个副本）

2.2.7 交运算 Set intersection

$$r \cap s = \{t | t \in r \wedge t \in s\}$$

- 输入具有相同可比较属性的关系，输出交集
- 等价于$r-(r-s)$

2.2.8 自然连接运算 Natural-Join

$$r \bowtie s = \{t|_{pre} t \in r \wedge_{suf} t \in s\}$$

- 输入具有**公共/重叠/同名属性**的关系，求公共属性值相等的元组集合，输出消去公共属性的结果
- 例子

r: (A,B,C)	S: (B,C,D)	$r \bowtie s$： （A,B,C,D)
(1,2,3)	(2,3,4),(2,3,5)	(1,2,3,4),(1,2,3,5)

- 条件连接：$r \bowtie_C s = \sigma_C(r \times s)$，输出笛卡尔积的子集，即使有公共属性也不消去。

2.2.9 除法运算 Division

$$r \div s = t$$

使得

$$t \times s \subset r \wedge t(max)$$

实际上

$$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

-

r:(A,B)	s:(B)	$r \div s$
(A,1)(A,2)(B,1)(C,1)(C,2)(C,3)	(1),(2)	(A),(C)

2.2.10 赋值运算 Assignment

$$var \leftarrow operation$$

- 使用关系代数定义一个变量

2.2.11 泛化投影 Generalized Projection

$$\Pi_{F_1,F_2,\ldots,F_n}(E)$$

- F是任何运算，E是任何关系集合（包括常数和属性）
- 考虑如下的例子：已知员工的月薪，输出员工的名字和对应的年薪：
- $$\Pi_{name,(salary \times 12 \rightarrow annual)}(EMP)$$

2.2.12 聚集 Aggregate

$$_{G_1,G_2,\ldots G_n}\gamma_{F_1(A_1),F_2(A_2),\ldots,F_n(A_n)}(E)$$

- γ和g函数等价，表示在属性G的分组条件下，使用F(A)进行聚集，输出的属性集为$G \cup A$
- 可以使用换名运算（简略为as）给输出的新关系F(A)赋予名称
- 考虑如下的例子：按地区计算员工的总薪资，输出是(domain,d_salary)
- $$_{domain}\gamma_{SUM(salary)\ AS\ d\_salary}(EMP)$$

2.2.13 外连接 Outer Join

$$r \rtimes s = \{t|_{pre} t \in r\}$$

$$r \ltimes s = \{t|_{suf} t \in s\}$$

$$r \rtimes\ltimes s = \{t|_{pre} t \in r \vee_{suf} t \in s\}$$

- 对于无法匹配的属性使用NULL补齐
- NULL的代数性质：
  - `NULL = NULL`
  - `NULL`值所有代数运算的结果均为NULL
  - 与`NULL`的比较均返回`unknown`
    - `(unknown or true) = true; (unknown or OTHERS) = unknown`
    - `(unknown and false) = false; (unknown and OTHERS) = unknown`

2.3 元组关系演算

- 对t的谓词表达式操作
- 可以编写生成无限关系的元组演算表达式
- $\{t|P(t)\}$如果 t 的每个分量都出现在 P 中出现的关系、元组或常量之一中，则是安全的

2.4 域关系演算

- 对属性对应的集合（域）<attribute1,attribute2,...>的谓词表达式操作
- 表达式安全当且仅当
  - 表达式的元组中出现的所有值都是来自 $dom(P)$ 的值（即，这些值出现在 P 中或 P 中提到的关系的元组中）；
  - 对于每个形式为$\exists x(P_1(x)$ 的 “there exists” 子公式，当且仅当$P_1(x)$ 对于 $dom(P_1)$ 中的一个值 x 为真时，该子公式为真；
  - 对于每个形式为$\forall x(P_1(x)$的“for all”子公式，当且仅当$P_1(x)$ 对于 $dom(P_1)$ 中的所有值 x 为真时，该子公式为真；

3. 初级SQL

3.1 SQL数据定义语言

- SQL数据语言使用数据定义语言定义关系集合：$R(A, D, dom, F)$
  - R，关系名称
  - A，关系包含的属性值
  - D，域集合
  - dom，A与D的对应模式
  - F，约束
- SQL语言自带多种固有类型，包括
  - `char(n)` 定长字符串，`varchar(n)` 可变字符串，`int` 整数，`smallint` 小整数，`numeric(p,d)` 指定精度定点数，`real` 浮点数，`double precision` 双精度浮点数，`float(n)` n位精度浮点数
  - 每种类型都包含特殊值 `null`

3.1.1 基本模式定义

create table

```
create table r(A1 D1, A2, D2, ... , An Dn, <完整性约束1>, ... , <完整性约束k>);

create table teaches
(ID varchar(5) ,
course_id varchar (8),
sec_id varchar(8),
semester varchar(6),
year numeric(4,0),
primary key (ID, cours_id, sec_d, semester, year),
foreign key (course_id, sec_id, semester, year) references section,
foreign key (ID) references instructor);
```

- A：关系中的属性名
- D：对应属性A的域，指定A的类型以及可选的约束
- 完整性约束
  - `primary key(A)`：**主码约束**，表示属性构成关系的主码。 主码属性必须是非空且唯一的；
  - `foreign key(A) references s`：**外码约束**，表示 r 引用关系 s 的主码作为外码约束，即 r 中任意元组在A上的趋势必须对应于 s 中某个元组在主码属性上的取值；
  - `not null`：**非空约束**，表示该属性上不允许存在空值
- SQL禁止破坏完整性约束的任何数据库更新

drop table

```
drop table r
```

- 删除 r 中所有元组，并删除 r 的模式

```
delete table r
```

- 删除 r 中所有元组，但保留关系模式

alter table

```
alter table r add A D
```

- 在 r 中添加类型为D的属性A

```
alter table r drop A
```

- 在 r 中去掉属性A
- 很多数据库系统不支持去掉属性，因为表定义放在数据字典/表目录中

3.2 SQL数据操作语言：查询

3.2.1 SQL查询基本结构

$$\Pi_{A_1,A_2,\ldots,A_n}(\sigma_P(r_1 \times r_2 \times \cdots \times r_m))$$

- 关于**多值重关系代数**的定义见书P81N3-1

select (1)

```
select name from instructor
```

- 查询单个属性构成的关系（默认不去除重复）

```
select distinct name from instructor
```

- 查询单个属性构成的关系，指明去除重复

```
select all name from instructor
```

- 查询单个属性构成的关系，指明不去除重复

```
select * from instructor
```

- 查询所有属性构成的关系

```
select '437'
select '437' as FOO # 属性名为FOO，内容为437
select '437' from instructor # 单列表，内容为437，行数为instructor的元组数
```

- 获取常量构成的关系

```
select AF as B from instructor
```

- 对属性进行运算或重命名，F包括$+, -, *, /$

```
select A from instructor where P
```

- 查询单个属性构成的关系（附带条件），P是谓词表达式

select (n)

```
select A from r, t where P
```

- 首先使用from将关系连接成笛卡尔积，之后使用where选择出对应的元组，最后使用select实现对应的输出

3.2.2 附加的基本运算

as

```
old-name as new-name
```

- 放在select子句或from子句中，表示更换属性名

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.depr_name ='Biology';
```

- 像T和S一样被用来重命名关系的标识在SQL标准中被称作**相关名称**，但通常也被称作**表别名、相关变量、元组变量**

like / not like / similar to / escape

- sql的字符串操作是大小敏感的

```
where building like '%Watson%'
```

- 字符串操作
  - **upper(s)**表示将 s 转大写
  - **lower(s)**表示将 s 转小写
  - **trim(s)**表示去掉 s 末尾的空格
  - **%**表示匹配任意子串
  - _表示匹配任意单个字符

```
where building like '\%Watson%' escape '\'
```

- escape可以定义转义字符

```
where building not like 'Liujiao'
```

- not like表示like的补集
- similar to比like更强（可表示正则表达式），但部分sql不支持

select ... order by

```
select distinct dept_name, name
 from instructor
 order by dept_name, name
```

- order by表示按属性**升序**排列结果

```
select distinct dept_name, name
 from instructor
 order by dept_name desc, name asc
```

- desc 表示降序，asc 表示升序，主要属性在前，次要属性在后

where ... between

```
select name
from instructor
where salary between 90000 and 100000;
```

where ... ()

```
select name, course_id
 from instructor, teaches
 where (instructor.ID, dept_name) = (teaches.ID, 'Biology');
```

- sql支持按元组进行比较

is null

```
 select name
 from instructor
 where salary is null;
```

- 可以使用is null筛选null值

3.2.3 集合运算

union

```
(select course_id
from section
where semester='Fall'and year= 2017)
union
(select rourse_id
from section
where semester='Spring'and year= 2018) ;
```

- union子句取并集，自动去除重复
- **union all** 子句保留重复，重复元组数为$m+n$，即两关系中出现次数之和

intersect

```
(select course_id
from section
where semester ='Fall'and year= 2017)
intersect
(select course_id
from section
where semester='Spring' and year= 2018);
```

- intersect 子句取交集，自动去除重复
- **intersect** 子句保留重复，重复元组数为$min(m,n)$，即两关系出现的重复次数中较少的一个

except

```
(select course_id
from section
where semester='Fal l'and year= 2017)
except
(select course_id
from section
where semester='Spring'and year= 2018);
```

- except 子句将第一个输入对第二个输入取差集，自动去除重复
- **except all**子句保留重复，重复元组数为$m-n$，如果第一个输入重复次数少于第二个输入则取0

3.2.4 聚集函数

基本聚集

- **avg**：平均值
- **min**：最小值
- **max**：最大值
- **sum**：求和
- **count**：计数

```
select avg (salary)
 from instructor
 where dept_name= 'Comp. Sci.';
```

group by

```
select dept_name, avg (salary) as avg_salary
 from instructor
 group by dept_name;
```

- group by 允许按属性分组进行聚集，而不是聚集在一组里
- 被 select （在select子句中但不是聚集函数参数）的属性只能是**被聚集过的**

having

```
select dept_name, avg (salary)
 from instructor
 where salary > 80000
 group by dept_name
 having avg (salary) > 82000;
```

- having 允许按条件筛选分组（在where筛选元组之后）
- 被 having （在having子句中但不是聚集函数参数）的属性只能是**被聚集过的**

对null和boolean的采集

- avg/min/max/sum忽略null，对空集返回null
- count不忽略null，对空集返回0

3.2.5 子查询

in

```
select distinct course_id
from section
where semester='Fall'and year= 2017 and
course_id in (select cours_id
        from section
        where semester='Spring'and year= 2018);
```

- 在where子句中可以通过 in/not in进行嵌套子查询
- 或者使用枚举集合代替子查询的结果：
- ```
  select distinct name
  from instructor
  where name not in ('Mozart','Einstein');
  ```

some/all

```
select name
from instructor
where salary> some (select salary
            from instructor
            where dept...name ='Biology');
```

- 可以使用 some/all 进行集合比较
- some/all关系支持>|<|>=|<=|=|<>
- **=some** 和 in 等价，**<>all** 和 not in 等价
```

exists

```
select course_id
from section as S
where semester = 'Fall' and year = 2017 and
   exists (select *
      from section as T
      where semester = 'Spring' and year= 2018
      and S.course_id = T.course_id);
#  " 找出在2017 年秋季学期和 2018 年春季学期都开课的所有课程"
```

- exists / not exists 用于判断关系中是否存在非空元组
- 在上例中，来自外层查询的 S 是**相关名称**，使用 S 的内层子查询是**相关子查询**
- **not exists (B except A)**与$B \subset A$布尔值相同

unique

```
select T.course_id # 找出在2017年最少开设两次的所有课程"
 from course as T
 where unique (select R.course_id
         from section as R
         where T.course_id= R.course_id
         and R.year = 2017);
```

- unique 用于判断子查询的结果中是否存在重复元组（空集**unique为真**）

lateral

```
#找出系平均工资超过42000美元的那些系的教师平均工资
select dept_name, avg...salary
from (select dept_name, avg (salary)
   from instructor
   group by dept_name)
   as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

- from子句可以嵌套子查询（通常需要重命名子查询输出的关系）

```
# 在所有系中找出所有教师工资总额最大的系
select max (tot_salary)
 from (select dept_name, sum(salary)
   from instructor
   group by dept_name) as depUotal (dept_name, tot_salary);
```

- 一般 from 子句嵌套的子查询中不能使用来自同一from子句的其他关系的相关变量（即不允许 from A, select A...的形式）

```
# 打印每位教师的姓名，以及他们的工资和他们所在系的平均工资
select name, salary, avg_salary
from instructor I1, lateral (select avg(salary) as avg_salary
                from instructor I2
                where I2.dept_name= I1.dept_name);
```

- 如果要在 from 语句嵌套的子查询中访问来自外层查询的变量，使用 lateral 子句（只有最新的 sql 支持）

with

- with用于将子查询提前作为临时变量。
- ```
  # 找出具有最大预算值的系
  with max_budget (value) as
      (select max(budget)
      from department)
  select budget
  from department, max_budget
   where department.budget= max_budget.value;
  ```

with recursive （暂时略）

3.2.6 标量子查询

- SQL 允许子查询出现在返回单个值的表达式能够出现的任何地方，只要该子查询只返回一个包含单个属性的元组；这样的子查询称为**标量子查询** (scalar subquery)

