

# 数据库系统原理笔记

陈鸿峥

2020.01\*

## 目录

<b>1</b>	<b>数据库系统概述</b>	<b>3</b>
1.1	高层概述 . . . . .	3
1.2	数据库语言 . . . . .	3
1.3	关系型数据库 . . . . .	4
<b>2</b>	<b>SQL简介</b>	<b>5</b>
2.1	数据定义语言(DDL) . . . . .	5
2.2	数据查询语言 . . . . .	6
<b>3</b>	<b>进阶SQL</b>	<b>9</b>
3.1	内外连接 . . . . .	9
3.2	视图 . . . . .	10
3.3	事务 . . . . .	10
3.4	完整性约束 . . . . .	10
3.5	授权 . . . . .	11
<b>4</b>	<b>形式化关系查询语言</b>	<b>11</b>
4.1	关系代数 . . . . .	11
4.2	其他关系演算 . . . . .	12
<b>5</b>	<b>E-R模型</b>	<b>12</b>
5.1	实体-关系模型 . . . . .	12
5.2	扩展的E-R属性 . . . . .	15

---

\*Build 20200104

<b>6</b>	<b>关系数据库设计</b>	<b>15</b>
6.1	范式理论 . . . . .	15
6.2	函数依赖 . . . . .	17
6.3	分解算法 . . . . .	19
6.4	多值依赖 . . . . .	20
<b>7</b>	<b>存储与文件结构</b>	<b>20</b>
7.1	存储器件 . . . . .	20
7.2	文件结构 . . . . .	21
7.3	文件中记录的组织 . . . . .	22
<b>8</b>	<b>索引与散列</b>	<b>23</b>
8.1	基本概念 . . . . .	23
8.2	顺序索引 . . . . .	23
8.3	B+树索引 . . . . .	25
8.4	静态散列 . . . . .	30
8.5	动态散列 . . . . .	31
8.6	位图索引 . . . . .	32
<b>9</b>	<b>事务</b>	<b>32</b>
9.1	基本概念 . . . . .	32
9.2	可串行化 . . . . .	33
9.3	隔离性级别 . . . . .	34
<b>10</b>	<b>并发控制</b>	<b>34</b>
10.1	基于锁的协议 . . . . .	34
10.2	基于图的协议 . . . . .	36
10.3	死锁处理 . . . . .	36
10.4	基于时间戳的协议 . . . . .	37
<b>11</b>	<b>恢复系统</b>	<b>38</b>
11.1	故障分类 . . . . .	38
11.2	恢复与原子性 . . . . .	38

本课程采用书目Avi Silberschatz, Henry F. Korth, S. Sudarshan, *Database System Concepts (6th ed)*<sup>1</sup>。

---

<sup>1</sup><http://www.db-book.com>

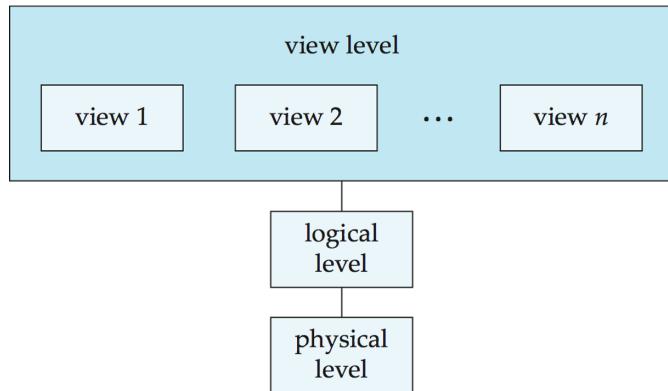
# 1 数据库系统概述

## 1.1 高层概述

早期的数据库直接建立在文件系统上，但这会导致：

- 数据冗余与不一致
- 访问数据非常麻烦
- 数据孤立：数据分散在不同文件中，文件又有不同格式
- 完整性问题：难以添加限制（如年龄为非负整数）
- 原子性问题：要么全部发生要不根本不发生
- 多用户的并发访问
- 安全性问题：权限

数据抽象包括下述三级



- 视图层：屏蔽数据类型细节的一组应用程序，同时提供了访问权限
- 逻辑层：通过类型定义进行描述，同时记录类型之间的相互关系
- 物理层：存储块、物理组织

查询过程：解释编译+求值(evaluation)

数据模型：关系模型、实体-联系模型(ER)、基于对象的数据模型、半结构化数据模型(XML)

## 1.2 数据库语言

- 数据操纵语言(Data Manipulation Language, DML)
- 数据定义语言(Data Definition Language, DDL): DDL的输出放在数据字典中，数据字典包含元数据(metadata)，需要满足一致性约束
  - 域约束：如整型、字符型等
  - 参照完整性(referential integrity): 一个关系中给定属性集上的取值也在另一关系的某一属性集的取值中出现，如 course记录中的dept\_name必须出现在department关系dept\_name属性中

- 断言(assertion): 域约束和参照完整性只是断言的特殊形式, 如“每一学期每一个系必须至少开设5门课程”
- 权限(authorization)

## 1.3 关系型数据库

### 1.3.1 结构

每一个表(table)都是一个关系(relation), 纵向为属性(attributes/columns), 横向为元组(tuples/rows)。一组特定的行称为关系实例(instance)。

ID	name	dept_name
12345	Chen	CS
10001	Bob	Biology
10101	Alice	Physics

注意关系都是无序的, 元组可以以任意顺序存储。

数据库模式(schema)是数据库的逻辑设计, 如`instructor(ID, name, dept_name, salary)` (可以理解为是一个抽象); 数据库实例(instance)则是给定时刻数据库中数据的一个快照, 与具体数据相关。

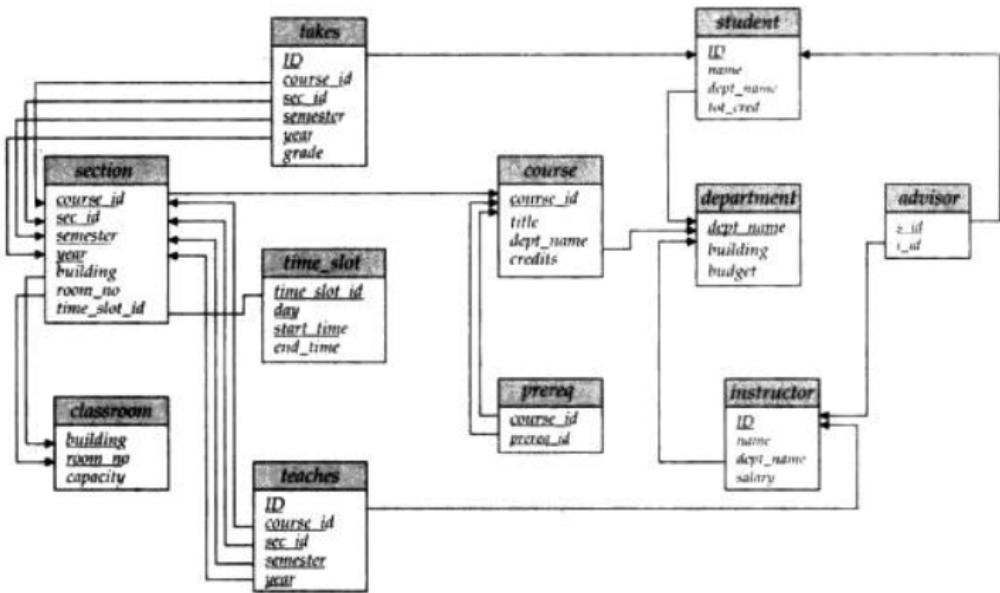
关系对应着程序设计语言中变量的概念, 而关系模式(relation schema)的概念对应于程序设计语言中类型定义的概念。关系模式一定是 $\langle \text{name} \rangle (\langle \text{attr1}, \dots, \text{attrn} \rangle)$ 的形式。

### 1.3.2 码

一个元组的属性值必须是能唯一区分元组的, 即一个关系中没有两个元组在所有属性上取值都相同。

- 超码(superkey): 一个或多个属性的集合使得在一个关系中可以唯一地标识一个元组, 如ID属性是超码, 而名字不是
- 候选码(candidate key): 最小超码, 即其任意真子集都不能成为超码
- 主码(primary key): 数据库设计者选中的、用在一个关系中区分不同元组的候选码。选择要慎重, 哪怕每个人只有一个地理地址, 但是也不应该用其当作主码
- 外码(foreign key): 一个关系模式 $r_1$ 可能在它的属性中包括另一个关系模式 $r_2$ 的主码, 则这个属性在 $r_1$ 上称作参照 $r_2$ 的外码。关系 $r_1$ 也被称为外码依赖的参照关系,  $r_2$ 叫外码的被参照关系。

含有主码和外码以来的数据库模式可用模式图(schema diagram)表示。每个关系用一个矩形表示, 关系名字显示在矩形上方, 矩形内列出各属性, 主码属性用下划线标注, 外码依赖用箭头表示。



## 2 SQL简介

SQL即结构化查询语言(Structured Query Language, SQL)

### 2.1 数据定义语言(DDL)

数据类型

- `char(n)`: 固定长度字符串
- `varchar(n)`: 可变长度字符串, 用得最多
- `int`
- `smallint`
- `numeric(p,d)`: 定点数, 共p位 (包括符号位), 其中d位在小数点右侧
- `real, double precision`
- `float(n)`: 精度至少为n位的浮点数

```

create table instructor (
    ID char(5),
    name varchar(20) not null,
    dept_name varchar(20),
    salary numeric(8,2) default 0,
    primary key (ID), -- integrity constraints
    foreign key (dept_name) references department);

```

## 2.2 数据查询语言

基本的形式如下所示

```
select A1, A2, ..., An  
from r1, r2, ..., rm  
where P
```

其中  $A_i$  为属性(attribute)、 $R_i$  为一个关系(relation)/一个表、 $P$  是谓词(preicate)，返回满足关系的数组。

- 自然连接(natural join): 将两个关系的各个元组进行连接，并且只考虑那些在两个关系模式中都出现的属性上取值相同的元组对（相当于取交）
- 连接(join): 用using可以只考虑某一特定属性的连接

```
select name, title  
from (instructor natural join teachers) join course using (course_id);  
-- only join two relations with the same course_id
```

字符串运算: 模式匹配

- upper(s)、lower(s)
- %: 匹配任意字符串
- \_: 匹配任意一个字符
- like: 以这些字符串进行匹配

```
select distinct dept_name  
from instructor  
  
select * -- all attributes  
from instructor  
  
select '437' as FOO  
  
select ID, name, salary/12 as monthly_salary  
  
select name  
from instructor  
where dept_name = 'Comp. Sci.' and salary > 80000  
  
select *  
from instructor, teaches -- Cartesian product  
  
select distinct T.name  
from instructor as T, instructor as S -- rename, Cartesian product  
where T.salary > S.salary and S.dept_name = 'Comp. Sci.'
```

```

-- salary between 90000 and 100000

/** 
percent ( % ). The % character matches any substring.
underscore ( _ ). The _ character matches any character.
**/

select name
from instructor
where name like '%dar%' matches any string containing "dar" as a substring

select distinct name
from instructor
order by name -- sorted

select *
from instructor
order by salary desc, name asc;

(select course_id from section where sem = 'Fall' and year = 2017)
union -- intersect, except
(select course_id from section where sem = 'Spring' and year = 2018)

select name
from instructor
where salary is null

-- aggregate functions
-- arg, min, max, sum, count
select dept_name, avg (salary) as avg_salary
from instructor
where dept_name= 'Comp. Sci.'
group by dept_name;

select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2018;

-- group constraints
-- find the average salary in each dept
select dept_name, avg(salary) as avg_salary
from instructor
group by dept_name -- can only be contained by select clause
having avg(salary) > 42000; -- effected after grouping

