

第一章 引言

本章概述

数据库用于有效管理数据的概念和技术，本章将简要介绍数据库系统的基本原理，具体将在每一章内单独讨论，引言部分主要介绍几个基本概念。

1.1.基本概念

1.1.1.数据（Data）

- 数据(Data)是数据库中存储的基本对象
- 数据的定义：描述事物的符号记录
- 数据的种类：文字、图形、图像、声音
- 数据的特点：数据与其语义是不可分的

1.1.2.数据库(Database，简称DB)

数据库是长期储存在计算机内、有组织的、可共享的大量数据集合

数据库的特征：数据按一定的数据模型组织、描述和储存；可为各种用户共享；冗余度较小；数据独立性较高；易扩展

1.1.3.数据库管理系统(Database Management System，简称DBMS)

DBMS由一个互相关联的数据的集合和一组用以访问这些数据 的程序组成，是位于用户与操作系统之间的一层数据管理软件

DBMS的用途：科学地组织和存储数据、高效地获取和维护数据

DBMS的主要功能：数据定义功能、数据操纵功能、数据库的运行管理、数据库的建立和维护功能

1.1.4.数据库系统(Database System，简称DBS)

数据库系统是指在计算机系统中引入数据库后的系统。在不引起混淆的情况下常常把数据库系统简称为数据库

数据库系统的构成：数据库、数据库管理系统、应用系统（及其开发工具）、数据库管理员（和用户）

1.2.数据视图

1.2.1.数据抽象

由于许多数据库系统的用户并未受过计算机专业训练，系统开发人员通过如下几个层次上的抽象来对用户屏蔽复杂性，以简化用户与系统的交互：

- 物理层:最低层次的抽象，描述数据存储
- 逻辑层:描述存储在数据库中的数据，以及数据之间的关系
- 视图层:最高层次的抽象，只描述整个数据库的某部分数据

1.2.2.实例与模式

- 实例-特定时刻存储在数据库中的信息的集合
- 模式-数据库的总体设计
- 数据独立性 -物理数据独立性/逻辑数据独立性

1.2.3.数据模型

数据库结构的基础是数据模型。数据模型是一个描述数据、数据联系、数据语义以及一致性约束的概念工具的集合。

数据模型分四类：关系模型、实体-联系模型、基于对象的数据模型、半结构话数据模型

第二章 关系模型介绍

本章概述

关系数据库系统：是支持关系模型的数据库系统

关系模型的组成：关系数据结构、关系操作集合、关系完整性约束

2.1.关系数据库的结构

2.1.1.几个概念

- 关系——指代“表” 关系是无序的，元组的顺序是无关紧要的。

笛卡尔积D1×D2×...×Dn的子集叫做在域D1，D2，..., Dn上的关系，用R(D1，D2，..., Dn)表示

R是关系的名字，n是关系的度或目

关系是笛卡尔积中有意义的子集

关系也可以表示为二维表

- 元组——指代行（是一组值的序列）
- 属性——指代列 属性的值（通常）要求为原子的，也就是说，不可再分
- 关系实例——表示一个关系的特定实例，也就是所包含的一组特定的行
- 域——每个属性可能的取值范围（集合）叫属性的域
- 空值——表示值为止或不存在

2.2.数据库模式

2.2.1.数据库模式和实例

数据库模式是数据库的逻辑设计，数据库实例是给定时刻数据库中数据的一个快照

- 关系模式——对应于程序设计语言中类型定义的概念
- 关系实例——对应于程序设计语言中变量的值的概念

A1, A2, ..., An 是属性

R = (A1, A2, ..., An) 是一个关系模式

2.3.码

2.3.1.码的作用

我们必须有一种能够区分给定关系中的不同元组的方法。我们一般用元组中的属性来表明，即一个元组的属性值必须是能够唯一区分元组的，一个关系中没有两个元组在所有属性上的取值都相同。

2.3.2.超码

超码是一个或者多个属性的集合，这些属性的组合可以使我们在一个关系中唯一地标识一个元组

2.3.3.候选码

最小的超码称为候选码，即超码的任意真子集都不能成为超码

2.3.4.主码

从一个关系的多个候选码中选定一个作为主码

2.3.5外码

一个关系模式r1可能在它的属性中包含另一个关系模式r2的主码，这个属性在上称作在r1上参照r2的外码（ r1和r2可以是同一个关系）。

关系r1称作外码依赖的参照关系

关系r2称作外码的被参照关系

2.4.模式图

一个含有主码和外码依赖的数据库模式可以用模式图来表示。

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。

2.5.关系查询语言

- 查询语言使用户用来从数据库中请求获取信息的语言
- 过程化语言 vs 非过程化语言/声明式语言

2.6.关系运算

2.6.1.基本运算

一元运算：选择、投影、更名 多元运算：笛卡尔积、并、集合差

2.6.2.其他运算

集合交、θ连接、自然连接、除、赋值

2.6.3.自然连接

设r和s是关系模式R和S的实例，R和S关系实例的“自然连接”是关系模式R □ S的实例，遵守以下规则：

- 对于每一对元组tr和ts，其中tr 来自r， ts来自s
- 如果 tr 和 ts 在属性组R∩S上每个属性值都一样,添加一个元组 t 到结果集, 其中 t 有tr 在r上相同的值 t 有ts 在s上相同的值

2.6.4.笛卡尔积

笛卡尔积运算从两个关系中合并元组，但不同于连接运算的是，其结果包含来自两个关系元组的所有对，无论它们的属性是否匹配

第三章 SQL

本章概述

本章介绍SQL的基本结构和概念，它除了数据库查询，还可以定义数据结构修改数据库中的数据以及说明安全性约束条件等。本章主要给出对SQL的基本DML和DDL特征的概述。

3.1.SQL查询语言概览

DDL：数据定义语言，Data-Deifinition Language，提供定义关系模式，删除关系以及修改关系模式的命令。

DML：数据操纵语言，Data-Manipulation Language，提供从数据库中查询信息，以及在数据库中插入元组、删除元组、修改元组的能力。

3.2.SQL数据定义

SQL的数据定义语言（DDL）能够定义每个关系的信息，包括：

- 每个关系的模式
- 每个属性的值域
- 完整性约束
- 每个关系的索引集合
- 每个关系的安全性和权限信息
- 磁盘上每个关系的物理存储结构

3.2.1.基本类型

SQL支持的多种固有类型：

- char(n)：固定长度的字符串，用户指定长度 n。也可使用全称 character。
- varchar(n)：可变长度的字符串，用户指定最大长度，等价于全称 character varying。（作者注：在实际使用中，也可以写成 varchar2）
- int：整数类型，等价于全称 integer。（-2³¹~2³¹-1）

- smallint: 小整数类型(-2¹⁵~-2¹⁵-1)
- numeric(p,d): 定点数, 精度由用户制定。这个数有p位数字(加上一个符号位), 其中d位数字在小数点后面。
- real,double precision: 浮点数与双精度浮点数, 精度与机器有关。
- float(n): 精度至少为 n 位的浮点数。
- date: 日期(年、月、日)。输入日期类型数据的格式是: date 'xxxx-xx-xx' 或者to_date('xxxxxxx','yyyymmdd')其中 x 为数字, 其它部分直接引用即可。
- time: 时间(小时、分、秒) (oracle 中没有)

3.2.2.基本模式定义

创建语句

```
```sql create table r
```

(A1 D1,

A2 D2,

...,

An Dn,

<完整性约束1>,

...,

<完整性约束k>)

...

- r是关系名,每个 Ai是关系模式 r 的一个属性名,Di是属性 Ai 的域的类型

完整性约束

not null

primary key (A1, ..., An )

foreign key (Am, ..., An ) references r

常用完整性约束:

- 主码约束: PRIMARY KEY, 必须非空且唯一
- 唯一性约束: UNIQUE
- 非空值约束: NOT NULL
- 参照完整性约束: FOREIGN KEY, 例如, course表的定义中声明了“foreign key(deptname) reference department”, 此外码保证course中的系名一定存在于department关系的主码属性 (deptname) 中

也可以把主码的声明和属性的声明放在一起

删除和更改表结构

- drop table 表名;

删除表和它的内容

- delete from 表名;

删除表中的内容, 但是保留表

- alter table r add A D;

属性A是关系 r 将要增加的属性, D 是A的域。对于关系 r 中的所有元组, 在新增加的属性上的取值都为 null。

- alter table r drop A;

从关系r中去掉属性A

- alter table r modify A D;

修改列的属性

- alter table r drop constraint 约束名

删除约束

3.3.SQL查询的基本结构

3.3.1.单关系查询

- select \*/属性名 from 表名

\*是指全部属性。“select all 属性名 from 表名”, 是指明不去除重复。“select distinct 属性名 from 表名”, 会在结果中删除重复元组。

select 子句可含有包含+、-、\*、/运算符的算术表达式, 运算对象可以是常量或元组的属性

- select 属性名 from 表名 where.....

where 子句指定查询结果必须满足的条件

3.3.2.多关系查询

通常查询需要从多个关系中获取信息, 典型SQL查询具有如下形式:

```
``` select A1, A2, .....An //代表属性
```

from r1, r2,, rm//代表关系

where P; //P是一个谓词 ```

属性名位于多关系时, 加上前缀。写为: 表名.属性名

from后面的关系连接, 即笛卡尔积

3.3.3.自然连接

- select *from 表1 natural join 表2;

自然连接只考虑两个关系模式中都出现的属性上取值相同的元组对，并且相同属性的列只保留一个副本。

举个例子： 列出教师的名字和他们所讲授课程的课程标识

```
select name, course_id from instructor, teaches where instructor.ID = teaches.ID;
```

用自然连接运算更简洁地写作：

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

- r1 join r2 using(A1,A2)

与r1和r2的自然连接类似，但要求t1.A1=t2.A1且t1.A2=t2.A2成立，则来自r1的元组t1和来自r2的元组t2就能匹配，即使r1，r2都有A3的属性，也不需要满足t1.A3=t2.A3

3.4.附加的基本运算

3.4.1.更名运算

- old name as new name as子句既可以出现在select子句中，也可出现在from子句中，用于重命名关系。as可以省略。new name常被称作表别名（table alias）,或者相关变量（correlation variable）,或者元组变量（tuple variable）。

3.4.2.字符串运算

SQL中通过字符串匹配运算符来支持在字符串上的比较，使用"like"操作符来实现模式匹配，使用两个特殊字符（通配符）描述模式：

- 百分号 (%) % 字符匹配任何子串
- 下划线(_) _ 字符匹配任何字符
- like具有单向性，例如'济南市%' like '济南市山大路'，结果false

转义字符串/可以去掉特殊字符的特定含义，使其被作为普通字符看待。例：匹配字符串"100 %", like '100 %'

3.4.3.select子句中的属性说明

星号""可以用在select子句中表示"所有的属性"。

3.4.4.排列元组的显示次序

- order by

默认升序，desc表示降序，asc表示升序。只能是SQL的最后一个语句。null表示最大。

3.4.5.where子句谓词

- between比较运算符来说明一个值是≤某个值且≥某个值，同理还有not between。
- 用记号（v1,v2,.....,vn）来表示一个分量值分别为v1,v2,.....,vn的n维元组
- 元组比较用字典顺序进行比较

3.5.集合运算

- union，intersect，except 对应 ∪，∩，-

每个运算自动去重

- 如果要保留重复，则使用相应的多重集版本 union all（m + n），intersect all（min(m,n)）和 except all（max(0, m - n)）

（上面括号表示：如果一个元组在 r 中出现 m 次，在 s 中出现 n次,则运算后该元组出现几次）

3.6.空值

- 包含 null 的任何算术表达式的计算结果是null
- 谓词 is null 可以用来检测空值（不可以写为= null）
- 带有 null 的任何比较运算返回 unknown

空值和三值逻辑

- OR: (unknown or true) = true,

(unknown or false) = unknown

(unknown or unknown) = unknown

- AND: (true and unknown) = unknown,

(false and unknown) = false,

(unknown and unknown) = unknown

- NOT: (not unknown) = unknown

3.7.聚集函数

固有聚集函数：

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

以上所有都可以在属性名前加 distinct 来去重，但不允许使用 count(distinct*)

3.7.1.基本聚集

count(属性名)和 count(*)的区别：

count(*)返回满足条件的元组的总个数（即使一个元组的所有属性取值均为 null也会被计算在内），count(属性名)返回该属性中取值不为 null 的总个数；

3.7.2.分组聚集

group by 属性名 [having 条件表达式]

意义：

- group by:将表中的元组按指定列上值相等的原则分组，然后在每一分组上使用聚集函数，得到单一值
- having:对分组进行选择，只将聚集函数作用到满足条件的分组上，从关系代数角度来看，having 子句的谓词在形成分组后才起作用
- 注意事项：分组聚集函数中的属性需要包括 select 中的所有非聚集函数属性

示例：

列出每一年龄组中男学生（超过 50 人）的人数：

```
``sql select age, count(sno)
```

from S

where sex='M'

group by age

having count(*) > 50 ````

where 和 having 的异同

- 相同之处：二者均是选择运算
- 不同之处：二者的作用对象不同，where 的作用对象是元组，having 的作用对象是分组
- 具体使用方式：having 中的条件一般用于对一些聚集函数的比较，如 count() 等等。除此而外，一般的条件应该写在 where 子句中

综合上述，语句格式：

```
``sql select [distinct]属性名...
```

from 表、视图、临时关系...