```
# 列出所有的系以及每个系中的教师总数
select dept_name,
    (select count(*)
        from instructor
        where department.dept_name = instructor.dept_name)
    as num_instructors
 from department
```

- 在上面的例子中，如果使用 `group by`，结果会缺少没有教师的院系

3.3 SQL数据操作语言：修改

3.3.1 删除

delete from

```
delete from instructor
 where dept_name= 'Finance';
```

- delete 语句可以从关系中删除元组，可以使用 where 子句添加元组条件
- delete 语句**只能作用在一个关系上**
- delete 语句**不会删除关系本身**
- 在常见的 sql 实现（比如在 where 中使用子查询）中，删除不会影响后续行

insert into ... values

```
# 插入一个元组
 insert into course
 values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
 # 或者指定属性名
 insert into course (course_id, title, dept_name, credits)
 values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```
```

- 可以使用 values 子句插入一个元组
- 插入元组可以指定属性的排列顺序，属性对应的值可以指定为 null

```
insert into student
  select ID, name, dept_name, 0
  from instructor
```

- 使用 select 子句可以 insert 一个元组的集合
- 类似于 delete，在常见的 sql 实现中，插入不会影响后续的行（在上面的例子中，先 select 再 insert

update ... set ... (case)

```
update instructor
  set salary = salary * 1.03
  where salary > 100000;
```

- 使用 update set 子句改变关系中某个属性的值，可以使用 where 子句添加元组条件

```
update instructor
 set salary = case
        when salary <= 100000
        then salary * 1.05
         else salary * 1.03
         end
```

- 在不使用 case 子句的情况下，update 按先写先执行
- 在使用 case 子句的情况下，update 按 case 中的条件执行
- `set salary = ...` 后面的值显然是单值，因此可以用**标量子查询**替代
- 如果进行违规操作（比如，把主键约束的属性 set 成同一个值），update 会失败

3.4 数据库操作语言：连接

- **连接类型**：定义了在每个关系中，与基于连接条件的另一个关系中没有匹配的元组如何处理
- **连接条件**：定义了两个关系中哪些元组匹配，以及结果集中包含哪些属性

natural join

```
select name, course_jd
from student natural join takes;
# equals to
select name, course_id
from student, takes
where student.ID= takes .ID;
```

- 自然连接按以下顺序排列属性：**前后关系的公共属性**，**只在前关系中出现的属性**，**只在后关系中出现的属性**

inner join

```
loan  inner join borrower using(loan-number)
```

- join ... using 子句表示指定属性上的取值相匹配（而非所有公共属性的取值相匹配）

```
select count(distinct i1.id)
from instructor as i1 inner join instructor as i2 on
 i1.salary > i2.salary;
```

- join ... on 子句表示连接的谓词逻辑

outer join

```
select count(distinct i1.id)
from instructor as i1 left outer join instructor as i2
on i1.salary > i2.salary;
```

- outer join 的使用类似于外连接关系运算
- **natural left (outer) join**保留所有前关系的属性，对后关系无法对齐的属性值填 null
- **natural right (outer) join**保留所有后关系的属性，对前关系无法对齐的属性值填 null
- **natural full (outer) join**以上两个都保留（实际上在部分sql中就是并集）

on / where

- 在外连接中， on 子句过滤**参与连接的元组**，where 子句过滤**连接的结果**

```
        (LEFT) outer join
dept ──────┐
           │ON d.did = m.did
           │AND m.region = 'US'
           ▼
      中间结果 (含NULL行) ──► WHERE d.open_year >= 2000
                                ▲
    这里再过滤，NULL行若条件不满足就被扔掉

```

- 所以应该先写好 on 子句再写 where 子句。

4. 物理存储

> 跳到书第12章P405

4.1 概述

- 物理存储分为：
  - 高速缓存、主存、闪存
  - 磁盘、光学、磁带
- 或者分为：**易失性**，**非易失性**
- 物理存储的评价指标：**速度**、**成本**、**可靠性**
- 存储的层次结构是：缓存、主存、闪存、磁盘、光盘、磁带

- 存储介质的层级结构是：**主存储器，辅助存储器/在线存储器，三级存储器/离线存储器**
- 存储介质的特征：这不会考吧?

4.2 磁盘

- 结构：（略）
- 访问时间=寻道时间+旋转延迟时间+数据传输时间
- 调度优化：**电梯算法**

4.3 硬盘访问策略

- 从磁盘读取的块暂时存储在内存**缓冲区**中，以满足将来的请求。 缓冲是通过操作系统和数据库系统共同运作的。
- 块的读写过程中需要加锁 PIN/UNPIN
- 策略1：LRU-最近最少使用
- 策略2：MRU-最近最多使用

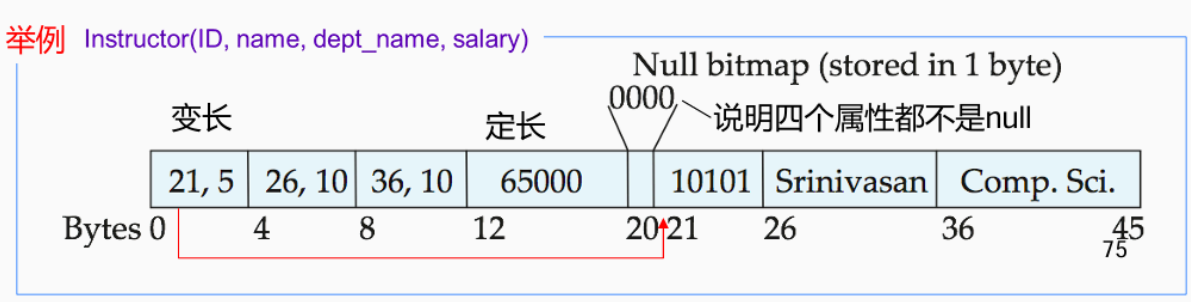
4.4 文件管理策略

- 一个数据库被映射为多个不同的文件，这些文件由底层的操作系统来维护。 这些文件永久地驻留在磁盘上。 一个**文件** (file) 在逻辑上被组织为记录的一个序列
- 每个文件还从逻辑上被分成定长的存储单元，称为**块** (block)
- 要求每条记录设备完全包含在单个块中

4.4.1 定长记录

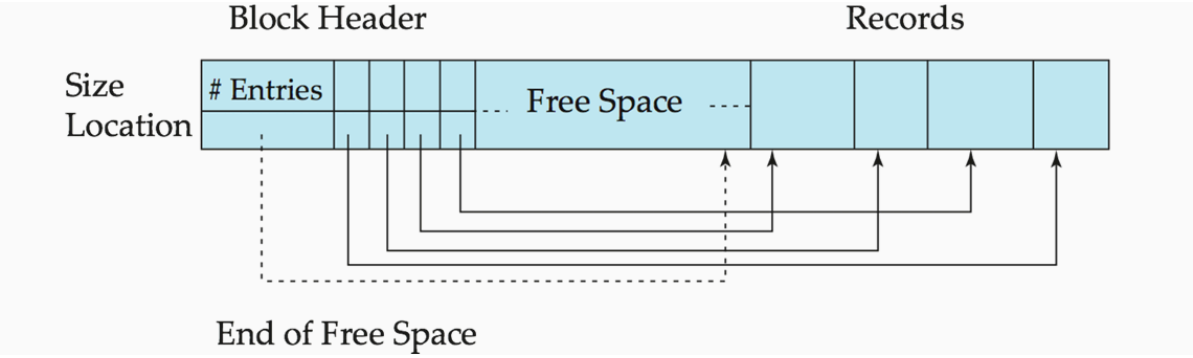
- 在一个块中只分配它能完整容纳的最多记录的数目（剩余字节留空）
- 由于插入通常比删除更频繁，因此让被删除的记录所占据的字节空着，直到有新的插入重新使用这个空间
- 为了让新的插入能够找到这个空间，分配特定数量的字节作为**文件头**，并且用**自由链表**的方式管理数据

4.4.2 变长记录和槽页*



- **变长属性信息**：记录`（偏移量，长度）`的字节对
- **定长属性内容**：储存定长属性（例如salary）的区域
- **空位图**：一组0,1构成的值，如果某个属性是 null 值，将对应位的0置为1
- **变长属性内容**：储存变长属性的区域
    - 对于大部分属性为null的数据，可以将空位图提前并适当省略变长属性信息

分槽的页结构 Slotted Page Structure



- **块头**（Block Header）
  - **项的数量**(Entries)：记录项的数量
  - **自由空间末尾**(End of Free Space)：块中的自由空间是连续的，位于块头数组的最后一项和第一条记录之间；对于新插入的记录，从块的尾部开始从后向前给记录分配数据
  - **项头数组**：一个由包含每条记录的`(位置，长度)`的项组成的数组
- **自由空间**：可以存放数据的空间
  - 如果一条记录被删除，它所占用的空间被释放，并且它的项被置为deleted
  - 块中（空间上）位于被删除记录之前（时间上是后插入）的记录将被移动，使得由删除而产生的自由空间能被重新使用
    - 由于块的大小有限，移动在多项式时间内进行
- **记录**：已经存放数据的空间

用定长单元表达变长记录