```

三值逻辑，添加了Unknown（其实就是T/F都恒成立的关系才会输出正常值）

AND	OR	NOT
$T \wedge U = U$	$T \vee U = T$	$\neg U = U$
$F \wedge U = F$	$F \vee U = U$	
$U \wedge U = U$	$U \vee U = U$	

SQL在谓词中使用null测试空值，因而为找出instructor关系中salary为空值的所有教师，可写

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

集合成员资格通过in来判断，同时有some和all的关系测试，下面给出了一些嵌套子查询的例子。

```
select name
from instructor
where salary > some (select salary
                      from instructor
                      where dept_name = 'Biology')

select S.ID, S.name
from student as S
where not exists ((select course_id
                   from course
                   where dept_name = 'Biology')
                  except
                  (select T.course_id
                   from takes as T
                   where S.ID = T.ID))

select T.course_id
from course as T
where 1 >= (select count(R.course_id)
             from section as R
             where T.course_id = R.course_id and R.year = 2009);
-- there is no unique in MySQL

with max_budget(value) as
  (select max(budget)
   from department)
select budget
from department, max_budget
where department.budget = max_budget.value;

select dept_name,
```

```

(select count(*)
from instructor
where department.dept_name = instructor.dept_name)
as num_instructors
from department; -- scalar subquery

```

数据库的修改操作如下

```

delete from instructor
where dept_name = 'Finance';

insert into course(course_id, title)
values ('CS-437', 'Database Systems');

update instructor
set salary = salary * 1.05
where salary < 70000;

update instructor
set salary = case
    when salary <= 1000 then salary * 1.05
    when salary <= 2000 then salary * 2.05
    else salary * 1.03
end

```

## 3 进阶SQL

### 3.1 内外连接

on条件可以提供比自然连接更为丰富的连接条件

```

select *
from students join takes on student.ID = takes.ID; -- condition

```

直接使用natural join可能导致元组丢失，比如有一些学生没有选修任何课程，则其在student关系中不会与takes关系中任何元组配对。因此有外连接：

- 左外连接(left outer join)：只保留出现在左外连接运算之前（左边）关系中的元组，若右侧关系中没有对应属性则用null代替
- 右外连接(right outer join)
- 全外连接(full outer join)：左外连接和右外连接的组合

从而可以很容易查询出“所有课程一门也没有选修的学生”：

```

select ID
from student natural left outer join takes

```

```
where course_id is null;
```

为了将常规连接和外连接区分开，SQL中将常规连接称为内连接(inner join)，默认的都是内连接。

## 3.2 视图

让用户看到整个逻辑模型显然是不合适的，出于安全考虑，可能需要向用户隐藏特定的数据。本来通过select可以把需要的数据计算并存储下来，但一旦底层数据发生变化，查询的结果就不再匹配。因此，为了解决这样的问题，SQL提供一种虚关系，称为视图(view)，只有在使用的时候才会被计算。

```
create view faculty as  
select ID, name, dept_name  
from instructor;
```

之后便可以直接使用from语句访问视图。

**定义 1** (物化视图(materialized view))。如果用于定义视图的实际关系改变，视图也会跟着修改。

由于视图并不是数据库底层的关系，故一般数据库不允许对视图关系进行修改。

## 3.3 事务

SQL标准规定当一条SQL语句被执行，就隐式开始了一个事务。下列SQL语句之一会结束一个事务：

- Commit work：将事务所做的更新在数据库中持久保存。在事务被提交后，一个新的事务自动开始。
- Rollback work：撤销该事务中所有SQL语句对数据库的更新，数据库将恢复到执行该事务的第一条语句之前的状态。如果遇断电，回滚会在下一次重启时自动执行。

## 3.4 完整性约束

单个关系上的约束

- not null：跟在属性定义后面
- unique (A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>m</sub>)：声明 A<sub>1</sub>, ..., A<sub>m</sub> 构成一个候选码
- check(<predicate>)：关系中每个元组都必须满足该谓词，如check(budget>0)

参照完整性/子集依赖：外码

```
foreign key (dept_name) references department  
on delete cascade  
on update cascade
```

会级联(cascade)删除或更新。

### 3.5 授权

`grant`授权, `revoke`回收

- `select`: 允许用户修改关系中任意数组
- `insert`
- `delete`

用户名`public`包括了当前所有用户和未来用户的权限。

```
grant <authorization list>
on <relation name/view name>
to <user/user list>

grant update (budget) on department to Amit, Satoshi;
```

可以先创建角色(role)然后给用户授予角色。

## 4 形式化关系查询语言

### 4.1 关系代数

选择	$\sigma$	挑选出符合一定性质的元组 $\sigma_{\text{Sub}=\text{"Phy"} \wedge \text{age} > 30}(\text{teachers})$
投影	$\Pi$	只选出对应属性 $\Pi_{\text{ID}, \text{name}, \text{salary}}(\text{teachers})$
笛卡尔积	$\times$	将两个关系整合（简单并置，需要进一步筛选）
自然连接	$r \bowtie s = \prod_{R \cup S} (\sigma_{r.A_1=s.A_1 \wedge \dots \wedge r.A_n=s.A_n})(r \times s)$	
$\theta$ 连接	$r \bowtie_\theta s = \sigma_\theta(r \times s)$	
外连接	$\bowtie_L, \bowtie_R, \bowtie_{LR}$	
并集	$\cup$	数目应相同，属性可兼容
交集	$\cap$	
差集	$-$	
赋值	$\leftarrow$	
重命名	$\rho_x(E)$	给 $E$ 的返回值赋名为 $x$

扩展的关系代数运算:

- 广义投影:  $\prod_{F_1, F_2, \dots, F_n}(E)$ , 其中 $F_i$ 中每一个都是涉及常量及 $E$ 的模式中属性的算术表达式
- 聚类:  $\mathcal{G}$ , 如`count`, `min`, `max`, 考虑`group by`可以写成下列形式

$$\text{dept\_name} \mathcal{G}_{\text{average}(\text{salary})}(\text{instructor})$$

**定义 2** (聚集运算). 聚集运算 $\mathcal{G}$ 的通常形式如下:

$$G_1, G_2, \dots, G_n \mathcal{G}_{F_1(A_1), \dots, F_n(A_n)}(E)$$

其中 $E$ 是任意关系代数表达式,  $G_1, \dots, G_n$ 是用于分组的一系列属性; 每个 $F_i$ 是一个聚集函数, 每个 $A_i$ 是一个属性名。 $E$ 结果中的元组将会以如下方式分为若干组:

- 同组中所有元组在 $G_1, \dots, G_n$ 上的取值相同
- 不同组中的元组在 $G_1, \dots, G_n$ 上的取值不同

## 4.2 其他关系演算

### 4.2.1 元组关系演算

元组关系演算是非过程化的查询语言 (有点像声明式语言), 它只描述所需信息, 而不给出获得该信息的具体过程。

$$t \in \text{instructor} \wedge \exists s \in \text{department}(t[\text{dept\_name}] = s[\text{dept\_name}])$$

前者 $t$ 为自由变量, 后者 $s$ 为受限变量。

### 4.2.2 域关系演算

$$\{ < x_1, x_2, \dots, x_n > \mid P(x_1, x_2, \dots, x_n) \}$$

其中 $< x_1, \dots, x_n > \in r$ , 每一个 $x_i$ 为域变量,  $r$ 为 $n$ 个属性上的关系。

## 5 E-R模型

在设计一个数据库模式时, 要避免以下两个缺陷:

- 冗余: 可能导致不同关系表中的信息没有及时更新
- 不完整: 只有对应开课的实体而没有对应课程的实体

### 5.1 实体-关系模型

**定义 3** (实体). 实体是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/属性(*attribute*), 其中一些性质的值可以唯一标识一个实体。实体集则是相同类型具有相同属性或性质的一个实体集合。实体集是一个抽象概念, 而实体集的外延(*extension*)则是指属于实体集实体的实际集合。

大学中实际教师的集合构成了实体集`instructor`的外延。实体集不必互不相交, 如定义大学里所有人的实体集(`person`)。一个`person`实体可以是`instructor`实体, 可以是`student`实体, 可以既是`instructor`实体又是`student`实体, 也可以都不是。

**定义 4 (联系).** 联系(*relationship*)是指多个实体间的相互关联。联系集是相同类型联系的集合。

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

联系也可以具有描述性属性（如下右图）。

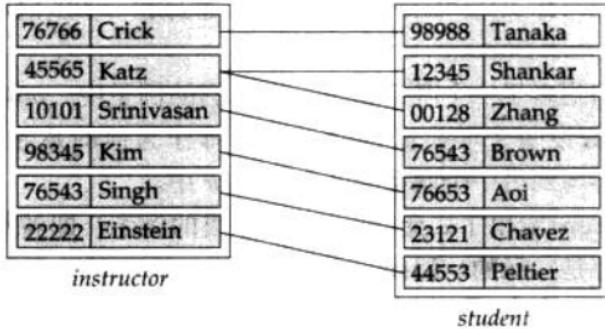


图 7-2 联系集 *advisor*

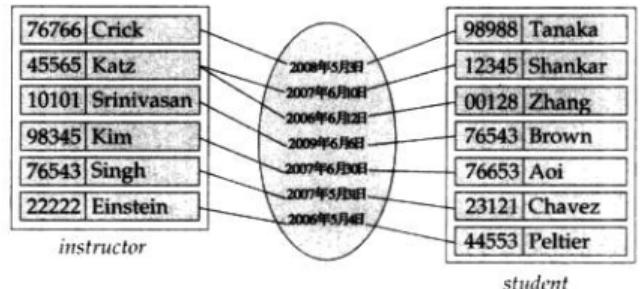


图 7-3 *date* 作为联系集 *advisor* 的属性

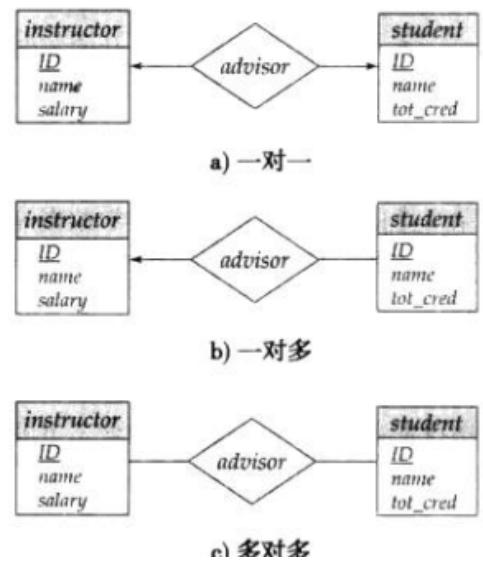
实体集之间的关联称为参与(*participation*)，若 $E$ 中每个实体都参与到联系集 $R$ 中的至少一个联系中，则称参与是全部的(*total*)的，否则是部分的(*partial*)。E-R模式中的联系实例则为具体命名实体间的关联。实体在联系中扮演的功能称为实体的角色(*role*)。参与联系集的实体集数目称为联系集的度(*degree*)，二元联系集的度为2，三元联系集度为3。

**定义 5 (属性).** 每个属性都有一个可取值的集合，称为该属性的域(*domain*)，或值集(*value set*)。属性可以分为简单和复合(*composite*)属性，复合属性可以再划分为更小的部分，如名字分为名和姓。也可能是单值或多值属性，比如老师可以有多个电话号码，这是多值属性。还有派生(*derived*)属性，可以通过其他属性计算得出。

通常如果一个属性在两个实体集中出现，且这两个实体集存在关联，则该属性是冗余的，需要被移除。

**定义 6 (映射基数(mapping cardinality)/基数比率).** 一个实体通过一个联系集能关联的实体个数，包括了一对一、一对多(*many*)、多对一、多对多这几种情况。*one*代表至多一个，*many*代表零个或多个。