[where 条件表达式...]

[group by 属性名[having 条件表达式]...]

[order by 属性名[desc/asc]] ````

3.8.嵌套子查询

一个子查询是一个嵌套在其他的查询中的select-from-where 表达式

3.8.1.集合成员资格

子查询通常用于对集合成员的资格、集合的比较、集合的基数进行检查

- 集合成员资格 in
- 集合之间的比较 θ
- 测试集合是否为空 exists
- 测试集合是否存在重复元组 unique

注：子查询不能使用 order by 子句，有些嵌套查询可以用连接运算替代

3.8.2.集合的比较

where 属性名/聚集函数 比较运算符 some/all(子查询...)

some 表示某一个;all 表示其中所有

- some 子句的定义

- all 子句的定义

- some与ALL与聚集函数的对应关系

3.8.3.空关系测试

含义：exists 结构测试子查询结果是否有元组，子查询非空的时候，返回 true

相关子查询：in后的子查询与外层查询无关，每个子查询执行一次，而exists后的子查询与外层查询有关，需要执行多次，称之为相关子查询

关于子查询的解释：首先取外层查询中表的第一个元组，根据它与内层查询相关的属性值处理内层查询，若 WHERE 子句返回值为真，则取此元组放入结果集合中。然后再取外层表的下一个元组。重复这一过程，直至外层表全部检查完为止。

SQL中“全部”概念处理

- 存在 not exists(not exists)
- 超集superset: not exists (X except Y)
- + not in(not in)

3.8.4.重复元组存在性测试

unique 结构测试一个子查询的结果中是否有重复的元组 （在空集中其值为“true”）

示例：

- 找出所有只教授一门课程的老师姓名

```sql select tname

```
from t

where unique

(select tno

from tc

where tc.tno = t.tno)
```

...

- 找出至少选修了两门课程的学生姓名

```sql select sname

```
from s

where not unique

(select sno

from sc

where sc.sno = s.sno)
```

...

3.8.5.from子句中的子查询

SQL 允许在from子句中使用子查询表达式

另一个表达上述查询的方式：lateral 子句。Lateral 子句允许 from 子句中后面的部分 （在关键词 lateral 后面的）访问前面部分的相关变量

3.8.6.with子句

with 子句提供了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。它表中属性的数据通过查询返回的结果进行赋值

- 语法：

with 关系名 1(属性名 1...) as(子查询...),

关系名 2(属性名 2...) as(子查询...)...

之后就可以在下面的查询中使用 with 定义的关系了

- 好处：with 子句使得查询在逻辑上更加清晰，此时它比嵌套语句要好理解。
- 示例：

查询最高成绩的学生学号

```sql with max\_score (mscore) as

select max (score)

from sc

Select sno

from sc,max\_score

where sc.score=max\_score.mscore;

...

注：max\_score 这个名字和其 mscore 属性是我们自己定义的

3.8.7.标量子查询

标量子查询只返回包含单个属性的单个元组

3.9.数据库的修改

3.9.1.删除

- delete from 表名 [where 条件表达式]

只能删除整个元组，而不能只删除某些属性上的值

3.9.2.插入

- insert into 表名 values(值 1， 值 2...)

上述的是按照元组顺序进行插入对应数值的

- 如果忘记了属性的顺序可以：

insert into 表名 (属性名 1， 属性名 2...)

values(值 1， 值 2...)

- 带有子句的插入示例：

将平均成绩大于 90 的学生加入到 EXCELLENT 中

```sql insert into EXCELLENT( sno, score)

select sno, avg(score)

```
from sc
```

group by(sno)

having avg(score)>90 ```

注：DBMS 在执行插入语句时会检查所插元组是否破坏表上已定义的完整性规则

3.9.3.更新

- 语句

```sql update 表名

set 属性名 = 表达式 | 子查询

[属性名 = 表达式 | 子查询]...

[where 限制条件]

```

- 对于多重 update，可能会因为顺序而导致不同的结果，此时使用 case

语句：

```sql update 表名

set 属性名 = case

when 条件语句 then 各种运算

when 条件语句 then 各种运算...

else 各种运算

```
end
```

```

- 示例：工资超过 3500 的缴纳 10%所得税，其余的缴纳 5%税，计算扣除所得税后的工资

```
sql update T set sal = case when sal > 3500 then sal*0.9 else sal*0.95 end
```

注：DBMS 在执行修改语句时会检查修改操作是否破坏表上已定义的完整性规则

第四章 中级SQL

本章概述

本章考虑具有更加复杂形势的SAL查询、视图定义、事务、完整性约束、关于SQL数据定义的更详细介绍以及授权。

4.1.连接表达式

连接操作作用于两个关系并返回一个关系作为结果。

基本分类

- 连接成分

包括两个输入关系、连接条件、连接类型

- 连接条件

决定两个关系中哪些元组相互匹配，以及连接结果中出现哪些属性

- 连接类型

决定如何处理与连接条件不匹配的元组

4.1.1.连接条件

on条件允许在参与有连接的关系的关系上设置通用谓词，该谓词的写法与where子句谓词类似，但在外连接中，on 条件的表现与where条件是不同的。

on优点：

- 对被称作外连接的这类连接来说，on条件的表现与where条件是不同的。
- 如果在on子句中指定连接条件，并在where子句中出现其余的条件，这样的SQL查询通常更容易让人读懂。

4.1.2.外连接

外连接（outer join）运算与我们已经学过的连接运算类似，但通过在结果中创建包含空值元组的方式，保留了那些在连接中丢失的元组。

- 内连接：舍弃不匹配的元组
- 左外连接：内连接+左边失配的元组（缺少的右边关系属性用 null）
- 右外连接：内连接+右边失配的元组（缺少的左边关系属性用 null）
- 全外连接：内连接 + 左边失配的元组（缺少的右边关系属性用 null） + 右边失配的元组（缺少的左边关系属性用 null）

示例

on和where的不同

外连接只为那些对应内连接没有贡献的元组补上空值并加入结果。on条件是外连接声明的一部分（只有不需要进行补充空值时才考虑 on 条件）。但 where 子句是在外连接完成之后才进行的，这就会导致部分元组因为使用了空值填充而不满足 where 条件因而被排除了。

4.1.3.连接类型和条件

任意的连接形式可以和任意的连接条件进行组合。

4.2.视图

视图提供一个对某些用户从视图中隐藏某些数据的机制。

任何不是逻辑模型的一部分，但作为虚关系对用户可见的关系称为视图。

视图特点

- 虚表，是从一个或几个基本表（或视图）导出的关系
- 只存放视图的定义，不会出现数据冗余
- 基表中的数据发生变化，从视图中查询出的数据也随之改变
- 查询时,视图名可以出现在任何关系名可以出现的地方

视图 vs 派生查询(with)

- 视图存储在DB数据字典中，是数据库模式的一部分
- with定义的派生关系，仅在所属SQL有效，不属于DB模式

视图(view) vs 表(table)

- 视图和表都是关系，都可在SQL中直接应用
- DB中存储表的模式定义和数据
- DB中只存储视图的定义，不存储视图的数据
- 视图数据在使用视图时临时计算
- 物化视图是提高计算的一种手段，结果等价于临时计算

视图的作用

- 对外模式的支持
- 安全性、方便性
- 适应数据库共享的需要
- 对重构数据库在一定程度上提供了一定程度的逻辑独立性

4.2.1.视图定义

定义：create view 视图名 [(属性名...)] as （查询表达式）

删除视图：drop view view_name

4.2.2.SQL查询中使用视图

一旦定义了一个视图，我们就可以用视图名指代该视图生成的虚关系。

create view 视图名（视图中属性名）**as**

select 属性名

from 关系名

4.2.3.物化视图

创建一个物理表，此表包含定义视图的查询结果的所有元组。

如果查询中使用的关系发生了更新，则物化视图中的结果就会过期。

每当视图的底层关系进行更新时要更新视图，以此来维护视图。

物化视图目的：

使用物化视图的目的是为了提高查询性能，是以空间换时间的一种有效手段，更少的物理读/写，更少的cpu时间，更快的响应速度；规模较大的报表适合使用物化视图来提高查询性能。

4.2.4.视图更新

一般来说，如果定义视图的查询对下列条件都满足，则称SQL视图是可更新的。

- from 子句中只有一个数据库关系
- select 子句中只包含关系的属性名，不包含任何表达式、聚集或distinct
- 任何没有出现在 select 子句中的属性可以取空值；它们也不构成主码的一部分（在这种情况下，插入元组时没有声明的属性值用 null 代替）
- 查询中不含有 group by 或 having 子句

视图with check option

如果向视图中插入一条不满足视图的where子句条件的元组，数据库系统将拒绝该插入操作。

4.3.事务

一个事务由查询和更新的语句的序列组成。

一个 SQL 语句开始执行隐含 一个事务的开始。

以下列语句之一表示结束一个事务：

commit [work]: 提交当前事务，即将该事务所做的更新在数据库中永久保存。

rollback [work]: 回滚当前事务，即撤销该事务中所有 SQL 对数据库的更新，数据库恢复到执行该事务的第一条语句之前的状态。

注：DDL 和 DCL 与事务无关

4.4.完整性约束

完整性约束保证授权用户对数据库进行修改时不会破坏数据的一致性。防止对数据的意外破坏。

完整性约束通常被看成是数据库模式设计过程的一部分，它作为用于创建关系的**create table**命令的一部分被声明。

4.4.1.单个关系上的约束

- not null
- primary key

- unique
- check (P)，P是一个谓词

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声明指明下列属性A1, A2, ... Am 形成了一个候选码；即在关系中没有两个元组能在所有列出的属性上取值相同。候选码属性可以为null。空值不等于其他的任何值。

4.4.4.check子句

check (P) ， P是一个谓词

关系上的每一个元组，都必须满足P。

如果S中删除元组，不会触发CHECK子句，只有对SC表的更新才会触发。

主码约束：主码值不允许空，也不允许出现重复

4.4.5.参照完整性

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。这种情况称为参照完整性。

A是一个属性的集合，R和S是两个包含属性A关系，并且A是S的主码 如果对于每个在R中出现的A在S中也出现，则A被称为R的 外码。不同于外码约束，参照完整性约束通常不要求A是S的主码；其结果是，S中可能有不止一个元组在属性A上取值相同。

4.4.6.事务中对完整性约束的违反

事务可能包括几个步骤，在某一步之后完整性约束也许会暂时被违反，但是后面某一步会消除这个违反。

如何解决？

- 在插入sc 关系的一个元组之前，先插入表示他的父亲和母亲的元组
- 或者，先将新插入元组的学生和课程属性设为空值，将sc关系的所有元组都插入后，再更新（如果学生和课程属性被声明为not null的话这种方法不可行）
- 或者推迟完整性检查。即需要的时候可以延迟检查。

4.5.SQL的数据类型与模式

关于如何在SQL中创建基本的用户定义类型。

4.5.1.SQL中的日期和时间类型

- **date:** 日期，包括年（四位）、月和日

示例：

```
date '2014-3-10'
```

- **time:** 时间，包括小时、分和秒。time(p)可以表示秒的小数点后的数字位数（默认值为0）

示例：

```
time '09:00:30'
```

- **timestamp:** date和time 的组合

示例：

```
timestamp '2014-3-10 09:00:30'
```

- **interval:** 时间段

示例：

```
interval '1' day
```

SQL允许在上面列出的所有类型上进行比较运算，也允许在各种数字类型上进行算术运算和比较运算。两个 date/time/timestamp 类型值相减产生一个 interval 类型值。也可以在 date/time/timestamp 类型的值上加减interval 类型的值。例如x、y都是data类型，x-y的值是日期x到日期y间隔的天数。

4.5.2.默认值

给某一属性指定默认值后，当一元组插入关系时没有给定该属性的值，就会被设置为默认值。

示例：

```
create table s (sno char (5), sname varchar (20) not null, dno char (20), sex char(1) default '1', primary key (sno))
```

当没有给出性别的值时，默认为'1'

4.5.3.创建索引

格式：

```
create index studentsno_index on s(sno)
```

作用：索引是一种数据结构，它允许数据库高效地找到关系中那些在索引属性上取给定值的元组，而不用扫描关系中的所有元组。用于加快查询在索引属性上取给定值的元组的速度。

更多关于索引的内容在第十章

4.5.4.大对象类型

大对象（照片，视频，CAD文件等）以large object 类型存储：

blob：二进制数据的大对象数据类型--对象是没有被解释的二进制数据的大集合（对二进制数据的解释由数据库系统以外的应用程序完成）

clob：字符数据的大对象数据类型--对象是字符数据的大集合

当查询结果是一个大对象时，返回的是指向这个大对象的指针，而不是大对象本身

4.5.5.用户定义的类型

格式：

```
create type 类型名 as 数据类型 [final根据系统自身决定]
```

示例：

```
create type person-name as char (20) [final]
```

SQL提供了**drop type**和**alter type**子句里删除或修改以前创建过的类型

域定义：可以在基本类型上施加完整性约束

格式：

```
create domain 域名 as 数据类型
```

示例：

```
create domain person-name as char (20)
```

类型定义与域定义的区别：+域上可以声明约束，例如not null，也可以为域类型变量定义默认值，然而在用户定义类型上不能声明约束或默认值。+域并不是强类型的。因此一个域类型的值可以被赋给另一个域类型，只要它们基本类型是相容的。