```
┌──────────────────── 一张 8 KB 页 ────────────────────┐
| Slot0 | Slot1 | Slot2 | Slot3 | ... | SlotN |
└──────────────────────────────────────────────────────┘
      ▲       ▲       ▲
      |       |       |
 记录A(2.5 KB)  |  记录B(7 KB 跨 2 页)
  占 Slot0~2    |
      └─────────┴──> Slot directory 告诉引擎:
                "B 从 Slot2 开始，长度跨 4 个 slot"
```

- 每个格子的偏移量 = *起始地址* + *格子大小* × *格子序号*；不需要在盘面做“深入计算”或线性扫描就能跳到指定格子
- **碎片控制** —— 当记录增长时，只需多占几个后续格子，而不是整体搬迁

最大长度记录

- 对每条记录“宁可浪费也不挪动”，一次性预留 **MaxLen** 字节位移窗

4.4.3 文件中记录的组织

- **堆文件组织**：任何记录可以放在文件系统中有空间的任何地方
- **顺序文件组织**：根据每条记录中“搜索码”的值顺序存储记录
- **散列文件组织**：在每条记录的某些属性上计算一个散列函数，散列函数的结果确定了记录应放到文件的哪个块中

顺序文件组织

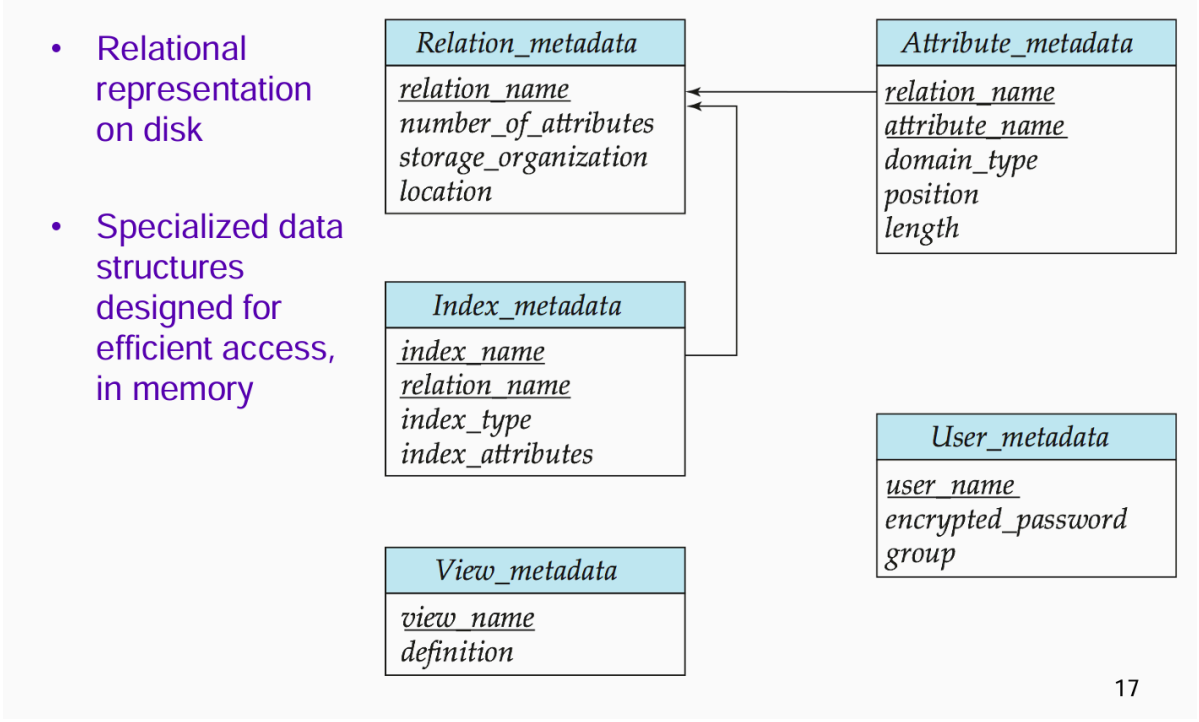
- **搜索码**：任意属性或属性集，只要有序，无需是主键或超键
- 按搜索码的顺序，或者按尽可能接近搜索码俄顺序物理存储记录
  - 删除：保留自由块并调整指针
  - 插入：如果这条记录所在块中有一条自由块，在自由块插入新的记录；否则将新纪录插入**溢出块**中
  - 如果溢出块过多，进行重组

多表聚集文件组织

- 使用**聚簇码**规定哪些数据被组织在一起
  - 例如，使用`dept_name`作为聚簇码，可以在`department`附近聚集同系的`instructor`
- 该文件组织适用于 join
- 该文件组织不适用于对主关系的普通查询

4.5 数据字典存储

- **元数据**包含关系的模式
- 关系模式和关于关系的其他元数据存储在一种称作**数据字典** (data dictionary) 或**系统目录** (system catalog) 的结构中
  - 关系的名称和关系中属性的名称
  - 属性的域和长度
  - 在数据库上定义的视图的名称和视图的含义
  - 完整性约束
  - 用户信息
  - 其他静态数据
  - 文件系统的存储信息



4.6 分布式文件系统（略）

5. 索引

5.1 概念

- 基本概念
  - **搜索码/查找键**：用于组织索引的属性或属性集
  - 索引文件：指向数据文件的指针型文件（比较小）
- 顺序索引：按照排好的顺序存储搜索码的值，并将每个搜索码与包含该搜索码的记录关联起来
- 散列索引：基于将值平均分布到若干桶中，一个值所属的桶是由散列函数决定的
- 指标：支持的访问类型、访问时间、空间开销

5.2 顺序索引

- **主索引/聚集索引**：搜索码次序与文件次序对应
  - 主索引的"主"不代表搜索码一定是主键，聚集索引的"聚集"和多表聚集没关系
- **辅助索引/次级索引**：搜索码指定的次序与文件的排列次序不同
- **索引项**：由`(搜索码，(标识，偏移量))`组成的指针型数据

5.2.1 （主）稠密索引

- 每个搜索码值都有对应的索引项
- 每个索引项都指向**具有这个搜索码值**的文件存储块头
- 具有相同搜索码值的其余记录会顺序地存储在第一条记录之后

5.2.2 （主）稀疏索引

- 只为某些搜索码值建立索引项
- 具有相同搜索码值的其余记录会顺序地存储在第一条记录之后
- 为了定位一条记录，我们找到所具有的最大搜索码值小于或等于我们所找记录的搜索码值的索引项

5.2.3 辅助索引

- 辅助索引必须是**稠密**的，对每个搜索码值都有一个索引项
- 辅助索引一般采用**非唯一搜索码**，例如第一个搜索码指向一个桶，该桶继而又包含指向文件的指针
- 代价：
  - 附加的间接指针层可能需要随机I/O操作
  - 桶会浪费空间

5.2.4 多级索引

- 在内层索引上建立稀疏的外层索引，称为多层索引。
  - 例如，对于稠密索引`1,...,100`，可以类似二叉树地按块建立外层索引`1,25,50,75`
  - 外层索引可以放入内存以增加效率

5.2.5 索引更新

插入

- 稠密索引
  - 如果该搜索码值并未出现在索引中，系统就在索引中适当的位置插入
  - 否则：
    - 如果索引项存储的是指向具有相同搜索码值的所有记录的指针，那么系统就在 索引项中增加一个指向新记录的指针
    - 否则，索引项存储一个仅指向具有相同搜索码值的第一条记录的指针，系统把待插入的记录放到具有相同搜索码值的其他记录之后（有空插空，没空溢出，索引不改）
- 稀疏索引
  - 如果系统创建了一个新的块，它会将出现在新块中的第一个搜索码值（按照搜索码的次序）插入索引中
  - 如果这条新插入的记录具有它所在块中的最小搜索码值，那么系统就更新指向该块的索引项；否则，系统对索引不做任何改动

删除

- 稠密索引
  - 删除记录后搜索值没有对应的记录，删索引项
  - 删除记录后搜索值还有对应的记录
    - 原来的指针有效，索引不变
    - 原来的指针失效，新指针指向剩下的第一条记录
- 稀疏索引
  - 删除记录对应的搜索值无对应的索引项，不改动
  - 删除记录对应的搜索值有对应的索引项
    - 删除记录后搜索值没有对应的记录，索引项调整至下一个搜索码
      - 下一个搜索码已有索引项，则将索引项删除
    - 删除记录后搜索值还有对应的记录
      - 原来的指针有效，索引不变
      - 原来的指针失效，新指针指向剩下的第一条记录

5.3 B+树

- **B+树索引** (B+-tree index) 结构是使用最广泛的、在数据插入和删除的情况下仍能保持 其执行效率的几种索引结构之一。

5.3.1 B+树的结构

- B+树索引采用平衡树 (balanced tree) 结构，其中从树根到树叶的每条路径的长度都是相同的
- 除根节点的每个非叶子节点有$\frac{n}{2}$到n个子节点；根节点有2到n个子节点
- B+树的节点按顺序存最多n个**节点指针**和最多n-1个**搜索码**

- 

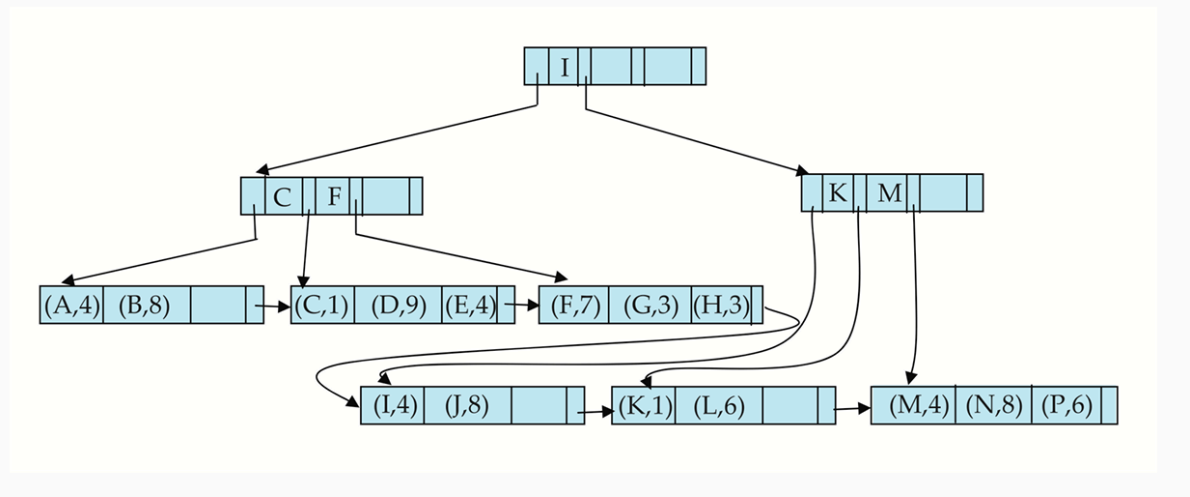
图 14-7　典型的 B^+ 树节点

- B+树的叶子结点结构是`指针1|搜索码1|指针2|搜索码2|...|下一个叶子结点的指针`
  - 叶子节点内部的指针不一定有序，叶节结点之间的指针有序（即左边叶子任意指针指向的值一定小于右边叶子任意指针指向的值）
  - 如果B+树索引被用作稠密索引（这是通常情况），每个搜索码值都必须出现在某个叶节点中
- B+树的非叶子节点结构是`子节点指针1|搜索码1|...|下一个同层节点指针`
  - 指针对应的子节点的搜索码一定小于指针后的搜索码
  - 指针对应的子节点的搜索码一定小于指针前的搜索码
  - 非叶子节点相当于对叶子节点的**稀疏索引**

- B+树的查询过程
  - ```
    root page (内存缓存)
    ↓ 二分/顺序查键区
    child page
    ↓
    ...
    ↓
    leaf page → 若为范围查询直接顺链下页
    ```
  - 查询过程
    - 从根节点开始
    - 找最小的比需要查询的搜索值大的搜索值对应的节点
      - 找到了，下滑
      - 当前节点所有搜索值都比查询的搜索值小，（叶子）右滑，（非叶）回滚
    - 遍历到叶节点或到了最右端，停止
- B+树的插入过程
  - 先类似查询地将搜索码插入叶节点
  - 如果叶节点未满，不变
  - 如果叶节点已满，二分将叶节点拆分为两个叶节点
    - 当拆分为新的节点时，新节点的头部搜索码将被添加到父节点中
    - 在非叶子节点被拆分时，如果有搜索码位于左新节点所有子树和右新节点所有子树的搜索码集之间，则将搜索码上提到父节点，作为新的两个节点的分割
    - 就沿着树向上递归处理， 直到一个插入不再产生拆分或创建了一个新的根节点为止
- B+树的删除过程
  - 先类似查询地将搜索码移出叶节点
  - 如果叶节点半满，不变
  - 如果叶节点少于半满，合并或重新分配搜索码
    - 如果可以和相邻的同父节点合并，合并
    ```

        - 否则在相邻节点之间重新分配搜索码
        - 最后将父节点中失效的搜索码删除
    - 就沿着树向上递归处理， 直到一个删除不再产生合并或消去了根节点为止
- 非唯一性搜索码（略）

5.3.2 B+树文件结构



- B+树的节点总是2/3满，因此在拆分和合并节点时尝试将搜索码在三个节点之间均匀分配

5.3.3 静态哈希索引

- **桶** (bucket) 表示可以存储一条或多条记录的存储单元， 对于内存中的散列索引，桶可以是索引项或记录的链表
- 使用溢出链的散列索引也称为**闭寻址** (closed addressing)
- 如果该桶没有足够的空间，就说发生了**桶溢出** (bucket overflow)，我们通过 使用**溢出桶** (overflow bucket) 来处理桶的溢出
- 一个给定桶的所有溢出桶都链接在一起并存放在链表中
- 桶值$B = (n_r/f_r) * (1 + d)$，d是负载因子（桶的实际容量比标准容量多出的比例）
- 如上所述的散列索引称为**静态散列** (static hashing)，在创建这种索引时，桶的数盐是固定的

5.3.4 可拓展哈希（暂略）

5.3.5 顺序索引和哈希索引的比较

- 预期的查询类型：
    - 哈希通常在检索具有特定键值的记录时表现更好
    - 如果范围查询很常见，则应优先考虑有序索引
- 在实践中：
    - PostgreSQL支持哈希索引，但由于性能差，鼓励谨慎使用
    - Oracle支持静态哈希组织，但不支持哈希索引
    - SQL Server仅支持B+树

5.3.6 位图索引

- 将向量变化值不多的属性压缩为位图：
- 例如，对于元组(0,M,A)(0,F,B)(0,F,A)(0,M,C)
  - 压缩后的位图是`M:1001`,`F:0110`,`A:1010`,`B:0100`,`C:0001`
  - 将位图相**交**'逻辑与'可以得到序号建立索引：
    - 例如查询(M,A)，`1001&1010=1000`，第1位是1，所以是第1个数据

5.3.7 在SQL中创建和删除索引

```
create index <index-name> on <relation-name>
(<attribute-list>)
# 例子
create index b-index on branch(branch_name)
```

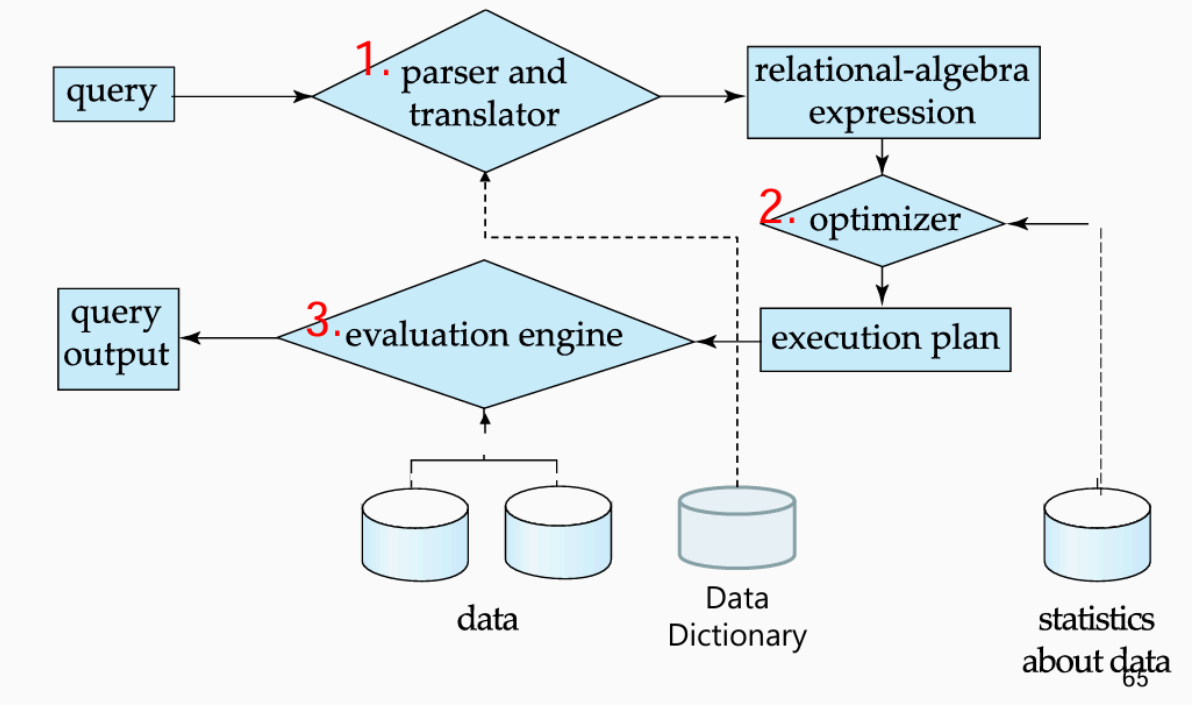
- 在<关系>的<属性>上创建一个<名字>索引

```
drop index <index-name>
```