- 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。



也可以采用更为复杂的映射基数 $l..h$ 的形式表示, 其中 $l$ 表示最小映射基数,  $h$ 为最大映射基数。最小值为1代表这个实体集在该联系集中全部参与, 最大值为1表示这个实体至多参与一个联系, 最大值为\*则没有限制。

**定义 7** (强弱实体集). 没有足够的属性形成主码的实体集称为弱实体集, 有主码的实体集则称为强实体集。弱实体集必须与另一个称作标识(*identifying*)或属主(*owner*)实体集的实体集关联才有意义。弱实体集的分辨符(*discriminator*)/部分码是使得能够区分弱实体集中实体的方法, 用虚下划线标明。

E-R图, 菱形代表联系, 双横线代表全部参与。

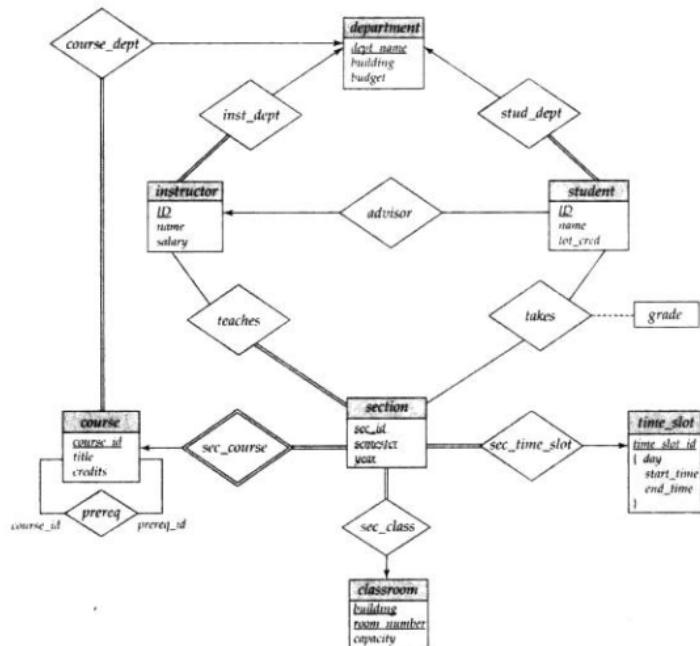


图 7-15 大学的 E-R 图

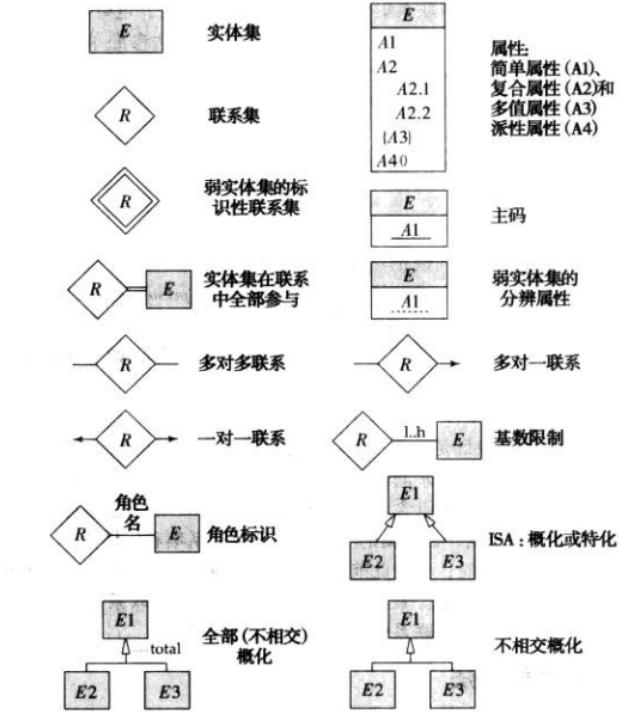


图 7-24 E-R 图表示法中使用的符号

一个常见的错误是将一个实体集的主码作为另一个实体集的属性（关系被隐藏了），而不是使用联系。

## 5.2 扩展的E-R属性

**定义 8 (特化和泛化).** 在实体集内部进行分组的过程称为特化(specialization)，如将实体集 *person* 划分为 *employee* 和 *student*。泛化(generalization)则是自底向上的反过程。高层与低层实体集也可被分别称为超类和子类，同样有继承(inheritance)的特性。如果一个实体集作为低层实体集参与到多个联系中，则称这个实体集具有多继承，且产生的结构称为格(lattice)。

**定义 9 (聚集).** 聚集是一种抽象，其中联系集和跟它们相关的实体集一起被看作高层实体集，并且可以参与联系。

## 6 关系数据库设计

大的模式会存在大量冗余，小的模式会导致信息丢失（有损分解）。

### 6.1 范式理论

定义属性集  $\alpha$ ，关系模式为  $r(R)$ 。关系模式是一个属性集，但不是所有属性集都是模式。超码则用  $K$  来表示。

**定义 10 (超码).**  $R$  的子集  $K$  是  $r(R)$  的超码的条件是：在关系  $r(R)$  的任意合法实例中，对于  $r$  实例中的元组对  $t_1$  和  $t_2$  总满足，若  $t_1 \neq t_2$ ，则  $t_1[K] \neq t_2[K]$ ，即  $K$  唯一标识一条元组。（而 **函数依赖** 则是唯一标识某些

## 属性)

设计需要满足一定的范式(normal form)，核心目的是减少冗余：

- 第一范式(1NF)：全部属性都是**单值属性**（原子性）
- 第二范式(2NF)：关系模式 $R \in 1NF$ ，且每一个**非主属性完全依赖于R的主码**，而不是只依赖于其中一部分属性（即其中一部分属性的值即可确定该属性的值，部分依赖）

例 1. 一个学生-课程关系模式如下：

<i>Stud_No</i>	<i>Course_No</i>	<i>Course_Fee</i>
1	C1	1000
2	C2	1500
1	C4	2000
4	C3	1000
4	C1	1000
2	C5	2000

如此例中 *Stud\_No* 和 *Course\_No* 是主码，但是实际上只由 *Course\_No* 就已经可以决定 *Course\_Fee* 了，因此这个例子不满足 2NF。

- 第三范式(3NF)：关系模式 $R \in 2NF$ ，且对于 $F^+$ 中所有形如 $\alpha \rightarrow \beta$ 的函数依赖，至少下列一项成立
  - $\alpha \rightarrow \beta$  是一个平凡的函数依赖
  - $\alpha$  是 *R* 的一个超码
  - $\beta - \alpha$  中的每个属性 *A* 都含于 *R* 的一个候选码中

(非主属性依赖于主码但不能通过另一非主属性进行依赖，即**不存在传递依赖**)

例 2. 一个学生-国家关系模式如下：

<i>Stud No</i>	<i>Stud Name</i>	<i>Stud State</i>	<i>Stud Country</i>	<i>Stud Age</i>
1	Alice	s1	c	18
2	Alice	s2	c	19
3	Bob	s2	c	21

有传递依赖 *Stud No*  $\rightarrow$  *Stud State*  $\rightarrow$  *Stud Country*，因此不满足 3NF。

- 巴斯-科德/BC范式(Boyce-Codd NF, 3.5NF)：关系模式 $R \in 3NF$ ，对于 $F^+$ 中所有形如 $\alpha \rightarrow \beta$ 的函数依赖，至少有以下一项成立：
  - $\alpha \rightarrow \beta$  是平凡的函数依赖（即 $\beta \subset \alpha$ ）
  - $\alpha$  是模式 *R* 的一个超码（即  $\alpha$  可以整条元组）

## 6.2 函数依赖

**定义 11** (函数依赖(functional dependency)). 设 $R$ 为关系模式,  $\alpha \subset R, \beta \subset R$ , 函数依赖 $\alpha \rightarrow \beta$ 在 $R$ 上满足, 当且仅当对于任意合法的关系 $r(R)$ , 任何两个关于 $r$ 的数对 $t_1$ 和 $t_2$ , 如果满足属性 $\alpha$ , 那么它们也满足属性 $\beta$ , 即

$$t_1[\alpha] = t_2[\alpha] \implies t_1[\beta] = t_2[\beta]$$

实际上就是**函数单射**的概念, 属性 $\alpha$ 可以**唯一确定**属性 $\beta$ 的值。若函数依赖 $K \rightarrow R$ 在 $r(R)$ 上成立 (注意这里 $R$ 相当于全部属性, 或者写成 $K \rightarrow (R - K)$ ), 则 $K$ 是 $r(R)$ 的一个**超码**。

**例 3.** 考虑下例 $r(A, B)$ 的关系

A	B
1	4
1	5
3	7

关系 $A \rightarrow B$ 不成立, 但关系 $B \rightarrow A$ 成立。

**定义 12** (平凡). 在所有关系中都满足的函数依赖则是平凡的函数依赖, 比如 $A \rightarrow A$ 。一般地, 若 $\beta \subset \alpha$ , 则 $\alpha \rightarrow \beta$ 的函数依赖是平凡的。

### 6.2.1 闭包

**定义 13** (逻辑蕴含与闭包). 若关系模式 $r(R)$ 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。 $F$ 的闭包是被 $F$ 逻辑蕴含的所有函数依赖的集合, 记作 $F^+$ 。设 $\alpha$ 为属性集, 将函数依赖集 $F$ 下被 $\alpha$ 函数确定的所有属性的集合称为 $F$ 下 $\alpha$ 的闭包。

**定理 1** (逻辑蕴含公理). 下面的前三条为最基本的公理(Armstrong), 可以找出给定 $F$ 的所有 $F^+$

- 自反律(reflexivity): 若 $\alpha$ 为一属性集且 $\beta \subset \alpha$ , 则 $\alpha \rightarrow \beta$
- 增补律(augmentation): 若 $\alpha \rightarrow \beta$ 成立且 $\gamma$ 为一属性集, 则 $\gamma\alpha \rightarrow \gamma\beta$ 成立
- 传递律(transitivity): 若 $\alpha \rightarrow \beta$ 和 $\beta \rightarrow \gamma$ 成立, 则 $\alpha \rightarrow \gamma$ 成立
- 合并律(union): 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

---

#### Algorithm 1 计算 $F^+$

```

1:  $F^+ := F$ 
2: repeat
3:   对 $F^+$ 的函数依赖应用自反律、增补律和传递律, 将结果加入 $F^+$ 
4: until result 不变

```

---

---

**Algorithm 2** 计算 $F$ 下属性 $\alpha$ 的闭包 $\alpha^+$ 

---

```
1: result :=  $\alpha$ 
2: repeat
3:   for 每一个函数依赖 $\beta \rightarrow \gamma \in F$  do
4:     若 $\beta \subset \text{result}$ , 则 $\text{result} := \text{result} \cup \gamma$ 
5: until result 不变

---


```

属性闭包算法有多种用途:

- 用于**判断 $\alpha$ 是否为超码**, 计算 $\alpha^+$ , 检查 $\alpha^+$ 是否包含 $R$ 中所有属性
- 通过检查是否 $\beta \subset \alpha^+$ , 我们可以检查 $\alpha \rightarrow \beta$ 是否成立 (即是否属于 $F^+$ ), 也就是用属性闭包计算 $\alpha^+$ , 看是否包含 $\beta$
- 另一种计算 $F^+$ 的方法:  $\forall \gamma \subset R$ , 找出 $\gamma^+$ ;  $\forall S \subset \gamma^+$ , 输出函数依赖 $\gamma \rightarrow S$

### 6.2.2 正则覆盖

**定义 14** (无关(extraneous)). 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, 即**去掉该属性依然可以推得其函数依赖成立**, 形式化定义即

- 若 $A \in \alpha$ 且 $F$ 逻辑蕴含 $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ , 则属性 $A$ 在 $\alpha$ 中无关
- 若 $A \in \beta$ 且函数依赖集 $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ 逻辑蕴含 $F$ , 则属性 $A$ 在 $\beta$ 中无关

