

数据库复习笔记

《数据库系统概念 Database System Concepts 第六版》机械工业出版社

制作人：匡乾 Sun Yat-sen University School of Data and Computer Science

前言：

某些不在考试范围内的内容没有做，个人认为不重要的内容没有做，仅供参考

第一部分

第一章 引言

- 1.1 数据库系统的应用
- 1.3 数据视图
 - 1.3.1 数据抽象
 - 1.3.2 实例和模式
 - 1.3.3 数据模型
- 1.4 数据库语言
 - 1.4.1 数据操纵语言(DML)
 - 1.4.2 数据定义语言(DDL)
- 1.5 关系数据库
 - 1.5.1 表
 - 1.5.2 数据操纵语言(DML)
 - 1.5.3 数据定义语言(DDL)
- 1.6 数据库设计
 - 1.6.1 设计过程
 - 1.6.3 实体-联系模型(E-R模型)
 - 1.6.4 规范化
- 1.7 数据存储和查询
 - 1.7.1 存储管理器
 - 1.7.2 查询处理器
- 1.8 事务
- 1.12 数据库用户和管理员
 - 1.12.1 数据库用户
 - 1.12.2 数据库管理员
- 1.14 总结

第二章 关系模型介绍

- 2.1 关系数据库的结构
- 2.2 数据库模式
- 2.3 码
- 2.4 模式图
- 2.5 关系查询语言
- 2.6 关系运算
- 2.7 总结

第三章 初级SQL

- 3.1 SQL查询语言概览
- 3.2 SQL数据定义
 - 3.2.2 基本模式定义
- 3.3 SQL查询的基本结构
- 3.4 附加的基本运算

- 3.4.1 更名运算
 - 3.4.2 字符串运算
- 3.5 集合运算
- 3.6 空值
- 3.7 聚集函数
 - 3.7.2 分组聚集
 - 3.7.3 having子句
- 3.8 嵌套子查询
 - 3.8.1 集合成员资格
 - 3.8.2 集合的比较
 - 3.8.3 空关系测试
 - 3.8.5 from子句中的子查询
 - 3.8.6 with子句
- 3.9 数据库的修改
 - 3.9.1 删除
 - 3.9.2 插入
 - 3.9.3 更新
- 3.10 总结
- 第四章 中级SQL
 - 4.1 连接表达式
 - 4.1.2 外连接
 - 4.2 视图
 - 4.2.1 视图定义
 - 4.2.2 SQL查询中使用视图
 - 4.2.3 物化视图
 - 4.2.4 视图更新
 - 4.3 事务
 - 4.4 完整性约束
 - 4.4.2 not null 约束
 - 4.4.3 unique 约束
 - 4.4.4 check 子句
 - 4.4.5 参照完整性
 - 4.4.7 断言
 - 4.5 SQL的数据类型与模式
 - 4.5.2 默认值
 - 4.5.3 创建索引
 - 4.6 授权
 - 4.7 总结
- 第五章 高级SQL
 - 5.3 触发器
- 第六章 形式化关系查询语言
 - 6.1 关系代数
 - 6.1.1 基本运算
 - 6.1.1.1 选择运算
 - 6.1.1.2 投影运算
 - 6.1.1.3 关系运算的组合
 - 6.1.1.4 并运算
 - 6.1.1.5 集合差运算
 - 6.1.1.6 笛卡尔积运算
 - 6.1.1.7 更名运算
 - 6.1.3 附加的关系代数运算
 - 6.1.3.1 集合交运算
 - 6.1.3.2 自然连接运算

- 6.1.3.3 赋值运算
 - 6.1.3.4 外连接运算
 - 6.1.4 扩展的关系代数运算
 - 6.1.4.1 广义投影
 - 6.1.4.2 聚集
- 6.2 元组关系演算
 - 6.2.1 查询示例
 - 6.2.2 形式化定义
 - 6.2.3 表达式的安全性
- 6.3 域关系演算
 - 6.3.1 形式化定义
 - 6.3.2 查询的例子
- 6.4 总结

第二部分 数据库设计

第七章 数据库设计和E-R模型

- 7.1 设计过程概览
 - 7.1.1 设计阶段
 - 7.1.2 设计选择
- 7.2 实体-联系模型
 - 7.2.1 实体集
 - 7.2.2 联系集
 - 7.2.3 属性
- 7.3 约束
 - 7.3.1 映射基数
 - 7.3.2 参与约束
 - 7.3.3 码
- 7.5 实体-联系图
 - 7.5.1 基本结构
 - 7.5.2 映射基数
 - 7.5.3 复杂的属性
 - 7.5.6 弱实体集
 - 7.5.7 大学的E-R图
- 7.7 实体-联系设计问题
 - 7.7.1 用实体集还是用属性
 - 7.7.2 用实体集还是用联系集
 - 7.7.3 二元还是n元联系集
 - 7.7.4 联系属性的布局
- 7.8 扩展的E-R特性
 - 7.8.1 特化
 - 7.8.2 概化
 - 7.8.3 属性继承
 - 7.8.4 概化上的约束
 - 7.8.5 聚集
- 7.11 总结

第八章 关系数据库设计

- 8.2 原子域和第一范式
- 8.3 使用函数依赖进行分解
 - 8.3.1 码和函数依赖
 - 8.3.2 Boyce-Codd 范式 (BCNF)
 - 8.3.4 第三范式
- 8.4 函数依赖理论
 - 8.4.1 函数依赖集的闭包
 - 8.4.2 属性集的闭包

- 8.4.3 正则覆盖
- 8.4.4 无损分解
- 8.4.5 保持依赖
- 8.5 分解算法
 - 8.5.1 BCNF分解
 - 8.5.1.1 BCNF的判定方法
 - 8.5.1.2 BCNF分解算法
 - 8.5.2 3NF分解
 - 8.5.3 3NF算法的正确性
 - 8.5.4 BCNF和3NF的比较
- 8.6 使用多值依赖的分解
- 8.8 数据库设计过程
- 8.10 总结

第三部分 数据存储和查询

第十章 存储和文件结构

- 10.1 物理存储介质概述
- 10.2 磁盘和快闪存储器
 - 10.2.2 磁盘性能的度量
 - 10.2.3 磁盘块访问的优化
- 10.3 RAID
 - 10.3.1 通过冗余提高可靠性
 - 10.3.2 通过并行提高性能
 - 10.3.3 RAID级别
- 10.5 文件组织
 - 10.5.1 定长记录
 - 10.5.2 变长记录
- 10.6 文件中记录的组织
 - 10.6.1 顺序文件组织
 - 10.6.2 多表聚簇文件组织
- 10.7 数据字典存储
- 10.9 总结

第十一章 索引与散列

- 11.1 基本概念
- 11.2 顺序索引
 - 11.2.1 稠密索引和稀疏索引
 - 11.2.2 多级索引
 - 11.2.3 索引的更新
 - 11.2.4 辅助索引
 - 11.2.5 多码上的索引
- 11.5 多码访问
 - 11.5.1 使用多个单码索引
 - 11.5.2 多码索引
 - 11.5.3 覆盖索引
- 11.6 静态散列
 - 11.6.1 散列函数
 - 11.6.2 桶溢出处理
 - 11.6.3 散列索引
- 11.7 动态散列
 - 11.7.1 数据结构
 - 11.7.2 查询和更新
 - 11.7.3 静态散列与动态散列比较
- 11.8 顺序索引和散列的比较
- 11.9 位图索引

11.9.1 位图索引结构

11.11 总结

第十二章 查询处理

12.1 概述

12.2 查询代价的度量

12.3 选择运算

12.3.1 使用文件扫描和索引的选择

12.4 排序

12.4.1 外部排序归并算法

12.4.2 外部排序归并的代价分析

12.5 连接运算

12.5.1 嵌套循环连接

12.5.2 块嵌套循环连接

12.5.3 索引嵌套循环连接

12.6 其他运算

12.6.1 去除重复

12.6.2 投影

12.7 表达式计算

12.7.1 物化

12.7.2 流水线

12.7.2.1 流水线的实现

12.8 总结

第十三章 查询优化

13.1 概述

13.2 关系表达式的转换

13.2.1 等价规则

13.2.2 转换的例子

13.2.3 连接的次序

13.2.4 等价表达式的枚举

13.3 表达式结果集统计大小的估计

13.3.1 目录信息

13.3.2 选择运算结果大小的估计

13.3.3 连接运算结果大小的估计

13.3.4 其它运算的结果集大小的估计

13.4 执行计划选择

13.4.1 基于代价的连接顺序选择

13.4.3 启发式优化

13.7 总结

第四部分 事务管理

第十四章 事务

14.1 事务概念

14.2 一个简单的事务模型

14.3 存储结构

14.4 事务原子性和持久性

14.5 事务隔离性

14.6 可串行化

14.7 事务隔离性和原子性

14.7.1 可恢复调度

14.7.2 无级联调度

14.8 事务隔离性级别

14.9 隔离性级别的实现

14.11 总结

第十五章 并发控制

- 15.1 基于锁的协议
 - 15.1.1 锁
 - 15.1.2 锁的授予
 - 15.1.3 两阶段封锁协议
 - 15.1.4 封锁的实现
 - 15.1.5 基于图的协议
- 15.2 死锁处理
 - 15.2.1 死锁预防
 - 15.2.2 死锁检测与恢复
 - 15.2.2.1 死锁检测
 - 15.2.2.2 从死锁中恢复
- 15.3 多粒度
- 15.4 基于时间戳的协议
 - 15.4.1 时间戳
 - 15.4.2 时间戳排序协议
 - 15.4.3 Thomas写规则
 - 15.4.4 视图可串行化的
- 15.5 基于有效性检查的协议
- 15.6 多版本机制
- 15.7 快照隔离
- 15.11 总结
- 第十六章 恢复系统
 - 16.1 故障分类
 - 16.2 存储器
 - 16.2.1 稳定存储器的实现
 - 16.2.2 数据访问
 - 16.3 恢复与原子性
 - 16.3.1 日志记录
 - 16.3.2 数据库修改
 - 16.3.3 并发控制和恢复
 - 16.3.5 使用日志来重做和撤销事务
 - 16.3.6 检查点
 - 16.4 恢复算法
 - 16.4.1 事务回滚
 - 16.4.2 系统崩溃后的恢复
 - 16.5 缓冲区管理
 - 16.5.1 日志记录缓冲
 - 16.5.2 数据库缓冲
 - 16.10 总结

第一部分

第一章 引言

1.1 数据库系统的应用

数据库管理系统(DataBase-Management System, DBMS)

主要目标：提供一种**方便、高效地**存取数据库信息的途径

文件处理系统中存储组织信息的主要弊端：

- 数据的冗余和不一致
- 数据访问困难
- 数据孤立
- 完整性问题
- 原子性问题
- 并发访问异常
- 安全性问题

1.3 数据视图

1.3.1 数据抽象

- 物理层
最低层次的抽象，描述数据是怎样存储的
- 逻辑层
逻辑层用户不必知道物理层结构，保证了物理数据*独立性*
- 视图层

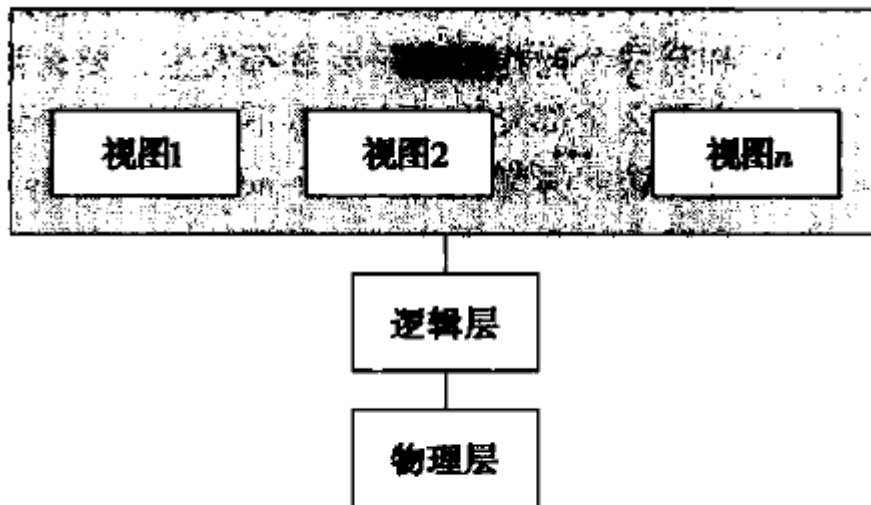


图 1-1 数据抽象的三个层次

1.3.2 实例和模式

实例(instance): 特定时刻数据库中的信息集合

模式(schema): 数据库的总体设计（不频繁发生改变）

物理模式: 在物理层描述数据库的设计

逻辑模式: 在逻辑层描述数据库的设计

1.3.3 数据模型

数据模型: 数据库结构的基础

- 关系模型
用表的集合来表示数据和数据间的关系

- **实体-联系模型(E-R)**

现实世界由一组称作实体的基本对象以及这些对象间的联系构成

- 基于对象的数据模型
- 半结构化数据模型

1.4 数据库语言

数据定义语言(Data-Definition Language): 定义数据库模式

数据操纵语言(Data-Manipulation Language): 表达数据库的查询和更新

1.4.1 数据操纵语言(DML)

- 过程化DML: 要求用户指定需要什么数据以及如何获得这些数据
- 声明式DML (非过程化DML) : 只要求用户指定需要什么数据

查询(query): 对信息进行检索的语句

[返回](#)

1.4.2 数据定义语言(DDL)

数据库中的数据值必须满足某些**一致性约束**

- 域约束
每个属性都有值域
- 参照完整性
一个关系中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
- 断言
数据库需要时刻满足的某一条件
- 授权
对用户加以区别 (读权限、插入权限、更新权限、删除权限)

DDL的输出放在**数据字典**中, 数据字典包含了**元数据**, 元数据是关于数据的数据

1.5 关系数据库

1.5.1 表

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

b) *departments*表

1.5.2 数据操纵语言(DML)


```
select instructor. name
from instructor
where instructor. dept_name = 'History' ;
```

1.5.3 数据定义语言(DDL)

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

1.6 数据库设计

数据库设计的主要内容是数据库模式的设计

1.6.1 设计过程

1. 制定出用户需求的规格文档
2. 概念设计阶段，将需求转换为数据库的概念模式
3. 逻辑设计阶段
4. 物理设计阶段

1.6.3 实体-联系模型(E-R模型)

实体通过属性集合来描述

联系是几个实体之间的关联

实体集、联系集：同一类型所有 实体 / 联系的集合

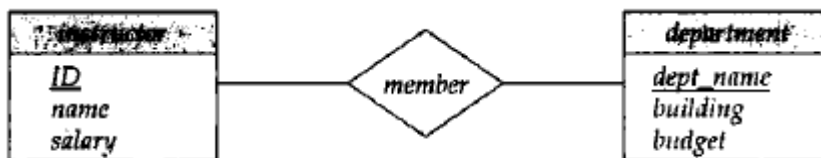


图 1-3 E-R 图示例

1.6.4 规范化

目标：

- 1、没有不必要的冗余
- 2、能轻易地检索数据

使用函数依赖设计范式

1.7 数据存储和查询

数据库系统的功能部件可分为 存储管理器和 查询处理部件

1.7.1 存储管理器

存储管理部件包括：

- 权限及完整性管理器

- 事务管理器
- 文件管理器
- 缓冲区管理器

数据结构：

- 数据文件：存储数据库本身
- 数据字典：存储关于数据库结构的元数据（数据库模式）
- 索引：提供对数据项的快速访问

1.7.2 查询处理器

查询处理器组件包括：

- DDL解释器：解释DDL语句并将定义记录在数据字典中
- DML编译器：将查询语言翻译为一个执行方案
- 查询执行引擎：执行由DML编译器产生的低级指令

1.8 事务

事务是数据库应用中完成单一逻辑功能的操作集合

其具有*原子性、一致性、持久性*

恢复管理器保证数据库系统的原子性和持久性

并发控制管理器控制并发事务间的相互影响，保证数据库的一致性

事务管理器包括并发控制管理器和恢复管理器

1.12 数据库用户和管理员

1.12.1 数据库用户

数据库系统的用户可以分为四种不同类型：

- 无经验的用户
- 应用程序员
- 老练的用户
- 专门的用户

1.12.2 数据库管理员

数据库管理员(DataBase Administrator, DBA)的作用包括：

- 模式定义
- 存储结构及存取方法定义
- 模式及物理组织的修改
- 数据访问授权
- 日常维护

1.14 总结

见书本P18

第二章 关系模型介绍

2.1 关系数据库的结构

关系数据库由表的结合构成

关系 用来指代表，元组 用来指代行，属性 用来指代表中的列

course_id	title	dept_name	credits
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

图 2-2 course 关系

关系实例表示一个关系的特定的实例，即所包含的一组特定的行

域：关系中的属性允许取值的集合

若域中元素被看作是不可再分的单元，则域是原子的

空值(null)：一个特殊的值，表示值未知或不存在

2.2 数据库模式

数据库模式是数据库的逻辑设计

数据库实例是特定时刻数据库中数据的一个快照

关系模式对应与程序设计中的类型定义

dept_name	building	budget
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

图 2-5 department 关系

department关系的模式：

department (dept_name , building , budget)

2.3 码

超码：一个或多个属性的集合，可以使我们在一个关系中唯一地标识一个元组

候选码：最小的超码

主码：被设计者选中用来在一个关系中区分不同元组的**候选码**

r_1 在属性中包括 r_2 的主码，这个属性在 r_1 上被称作参照 r_2 的**外码**

关系 r_1 称为外码依赖的**参照关系**，关系 r_2 称作外码的**被参照关系**

参照完整性约束：要求参照关系的元组在特定属性上的取值等于被参照关系中某个元组在该属性上的取值

2.4 模式图

大学组织的模式图。每一个关系用一个矩形来表示，关系的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。外码依赖用从参照关系的外码属性到被参照关系的主码属性之间的箭头来表示。

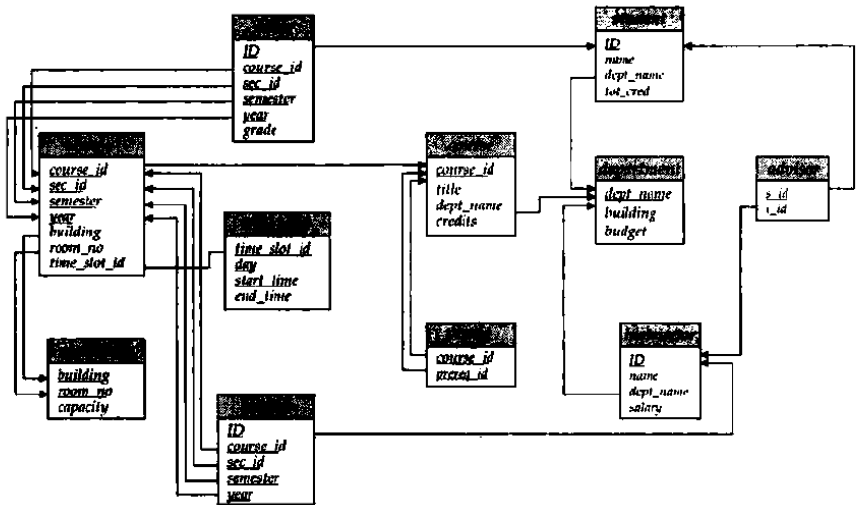


图 2-8 大学数据库的模式图

2.5 关系查询语言

查询语言：请求获取数据库信息的语言

过程化语言、非过程化语言 [见1.4.1](#)

2.6 关系运算

自然连接：两个关系上所匹配的元组在两个关系共有的所有属性上取值**相同**

笛卡尔积运算从两个关系中合并元组，结果包含两个关系元组的**所有对**，不论属性值是否匹配

第6章将详细介绍关系代数，因此此处不再概述

2.7 总结

见P28

第三章 初级SQL

3.1 SQL查询语言概览

SQL语言有以下几个部分：

- 数据定义语言(DDL)：定义关系模式、删除关系、修改关系模式
- 数据操纵语言(DML)：从数据库中查询信息、在数据库中插入元组、删除元组、修改元组
- 完整性：完整性约束
- 视图定义
- 事务控制
- 嵌入式SQL和动态SQL
- 授权：DDL可以定义对关系和视图的访问权限

3.2 SQL数据定义

3.2.2 基本模式定义

create table命令的通用形式：

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

create table命令的例子：

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

三个基本的完整性约束：

- **primary key**(*A*₁, *A*₂, ...): 属性 (*A*₁, *A*₂, ...) 构成关系的主码
- **foreign key**(*B*₁, *B*₂, ...) **references** *s*: 属性 (*B*₁, *B*₂, ...) 上的取值必须对应于关系*s*中某元组在主码属性上的取值
- **not null**: 该属性上不允许空值

3.3 SQL查询的基本结构

自然连接运算作用于两个关系，并产生一个关系作为结果。自然连接只考虑哪些在两个关系模式中都出现的属性上取值相同的元组对。

```
select name, course_id
from instructor natural join teachers;
```

3.4 附加的基本运算

3.4.1 更名运算

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

这种用as实现的重命名，被称作**表别名**或者是**相关名称**

3.4.2 字符串运算

```
select dept_name
from department
where building like '% Watson%';
-- 找出任意所在建筑名称中包含子串'Watson'的所有系名
```

字符串使用like操作符来实现模式匹配。用两个特殊字符来描述模式：

- 百分号(%): 匹配任意字符串
- 下划线(_): 匹配任意一个字符

3.5 集合运算

union, intersect, except运算分别对应于集合论中的 \cup (并), \cap (交), $-$ (差)运算

```
( select course_id
  from section
  where semester = ' Fall' and year = 2009 )
union
( select course_id
  from section
  where semester = ' Spring' and year = 2010 );
```

3.6 空值

空值给关系运算带来了特殊的问题，包括算术运算、比较运算和集合运算

- 算术运算
如果算术表达式任一输入为空，则该算术表达式结果为空
- 比较运算
 $1 < \text{null} = \text{unknown}$
- 布尔运算

运算符	true R unknown	false R unknown	unknown R unknown
and	unknown	false	unknown
or	true	unknown	unknown

not unknown = unknown

3.7 聚集函数

聚集函数是以值的一个集合为输入、返回单个值的函数

常见聚集函数：avg, min, max, sum, count

3.7.2 分组聚集

```
select dept_name, avg(salary) as avg_salary
from instructor
group by dept_name;
-- 找出每个系的平均工资
```

查询结果：

dept_name	avg_salary
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

出现在select语句中但没有被聚集的属性只能是出现在group by子句中的那些属性

3.7.3 having子句

having子句中的谓词在形成分组后才起作用

```
select dept_name, avg(salary) as avg_salary
from instructor
group by dept_name
having avg(salary) > 42000
-- 找出每个平均工资大于42000的系
```