- 删除<名字>索引（包括索引表）

6. 查询

6.1 查询的步骤



查询的基本步骤包括：

- 语法分析与翻译：将sql语言转化为合适的关系代数
- 优化：构造具有最小查询执行代价的查询执行计划应当是系统的责任
- 执行：**查询执行引擎** (query-execution engine) 接受一个查询执行计划

6.2 计算查询代价

- 查询执行的代价可以以不同资源的形式来进行度量，这些资源包括磁盘存取、执行一个查询所用的CPU时间，还有并行和分布式数据库系统中的通信代价
- 访问数据的代价主要包含以下方面：
  - **寻道代价**：平均搜索事件
  - **读代价**：从块中读数据的平均代价
  - **写代价**：向块中写数据的平均代价
- 简化计算：对于b个块的S次I/O操作：$b \times t_T + S \times t_s$
- 最坏代价：只有最小的可用缓存

6.3 选择运算

6.3.1 文件扫描

	算法	**代价 tr是tT，写错了**	**原因**
A1	线性搜索	$t_s + b_r \times t_r$	一次初始搜索+所有块传输（读取所有记录）
A1	线性搜索，码上的等值比较	$E = t_s + (\frac{b_r \times t_r}{2})$	一次初始搜索+所有块传输，记录期望位置是中间
A2	B+树聚集索引，码上的等值比较	$(h_i + 1) \times (t_r + t_s)$	（遍历树的高度+获取记录）*每个I/O操作需要一次寻道和一次块传输
A3	B+树聚集索引，非码上的等值比较	$(h_i) \times (t_r + t_s) + t_s + b \times t_r$	（遍历树的高度）*（一次寻道和一次块传输）+第一个块的一次寻道+读取所有记录
A4	B+树辅助索引，码上的等值比较	$(h_i + 1) \times (t_r + t_s)$	同A2
A4	B+树辅助索引，非码上的等值比较	$(h_i + n) \times (t_r + t_s)$	每条记录可能存储在不同的块上（次级索引和地址无关），因此需要对每条记录进行寻道
A5	B+树聚集索引，比较	$(h_i) \times (t_r + t_s) + t_s + b \times t_r$	同A3
A6	B+树辅助索引，比较	$(h_i + n) \times (t_r + t_s)$	同A4-2

- **码上的等值比较**：`WHERE emp_id = 1001`，主码查询，返回1行
  - 一定优先走主键索引，不会做全表扫描
- **非码上的等值比较**：`WHERE dept = 'Sales'`，非主码查询，期望是N和1之间的log/指数均值
  - 若该列有二级 B+ 树或位图索引，可用“值→行指针”集合直接取行，若无索引，只能 **顺序扫描** 整个文件，再在缓冲区做比较

- **比较**：`WHERE salary > 8000`，通常返回一个区间
  - 触发**范围扫描**

重要例题

> student为本科生表，ID为主键，有100000 个元组，占用了4000个磁盘块；在ID上建立B+树索引，其度数为100，1个B+树节点为一个磁盘块，查学号ID为202301001的本科生名字

1. 首先确定ID是主键，是码上的等值比较
2. $log_{100}(100000)$取整是3，相应地按块数算是2+1，树高是3
3. 调用A2

6.3.2 复杂选择

- 合取条件查找
  - A7 （使用一个索引的合取选择）：A2-A6最小的一个
  - A8 （使用组合索引的合取选择）：A2-A4最小的一个
  - A9 （使用标识交集的合取选择）：$\sum_{i=1}^{8} A_i$
- 析取条件查找
  - A10（使用标识并集的析取选择）：略

6.3.3 排序（略）

- 就是外归并排序
  - 创建n个有序的归并段
  - 对归并段进行N路归并
  - 内存用满，一次性合并M个归并段
- 代价：b_r代表包含关系r的块数，b_b代表一次寻道读出的块数，M是主存的缓冲区中可用于排序的块数
  - $$b_r(2log_{\lfloor M/b_b \rfloor -1}(b_r/M)+1)\text{块传输次数}$$
  - $$2\lceil b_r/M \rceil + \lceil b_r/b_b \rceil (2log_{\lfloor M/b_b \rfloor -1}(b_r/M)-1)\text{寻道次数}$$
  - $$b_r\text{写结果的块传输次数}$$

6.3.4 连接

嵌套-循环连接

- 在$n \bowtie_\theta s$中，r是**外层关系**，s是**内层关系**，所以r的循环包含s的循环
  - 需要检查所有的元组对，就是$t_r \times t_s$
  - 坏情况：缓冲区只能容纳每个关系的一个块
    - 总寻道次数是$n_r + b_r$，总块传输次数是$n_r \times b_s + b_r$
  - 好情况：缓冲区可以容纳内层关系的所有块
    - 总寻道次数是2，总块传输次数是$b_s + b_r$

块嵌套 - 循环连接

- 使用块作为循环边界，而不是使用整个关系作为循环边界
  - 坏情况：缓冲区只能容纳每个关系的一个块
    - 总寻道次数是$b_r + b_r$，总块传输次数是$b_r \times b_s + b_r$
  - 好情况：缓冲区可以容纳内层关系的所有块
    - 总寻道次数是2，总块传输次数是$b_s + b_r$
  - 一般情况：缓冲区容不下任何一个关系，但是可以容纳外层关系的M个块
    - 将M-2个块作为外层循环边界
    - 总寻道次数是$\lceil (b_r + b_r)/(M-2) \rceil$，总块传输次数是$\lceil b_r/(M-2) \rceil \times b_s + b_r$

索引嵌套循环连接

- 对于外层关系r中的每一个元组t_r, 可以利用索引来查找s中与元组t_r满足连接条件的元组
  - 坏情况：缓冲区只能容纳外层关系和索引的一个块
    - 对外层关系b，总寻道次数和总块传输次数都是b_r
    - 对索引，由于每个元组执行一次select，总代价是$n_r \times c$
    - 组合代价是$b_r(t_T + t_S) + n_r \times c$

归并-连接

- 对于已经排好序的元组，连接属性上的相同值是连续存放的
  - 为每个关系分配B个缓冲块，总寻道次数是$\lceil b_r/B \rceil + \lceil b_s/B \rceil$
  - 总块传输次数是$b_r + b_s$
- 如果元组没有排好序，先使用外归并算法排序

散列连接

- 使用哈希算法将r和s映射到若干个哈希分组r_i和s_i上，使用分组作为循环边界，前者称为**探查用输入**，后者称为**构造用输入**
- 如果分区太大，需要使用**递归分区**（对分区再进行分区）直到一个分区能够被内存完整读取
  - $M > \sqrt{(}b_s)$（即内存规模大于关系规模的开方）时无需递归分区
- 倾斜与溢出处理
  - 避让因子：一般增加20%的分区数量
  - **溢出分解**：如果分区s_i的桶溢出，使用新的散列函数对s_i和r_i进行分解，并在i上使用更小的分区进行构造
  - **溢出避免**：第一次分区就分成足够小的分区，对于过小的分区，如果内存空间允许就和其他分区合并
- 代价
  - 好情况：不需要递归分区
    - 总块传输次数，假设分区后每个关系多出n_n个没满的块
      - 分区的总块传输次数是$2(b_r + b_s) + 2n_n$
      - 分区后的构造和探查阶段，读取的总块传输次数是$b_r + b_S + 2n_n$
    - 类似，为每个关系分配B个缓冲块，总寻道次数是 $2(\lceil b_r/B \rceil + \lceil b_s/B \rceil) + 2n_n$
  - 坏情况：需要$\lceil log_{\lfloor M/b_b \rfloor - 1}(b_s/M) \rceil$次递归次数

- 总块传输次数，假设分区后每个关系多出n_n个没满的块
  - 分区的总块传输次数是$2(b_r+b_s)\left\lceil log_{\lfloor M/b_b\rfloor-1}(b_s/M)\right\rceil+2n_n$
  - 分区后的构造和探查阶段，读取的总块传输次数是$b_r+b_S+2n_n$
- 类似，为每个关系分配B个缓冲块，总寻道次数是
  $2(\lceil b_r/B\rceil+\lceil b_s/B\rceil)\left\lceil log_{\lfloor M/b_b\rfloor-1}(b_s/M)\right\rceil+2n_n$
- （一般此时可以忽略n_n）

6.3.5 其他运算

- 去重：归并排序时删除和相邻元组相同的元组；哈希是构造单射
- 集合运算：对两个关系的散列索引进行merge
- 投影：先直接截断，再对新关系的元组去重
- 外连接：考虑对排好序的关系进行归并连接，对无法匹配的序号置空。
  - 例如，对于有序的元组集合(A,1)(B,2)(C,4)和(1,a)(2,b)(3,c)，易得全外连接的新增元组是(null,3,c)和(C,4,null)，左右外连接同理
- 聚集函数：略

6.3.6 表达式执行

- 物化：使用**算子树**，自底向顶每次执行一个运算，结果作为临时变量参与后续运算
- 流水线：类似计组，需要看书记每个操作的对应实现

7. 查询优化 （完全看不懂）

- 查询优化的两个层次：**逻辑优化/代数优化**，**物理优化**
- 查询执行计划的产生涉及三个步骤：
  - （l） 产生逻辑上与给定表达式等价的表达式
  - （2）以可替代的方式对所产生的表达式做注释，以产生备选的查询计划
  - （3）估计每个执行计划的代价，并选择估计代价最小的那一个
- 代价估计的依据
  - **关系统计信息**：元组数量，属性的不同值数量
  - **中间结果统计**
  - **算法代价估计**

7.1 关系表达式的转换

7.1.1 等价规则

1. 级联运算规则：$\sigma_{\theta_1\wedge\theta_2}(E)=\sigma_{\theta_1}(\sigma_{\theta_2}(E))$
2. 选择运算交换律：$\sigma_{\theta_1}(\sigma_{\theta_2}(E))=\sigma_{\theta_2}(\sigma_{\theta_1}(E))$
3. 投影运算级联：$\Pi_{L_1}(\Pi(E))=\Pi_{L_1}(E)$
4. 连接和笛卡尔积变换：$\sigma_\theta(E_1\times E_2)=E_1\bowtie_\theta E_2$

   连接和笛卡尔积变换：$\sigma_{\theta_1}(E_1\bowtie_{\theta_2}E_2)=E_1\bowtie_{\theta_1\wedge\theta_2}E_2$
5. 条件连接交换律：$E_1\bowtie_\theta E_2=E_2\bowtie_\theta E_1$