当把check子句应用到域上时，允许模式设计者指定一个谓词，被声明为来自该域的任何变量都必须满足这个谓词。

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

权限：每种数据的授权都称为一个权限

4.6.1.权限的授予与收回

定义：允许用户把已获得的权限转授给其他用户，也可以把已授给其他用户的权限再回收上来

权限类型：select、insert、update、delete和all privileges（所有权限）

授予权限：

```
grant 表级权限
on {表名 | 视图名}
to {用户 [, 用户]... | public}
```

表级权限包括：

- select, update, insert, delete, index, alter, drop, resource以及它们的总和all，其中对select, update可指定列名
- with grant option表示获得权限的用户可以把权限再授予其它用户

示例：

```
grant select,insert on S to Liming
with grant option

grant all on S to public

grant UPDATE(sno),SELECT ON TABLE S to U4

grant ALL PRIVILIGES to public
```

示例：把对表SC的INSERT权限授予U5用户，并允许他再将此权限授予其他用户

```
grant insert on table sc to u5 with grant option
```

收回权限

```
revoke <权限列表>
on <关系名或视图名>
from <用户/角色列表>
```

4.6.2.角色

创建角色

```
create role instructor
```

角色可以被授以权限：grant select on takes to instructor;

角色可以授以用户，也可以被授以**其他角色**

```
create role teaching_assistant

grant teaching_assistant to instructor;
```

Instructor 继承teaching_assistant的所有权限

角色链

```
create role dean;
grant instructor to dean;
grant dean to Satoshi;
```

4.6.3.视图的授权

用户在使用视图的时候，系统会根据用户权限判定用户请求是否合法。

4.6.4模式的授权

SQL标准为数据库模式指定了一种基本的授权机制：只有模式的拥有者才能够执行对模式的任何修改，诸如创建或删除关系，增加或删除关系的属性，以及增加或删除索引。

4.6.5权限的转移

转移权限：

grant 权限 on 表名 to 用户名 with grant option

with grant option表示获得权限的用户可以把权限再授予其它用户

权限图

结点是用户，根结点是DBA，有向边U→U_j，表示用户U把某权限授给用户U_j一个用户拥有权限的充分必要条件是权限图中有一条从根结点到该用户结点的路径



4.6.6权限的收回

格式：

```
revoke 表级权限 on {表名|视图名} from {用户 [, 用户]... | public}[restrict]
```

关键字 restrict的意思是防止级联收回，默认级联收回

级联收回： 从一个 用户/角色 哪里收回权限可能导致其他 用户/角色 也失去该权限。

示例：

```
revoke insert on S from Liming
```

注：下面语句仅仅收回grant option而不是收回select权限

revoke grant option for select on department from Amit （有的数据库不支持上述语法）

特别： 支持多库的数据库系统中授权对象可以是数据库

```
grant 数据库级权限 to {用户 [, 用户]... | public}
```

数据库级权限包括：

- connect: 允许用户在database语句中指定数据库
- resource: connect权限+建表、删除表及索引权利
- dba: resource权限
- 授予或撤消其他用户的connect、resource、dba权限，不允许dba撤消自己的dba权限

第六章 形式化关系查询语言

本章概述

介绍SQL所基于的形式化模型，包括关系代数、元组关系演算和域关系演算。

注：本章多次使用下列表 D(dno,dname,dean)院系

S(sno,sname,sex,age,dno)学生

C(cno,cname,credit)课程

T(tno,tname,dno,sal)老师

SC(sno,cno,score)学生选课信息

TC(tno,cno)教师教课信息

6.1 关系代数

6.1.1 基本运算 $\sigma \Pi \cup - \times \rho$

仅使用基本运算，能表达全部关系代数查询

对空值 null 的处理：

- σ 保留确定为真的元组
- $\Pi \cup \cap$ - 多个 null 只保留一个

选择(select) σ ：

$\sigma_F(R)$ 从行的角度，选出 R 中符合条件 F 的那些元组

*F 的形式：*由逻辑运算符连接关系表达式而成

示例：找年龄不小于 20 的男学生 $\sigma_{age \geq 20 \wedge sex = 'm'}(S)$

投影(project) Π ： $\Pi_A(R)$ 从列的角度，选出 R 里面的 A_i 属性，在结果中去掉相同的行

示例：查询所有学生的姓名和年龄

$\Pi_{sname, age}(S)$

集合并(union) \cup ： $R \cup S$ 相当于罗列出 R 和 S 中所有的元组，其中 RS “相容”

相容：R、S 属性的数目相同并且对应第 i 个属性的域相同

示例：查询选修了 c1 号或 c2 号课程的学生号

方案一： \cup

方案二： \cup

集合差(differenct) :- **R-S,R 和 S 相容**并且结果集里面为 R 中有而 S 中没有的

注意：在“没有做***”的题目中，通常用总的与的做差而不是 直接用 \subsetneq (不等于)

例如：查询未选修 c1 号课程的学生号

方案 1: $\neg \text{sno}(S) - \neg \text{sno}(\sigma_{\text{cno} = 'c1'}(SC)) \vee$

方案 2: $\neg \text{sno}(\sigma_{\text{cno} \subsetneq 'c1'}(SC)) \times$

方案二中，首先，没有将未选修的同学包含在内，其次也是最大的问题在于，SC 中选了别的课并且选了 c1 课的同学也会被选中

笛卡尔积(Cartesian-product) \times :

$R \times S$ ，与离散数学中的笛卡尔积相同，就是 R 的每一个元组与 **S** 的每一个元组合并，属性名全部变成 **R.属性名**，**S.属性名**，当属性名不重复的时候可以直接写属性名。

更名(rename) ρ :

$\rho_X(E)$ 这个式子返回 E 的值同时将 E 这个表达式更名为 X;

$\rho_{X(A1, A2, \dots, A_n)}(E)$ 与上面类似，将 E 中的每个属性进行重命名

示例：基于关系 **customer(name,street,city)**，实现下列查询：

查询所有与 smith 居住在同一城市同一街道的客户

```

[customer.name(<sub>customer.street=s_add.street^customer.city=s_add.city />sub>)(customer<sub>s_add(street,city)</sub>)^street,city(

```

选择史密斯所在的元组更名为 **a_add**，然后与所有的customer 进行笛卡尔积，挑选出符合条件的元组，再投影

基本运算的分配律：投影和并可以分配： $\neg \text{pid,name}(S \cup T) = \neg \text{pid,name}(S) \cup \neg \text{pid,name}(T)$

投影和差不可分配： $\neg \text{pid,name}(S - T) \neq \neg \text{pid,name}(S) - \neg \text{pid,name}(T)$

解释：因为 **pid** 和 **name** 可能只是 S、T 的部分属性，这意味着存在这样的情况，假设 s、t 分别为 S、T 的一个元组，它们在 pid 和 name 上相等而它们本身不相等，这就会导致 s 会存在于 S-T 中，即式子左侧有 s 而式子右侧却没有。

6.1.2 关系代数形式化定义

关系代数的基本表达式：数据库中的一个关系一个常数关系 关系代数中的表达式是由更小的子表达式构成的，假设 **E1** 和 **E2** 是关系代数表达式，则下列都是关系代数表达式：
o 关系代数表达式仅限于上述运算的**有限次**复合

6.1.3 附加运算 $\cap \theta \bowtie \leftarrow$

所有的附加运算都可以通过基本运算转化而成，它没有实质地扩展关系代数的表达能力，仅仅是为了方便

集合交(intersection) \cap :所有同时出现在两个关系中的元组集合，要求**相容通过差运算来重写**： $R \cap S = R - (R - S)$

示例：查询同时选修了 **c1** 号和 **c2** 号课程的学生号

$\neg \text{sno}(\sigma_{\text{cno} = 'c1'}(SC)) \cap \neg \text{sno}(\sigma_{\text{cno} = 'c2'}(SC))$

θ 连接(θ -join):AB 之间进行笛卡尔积同时删选出给定属性之间满足一定条件的结果

o

A,B 为 R 和 S 上度数相等且可比的属性列

θ 为算术比较符，为等号时称为等值连接

自然连接(natural-join) \bowtie :

$R \bowtie S$ 从两个关系的广义笛卡尔积中选取在相同属性列 B 上取值相等的元组，并去掉重复的属性。

与等值连接的区别：自然连接中相等的分量必须是**相同的属性组**，并且要在结果中去掉重复的属性（两个关系中相同的属性在自然连接的结果关系模式中只出现一次），而等值连接则不必。

可交换，可结合

除运算(division) \div :象集 Yx 将 R 的所有属性分为两组 X 和 Y，筛选出所有属性 X 取值为 x 的元组，投影获得这个元组的 Y 属性，就是 x 的象集 示例：
 $R(X,Y) \div S(Y)$ 筛选出 X 每一个值的象集，若 Y 是此象集的子集，那么这个 X 值进入结果集。 **$R(X,Y) \div S(Y) = \neg X(R) - \neg X(\neg X(R) \times \neg Y(S) - R)$**

这个表达式的意思就是，X与Y做笛卡尔积以后得到的是X和Y的所有组合，与R做差，得到的是R中没有的X和Y的组合，也就是这个x没有在R中匹配所有的Y，不该留在结果集中，**减掉**。 **示例：**需要注意，X、Y是属性集，每个X代表X中的所有属性都相同 查询至少选修了c1和c2号课程的学生号 方案1：

$\neg \text{sno}_{\text{cno}}(SC) \div \neg \text{cno}(\sigma_{\text{cno} = 'c1' \vee \text{cno} = 'c2'}(C)) \vee$

方案2:

$\neg \text{sno}(SC \div \neg \text{sno}(\sigma_{\text{cno} = 'c1' \vee \text{cno} = 'c2'}(C))) \times$

方案二中没有事先对SC做投影，也就是说SC的所有属性都会参与到计算之中，只有score和sno都相同的才能进入结果集中 **赋值运算(assignment)：**临时关系变量 \leftarrow 关系代数表达式 **示例：** 用赋值重写为：

6.1.4 扩展关系代数

广义投影(generalized-projection)： $\neg F_1, F_2, \dots, F_n(E)$ 允许将各个属性值进行计算以后成为扩展属性值，其中扩展属性值可以更名

示例： $\neg \text{tno}, \text{sal}5/100 \text{ as income-tax}(T)$ 和 $\text{PTAX}(\text{tno}, \text{INCOME-TAX})(\neg \text{tno}, \text{sal}5/100(T))$

聚集(aggregate)G:

聚集函数中的null:

- 多重集中忽略null
- 聚集函数作用于空集合: count(Φ)=0; 其它聚集函数作用于空集合，结果为null

多重集(multiset):

常用聚集函数:

count（计数），sum（求和），average（求平均值），max（求最大值），min（求最小值）

用处：

求一组值的统计信息，返回单一值，其中属性允许值重复，如果想要改成不允许重复，需要在后面加上*distinct*，如*G_{count}distinct(sno)*（S）

示例：

求全体教工的总工资GSum(sal)(T)

注意：

结果集不是一个树枝，而是一个关系

聚集运算（分组运算）：

格式：

G1,G2,...,Gn G F1(A1),F2(A2),...,Fm(Am)（E）

解释：

将关系E按照G1,G2,...,Gn进行分组，分别计算出每一组的F1(A1),F2(A2),...,Fm(Am)值并且满足：

- 同一组中所有元组在G1,G2,...,Gn上的值相同
- 不同组中元组在G1,G2,...,Gn上的值不同

示例：查询每位学生的总成绩和平均成绩

sno Gsum(SCORE), avg(SCORE)（SC）

外连接(outer-join): 为避免自然连接时因失配而发生的信息丢失，可以假定在参与连接的一方表中附加一个取值全为空值的行，它和参与连接的另一方表中的任何一个未匹配上的元组都能匹配

- ⋈_L左外连接 = 自然连接 + 左侧表中失配的元组
- ⋈_R右外连接 = 自然连接 + 右侧表中失配的元组
- ⋈_{ALL}全外连接 = 自然连接 + 两侧表中失配的元组

外连接用基本关系代数运算表示：

$R \bowtie S = (R \bowtie S) \cup ((R - \bigcap_{A_1, \dots, A_n} (R \bowtie S)) \times (\text{null}, \dots, \text{null}))$

其中A1,...,An是R中的属性

6.1.5 关系代数例题解析：

- 查询仅选修一门课程的学生学号

$\bigcap \text{sno}(\text{SC}) - \bigcap \text{a.sno}(\sigma_{\text{a.sno} = \text{b.sno} \wedge \text{a.cno} < \text{b.cno}} (\rho_{\text{a}}(\text{SC}) \times \rho_{\text{b}}(\text{SC})))$