3.8 嵌套子查询

子查询是嵌套在另一个查询中的select-from-where表达式。

3.8.1 集成员资格

连接词in/not in测试元组是否是集合中的成员

```
select distinct course_id
from section
where semester = 'Fall' and year = 2009 and
       course_id in (select course_id
                     from section
                     where semester = 'Spring' and year = 2010);
```

3.8.2 集合的比较

集合的比较需要用到比较运算符以及some/all关键词

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

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

3.8.3 空关系测试

exists结构在作为参数的子查询非空时返回true值

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

3.8.5 from子句中的子查询

任何select-from-where表达式返回的结果都是关系，因而可以被插入到另一个select-from-where中任何关系可以出现的位置

```
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;
```

3.8.6 with子句

with子句提供定义临时关系的方法，这个定义只对包含with子句的查询有效

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

3.9 数据库的修改

3.9.1 删除


```
delete from r
where P;
```

3.9.2 插入

```
insert into course
values('CS-437', 'C++', 'CS', 4)
```

3.9.3 更新

```
update instructor
set salary = salary * 1.05;
```

3.10 总结

见P57

第四章 中级SQL

4.1 连接表达式

4.1.2 外连接

外连接的三种形式：

1. **左外连接(left outer join)**：只保留出现在左边的关系中的元组
2. **右外连接(right outer join)**：只保留出现在右边的关系中的元组
3. **全外连接(full outer join)**：保留出现在两个关系中的元组

ID	name	dept name	tot_cred	course_id	sec_id	semester	year	grade
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2010	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2009	A
19991	Brandt	History	80	HIS-351	1	Spring	2010	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2010	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B+
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A-
70557	Snow	Physics	0	null	null	null	null	null
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	null

图 4-4 student natural left outer join takes 的结果

on和where在外连接中的表现是不同的，on条件是外连接声明的一部分，但where子句却不是

4.2 视图

视图：不是逻辑模型的一部分，但作为虚拟关系对用户可见

4.2.1 视图定义

create view命令的格式为：

```
create view v as <query expression>;
-- <query expression> 可以是任何合法的查询表达式，v表示视图名
```

4.2.2 SQL查询中使用视图

在查询中，视图名可以出现在关系名可以出现的任何地方。

```
select course_id
from physics_fall_2009
where building = 'Watson' ;
```

4.2.3 物化视图

物化视图：如果用于定义视图的实际关系改变，视图也跟着修改

保持物化视图一直在最新状态的过程称为物化视图维护或视图维护

4.2.4 视图更新

一个视图是**可更新的**，如果其满足以下条件：

- from子句中只有一个数据库关系
- select子句只包含关系的属性名，不包含表达式、聚集或distinct声明
- 任何没有出现在select子句中的属性都没有not null约束，同时也不是主码的一部分
- 查询中不含有group by或having子句

```
create view history_instructors as
select *
from instructor
where dept_name = 'History';
```

4.3 事务

事务由查询和更新语句的序列组成

事务结束标志：

- **Commit work**：提交当前事务
- **Rollback work**：回滚当前事务

4.4 完整性约束

4.4.2 not null 约束

not null声明禁止在该属性上插入空值

4.4.3 unique 约束

```
unique(A1,A2, ...,Am)
```

unique声明指出属性A1,A2,...Am形成了一个候选码

4.4.4 check 子句

check(P)子句指定一个谓词P，关系中的每个元组都必须满足谓词P

```
create table department
(dept_name varchar (20),
budget int,
check (budget>0));
```

4.4.5 参照完整性

参照完整性：保证在一个关系中给定属性集上的取值也在另一关系的特定属性集的取值中出现

4.4.7 断言

一个**断言**就是一个谓词，它表达了我们希望数据库总能满足的一个条件

断言为如下形式：`create assertion <assertion-name> check <predicate>;`

4.5 SQL的数据类型与模式

4.5.2 默认值

SQL允许为属性指定默认值

```
tot_cred numeric (3, 0) default 0,
```

4.5.3 创建索引

索引：创建在关系的属性上，允许数据库高效地找到关系中那些在属性上取给定值的元组，而不用扫描关系中的所有元组

创建索引示例：`create index studentID_index on student(ID);`

4.6 授权

对数据的授权包括：

- 授权读取数据
- 授权插入新数据
- 授权更新数据
- 授权删除数据

每种类型的授权都称为一个**权限**

SQL包括select, insert, update, delete权限

grant语句用来授予权限， revoke语句用来收回权限

4.7 总结

见P85

第五章 高级SQL

5.3 触发器

触发器是一条语句，当对数据库做修改时，它自动被系统执行

设置触发器机制的要求：

- 指明什么条件下执行触发器
 - 一个引起触发器被检测的事件
 - 一个触发器执行必须满足的条件
- 指明触发器执行的动作

第六章 形式化关系查询语言

6.1 关系代数

关系代数是一种**过程化**查询语言

关系代数的基本运算有：选择、投影、并、集合差、笛卡尔积、更名

其它运算：集合交、自然连接、赋值

6.1.1 基本运算

一元运算：选择、投影、更名

二元运算：并、集合差、笛卡尔积

6.1.1.1 选择运算

选择(σ)运算选出满足给定谓词的元组

通常，我们允许在选择谓词中进行比较，使用的是： $=, \neq, <, \leq, >, \geq$

另外可以用连词 $and(\wedge)$, $or(\vee)$, $not(\neg)$ 将多个谓词合并为一个较大的谓词

$$\sigma_{dept_name = 'Physics' \wedge salary > 90\ 000}(instructor)$$

6.1.1.2 投影运算

投影(Π)运算返回作为参数的关系

$$\Pi_{ID, name, salary}(instructor)$$

6.1.1.3 关系运算的组合

由于关系代数运算的结果类型仍为关系，因此可以把多个关系代数运算组合成一个关系代数表达式

$$\Pi_{name}(\sigma_{dept_name = 'Physics'}(instructor))$$

6.1.1.4 并运算

$$\Pi_{course_id}(\sigma_{semester = 'Fall' \wedge year = 2009}(section)) \cup \Pi_{course_id}(\sigma_{semester = 'Spring' \wedge year = 2010}(section))$$

要使并运算 $r \cup s$ 有意义（相容），必须满足以下两个条件：

1. 关系 r 和 s 必须是同元的，即它们的属性数目必须相同
2. 对所有的 i ， r 的第 i 个属性的域必须和 s 的第 i 个属性的域相同

6.1.1.5 集合差运算

用 $-$ 表示**集合差**运算表示在一个关系中而不在另一个关系中的那些元组

$$\Pi_{course_id}(\sigma_{semester = 'Fall' \wedge year = 2009}(section)) - \Pi_{course_id}(\sigma_{semester = 'Spring' \wedge year = 2010}(section))$$

集合差运算必须在**相容**的关系间进行（与并运算类似），即关系同元且属性的域相同

6.1.1.6 笛卡尔积运算

笛卡尔积(\times)运算可以将任意两个关系的信息组合在一起

$$\Pi_{name, course_id}(\sigma_{instructor.ID = teaches.ID}(\sigma_{dept_name = 'Physics'}(instructor \times teaches)))$$

6.1.1.7 更名运算

更名(ρ)运算可以给关系赋予名字

更名运算的另一形式如下。假设关系代数表达式 E 是 n 元的, 则表达式

$$\rho_{x(A_1, A_2, \dots, A_n)}(E)$$

返回表达式 E 的结果, 并赋给它名字 x , 同时将各属性更名为 A_1, A_2, \dots, A_n 。

6.1.3 附加的关系代数运算

6.1.3.1 集合交运算

集合交(\cap)运算: $r \cap s = r - (r - s)$

$$\Pi_{\text{course_id}}(\sigma_{\text{semester} = \text{"Fall"} \wedge \text{year} = 2009}(\text{section})) \cap \Pi_{\text{course_id}}(\sigma_{\text{semester} = \text{"Spring"} \wedge \text{year} = 2010}(\text{section}))$$

6.1.3.2 自然连接运算

r 和 s 的自然连接表示为 $r \bowtie s$

$$r \bowtie s = \Pi_{R \cup S}(\sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \dots \wedge r.A_n = s.A_n}(r \times s))$$

其中 $R \cap S = \{A_1, A_2, \dots, A_n\}$ 。

6.1.3.3 赋值运算

赋值(\leftarrow)运算可以给临时关系变量赋值

$$\text{temp1} \leftarrow R \times S$$

$r \bowtie s$ 可以表示为: $\text{temp2} \leftarrow \sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \dots \wedge r.A_n = s.A_n}(\text{temp1})$

$$\text{result} = \Pi_{R \cup S}(\text{temp2})$$

6.1.3.4 外连接运算

外连接运算是连接运算的扩展, 它在结果中创建带空值的元组, 以此来保留在连接中丢失的那些元组

外连接运算有三种形式:

- 左外连接: $= \bowtie \text{\textcolor{red}{\backslash leftouterjoin}}$
取出左侧关系中所有与右侧关系的任一元组都不匹配的元组, 用空值填充所有来自右侧关系的属性
- 右外连接: $\bowtie = \text{\textcolor{red}{\backslash rightouterjoin}}$
与左外连接相反
- 全外连接: $= \bowtie = \text{\textcolor{red}{\backslash fullouterjoin}}$
既做左外连接也做右外连接

6.1.4 扩展的关系代数运算

6.1.4.1 广义投影

广义投影允许在投影列表中使用算数运算和字符串函数来对投影进行扩展

$$\Pi_{ID, name, dept_name, salary/12}(instructor)$$

6.1.4.2 聚集

聚集函数：输入值的一个汇集，将单一值作为返回值

$$\mathcal{G}_{count_distinct}(ID)(\sigma_{semester = "Spring" \wedge year = 2010}(teaches)) \quad dept_name \mathcal{G}_{average}(salary)(instructor)$$

聚集运算 \mathcal{G} (花体g)通常的形式： $\mathcal{G}_{c_1, c_2, \dots, c_n} \mathcal{G}_{F_1(A_1), F_2(A_2), \dots, F_n(A_n)}(E)$

6.2 元组关系演算

代数表达式产生过程序列，而**元组关系演算**是**非过程化**的

查询表达式为： $\{t | P(t)\}$

6.2.1 查询示例

用这种记法，我们可以将查询“找出工资大于 80 000 美元的所有教师的 ID”表述为：

$$\{t | \exists s \in instructor(t[ID] = s[ID] \wedge s[salary] > 80000)\}$$

6.2.2 形式化定义

如果元组变量不被 \exists 或 \forall 修饰，这称为**自由变量**，否则称为**受限变量**

$$t \in instructor \wedge \exists s \in department(t[dept_name] = s[dept_name])$$

t 是自由变量，而元组变量 s 称为受限变量。

元组关系演算的公式由**原子**构成，原子可以是一下形式之一：

- $s \in r$: s 是元组变量， r 是关系
- $s[x] \Theta u[y]$: 其中 s 和 u 是元组变量， x 和 y 是属性， Θ 是比较运算符（如： $<$ ）
- $s[x] \Theta c$: 其中 s 是元组变量， x 是属性， c 是常量， Θ 是比较运算符

6.2.3 表达式的安全性

P 的**域** $dom(P)$ 是 P 所引用的所有值的集合

6.3 域关系演算

关系演算的另一种形式称为**域关系演算**，使用从属性域中取值的域变量，而不是整个元组的值

6.3.1 形式化定义

域关系演算中的表达式形式： $\{ \langle x_1, x_2, \dots, x_n \rangle \mid P(x_1, x_2, \dots, x_n) \}$ (x_n 代表域变量)

域关系演算中的原子具有如下形式之一：

- $\langle x_1, x_2, \dots, x_n \rangle \in r$, 其中 r 是 n 个属性上的关系, 而 x_1, x_2, \dots, x_n 是域变量或域常量。
- $x \Theta y$, 其中 x 和 y 是域变量, 而 Θ 是比较运算符 ($<, \leq, =, \neq, >, \geq$)。我们要求属性 x 和 y 所属域可用 Θ 比较。
- $x \Theta c$, 其中 x 是域变量, Θ 是比较运算符, 而 c 是 x 作为域变量的那个属性域中的常量。

6.3.2 查询的例子

- 找出在物理系的所有教师姓名, 以及他们教授的所有课程的 *course_id*:

$$\{ \langle n, c \rangle \mid \exists i, a, s, y (\langle i, c, a, s, y \rangle \in teaches \\ \wedge \exists d, s (\langle i, n, d, s \rangle \in instructor \wedge d = "Physics")) \}$$

6.4 总结

见P140

第二部分 数据库设计

第七章 数据库设计和E-R模型

7.1 设计过程概览

7.1.1 设计阶段

- 完整刻画用户的数据需求
- 选择数据模型
- 概念设计阶段
- 完善概念模式, 指明功能需求 (功能需求规格说明)
- 逻辑设计阶段、物理设计阶段

7.1.2 设计选择

实体指示所有可以明确的个体

设计数据库模式时, 必须避免两个主要缺陷:

- **冗余**: 不好的设计可能会导致重复信息
最大的问题是, 当对一条消息进行更新时, 这条消息的拷贝可能会与其不一致
- **不完整**: 某些方面可能难以甚至无法建模

7.2 实体-联系模型

E-R数据模型采用三个基本概念: 实体集、联系集、属性

7.2.1 实体集

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”

实体集是相同类型具有相同性质 (属性) 的一个实体集合

实体通过一组**属性**来表示

每个实体的每个属性都有一个**值**

7.2.2 联系集

联系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

实体集 E_1, E_2, \dots, E_n 参与联系集R

实体在联系中扮演的功能称为实体的**角色**

联系也可以具有**描述性属性**

参与联系集的实体集的数目称为联系集的**度**（二元联系集的度为2，三元联系集的度为3）

7.2.3 属性

域（值集）：每个属性可取值的集合

属性类型：

- **简单和复合属性**：

复合属性可以划分为更小的部分（其它属性）



- **单值和多值属性**：

单值属性：一个属性只有单独的一个值

多值属性：一个属性可能对应于一组值

- **派生属性**：

这类属性的值可以从别的相关属性或实体派生出来

7.3 约束

7.3.1 映射基数

映射基数：表示一个实体通过一个联系集能关联的实体的个数

- 一对一
- 一对多
- 多对一
- 多对多

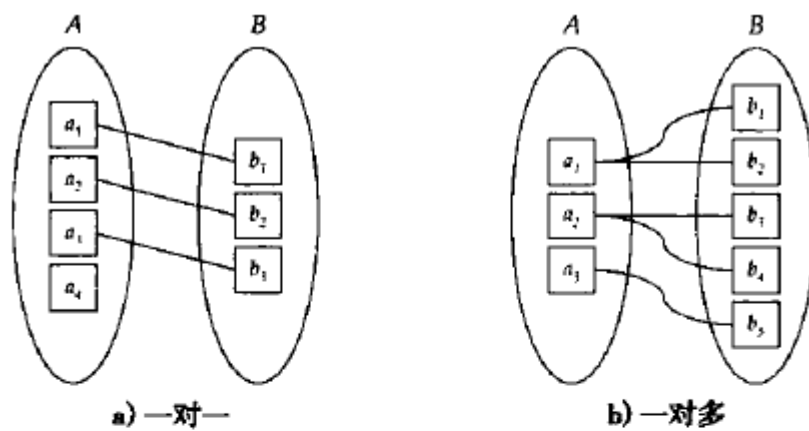


图 7-5 映射基数

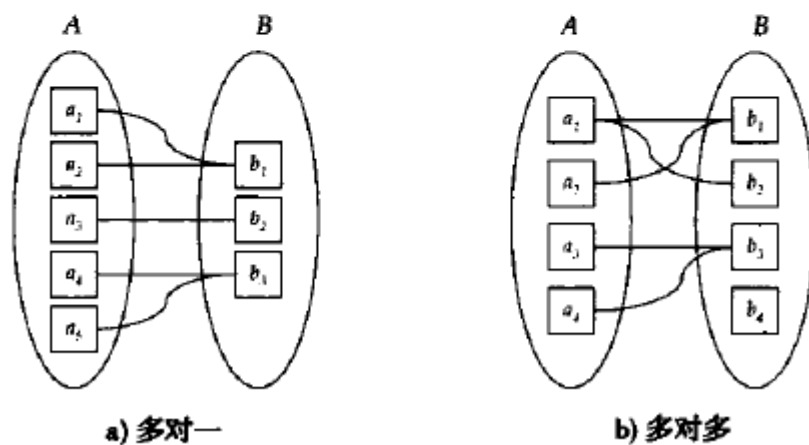


图 7-6 映射基数

7.3.2 参与约束

如果实体集E中的每个实体都参与到联系集R的至少一个联系中，则称E在R中的参与是**全部的**

7.3.3 码

一个实体的属性的值必须可以**唯一**标识该物体

联系集的主码结构依赖于联系集的映射基数

7.5 实体-联系图

E-R图可以图形化标识数据库的全局逻辑结构

7.5.1 基本结构

E-R 图包括如下几个主要构件：

- 分成两部分的矩形代表实体集。本书中有阴影的第一部分包含实体集的名字，第二部分包含实体集中所有属性的名字。
- 菱形代表联系集。
- 未分割的矩形代表联系集的属性。构成主码的属性以下划线标明。
- 线段将实体集连接到联系集。
- 虚线将联系集属性连接到联系集。
- 双线显示实体在联系集中的参与度。
- 双菱形代表连接到弱实体集的标志性联系集（我们将在 7.5.6 节讲述标志性联系集和弱实体集）。



图 7-7 对应于教师和学生的 E-R 图

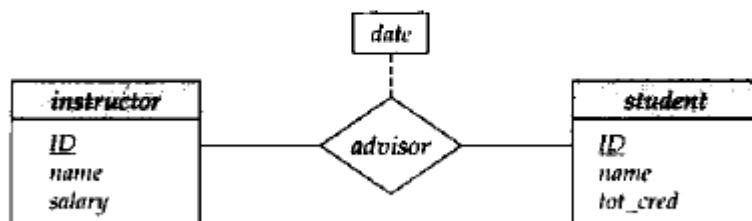
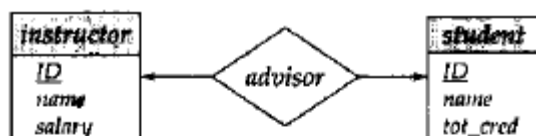


图 7-8 联系集上附带了一个属性的 E-R 图

7.5.2 映射基数



a) 一对一



b) 一对多



c) 多对多

图 7-9 联系

7.5.3 复杂的属性

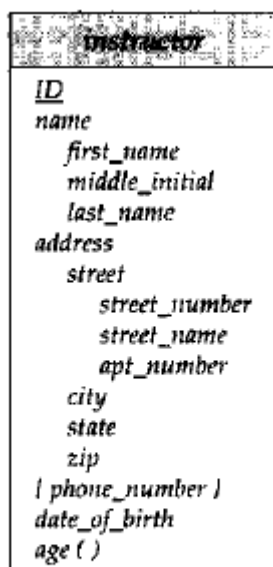


图 7-11 包含复合、多值和派生属性的 E-R 图

7.5.6 弱实体集

弱实体集：没有足够的属性以形成主码的实体集

强实体集：有主码的实体集

弱实体集必须与另一个称作**标识或属主实体集**的实体集关联才能有意义

弱实体集**存在依赖**于标识实体集；标识实体集**拥有**它所标识的弱实体集

弱实体集与其标识实体集相连的联系称为**标识性联系**

弱实体集的**分辨符**可以区分其中的实体

在 E-R 图中，弱实体集和强实体集类似，以矩形表示，但是有两点主要的区别：

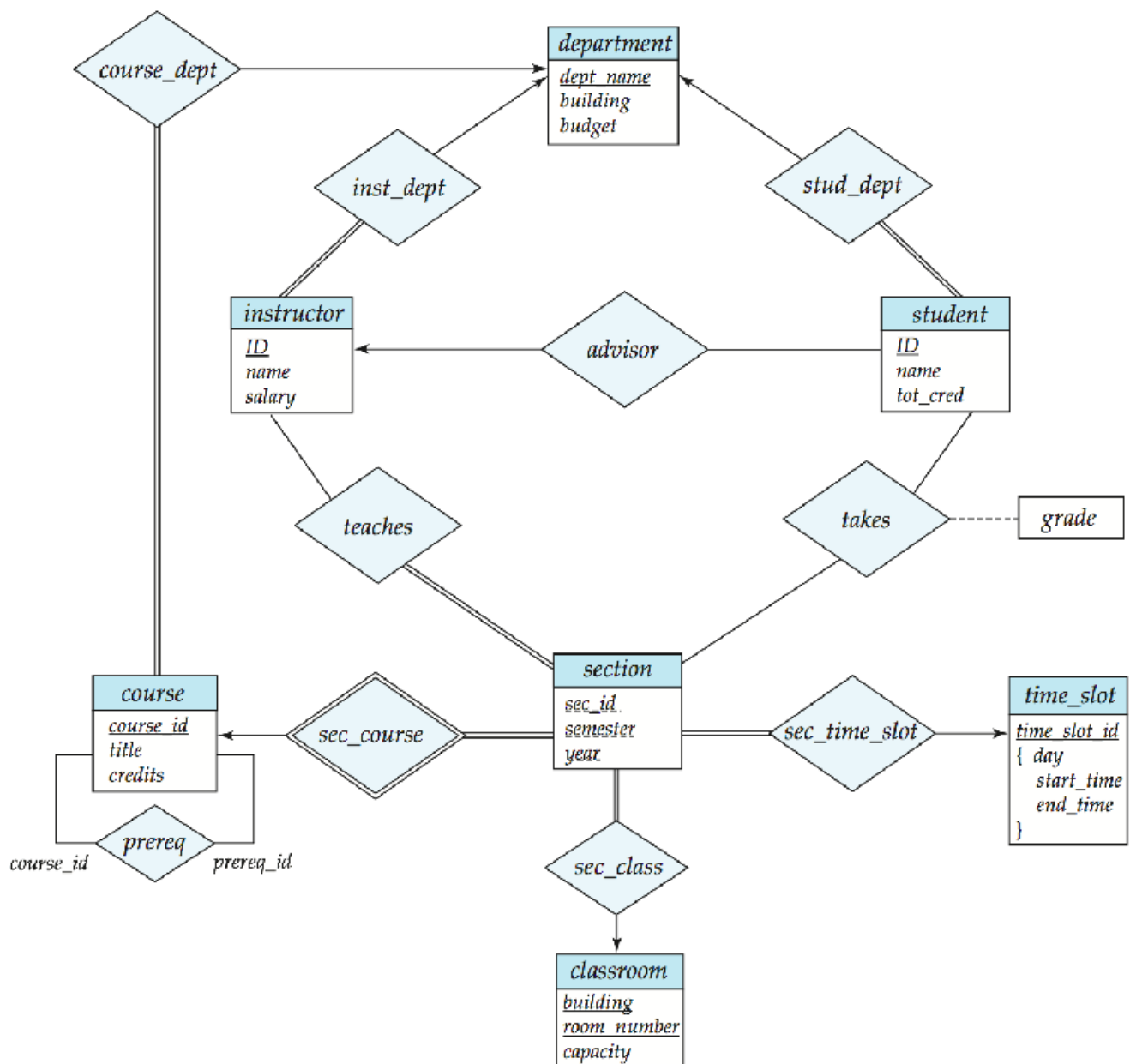
- 弱实体集的分辨符以虚下划线标明，而不是实线。
- 关联弱实体集和标识性强实体集的联系集以双菱形表示。