6. 自然连接运算结合律：$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$

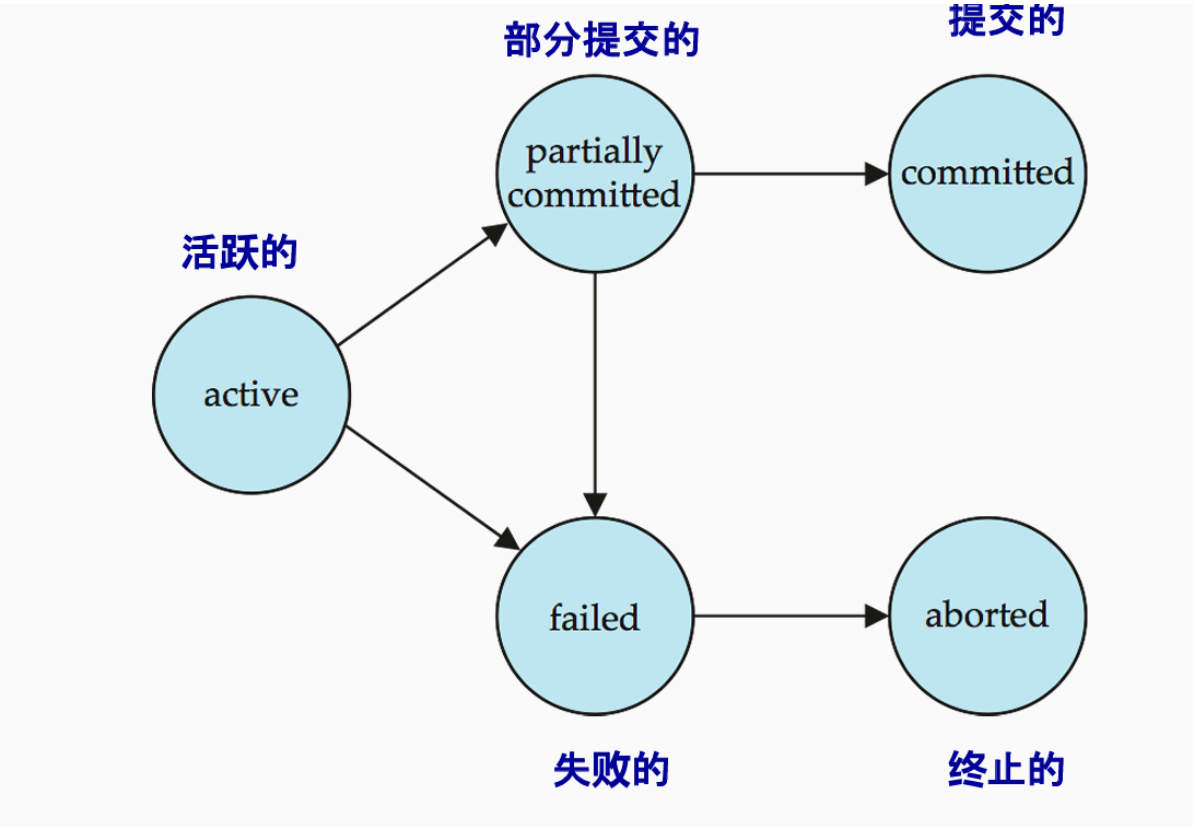
8. 事务管理

8.1 概念和存储器

- **事务**是访问并可能更新各种数据项的一个程序执行单元
  - **原子性**：要么执行其全部操作，要么就根本不执行
  - **一致性**：以隔离方式执行事务，以保证数据库的一致性
  - **隔离性**：每个事务都感觉不到系统中有其他事务在并发地执行（物理上可能并发执行）
  - **持久性**：在一个事务成功完成之后，它对数据库的改变必须是永久的，即使出现系统故障也是如此
    - 以上四种特性被称为事务的**酸性**（ACID）
- **稳定存储/可靠存储器**：可以将信息复制到几个非易失性存储器介质

8.2 事务的原子性：状态

- **活跃**(active)：初始状态，当事务执行时就处于这种状态
- **部分提交**(partially committed)：最后一条语句被执行
- **失效**(failed)：正常执行不能再继续
- **中止**(aborted) `ppt上是终止`：在事务**已回滚**并且数据库已被恢复到它在事务开始前的状态之后
  - **重启**：在事务中止后，重新启动一个同样的新事务
  - **杀死**：清除事务
- **提交**(committed)：在成功完成之后
  - **终止**：提交或者中止，即事务产生的影响已经稳定



8.3 事务的隔离性：并发执行

- 提高吞吐量和资源利用率：将CPU处理和磁盘I/O同时执行
- 减少等待时间：共享CPU周期和磁盘存取

8.3.1 调度

- **调度**：表示指令在系统中执行的时间顺序
-

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

图 17-4　调度 3：等价于调度 1 的一个并发调度

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

图 17-5　调度 4：一个导致不一致状态的并发调度

- 调度应该在某种意义上等价于一个串行调度。 这种调度被称为可串行化的 (serializable) 调度。

8.3.2 可串行化

冲突可串行化

- 冲突：将数据操作归化为`Write(Q)`和`Read(Q)`
  - `read,read`：互不冲突，互不影响
  - `read,write`：相互影响，次序重要
  - `write,read`：相互影响，次序重要
  - `write,write`：互不影响，但是次序会影响之后的read
- 如果I与J是由不同事务在相同数据项上执行的操作，并且其中至少有一条指令是 write 操作，那么我们说I与J是**冲突**的
- **转换**：指与某个串行调度产生相同的最终状态
- **冲突等价**：调度S可以经一系列非冲突指令的交换而转换成另一个调度S'
- **冲突可串行化**：一个调度和一个可串行调度冲突等价

优先图

- **优先图/先序图**：包含指令依赖的有向图，如果指令A先于B执行，则$A \rightarrow B$
- **拓扑排序**：对指令进行现行排序，可以得到可串行化的指令序列
  - 如果优先图中存在**环**，则不可串行化

8.3.3 可恢复性和级联回滚

- **可恢复调度**：对于事务I,J，如果J在读取了I写过的数据项，则I的commit应该出现在J的commit之前（即J不能再I之前终止）
- **级联回滚**：存在多级`读未提交写`的事务依赖，这样当最初写事务回滚时，所有依赖于此事务的读都需要回滚。 这种因单个事务失效而导致一系列事务回滚的现象称为级联回滚
- **无级联调度**：只`读已提交写`

8.4.4 事务的隔离级别

隔离级别	脏读	不可重复读	幻读
读未提交	√	√	√
读提交		√	√
可重复读			√
可串行化			

- **脏读**：读了未提交的数据：`A:write-B:read`
- **不可重复读**：同一行两次读到不一样的数据`B:read-A:write-A:commit-B:read`
- **幻读**：同一条件两次读到不一样的行`B:read-A:INSERT/DELETE-B:read`

8.4.5 隔离级别实现*

锁

时间戳

多版本和快照隔离

- 我们可以想象每个事务在它开始时有其自己的数据库版本或者快照，它从这个私有版本中读取数据
- 快照隔离的问题是它提供了太多的隔离
- 部分SQL使用快照隔离实现了**可串行化**隔离性级别，结果是它们的可串行化实现在特殊情况下会导致允许非可串行化的执行；SQL Server在标准级别以外增加了一个称为快照的附加的隔离性级别

9. 并发控制

9.1 基于锁的协议

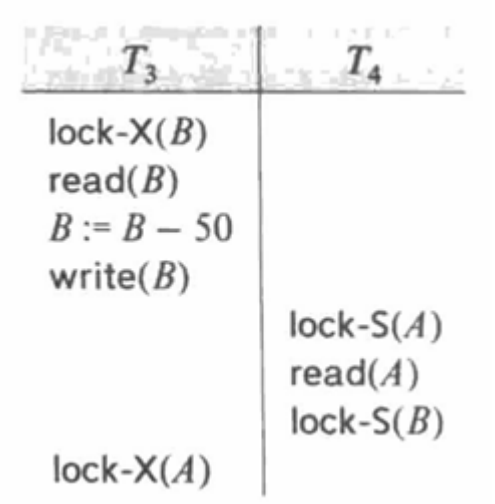
- 锁的模式
  - **共享模式锁**（S）：如果事务获得了数据项上的共享模式锁，**可读不可写**
  - **排他模式锁**（X）：如果事务获得了数据项上的排他模式锁，**可读可写**
  - 每个事务在操作数据项都要申请相应的锁
- 锁的权限：
  - 申请锁的权限使用**锁相容矩阵**确定，比如在S/X模式中锁相容矩阵是

-

	S	X
S	T	F
X	F	F

  - 如果事务申请一个锁，需要**等待**所有不相容的锁释放
  - 如果所有事务都在等待锁释放而不能正常执行，则进入**死锁**状态
  - 死锁状态必须对事务进行回滚
  - 合法的调度一定无环，无环的调度不一定合法

9.1.1 两阶段封锁协议

- 两阶段封锁协议
  - **增长阶段**：一个事务可以获得锁，但不能释放任何锁
  - **缩减阶段**：一个事务可以释放锁，但不能获得任何新锁
    - 封锁点：事务获得最后一个锁的时间点，增长阶段的结束（可排序）
- 两阶段封锁协议无法避免死锁
  - 

- 两阶段封锁协议无法避免**饥饿**
  - 考虑现在有一个S锁，等待队列是（S,X,S,S,S,S...），如果不加限制总会一直有一个S锁，即申请X锁的进程会**饿死**
- 两阶段协议不能避免级联回滚
  - 写事务在commit之前释放锁，其他事务就可以读未提交写
- **严格两阶段封锁协议**：X锁必须先提交再释放
- **严厉/强两阶段封锁协议**：所有锁必须先提交再释放
- 锁转换
  - 上升阶段可以锁升级
  - 下降阶段可以所降级
- S锁的实现

```
if Ti has a lock on D
 then
 read(D)
else begin
 if necessary wait until no other
transaction has a lock-X on D
 grant Ti a lock-S on D;
 read(D)
 end
```

- X锁的实现

```
 if Ti has a  lock-X on D
 then
 write(D)
 else begin
 if necessary wait until no other trans. has any lock on D,
 if Ti has a lock-S on D
 then
 upgrade lock on D to lock-X
 else
 grant Ti a lock-X on D
 write(D)
 end;
```

- 锁管理器
  - 它为目前已加锁的每个数据项维护一个记录的链表**锁表**，每一个请求对应链表中的一条记录、并按请求到达的顺序排序。
  - 锁表采用溢出链，采取先来先到原则，因此不会出现饥饿

9.2 死锁处理

- 处理死锁的两种方法
  - 死锁预防
  - 死锁检测与恢复

9.2.1 死锁预防

基于时间戳的死锁预防

- **等待 - 死亡 (wait-die) 机制**是一种非抢占技术：当事务I申请的数据项被J持有时，仅当I的时间戳小于J的时间戳才允许等待，否则I回滚（老等待，新回滚）
- **伤害 - 等待 (wound-wait) 机制**是一种抢占技术：当事务I申请的数据项被J持有时，仅当I的时间戳大于J的时间戳才允许等待，否则J回滚（新等待，老伤害）

基于锁超时的死锁预防

- 事务等待锁超时则立刻回滚
- 能够一定程度上解决死锁，无法解决饥饿

9.2.2 死锁检测

- **等待图**：描述锁依赖的有向图
  - 如果事务I等待事务J释放数据项上的锁，增加一条$J \rightarrow I$的边
  - 如果等待图上出现环，则出现了死锁
- 恢复策略：
  - 回滚将**产生最低代价**的事务（计算进度低，使用数据项少，还需要使用数据项多的，牵涉事务少的，**回滚次数少的**）
  - **回滚**：将事务只回滚到可以解除死锁的地方会更有效

9.3 多粒度*

- 数据库的粒度层次：database-domain-file-record
- 锁定上层意味着锁定下层的底层节点
- 在一个节点被加锁之前，所有祖先节点均要先加**意向锁**
  - **意向共享模式锁**（IS）：后代节点要加S锁
  - **意向排他模式锁**（IX）：后代节点要加X锁或S锁
  - **共享意向排他模式锁**（SIX）：自己和后代节点加S锁，后代节点要加X锁
- 锁相容矩阵为：
-

	IS	IX	S	SIX	X
IS	T	T	T	T	F
IX	T	T	F	F	F
S	T	F	T	F	F
SIX	T	F	F	F	F
X	F	F	F	F	F

9.4 插入操作、删除操作与谓词读（略

9.5 基于时间戳的协议

- 时间戳
  - 任务时间戳：$TS(T_i)$，任务在串行上的时间戳
  - 数据项时间戳：在执行读/写指令时更新的时间戳
    - 最后写时间戳：**W-timestamp(Q)**，成功执行`write(Q)`的**事务**的最大时间戳
    - 最后读时间戳：**R-timestamp(Q)**，成功执行`read(Q)`的事务的最大时间戳

9.5.1 时间戳排序协议*

- 时间戳排序协议
  - Ti发出read(Q)：
    - **写覆盖**：$TS(T_i) < W.timestamp(Q)$，有新的write操作覆盖了Q，Ti回滚

        - $TS(T_i) \geq W.timestamp(Q)$，正常读操作，更新最后读时间戳 $R.timestamp(Q) = MAX(R.timestamp(Q), TS(T_i))$
    - Ti发出write(Q)
        - **读冲突**：$TS(T_i) < R.timestamp(Q)$，有事务在读数据，回滚
        - **写冲突**：$TS(T_i) < W.timestamp(Q)$，有事务在写数据，回滚、
        - $TS(T_i) \geq W.timestamp(Q)$，正常写操作，更新最后写时间戳 $W.timestamp(Q) = MAX(W.timestamp(Q), TS(T_i))$
    - 如果事务被回滚，创建新的时间戳并重启
- Thomas 写规则
    - **写冲突**：$TS(T_i) < W.timestamp(Q)$，试图写入的数据已经过时，忽略该write操作（不回滚事务）

9.6 基于有效性检查的协议

- 事务执行的三个阶段：
    - 读阶段：准备数据但不更新数据库
    - **有效性检查阶段**：决定事务能否继续执行到写阶段
    - 写阶段：（非只读事务）写入数据库
- 三个时间戳
    - **开始时间戳**：$StartTS(T_i)$，开始执行的时间
    - **有效性检查时间戳**：$ValidationTS(T_i)$，开始有效性检查的时间，也是$TS(T_i)$
    - **完成时间戳**：$FinishTS(T_i)$，事务完成写操作的时间
- 有效性检查的条件
    - 对于$TS(T_k) < TS(T_i)$的所有事务Tk
        - $FinishTS(T_k) < StartTS(T_i)$
        - $FinishTS(T_k) < TS(T_i)$且Tk写的数据集合和Ti读的不相交
- 有效性检查的**乐观的并发控制**，因为事务最终能够完成执行且是有效的