分析：所有选课的学生，减去选修了不止一门课的学生 其中减号后的的关系代数可以选择选修了两门以上课程的学生

- 查询平均成绩高于s2平均成绩的学生学号

$\bigcap \text{sno}(\sigma_{\text{as} > \text{bs}} (\rho_{\text{a}}(\text{sno}, \text{as})(\text{sno Gavg}(\text{score})(\text{SC})) \times \rho_{\text{b}}(\text{bs})(\text{Gavg}(\text{score})(\sigma_{\text{sno} = \text{'s2'}}(\text{SC}))))))$

分析：将s2与其他同学做个笛卡尔积然后选出比s2高的即可

- 查询平均成绩最高的学生学号

$\bigcap \text{sno}(\text{SC}) - \bigcap \text{A.sno}(\sigma_{\text{aa} < \text{ba}} (\rho_{\text{A}}(\text{sno}, \text{aa})(\text{sno Gavg}(\text{score})(\text{SC})) \times \rho_{\text{B}}(\text{sno}, \text{ba})(\text{sno Gavg}(\text{score})(\text{SC}))))$

记住一件事，avg返回的是一个值，绝不可以

$\text{sno Gavg}(\text{avg})(\rho_{\text{A}}(\text{sno}, \text{avg})(\text{sno Gavg}(\text{score})))$,

这样得到的是每个学生的最高平均成绩 这里的方式是除去所有同学之中平均成绩会比另一个同学低的同学

6.2元组关系演算

6.2.1定义

把数理逻辑的谓词演算推广到关系运算中。 查询表达式（泛式）： $\{t|P(t)\}$ (使得P（t）为真的所有的t组成的集合)

PS：

t[A]表示元组t在属性A上的取值

t表示的是元组

P是公式，由原子公式和运算符组成。

原子公式：

- $t \in R$ （t是关系R中的一个元组）
- $t[x] \theta s[y]$ （ $t[x]$ 与 $s[y]$ 为元组分量，他们之间满足比较关系 θ ）
- $t[x] \theta c$ （元组分量 $t[x]$ 与常量c之间满足比较关系 θ ） 在公式中各种运算符的优先级从高到低依次为： θ 、 \exists 和 \forall 、 \neg 、 \wedge 和 \vee 、 \square 加括号时，括号中的运算优先。

原子构造公式：

- 原子公式是公式
- 如果P是公式，那么 $\neg P$ 和 (P) 也是公式
- 如果P1,P2是公式，则 $P1 \wedge P2, P1 \vee P2, P1 \square P2$ 也是公式
- 如果P(t)是公式，R是关系，则 $\exists t \in R(P(t))$ 和 $\forall t \in R(P(t))$ 也是公式
- 公式只能由上述四种形式有限次复合组成，除此之外构成的都不是公式

6.2.2 量词

- 全称量词：“ $\forall x(P(x))$ ”表示对于域中的所有x，谓词P(x)均为真
- 存在量词：“ $\exists x(P(x))$ ”表示对于域中的某些x，谓词P(x)为真
- 量词的辖域：每个量词后面的最短公式 量词后面的变量若在作用范围内则称为约束变量，否则称为自由变量
- 格式： 课本写法： $\forall x \in S(P(x))$ 正规写法： $\exists x(x \in S \wedge P(x))$ 上述两种方式都可以

6.2.3 等价公式

（离散貌似学过，真值表也可以推出来）

- $P1 \wedge P2 \Leftrightarrow \neg (\neg P1 \vee \neg P2)$
- $\forall t \in R(P(t)) \Leftrightarrow \neg \exists t \in R(\neg P(t))$
- $P1 \sqsubset P2 \Leftrightarrow \neg P1 \vee P2$

6.2.4 元组关系演算与关系代数的等价性

- 投影: $\pi_A(R) = \{t | \exists s \in R(t[A] = s[A])\}$
- 选择: $\sigma_{F(A)}(R) = \{t | t \in R \wedge F(t[A])\}$
- 广义笛卡儿积: $R(A) \times S(B) = \{t | \exists u \in R, \exists s \in S(t[A] = u[A] \wedge t[B] = s[B])\}$
- 并: $R \cup S = \{t | t \in R \vee t \in S\}$
- 差: $R - S = \{t | t \in R \wedge \neg t \in S\}$

6.2.5 元组关系演算实现自然连接

查询计算机系老师的姓名:

$\{t | \exists u \in D(u[dname] = \text{'计算机系'}) \wedge \exists s \in T(s[dno] = u[dno] \wedge t[tname] = s[tname])\}$

6.2.6 表达式的安全性

元组关系演算有可能会产生无限关系，这样的表达式是不安全的

如 $\{t | \neg (t \in R)\}$

公式P的域 $\text{dom}(P)$:

$\text{dom}(P)$ = 显式出现在P中的值与在P中出现的关系的元组中出现的值（不必是最小集）如 $\text{dom}(\neg (t \in R))$ 是R中出现的值的集合 $\text{dom}(t \in R \wedge t[salary] > 8000)$ 是包括8000和出现在R中的所有值集合 **安全的表达式**: 如果出现在表达式 $\{t | P(t)\}$ 结果中的所有值均来自 $\text{dom}(P)$ 安全的表达式中元组个数是有限的，而不安全的表达式中元组个数是无限的，因此我们只允许安全的元组关系演算 **例题**:

解释: $\text{dom}(\neg (t \in R)) = \pi_A(R) \times \pi_B(R) - R$ $\text{dom}(\neg (t \in R))$ 是R中各属性中元素的笛卡儿积与R的差集

6.2.7 例题

几个小例子:

t的属性包括两种情况: + t属于某一个表 示例: 查询d1院系的学生

$\{t | t \quad \&\#8712; S \quad \&\#8743; t[dno] = \text{'d1'}\}$

- t的属性通过表达式定义 示例: 查询d1院系的学生的姓名

$\{t | \exists s \in S \wedge (s[sname] = t[sname] \wedge s[dno] = \text{'d1'})\}$

因为结果集是一元的，关系S是五元的，因此不能直接用 $t \in S$ ，需要引入约束变量s

注意: + 查询d1院系的学生的姓名

$\{t | \forall s \in S (s[sname] = t[sname] \wedge s[dno] = \text{'d1'})\}$

注: 运算符 \Rightarrow 运用错误，得到的为空集合。+ 查询d1学院或者学习了c1课程的学生学号

$\{t | \exists w \in S(w[dno] = \text{'d1'} \wedge t[sno] = w[sno]) \vee \exists u \in SC(u[sno] = t[sno] \wedge u[cno] = \text{'C1'})\}$

关键在于考虑变量的作用范围。 **蕴含的运用**: 查询选修了002号学生选修的全部课程的学生学号 思路: \forall 课程, 002选之 \Rightarrow 所查询同学选之

$\{t | (\exists x \in S(x[sno] = t[sno]) \wedge \forall u \in SC \exists v \in SC(u[sno] = 002 \Rightarrow (u[cno] = v[cno] \wedge t[sno] = v[sno])))\}$

\Rightarrow 经常与 \forall 一起使用

第七章 数据库设计和E-R模型

本章概述

本章关注于实体-联系（E-R）数据模型，它提供了一个找出在数据库中表示的实体以及实体间如何关联的方法。讲述一个E-R设计如何转换成一个关系模式的集合以及如何在该设计中找到某些约束。

7.2 实体-联系模型（entity-relationship,E-R）

注: E-R图中，实体必须带有属性，不能省略，后面的介绍中有省略是一种不规范的写法

7.2.1 实体集

- 实体 (entity): 客观存在可相互区分的事物（唯一标识）
- 实体集 (entity set): 是具有相同类型及共享相同性质（属性）的实体集合,组成实体集的各实体称为实体集的外延 (Extension), 实体集可相
- 域 (domain): 属性的取值范围
- 属性 (attribute): 实体集中每个成员具有的描述性性质，是将实体集映射到域的函数,在E-R图中如下图所示
 -
 - 简单属性: 不可再分的属性
 - 复合属性: 可以划分为更小属性（如将姓名分为姓与名）
 - 单值属性: 每个实体在该属性上取值唯一
 - 多值属性: 某个实体在该属性上取多个值（一个人可能有多个电话号码），为表示多值属性，使用花括号将属性名括住如 {phone_number}
 - 派生属性: 可以由其他相关属性派生出来（如年龄可以由生日计算），数据库中通常只存储派生属性的定义或依赖关系，用到时再从基础性中计算出来
 - 基础性: 基础属性

7.2.2 联系集

- 联系 (relationship): 是多个实体之间的相互关联，这些实体不必互异
- 联系集 (relationship set): $n \geq 2$ （可能相同的）个实体集上的数学关系 例如 $E1, E2, \dots, En$ 为实体集，那么联系集 R 是: $\{(e1, e2, \dots, en) | e1 \in E1, e2 \in E2, \dots, en \in En\}$ 的一个子集。而 $(e1, e2, \dots, en)$ 是一个联系
- 元成度 (degree): 参与联系集的实体集的个数

- 码(key): 参与联系的实体集的主码集合形成联系集的超码
- 参与(participation): 实体集之间的关联称为参与, 即实体参与联系
 示例: 如王军选修"数据库系统", 表示实体"王军"与实体"数据库系统"参与了联系"选修"
 全部参与: 实体集E中的每个实体都参与到联系集R中的至少一个联系 (双线表示全部参与)
 部分参与: 实体集E中只有部分实体参与到联系集R的联系中 (单线表示部分参与)

角色(role): 实体在联系中的作用称为实体的角色,当需要显示区分角色时,在连接菱形和矩形的线上加上说明性的标注以区别不同的角色

7.3 约束

7.3.1映射基数 (Mapping Cardinalities)

一个实体通过一个联系集能关联到实体的数目, 有一对一、一对多、多对多
对于ER图, 有箭头的一方代表一, 左图表示一对多, 一对一是两边都有箭头, 多对多是都没有箭头。

7.3.2基数约束

当需要更精确的约束的时候, 在横线上写出上下界l..h
0..* 代表 "多", 0..1 代表 "1"

7.3.3码

- 能唯一标识实体的属性或属性组称作超码(Superkey)
- 其任意真子集都不能成为超码的最小超码称为候选码(Candidate Key)
- 从所有候选码中选定一个用来区别同一实体集中的不同实体, 称为主码(Primary Key)

7.5 实体-联系图

关于基本结构、映射基数、复杂属性、角色以及非二元联系集已经在前面提到, 此处不再赘述

7.5.6弱实体集

- 定义: 一个实体集的所有属性都不足以形成主码, 弱实体集必须依赖于强实体集, 称作它的标识实体集或者属主实体集
- 分辨符: 用于区别依赖于某个特定强实体集的属性集合, 也称作部分码。弱实体集的主码由其依赖的强实体集主码和它的分辨符组成
- E-R图: 标示性联系是双边框菱形; 弱实体集必须双线全部参与; 分辨符用下划虚线表示

优点: 避免数据冗余 (强实体集码重复), 以及因此带来的数据的不一致性 弱实体集反映了一个实体对其它实体依赖的逻辑结构 弱实体集可以随它们的强实体集的删除而自动删除

7.6转换为关系模式

7.6.1基本转换

- 实体转换为关系模式 实体→关系, 属性→关系的属性, 弱实体→将标示性实体主码引入 复合属性→分解为简单属性, 多值属性→新的关系+所在实体主码
- 联系转换为关系模式 一个联系化成一个表 表的属性: 参与联系的实体主码+联系的属性 注: 实体集主码重名时, 参考7.3.3

7.6.2 关系模式的合并

在实体和联系各自转化为关系模式以后, 需要进行合并 + 二元一对一:
联系的主码可以是任一端实体的主码; 联系转化的表可以与任一端实体转化的表进行合并 二元一对一联系不能导致相关实体转化成的表合并, 3→2

示例: E-R图如下(假设每个实体都有属性编号和姓名)
转化成的表:

```
Dept(dno,dname)
President(pid, name)
Manage(dno,pid)//dno,pid均可作主码, 假设选dno作主码
```

表的合并:

可以: Dept+Manage=Dept(dno,dname,pid)

或者: President+Manage=President(pid,name,dno)

不能进行下述合并: Dept+Manage+President=? (不能接受的合并) + 二元一对多: 联系转变的表的主码必须是"多端"的主码 联系的表可以与多端的表进行合并, 3→2 示例: E-R图如下(假设每个实体都有属性编号和姓名)

转化成的表:

```
Dept(dno,dname)
Student(sno,sname)
SD(sno,dno,time) //dno非空
```

表的合并: Student+SD=Student(sno,sname,dno,time)//dno可以为空 + 二元多对多: 联系定义为新的关系, 主码包含两端的主码, 不能合并, 最终获得3个关系 + 多元联系: 联系转化的表 (主码包含所有多端的主码) 和实体转化的表不能合并 (即便含有一对一或一对多) + 总结: •联系转化成的表, 和实体转化成的表, 可以机械地按照上述原则合并 •实体转化成的表, 相互之间不能机械合并 •联系转化成的表, 相互之间不能机械合并 •合并以后, 能够优化的可以继续优化, 优化没有技巧, 要求: 冗余小、访问效率高、易扩展...