在图 7-14 中，弱实体集 *section* 通过联系集 *sec_course* 依赖于强实体集 *course*。



图 7-14 包含弱实体集的 E-R 图

7.5.7 大学的E-R图



7.7 实体-联系设计问题

7.7.1 用实体集还是用属性

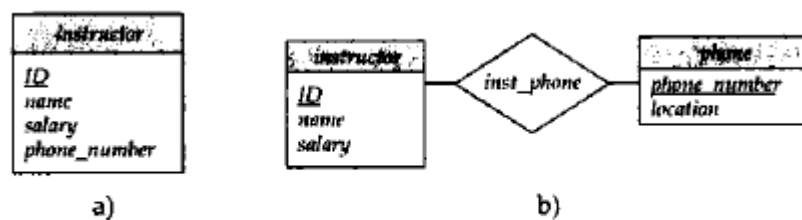


图 7-17 给 *instructor* 实体集增加 *phone* 的两种方法

常见错误：用一个实体集的主码作为另一个实体集的属性，而不是用联系

7.7.2 用实体集还是用联系集

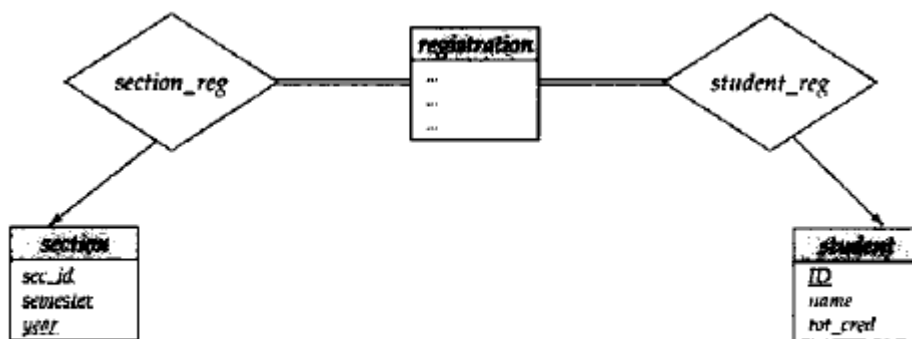


图 7-18 用 *registration* 和两个联系集替代 *takes*

原则：当描述发生在实体间的行为时采用联系集

7.7.3 二元还是n元联系集

数据库中的联系通常都是二元的，一些看来非二元的联系也可以用多个二元联系表示

7.7.4 联系属性的布局

一对一或一对多联系集的属性可以放到一个参与该联系的实体集中，而不是放到实体集中

7.8 扩展的E-R特性

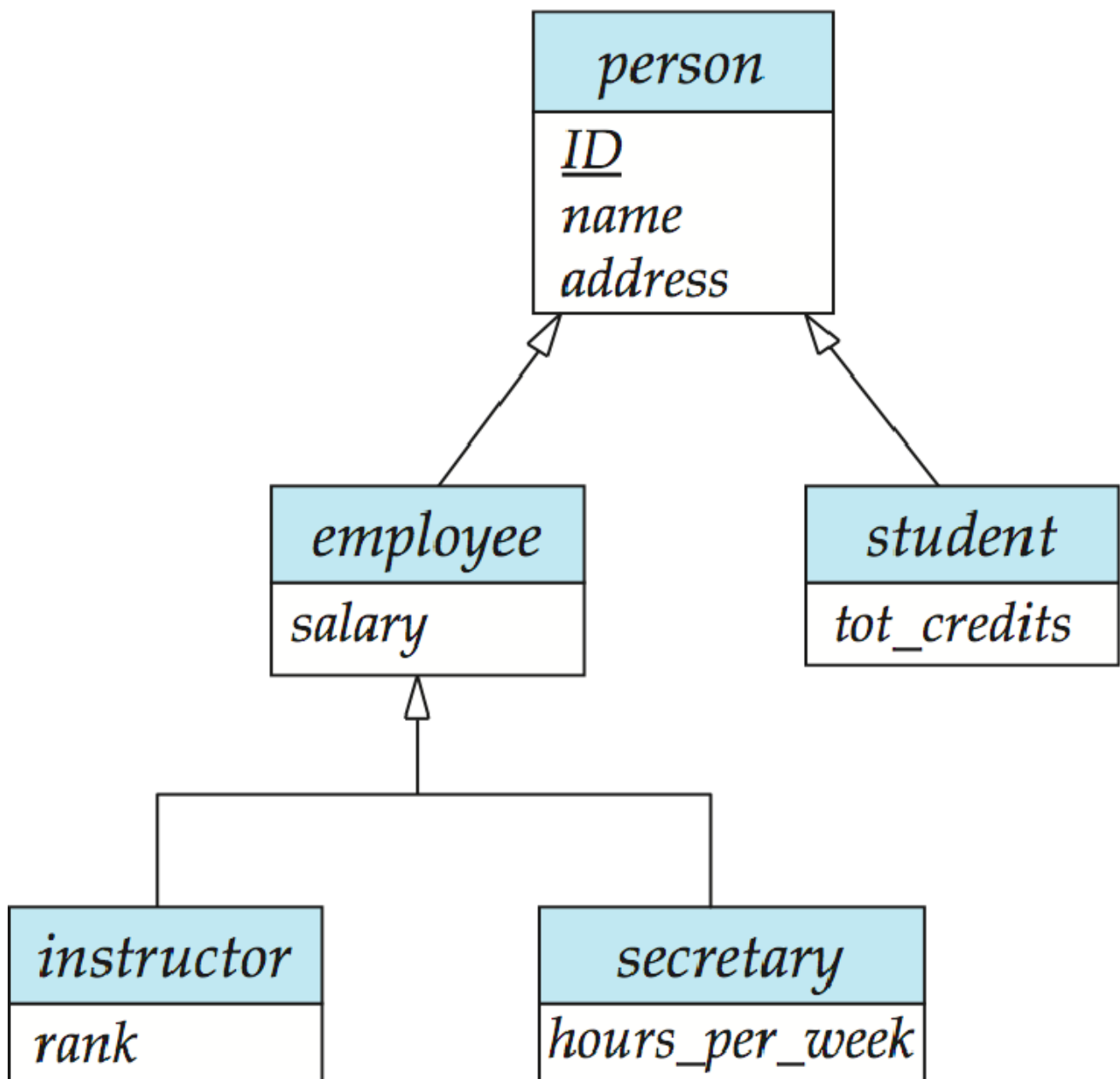
扩展E-R特性：特化、概化、高层和低层实体集属性继承、聚集

7.8.1 特化

在实体集内部进行分组的过程称为**特化** (e.g. person->employee,student)

重叠特化：一个实体集可能属于多个特化实体集

不相交特化：一个实体集属于至多一个特化实体集



7.8.2 概化

实体集间包含相同的属性可以通过**概化**来表达，概化是高层实体集与一个或多个实体集间的包含关系

高层与低层实体集也可以分别称作**超类**和**子类**

概化本质上是**特化**的逆过程

7.8.3 属性继承

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**

7.8.4 概化上的约束

判定哪些实体能成为给定低层实体集的成员的条件：

- **条件定义的**：成员资格的确定基于实体是否满足一个显式的条件或谓词
- **用户定义的**：低层实体集由用户将实体指派给某个实体集

一个概化中一个实体是否可以属于多个低层实体集：

- **不相交**：一个实体至多属于一个低层实体集
- **重叠**：同一个实体可以同时属于多个低层实体集

对概化的**完全性约束**：

- **全部概化或特化**：每个高层实体必须属于一个低层实体集
- **部分概化或特化**：一些高层实体不属于任何低层实体集

7.8.5 聚集

E-R模型的一个局限性在于它不能表达联系间的联系

聚集是一种抽象，通过这种抽象，联系被视为高层实体

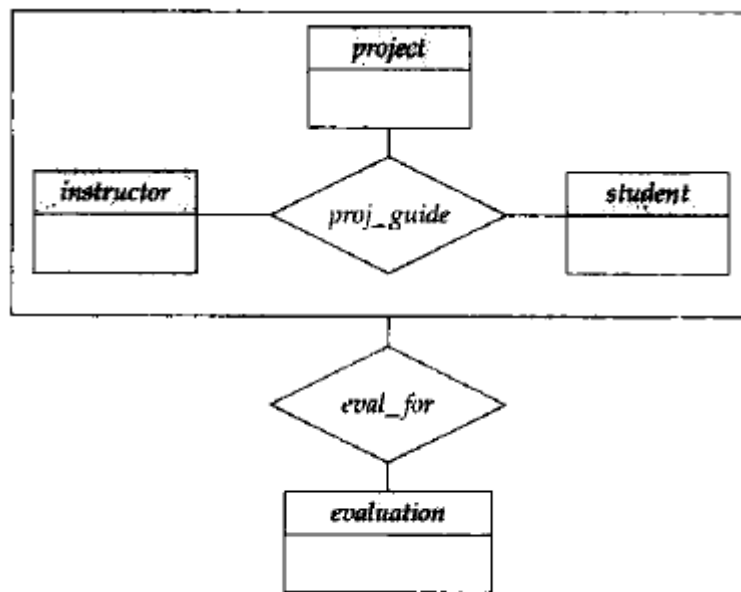


图 7-23 包含聚集的 E-R 图

7.11 总结

见P176

第八章 关系数据库设计

8.2 原子域和第一范式

一个域是**原子的**，如果该域的元素被认为是不可分的单元

关系模式R属于**第一范式 (1NF)**，如果R的所有属性的域都是原子的

8.3 使用函数依赖进行分解

属性集： α

关系模式： $r(R)$

超码: K

一个关系模式是一个属性集, 但是并非所有的属性集都是模式

8.3.1 码和函数依赖

一个关系的满足所有这种现实世界约束的实例, 称为关系的**合法实例**

R的子集K是r(R)的**超码**的条件: 在关系r(R)的任一合法实例中, 对于r的实例中的所有元组对 t_1 和 t_2 总满足, 若 $t_1 \neq t_2$, 则 $t_1[K] \neq t_2[K]$,

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

- 满足**函数依赖** $\alpha \rightarrow \beta$ 的条件是: 对实例中所有元组对 t_1 和 t_2 , 若 $t_1[\alpha] = t_2[\alpha]$, 则 $t_1[\beta] = t_2[\beta]$ (单射)
- 如果在r(R)的每个合法实例中都满足函数依赖 $\alpha \rightarrow \beta$, 则函数依赖在模式r(R)上**成立**

两种方式使用函数依赖:

- 判定关系的实例是否满足给定函数依赖集F
- 说明合法关系集上的约束

有些函数依赖称为**平凡的**, 因为它们在所有关系中都满足

如果 $\beta \subseteq \alpha$, 则形如 $\alpha \rightarrow \beta$ 的函数依赖是**平凡的**

F^+ 符号表示F集合的**闭包**, 表示F集合推导出的所有函数依赖的集合

8.3.2 Boyce-Codd 范式 (BCNF)

具有函数依赖集F的关系模式R属于BCNF的条件:

对 F^+ 中所有形如 $\alpha \rightarrow \beta$ 的函数依赖 (其中 $\alpha \subseteq R$ 且 $\beta \subseteq R$), 下面至少有一项成立:

- $\alpha \rightarrow \beta$ 是平凡的函数依赖 (即 $\beta \subseteq \alpha$)
- α 是模式R的一个超码

Example schema *not* in BCNF:

instr_dept (ID, name, salary, dept_name, building, budget)

because $dept_name \rightarrow building, budget$

holds on *instr_dept*, but *dept_name* is not a superkey

一个数据库属于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

非BCNF模式转换为BCNF的分解:

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

• $(\alpha \cup \beta) = (dept_name, building, budget)$

• $(R - (\beta - \alpha)) = (ID, name, salary, dept_name)$

8.3.4 第三范式

具有函数依赖集F的关系模式R属于**第三范式**的条件是：

对于 F^+ 中所有形如 $\alpha \rightarrow \beta$ 的函数依赖（其中 $\alpha \subseteq R$ 且 $\beta \subseteq R$ ），以下至少一项成立：

- $\alpha \rightarrow \beta$ 是一个平凡的函数依赖
- α 是R的一个超码
- $\beta - \alpha$ 中的每个属性A都包含于R的一个候选码中（可以包含于不同的候选码）

8.4 函数依赖理论

8.4.1 函数依赖集的闭包

如果关系模式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**逻辑蕴涵**（ $A \rightarrow B, B \rightarrow C$ 逻辑蕴涵 $A \rightarrow C$ ）

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：正确有效的，完备的

- **自反律**：若 α 为一属性集且 $\beta \subseteq \alpha$ ，则 $\alpha \rightarrow \beta$ 成立
- **增补律**：若 $\alpha \rightarrow \beta$ 成立且 γ 为一属性集，则 $\gamma\alpha \rightarrow \gamma\beta$
- **传递律**：若 $\alpha \rightarrow \beta$ 和 $\beta \rightarrow \gamma$ 成立，则 $\alpha \rightarrow \gamma$ 成立
- **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 rule)。若 $\alpha \rightarrow \beta$ 和 $\gamma\beta \rightarrow \delta$ 成立，则 $\alpha\gamma \rightarrow \delta$ 成立。

```
 $F^+ = F$ 
repeat
  for each  $F^+$  中的函数依赖  $f$ 
    在  $f$  上应用自反律和增补律
    将结果加入到  $F^+$  中
  for each  $F^+$  中的一对函数依赖  $f_1$  和  $f_2$ 
    if  $f_1$  和  $f_2$  可以使用传递律结合起来
      将结果加入到  $F^+$  中
until  $F^+$  不再发生变化
```

图 8-7 计算 F^+ 的过程

8.4.2 属性集的闭包

如果 $\alpha \rightarrow B$ ，我们称属性B被 α **属性确定**

令 α 为一个属性集，F下被 α 确定的所有属性的集合称为F下 α 的**就闭包**，记为 α^+

```

result :=  $\alpha$ ;
repeat
  for each 函数依赖  $\beta \rightarrow \gamma$  in  $F$  do
    begin
      if  $\beta \subseteq \text{result}$  then  $\text{result} := \text{result} \cup \gamma$ ;
    end
until (result 不变)

```

图 8-8 计算 F 下 α 的闭包 α^+ 的算法

■ $R = (A, B, C, G, H, I)$

■ $F = \{A \rightarrow B$
 $A \rightarrow C$
 $CG \rightarrow H$
 $CG \rightarrow I$
 $B \rightarrow H\}$

■ $(AG)^+$

1. $\text{result} = AG$

2. $\text{result} = ABCG$ ($A \rightarrow C$ and $A \rightarrow B$)

3. $\text{result} = ABCGH$ ($CG \rightarrow H$ and $CG \subseteq AGBC$)

4. $\text{result} = ABCGHI$ ($CG \rightarrow I$ and $CG \subseteq AGBCH$)

属性闭包算法有多种用途：

- 判断 α 是否为超码：计算 α^+ ，检查 α^+ 是否包含 R 中的所有属性
- 检查函数依赖 $\alpha \rightarrow \beta$ 是否成立：通过检查是否 $\beta \subseteq \alpha^+$ （是否属于 F^+ ），即计算 α^+ 看是否包含 β
- 计算 F^+ 方法：对任意的 $\gamma \subseteq R$ ，我们找出闭包 γ^+ ；对任意的 $S \subseteq \gamma^+$ ，我们输出一个函数依赖 $\gamma \rightarrow S$

Is AG a candidate key?

1. Is AG a super key?

1. Does $AG \rightarrow R$? == Is $(AG)^+ \supseteq R$

2. Is any subset of AG a superkey?

1. Does $A \rightarrow R$? == Is $(A)^+ \supseteq R$

2. Does $G \rightarrow R$? == Is $(G)^+ \supseteq R$

8.4.3 正则覆盖

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**

无关属性的形式化定义如下：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 逻辑蕴含 $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ ，则属性 A 在 α 中是无关的。
- 如果 $A \in \beta$ 并且函数依赖集 $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ 逻辑蕴含 F ，则属性 A 在 β 中是无关的。

有效检验一个属性是否无关的方法，考虑 $\alpha \rightarrow \beta$ 中的一个属性 A ：

- 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 α^+ (α 的闭包)；如果 α^+ 包含 A ，则 A 在 β 中是无关的。
- 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，计算在 F 下的 γ^+ (γ 的闭包)；如果 γ^+ 包含 β 中的所有属性，则 A 在 α 中是无关的。

example:

例如，假定 F 包含 $AB \rightarrow CD$ 、 $A \rightarrow E$ 和 $E \rightarrow C$ 。为检验 C 在 $AB \rightarrow CD$ 中是否是无关的，我们计算 $F' = \{AB \rightarrow D, A \rightarrow E, E \rightarrow C\}$ 下 AB 的属性闭包。闭包为 $ABCDE$ ，包含 CD ，所以我们推断出 C 是无关的。

F 的**正则覆盖** F_c 是一个依赖集，使得 F 逻辑蕴含 F_c 中的所有依赖，并且 F_c 逻辑蕴含 F 中的所有依赖，且 F_c 必须具有如下性质：

- 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 $\alpha_1 = \alpha_2$

```
 $F_c = F$ 
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  $F_c$  中寻找到一个函数依赖  $\alpha \rightarrow \beta$ ，它在  $\alpha$  或在  $\beta$  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 ( $F_c$  不变)
```

图 8-9 计算正则覆盖

example:

- $R = (A, B, C)$
 $F = \{A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C\}$
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
 - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous in $AB \rightarrow C$
 - Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
 - ☞ Yes: in fact, $B \rightarrow C$ is already present!
 - Set is now $\{A \rightarrow BC, B \rightarrow C\}$
- C is extraneous in $A \rightarrow BC$
 - Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
 - ☞ Yes: using transitivity on $A \rightarrow B$ and $B \rightarrow C$.
 - Can use attribute closure of A in more complex cases
- The canonical cover is:

$A \rightarrow B$
 $B \rightarrow C$

8.4.4 无损分解

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，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**

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属于 F^+ ：

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

example:

为举例说明，考虑模式

inst_dept (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)

我们在 8.1.2 节中将其分解为 *instructor* 和 *department* 模式：

instructor (*ID*, *name*, *dept_name*, *salary*)
department (*dept_name*, *building*, *budget*)

考虑这两个模式的交集，即 *dept_name*。我们发现，由于 $dept_name \rightarrow dept_name, building, budget$ ，因此满足无损分解条件。

8.4.5 保持依赖

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

令 $F' = F_1 \cup F_2 \cup \dots \cup F_n$ ，若分解具有性质 $F'^+ = F^+$ ，则该分解被称为**保持依赖的分解**

验证方法一：

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

图 8-10 保持依赖性的验证

验证方法二：

```
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 没有变化)
```

8.5 分解算法

8.5.1 BCNF分解

8.5.1.1 BCNF的判定方法

方法一：（对某些存在 F 中不存在而在 F^+ 中的函数依赖情况不适用）

- 为了检查非平凡的函数依赖 $\alpha \rightarrow \beta$ 是否违反 BCNF，计算 α^+ (α 的属性闭包)，并且验证它是否包含 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，仅须检查给定集合 F 中的函数依赖是否违反 BCNF 就足够了，不用检查 F^+ 中的所有函数依赖。

方法二：

- 对于 R_i 中属性的每个子集 α ，确保 α^+ (F 下 α 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。

即，如果存在这样的函数依赖 $\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$ 在 F^+ 中，说明 R_i 违反 BCNF

8.5.1.2 BCNF分解算法

```

result := {  $R$  };
done := false;
计算  $F^+$ ;
while (not done) do
    if (result 中存在模式  $R_i$  不属于 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;

```

图 8-11 BCNF 分解算法

example:

模式:

class (course_id, title, dept_name, credits, sec_id, semester, year, building, room_number, capacity, time_slot_id)

函数依赖:

course_id \rightarrow title, dept_name, credits

building, room_number \rightarrow capacity

course_id, sec_id, semester, year \rightarrow building, room_number, time_slot_id

分解结果:

course(course_id, title, dept_name, credits)

classroom (building, room_number, capacity)

section (course_id, sec_id, semester, year, building, room_number, time_slot_id)

这三个关系模式都属于BCNF

8.5.2 3NF分解

```

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;
/* (可选) 移除冗余关系 */
repeat
    if 模式  $R_j$  包含于另一个模式  $R_i$  中
    then
        /* 删除  $R_j$  */
         $R_j := R_i$ ;
         $i := i - 1$ ;
until 不再有可以删除的  $R_j$ 
return( $R_1, R_2, \dots, R_i$ )

```

图 8-12 到 3NF 的保持依赖且无损的分解

example:

关系模式:

cust_banker_branch (customer_id, employee_id, branch_name, type)

函数依赖:

customer_id, employee_id \rightarrow branch_name, type

employee_id \rightarrow branch_name

customer_id, branch_name \rightarrow employee_id

正则覆盖:

customer_id, employee_id \rightarrow type

employee_id \rightarrow branch_name

customer_id, branch_name \rightarrow employee_id

分解结果

(customer_id, employee_id, type)

(customer_id, branch_name, employee_id)

8.5.3 3NF算法的正确性

(好像不重要, 不管了)

8.5.4 BCNF和3NF的比较

3NF的优缺点:

优点: 总可以在满足无损并保持依赖的前提下得到3NF设计

缺点: 可能不得不用空值表示数据项间的某些可能有意义的联系

数据库设计目标：

1. BCNF
2. 无损
3. 保持依赖

8.6 使用多值依赖的分解

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

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$

$$t_3[\beta] = t_1[\beta]$$

$$t_3[R - \beta] = t_2[R - \beta]$$

$$t_4[\beta] = t_2[\beta]$$

$$t_4[R - \beta] = t_1[R - \beta]$$

$\alpha \twoheadrightarrow \beta$ 的表格表示：

	α	β	$R - \alpha - \beta$
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	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$

We say that $Y \twoheadrightarrow Z$ (Y **multidetermines** Z) if and only if for all possible relations $r(R)$

$$\langle y_1, z_1, w_1 \rangle \in r \text{ and } \langle y_1, z_2, w_2 \rangle \in r$$

then

$$\langle y_1, z_1, w_2 \rangle \in r \text{ and } \langle y_1, z_2, w_1 \rangle \in r$$

由多值依赖的定义，我们可以得出以下规则，对于 $\alpha, \beta \subseteq R$ ：

- 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 $\alpha \twoheadrightarrow R - \alpha - \beta$ 。