检验属性 $A$ 是否无关的方法如下:

- 若 $A \in \beta$ , 为检验 $A$ 是否无关, 考虑集合

$$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$$

**检验 $\alpha \rightarrow A$ 是否能由 $F'$ 推出**, 计算 $F'$ 下的 $\alpha^+$ , 若 $\alpha^+$ 包含 $A$ , 则 $A$ 在 $\beta$ 中无关。

- 若 $A \in \alpha$ , 为检验 $A$ 是否无关, 令 $\gamma = \alpha - \{A\}$ , **检验 $\gamma \rightarrow \beta$ 是否可以由 $F$ 推出**, 计算 $F$ 下的 $\gamma^+$ , 若 $\gamma^+$ 包含 $\beta$ 中所有属性, 则 $A$ 在 $\alpha$ 中无关。

简言之, 计算 $A \subset \alpha_{F'}^+$ 和 $\beta \subset \gamma_F^+$ , 但这很麻烦, 通常直接通过逻辑蕴含判断。

**定义 15** (正则覆盖(canonical cover)).  $F$ 的正则覆盖 $F_c$ 是一个依赖集, 使得 $F_c$ 和 $F$ 双向逻辑蕴含, 且

- $F_c$ 中任何函数依赖都不含无关属性
- $F_c$ 中函数依赖的左半部都是唯一的, 即不存在 $\alpha_1 \rightarrow \beta_1$ 和 $\alpha_2 \rightarrow \beta_2$ 满足 $\alpha_1 = \alpha_2$

### 6.2.3 无损分解与保持依赖的分解

**定义 16** (无损分解). 用 $r(R_1)$ 和 $r(R - 2)$ 代替 $r(R)$ 没有信息损失, 则为无损分解。

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

---

**Algorithm 3** 计算正则覆盖

---

- 1:  $F_c = F$
  - 2: **repeat**
  - 3:     用合并律将  $F_c$  中  $\alpha_1 \rightarrow \beta_1$  和  $\alpha_1 \rightarrow \beta_2$  替换为  $\alpha_1 \rightarrow \beta_1\beta_2$
  - 4:     对  $F_c$  每一个函数依赖移除无关属性
  - 5: **until**  $F_c$  不变
- 

**定义 17** (保持依赖的分解).  $F$  是模式  $R$  上一个函数依赖集,  $R_i$  为  $R$  的一个分解,  $F$  在  $R_i$  上限定 (restriction) 是  $F^+$  中所有只包含  $R_i$  中属性的函数依赖的集合  $F_i$ 。令  $F' = \bigcup_i F_i$ , 具有  $F'^+ = F^+$  的分解为保持依赖的分解。

## 6.3 分解算法

### 6.3.1 BCNF 分解

简化判定方法:

- 检查非平凡函数依赖  $\alpha \rightarrow \beta$ , 计算  $\alpha^+$ , 验证它是否包含  $R$  中所有属性, 即验证它是否为  $R$  的超码; 不需检查  $F^+$  中所有函数依赖 (但分解后就不能这么做了)
- 对  $R_i$  中属性每个子集  $\alpha$ , 确保  $F$  下  $\alpha^+$  要么不含  $R_i - \alpha$  的任何属性, 要么包含  $R_i$  所有属性。若  $R_i$  上有某个属性集  $\alpha$  违反条件, 即下述函数依赖会出现在  $F^+$  中, 说明违反 BCNF

$$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$$

---

**Algorithm 4** BCNF 分解

---

- 1: 初始化  $result := \{R\}$
  - 2: 计算  $F^+$
  - 3: 若  $result$  中存在模式  $R_i$  不属于 BCNF, 则将其分解为  $(R_i - \beta)$  和  $(\alpha, \beta)$ , 其中  $\alpha \rightarrow \beta$  为  $R_i$  上成立的非平凡函数依赖, 满足  $\alpha \rightarrow R_i$  不属于  $F^+$ , 且  $\alpha \cap \beta = \emptyset$
- 

### 6.3.2 3NF 分解

---

**Algorithm 5** 3NF 分解

---

- 1: 令  $F_c$  为  $F$  的正则覆盖
  - 2: 对于每一个  $F_c$  中的函数依赖  $\alpha \rightarrow \beta$ ,  $R_i = \alpha\beta$
  - 3: 若  $R_i$  都不包含  $R$  的候选码, 则新增  $R_{i+1}$  为  $R$  的候选码
  - 4: 若  $R_j \in R_k$ , 则删除  $R_j$ , 直至不再有可删除的  $R_j$
- 

### 6.3.3 总结

应用函数以来进行数据库设计的目标是:

1. BCNF

2. 无损
3. 保持依赖

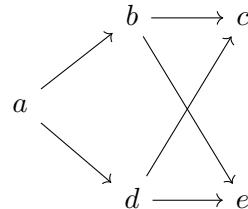
#### 6.4 多值依赖

函数依赖规定了某些元组不能出现在关系中。如果  $A \rightarrow B$ , 则不能有两个元组在  $A$  上的值相同, 而在  $B$  上值不同。多值依赖不排除某些元组的存在, 而要求某种形式的其他元组存在于关系中。函数依赖称为相等产生(equality-generating)依赖, 多值依赖称为元组产生依赖。

**定义 18** (多值依赖(multivalued dependency)). 设  $R$  为关系模式,  $\alpha \subset R, \beta \subset R$ , 多值依赖在  $R$  上满足  $\alpha \rightarrow\rightarrow \beta$ , 当且仅当对于所有数对  $t_1$  和  $t_2$ , 使得  $t_1[\alpha] = t_2[\alpha]$ , 都存在数对  $t_3$  和  $t_4$  使得

$$\begin{aligned} t_1[\alpha] &= t_2[\alpha] = t_3[\alpha] = t_4[\alpha] \\ t_3[\beta] &= t_1[\beta] \\ t_3[R - \beta] &= t_2[R - \beta] \\ t_4[\beta] &= t_2[\beta] \\ t_4[R - \beta] &= t_1[R - \beta] \end{aligned}$$

简而言之, 若  $(a, b, c)$  和  $(a, d, e)$  在  $R$  中, 则  $(a, b, e)$  和  $(a, d, c)$  在  $R$  中。即两个属性互相独立, 但是都依赖于第三个属性。



典型例子是(课程, 课本, 老师), 一门课有两本课本, 两个老师, 课本和老师之间不存在函数依赖, 但都依赖于课程。

令  $D$  表示函数依赖和多值依赖的集合, 则  $D^+$  是由  $D$  逻辑蕴含的所有函数依赖和多值依赖的集合。对于多值依赖有以下规则

- 每个函数依赖都是一个多值依赖, 即若  $\alpha \rightarrow \beta$ , 则  $\alpha \rightarrow\rightarrow \beta$
- 若  $\alpha \rightarrow\rightarrow \beta$ , 则  $\alpha \rightarrow\rightarrow R - \alpha - \beta$

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

- $\alpha \rightarrow\rightarrow \beta$  是一个平凡的多值依赖
- $\alpha$  是  $R$  的一个超码 (这里已经蕴含了  $4NF \in BCNF$ )

# 7 存储与文件结构

## 7.1 存储器件

物理存储：易失(volatile)、非易失

磁盘被划分为道、扇区、柱面

常见磁盘性能衡量指标：

- 访问时间：
    - 寻道(seek)时间：到达特定的道
    - 旋转时间
  - 数据传输速率
  - 平均失效时间(MTTF)：通常3-5年
- 通过并行来提高性能：数据拆分(striping)
- 比特级拆分：每个字节按比特分开，存储到多个磁盘上
  - 块级拆分：将块拆分到多张磁盘，即将磁盘阵列看成一张单独的大磁盘，并给块进行逻辑编号
- 独立磁盘冗余阵列(Redundant Array of Independent Disk, RAID)
- RAID-0：块级拆分但无冗余
  - RAID-1：块级拆分+冗余，数据库常用
  - RAID-2：纠 错 码(Error-Correcting-Code, ECC)，可以读出其余位和纠错位重建信息
  - RAID-3：可以检测一整个扇区是否被正确读出
  - RAID-4：在一张独立磁盘上为其他 $N$ 张磁盘对应的块保留一个奇偶校验块
  - RAID-5：将数据和奇偶校验位分布到 $N + 1$ 张磁盘中，所有磁盘都能参与读请求
  - RAID-6：P+Q冗余方案，可应对多张磁盘故障情况
- The diagram illustrates seven RAID levels (a-g) using cylinders to represent disks:

  - a) RAID 0: 无冗余拆分 (No redundancy striping). Shows four identical cylinders.
  - b) RAID 1: 镜像磁盘 (Mirrored disk). Shows two identical cylinders followed by three identical cylinders labeled 'C'.
  - c) RAID 2: 内存风格的纠错码 (Memory-style error-correcting code). Shows four identical cylinders followed by two cylinders labeled 'P'.
  - d) RAID 3: 位交叉奇偶校验 (Bit-interleaved parity). Shows four identical cylinders followed by one cylinder labeled 'P'.
  - e) RAID 4: 块交叉奇偶校验 (Block-interleaved parity). Shows four identical cylinders followed by one cylinder labeled 'P'.
  - f) RAID 5: 块交叉的分布奇偶校验 (Distributed parity). Shows five cylinders: the first two are labeled 'P', the next three are labeled 'P'.
  - g) RAID 6: P+Q冗余 (P+Q redundancy). Shows six cylinders: the first two are labeled 'P', the next two are labeled 'P', and the last two are labeled 'P'.

## 7.2 文件结构

### 7.2.1 定长记录

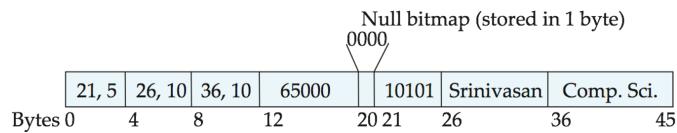
每次插入删除都涉及后续元组的移动，可以如下维护一个空闲列表(free list)

header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

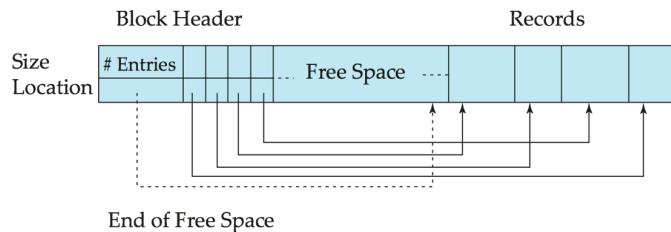
### 7.2.2 变长记录

变长记录以以下几种方式出现:

- 多种记录类型在一个文件中存储
- 允许一个或多个字段是变长的记录类型
- 允许可重复字段的记录类型，如数组或多重集合



用分槽的页结构(slotted-page)来处理块中变长记录的问题



块头包含以下信息

- 块头中记录条目的个数
- 块中空闲空间的末尾处
- 由包含记录位置和大小的记录条目组成的数组

注意实际记录是从**块的尾部**往前插。

## 7.3 文件中记录的组织

- 堆文件组织：一条记录可放在文件中任何地方，只要那个地方有空间存放这条记录

- 顺序文件组织：根据其搜索码值顺序存储
- 散列文件组织：散列函数

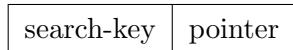
多表聚簇组织

Comp. Sci.	Taylor	100000	
45564	Katz	75000	
10101	Srinivasan	65000	
83821	Brandt	92000	
Physics	Watson	70000	
33456	Gold	87000	