9.7 基于多版本机制的协议（略）

- 对于数据项维护一个时间戳序列，记录不同的版本

10. 恢复

10.1 故障分类

- 事务故障
    - 逻辑错误：事务由于某些内部情况而无法继续其正常执行，这样的内部情况诸如非法输入、找不到数据、溢出或超出资源限制。
    - 系统错误：系统进入一种不良状态（如死锁），其结果是事务无法 继续其正常执行。 但该事务可以在之后的某个时间重新执行。
- 系统崩溃：硬件故障，或者是数据库软件或操作系统的漏洞，导致易失性存储器内容的丢失，并使得事务处理停止，但非易失性存储器内容完好无损且没被破坏。
- 磁盘故障：在数据传输操作中由千磁头损坏或故障造成磁盘块上的内容丢失。

解决方法

- 正常事务中存储数据以供后续恢复
- 故障发生后采取措施，将数据库内容恢复到某个保证数据库一致性、事务原子性及持久性的状态

10.2 储存

- 工作区到内存：read, write
- 内存到磁盘：input, output
- 缓存中放日志 buffer，磁盘中放日志 file

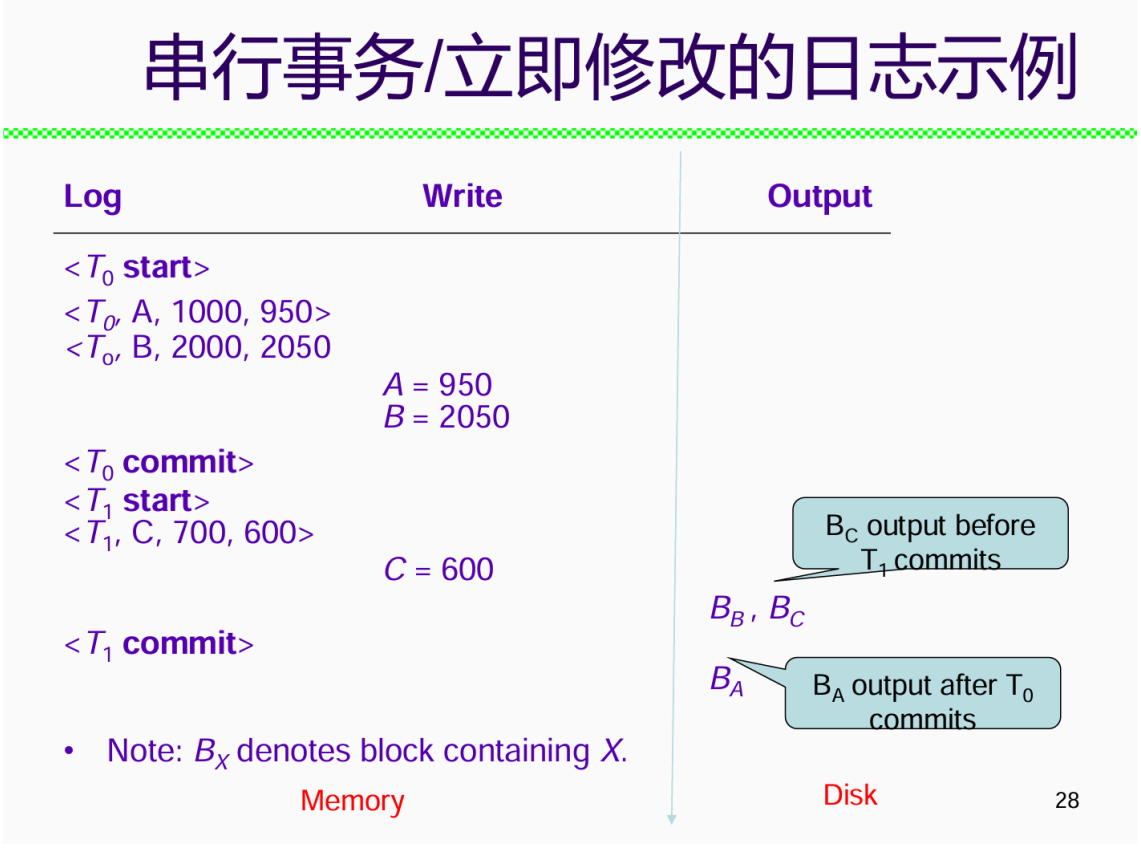
10.3 恢复和原子性

10.3.1 日志记录

- **事务日志记录**：`<T start>|<T commit>|<T abort>`
- **更新日志记录**：`<事务标识，数据项标识，旧值，新值>`（只管写，不管读）

10.3.2 数据库修改

- 数据库修改步骤
  - 事务在主存中私有部分进行计算
  - 事务修改主存的磁盘缓冲区中包含该数据项的块
  - 数据库系统执行output操作，将数据块写入磁盘
- **延迟修改**：事务直至提交后才修改数据库
  - 必须在主存保留计算出新值的拷贝
- **立即修改**：事务仍然活跃时就发生数据库修改
- 事务commit指commit日志被输出到稳定存储中
-



- 严格的两阶段封锁可以杜绝读未提交写的情况

10.3.3 基于日志的事务恢复逻辑

撤销逻辑 undo

- 对于有`<T start>`但没有`<T commit>`和`<T abort>`的事项，需要undo
- 从后往前将事务Ti更新过的所有数据项的值都恢复为旧值
- 使用 redo-only日志记录 `<事项，数据，要恢复的旧值>`记录回滚
- 结束后使用`<事项 abort>`日志，标明undo完成

重做逻辑 redo

- 对于有`<T start>`，也有`<T commit>`或`<T abort>`的事项，需要redo
- 从前往后将start 和 terminate之间所有的日志进行重做
  - 如果commit，正常重做
  - 如果abort，重做undo

检查点逻辑 checkpoint

- 在检查时间，将当前位于主存的所有日志记录输出到稳定存储器，将修改过的缓冲块输出到磁盘，将活跃事务列表`<checkpoint L>`输出到稳定存储器
- 崩溃之后，找到最后一条checkpoint，将L中的事务以及所有后面开始的事务记为事务集L，只对L进行undo和redo操作
- 删除之前所有已经commit的事务的日志

10.3.4 恢复算法

正常回滚

- 从后向前扫描，对于每一条`<事务标识，数据项标识，旧值，新值>`
  - 写入旧值
  - 输出日志 `<事项，数据，要恢复的旧值>`**补偿日志记录**
- 直到发现`<T start>`，停止，输出`<T abort>`

系统崩溃恢复

- 重做阶段
  - 找最后一个检查点，获取事务列表L
  - 试图重做任务列表L，并确定要插销的任务
    - 对`<事项，数据，要恢复的旧值>`和`<事项，数据，要恢复的旧值>`进行重做
    - 遇到`<T start>`将T放入undo-list
    - 遇到`<T commit>`或`<T abort>`将T取出undo-list
- 撤销阶段
  - 对undo-list中的事务从后向前进行undo操作
  - 发现`<T start>`，停止，输出`<T abort>`，将T移出undo-list
  - 直到undo-list空为止

10.3.5 缓冲管理（略

- **先写日志规则**（WAL）：`输出T其他日志->输出<T commit>->事务进入提交状态，数据输出到稳定存储器`
- **数据库缓冲**：略

10.3.6 磁盘数据丢失（略

- 基本的机制是周期性地将整个数据库的内容**转储** (dump) 到稳定存储器中

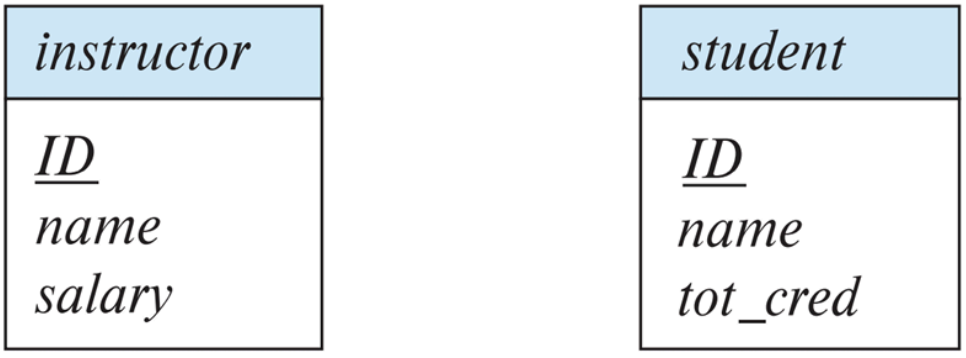
10.3.7 远程备份系统（略

11. 实体联系模型*

11.1 实体联系模型

- ER模型的组成部分
  - **实体集**（entity sets）
  - **联系集**（relationship sets）
  - **属性**（attributes）

11.1.1 实体集

- **实体**：现实世界中可区别于所有其他对象的一个“事物”或“对象”
  - 每个实体有一组性质，性质的值必须唯一标识一个实体
  - 实体通过一组**属性** (attribute) 来表示，每个实体在每个属性上都有**值**
    - 简单的属性是不能划分为子部分的属性
- **实体集**：共享相同性质或属性的、具有相同类型的实体的集合
  - 实体集在E-R图中使用方形表示
  - 

11.1.2 联系集

- **联系**：多个实体间的相互关联，使用**联系实例**表示
- **联系集**：相同类型联系的集合，设E是实体，联系集R是：
  - $$R \subset \{(e_1, e_2, \ldots, e_n) | e_i \in E_i, i = 1, \ldots, n\}$$
  - 出现的实体E参与联系R
  - 实体在联系中扮演的功能称为实体的**角色**
  - 联系集在E-R图中用菱形表示，角色在菱形和矩形之间的线上标注

- 联系可以具有被称作**描述性属性**的属性，描述性属性使用未分割的矩形描述

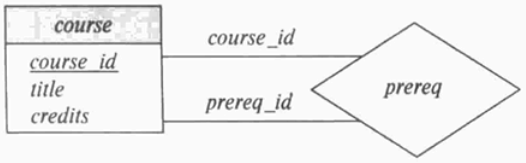


图 6-4　带有角色标识的 E-R 图

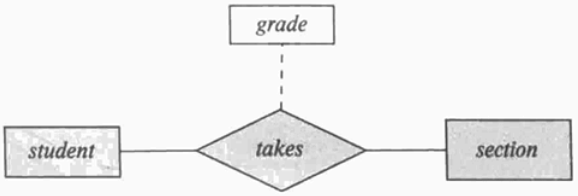


图 6-5　将一个属性附加到联系集的 E-R 图

- 参与联系集的实体数量被称为联系集的度

11.2 属性

- **域/值集**：属性取值的集合
- 简单属性：不能划分为子部分的属性
- 复杂属性：可以被划分为子部分（子部分还可以被继续划分）的属性

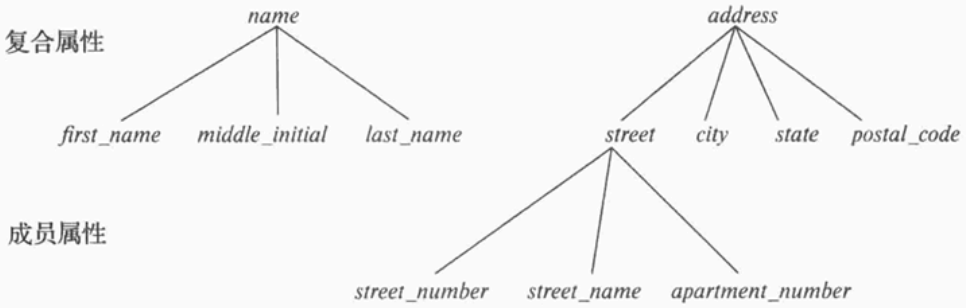


图 6-7　教师的 *name* 和 *address* 复合属性

- 单值属性：每个实体只能具有一个的属性，如ID
- 多值属性：每个实体可以具有多个的属性，如电话号码
- **派生属性**：可以被存储的其它属性计算出来的属性，如年龄=今年-出生年