8.8 数据库设计过程

在本章的前面，从 8.3 节开始，我们假定给定关系模式 $r(R)$ ，并对之进行规范化。我们可以采用以下几种方法得到模式 $r(R)$ ：

1. $r(R)$ 可以是由 E-R 图向关系模式集进行转换时生成的。
2. $r(R)$ 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。
3. $r(R)$ 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。

8.10 总结

见 P206

第三部分 数据存储和查询

第十章 存储和文件结构

10.1 物理存储介质概述

- 高速缓冲存储器 (cache)
- 主存储器 (main memory)
- 快闪存储器 (flash memory)
- 磁盘存储器 (磁盘)
- 光学存储器 (光盘)
- 磁带存储器

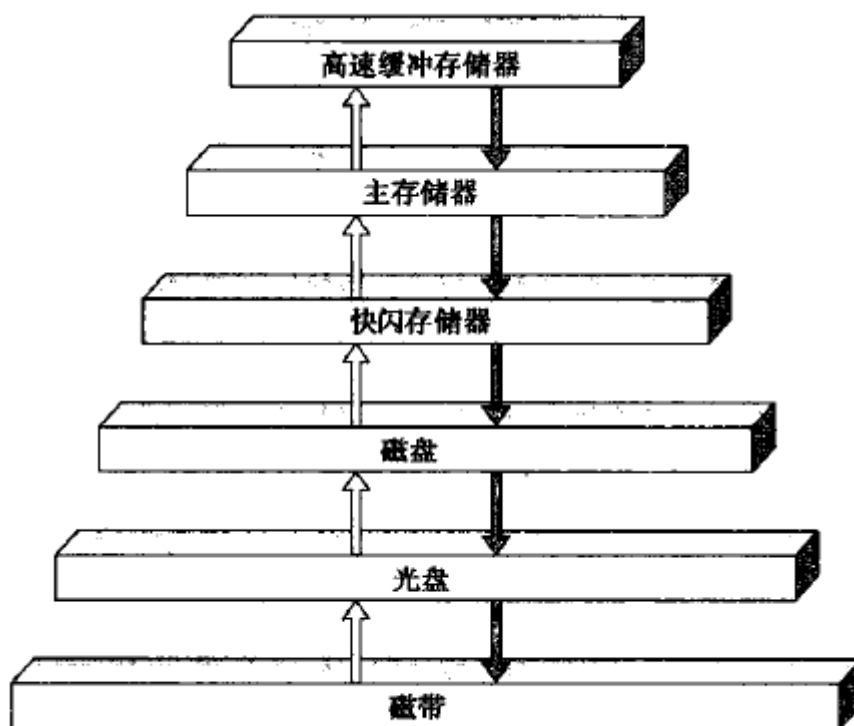


图 10-1 存储设备层次结构

10.2 磁盘和快闪存储器

10.2.2 磁盘性能的度量

- 访问时间：发出读写请求到数据开始传输之间的时间
 - 寻道时间：磁盘臂重定位的时间
- 旋转等待时间：等待访问的扇区出现在读写头下所花费的时间
- 数据传输率：磁盘获取数据或者向磁盘存储数据的速率
- 平均故障时间：磁盘可靠性的度量标准

10.2.3 磁盘块访问的优化

一个**块**是一个逻辑单元，它包含固定数目的连续扇区

10.3 RAID

为提高性能和可靠性，**独立磁盘冗余阵列(RAID)**技术被引入

10.3.1 通过冗余提高可靠性

引入**冗余**可以提高可靠性，即存储正常情况下不需要的额外信息

镜像：复制每一张磁盘，这样一张逻辑磁盘由两张物理磁盘组成

10.3.2 通过并行提高性能

通过在多张磁盘上进行**数据拆分**来提高传输速率

有比特级拆分和块级拆分等方式

10.3.3 RAID级别

- RAID0 块级拆分但没有任何冗余
- RAID1 块级拆分的磁盘镜像
- RAID5 块交叉的分布奇偶校验位的组织结构

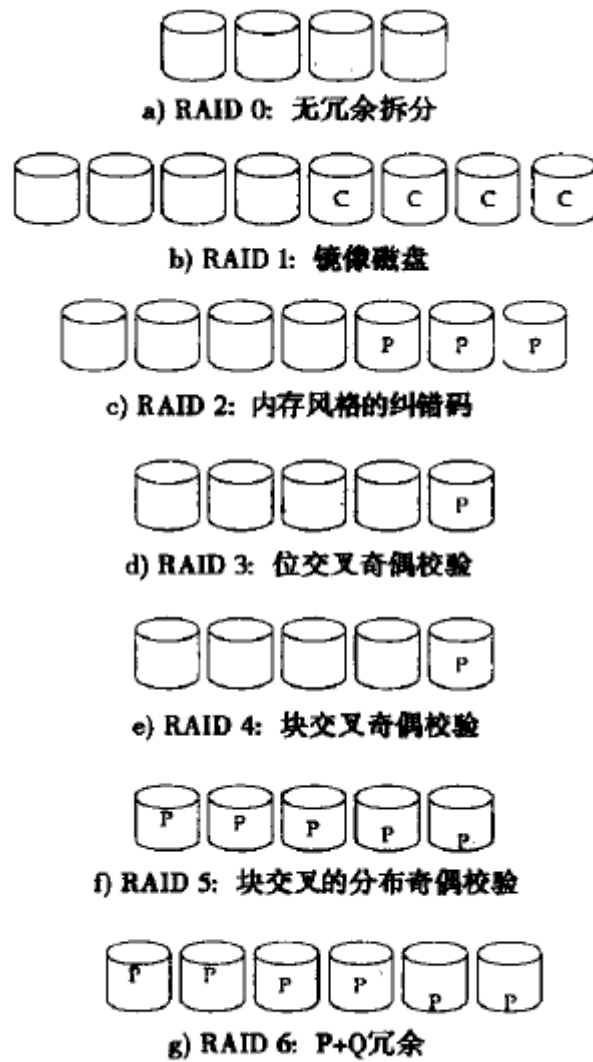


图 10-3 RAID 级别

10.5 文件组织

一个**文件**在逻辑上组织成为记录的一个序列

每个文件分成定长的存储单元，称为**块**

10.5.1 定长记录

在文件开始处，分配一定数量的字节作为**文件头**，包含有关文件的各种信息

被删除的记录形成链表，被称为**空闲链表**

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

图 10-7 删除了第 1、4 和 6 条记录的图 10-4 中的文件

10.5.2 变长记录

变长记录有两个部分：初始部分是定长属性，后面是变长属性

对于定长属性，存储它们的值所需的字节数

对于变长属性，在记录的初始部分中表示为一个对（偏移量，长度）值

空位图：用来表示记录的哪个属性是空值

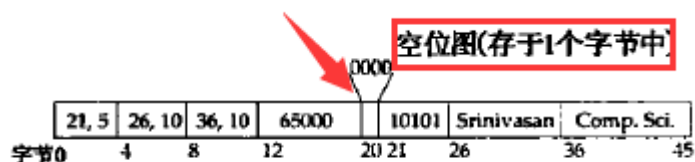


图 10-8 变长记录的表示

分槽的页结构，一般用于在块中组织记录，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
- 一个由包含记录位置和大小记录组成的数组

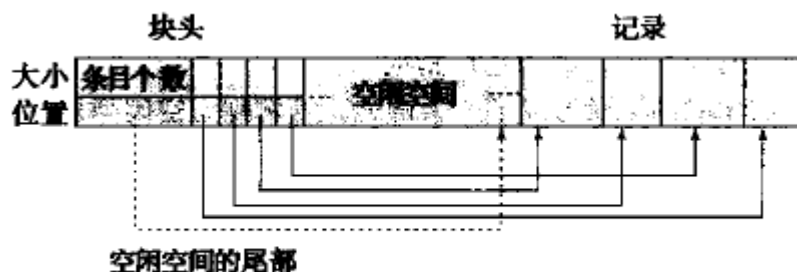


图 10-9 分槽的页结构

实际记录从块的尾部开始连续排列，块中的空闲空间是连续的

10.6 文件中记录的组织

- 堆文件组织：一条记录可以放在文件中的任何地方
- 顺序文件组织：记录根据“搜索码”的值顺序存储
- 散列文件组织：散列函数的结果去定了记录应该放到文件的哪个块中

10.6.1 顺序文件组织

顺序文件按某个搜索码的顺序排序

搜索码是任何一个属性或者属性的集合

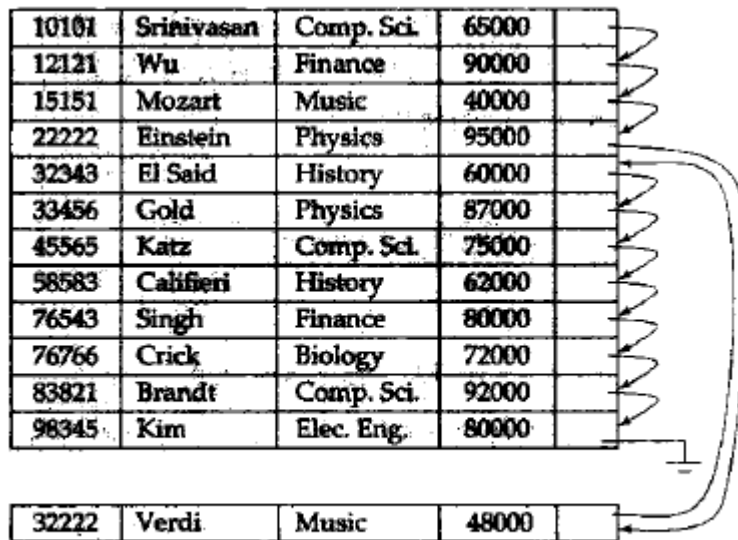


图 10-11 执行插入后的顺序文件

10.6.2 多表聚簇文件组织

多表聚簇文件组织是一种在每一块中存储两个或者更多个关系的相关记录的文件结构

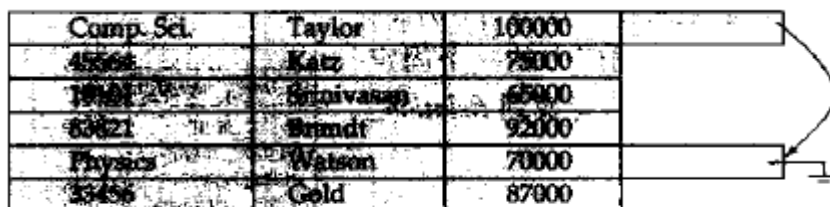


图 10-15 带指针链的多表聚簇文件结构

10.7 数据字典存储

元数据：关于数据的数据（如关系的模式）

元数据存储被称为数据字典或系统目录中的结构中

系统必须存储的信息类型有：

- 关系的名字。
- 每个关系中属性的名字。
- 属性的域和长度。
- 在数据库上定义的视图的名字和这些视图的定义。
- 完整性约束(例如,码约束)。

此外,很多系统为系统的用户保存了下列数据:

- 授权用户的名字。
- 关于用户的授权和账户信息。
- 用于认证用户的密码或其他信息。

10.9 总结

见P264

第十一章 索引与散列

11.1 基本概念

两种基本索引类型:

- **顺序索引**: 基于值的顺序排序
- **散列索引**: 基于将值平均分布到若干散列桶中,一个值所属的散列桶是由一个函数决定的,这个函数称为*散列函数*

对每种技术的评价基于以下因素:

- **访问类型**(access type): 能有效支持的访问类型。访问类型可以包括找到具有特定属性值的记录,以及找到属性值落在某个特定范围内的记录。
- **访问时间**(access time): 在查询中使用该技术找到一个特定数据项或数据项集所需的时间。
- **插入时间**(insertion time): 插入一个新数据项所需的时间。该值包括找到插入这个新数据项的正确位置所需的时间,以及更新索引结构所需的时间。
- **删除时间**(deletion time): 删除一个数据项所需的时间。该值包括找到待删除项所需的时间,以及更新索引结构所需的时间。
- **空间开销**(space overhead): 索引结构所占用的额外存储空间。倘若存储索引结构的额外空间大小适度,通常牺牲一定的空间代价来换取性能的提高是值得的。

用于在文件中查找记录的属性或属性集称为**搜索码**

11.2 顺序索引

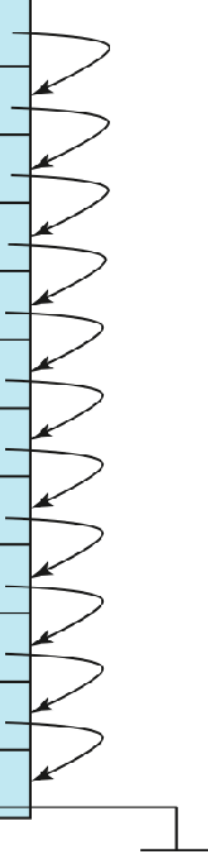
如果文件按照某个搜索码指定的顺序排序,那么该搜索码对应的索引称为**聚集索引**,也称为**主索引**

搜索码制定的顺序与文件中记录的物理顺序不同的索引称为**非聚集索引**或**辅助索引**

索引顺序文件: 搜索码上有聚集索引的文件

example:

10101	Srinivasan	Comp. Sci.	65000		
12121	Wu	Finance	90000		
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		

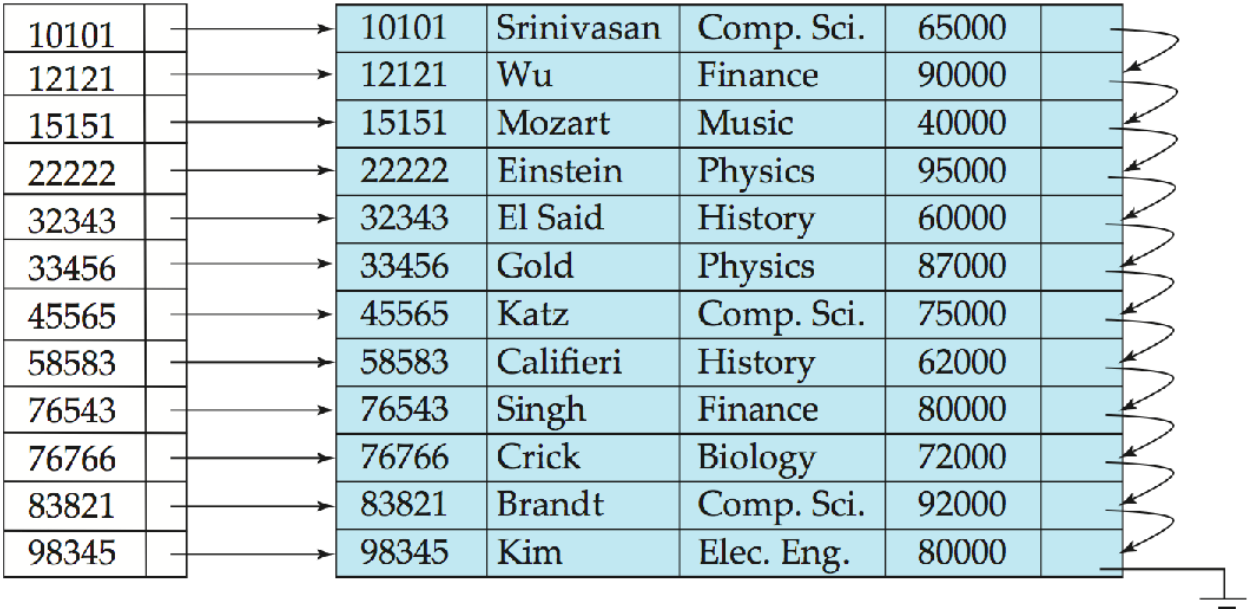


11.2.1 稠密索引和稀疏索引

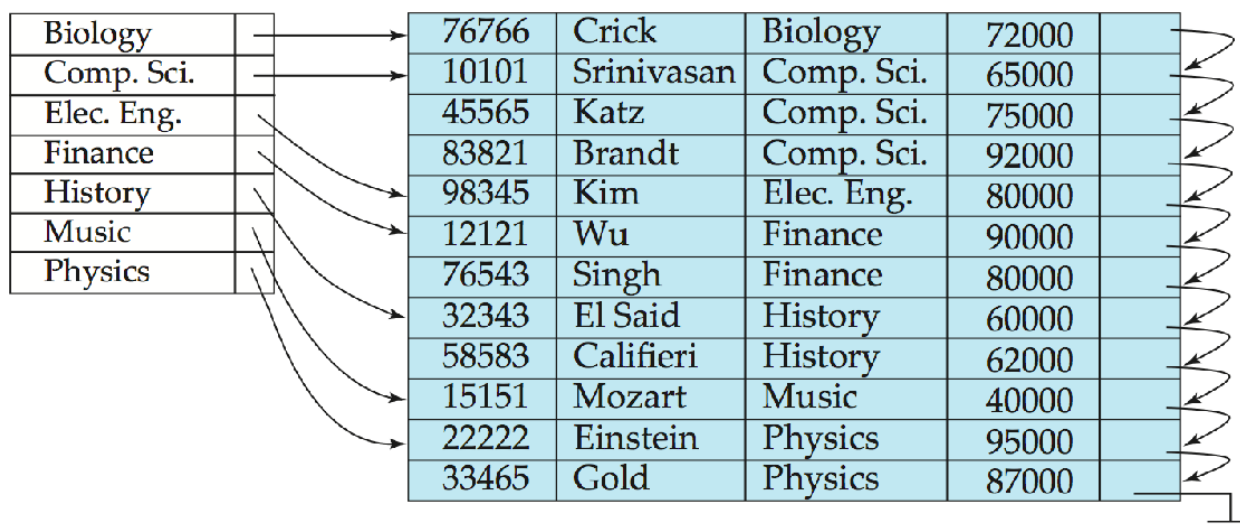
索引项或索引记录由一个搜索码值和指向具有该搜索码的一条或多条记录的指针构成

- **稠密索引**：在稠密索引中，文件中的每个搜索码值都有一个索引项

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	



- **稀疏索引**：在稀疏索引中，只为搜索码的某些值建立索引项



11.2.2 多级索引

具有两级或两级以上的索引称为**多级索引**

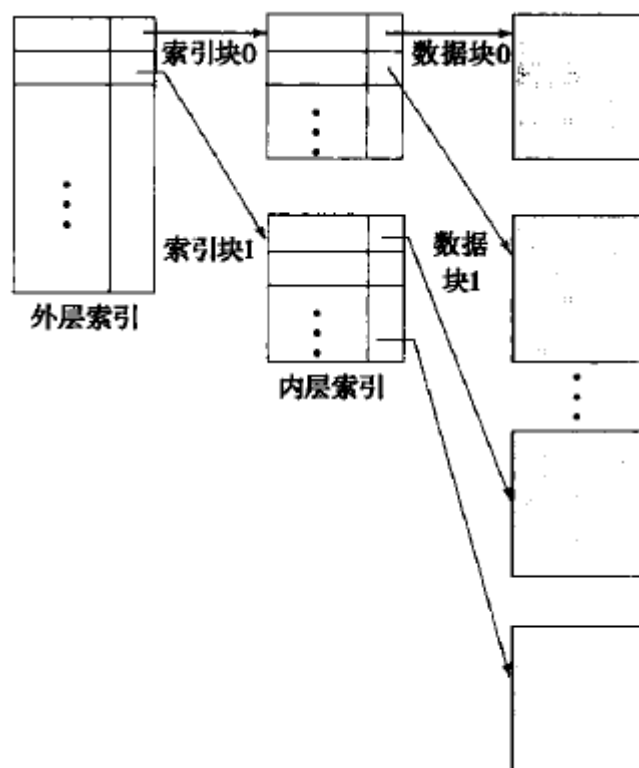


图 11-5 二级稀疏索引

11.2.3 索引的更新

每当文件有记录更新时，索引都需要更新

- **插入。**系统首先用出现在待插入记录中的搜索码值进行查找，并根据索引是稠密索引还是稀疏索引而进行下一个操作。

- **稠密索引：**

1. 如果该搜索码值不在索引中，系统就在索引中合适的位置插入具有该搜索码值的索引项。
2. 否则进行如下操作：
 - a. 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，系统就在索引项中增加一个指向新记录的指针。
 - b. 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。

- **稀疏索引：**我们假设索引为每个块保存一个索引项。如果系统创建一个新的块，它会将新块中出现的第一个搜索码值(按照搜索码的顺序)插入到索引中。另一方面，如果这条新插入的记录含有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。**为删除一条记录，系统首先查找要删除的记录，然后下一步的操作取决于索引是稠密索引还是稀疏索引。

- **稠密索引：**

1. 如果被删除的记录是具有这个特定搜索码值的唯一的一条记录，系统就从索引中删除相应的索引项。
2. 否则采取如下操作：
 - a. 如果索引项存储的是指向所有具有相同搜索码值的记录的指针，系统就从索引项中删除指向被删除记录的指针。
 - b. 否则，索引项存储的是指向具有该搜索码值的第一条记录的指针。在这种情况下，如果被删除的记录是具有该搜索码值的第一条记录，系统就更新索引项，使其指向下一条记录。

- **稀疏索引：**

1. 如果索引不包含具有被删除记录搜索码值的索引项，则索引不必做任何修改。
2. 否则系统采取如下操作：
 - a. 如果被删除的记录是具有该搜索码值的唯一记录，系统用下一个搜索码值(按搜索码顺序)的索引记录替换相应的索引记录。如果下一个搜索码值已经有一个索引项，则删除而不是替换该索引项。
 - b. 否则，如果该搜索码值的索引记录指向被删除的记录，系统就更新索引项，使其指向具有相同搜索码值的下一条记录。

11.2.4 辅助索引

辅助索引必须是稠密索引，对每个搜索码值都有一个索引项

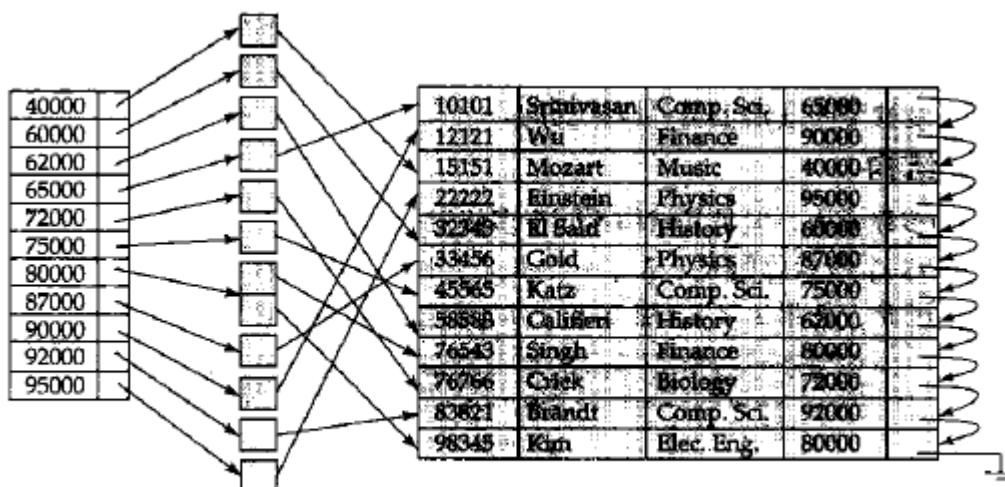


图 11-6 *instructor* 文件的辅助索引，基于非候选码 *salary*

11.2.5 多码上的索引

一个包含多个属性的搜索码被称为**符合搜索码**

11.5 多码访问

11.5.1 使用多个单码索引

```
select ID
from instructor
where dept_name = "Finance" and salary = 80000
```