示例:

实体转化成表:

```
sql project(pid,pname) employee(eid,ename) supplier(sid,sname) component(cid,cname) warehouse(wid,wname)
```

联系转化为表:

```
participate(pid,eid) lead(eid,leid) //leid非空 supply(sid,pid,cid,quantity) produce(sid,cid) store(cid,wid) manager(eid,wid)
```

表的合并:

```
employee+lead+employee(eid,ename,leid)//leid可为空
```

7.7 实体-联系设计问题

7.7.1实体集还是属性

使用实体集: 想要保存关于一个属性的额外信息时, 将这个属性改为实体集 使用属性: 希望每个实体都保存一个这样的信息即可时, 而且可以简化ER图 与原来的实体以联系来发生联系属性可以简化ER图, 但是让原来的实体变得庞大实体有很多性质, 属性没有

7.7.2实体集还是联系集

使用联系集: 当描述发生在实体间的行为时 很多情况下两者皆可, 此时能用联系就不用实体, 可以简化ER图

7.7.3二元还是n元

多元可以转化为二元，但是这样会浪费存储空间，造成语义不清晰，且可能有信息丢失

7.8扩展E-R特性

7.8.1特化(specialization)

自顶而下的设计过程 实体集可能包含一些子集，子集中的实体在某些方面区别于实体集中其他实体，这种关系被称作IS-A关系。对于属性继承，子类可以继承其超类的属性及超类所参与的实体集,如下图：

7.8.2概化(generalization)

自底而上的设计过程，多个实体集根据共同的特征综合成一个较高层的实体集,是特化的逆过程，二者在E-R图中不作区分

7.8.4特化/概化上的约束

限定实体成为低层实体集的成员： 条件定义（如cloth实体集，根据sex属性，决定低层是男装还是女装） 用户定义 一个实体是否可以属于多个底层实体集：

- 不相交特化：实体集必须属于至多一个特化实体集 使用一个箭头（如instructor和secretary）
- 重叠特化：实体集可能属于多个特化实体集 分开使用多个箭头（如employee和student） 完全性约束： 全部概化：每个高层必须属于一个低层实体集 部分概化：允许高层不属于任何低层实体集

7.8.5 聚集

将关系作为抽象实体使用,可以减少冗余

附：E-R图表示法中使用的符号

第八章 关系数据库设计

本章概述

本章介绍基于函数依赖概念的关系数据库设计的规范方法。然后根据函数依赖及其他类型的数据依赖来定义范式。

8.1 好的关系设计的特点

8.1.1 设计选择：更大的模式

缺点：数据冗余大，容易出现数据不一致，不能表示某些信息

8.1.2 设计选择：更小的模式

关键：小模式不一定是好的模式，简单地模式变小不是追求目标 有损分解：模式拆分后，使用自然连接与原来的模式不符。

8.1.3 好的模式

该大则大，该小则小；同数据本质结构相吻合

8.2 原子域和第一范式

- 原子域：如果某个域的元素被认为是不可再分的单元
- 第一范式(First Normal Form,1NF)：一个关系模式R的所有的属性的域都是原子的

8.3使用函数依赖进行分解

8.3.1 函数依赖：

- 定义：关系模式上的属性 α 决定 β ，那么 β 依赖于 α ，即 $\alpha \rightarrow \beta$ 对 $\forall t, s$ ，若 $t[\alpha] = s[\alpha]$ ，则 $t[\beta] = s[\beta]$ 称 α 为决定因素， β 为被决定因素（ α 即函数决定 β ）
- 平凡函数依赖： $\alpha \rightarrow \beta$ 且 $\beta \subseteq \alpha$ 。平凡函数依赖一定成立
- 完全函数依赖： $\alpha \rightarrow \beta$ 并且任意 α 的真子集都无法决定 β 完全依赖记做：
- 部分函数依赖： $\alpha \rightarrow \beta$ 并且存在 α 的真子集，其能决定 β 部分依赖记作：
- 传递函数依赖： $\alpha \rightarrow \beta, \beta \rightarrow \alpha, \beta \dashv \alpha$ 且 $\beta \not\rightarrow \alpha$ ，则称 γ 对 α 传递函数依赖

8.3.2 码

- 超码：设K为R<U,F>的属性或属性组，若 $K \twoheadrightarrow U$ ，则称K为R的超码
- 候选码：设K为R<U,F>的超码，若U完全依赖于K，则称K为R的候选码
- 主码：若R<U,F>有多个候选码，则可以从中选定一个作为R的主码
- 主属性：包含在任一候选码中的属性，称作主属性

8.4函数依赖理论

8.4.1逻辑依赖集的闭包

- 逻辑蕴涵：关系模式R，F是其函数依赖集，如果从F的函数依赖能够推出 $\alpha \rightarrow \beta$ ，则称F逻辑蕴涵 $\alpha \rightarrow \beta$ ，记作 $F \vdash \alpha \rightarrow \beta$

- 闭包：被F所逻辑蕴涵的所有函数依赖的集合 记作 $F^+ = \{ \alpha \rightarrow \beta \mid F \vdash \alpha \rightarrow \beta \}$

- Armstrong公理系统： *自反律(reflexivity rule)：若 $\beta \subseteq \alpha$ ，则 $\alpha \rightarrow \beta$ *增广律(augmentation rule)：若 $\alpha \rightarrow \beta$ ，则 $\alpha \gamma \rightarrow \beta \gamma$ *传递律(transitivity rule)：若 $\alpha \rightarrow \beta, \beta \rightarrow \gamma$ ，则 $\alpha \rightarrow \gamma$ 其中 $\alpha, \beta, \gamma, \delta$ 均为属性集，下同
- Armstrong公理推导规则： *合并律(union rule)：若 $\alpha \rightarrow \beta, \alpha \rightarrow \gamma$ ，则 $\alpha \rightarrow \beta \gamma$ *分解律(decomposition rule)：若 $\alpha \rightarrow \beta \gamma$ ，则 $\alpha \rightarrow \beta, \alpha \rightarrow \gamma$ *伪传递律(pseudotransitivity rule)：若 $\alpha \rightarrow \beta, \beta \gamma \rightarrow \delta$ ，则 $\alpha \rightarrow \delta$ 注：由F计算F+是NP完全问题，不常用，常用的是属性集闭包的计算

8.4.2属性集的闭包

定义： 令 α 为属性集，将函数依赖集F下被 α 函数确定的所有属性的集合称作F下 α 的闭包，记作 α^+ 记作： $\alpha^+ = \{ A \mid \alpha \rightarrow A \text{ 能由F根据Armstrong公理导出} \}$

计算F下属性集 α 闭包的算法： 对任意函数依赖 $\beta \rightarrow \gamma$ ，如果 β 在当前的结果集中，就把 γ 也放入结果集里面，循环直到没有新的元素加入 用途： *判断属性集是否为超码（ α +最终的结果集是否包含了所有的属性集） *通过检验 $\beta \subseteq \alpha$ +是否成立，可以验证函数依赖 $\alpha \rightarrow \beta$ 是否成立 *是另一种计算F+的方法：对任意 $\gamma \subseteq R$ ，找出 γ^+ ，对于任意的 $S \in \gamma^+$ ，得到 $\gamma \rightarrow S$ 求解候选码的方法：（非万能方法） 对于给定的关系R(U,F)，可将其属性分为4类：

- L类：仅出现在F的函数依赖左部的属性
- R类：仅出现在F的函数依赖右部的属性
- N类：在F的函数依赖两边均未出现的属性
- LR类：在F的函数依赖两边均出现的属性 显然，L和N类一定是候选码一部分，R类一定不是 **推论：** 对于给定的关系模式R及其函数依赖集F，若 $\alpha(\alpha \subseteq U)$ 是L类和N类属性集，且 α 包含了U中的全部属性，则 α 一定是R的唯一候选码

8.4.3正则覆盖

产生原因： 数据库的更新必须保证所有函数依赖都能保持，通过测试与给定函数依赖集有相同闭包的简化集的方式，来减少开销 **函数依赖集的等价性：** 函数依赖集F，G，若 $F \Rightarrow G$ ，则称F与G等价。若F与G等价，则称F是G的一个覆盖，G是F的一个覆盖。 **无关属性(extraneous attribute)：** 去除一个函数依赖中的属性，不会改变该函数依赖集的闭包 形式化定义，考虑函数依赖 $\alpha \rightarrow \beta$ ： A在 α 中无关：如果 $A \in \alpha$ ，并且 $F \vdash (F - \{ \alpha \rightarrow \beta \}) \cup \{ (\alpha - A) \rightarrow \beta \}$ A在 β 中无关：如果 $A \in \beta$ ，并且 $(F - \{ \alpha \rightarrow \beta \}) \cup \{ \alpha \rightarrow (\beta - A) \} \vdash F$ 核心：能够被函数依赖集F逻辑蕴涵的函数依赖不必在F中写明 检验无关属性方法，考虑函数依赖 $\alpha \rightarrow \beta$ ：

- 如果 $A \in \alpha$ ，令 $\gamma = \alpha - \{A\}$ ，并计算 $\gamma \rightarrow \beta$ 是否可以由F推出，即计算在F下的 γ^+ ，如果 γ^+ 包含 β 的所有属性，则A在 α 中是无关的
- 如果 $A \in \beta$ ， $F' = (F - \{ \alpha \rightarrow \beta \}) \cup \{ \alpha \rightarrow (\beta - A) \}$ ，检验 $\alpha \rightarrow A$ 是否能由F'推出，即计算F'下的 α^+ ，如果 α^+ 包含A，则A在 β 中是无关的

正则覆盖要求：（F的正则覆盖记作Fc）+Fc与F等价 +Fc中任何函数依赖都不含无关属性 +Fc中函数依赖的左半部都是唯一的，即不存在两个依赖 $\alpha_1 \rightarrow \beta_1$ ， $\alpha_2 \rightarrow \beta_2$ 满足 $\alpha_1 = \alpha_2$

求解方式： 删除无关属性的函数依赖，合并左边相同的依赖，直至无法继续 注：检查无关属性是在当前Fc中的函数依赖，而不是F。不能同时讨论F中的两个属性的无关性，一次只能讨论一个属性。正则覆盖未必唯一 **最小覆盖Fm：** 不含无关属性且函数依赖右端属性只有一个 **例题：** 计算关系模式R（U，F）的正则覆盖 $U=\{A,B,C,D,E,F\}$ ， $F=\{AB \rightarrow C, C \rightarrow A, BC \rightarrow D, ACD \rightarrow B, BE \rightarrow C, CE \rightarrow FA, CF \rightarrow BD, D \rightarrow EF\}$ 解：由 $C \rightarrow A$ ：（ $ACD \rightarrow B$ ） \rightarrow （ $CD \rightarrow B$ ），（ $CE \rightarrow FA$ ） \rightarrow （ $CE \rightarrow F$ ）由 $CD \rightarrow B$ ， $CF \rightarrow D$ ： $CF \rightarrow B$ ，（ $CF \rightarrow BD$ ） \rightarrow （ $CF \rightarrow D$ ）因此 $F_c = \{AB \rightarrow C, C \rightarrow A, BC \rightarrow D, CD \rightarrow B, BE \rightarrow C, CE \rightarrow F, CF \rightarrow D, D \rightarrow EF\}$

8.4.4无损分解和保持依赖

- 关系模式R<U,F>的一个分解是指 $p = \{R_1 / sub < U_1, F_1 >, R_2 < U_2, F_2 >, \dots, R_n < U_n, F_n >\}$ 其中 $U = U_1 \cup U_2 \dots \cup U_n$ ，并且设 $U_i \subseteq U_j$ ， $1 \leq i, j \leq n$ 关系模式的分解是将R所有的属性分解到不同的子关系里面
- 分解的基本代数运算：投影和自然连接
- 分解的要求：无损连接分解和保持函数依赖
- 无损分解：模式分解以后进行自然连接，得到的关系依旧是原来的关系
- 保持函数依赖：原有的函数依赖依旧保持
- 判断无损连接分解（假设关系模式R(U)的分解是 $p\{R_1, R_2, R_3, \dots\}$ ） **快速法（充分条件）**（分解后的关系模式只有两个）：
 $R_1 \cap R_2 \rightarrow R_1$ 或 $R_1 \cap R_2 \rightarrow R_2$ 或 $R_1 \cap R_2 \rightarrow R_1 - R_2$ 或 $R_1 \cap R_2 \rightarrow R_2 - R_1$ 一个成立即可 **表格法（充要条件）**（分解后的关系模式多于两个）： 表格横轴为属性，纵轴为函数依赖，如果一个函数依赖与属性有关，那么这个位置的数值为 a_{ij} ，i是表示第几个属性，其他位置的数值就是 b_{ij} 。初始化以后，根据每一个函数依赖关系 $\alpha \rightarrow \beta$ 如果 β 里面有 a 那么 α 相同的几行 β 也要变成 a_{ij} ，如果有只 b 那么相同的就变成最小的那个 b 。这样循环最后如果能够得到某一行全都是 a ，则此函数依赖是无损的