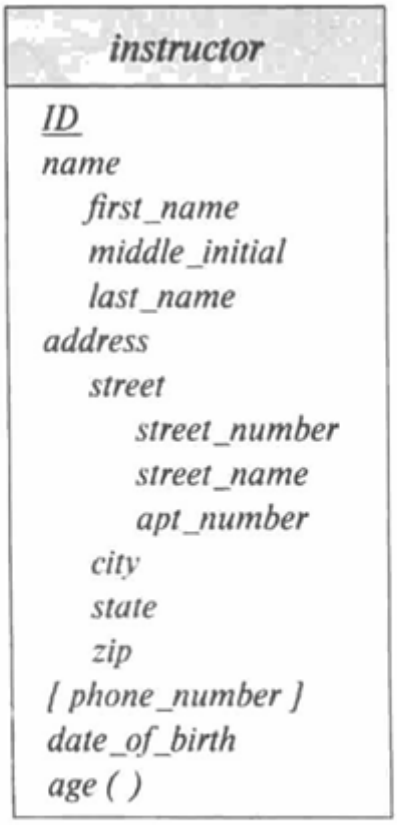


图 6-8　包含复合、多值和派生属性的 E-R 图

11.3 映射基数

映射约束

对于二元关系R来说，映射只有以下四种：**一对一**、**一对多**、**多对一**、**多对多**

a) 一对一

b) 一对多

c) 多对一

d) 多对多

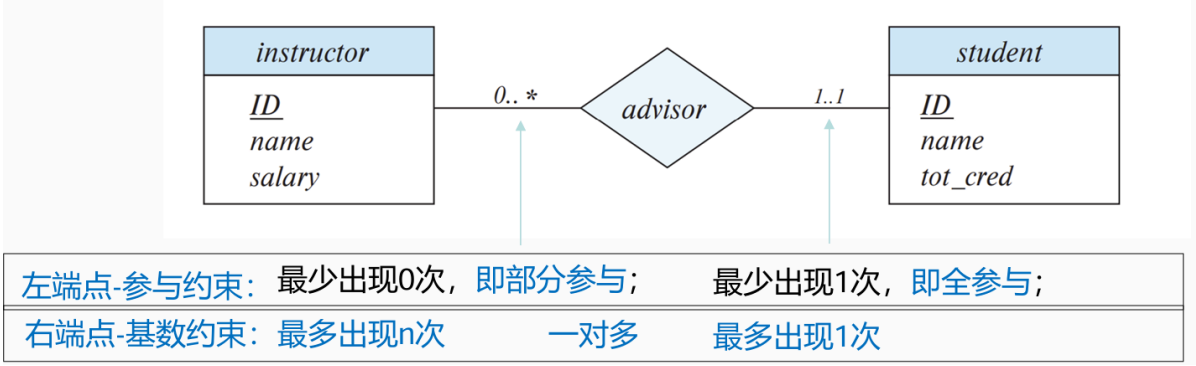
图 6-11　联系基数

参与约束

参与只有以下两种：**全部参与**，**部分参与**

在上面的例子中，学生一定有老师（双线，全部参与），老师不一定有学生（单线，部分参与）

统一约束



- instructor可以对应(0...n)位学生，可以取0表示部分参与，可以取n表示一对多
- student只能对应1位老师，不能取0表示全部参与，只能取1表示多对一

11.4 主码

- 实体集中的主码类似于数据库中的观点

11.4.1 联系集中的主码

- 联系集R中单独的联系：参与联系的实体集主码的并集 ∪ 联系关联的属性集
- 联系集的主码以最小的能区分联系的集合为准：
  - 多对多：关系的**主码的并集**
  - 一对多、多对一：**多方关系的主码**
  - 一对一：**任意一方关系的主码**
  - 非二元联系：使用**函数依赖**允许解释，然后对多方关系的主码做并操作

11.4.2 弱实体集

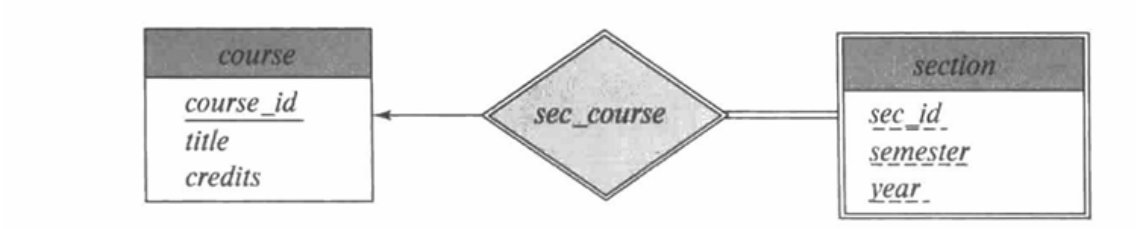
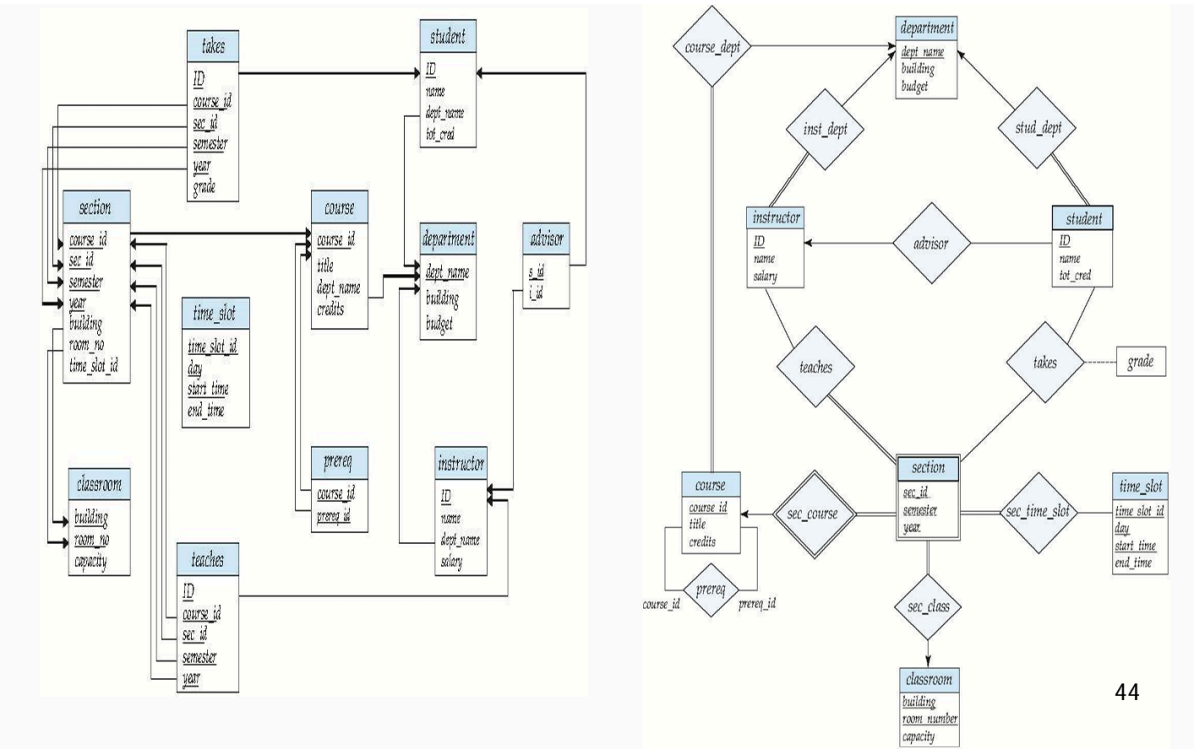
- 考虑课程course(course_id, title, credits)和课程段section(course_id,title,credits)，显然课程段必须和一个课程构建多对一关联，因此可以删去section中的course_id属性，改用联系集sec_course构建关联
- 

图 6-14　具有弱实体集的 E-R 图

- course称为**标识性实体集**，section称为**弱实体集**（双矩形）
- 弱实体集的主码由标识性实体集的主码和**分辨符属性**（下划虚线）并成
- sec_course由于包含依赖关系，成为**标识性联系**（双菱形）
- 通常，弱实体集必须全部参与其标识性联系集，并且该联系是到标识性实体集的多对一联系，因此双矩形和双菱形一般用双线连接
- 弱实体集可以与多个标识性实体集和其他强实体集建立联系

11.4.3 删除冗余属性

好的E-R图不包含冗余属性或者外码，而是全部使用联系来关联实体



关系可以转化为实体，也可以转化为联系。

11.4.4 E-R图转化为关系模式

强实体集

具有简单属性的强实体集可以直接转化为关系

具有复杂属性的强实体集

- 对于复杂属性，展开到最底层，全部作为单独的属性
- 对于多值属性，构造成形如(原实体集的外码约束 ， 多值)的关系
    - 多值的复杂属性也可以构造单独的关系，但需要按实际分配主码

弱实体集

- 弱实体集转化的关系的属性集是弱实体集属性 ∪ 标识性实体集主码
    - 该关系需要在强实体集转化的关系上建立外码约束

一般联系集

- 联系集转化的关系的属性集是联系集关联属性 ∪ 所有参与实体集主码
    - 联系集的主码属性继承到关系模式的主码属性，一般需要重命名
- 参与的实体集转化成的关系需要在该关系上建立外码约束

标识性联系集

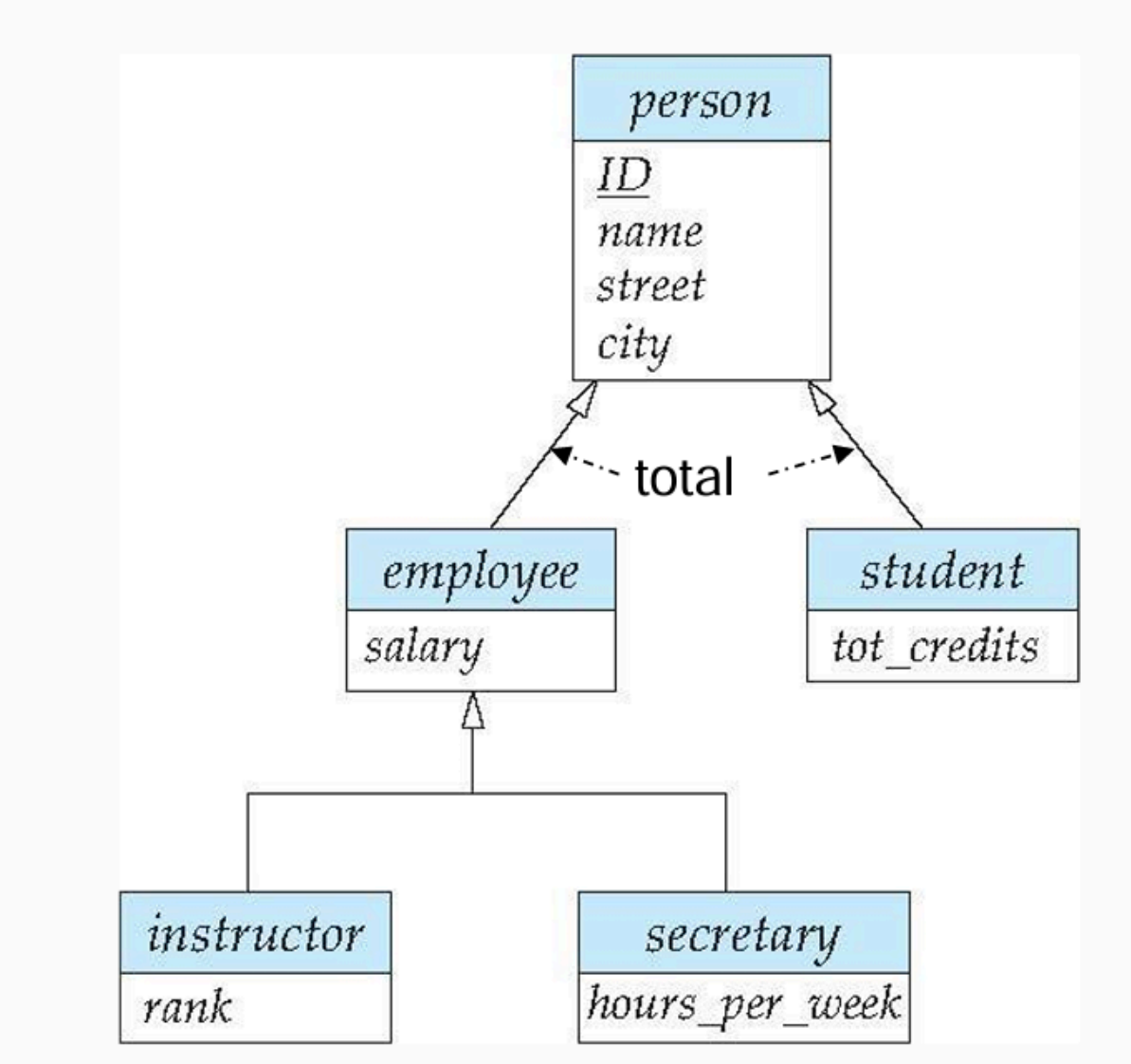
- 一般没用

一对一联系集

- 和任意一个参与联系的实体集进行合并

11.5 高级E-R逻辑

11.5.1 特化/精化和概化/泛化



- **超类**和**子类**的关系类似于类的继承，用空心箭头表示
- 子类从超类中隐式**继承属性**
- 完全性约束：
  - **全部特化/泛化**：实体全部属于实体集的子类（person要么是employee要么是student）
  - **部分特化/泛化**：实体不全部属于实体集的子类

转化为关系模式

- 可以垂直分割

schema	attributes
person	ID, name, street, city
student	ID, tot_cred
employee	ID, salary

- 不相交且完全，可以水平展开
schema	attributes
person	ID, name, street, city
student	ID, name, street, city, tot_cred
employee	ID, name, street, city, salary
    - person留着是为了产生外码约束