处理这个查询可以有三种策略：

1. 利用 *dept_name* 上的索引，找出属于金融系的所有记录。检查每条记录是否满足 *salary* = 80 000。
2. 利用 *salary* 上的索引，找出所有工资等于 \$80 000 的记录。检查每条记录是否满足 *dept_name* = "Finance"。
3. 利用 *dept_name* 上的索引找出指向属于金融系的记录的所有指针。同样，利用 *salary* 上的索引找出指向工资等于 \$80 000 的记录的所有指针。计算这两个指针集合的交。交集集中的那些指针指向金融系中工资等于 \$80 000 的记录。

11.5.2 多码索引

在复合的搜索码上建立和使用索引

11.5.3 覆盖索引

覆盖索引存储一些属性（但不是搜索码属性）的值以及指向记录的指针

11.6 静态散列

基于散列技术的文件组织使我们能够避免访问索引结构

桶：表示能存储一条或多条记录的一个存储单位

散列文件组织中，我们通过计算所需记录搜索码值上的一个函数直接获得包含该记录磁盘块地址

散列索引组织中，我们把搜索码以及与它们相关联的指针组织成一个散列文件结构

11.6.1 散列函数

我们希望搜索码值分配到桶中并且具有以下分布特性的散列函数：分布是*均匀的、随机的*

11.6.2 桶溢出处理

如果桶没有足够的空间，就会发生**桶溢出**，有以下几个可能原因：

- **桶不足**
- **偏斜**（某些桶分配到的记录比其它的桶多）

偏斜发生的原因：

- 多条记录可能具有相同的搜索码
- 所选的散列函数可能会造成搜索码的分布不均

我们用**溢出桶**来处理溢出问题，这种链接列表的溢出处理称为**溢出链**

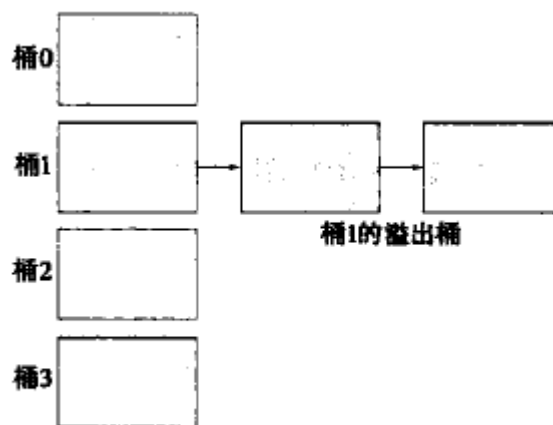


图 11-24 散列结构中的溢出链

上面这种形式的散列结构称为**闭地址**，下面这种方法称为**开地址**

桶集合是固定的，没有溢出链

当一个桶满了以后，系统将记录插入到其它桶中

该方法在数据库系统不使用，因为删除操作太麻烦

11.6.3 散列索引

散列索引将搜索码及其相应的指针组织成散列文件结构

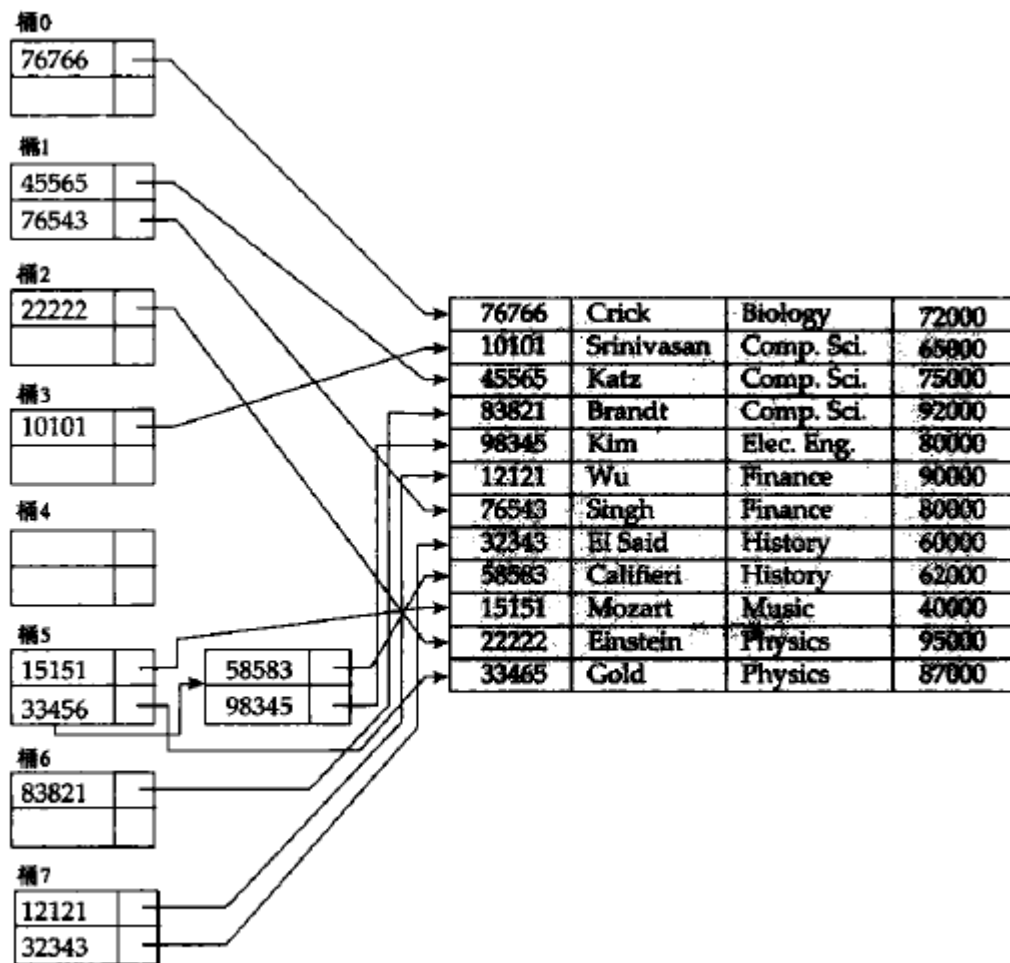


图 11-25 *instructor* 文件中在搜索码 ID 上的散列索引

11.7 动态散列

动态散列技术允许散列函数动态改变，以适应数据库增大或缩小的需要

11.7.1 数据结构

选择一个具有均匀性和随机性的散列函数 h ， h 可以产生值的范围较大，是 b 位二进制整数

开始时我们不使用 b 位，而是使用 i 位， i 会随着文件增长而变化

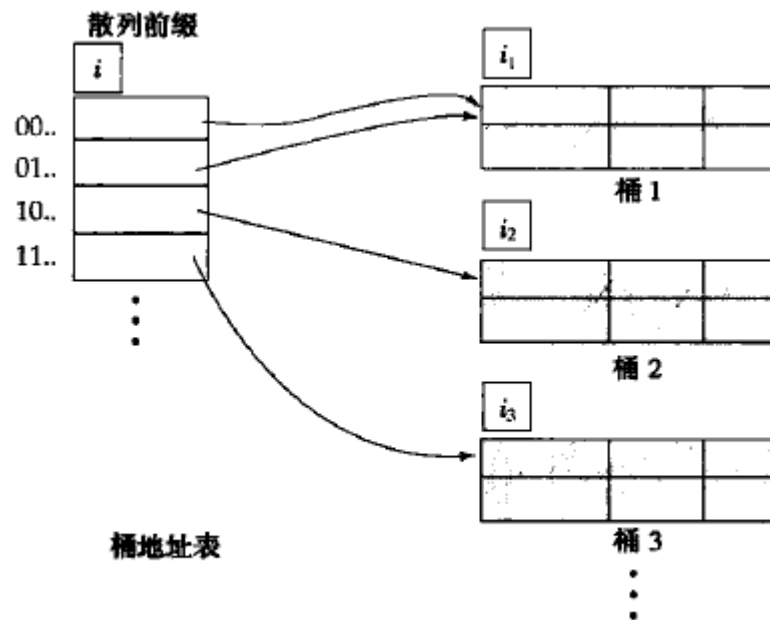


图 11-26 可扩充散列的一般结构

11.7.2 查询和更新

假定要插入一条搜索码值为K的记录，最终定位到某个桶j

算法：

若桶j中有剩余空间，直接将记录插入

若桶j已经满，则分裂桶并重新分配记录

若 $i > i_j$ (前缀位数)，那么在桶地址表中有多个表项指向桶j，直接分裂桶j，将i值设置为 $i_j + 1$ 即可

若 $i = i_j$ ，那么在桶地址表中只有一个表项指向桶j，增加桶地址表的大小，以容纳由于桶j分裂而产生的两个桶指针， $i = i + 1$ ，分裂出来的桶i值设置为 $i+1$ ，将原来桶中的记录重新分配

若分裂多次还不能成功（如搜索码值都是相同的情况），则采用溢出桶来记录

example:

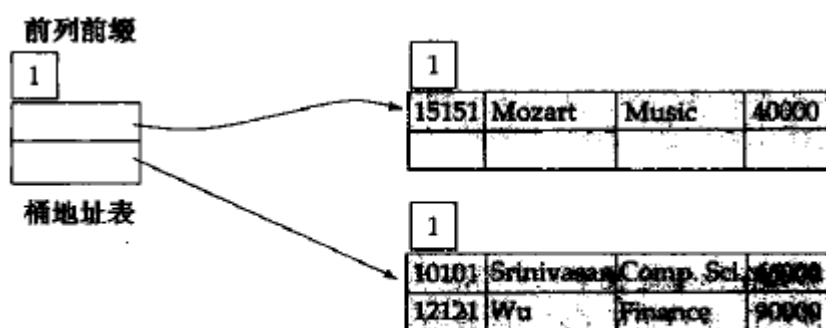


图 11-29 3 次插入操作后的散列结构

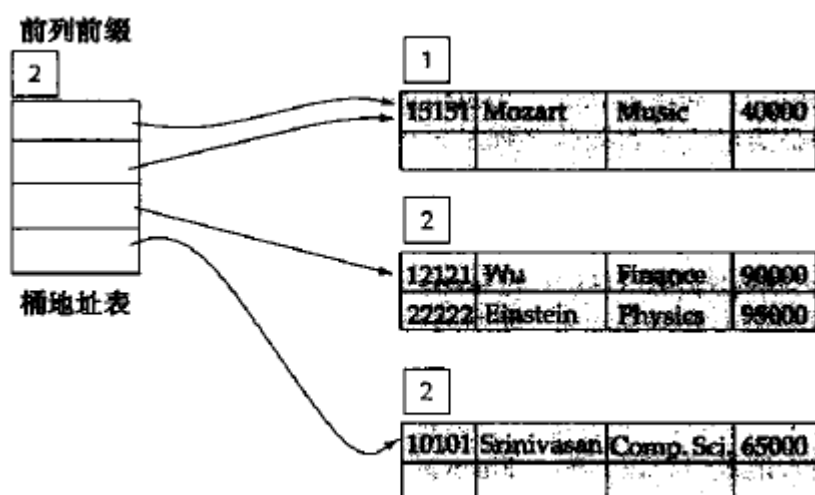


图 11-30 4 次插入操作后的散列结构

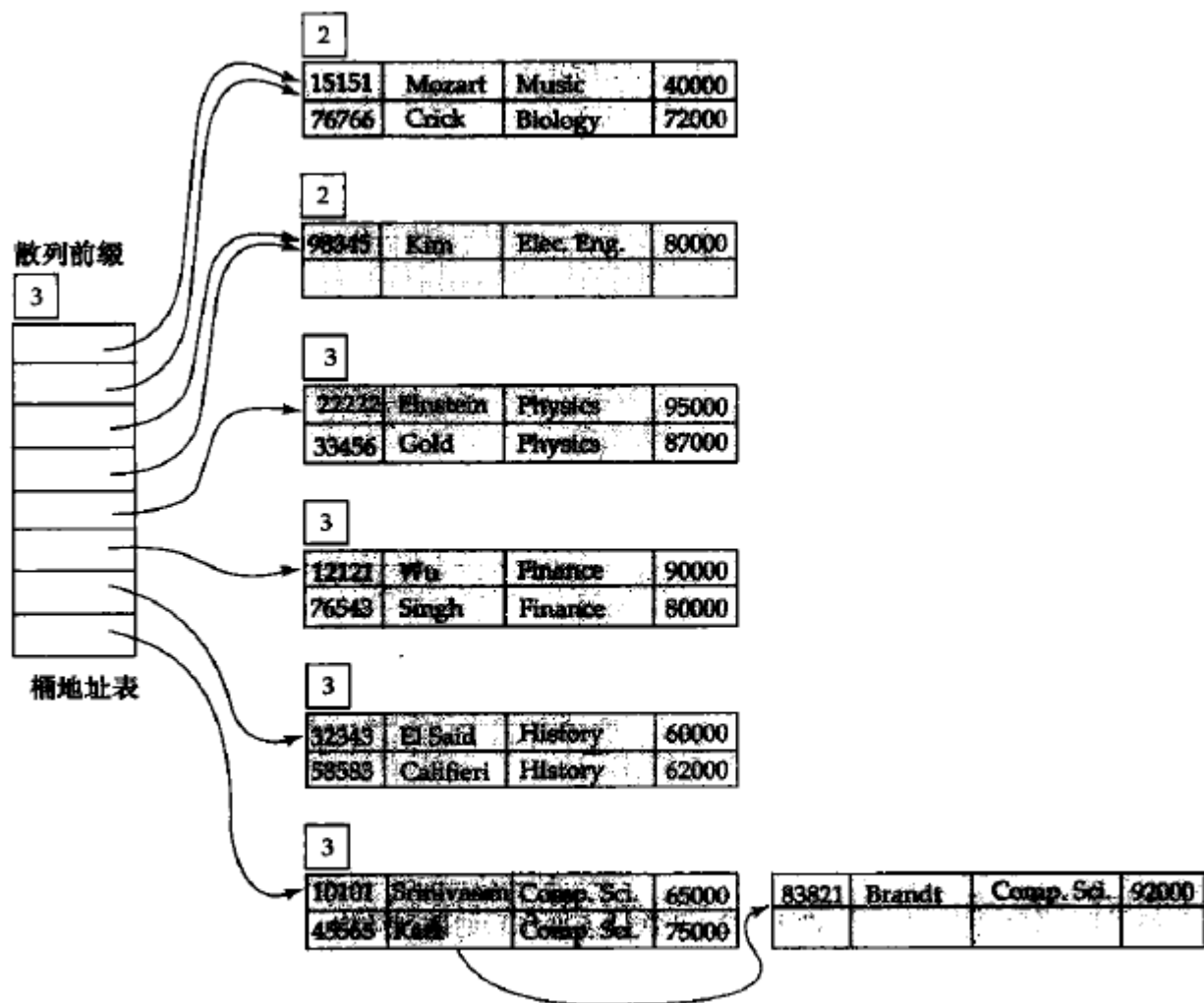


图 11-34 文件 *instructor* 的可扩充散列结构

11.7.3 静态散列与动态散列比较

可扩充散列

- 优点：
 - 性能不随文件的的增长而降低
 - 空间开销是最小的，尽管桶地址带来了额外的开销，但该表较小
- 缺点：
 - 查找涉及一个附加的间接层，对性能有微小的影响

11.8 顺序索引和散列的比较

普通查询散列更有优势

```
select A1, A2, ..., An
from r
where Ai = c;
```

范围查询顺序索引更可取


```

select A1, A2, ..., An
from r
where A1 ≤ c1 and Ai ≥ ci

```

11.9 位图索引

位图索引是一种为多码上的简单查询设计的特殊索引，尽管每个位图索引都是建立在一个码之上的

11.9.1 位图索引结构

位图就是位的一个简单数组

关系r的属性A上的**位图索引**是由A能取的每个值建立的位图构成的

example:

gender: mffmf income_level: 12143

record number				Bitmaps for <i>gender</i>		Bitmaps for <i>income_level</i>	
	<i>ID</i>	<i>gender</i>	<i>income_level</i>	m	f	L1	L2
0	76766	m	L1	10010	01101	10100	01000
1	22222	f	L2				01000
2	12121	f	L1			00001	
3	15151	m	L4			00010	
4	58583	f	L3			00000	

查询 where gender = 'f' and income_level = 'L2'

我们执行两个位图的**交**操作，得到位图01000

统计满足条件的元组数时，甚至可以在不需要访问关系的条件下就可以从位图索引中得到需要的结构

11.11 总结

见P300

第十二章 查询处理

查询处理：从数据库中提取数据时涉及的一系列活动

12.1 概述

查询处理步骤如图所示，基本步骤包括：语法分析与翻译、优化、执行

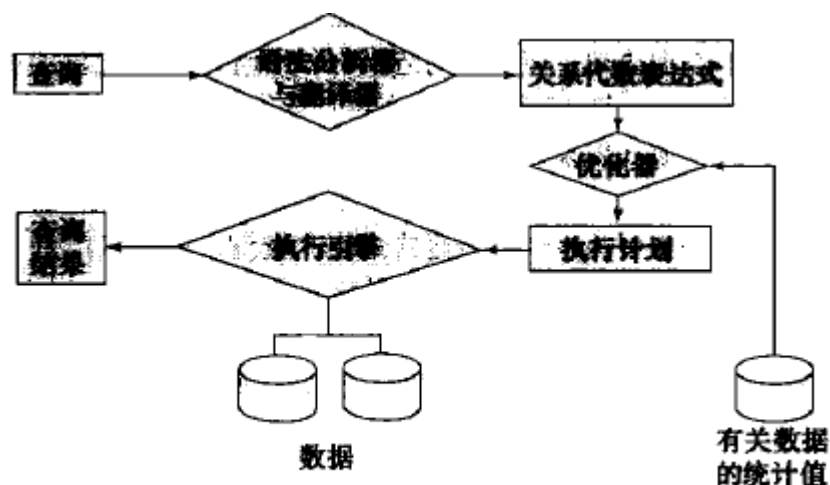


图 12-1 查询处理的步骤

查询处理开始之前，系统必须将查询语句翻译成可使用的形式

给定一个查询，一般会有多种计算结果的方法

example:

```

select salary
from instructor
where salary < 75000;
  
```

该查询语句可翻译成下面两个关系代数表达式中的任意一个：

- $\sigma_{salary < 75000} (\Pi_{salary} (instructor))$
- $\Pi_{salary} (\sigma_{salary < 75000} (instructor))$

计算原语：加了“如何执行”注释的关系代数运算

查询执行计划：执行一个查询的愿与操作序列

查询执行引擎：接受一个查询执行计划，执行该计划并把结果返回给查询

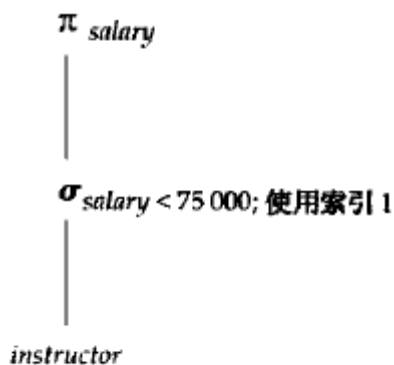


图 12-2 一个查询执行计划

12.2 查询代价的度量

在数据库系统中，最主要的代价是在磁盘上存取数据的代价

传送磁盘块数以及搜索磁盘次数被用来度量查询计算机化的代价

- 传输一个块的数据平均消耗 t_T 秒
- 磁盘块平均访问时间 t_S 秒

显然一次传输 b 块且执行 S 次磁盘搜索将消耗 $b * t_T + S * t_S$ 秒

一个查询计算计划**响应时间**就是所有的这些开销，并且可以作为计划的代价的度量

12.3 选择运算

在查询处理中，**文件扫描**是存取数据最低级的操作

12.3.1 使用文件扫描和索引的选择

- **A1 (线性搜索)：**系统扫描每一个文件块，对所有记录都进行测试

使用索引的搜索算法称为**索引扫描**

- **A2 (主索引，码属性等值比较)：**使用索引检索到满足相应等值条件的唯一一条记录
- **A3 (主索引，非码属性等值比较)：**利用主索引检索到多条记录，与A2相比需要取多条记录
- **A4 (辅助索引，等值比较)：**

若等值条件是码属性上的，则该策略可检索到唯一一条记录

这种情况下代价与A2相似

若索引字段是非码属性的，则可能检索到多条记录

这种情况下每检索一条记录都需要一次I/O操作（搜索+磁盘块传输）

- (A5, A6不重要不管)

	算 法	开 销	原 因
A1	线性搜索	$t_i + b_r * t_r$	一次初始搜索加上 b_r 个块传输， b_r 表示在文件中的块数量
A1	线性搜索，码属性等值比较	平均情形 $t_i + (b_r/2) * t_r$	因为最多一条记录满足条件，所以只要找到所需的记录，扫描就可以终止。在最坏的情形下，仍需要 b_r 个块传输
A2	B* 树主索引，码属性等值比较	$(h_i + 1) * (t_r + t_i)$	(其中 h_i 表示索引的高度)。索引查找穿越树的高度，再加上一次 I/O 来取记录；每个这样的 I/O 操作需要一次搜索和一次块传输
A3	B* 树主索引，非码属性等值比较	$h_i * (t_r + t_i) + b * t_r$	树的每层一次搜索，第一个块一次搜索。 b 是包含具有指定搜索码记录的块数。假定这些块是顺序存储(因为是主索引)的叶子块并且不需要额外搜索
A4	B* 树辅助索引，码属性等值比较	$(h_i + 1) * (t_r + t_i)$	这种情形和主索引相似
A4	B* 树辅助索引，非码属性等值比较	$(h_i + n) * (t_r + t_i)$	(其中 n 是所取记录数。)索引查找的代价和 A3 相似，但是每条记录可能在不同的块上，这需要每条记录一次搜索。如果 n 值比较大，代价可能会非常高
A5	B* 树主索引，比较	$h_i * (t_r + t_i) + b * t_r$	和 A3，非码属性等值比较情形一样
A6	B* 树辅助索引，比较	$(h_i + n) * (t_r + t_i)$	和 A4，非码属性等值比较情形一样