例：已知R<U,F>， $U=\{A,B,C,D,E\}$ ， $F=\{A \rightarrow C, B \rightarrow C, C \rightarrow D, DE \rightarrow C, CE \rightarrow A\}$ ，R的一个分解为 $R_1(AD)$ ， $R_2(AB)$ ， $R_3(BE)$ ， $R_4(CDE)$ ， $R_5(AE)$
 $A \rightarrow C$ ，所有A中有a的都没有对应的 a_3 ，那么就所有 a_1 对应的C都改成相同的 $B \rightarrow C$ ，同理，将 a_3 对应的C改成相同的
 $C \rightarrow D$ ，将相同的C对应的D改成相同的，如果有a优先改成a 余下的步骤与前面相同，最终的结果如下： 结果表中无一行为全a，分解不是无损连接（必要性） 结果表中有一行a，分解是无损连接（充分性） **判定保持函数依赖 算法一：**对每个分解后的模式获得它的依赖关系闭包，将所有依赖关系合并，如果与原依赖闭包相同那么就保持了函数依赖 算法二：为了减少工作量，可以分别对每个函数依赖进行计算，如果刚才的算法能推出 β ，那么 $\alpha \rightarrow \beta$ 就能被保持

8.5分解算法

范式定义： 是对关系的不同数据依赖程度的要求，通过模式分解将一个低级范式转换为若干个高级范式的过程称作规范化

8.5.1 1NF

定义：关系中每一分量不可再分。即不能以集合、序列等作为属性值

8.5.2 2NF

定义：关系中的每个属性，要么是一个候选码之一，要么完全依赖于一个候选码，不可以部分依赖于候选码

8.5.3 3NF

- 定义：关系模式R<U,F>中，F+中所有函数依赖 $\alpha \rightarrow \beta$ ，至少有以下之一成立：
• $\alpha \rightarrow \beta$ 是一个平凡的函数依赖
• α 是R的一个超码
• $\beta - \alpha$ 的每个属性A都包含在R的一个候选码中。（注： $\beta - \alpha$ 的每个属性可能包含于不同的候选码中）
- 另一种定义方式：关系模式R<U,F>中，若不存在这样的码X，属性组Y及非主属性Z($Z \not\subseteq Y$)，使得下式成立： $X \rightarrow Y, Y \rightarrow Z, Y \not\rightarrow X$ ，也就是非主属性对码没有传递依赖

判断3NF的优化： 可以只考虑F上的函数依赖，而不是F+，也可以分解F上的函数依赖，让它们的右半部只包含一个属性，并用这个结果代替F **3NF分解算法：** 达到3NF且保持函数依赖和无损连接 + 求F的正则覆盖Fc + 按照Fc的每一个函数依赖对其进行分解 + 如果某一个分解符合3NF条件，算法结束，否则分解成为两个属性集，其中一个属性集为对应依赖关系的两端属性总和，另一个为当前总属性集除去对应依赖关系右侧的部分 + （可选）如果模式中有包含关系，那么删除被包含的模式

8.5.4 BCNF

定义： 关系模式R<U,F>中，F中所有函数依赖 $\alpha \rightarrow \beta$ ，至少有以下之一成立：
• $\alpha \rightarrow \beta$ 是平凡的函数依赖
• α 是R的一个超码 检查关系模式R只需要检查F上的所有函数依赖，不需要F+。 **BCNF无损连接分解算法：**（可能会丢失函数依赖） 对于每个不属于BCNF的模式上的非平凡函数依赖 $\alpha \rightarrow \beta$ ，一定有 $\alpha \rightarrow \beta \in F^+$ ，并且 α 不是超码，这个时候，把这个模式分解为两部分 $R_1 = \alpha \beta$ ， $R_2 = \alpha(R - \beta)$ ，每次分解出来至少一个是BCNF的 **结论：** 若要求分解保持函数依赖，那么分解后的模式总可以达到3NF，但不一定能达到BCNF。 **示例1：**指出下列关系模式是第几范式？并说明理由

(1)R(X,Y,Z), F={XY→Z} (2)R(X,Y,Z), F={Y→Z, XZ→Y} (3)R(X,Y,Z), F={Y→Z, Y→X, X→YZ} (4)R(X,Y,Z), F={X→Y, X→Z} (5)R(W,X,Y,Z), F={X→Z, WX→Y}

(1)BCNF，候选码为XY，F只有一个函数依赖，左半部是候选码 (2)3NF，候选码是XY和XZ，R中所有属性都是主属性 (3)BCNF，候选码是X和Y，左半部是超码 (4)BCNF，候选码是X，左半部是超码 (5)1NF，候选码是WX，Y和Z是非主属性，X→Z是非主属性对候选码的部分函数依赖 **例题8.29** 考虑关系模式r(A,B,C,D,E,F)上的函数依赖集F A→BCD BC→DE B→D D→A

- 计算B+ 属性集闭包计算很简单，就是从B开始根据依赖关系依次向里面加内容 首先加入B，根据B→D加入D，根据D→A加入A，根据A→BCD加入C，根据BC→DE加入E没有依赖关系了，得到最终的结果 B+={ABCDE}
- 证明AF是超码 超码证明比较简单，只要AF的属性集闭包是r就可以了，但是候选码的证明还需要其每一个子集都不是候选码 A→BCD,BC→DE,∴A+={ABCDE},AF+ =r

8.6多值依赖α，β，γ，δ

描述型定义： 关系模式R(U)， $\alpha, \beta, \gamma \subseteq U$ ，并且 $\gamma = U - \alpha - \beta$ ，多值依赖 $\alpha \twoheadrightarrow \beta$ 成立当且仅当对R(U)的任一关系r，给定的一对（ α_1, β_1 ）值，有一组 β 的值，这组值仅仅决定于 α 值而与 β 值无关 **形式化定义：** 关系模式R(U)， $\alpha, \beta, \gamma \subseteq U$ ，并且 $\gamma = U - \alpha - \beta$ ，对于R(U)的任一关系r，若存在元组 t_1, t_2 ，使得 $t_1[\alpha] = t_2[\alpha]$ ，那么就必然存在元组 t_3, t_4 ，使得：

$t_3[\alpha] = t_4[\alpha] = t_1[\alpha] = t_2[\alpha] \quad t_3[\beta] = t_1[\beta], \quad t_3[\gamma] = t_2[\gamma] \quad t_4[\beta] = t_2[\beta], \quad t_4[\gamma] = t_1[\gamma]$

则称 β 多值依赖于 α ，记作 $\alpha \twoheadrightarrow \beta$ 具体的例子： 若存在元组 $t_1=(c_1,t_1,b_1)$ ， $t_2=(c_1,t_2,b_2)$ ， 则也一定含有元组 $t_3=(c_1,t_1,b_2)$ ， $t_4=(c_1,t_2,b_1)$ 可以理解为是一种补充，它用于保证不论属性b取什么值都不影响属性的取值 **性质**：

- 对称性：若 $\alpha \twoheadrightarrow \beta$ ，则 $\alpha \twoheadrightarrow \gamma$ ， 其中 $\gamma=U- \alpha-\beta$
- 函数依赖是多值依赖的特例，即： 若 $\alpha \rightarrow \beta$ ，则 $\alpha \twoheadrightarrow \beta$
- 平凡的多值依赖： 若 $\alpha \twoheadrightarrow \beta$ 且 $U- \alpha-\beta=\varnothing$ 或 $\beta \subseteq \alpha$
- 传递性： 若 $\alpha \twoheadrightarrow \beta$ ， $\beta \twoheadrightarrow \gamma$ ， 则 $\alpha \twoheadrightarrow \gamma-\beta$
- 其他： 若 $\alpha \twoheadrightarrow \beta$ ， $\alpha \twoheadrightarrow \gamma$ ， 则 $\alpha \twoheadrightarrow \beta \cup \gamma$ 、 $\alpha \twoheadrightarrow \beta \cap \gamma$ 、 $\alpha \twoheadrightarrow \beta-\gamma$ ， $\alpha \twoheadrightarrow \gamma-\beta$

多值依赖与函数依赖 + 区别 函数依赖规定某些元组不能出现在关系中，也称为相等产生依赖 多值依赖要求某种形式的其它元组必须在关系中，称为元组产生依赖 + 有效性范围 • $\alpha \rightarrow \beta$ 的有效性仅决定于 α 、 β 属性集上的值，它在任何属性集 W （ $\alpha\beta \subseteq W \subseteq U$ ）上都成立 若 $\alpha-\beta$ 在 $R(U)$ 上成立，则对于任何 $\beta' \subseteq \beta$ ，均有 $\alpha-\beta'$ 成立 • $\alpha \twoheadrightarrow \beta$ 的有效性与属性集范围有关 $\alpha \twoheadrightarrow \beta$ 在属性集 W （ $\alpha\beta \subseteq W \subseteq U$ ）上成立，但在 U 上不一定成立 $\alpha \twoheadrightarrow \beta$ 在 U 上成立 \rightarrow 在属性集 W （ $\alpha\beta \subseteq W \subseteq U$ ）上成立 若 $\alpha \twoheadrightarrow \beta$ 在 $R(U)$ 上成立，则不能断言对于 β' $\subseteq \beta$ ，是否有 $\alpha \twoheadrightarrow \beta'$

第四范式（4NF） 函数依赖和多值依赖集为D的关系模式R属于4NF的条件是： 对于所有D+中形如： $\alpha \twoheadrightarrow \beta$ 的多值依赖（其中 $\alpha \subseteq R \wedge \beta \subseteq R$ ），至少有以下条件之一成立： $\alpha \twoheadrightarrow \beta$ 是一个平凡的多值依赖； α 是模式R的超码。 所有的二元联系都是4NF，4NF必是BCNF判断BCNF时不考虑多值依赖 **无损分解判定**： 令R为一关系模式，D为R上的函数依赖和多值依赖集合。令 R_1 和 R_2 是R的一个分解，该分解是R的无损分解，当且仅当下面的多值依赖中至少有一个属于D+：

$R_1 \cap R_2 \twoheadrightarrow R_1 \quad R_1 \cap R_2 \twoheadrightarrow R_2$

D在R₁上的限定是集合D₁，它包含以下内容： D+中所有只含R_i中属性的函数依赖； 所有形如 $\alpha \twoheadrightarrow \beta \cap R_i$ 的多值依赖，其中 $\alpha \subseteq R_i$ 并且 $\alpha \twoheadrightarrow \beta$ 属于D+ **4NF分解算法** 与BCNF算法相同，除了它使用多值依赖以及D+在R_i上的限定 **分解示例**： 公理系统：考虑关系模式（U,D）
推理规则：

第十章 数据存储和数据存取

本章概述

本章的重点在于索引。大家在复习的过程中会发现，存储和数据缓冲区这一部分和操作系统有相似之处。

10.1物理存储介质

10.1.1常见的存储介质

①高速缓冲存储器(Cache) ②主存储器(Main memory) ③快闪存存储器 (Flash memory) ④光学存储器(CD-ROM/DVD) ⑤磁盘 ⑥磁带存储器 注：①②都为易失性存储介质，cache和主存配合工作。 ③④⑤⑥都为非易失性存储介质，磁盘是主要的辅存，磁带主要用来脱机备份。

10.1.2存储层次

- 基本存储 访问速度最快的存储介质，但是易失(cache, 主存)，可以直接被cpu访问。
- 辅助存储 层次结构中基本存储介质的下一层介质, 非易失, 访问速度较快。如：闪存, 磁盘
- 第三级存储 层次结构中最底层的介质, 非易失，访问速度慢。如：磁带，光学存储器

10.2 文件组织和记录组织

10.2.1文件组织的基本概念

- 逻辑层面 数据库被映射到多个不同的文件：一个文件在逻辑上组织成为记录的一个序列；一个记录是多个字段的序列；
- 物理层面 每个文件分成定长的存储单元，称作块（block），块是存储分配和数据传输的基本单元。（大多数数据库默认使用4-8KB的块，数据库允许修改块的大小）一个块包含很多记录，一个块包含的确切的记录集合是由使用的物理数据组织形式所决定的。 一般假定没有记录比块更大，这个假定对于大多数数据处理应用都是现实的。 要求每条记录包含在单个块中，这个限定简化并加速数据项访问。

实例解读： 其中每一行是一条记录，每个记录包含4个字段，这些记录组成一个文件。

10.2.2定长记录

实现方法 从字节n*(i-1)开始存储记录i, n是每个记录的长度. **缺点** + 访问记录很容易，但是记录可能会分布在不同的块上。 **解决方案**：修改约束——不允许记录跨越块的边界 + 删除记录困难 删除记录所占的空间必须由文件的其他记录来填充，或者我们自己必须用一种方法标记删除的记录，使得它可以被忽略。核心思想是，使有效记录在逻辑上连续。

具体方案： 1、移动记录i + 1, . . . , n 到 i, . . . , n-1； 2、移动记录n到i； 3、不移动记录, 但是链接所有的空闲记录到一个 free list;

10.2.3变长记录

可变长度记录的几种方式 ①存储在一个文件中的记录有多个记录类型 ②记录类型允许记录中某些字段值的长度可变(如: varchar) **分槽的页结构** 为了支持变长记录，设计出了分槽的页结构。分槽的页结构一般用于在块中组织记录。 + 分槽页页头在每个块的块头（此处“页”=“块”）它的作用：记录条目的个数；记录块中空闲空间的末尾；维护一个包含每条记录位置和大小 的条目组成的数组。 + 可以将记录在一页内移动以保证记录之间没有空闲的空间，则数组中信息也要更新。