11.5.2 聚集

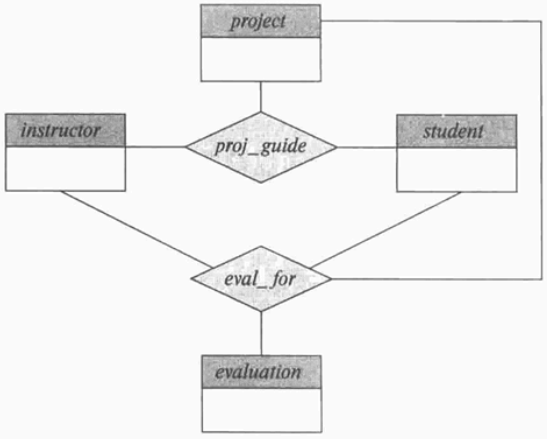


图 6-19　具有冗余联系的 E-R 图

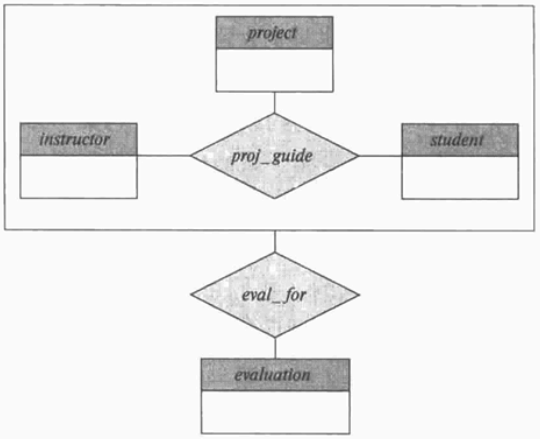
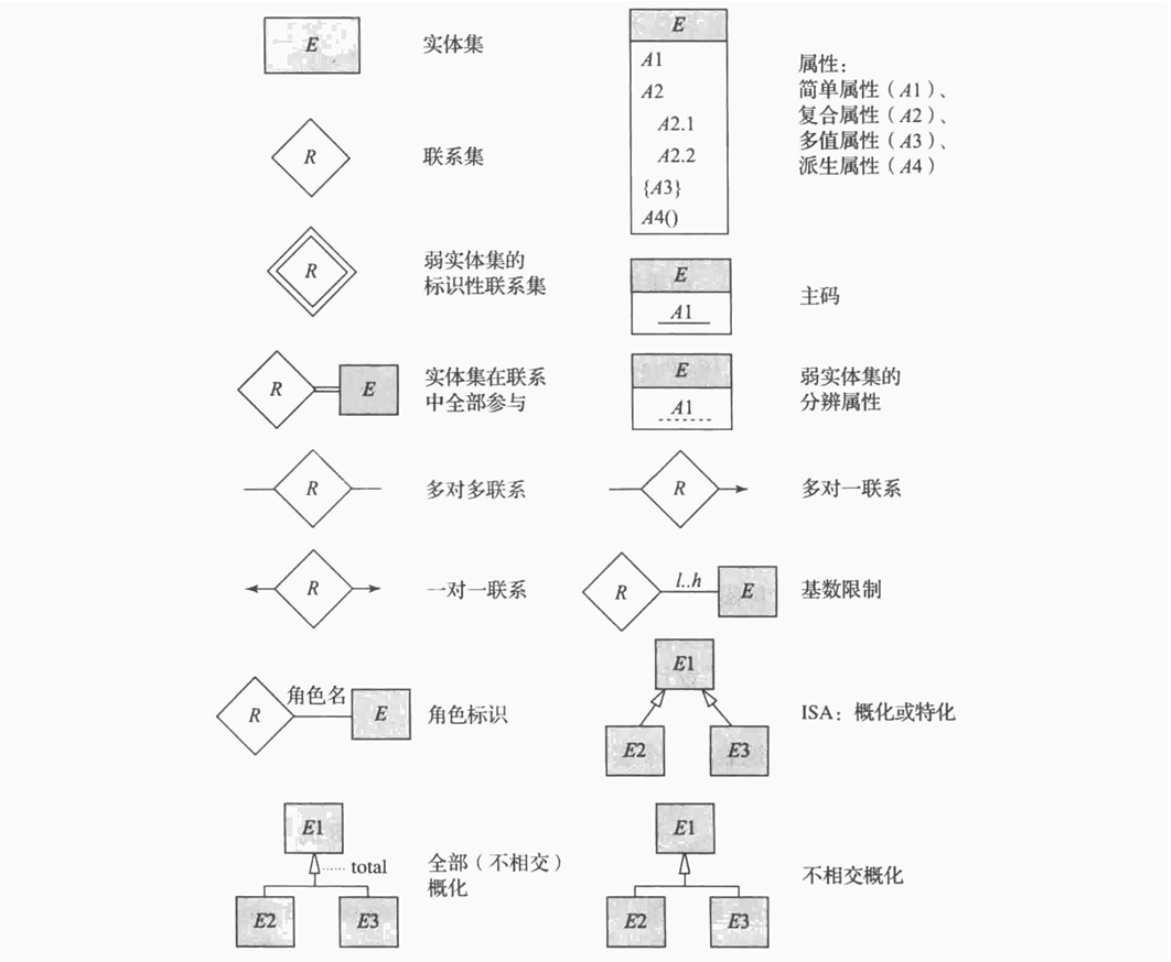


图 6-20　具有聚集的 E-R 图

- 可以把联系集看做高层实体集
- 转化为关系模式直接转大联系集

11.6 E-R总表



12 函数依赖

- **超码**（K）：在关系 r(R) 的任意合法实例中，不存在两个元组在属性集K上具有相同的值
- **函数依赖**：考虑关系模式R，令$\alpha \subset R,\beta \subset R$
  - 若对于r(R)实例中的所有元组t_1,t_2，有$t_1[\alpha]=t_2[\alpha] \to t_1[\beta]=t_2[\beta]$，那么该实例满足函数依赖
  - 如果r(R)的每个合法实例都满足依赖$\alpha \to \beta$，则该依赖在r(R)上成立
  - 超码K存在函数依赖$K \to R$
- 假设数据库属性名含义唯一，则数据库上$\alpha \to \beta$成立，任何模式$$$$\alpha \to \beta$$也成立
- 给定关系r(R)上函数依赖集F，可以推导出函数依赖的**闭包**F^+
- **无损分解**：如果$R_1 \cap R_2 \to R_1 \vee R_1 \cap R_2 \to R_2$，即分解的公共属性是其中一个分解的超码，则分解是无损的

12.1 F的闭包

- 阿姆斯特公理
  - $$\beta \subseteq \alpha \Longrightarrow \alpha \to \beta$$
  - $$\alpha \to \beta \Longrightarrow \gamma\alpha \to \gamma\beta$$
  - $$\alpha \to \beta \wedge \beta \to \gamma \Longrightarrow \alpha \to \gamma$$

    可以推出

- $$\alpha \to \beta \wedge \alpha \to \gamma \Longleftrightarrow \alpha \to \beta\gamma$$
- $$\alpha \to \beta \wedge \beta\rho \to \gamma \Longrightarrow \alpha\rho \to \gamma$$

- 属性集的闭包
  - 如果在F的闭包中能够推出$\alpha \to \beta$，则$\beta \subseteq \alpha^+$
  - 属性集的闭包有如下性质
    - $\alpha^+ = R \Longleftrightarrow \alpha$是超码
    - $\alpha \to \beta \Longleftrightarrow \beta \subseteq \alpha^+$
    - 知道属性集的闭包可以反向求函数集的闭包
- 冗余属性
  - 左冗余：形如 $A \to C, AB \to C$ 中的B
  - 右冗余：形如 $A \to C, AB \to CD$ 中的D
- **正则覆盖/标准覆盖**F_c
  - F_c中任何函数依赖无无关/冗余属性
  - F_c中所有函数依赖的左侧唯一
  - 正则覆盖未必是唯一的

12.2 BCNF分解：拆依赖*

- 核心规则：在 BCNF 中，**任何非平凡依赖** $X \to Y$ 必须满足 X 是候选码；否则就违反
- 算法：
  - **找违反 BCNF 的依赖**：某个 $X \to Y$, X 不是超码
  - **分裂**
    - $R_1 = X \cup Y$ （含这条依赖用到的全部属性）
    - $R_2 = R - (Y - X)$（剩余属性 + X）
  - 用原来的依赖集合 F 投影到 R_1, R_2 上；对每个子表 **递归** 步骤 1–3，直到所有子表都满足 BCNF
- 实例 P7.21

> 假设我们有一个模式$R = (A, B, C, D, E)$，如下函数依赖集成立：
>
> - $A \to BC$
> - $CD \to E$
> - $B \to D$
> - $E \to A$
>
> 请给出模式R的一个无损的BCNF分解

1. 显然B不是候选码（B推不出E），$B \to D$不满足BCNF，分解成(B,D)(A,B,C,E)
2. 对剩下的逐一检查，$B \to BD$成立，
   $A \to BC, BC \to DC, CD \to E, A \to ABCE, E \to A, E \to ABCE$，1,4式成立，2式已经丢失，所以分解到此为止。

12.3 3NF分解：保依赖*

- 核心：在 3NF 中，若 $X \to Y$ 非平凡，则满足 X **是超码** *或* Y **属于某个候选码。**
- 算法：
  - **求最小依赖集** G
    - 把右边多属性拆单独依赖
    - 去掉多余左边属性
    - 去掉冗余依赖
  - **每条依赖建一张表**
    对 $X \to Y$，生成 $R_i = X \cup Y$
  - **消除包含**
    若某 $R_j \subseteq R_i$，删除 R_j
  - **补主键**
    若所有 R_i 的并集还**缺少**某个原表候选码属性 → 再加一张只含该候选码的表

> 假设我们有一个模式$R = (A, B, C, D, E)$，如下函数依赖集成立：
>
> - $A \to BC$
> - $CD \to E$
> - $B \to D$
> - $E \to A$
>
> 请给出模式R的一个无损并保持依赖的3NF分解

1. 算得没有冗余属性，对每个函数依赖打表得到(A,B,C)(C,D,E)(B,D)(A,E)
2. 并集已经包含所有属性，结束

13. 高级SQL

13.1 视图

create view

```
# 定义视图
create view Perryridge-branch (p_branch_name, p_assets) as
    select branch-name, assets
    from branch
    where branch-city = Perryridge'
```

- 视图可以插入元组所在的任何地方
- 随实际关系变化而实时变化的试图称为**物化视图**
- 视图更新的条件
  - from子句中只有一个关系
  - select子句中只包含关系的属性名
  - 没有出现在select子句中的其他属性都可取null值
  - 查询无 group by或having子句

13.2 事务

commit work/rollback work

- 如其名

begin atomic ... end

- 将内部语句包裹为一个事务

13.3 完整性约束

13.3.1 单表约束

not null

```
branch_name char(15) not null,
```

- 非空约束

primary key

```
branch_name char(15) primary key,
```

- 主码约束

unique

```
create table department
    (dept_name varchar (20),
    building varchar (15),
    budget numeric (12,2),
    unique (dept_name));
```

- 唯一性约束

check

```
check (semester in ('Fall','Winter','Spring','Summer'));
```

- 枚举约束

13.3.2 引用约束

foreign key ... references ...

```
create table account
 (account-number char(10),
 branch-name char(15),
 balance integer,
 primary key (account_number),
foreign key (branch_name) references branch)
```

- 引用外码

cascade

```
create table account (. . .
    foreign key(branch-name) references branch
    on delete cascade
    on update cascade
    . . . )
```

- 在删除和插入操作破坏引用性质时，级联操作引用的关系

13.4 内嵌数据类型

13.4.1 日期和时间类型

- date：日期 `yyyy-mm-dd`
- time：时间 `hh:mm:ss`
- timestamp：时间戳 `date time`

13.4.2 创建索引

create index

```
create table student
 (ID varchar (5),
 name varchar (20) not null,
 dept_name varchar (20),
 Creation
 tot_cred numeric (3,0) default 0,
 primary key (ID))
 create index studentID_index on student(dept_name)
```

13.4.3 类型转换

create type ... as ...

```
create type Dollars as numeric (12,2) final
```

13.4.4 大对象类型

clob/blob

```
book_review clob(10KB)
image blob(10MB)
movie blob(2GB)
```

13.4.5 授权（略

14 JDBC

14.5 递归*

```
WITH RECURSIVE cte_name(col1, ...) AS ( /*锚查询*/ SELECT ... -- 起点
                                        UNION ALL /*递归查询*/ SELECT ... FROM cte_name ... -- 逐层扩展
                                       ) SELECT * FROM cte_name;
```

- 锚查询：产生初始结果集
- 地柜查询：反复执行，每轮把上轮输出当输入，直到查询不再产生新行
- 保留递归结果：**union all**
- 终止递归：**where或join ... on**
- 实例：找上级

```
WITH RECURSIVE emp_tree AS (
  /*锚：顶层经理*/
  SELECT emp_id, mgr_id, name, 0 AS lvl
  FROM   employee
  WHERE  mgr_id IS NULL

  UNION ALL

  /*递归：找下一级*/
  SELECT  e.emp_id, e.mgr_id, e.name, t.lvl + 1
  FROM   employee  AS e
  JOIN   emp_tree AS t ON e.mgr_id = t.emp_id
)
SELECT * FROM emp_tree
ORDER BY lvl, emp_id;
```

- 实例：生成1-10的数字序列

```
WITH RECURSIVE nums(n) AS ( # 声明临时关系nums(n)
  SELECT 1 # 在nums中添加 (1)
  UNION ALL # 不去重合并
  SELECT n + 1 FROM nums WHERE n < 10 # 当n<10时，在nums中添加(n+1)
)
SELECT * FROM nums;
```