图 12-3 选择算法代价估计

12.4 排序

数据排序有重要作用，有两个原因：

1. SQL查询会指明对结果进行排序
2. 当输入的关系已经排序时，关系运算中的一些运算（如连接运算）能够得到高效实现

12.4.1 外部排序归并算法

外排序：不能全部放在内存中的关系的排序

M表示内存缓冲区中可以用于排序的块数，即内存的缓冲区能容纳的磁盘块数

1. 第一阶段，建立多个排好序的归并段，每个归并段都是排序过的，但仅包含关系中的部分记录

```

i = 0;
repeat
    读入关系的 M 块数据或剩下的不足 M 块的数据;
    在内存中对关系的这一部分进行排序;
    将排好序的数据写到归并段文件 Ri 中;
    i = i + 1;
until 到达关系末尾
    
```

2. 第二阶段，对归并段进行归并。假定归并段总数 $N < M$ ，为 N 个归并段文件 R_i 各分配一个内存缓冲块，并分别读入一个数据块;

```

repeat
    在所有缓冲块中按序挑选第一个元组;
    将该元组作为输出写出，并将其从缓冲块中删除;
    if 任何一个归并段文件 Ri 的缓冲块为空并且没有到达 Ri 末尾
        then 读入 Ri 的下一块到相应的缓冲块
until 所有的缓冲块均空
    
```

该算法对 N 个归并段进行归并，因此它被称为 **N路归并**

一般而言，关系比内存大得多，即 $N \gg M$ ，因此归并要分多趟进行，每趟可以用 $M-1$ 个归并段作为输入。这样每趟下来，归并段的数目都减少到原来的 $1/(M-1)$ ，直至归并段数目小于 M 得到排序的输出结果

example: $M=3$ (两块用于输入，一块用于输出)

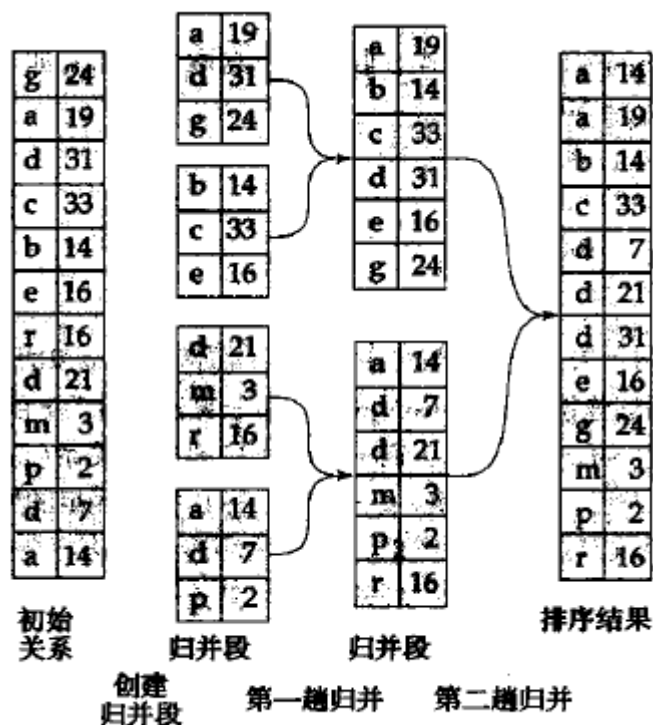


图 12-4 使用归并排序的外排序

12.4.2 外部排序归并的代价分析

b_r 代表包含关系 r 中记录的磁盘块数

磁盘块传输：

第一阶段：

- 磁盘块传输： $2b_r$ 次

第二阶段：

- 归并趟数： $\lceil \log_{M-1}(b_r/M) \rceil$
- 每一趟归并读写各一次
- 最后一趟归并可以只产生结果而不写入磁盘

磁盘块传输总数： $2b_r + b_r(2 * \lceil \log_{M-1}(b_r/M) \rceil - 1) = b_r(2\lceil \log_{M-1}(b_r/M) \rceil + 1)$

磁盘搜索：

第一阶段：

- 读取写回归并段： $2\lceil b_r/M \rceil$

第二阶段：

- 每次从一个归并段读取 b_b 块数据，每趟归并需要作 $\lceil b_r/b_b \rceil$ 次磁盘搜索以读取数据
- 假设输出阶段也分配了 b_b 个块，每一趟可以归并 $\lfloor M/b_b \rfloor - 1$ 个归并段

磁盘搜索总次数： $2\lceil b_r/M \rceil + \lceil b_r/b_b \rceil(2\lceil \log_{\lfloor M/b_b \rfloor - 1}(b_r/M) \rceil - 1)$

(本质上就是第二阶段的M和 b_r 都除以 b_b)

12.5 连接运算

等值连接：表示形如 $r \bowtie_{r.A=s.B} s$ 的连接，其中A,B分别为r与s的属性或属性组

- student记录数： $n_{student} = 5000$
- student磁盘块数： $b_{student} = 100$
- takes记录数： $n_{takes} = 10000$
- takes磁盘块数： $b_{takes} = 400$

12.5.1 嵌套循环连接

该算法主要由两个嵌套的for循环构成，因此被称为**嵌套循环连接**

```
for each 元组  $t_i$  in  $r$  do begin
    for each 元组  $t_j$  in  $s$  do begin
        测试元组对 $(t_i, t_j)$ 是否满足连接条件  $\theta$ 
        如果满足，把  $t_i \cdot t_j$  加到结果中
    end
end
```

图 12-5 嵌套循环连接

对于 $r \bowtie s$ ， r 被称为连接的**外层关系**， s 被称为连接的**内层关系**

- r,s 中元组数： n_r, n_s
- 包含关系 r,s 中元组的磁盘块数： b_r, b_s
- 需要考虑的元组对数目： $n_r * n_s$
- 对关系 r 中的每一条记录，我们必须对 s 作一次完整的扫描

	最坏情况	最好情况
块传输	$n_r * b_s + b_r$	$b_r + b_s$
磁盘搜索	$n_r + b_r$	2

12.5.2 块嵌套循环连接

```

for each 块  $B_r$  of  $r$  do begin
  for each 块  $B_s$  of  $s$  do begin
    for each 元组  $t_r$  in  $B_r$  do begin
      for each 元组  $t_s$  in  $B_s$  do begin
        测试元组对  $(t_r, t_s)$  是否满足连接条件
        如果满足, 把  $t_r \cdot t_s$  加到结果中
      end
    end
  end
end
end

```

图 12-6 块嵌套循环连接

块嵌套循环连接是嵌套循环连接的一个变种，内层关系的每一块与外层关系的每一块形成一对

	最坏情况	最好情况
块传输	$b_r * b_s + b_r$	$b_r + b_s$
磁盘搜索	$2b_r$	2

嵌套循环与块嵌套循环算法的性能可以进一步地改进：

- 如果自然连接或等值连接中的连接属性是内层关系的码，则对每个外层关系元组，内层循环一旦找到了首条匹配元组就可以终止。
- 在块嵌套循环连接算法中，外层关系可以不用磁盘块作为分块的单位，而以内存中最多能容纳的大小为单位，当然同时要留出足够的缓冲空间给内层关系及输出结果使用。也就是说，如果内存有 M 块，我们一次读取外层关系中的 $M - 2$ 块，当我们读取到内层关系中的每一块时，我们把它与外层关系中的所有 $M - 2$ 块做连接。这种改进使内层关系的扫描次数从 b_s 次减少到 $\lceil b_s / (M - 2) \rceil$ 次，这里的 b_s 是外层关系所占的块数。这样全部代价为 $\lceil b_s / (M - 2) \rceil * b_r + b_r$ 次块传输和 $2\lceil b_s / (M - 2) \rceil$ 次磁盘搜索。
- 对内层循环轮流做向前、向后的扫描。该扫描方法对磁盘块读写请求进行排序，使得上一次扫描时留在缓冲区中的数据可以重用，从而减少磁盘存取次数。
- 若内层循环的连接属性上有索引，可以用更有效的索引查找法替代文件扫描法。这一改进在 12.5.3 节讲述。

12.5.3 索引嵌套循环连接

(不管了)

12.6 其他运算

12.6.1 去除重复

用排序方法可以很容易地实现去除重复

12.6.2 投影

对每个元组作投影，所得结果关系可能有重复记录，然后去除重复记录

12.7 表达式计算

12.7.1 物化

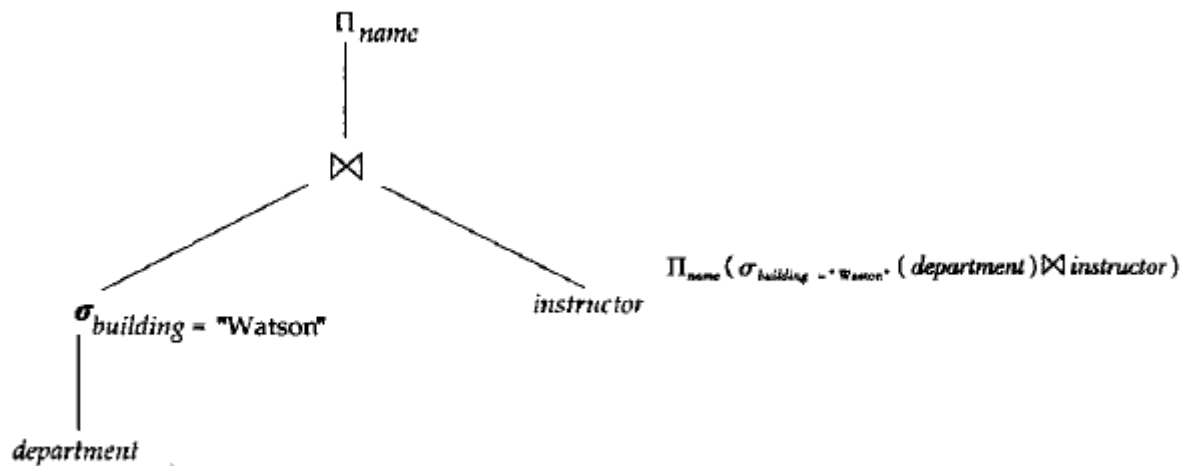


图 12-11 一个表达式的图形化表示

运算符数：对表达式做图形化表示

当采用物化方法时，从表达式的最底层的运算（在树的底部）开始

物化计算：运算的每个中间结果被创建（物化），然后用于下一层的运算

12.7.2 流水线

流水线计算：将多个关系操作组合成一个操作的流水线来实现，其中一个操作的结果将传送到下一个操作

优势：

1. 消除读和写临时关系的代价，减少查询计算代价
2. 如果一个查询计算计划的根操作符及其输入合并到流水线中，那么可以迅速开始产生查询结果

12.7.2.1 流水线的实现

1. *需求驱动的流水线*，系统不停地向位于流水线顶端的操作发出需要元组的请求
2. *生产者驱动的流水线*，各操作不等待元组请求，而是积极地产生元组

12.8 总结

见P325

第十三章 查询优化

查询优化：从给定查询的多种可能策略中找出最有效的查询执行计划的一种处理过程

13.1 概述

example:

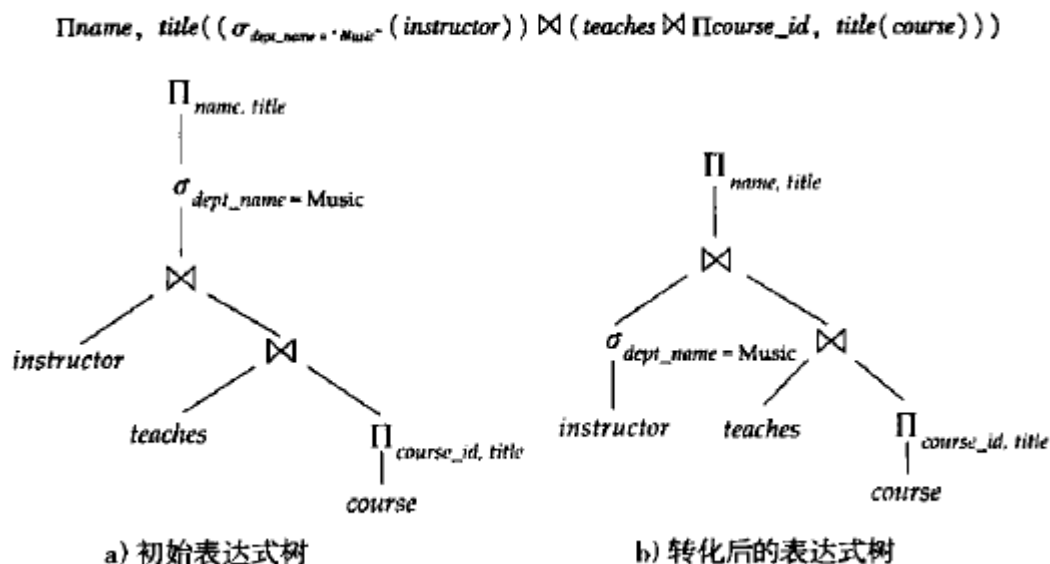


图 13-1 等价表达式

给定一个关系代数表达式，查询优化器的任务是产生一个查询执行计划，该计划能获得与原关系表达式相同的结果，并且得到结果集的执行代价最小

查询执行计划的产生步骤：

1. 产生逻辑上与给定表达式等价的表达式
2. 对所产生的表达式以不同方式作注释，产生不同的查询计划
3. 估计每个执行计划的代价，选择代价最小的一个

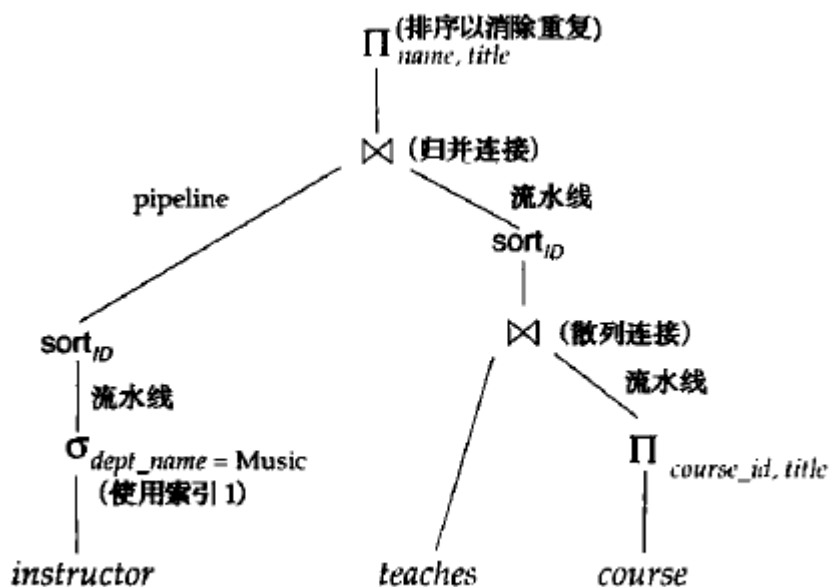


图 13-2 一个执行计划

13.2 关系表达式的转换

如果两个关系表达式在每一个有效数据库实例中都会产生相同的元组集，则我们称它们是**等价的**

13.2.1 等价规则

等价规则指出两种不同形式的表达式是等价的

1. $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
2. $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
3. $\Pi_{L_1}(\Pi_{L_2}(\dots(\Pi_{L_n}(E))\dots)) = \Pi_{L_1}(E)$

4. a. $\sigma_{\theta}(E_1 \bowtie E_2) = E_1 \bowtie_{\theta} E_2$
 b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$
5. $E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$
6. $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3 .

7. $\sigma_{\theta_0}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_{\theta} E_2$
 $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta} (\sigma_{\theta_2}(E_2))$
8. $\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) = (\Pi_{L_1}(E_1)) \bowtie_{\theta} (\Pi_{L_2}(E_2))$
 $\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_{\theta} (\Pi_{L_2 \cup L_4}(E_2)))$
9. $E_1 \cup E_2 = E_2 \cup E_1$
 $E_1 \cap E_2 = E_2 \cap E_1$
10. $(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$
 $(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$
11. $\sigma_{\theta}(E_1 - E_2) = \sigma_{\theta}(E_1) - \sigma_{\theta}(E_2)$ $\sigma_{\theta}(E_1 - E_2) = \sigma_{\theta}(E_1) - E_2$
12. $\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$

1. 合取选择运算可分解为单个选择运算的序列。该变换称为 σ 的级联：

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. 选择运算满足交换律(commutative)：

$$\sigma_{\theta}(\sigma_{\theta_1}(E)) = \sigma_{\theta_1}(\sigma_{\theta}(E))$$

3. 一系列投影运算中只有最后一个运算是必需的, 其余的可省略。该转换也可称为 Π 的级联:

$$\Pi_{L_1}(\Pi_{L_2}(\cdots(\Pi_{L_n}(E))\cdots)) = \Pi_{L_n}(E)$$

4. 选择操作可与笛卡儿积以及 θ 连接相结合:

$$a. \sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2.$$

该表达式就是 θ 连接的定义。

$$b. \sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$$

5. θ 连接运算满足交换律:

$$E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$$

事实上, 左端和右端的属性顺序不同, 所以如果考虑属性顺序的话, 等式不成立。可以对等价规则的其中一端加入一个投影操作以适当重排属性, 但为了简化, 我们省略了该投影并且在大部分例子中忽略属性顺序。

回忆一下, 自然连接是 θ 连接的特例, 因此自然连接也满足交换律。

6. a. 自然连接运算满足结合律(associative):

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

- b. θ 连接具有以下方式的结合律:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_1} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_2} (E_2 \bowtie_{\theta_2} E_3)$$

其中 θ_2 只涉及 E_2 与 E_3 的属性。由于其中的任意一个条件都可为空, 因此笛卡儿积(\times)运算也满足结合律。连接运算满足结合律、交换律在查询优化中对于重排连接顺序是很重要的。

7. 选择运算在下面两个条件下对 θ 连接运算具有分配律:

- a. 当选择条件 θ_0 中的所有属性只涉及参与连接运算的表达式之一(比如 E_1)时, 满足分配律:

$$\sigma_{\theta_0}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_{\theta} E_2$$

- b. 当选择条件 θ_1 只涉及 E_1 的属性, 选择条件 θ_2 只涉及 E_2 的属性时, 满足分配律:

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta} (\sigma_{\theta_2}(E_2))$$

8. 投影运算在下面条件下对 θ 连接运算具有分配律:

- a. 令 L_1 、 L_2 分别代表 E_1 、 E_2 的属性。假设连接条件 θ 只涉及 $L_1 \cup L_2$ 中的属性, 则:

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) = (\Pi_{L_1}(E_1)) \bowtie_{\theta} (\Pi_{L_2}(E_2))$$

- b. 考虑连接 $E_1 \bowtie_{\theta} E_2$ 。令 L_1 、 L_2 分别代表 E_1 、 E_2 的属性集; 令 L_3 是 E_1 中出现在连接条件 θ 中但不在 $L_1 \cup L_2$ 中的属性; 令 L_4 是 E_2 中出现在连接条件 θ 中但不在 $L_1 \cup L_2$ 中的属性。那么:

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_{\theta} (\Pi_{L_2 \cup L_4}(E_2)))$$

9. 集合的并与交满足交换律。

$$E_1 \cup E_2 = E_2 \cup E_1$$

$$E_1 \cap E_2 = E_2 \cap E_1$$

集合的差运算不满足交换律。

10. 集合的并与交满足结合律。

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. 选择运算对并、交、差运算具有分配律:

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - \sigma_P(E_2)$$

类似地, 上述等价规则将“-”替换成 \cup 或 \cap 时也成立。进一步有:

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - E_2$$

上述等价规则将“-”替换成 \cap 时也成立, 但将“-”替换成 \cup 时不成立。

12. 投影运算对并运算具有分配律。

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

13.2.2 转换的例子

下面举例说明等价规则的用法。我们用大学这个例子，其关系模式如下：

Instructor(*ID*, *name*, *dept_name*, *salary*)
teaches(*ID*, *course_id*, *sec_id*, *semester*, *year*)
course(*course_id*, *title*, *dept_name*, *credits*)

在 13.1 节的例子中，表达式：

$\Pi_{name, title} (\sigma_{dept_name = \text{"Music"}} (instructor \bowtie (teaches \bowtie \Pi_{course_id, title}(course))))$

转换成以下表达式：

$\Pi_{name, title} ((\sigma_{dept_name = \text{"Music"}} (instructor)) \bowtie (teaches \bowtie \Pi_{course_id, title}(course)))$

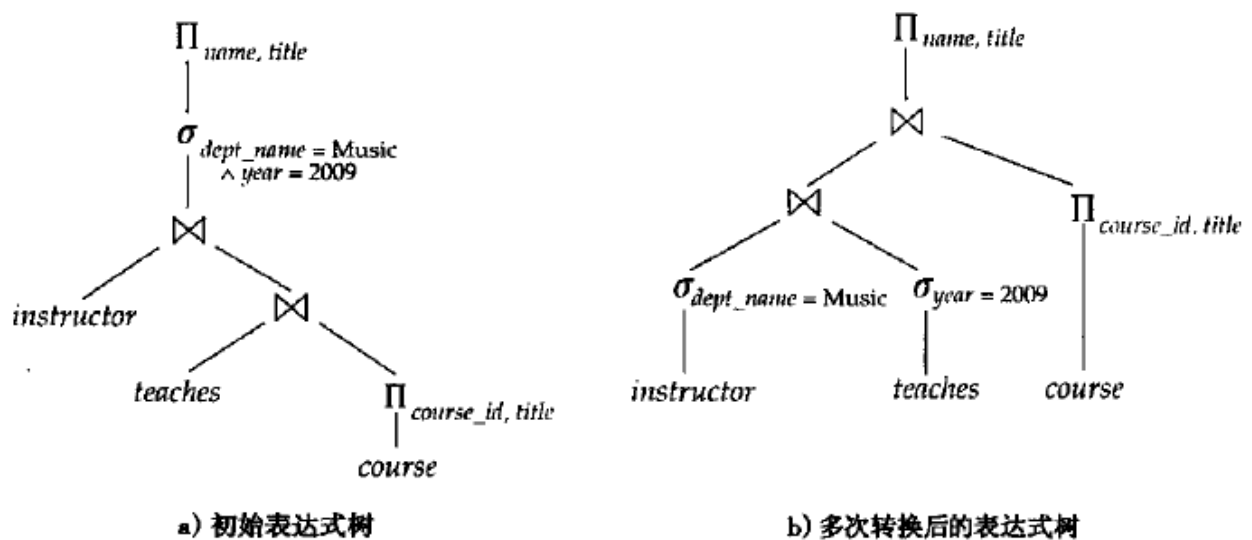


图 13-4 多次转换

见P333

13.2.3 连接的次序

一个好的连接运算次序对于减少临时结果的大小是很重要的

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

虽然这两个表达式等价，但是计算它们的代价可能不同

13.2.4 等价表达式的枚举

```

procedure genAllEquivalent( $E$ )
 $EQ = \{E\}$ 
repeat
    将  $EQ$  中的每条表达式  $E_i$  与每条等价规则  $R_j$  进行匹配
    如果  $E_i$  的某个子表达式与  $R_j$  的某一边相匹配
        生成一个与  $E_i$  等价的  $E'$ , 其中  $e_i$  被替换成  $R_j$  的另一边
        如果  $E'$  不在  $EQ$  中, 则将其添加到  $EQ$  中
until 不再有新的表达式可以添加到  $EQ$  中