## 8 索引与散列

### 8.1 基本概念

通常索引文件包含记录/索引项，要比原文件小很多



两种基本的索引类型：顺序(ordered)索引、散列(hash)索引

### 8.2 顺序索引

**定义 19** (聚集索引). 在顺序索引中，包含记录的文件按照某个搜索码指定顺序排序，则该搜索码对应的索引称为聚集索引(*clustering index*)，也被称为主索引(*primary*)。但注意并不是建立在主码上的索引。若搜索码指定的顺序与文件中记录的物理顺序不同，则称为非聚集索引(*secondary*)。

**定义 20** (稠密/稀疏索引). 对于每个搜索码都出现索引记录的称为稠密索引，只包含特定搜索码值的称为稀疏索引。非聚集索引一定是稠密的。

稀疏索引查找方法即找到最大搜索码键值小于  $K$  的，然后继续往下搜索。

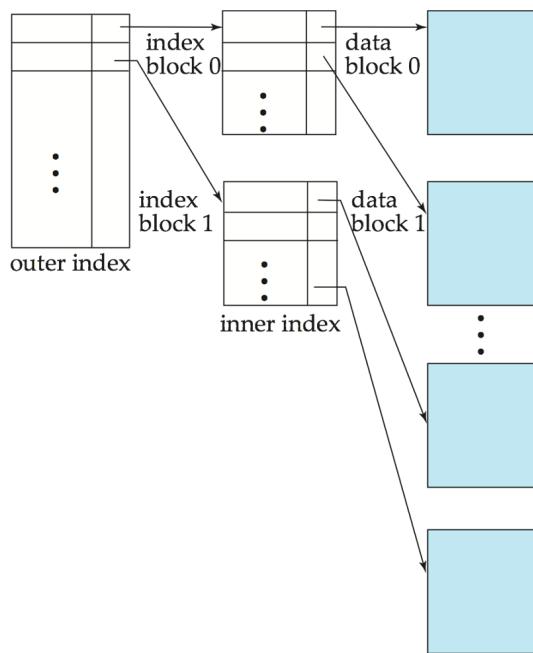
10101		10101	Srinivasan	Comp. Sci.	65000	
12121		12121	Wu	Finance	90000	
15151		15151	Mozart	Music	40000	
22222		22222	Einstein	Physics	95000	
32343		32343	El Said	History	60000	
33456		33456	Gold	Physics	87000	
45565		45565	Katz	Comp. Sci.	75000	
58583		58583	Califieri	History	62000	
76543		76543	Singh	Finance	80000	
76766		76766	Crick	Biology	72000	
83821		83821	Brandt	Comp. Sci.	92000	
98345		98345	Kim	Elec. Eng.	80000	

图 11-2 稠密索引

10101		10101	Srinivasan	Comp. Sci.	65000	
32343		12121	Wu	Finance	90000	
76766		15151	Mozart	Music	40000	
		22222	Einstein	Physics	95000	
		32343	El Said	History	60000	
		33456	Gold	Physics	87000	
		45565	Katz	Comp. Sci.	75000	
		58583	Califieri	History	62000	
		76543	Singh	Finance	80000	
		76766	Crick	Biology	72000	
		83821	Brandt	Comp. Sci.	92000	
		98345	Kim	Elec. Eng.	80000	

图 11-3 稀疏索引

通过建造多级稀疏索引，来减少访问时间。



索引更新操作：

- 删除：稠密连同搜索码直接删，稀疏判大小
- 插入：同上直接插

## 8.3 B+树索引

当文件大起来时上述索引变得很慢，周期性更新整个文件非常麻烦，因此有B+树自动管理小的局部插入删除。

### 8.3.1 基本性质

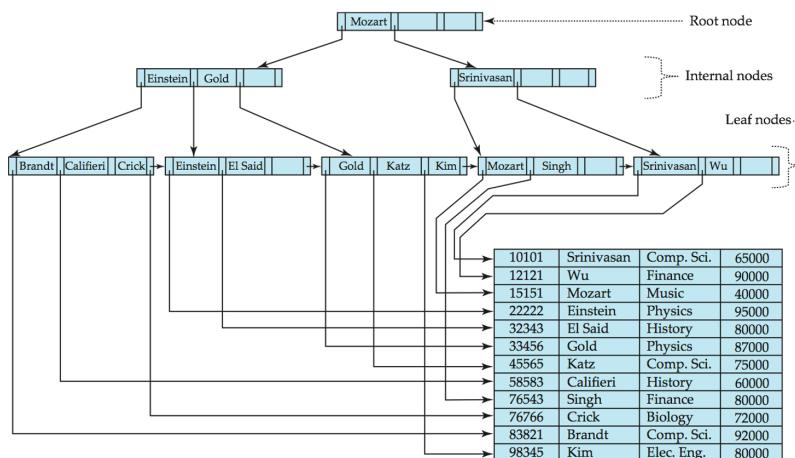
B+树索引采用平衡树结构，树根到树叶的每条路径长度相同。 $n$ -阶B树满足以下性质：

- 所有叶子结点都必须在同一层
- 除了根结点外的非叶子节点都至少有 $\lceil n/2 \rceil$ 个孩子，至多 $n$ 个孩子
- 叶子结点至少有 $\lceil (n-1)/2 \rceil$ 个关键字，最多 $n-1$ 个关键字
- 特殊情况：
  - 根节点不是叶子，则至少有2个孩子
  - 根节点是叶子，则可以有 $[0, n-1]$ 个值
- 每个结点有 $n$ 个指针， $n-1$ 个键值/搜索码升序排序（通常是严格不等式）

$P_1$	$K_1$	$P_2$	$\dots$	$P_{n-1}$	$K_{n-1}$	$P_n$
-------	-------	-------	---------	-----------	-----------	-------

- 左侧的叶子结点的搜索码值一定都小于等于右侧的叶子结点的搜索码值
- 指针与其后面的键值为一组，即 $P_i$ 指向搜索码值为 $K_i$ 的记录（叶子结点），或 $P_i$ 指向 $[K_{i-1}, K_i)$ （非叶子节点，注意右侧不包含）； $P_n$ 指向下一个叶子节点

而B+树则是将所有关键字存储在叶子结点，其他结点作为索引，并且为每个叶子结点增加一个链指针。

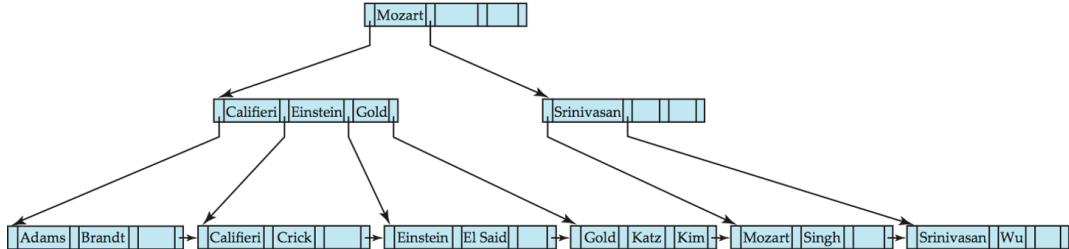
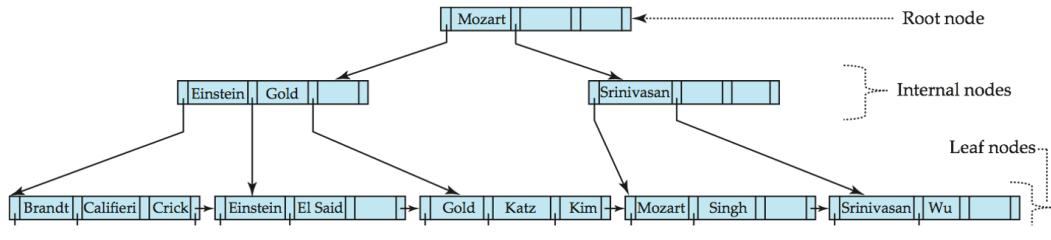


根节点下至少有 $2\lceil n/2 \rceil$ 个键值，下一层至少 $2\lceil n/2 \rceil^2$ 个，以此类推；若记录中有 $K$ 个搜索码值，则B+树高不会高过 $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$ 。

### 8.3.2 插入

对于叶子节点的插入<sup>2</sup>:

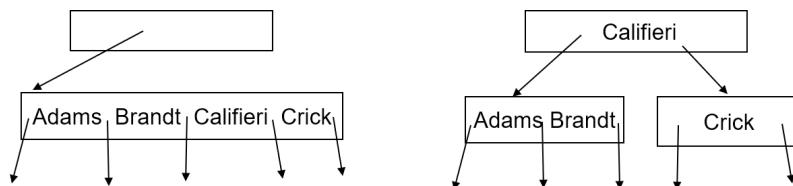
- 搜索码值已经存在于叶子结点: 直接加记录
- 搜索码值没有出现, 添加记录到主文件, 插入到叶子节点; 如果没有位置, 需要进行分裂
  - 将这 $n$ 个(搜索码值, 指针)进行排序, 取前 $\lceil n/2 \rceil$ 个在原来节点, 其余在新节点
  - 令新节点为 $p$ ,  $k$ 为 $p$ 中的最小值, 则插入 $(k, p)$ 到父亲中,  $p$ 为 $k$ 的后续指针
  - 若父亲满了, 则继续分裂并传播



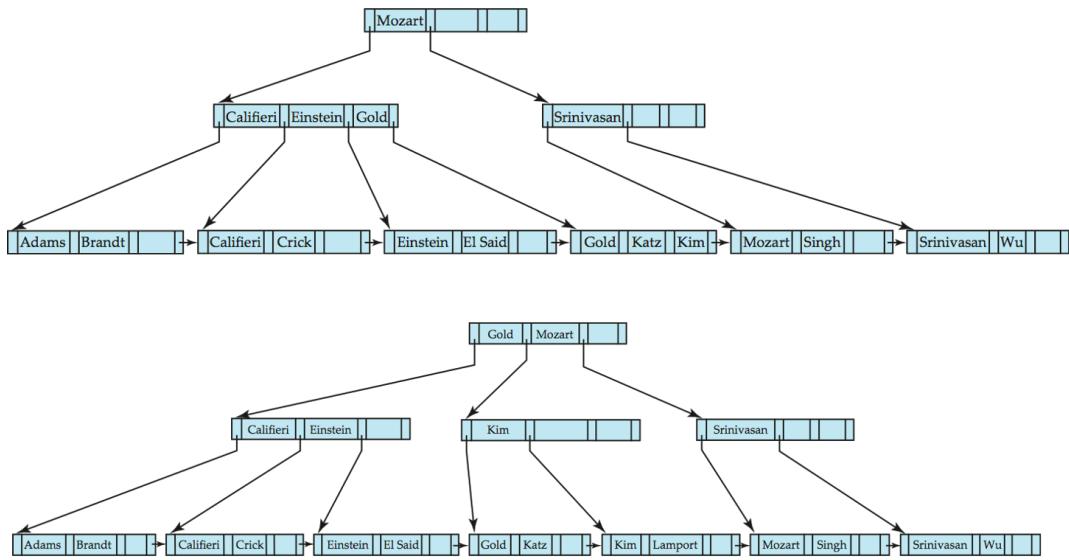
B<sup>+</sup>-Tree before and after insertion of "Adams"

对于非叶子节点, 当插入 $(k, p)$ 到一个已经满的内部节点时:

- 将 $P_1, K_1, \dots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ 放到第一个节点
- 将 $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \dots, K_n, P_{n+1}$ 放到第二个节点
- 将 $(K_{\lceil n/2 \rceil}, N')$ 插入到父节点 $N$