实现： 1、实际记录从块的尾部开始排列。2、块中空闲空间是连续的，在块头数组的最后一个条目和第一条记录之间。3、如果插入一条记录，在空闲的尾部给这条记录分配空间，并且将包含这条记录大小和位置的条目添加到块头中。 4、如果一条记录被删除，它所占用的空间被释放，并且它的条目被设置成删除状态，块中被删除记录之前的记录被移动，是的由此删除产生的空闲空间被重用，并且所有的空闲空间在块头数组的最后一个条目和第一条记录之间。

10.2.4大记录

对于图片、音频等数据，这些数据比块大很多，可以使用blob和clob数据类型，大对象一般存储到一个特殊文件中，而不是与记录的其他属性存储在一起，然后一个指向该对象的指针存储到包含该大对象的记录中。

10.2.5文件中记录的组织

- 堆文件 一个记录可以放在文件中任何地方只要有足够的空间。
- 顺序文件 记录根据“搜索码”的值顺序存储。
- 哈希文件 在每条记录的某些属性上计算一个哈希函数，哈希函数的结果确定了记录应放到文件的哪一块中。

*特殊的：在多表聚集文件组织中一个文件可以存储多个不同关系的记录。 动机: 将相关记录存储在同一个块上，在做多表查询时减少I/O。

10.3 数据字典存储

数据字典包含： + 关系的有关信息 比如：关系的名字，每个关系中属性的名字、类型和长度，视图的名字和视图的定义，完整性约束。 + 用户和账号信息，包括密码 + 统计和描述数据 + 文件的组织信息 比如：关系的存储组织、关系的存储位置 + 索引信息

10.4 数据缓冲区

数据缓冲区设计的目的：数据库系统尽量减少磁盘和内存之间的数据块传输数量。可以在主存中保留尽可能多的块来减少磁盘访问次数。缓冲区：部分主存用于存储磁盘块的副本。缓冲区管理：负责在主存中分配缓冲区空间的子系统。

10.4.1缓冲区管理程序

基本功能简介 当程序需要从磁盘中得到一个块时，调用缓冲区管理程序。如果这个块已经在缓冲区里，缓冲区管理程序返回这个块在主存中的地址。如果这个块不在缓冲区中，缓冲区管理程序为这个块在缓冲区中分配空间。如果缓冲区满了，按照某种算法替换（抛出）某些块，替换出的块如果被修改则需要写回磁盘。然后将这个块从磁盘中读到缓冲区中，并将这个块在主存中的地址返回给请求者。**缓冲区替换策略**

- LRU策略——系统替换掉那些最近最少使用的块
 - LRU的思想是用过去块访问模式来预测未来的访问查询已经是定义良好的访问模式（例：顺序扫描），数据库可以使用用户查询的信息来预测未来的访问。
 - 但是LRU存在缺点，比如重复扫描 例如：通过嵌套循环计算 2 个关系 r 和 s 的连接
for each tuple tr of r do for each tuple ts of s do if the tuples tr and ts match 在r中被处理过的元组，便不会被调用了，然而它们因为刚被调用不太可能被替换出去。在s中处理完一个块后，它要等待一个循环之后再被处理，然而下一个将要处理的块已经等待了一个循环，它必然是最近最少使用的块，很可能已经被置换出去了。显然我们不希望这种情况发生。
- MRU策略——替换时替换最近最常使用的块
- 为了介绍MRU策略，我们先来了解以下几个概念。
 - 被钉住的块——不允许写回磁盘的块。
 - 立即丢弃策略 一旦一个块中最后一个元组处理完毕，就命令缓冲区管理器释放这个块所占用的空间。
 - 块的强制写出 有些时候，尽管不需要一个块所占用的存储空间，但是也必须把这个块写回磁盘，这样的写操作称为块的强制写出。作用在于：主存的内容在系统崩溃时将丢失，而磁盘上的内容在系统崩溃时得以保留，块的强制写出能够保护数据。
 - 最近最常使用策略 系统必须把当前正在处理的块钉住。在块中最后一个元组处理完毕后，这个块就不再被钉住，成为最近最常使用的块，替换时替换最近最常使用的块。
 - 回看重复扫描的例子 for each tuple tr of r do for each tuple ts of s do if the tuples tr and ts match □ 一旦r中的一个元组被处理过，就不会再被使用了，因此一旦r中被处理过的元组构成一个块，即可被从主存中删除，尽管它刚刚被使用，这种策略被称为立即丢弃。现在考虑s中的元组，当s中的一个块被处理后，我们知道它要等到s中的其他块都被处理后，才能再次被访问。因此最近最常使用的s块，将是最后一个要再次访问的块，最近最少使用的s块，是即将要访问的块，这个假设与LRU策略正好相反，MRU策略如果要选择从缓冲区移除一个块，将选择最近最常使用的块（被钉住的除外）。这种情况下，MRU策略的效率明显有优势。

10.5 索引基本概念

10.5.1索引的作用

索引机制用于加快访问所需数据的速度。例如，图书馆作者目录

10.5.2索引文件

- 索引文件由如下形式的记录（被称为索引项）组成 search-key(搜索码)+pointer
- 索引文件通常远小于原始文件

10.5.3两种基本的索引类型


- 顺序索引 基于搜索码值的顺序排序
- 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，该函数称为散列函数


10.5.4索引评价指标

- 能有效支持的访问类型
- 访问时间
- 插入时间
- 删除时间
- 空间开销

10.6 顺序索引

10.6.1顺序索引的几个重要概念（容易混淆，注意区分）

- 主索引：包含记录的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主索引，也被称为聚集索引。尽管不必如此，但主索引的搜索码常常是主码。
- 辅助索引：搜索码指定的顺序与文件中记录的物理顺序不同的索引被称为辅助索引，也称为非聚集索引。下图是一个辅助索引的实例：
- 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。
- 稀疏索引：在稀疏索引中，只为搜索码的某些值建立索引项。只有索引是聚集索引时才能使用稀疏索引。

注：为了定位一个搜索码值为 K 的记录，我们需要：找到搜索码值 <K 的最大索引项 从该索引项所指向的记录开始，沿着文件中的指针查找，直到找到所需记录为止 下图是一个使用稀疏索引的实例：

10.6.2稀疏索引与稠密索引比较

- 稀疏索引的优点：所占空间较小，插入和删除时所需的维护开销也较小
- 稀疏索引的缺点：定位一条记录时，通常比稠密索引更慢
- 为文件中的每个块建一个索引项的稀疏索引是一个很好的折中

10.6.3多级索引



- 可能出现的问题：如果主索引比较大，不能放在内存中，访问效率将变低。
- 解决方案：把主索引当做一个连续的文件保留在磁盘上，创建一个它之上的稀疏索引。外层索引-主索引上的稀疏索引 内索层引-主索引文件

10.6.4索引更新

- 单级索引删除 如果被删除的记录是具有某个搜索码值的唯一记录，那么这个搜索码值同时也被删除。
 - 稠密索引 搜索码的删除与文件记录的删除类似。
 - 稀疏索引 对于对应某个搜索码值的索引项，它被删除时需要用下一个搜索码值替换该索引项，如果下一个搜索码值已经有一个索引项，此索引项被直接删除。
- 单级索引插入 用被插入记录的搜索码值进行一次检索。
 - 稠密索引 如果搜索码值没有出现在索引中，将其插入。
 - 稀疏索引 如果索引对文件中的每个块只存储一个索引项，如果这次插入创建了一个新的块，出现在新块中的第一个搜索码值被插入到索引项中。

否则不对索引做任何改变。

10.6.5多码上的索引

一个包含多个属性的搜索码称为复合搜索码。这个索引结构和其他结构不同的是搜索码是一个列表，这个搜索码可以表示为形如 (a1,...,an) 的一组值，其中a1,...,an是索引属性。索引码值按照字典顺序排序。

10.7 B+树索引文件

B+树索引文件是索引顺序文件的一种替代

10.7.1为什么采用B+树结构

索引顺序文件的缺点（主要在结构方面）+ 随着文件的增大，由于许多溢出块会被创建，索引查找性能和数据顺序扫描性能都会下降。

+ 插入和删除时，频繁重组整个文件。

+B+树索引文件的优点 + 在数据插入和删除时，能够通过小的自动调整来保持平衡。+ 不需要重组文件来维持性能。

B+树索引文件的缺点 增加文件插入和删除的时间开销，同时会增加空间开销。这是因为插入和删除可能会引发B+树的调整，并且树形结构比线性存储需要更大的空间。

总结：相较于顺序存储时频繁地调整整个文件，B+树只需要在局部调整以维持平衡，并且B+树查找每个叶节点的性能非常稳定。所以我们可以接受一定的时间和空间开销，使用这种数据结构。

10.7.2 B+树的结构

基本特征 + 从根结点到叶结点的所有路径长度是一致的。+ 每一个非根且非叶结点有n/2到n个孩子结点。+ 一个叶结点有(n-1)/2到n-1个值。+ 特殊的：如果根结点是非叶结点，它至少有两个孩子结点；如果根结点是一个叶结点(也就是说，树中没有其他结点)，它可以有0到(n-1)个值

B+树结点结构 **结构特征：** Ki 是搜索码值。Pi是指向孩子结点的指针(对于非叶结点)或者指向记录或记录桶的指针(对于叶子结点)。**+ 顺序特征：** 结点中的搜索码是有序的。K1 < K2 < K3 < ... < Kn-1(假设目前没有重复的码值) **+ B+树的叶子结点** + 指针Pi (i = 1, 2, ..., n-1) 指向搜索码值为Ki的文件记录 + 如果 Li, Lj 是叶结点并且i < j, Li的搜索码值小于或等于Lj的搜索码值 + Pn 指向按搜索码排序的下一个叶结点 **+ B+树的非叶结点** 非叶结点形成叶结点上的一个多级稀疏索引，对于一个包含m个指针的非叶结点：+ P1指针所指子树上的所有搜索码值小于K1 + 对2 ≤ i ≤ n-1, Pi指针所指子树上的所有搜索码值大于或等于Ki-1且小于Ki + Pn指针所指子树上的所有搜索码值大于或等于kn-1

10.7.3 B+树的特性

- 由于结点间通过指针进行连接，逻辑上邻近的块在物理上不一定邻近。
- B+树的一层非叶结点形成一级稀疏索引。
- B+树每层的数值的个数有如下特点：如果在文件中有K个搜索码值，树的高度不超过log_{⌈n/2⌉}(K) 从而可以有效地进行检索
- 可以高效地对主文件进行插入和删除操作，并且索引可以在对数时间内重构。

10.7.4 B+树的查询、插入和删除

这一部分内容在作为本课程先行课的数据结构课上已经作为重点学习过了，这里的重点不在于详解B+树的各种操作，因此不做介绍。如果想要了解有关内容可以回顾数据结构中的内容。

第十一章 查询处理和查询优化

本章概述

许多查询只涉及文件中的少量记录，如果要访问整个关系就会低效，为了能直接定位那少部分记录，设计与文件相关联的附加的结构。

11.1 查询处理的基本步骤

从sql语句提交到输出它的结果，期间经历了以下几个步骤：解析与翻译、优化和执行。

11.1.1解析与翻译

- 语法分析器检查语法，验证关系。
- 把查询语句翻译成系统的内部表示形式，也就是翻译成关系代数。

11.1.2执行

查询执行引擎接收一个查询执行计划，执行该计划并把结果返回给查询。

11.1.3优化

关于优化的几个重要概念 + 一个关系代数表达式可能有許多等价的表达式。+ 可以用多种不同的算法来执行每个关系代数运算。+ 用于执行一个查询的原语操作序列称为查询执行计划。

查询优化 + 使用来自数据库目录的统计信息来评估代价。+ 在所有等效执行计划中选择具有最小查询执行代价的计划。

11.2 查询代价的度量

我们的目标是提高查询的效率，那么查询效率具体的表现形式都有哪些呢？因此我们需要了解查询代价的度量。

11.2.1查询代价的度量

查询处理的代价可以通过该查询对各种资源的使用情况进行度量。这些资源包括磁盘存取，执行一个查询所用 CPU 时间，甚至是网络通信代价。

11.2.2查询代价的主要方面

在磁盘上存取数据的代价通常是主要代价。通过以下指标来对其进行度量：+ 搜索磁盘次数 * 平均寻道时间 + 读取的块数 * 平均块读取时间 + 写入的块数 * 平均块写入时间

注：为了方便计算和接下来的学习，作出以下规定：+ 只用传输磁盘块数以及搜索磁盘次数来度量查询计算计划的代价：tT – 传输一个块的时间 tS – 磁盘平均访问时间（磁盘搜索时间+旋转延迟）传输b个块以及执行s次磁盘搜索的操作代价：b * tT + s * tS + 忽略 CPU 时间（实际应用中 CPU 时间应被考虑）。+ 没有包括将操作的最终结果写回磁盘的代价。+ 若干算法可以通过使用额外的缓冲空间来减少磁盘 I/O 操作、所需数据可能已存在于缓冲池中，避免了磁盘 I/O。以上两种情况都不予以考虑。