```

图 13-5 产生所有等价表达式的过程

13.3 表达式结果集统计大小的估计

13.3.1 目录信息

- n_r : 关系 r 的元组数
- b_r : 包含关系 r 中元组的磁盘块数
- l_r : 关系 r 中每个元组的字节数
- f_r : 关系 r 的块引子——一个磁盘块能容纳关系 r 中元组的个数
- $V(A, r)$: 关系 r 中属性 A 中出现的非重复值个数

显然有等式 $b_r = \lceil n_r / f_r \rceil$

等宽直方图: 取值范围分成相等大小的区间

等深直方图: 每个区间的取值个数相同

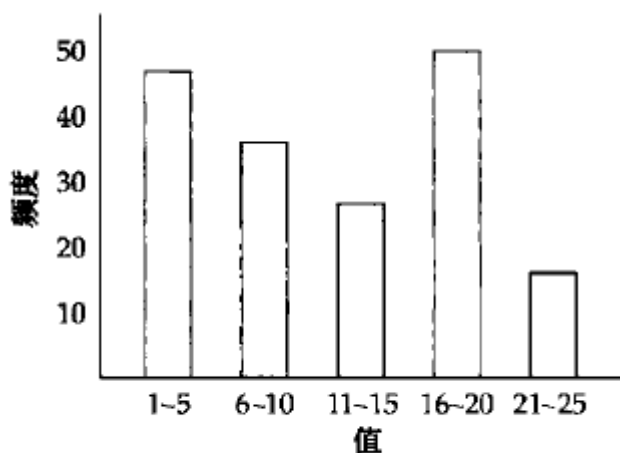


图 13-6 直方图示例

13.3.2 选择运算结果大小的估计

- $\sigma_{A=a}(r)$: $n_r / V(A, r)$ 个元组
- $\sigma_{A \leq a}(r)$: $n_r \cdot \frac{v - \min(A, r)}{\max(A, r) - \min(A, r)}$ 个元组
- 复杂选择:
 - 合取:

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r).$$

关系中的一个元组满足条件选择 θ_i 的概率为 s_i/n ，这概率被称为选择 $\sigma_{\theta_i}(r)$ 的**中选率**
合取的元组数量：

$$n_r * \frac{s_1 * s_2 * \dots * s_n}{n_r^n}$$

◦ 析取：

$$\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r).$$

析取的元组数量：

$$n_r * \left(1 - \left(1 - \frac{s_1}{n_r} \right) * \left(1 - \frac{s_2}{n_r} \right) * \dots * \left(1 - \frac{s_n}{n_r} \right) \right)$$

◦ 取反： $\sigma_{-\theta}(r)$

取反的元组数量： $n_r - \sigma_{\theta}(r)$

13.3.3 连接运算结果大小的估计

对于 $r \bowtie s$ ，令 $r(R)$ 和 $s(S)$ 为两个关系

- 若 $R \cap S = \emptyset$ ，则两关系没有共同属性，此时 $r \bowtie s = r \times s$
- 若 $R \cap S$ 是 R 或 S 的码，则此时得到元组数与 r 或 s 中的元组数相同
- 若 $R \cap S$ 既不是 R 也不是 S 的码，假定 $R \cap S = \{A\}$ ，此时应该有 $(n_r \times n_s)/V(A, s)$ 个元组

13.3.4 其它运算的结果集大小的估计

- 投影：估计为 $V(A, r)$ ，因为投影去除了重复元组
- 聚集：
： $G_f(r)$ 的大小是 $V(r)$ ，因为对 A 的任意一个不同取值在 $G_f(r)$ 中有一个元组与其对应。
- 集合运算：可以重写为合取、析取以及取反进行计算
- 外连接：
外连接： $r \Join s$ 的大小估计为 $r \bowtie s$ 的大小加上关系 r 的大小；对 $r \ltimes s$ 的估计与对 $r \Join s$ 的估计是对称的，而 $r \oslash s$ 的估计为 $r \bowtie s$ 的大小加上关系 r 和关系 s 的大小。以上三种估计可能不精确，但提供了结果集大小的上界。

13.4 执行计划选择

基于代价的优化器从给定查询等价的所有查询执行计划空间中进行搜索，并选择估计代价最小的一个

13.4.1 基于代价的连接顺序选择

考虑表达式 $r_1 \bowtie r_2 \bowtie \dots \bowtie r_n$ ，当 $n=3$ 时，有12种不同的连接顺序

$$\begin{array}{cccc}
r_1 \bowtie (r_2 \bowtie r_3) & r_1 \bowtie (r_3 \bowtie r_2) & (r_2 \bowtie r_3) \bowtie r_1 & (r_3 \bowtie r_2) \bowtie r_1 \\
r_2 \bowtie (r_1 \bowtie r_3) & r_2 \bowtie (r_3 \bowtie r_1) & (r_1 \bowtie r_3) \bowtie r_2 & (r_3 \bowtie r_1) \bowtie r_2 \\
r_3 \bowtie (r_1 \bowtie r_2) & r_3 \bowtie (r_2 \bowtie r_1) & (r_1 \bowtie r_2) \bowtie r_3 & (r_2 \bowtie r_1) \bowtie r_3
\end{array}$$

对于

$$(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$$

们需要检查 144 种连接顺序。然而，一旦我们给关系子集 $\{r_1, r_2, r_3\}$ 找到了最佳连接顺序，我们可以用此顺序进一步与 r_4 、 r_5 进行连接，舍弃 $r_1 \bowtie r_2 \bowtie r_3$ 中其他代价较大的连接顺序。这样，我们不必一一检查 144 种连接顺序，而只须检查 12 + 12 种顺序。

13.4.3 启发式优化

查询优化器使用**启发式方法**来减少优化代价

对关系代数查询进行转换的例子：

- 尽早执行选择运算
- 尽早执行投影运算

左深连接顺序用于流水线计算特别方便，因为右操作对象是一个已存储的关系，每个连接只有一个输入来自流水线

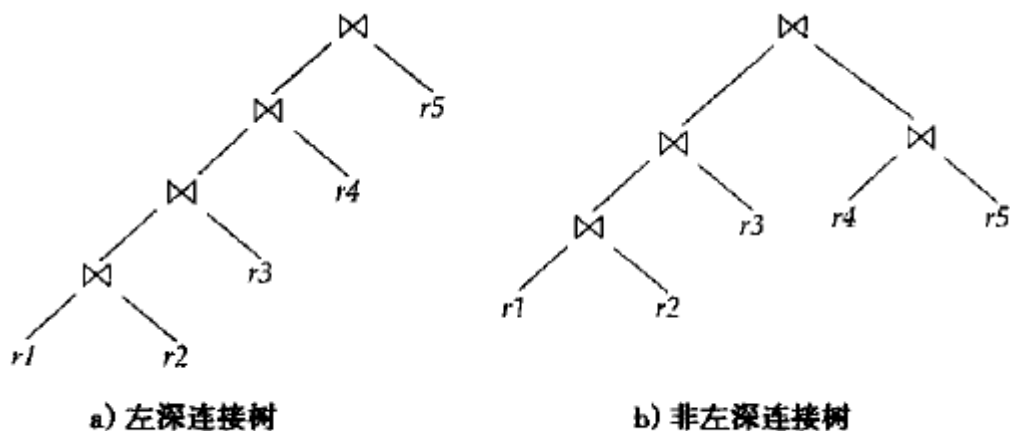


图 13-8 左深连接树

许多优化器允许为查询优化指定**优化成本预算**，当超过预算时停止搜索，返回当前找到的最优计划

13.7 总结

见P350

第四部分 事务管理

第十四章 事务

事务：构成单一逻辑操作单元的操作集合

数据库系统要么执行整个事务，要么属于该事务的操作一个也不执行

14.1 事务概念

事务是访问并可能更新各种数据项的一个程序执行单元

事务的性质（ACID特性）：

- **原子性(atomicity)**：事务的所有操作在数据库中要么全部正确反映出来，要么完全不反映
- **一致性(consistency)**：隔离执行事务时，保持数据库的一致性
- **隔离性(isolation)**：每个事务感觉不到系统中有其它事务在并发地执行
- **持久性(durability)**：一个事务成功完成后，对数据库的改变是永久的，即使出现系统故障

14.2 一个简单的事务模型

事务运用以下两个操作访问数据：

- **read(X)**：从数据库把数据库项X传送到执行**read**操作的事务的主存缓冲区的一个也称为X的变量中
- **write(X)**：从执行**write**的事务的主存缓冲区的变量X中把数据项X传回数据库中

```
Ti : read( A ) ;  
      A := A - 50 ;  
      write( A ) ;  
      read( B ) ;  
      B := B + 50 ;  
      write( B ).
```

14.3 存储结构

- **易失性存储器**：其中的信息在系统崩溃后不会幸存
- **非易失性存储器**：其中的信息在系统崩溃后幸存
- **稳定性存储器**：其中的信息永远不会丢失

14.4 事务原子性和持久性

中止(aborted)：事务没有成功地执行完成

回滚(rolled back)：中止事务造成的变更被撤销

日志(log)：负责管理事务中止

事务的状态：

- **活动的(active)**：初始状态，事务执行时处于这个状态
- **部分提交的(partially committed)**：最后一条语句执行后
- **失败的(failed)**：发现正常的执行不能继续后
- **中止的(aborted)**：事务回滚并且数据库已恢复到事务开始执行前的状态后
- **提交的(committed)**：成功完成后

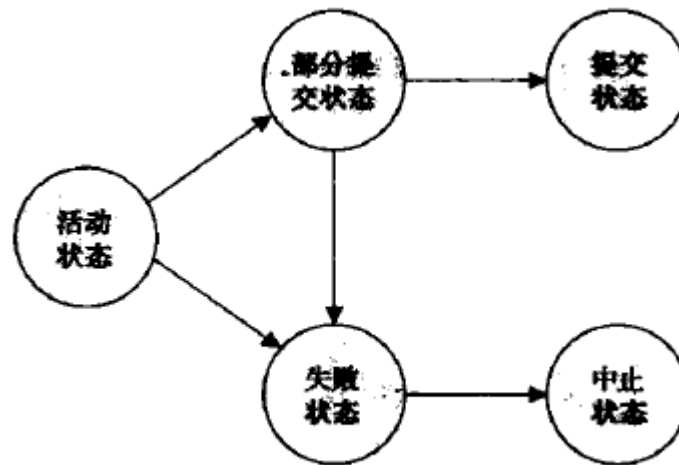


图 14-1 事务状态图

如果事务是提交的或中止的，它称为**已经结束的(terminated)**

事务进入中止状态后，系统有两种选择：

- 重启(restart)事务：当引起事务中止的是硬件错误，或不是由事务的内部逻辑所产生的软件错误时
- 杀死(kill)事务：事务的内部逻辑造成错误

14.5 事务隔离性

事务并发的优点：

- 提高吞吐率和资源利用率
- 减少等待时间

调度(schedule)：指令在系统中执行的时间顺序

串行的(serial)：每个串行调度由来自各事务的指令序列组成，其中属于同一事务的指令在调度中紧挨在一起

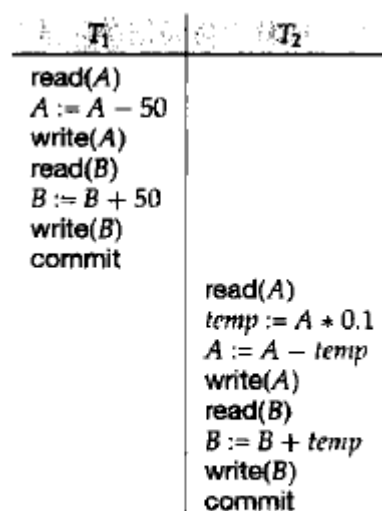


图 14-2 调度 1：一个串行调度， T_2 跟在 T_1 之后

可串行化调度：在并发执行中，通过保证所执行的任何调度的效果都与没有并发执行的调度效果一样，我们可以确保数据库的一致性。调度应该在某种意义上等价于一个串行调度。

14.6 可串行化

I与J的冲突:

- 1. $I_i = \text{read}(Q), I_j = \text{read}(Q)$. I_i and I_j don't conflict.
- 2. $I_i = \text{read}(Q), I_j = \text{write}(Q)$. They conflict.
- 3. $I_i = \text{write}(Q), I_j = \text{read}(Q)$. They conflict
- 4. $I_i = \text{write}(Q), I_j = \text{write}(Q)$. They conflict

只有都为I与都为read的时候，执行顺序才是无关紧要的

冲突等价: 如果调度S可以经过一系列非冲突指令交换成S'，则S与S'是**冲突等价**的

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

图 14-6 调度 3: 只显示 read 与 write 指令 图 14-8 调度 6: 与调度 3 等价的一个串行调度

若一个调度S与一个串行调度冲突等价，则称调度S是**冲突可串行化的**

优先图: S是一个调度，由S构造一个有向图

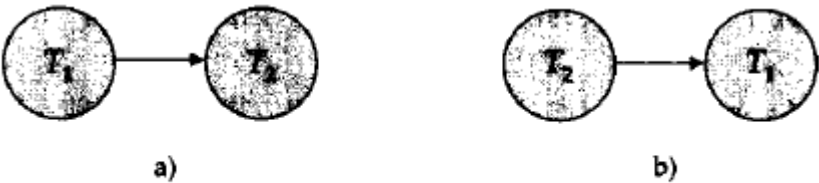


图 14-10 a) 调度 1 与 b) 调度 2 的优先图

串行化顺序可通过**拓扑排序**得到，拓扑排序用于计算与优先图的偏序相一致的线性顺序

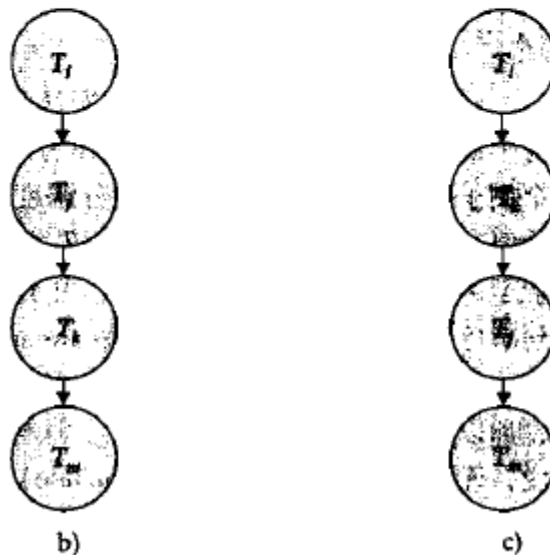
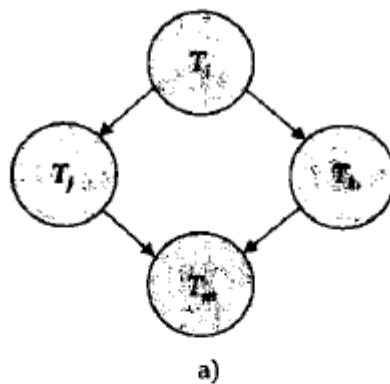


图 14-12 拓扑排序示例

14.7 事务隔离性和原子性

14.7.1 可恢复调度

若 T_j 读取了 T_i 写入的值，则 T_j 依赖于 T_i

可恢复调度：对于每对事务 T_i 和 T_j ，如果 T_j 读取了之前由 T_i 所写的的数据项，则 T_i 先于 T_j 提交

不可恢复调度例子：

T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

14.7.2 无级联调度

级联回滚：因单个事务故障导致一系列事务回滚

无级联调度：对于每对事务 T_i 和 T_j ，如果 T_j 读取了先前由 T_i 所写的的数据项，则 T_i 必须在 T_j 这一操作前提交

14.8 事务隔离性级别

SQL标准规定的隔离性级别如下：

- **可串行化(serializable)**：通常保证可串行化调度。然而，正如我们将来解释的，一些数据库系统对该隔离性级别的实现在某些情况下允许非可串行化执行。
- **可重复读(repeatable read)**：只允许读取已提交数据，而且在一个事务两次读取一个数据项期间，其他事务不得更新该数据。但该事务不要求与其他事务可串行化。例如：当一个事务在查找满足某些条件的数据时，它可能找到一个已提交事务插入的一些数据，但可能找不到该事务插入的其他数据。
- **已提交读(read committed)**：只允许读取已提交数据，但不要求可重复读。比如，在事务两次读取一个数据项期间，另一个事务更新了该数据并提交。
- **未提交读(read uncommitted)**：允许读取未提交数据。这是 SQL 允许的最低一致性级别。

以上所有隔离性级别都不允许**脏写**，即一个数据项已经被另外一个尚未提交或中止的事务写入，则不允许对该数据项执行写操作

14.9 隔离性级别的实现

- 锁
- 时间戳
- 多版本和快照隔离

14.11 总结

见P370

第十五章 并发控制

15.1 基于锁的协议

确保隔离性的方法之一是要求对数据项以**互斥**的方式进行访问

15.1.1 锁

1. **共享型锁**：lock-S(Q)，事务Ti可读但不能写数据项Q
2. **排他型锁**：lock-X(Q)，事务Ti既可读也能写数据项Q

事务根据自己对Q的操作类型**申请**适当的锁，等到并发控制管理器**授予**所需锁才能继续操作

锁相容性矩阵：

	S	X
S	true	false
X	false	false

若数据项已被另一事务加上了不相容类型的锁，则Ti只好**等待**直到锁被释放

死锁：两个事务都不能正常执行的状态

死锁发生时，系统必须回滚两个事务中的一个，一旦某个事务回滚，该事务回滚的数据项就被解锁

封锁协议：规定事务何时对数据项们进行加锁、解锁

封锁协议**保证**冲突可串行性 \iff 所有合法的调度都是冲突可串行化的（关联的 \rightarrow 关系无环）

15.1.2 锁的授予

饿死：事务 T_i 永远得不到进展，因为总是得不到锁

避免饿死的加锁方式：

1. 不存在在数据项 Q 上持有与 M 型锁冲突的锁的其它事务
2. 不存在等待对数据项 Q 加锁且先于 T_i 申请加锁的事务

此时一个加锁请求就不会被其后的加锁申请阻塞

15.1.3 两阶段封锁协议

两阶段封锁协议要求每个事务分两个阶段提出加锁和解锁申请：

1. **增长阶段：**事务可以获得锁，但不能释放锁
2. **缩减阶段：**事务可以释放锁，但不能获得锁

封锁点：调度中事务获得最后加锁的位置（增长阶段结束点）

严格两阶段封锁可以避免级联回滚，要求所有排他锁必须在事务提交后才能释放

强两阶段封锁要求事务提交之前不得释放任何锁

任何形式的两阶段封锁协议只产生冲突可串行化的调度

锁转换：

- 升级：共享锁 \rightarrow 排他锁
- 降级：排他锁 \rightarrow 共享锁
- 锁的升级只能发生在增长阶段，锁的降级只能发生在缩减阶段
- 当事务 T_i 进行 **read(Q)** 操作时，系统就产生一条 **lock-S(Q)** 指令，该 **read(Q)** 指令紧跟其后。
- 当事务 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。

15.1.4 封锁的实现

锁管理器可以实现为一个过程，它从事务接收消息并反馈消息

锁表：以数据项名称为索引的散列表来查找链表中的数据项

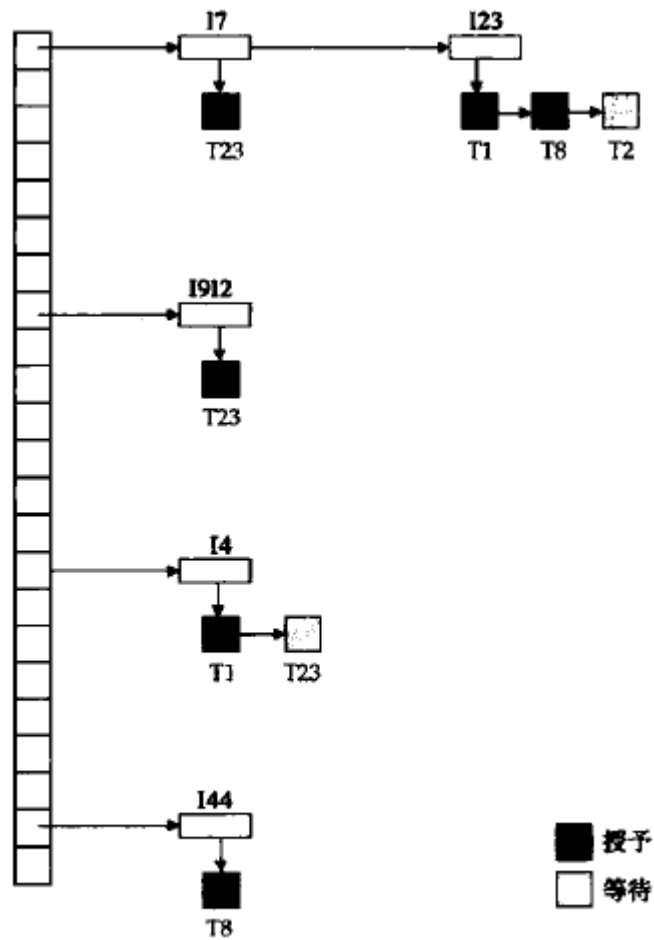


图 15-10 锁表

15.1.5 基于图的协议

数据库图：数据项集合D满足偏序关系，故可以视为一个有向无环图

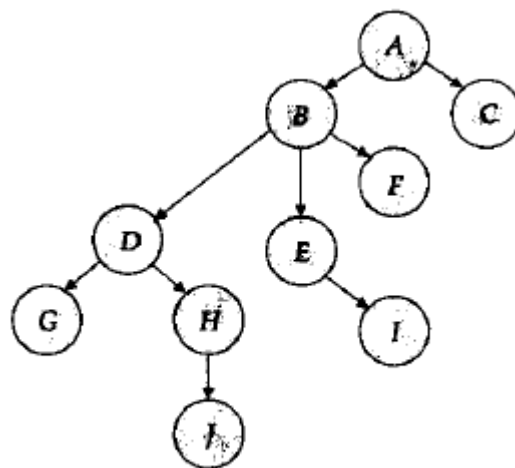


图 15-11 树形结构数据库图

树形协议：可用的加锁指令只有lock-X，并且要遵循以下规则：

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是冲突可串行化的。

例子:

T_{10} : **lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock-X(G); unlock(D); unlock(G).**

可以通过记录数据项最后被哪个事务修改而保证可恢复性

树形协议不产生死锁, 因此不需要回滚

树形协议可以在较早时候释放锁, 减少等待时间增加了并发性

缺点: 事务可能必须为其不需要的数据项加锁

15.2 死锁处理

处理死锁问题的两种方法:

1. **死锁预防**: 保证系统永不进入死锁状态
2. 允许系统进入死锁状态, 然后试着用**死锁检测**和**死锁恢复**机制进行恢复

15.2.1 死锁预防

预防死锁的两种方法:

1. 对加锁请求进行排序或要求同时获得所有的锁来保证不会发生循环等待
2. 每当等待有可能导致死锁时, 进行回滚而不是等待加锁

- 第一种方法: 要么一次全部封锁, 要么全不封锁

缺点:

- 很难预知哪些数据项要被使用
- 数据项使用率可能很低

另一种机制: 对数据项强加一个次序 (树形协议偏序)

- 第二种方法: 抢占与事务回滚

通过回滚事务 T_i 来将抢占其持有的锁, 并将其授予 T_j

通过**时间戳**来决定事务应当等待还是回滚

两种机制:

1. **wait-die**机制基于非抢占技术

当 T_i 时间戳小于 T_j 时 (T_i 比 T_j 老), 允许 T_i 等待, 否则 T_i 回滚 (死亡)

2. **wound-wait**机制基于抢占技术

当 T_i 时间戳大于 T_j 时 (T_i 比 T_j 年轻), 允许 T_i 等待, 否则 T_j 回滚 (T_j 被 T_i 伤害)

- 第三种方法: 锁超时

申请锁的事务至多等待一段给定的时间, 若超时则回滚并重启

15.2.2 死锁检测与恢复

为实现检测与恢复机制，系统必须：

- 维护当前将数据项分配给事务的有关信息，以及任何尚未解决的数据项请求信息。
- 提供一个使用这些信息判断系统是否进入死锁状态的算法。
- 当检测算法判定存在死锁时，从死锁中恢复。

15.2.2.1 死锁检测

死锁可以用等待图来精确描述

当事务 T_i 申请的数据项当前被 T_j 持有时，边 $T_i \rightarrow T_j$ 被插入等待图中

系统中存在死锁 \iff 等待图包含环

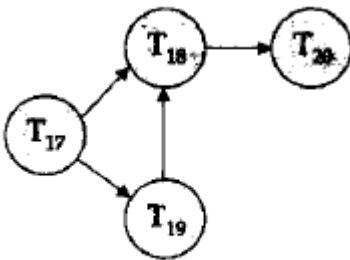


图 15-13 无环等待图

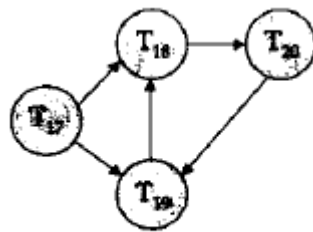


图 15-14 有一个环的等待图

15.2.2.2 从死锁中恢复

1. 选择牺牲者

决定回滚哪一个（哪一些）事务以打破死锁，使代价最小

2. 回滚

必须要决定事务要回滚多远（彻底回滚、部分回滚）

3. 饿死

保证一个事务被选为牺牲者的次数有限，避免总是被选为牺牲者

15.3 多粒度

允许系统定义多级粒度的机制，即通过各种大小的数据项定义数据粒度的层次结构

若事务 T_i 给 F_b 显式地加排他锁，则 T_i 也给所有属于该文件的记录隐式地加排他锁

意向锁：一个结点加上了意向锁，则意味着要在树的较低层进行显式加锁；在一个结点显式加锁之前，该结点的全部祖先结点均加上了意向锁

共享型意向锁(IS)：较低层进行显式封锁，但只能加共享锁

排他型意向锁(IX)：较低层进行显式封锁，可以加排他锁或共享锁

共享排他型意向锁(SIX)：以该结点为根的子树显式地加了共享锁，且树的更底层显式地加排他锁

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

多粒度封锁协议采用这些锁来保证可串行性

加锁按**自顶向下**的顺序（根到叶），释放锁按**自底向上**的顺序（叶到根）

该协议增加了并发性，减少了锁的开销

example:

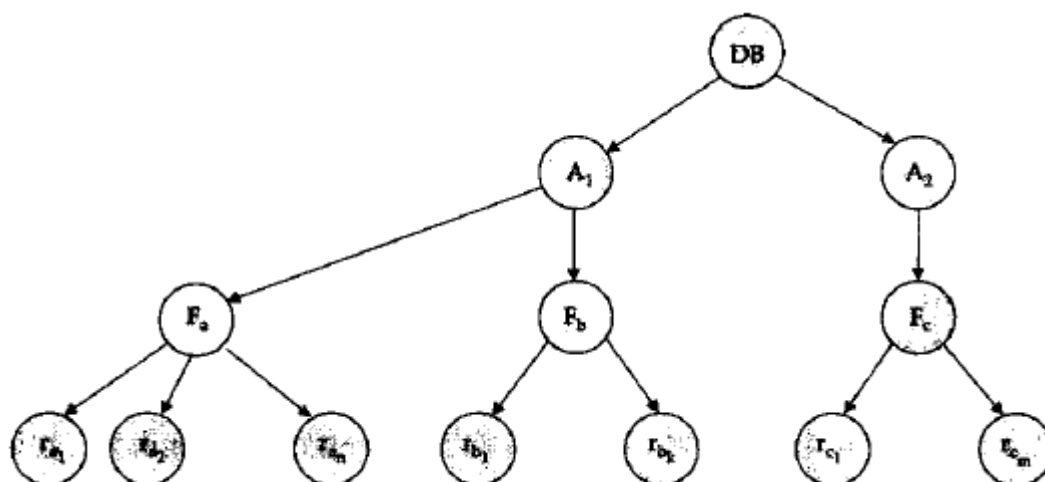


图 15-15 粒度层次图

- 假设事务 T_{21} 读文件 F_a 的记录 r_{a_i} 。那么, T_{21} 需给数据库、区域 A_1 , 以及 F_a 加 IS 锁(按此顺序), 最后给 r_{a_i} 加 S 锁。
- 假设事务 T_{22} 要修改文件 F_a 的记录 r_{a_i} 。那么, T_{22} 需给数据库、区域 A_1 , 以及文件 F_a (按此顺序)加 IX 锁, 最后给记录 r_{a_i} 加 X 锁。
- 假设事务 T_{23} 要读取文件 F_a 的所有记录。那么, T_{23} 需给数据库和区域 A_1 (按此顺序)加 IS 锁, 最后给 F_a 加 S 锁。
- 假设事务 T_{24} 要读取整个数据库。它在给数据库加 S 锁后就可读取。

15.4 基于时间戳的协议

15.4.1 时间戳

对于系统中的每个事务 T_i , 把时间戳 $TS(T_i)$ 与其联系起来

时间戳可以用**系统时钟**或**逻辑计数器**来表示

事务的时间决定了串行化顺序

W-timestamp(Q)执行**write(Q)**的所有事务的最大时间戳

R-timestamp(Q)执行**read(Q)**的所有事务的最大时间戳

每当有新的**read(Q)**和**write(Q)**指令执行时，这些时间戳就更新

15.4.2 时间戳排序协议

时间戳排序协议保证任何有冲突的**read**或**write**操作按时间戳顺序执行

1. 假设事务 T_i 发出 **read(Q)**。
 - a. 若 $TS(T_i) < W\text{-timestamp}(Q)$ ，则 T_i 需要读入的 Q 值已被覆盖。因此，**read** 操作被拒绝， T_i 回滚。
 - b. 若 $TS(T_i) \geq W\text{-timestamp}(Q)$ ，则执行 **read** 操作， $R\text{-timestamp}(Q)$ 被设置为 $R\text{-timestamp}(Q)$ 与 $TS(T_i)$ 两者的最大值。
2. 假设事务 T_i 发出 **write(Q)**。
 - a. 若 $TS(T_i) < R\text{-timestamp}(Q)$ ，则 T_i 产生的 Q 值是先前所需要的值，且系统已假定该值不会再产生。因此，**write** 操作被拒绝， T_i 回滚。
 - b. 若 $TS(T_i) < W\text{-timestamp}(Q)$ ，则 T_i 试图写入的 Q 值已过时。因此，**write** 操作被拒绝， T_i 回滚。
 - c. 其他情况，系统执行 **write** 操作，将 $W\text{-timestamp}(Q)$ 设置为 $TS(T_i)$ 。

该协议保证无死锁，因为不存在等待的事务

15.4.3 Thomas写规则

修改**write(Q)**的第二条规则为，若 $TS(T_i) < W\text{-timestamp}(Q)$ ，则忽略这个**write**操作

15.4.4 视图可串行化的

若 S 与 S' 是视图等价的，则应该满足以下三个条件：

1. 对于每个数据项 Q ，若 T_i 在调度 S 中读取了 Q 的初始值，那么在 S' 中 T_i 也必须读取 Q 的初始值
2. 对于每个数据项 Q ，若在调度 S 中 T_i 执行了 **read(Q)** 并且读取的值是由 T_j 执行 **write(Q)** 产生的；则在 S' 中， T_i 的 **read(Q)** 也必须是由 T_j 的同一个 **write(Q)** 操作产生的
3. 对于每个数据项 Q ，若在调度 S 中有事务执行了最后的 **write(Q)** 操作，则在调度 S' 中该事务也必须执行最后的 **write(Q)** 操作

条件1和2保证每个事务读取相同的值，进行相同的计算；条件3保证两个调度得到相同的最终状态

如果某个调度视图等价于一个串行调度，则我们说这个调度是视图可串行化的

15.5 基于有效性检查的协议

有效性检查协议要求每个事务 T_i 在其生命周期中按两个或三个阶段执行

1. **读阶段(read phase)**：在这一阶段中，系统执行事务 T_i 。各数据项值被读入并保存在事务 T_i 的局部变量中。所有 **write** 操作都是对局部临时变量进行的，并不对数据库进行真正的更新。
2. **有效性检查阶段(validation phase)**：对事务 T_i 进行有效性测试(下面将会介绍)。判定是否可以执行 **write** 操作而不违反可串行性。如果事务有效性测试失败，则系统终止这个事务。
3. **写阶段(write phase)**：若事务 T_i 已通过有效性检查(第2步)，则保存 T_i 任何写操作结果的临时局部变量值被复制到数据库中。只读事务忽略这个阶段。

1. **Start(T_i)**: 事务 T_i 开始执行的时间。
2. **Validation(T_i)**: 事务 T_i 完成读阶段并开始其有效性检查的时间。
3. **Finish(T_i)**: 事务 T_i 完成写阶段的时间。

事务 T_i 的**有效性测试**(validation test)要求任何满足 $TS(T_k) < TS(T_i)$ 的事务 T_k 必须满足下面两条件之一:

1. $Finish(T_k) < Start(T_i)$ 。因为 T_k 在 T_i 开始之前完成其执行, 所以可串行性次序得到了保证。
2. T_k 所写的数据项集与 T_i 所读数据项集不相交, 并且 T_k 的写阶段在 T_i 开始其有效性检查阶段之前完成 ($Start(T_i) < Finish(T_k) < Validation(T_i)$)。这个条件保证 T_k 与 T_i 的写不重叠。因为 T_k 的写不影响 T_i 的读, 又因为 T_i 不可能影响 T_k 的读, 从而保证了可串行性次序。

15.6 多版本机制

在**多版本并发控制**机制中, 每个write(Q)操作创建Q的一个新版本

当事务发出一个read(Q)操作时, 并发控制管理器选择Q的一个版本进行读取

15.7 快照隔离

快照隔离在事务开始执行时给它数据库的一份“快照”, 事务在该快照上操作, 和其它并发事务完全隔离

15.11 总结

见P397

第十六章 恢复系统

恢复机制负责将数据库恢复到故障发生前的一致状态

16.1 故障分类

- 事务故障: 逻辑错误、系统错误
- 系统崩溃 (硬件故障)
- 磁盘故障

16.2 存储器

16.2.1 稳定存储器的实现

内存和磁盘存储器间进行块传送有以下几种可能结果:

- 成功完成: 传送的信息安全地到达目的地
- 部分失败: 传送过程中发生故障, 目标块有不正确信息
- 完全失败: 传送过程中故障发生得足够早, 目标块仍完好无缺

16.2.2 数据访问

数据库分成称为**块**的定长存储单位, **块**是磁盘数据传送的单位, 可能包含多个数据项

位于磁盘上的块称为**物理块**, 临时位于主存的块称为**缓冲块**, 内存用于临时存放块的区域称为磁盘缓冲区

磁盘和主存间的块移动操作：

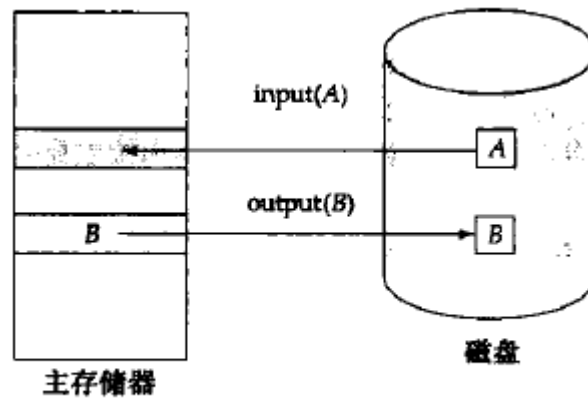


图 16-1 块存储操作

- $\text{input}(B)$ 传送物理块至主存
- $\text{output}(B)$ 传送缓冲块 B 至磁盘，并替换磁盘上相应的物理块

每个事务 T_i 有一个私有工作区，用于保存 T_i 所访问及更新的所有数据项的拷贝

传送数据的操作：

1. $\text{read}(X)$ 将数据项 X 的值赋予局部变量 x_i 。该操作执行如下：
 - a. 若 X 所在块 B_x 不在主存中，则发指令执行 $\text{input}(B_x)$ 。
 - b. 将缓冲块中 X 的值赋予 x_i 。
2. $\text{write}(X)$ 将局部变量 x_i 的值赋予缓冲块中的数据项 X 。该操作执行如下：
 - a. 若 X 所在块 B_x 不在主存中，则发指令执行 $\text{input}(B_x)$ 。
 - b. 将 x_i 的值赋予缓冲块 B_x 中的 X 。

执行 $\text{output}(B)$ 则称为数据库系统对缓冲块 B 进行强制输出

16.3 恢复与原子性

16.3.1 日志记录

日志记录数据库中的所有更新活动，它是日志记录的序列

更新日志记录 $\langle T_i, X_j, V_1, V_2 \rangle$ 描述一次数据库写操作：

- **事务标识 (transaction identifier)**，是执行 **write** 操作的事务的唯一标识。
- **数据项标识 (data-item identifier)**，是所写数据项的唯一标识。通常是数据项在磁盘上的位置，包括数据项所驻留的块的块标识和块内偏移量。
- **旧值 (old value)**，是数据项的写前值。
- **新值 (new value)**，是数据项的写后值。

一些日志记录类型：

- $\langle T_i \text{ start} \rangle$ ，事务 T_i 开始
- $\langle T_i \text{ commit} \rangle$ ，事务 T_i 提交
- $\langle T_i \text{ abort} \rangle$ ，事务 T_i 中止

16.3.2 数据库修改

延迟修改：一个事务直到它提交时都没有修改数据库

立即修改：数据库修改发生在事务仍然活跃时

16.3.3 并发控制和恢复

恢复算法要求如果一个数据项被一个事务修改了，那么在该事务提交或中止前不允许其它事务修改该数据项

16.3.5 使用日志来重做和撤销事务

$redo(T_i)$ 将事务 T_i 更新过的所有数据项的值都设置成新值

$undo(T_i)$ 将事务 T_i 更新过的所有数据项的值都恢复成旧值，同时写一个**read-only**日志($\langle T_i, X, V \rangle$)记录，当undo完成时写一个 $\langle T_i \text{ abort} \rangle$ 日志记录

如果日志包括 $\langle T_i \text{ start} \rangle$ 记录，但既不包括 $\langle T_i \text{ commit} \rangle$ ，也不包括 $\langle T_i \text{ abort} \rangle$ 记录，则需要对事务 T_i 进行撤销

如果日志包括 $\langle T_i \text{ start} \rangle$ 记录，以及 $\langle T_i \text{ commit} \rangle$ 或 $\langle T_i \text{ abort} \rangle$ 记录，则需要对事务 T_i 进行重做

16.3.6 检查点

检查点的执行过程如下：

1. 将当前位于主存的所有日志记录输出到稳定存储器
2. 将所有修改的缓冲块输出到磁盘
3. 将一个日志记录 $\langle \text{checkpoint L} \rangle$ 输出到稳定存储器，其中L是执行检查点时正活跃的事务的列表

在系统崩溃发生之后，系统检查日志以找到最后一条 $\langle \text{checkpoint L} \rangle$ 记录，只需要对L中的事务以及检查点后才开始执行的事务进行undo和redo操作

16.4 恢复算法

16.4.1 事务回滚

正常操作的事务回滚：

1. 从后往前扫描日志，对于所发现的 T_i 的每一个形如 $\langle T_i, X_j, V_1, V_2 \rangle$ 的日志记录：
 - 值 V_1 被写到数据项 X_j 中，并且
 - 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
2. 一旦发现了 $\langle T_i \text{ start} \rangle$ 日志记录，就停止从后往前的扫描，并往日志中写一个 $\langle T_i \text{ abort} \rangle$ 日志记录

16.4.2 系统崩溃后的恢复

崩溃发生后当数据库系统重启时，恢复动作分两阶段进行：

1. **重做阶段**，系统从最后一个检查点开始正向地扫描日志来重放所有事务的更新

扫描日志的过程中所采取的具体步骤如下：

- 将要回滚的事务的列表 undo-list 初始设定为 $\langle \text{checkpoint } L \rangle$ 日志记录中的 L 列表。
- 一旦遇到形为 $\langle T_i, X_j, V_1, V_2 \rangle$ 的正常日志记录或形为 $\langle T_i, X_j, V_2 \rangle$ 的 redo-only 日志记录，就重做这个操作；也就是说，将值 V_2 写给数据项 X_j 。
- 一旦发现形为 $\langle T_i, \text{start} \rangle$ 的日志记录，就把 T_i 加到 undo-list 中。
- 一旦发现形为 $\langle T_i, \text{abort} \rangle$ 或 $\langle T_i, \text{commit} \rangle$ 的日志记录，就把 T_i 从 undo-list 中去掉。

2. 撤销阶段，系统回滚 undo-list 中的所有事务，从尾端方向扫描日志来执行回滚

- 一旦发现属于 undo-list 中的事务的日志记录，就执行 undo 操作，就像在一个失败事务的回滚过程中发现了该日志记录一样。
- 当系统发现 undo-list 中事务 T_i 的 $\langle T_i, \text{start} \rangle$ 日志记录，它就往日志中写一个 $\langle T_i, \text{abort} \rangle$ 日志记录，并且把 T_i 从 undo-list 中去掉。
- 一旦 undo-list 变为空表，即系统已经找到了开始时位于 undo-list 中的所有事务的 $\langle T_i, \text{start} \rangle$ 日志记录，则撤销阶段结束。

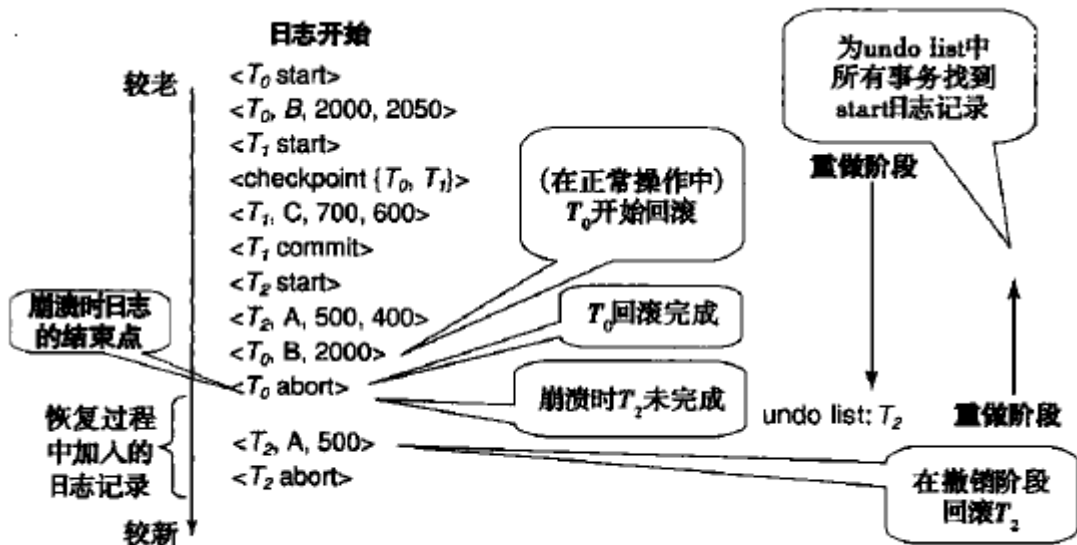


图 16-5 记录在日志中的动作和恢复中的动作的例子

当从崩溃中恢复时，在重做阶段，系统对最后一个检查点记录之后的所有操作执行 redo。在这个阶段中，undo-list 初始时包含 T_0 和 T_1 ；当 T_1 的 commit 日志记录被发现时， T_1 先被从表中去掉，而当 T_2 的 start 日志记录被发现时， T_2 被加到表中。当事务 T_0 的 abort 日志记录被发现时， T_0 被从 undo-list 中去掉，只剩 T_2 在 undo-list 中。撤销阶段从尾端开始反向扫描日志，当发现 T_2 更新 A 的日志记录时，将 A 恢复成旧值，并往日志中写一个 redo-only 日志记录。当发现 T_2 开始的日志记录时，就为 T_2 添加一条 abort 记录。由于 undo-list 不再包含任何事务了，因此撤销阶段终止，恢复完成。

16.5 缓冲区管理

16.5.1 日志记录缓冲

将一个块输出到稳定存储器的开销非常高，因此最好是一次输出多个日志记录

先写日志规则：由于系统崩溃时这种日志记录会丢失，因此我们必须增加要求来保证事务的原子性

- 在日志记录 `<Ti commit>` 输出到稳定存储器后，事务 T_i 进入提交状态。
- 在日志记录 `<Ti commit>` 输出到稳定存储器前，与事务 T_i 有关的所有日志记录必须已经输出到稳定存储器。
- 在主存中的数据块输出到数据库(非易失性存储器)前，所有与该数据块中数据有关的日志记录必须已经输出到稳定存储器。

16.5.2 数据库缓冲

强制策略：事务在提交时强制地将修改过的所有的块都输出到磁盘

非强制策略：即使一个事务修改了某些还没有写回到磁盘的块，也允许它提交

窃取策略：允许系统将修改过的块写到磁盘，即使做这些修改的事务还没有全部提交

非窃取策略：一个仍然活跃的事务修改过的块都不应该写出到磁盘

16.10 总结

见P424

[回到顶部](#)