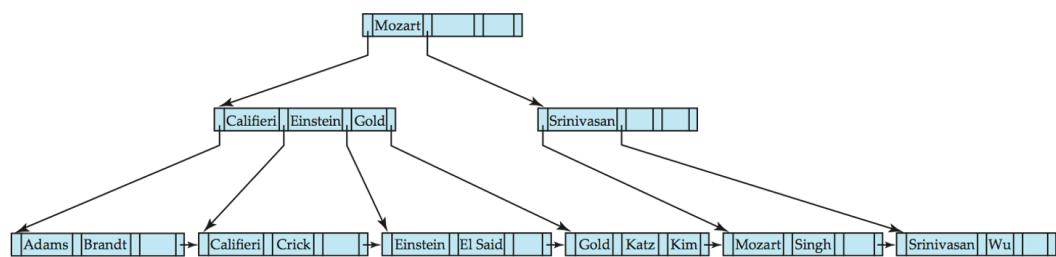
<sup>2</sup>B<sup>+</sup>树的动态演示可见<https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>



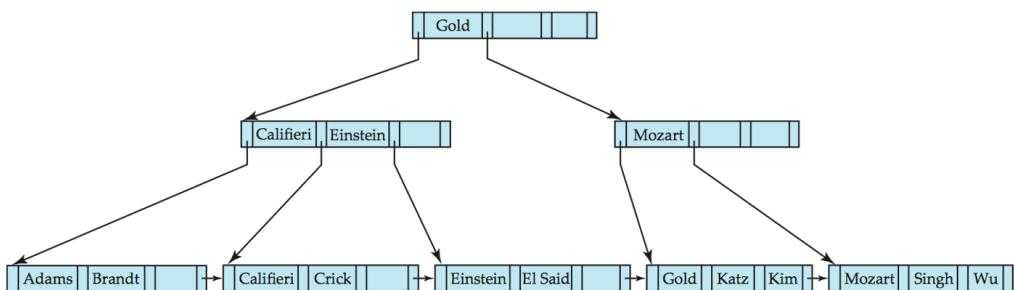
B<sup>+</sup>-Tree before and after insertion of "Lamport"

### 8.3.3 删 除

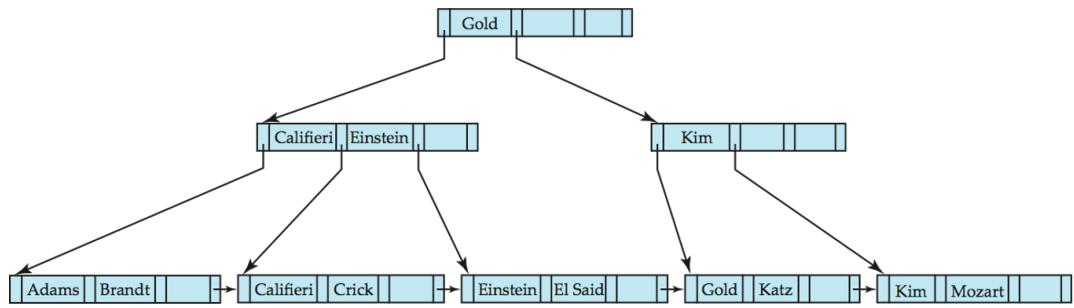
- 直接移除
- 如果删除导致下溢(underfull)且可合并到一个结点，则合并兄弟
  - 若**兄弟节点**未满，则与兄弟节点合并（注意不是同层都是兄弟）
  - 删除( $K_{i-1}, P_i$ )，其中 $P_i$ 是指向删除结点的指针，更新索引值，递归这个过程
- 如果合并没法合到一个节点，则需要**重分配(redistribute)**指针使得两个兄弟节点都含有多于最小键值数目的项，同时更新父亲节点的搜索码



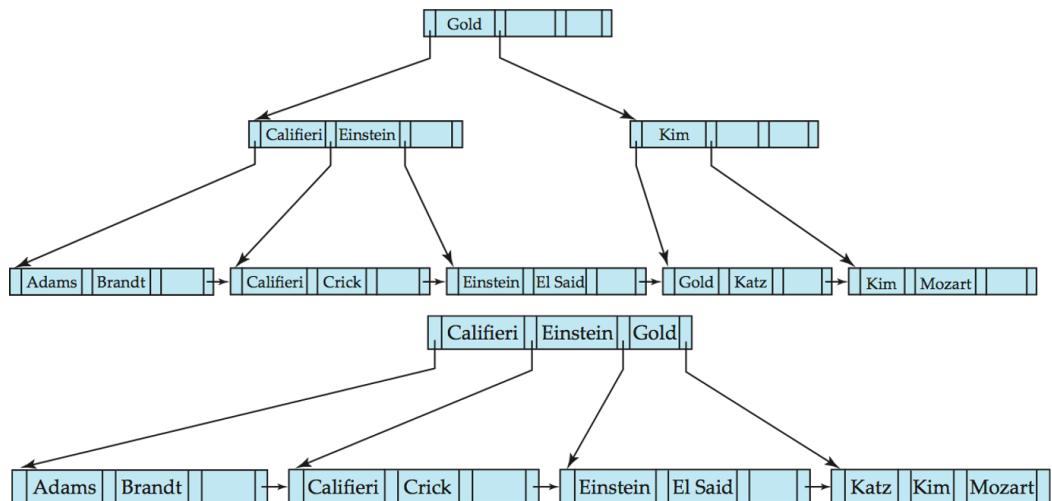
Before and after deleting "Srinivasan"



□ Deleting "Srinivasan" causes merging of under-full leaves



Deletion of “Singh” and “Wu” from result of previous example



Before and after deletion of “Gold” from earlier example

```

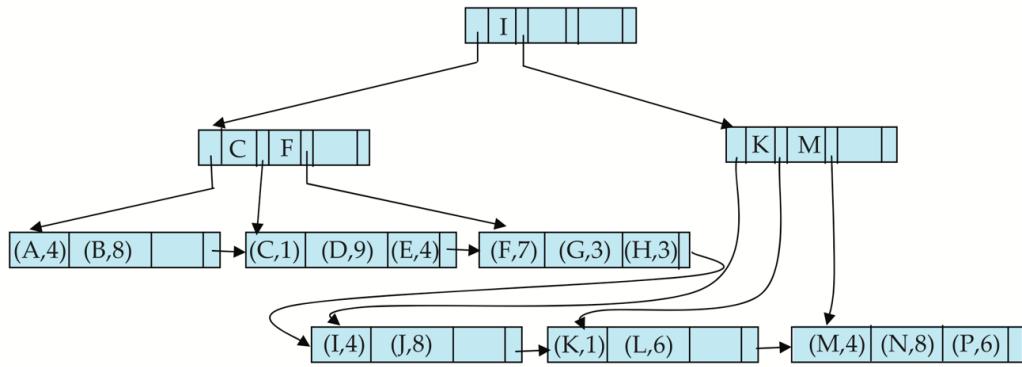
procedure delete(value K, pointer P)
    find the leaf node L that contains (K, P)
    delete_entry(L, K, P)

procedure delete_entry(node N, value K, pointer P)
    delete (K, P) from N
    if (N is the root and N has only one remaining child)
        then make the child of N the new root of the tree and delete N
    else if (N has too few values/pointers) then begin
        Let N' be the previous or next child of parent(N)
        Let K' be the value between pointers N and N' in parent(N)
        if (entries in N and N' can fit in a single node)
            then begin /* Coalesce nodes */
                if (N is a predecessor of N') then swap-variables(N, N')
                if (N is not a leaf)
                    then append K' and all pointers and values in N to N'
                    else append all (Ki, Pi) pairs in N to N'; set N'.Pn = N.Pn
                    delete_entry(parent(N), K', N); delete node N
                end
            else begin /* Redistribution: borrow an entry from N' */
                if (N' is a predecessor of N) then begin
                    if (N is a nonleaf node) then begin
                        let m be such that N'.Pm is the last pointer in N'
                        remove (N'.Km-1, N'.Pm) from N'
                        insert (N'.Pm, K') as the first pointer and value in N,
                            by shifting other pointers and values right
                        replace K' in parent(N) by N'.Km-1
                    end
                    else begin
                        let m be such that (N'.Pm, N'.Km) is the last pointer/value
                            pair in N'
                        remove (N'.Pm, N'.Km) from N'
                        insert (N'.Pm, N'.Km) as the first pointer and value in N,
                            by shifting other pointers and values right
                        replace K' in parent(N) by N'.Km
                    end
                end
                else ... symmetric to the then case ...
            end
        end
    
```

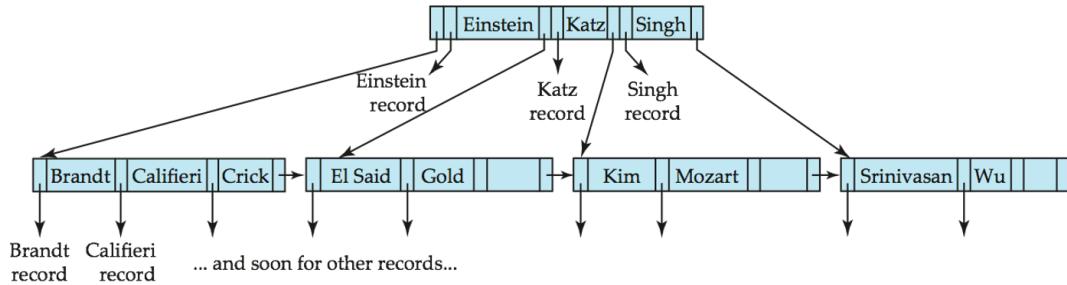
### 8.3.4 其他事项

- 前缀压缩：比如“abcde”和“abds”可以通过“ab”进行区分
- 多码索引：构成元组进行索引

### 8.3.5 B+树文件组织



### 8.3.6 B树



## 8.4 静态散列

散列(hash)函数应该满足：

- 分布均匀：为每个桶分配同样数量的搜索码值
- 分布随机：不应与搜索码的任何外部可见排序相关

两种散列结构：

- 闭散列：一个桶的溢出桶都用链表链接在该桶后面形成溢出链(overflow chaining)
- 开散列：线性探查法(probing)插入到下一个有空间的桶

下例的散列函数为ID各位数字之和对8取模

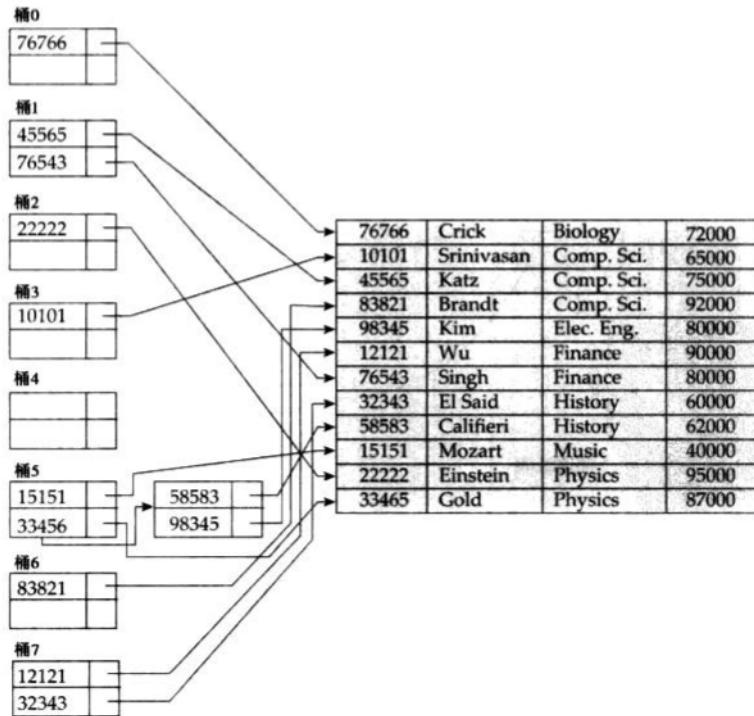


图 11-25 *instructor* 文件中在搜索码 ID 上的散列索引 .

静态散列问题：

- 太小导致后期冲突多性能下降
- 太大则大量空间被浪费

## 8.5 动态散列

当数据库增大或缩小时，可扩充散列(extendable hashing)可通过桶的分裂或合并来适应数据库大小变化。

通常选择一个具有均匀性和随机性特性的散列函数 $h$ ，且产生范围较大，是 $b$ 位二进制整数，通常 $b = 32$ 。通过哈希函数的前*i*位确定索引，每次桶满了才考虑新增一位，且不到万不得已不分裂（原来前缀相同的全指向同一个桶）。但如果散列值相同，则只能采用溢出桶的方法。

<i>dept_name</i>	<i>h(dept_name)</i>
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

图 11-27 *dept\_name* 的散列函数

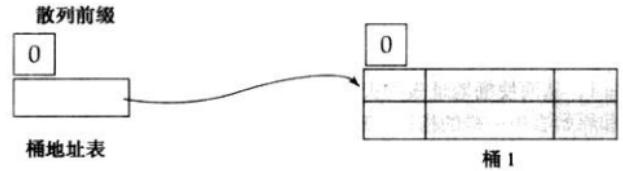
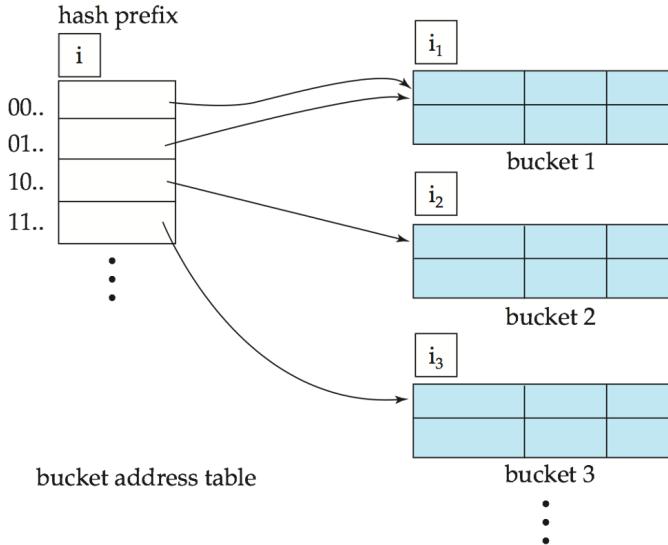


图 11-28 初始的可扩充散列结构

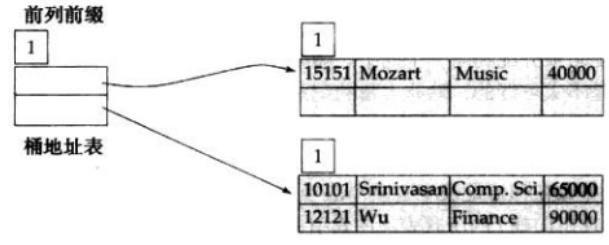


图 11-29 3 次插入操作后的散列结构

但缺点在于查找涉及一个附加的间接层，系统在访问桶本身之前必须先访问桶地址表。

## 8.6 位图索引

一共是元组个数  $n$  位，若第  $i$  个元组的该属性为某特定值，则设为 1，否则置 0。

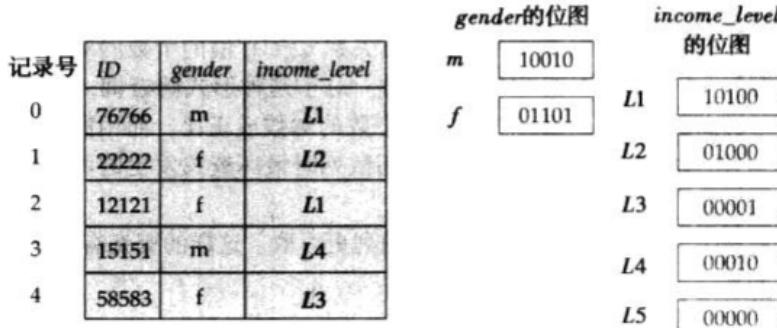


图 11-35 关系 *instructor\_info* 上的位图索引

## 9 事务

### 9.1 基本概念

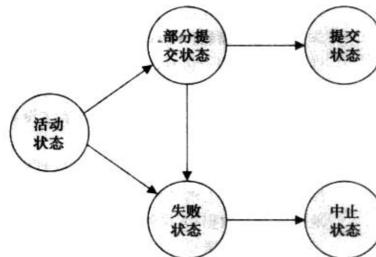
**定义 21** (事务(transaction))。构成单一逻辑工作单元的操作集合称为事务。即使有故障，数据库系统也必须保证数据库的正确执行——要么执行整个事务，要么属于该事务的操作一个也不执行。

事务具有以下的基本性质(ACID):

- 原子性(Atomicity): 要么执行完，要么不执行，不能执行到一半。
- 一致性(Consistency): 除了基本的数据完整性约束，还有更多的一致性约束。
- 隔离性(Isolation): 每个事务都察觉不到系统中有其他事务在并发执行，一定是完成一个再进行下一个。
- 持久性(Durability): 一个事务成功完成对数据库的改变是永久的，即使出现系统故障。

事务的基本状态:

- 活动的(active): 初始状态
- 部分提交的(partially committed): 最后一条语句执行后
- 失败的(failed): 执行出错
- 中止的(aborted): 事务回滚且数据库已恢复到事务开始执行前
- 提交的(committed): 成功完成



## 9.2 可串行化

**定义 22 (调度).** 按照时间顺序执行的一串指令即为调度。如果事务内的指令都不被打乱，如 $\langle T_1, T_2 \rangle$ 或 $\langle T_2, T_1 \rangle$ ，则为串行调度。

**定义 23 (冲突).** 当 $I$ 和 $J$ 是不同事务在相同数据项 $Q$ 上的操作，并且其中至少有一个是write指令时，则称 $I$ 与 $J$ 是冲突的。

**定义 24 (冲突等价(conflict equivalent)).** 如果调度 $S$ 可以经过一系列非冲突指令交换转换为 $S'$ ，则称 $S$ 和 $S'$ 是冲突等价的。若 $S'$ 为串行调度，则 $S$ 为冲突可串行化的(*conflict serializable*)。

$T_1$	$T_2$	$T_1$	$T_2$
read ( $A$ ) write ( $A$ )	read ( $A$ ) write ( $A$ )	read ( $A$ ) write ( $A$ ) read ( $B$ ) write ( $B$ )	
read ( $B$ ) write ( $B$ )	read ( $B$ ) write ( $B$ )		read ( $A$ ) write ( $A$ ) read ( $B$ ) write ( $B$ )

可以将左侧调度转为右侧，故是冲突可串行的。

可以通过构造一个优先图(precedence graph)来判断是否冲突可串行化，顶点集由所有参与调度的事务构成，边集由满足下列三个条件之一的边 $T_i \rightarrow T_j$ 构成：

- 在 $T_j$ 执行read(Q)前,  $T_i$ 执行write(Q)
  - 在 $T_j$ 执行write(Q)前,  $T_i$ 执行read(Q)
  - 在 $T_j$ 执行write(Q)前,  $T_i$ 执行write(Q)

即只有两者都读才不冲突。

如果无环（用环检测算法），则冲突可串行化，可以通过拓扑排序（所有父亲执行完才到孩子执行）得到串行化顺序。

**定义 25 (可恢复调度).** 若事务  $T_j$  读事务  $T_i$  先前写的数据，则  $T_i$  的提交必须在  $T_j$  的提交之前。

### 9.3 隔离性级别

- 可串行化(serializable)
  - 可重复读(repeatable read)
  - 已提交读(read committed)
  - 未提交读(read uncommitted)

10 并发控制

## 10.1 基于锁的协议

- 共享锁(shared): 如果事务 $T_i$ 获得了数据项 $Q$ 上的共享型锁 (记为 $S$ ), 则 $T_i$ 可读但不能写 $Q$
  - 排他锁(exclusive): 如果事务 $T_i$ 获得了数据项 $Q$ 上的排他型锁 (记为 $X$ ), 则 $T_i$ 既可读又可写 $Q$

若某个事务请求的锁与其他事务持有的锁相容，它才可以被授予锁；否则等待所有不相容锁释放。

容易发生如下死锁

表 1: 锁相容性矩阵comp

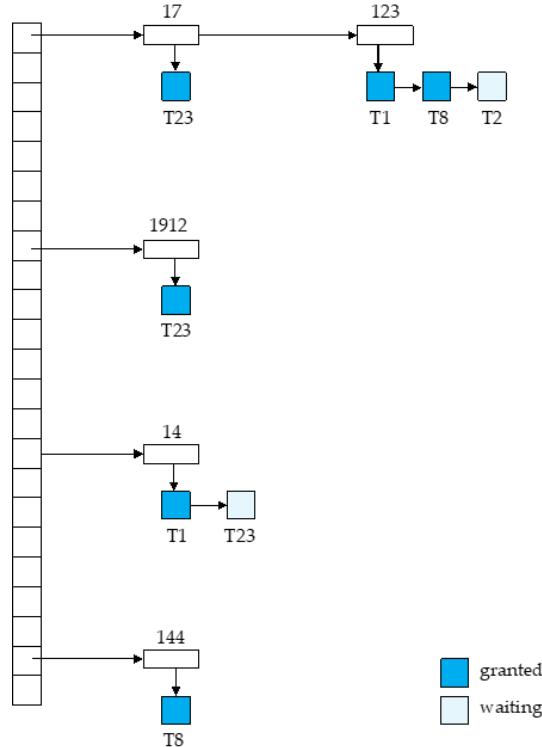
	$S$	$X$
$S$	true	false
$X$	false	false

$T_3$	$T_4$
lock-x ( $B$ ) read ( $B$ ) $B := B - 50$ write ( $B$ ) lock-x ( $A$ )	lock-s ( $A$ ) read ( $A$ ) lock-s ( $B$ )

令 $\{T_0, \dots, T_n\}$ 是参与调度 $S$ 的一个事务集，若存在数据项 $Q$ ，使得 $T_i$ 在 $Q$ 上持有 $A$ 型锁，之后 $T_j$ 在 $Q$ 上持有 $B$ 型锁，且 $comp(A, B) = false$ ，则称在 $S$ 中 $T_i$ 先于 $T_j$ ，记为 $T_i \rightarrow T_j$ ，即在任何等价的串行调度中， $T_i$ 必须出现在 $T_j$ 之前。

两阶段封锁协议：可保证冲突可串行化，但不能保证不发生死锁，对于每个事务有以下两个阶段

- 增长(growing)阶段：事务可以获得锁，但不能释放锁（也包括锁的升级，共享→排他）
- 缩减(shrinking)阶段：事务可以释放锁，但不能获得新锁（也包括锁的降级，排他→共享）



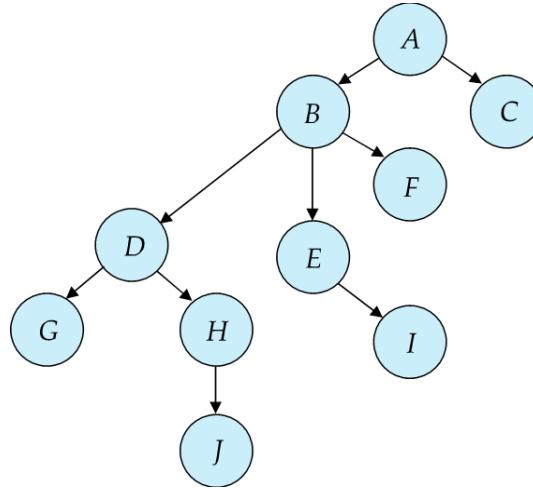
自动为事务产生适当的加锁、解锁指令：

- 事务 $T_i$ 进行 $read(Q)$ 操作时，系统产生一条 $lock - S(Q)$ 指令，该 $read(Q)$ 指令紧跟其后

- 事务 $T_i$ 进行 $write(Q)$ 操作时，系统检查 $T_i$ 是否已在 $Q$ 上持有共享锁。若有，则系统发出 $upgrade(Q)$ 指令，后接 $write(Q)$ 指令。否则系统发出 $lock - X(Q)$ 指令，后接 $write(Q)$ 指令。
- 当一个事务提交或中止后，该事务持有的所有锁都被释放。

## 10.2 基于图的协议

要求所有数据项集合 $D = \{d_1, d_2, \dots, d_n\}$ 满足偏序 $\rightarrow$ ：如果 $d_i \rightarrow d_j$ ，则任何既访问 $d_i$ 又访问 $d_j$ 的事务必须首先访问 $d_i$ ，然后访问 $d_j$ 。偏序意味着集合 $D$ 可以视为有向无环图，称为数据库图。



在树形协议中，可用的加锁指令只有 $lock-X$ 。每个事务 $T_i$ 对一数据项最多能加一次锁，并且遵从以下规则：

1.  $T_i$ 首次加锁可以对任何数据项进行
2. 此后， $T_i$ 对数据项 $Q$ 加锁的前提是 $T_i$ 当前持有 $Q$ 的父项上的锁
3. 对数据项解锁可以随时进行
4. 数据项被 $T_i$ 加锁并解锁后， $T_i$ 不能再对该数据项加锁

所有满足树形协议的调度都是冲突可串行化的，且保证不会发生死锁。

但是不满足可恢复性，而且需要给不需要访问的数据也上锁，增加上锁开销，降低并行程度。

## 10.3 死锁处理

### 10.3.1 死锁预防

- 每个事务都要在它执行前锁上它所有要用的数据（预声明）
- 强加偏序条件（基于图的协议）

更多方法包括：

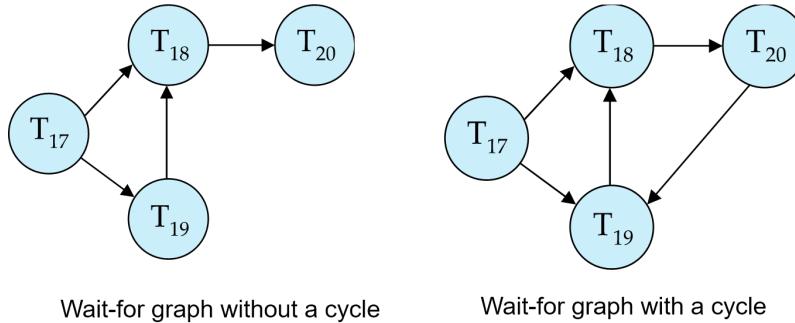
- wait-die机制，非抢占(non-preemptive)技术：老的事务（ $T_i$ 时间戳小于 $T_j$ 时间戳，则 $T_i$ 老于 $T_j$ ）会等待年轻的释放数据，年轻的事务不会等待老的事务，直接回滚

- wound-wait机制，抢占技术：老的事务强行令年轻事务回滚而不让其等待，年轻事务只能等待老的事务

另外还有锁超时(lock timeout)技术，即申请锁的事务至多等待一定时间，若此时间内未授予该事务锁，则事务超时回滚重启。

### 10.3.2 死锁检测

等待(wait-for)图顶点由所有事务组成，边 $T_i \rightarrow T_j$ 代表事务 $T_i$ 等待 $T_j$ 释放数据项。



当事务 $T_i$ 申请的数据项当前被 $T_j$ 持有时， $T_i \rightarrow T_j$ 被插入图中。只有当事务 $T_j$ 不再持有事务 $T_i$ 所需数据项时，这条边才从等待图删除。

当且仅当等待图存在环时，系统存在死锁。要检测死锁，系统需要维护等待图，并周期性激活在等待图中搜索环的算法。

### 10.3.3 死锁恢复

通常做的动作有三个：

- 选择牺牲者：决定回滚某一最小代价的事务
- 回滚：一旦确定回滚事务，则需确定该事务回滚多远，包括彻底回滚和部分回滚
- 饿死：如果选择牺牲者主要基于代价，则有可能同一事务总是被选为牺牲者，那该事务始终不能完成任务，就饿死(starvation)了

## 10.4 基于时间戳的协议

若事务 $T_i$ 已赋予时间戳 $TS(T_i)$ ，此时有一新事务 $T_j$ 进入系统，则 $TS(T_i) < TS(T_j)$ ，可利用系统时钟或者逻辑计数器实现。事务的时间戳决定了串行化顺序，因此若 $TS(T_i) < TS(T_j)$ ，则系统必须保证所产生的串行调度等价于事务 $T_i$ 出现在事务 $T_j$ 之前的某个串行调度。

- 若 $TS(T_i) \geq W\text{-timestamp}(Q)$ ，则执行读操作
- 若 $TS(T_i) \geq R/W\text{-timestamp}(Q)$ ，则执行写操作
- 其他情况都会导致回滚

# 11 恢复系统

## 11.1 故障分类

- 事务故障(failure)
  - 逻辑错误：事务由于某些内部条件而无法继续正常执行，如非法输入。
  - 系统错误：系统进入一种不良状态（如死锁），事务无法正常运行，但是之后某个时间点能够重新执行。
- 系统崩溃(crash)：硬件故障，或数据库软件/操作系统的漏洞，导致易失性存储器内容丢失，事务停止。
- 磁盘故障(failure)：数据传送过程中由于磁头损坏或故障造成的磁盘块上内容丢失。

恢复算法包括两个部分：

- 在正常事务执行过程中做的动作，获得之后能够从错误中恢复的信息
- 在数据库发生故障时执行的动作，使得恢复后可以确保原子性、一致性和持久性

## 11.2 恢复与原子性

记录以下操作（所有日志都应该在操作前被写入）

- 当一个事务 $T_i$ 开始时，它会记录一个日志 $\langle T_i \text{ start} \rangle$ 。
- 在每一个写操作 $write(X)$ 之前，记录日志 $\langle T_i, X, V_1, V_2 \rangle$ ，其中 $V_1$ 是旧值， $V_2$ 是新值。
- 当 $T_i$ 完成它最后一条指令时，记录日志 $\langle T_i \text{ commit} \rangle$ 被写入。只有当提交日志被写入稳定存储，这个事务才能被称为已提交。

事务回滚操作：

1. 从后往前扫描日志，对于所发现的 $T_i$ 的每个形如 $\langle T_i, X_j, V_1, V_2 \rangle$ 的日志记录：
  - (a) 值 $V_1$ 被写到数据项 $X_j$ 中，且
  - (b) 往日志中写一个特殊的只读日志记录 $\langle T_i, X_j, V_1 \rangle$ ，其中 $V_1$ 是在本次回滚中数据项 $X_j$ 恢复成的值。这种日志称作补偿日志记录(compensation log record)。
2. 一旦发现 $\langle T_i, start \rangle$ 的日志记录，就停止从后往前扫描，并往日志中写一个 $\langle T_i, abort \rangle$ 的日志记录

使用日志来重做(redo)和撤销(undo)事务：

- Redo指写入新值 $V_2$
- Undo指写回旧值 $V_1$

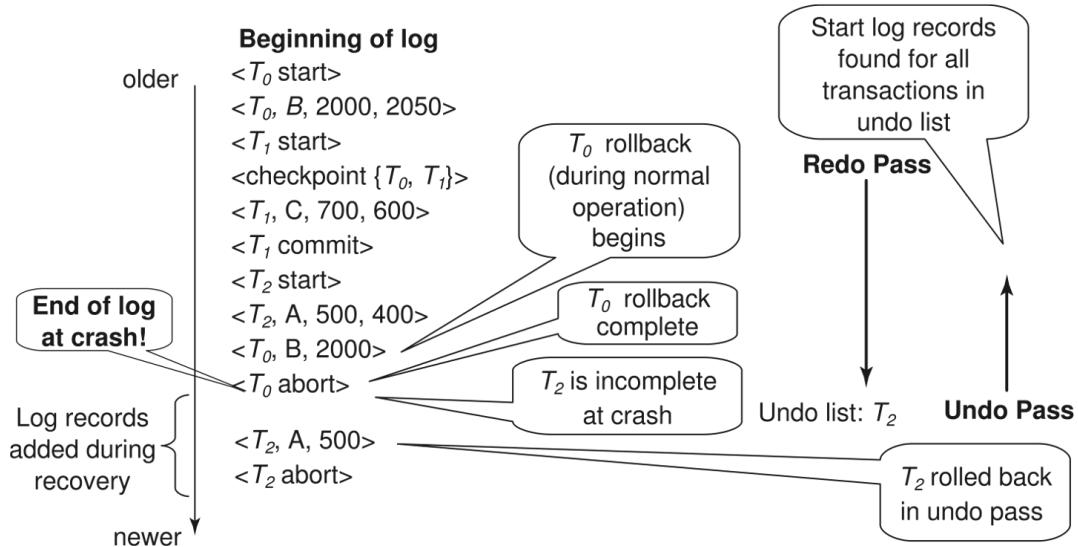
Redo阶段：

- 找到最后一个检查点 $\langle \text{checkpoint } L \rangle$ ，设置undo-list为 $L$
- 从上面的检查点做前向扫描，

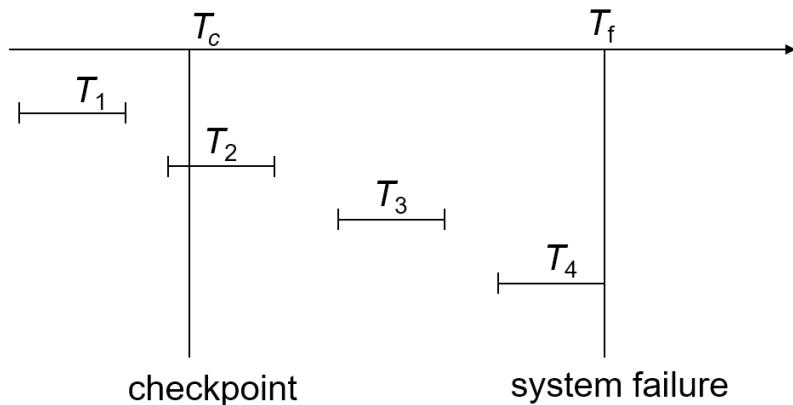
- 若  $\langle T_i, X_j, V_1, V_2 \rangle$  被发现，则重做，将  $V_2$  写到  $X_j$
- 当  $\langle T_i \text{ start} \rangle$  被发现，则添加  $T_i$  到 undo-list 中
- 当  $\langle T_i \text{ commit} \rangle$  或  $\langle T_i \text{ abort} \rangle$  被发现，从 undo-list 中移除  $T_i$

Undo阶段：从后面往前扫日志

- 若  $\langle T_i, X_j, V_1, V_2 \rangle$  被发现且  $T_i$  在 undo-list 中，则
  - 执行 undo，将  $V_1$  写到  $X_j$
  - 写日志  $\langle T_i, X_j, V_1 \rangle$
- 当  $\langle T_i \text{ start} \rangle$  被发现且  $T_i$  在 undo-list 中，则
  - 写日志  $\langle T_i \text{ abort} \rangle$
  - 将  $T_i$  从 undo-list 中移除
- 当 undo-list 为空时停止，即  $\langle T_i \text{ start} \rangle$  在 undo-list 中被每一个事务找到



但每个操作都要写日志就很慢，因此采用检查点(checkpoint)进行打包。



其中，

- $T_1$  可被忽略，因为已经写入磁盘
- 检查点是  $T_c$ ，故  $T_2$  和  $T_3$  需要 Redo
- 而  $T_4$  需要 undo