11.3 两种关系代数运算的执行

注：下文出现的“码”可以理解为“超码”或“候选码”

11.3.1选择运算

线性搜索 + 搜索方法：系统扫描每一个文件块，对所有记录都进行测试，看它们是否满足选择条件。+ 时间代价：Cost=br*IT + tS + 时间代价分析：开始时需要做一次磁盘搜索来访问文件的第一个块，如果文件的块不是顺序存放的，也许需要更多的磁盘搜索，为了简化起见，我们忽略了这种情况。因此时间代价由br次磁盘块传输和1次磁盘搜索产生。+ 特别的：对作用在码属性上的选择操作来说，系统在找到所需记录以后可以立即停止。因此时间代价的期望为 [(br/2) 次磁盘块传输 + 1 次磁盘搜索]。+ 线性搜索可以普遍应用于各种情况，不论记录是否有序，不论是否存在索引。

索引扫描 使用索引的搜索算法，选择条件必须是建立索引的搜索码。+ 主索引，码属性等值比较 + 对于具有主索引的码属性的等值比较，我们可以使用索引检索到满足相应等值条件的唯一一条记录。+ 时间代价：Cost = (hi + 1) * (IT + tS) + 时间代价分析：索引使用B+树结构（hi是B+树的高度），索引查找需要从树根到叶节点，再加一次I/O取记录，每个这样的I/O操作需要一次搜索和一次块传输。+ 主索引，非码属性等值比较 + 因为是非码属性，所以允许重复，有可能检索多条记录。+ 时间代价：Cost = hi * (IT + tS) + tS + tI * b + 时间代价分析：因为我们为搜索码建立了主索引，所以它们在文件中的记录是有序的，我们搜索的目标在文件中一定是连续存储的。B+树的每层都有一次搜索和传输，在B+树中找到满足条件的第一个索引后，需要在磁盘上根据这个索引搜索第一个块，b是包含具有指定搜索码的块数（假定这些块是顺序存储的叶子块，并且不需要额外搜索），因此我们需要传输b个块。+ 辅助索引，等值比较 + 如果等值条件是码属性上的，该策略可以检索到满足条件的一条记录 + 时间代价：Cost = (hi + 1) * (IT + tS) + 时间代价分析：索引查找穿越树的高度，再加一次I/O取记录，每个这样的I/O操作需要一次搜索和一次块传输。+ 若索引字段是非码属性，则可检索到多条记录 + 时间代价：Cost = (hi + n) * (IT + tS) + 时间代价分析：因为在非码属性上建立辅助索引，文件中记录的顺序和搜索码指定的顺序不同，所以满足条件的n个匹配的记录可能在不同的磁盘块中，这需要每条记录一次搜索和传输。索引查找穿越树的高度，再加n次I/O取记录。+ 主索引，比较 + 建立主索引的文件记录是按搜索码的顺序排序的，对于σA>=V(r)，使用索引找到>=v的第一个元组，从这里开始顺序扫描关系。+ 时间代价：Cost = hi * (IT + tS) + tS + tI * b + 时间代价分析：树的每层一次搜索，第一个块的搜索，b是包含具有指定搜索码的块数，假定这些块是顺序存储的叶子块，并且不需要额外搜索。+ 特别的：对于σA<=V(r)，只是顺序扫描关系找到>v的第一个元组，因为文件记录按搜索码的顺序排列，所以不使用索引，以节省B+树搜索的开销。+ 辅助索引，比较 + 对于σA>=V(r)，使用索引找到第一个>=v的索引项，从这里开始依次扫描索引，找到指向记录的指针。+ 对于σA<=V(r)，只需要扫描索引的叶子页来找到指针，直到找到第一个>v的索引项。因为辅助索引文件记录无序，所以必须在B+树的叶节点中搜索，不能直接在文件中搜索。+ 时间代价：Cost = (hi + n) * (IT + tS) + 时间代价分析：n是所取记录数，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比较大，查询代价非常大。

11.3.2连接运算

在实际应用中，我们要根据代价估算来选择合适的连接。将为大家介绍5种连接，使用下面的信息作为例子： 记录数（n）：Student-5,000 sc- 10,000 磁盘块数（b）：Student-100 sc-400 + 嵌套循环连接
for each 元组 tr in r do begin for each 元组 ts in s do begin 测试元组对 (tr,ts) 是否满足连接条件θ 如果满足，把 tr • ts 加入到结果中 end end + r 被称为连接的外层关系，而s称为连接的内层关系。+ 无需索引，并且不管连接条件是什么。+ 代价很大，因为算法逐个检查两个关系中的每一对元组AθB + 代价分析：+ 在最坏的情况下，缓冲区只能容纳每个关系的一个数据块，这时共需 nrbs + br 次块传输 nr + br 次磁盘搜索 + 如果较小的关系能被放入内存中，使用它作为内层关系，这时共需 br + bs 次块传输 2 次磁盘搜索 + 最坏的可用内存情况下的成本估算，用 student 作为外层关系: 5000400 + 100 = 2,000,100 次块传输 5000 + 100 = 5100 次磁盘搜索 + 最坏的可用内存情况下的成本估算，用 sc 作为外层关系: 10000*100 + 400 = 1,000,400 次块传输 10,400 次磁盘搜索

- 块嵌套循环连接
for each 块 Br of r do begin for each 块 Bs of s do begin for each 元组 tr in Br do begin for each 元组 ts in Bs do begin 测试元组对 (tr,ts) 是否满足连接条件θ 如果满足，把 tr • ts 加入到结果中 end end end end
 - 概述：它是嵌套循环连接的优化，其中内层关系的每一块与外层关系的每一块对应，形成块对，在每一个块对中，一个块的每一个元组与另一个块的每一个元组组成组对，从而得到全体组对。
 - 分析：我们发现，在块嵌套循环连接中外层关系中的一个元组与内层关系的当前块的每一个元组比较完之后，并没有像嵌套循环连接一样立即将内层关系的当前块置换出去，而是用外层关系的当前块的每一个元组都与它比较之后在将其置换出去。这样对于外层关系中的每一个块，内层关系的每一块只需读取一次，不需要对每一个元组读一次，明显减少了缓冲区置换的次数。
 - 代价分析：
 - 最坏情况 br*bs + br次块传输 2 * br 次磁盘搜索
 - 最好情况 br + bs 次块传输 2 次磁盘搜索
 - 最坏的可用内存情况下的成本估算，用student作为外层关系: 块传输：100400+100=40100 块搜索：2100=200
 - 补充：改进嵌套循环与块嵌套循环算法
 - 在块嵌套循环中，如果内存中有M块，使用M - 2个磁盘块作为外层关系的块单元；使用剩余的两个块作为内层关系和输出的缓冲区。 Cost = rbr / (M-2)¹ *bs + br次块传输 + 2 *br / (M-2)¹ 次磁盘搜索
 - 如果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。
 - 使用缓冲区的剩余块，对内层循环轮流做向前、向后的扫描（使用 LRU 替换策略）。
 - 若内层循环连接属性上有索引，可以用更有效的索引查找法替代文件扫描法。
 - 索引循环嵌套连接
 - 适用场景：当连接是自然连接或等值连接，并且内层关系的连接属性上存在可用索引时，索引查找法可以替代文件扫描法。对于外层关系r的每一个元组tr，可以利用索引查找满足与tr的连接条件的s中的元组。
 - 代价分析：
 - 最坏的情况: 缓冲区只能容纳关系r的一块和索引的一块，对于外层关系r的每一个元组，需要对关系s进行索引查找。
 - 连接的时间代价：br (IT + tS) + nr*c (c 是使用连接条件对关系 s 进行单次选择操作的代价)
 - 如果两个关系r和s上均有索引时，一般把元组较少的关系作外层关系时效果较好。因为外层关系决定了搜索次数。

11.4 两种表达式计算方法

目前只研究了单个关系运算如何执行，下面讨论如何计算包括多个运算的表达式。 计算一个完整表达式树的两种方法：

- 物化:输入一个关系或者已完成的计算，产生一个表达式的结果，在磁盘中物化它，重复该过程。（可以把物化理解为创建一个确实存在的临时关系）
- 流水线:一个正在执行的操作的部分结果传送到流水线的下一个操作，使得两操作可同时进行。

11.4.1物化计算

- 概述：从最底层开始，执行树中的运算，对运算的每个中间结果创建文件，然后用于下一层运算。
- 特点：
 - 任何情况下，物化计算都是永远适用的。
 - 将结果写入磁盘和读取它们的代价是非常大的。

11.4.2流水线执行

- 概述：同时执行多个操作，一个操作的结果传递到下一个，不储存中间结果。
- 特点：
 - 流水线并不总是可行的，比如需要排序的归并连接和产生配对的散列链接。
 - 比实体化代价小很多。
 - 对于有效流水线，当作为输入的元组被接收时，立即使用计算算法得到输出元组。

11.5 查询优化

11.5.1查询优化概述

- 查询优化就是从多个可能的策略中，找出最有效的查询执行计划的一种处理过程。
- 优化一方面可以在关系代数级别发生；另一方面是为处理查询选择一个详细的策略，比如执行算法、选择索引等。
- 结合实例体会查询优化 例name,title(odname = 'music'(instructor⋈ (teaches ⋈ [courseid,title(course)]))) 左侧的表达式树将产生很大的中间关系，instructor ⋈ (teaches ⋈ [courseid,title(course)，但是我们只对music学院的教师感兴趣，因此，优化后的表达式树变成右侧的。
- 基于代价的优化步骤
 - 使用等价规则产生逻辑上的等价表达式

- 注解结果表达式来得到替代查询计划
- 基于代价估计选择代价最小的计划（估计的根据是系统中的各种统计信息）

11.5.2关系表达式的转换

- 等价关系表达式 如果两个关系代数表达式在所有有效数据库实例中都会产生相同的元组集，则称它们是等价的。
- 等价规则
 - 等价规则的具体内容
 - 合取选择运算可以被分解为单个选择运算的序列
 - 选择运算满足交换律
 - 一系列投影中只有最后一个运算是必需的，其余的可省略
 - 选择操作可与笛卡尔积以及θ连接相结合 $\sigma_{\theta}(E1 \times E2) = E1 \bowtie_{\theta} E2$
 - θ连接运算满足交换律 $E1 \bowtie_{\theta} E2 = E2 \bowtie_{\theta} E1$
 - 自然连接运算满足结合律 $(E1 \bowtie E2) \bowtie E3 = E1 \bowtie (E2 \bowtie E3)$
 - θ连接具有以下方式的结合律 $(E1 \bowtie_{\theta 1} E2) \bowtie_{\theta 2 \wedge \theta 3} E3 = E1 \bowtie_{\theta 1 \wedge \theta 3} (E2 \bowtie_{\theta 2} E3)$
 - 选择运算在下面两个条件下对θ连接运算具有分配率
 - 当选择条件θ0的所有属性只涉及参与连接运算的表达式之一 (比如 E1)时，满足分配率： $\sigma_{\theta 0}(E1 \bowtie_{\theta} E2) = (\sigma_{\theta 0} E1) \bowtie_{\theta} E2$
 - 当选择条件θ1只涉及E1的属性，选择条件θ2只涉及E2的属性时，满足分配率： $\sigma_{\theta 1 \wedge \theta 2}(E1 \bowtie_{\theta} E2) = (\sigma_{\theta 1}(E1)) \bowtie_{\theta} (\sigma_{\theta 2}(E2))$
 - 投影运算在下列条件下对θ连接运算具有分配率
 - 假设连接条件θ只涉及L1∪L2中的属性 $\Pi_{L1 \cup L2}(E1 \bowtie_{\theta} E2) = (\Pi_{L1}(E1)) \bowtie_{\theta} (\Pi_{L2}(E2))$
 - 考虑连接 E1∗θE2 令L1和L2分别代表 E1 和 E2 的属性集 令L3是E1中出现在连接条件θ中但不在L1∪L2中的属性 令L4是E2中出现在连接条件θ中但不在L1∪L2中的属性 $\Pi_{L1 \cup L2}(E1 \bowtie_{\theta} E2) = \Pi_{L1 \cup L2}((\Pi_{L1 \cup L3}(E1)) \bowtie_{\theta} (\Pi_{L2 \cup L4}(E2)))$
 - 集合的并与交满足交换律 $E1 \cup E2 = E2 \cup E1$ $E1 \cap E2 = E2 \cap E1$
 - 集合的并与交满足结合律 $(E1 \cup E2) \cup E3 = E1 \cup (E2 \cup E3)$ $(E1 \cap E2) \cap E3 = E1 \cap (E2 \cap E3)$
 - 选择运算对∪,∩和-运算具有分配率 $\sigma_{\theta}(E1 \cup E2) = \sigma_{\theta}(E1) \cup \sigma_{\theta}(E2)$ 上述规则将“-”替换成∩或∪时也成立。 $\sigma_{\theta}(E1 - E2) = \sigma_{\theta}(E1) - \sigma_{\theta}(E2)$ 上述规则将“-”替换成∩或∪时不成立
 - 投影运算对并运算具有分配率 $\Pi_L(E1 \cup E2) = (\Pi_L(E1)) \cup (\Pi_L(E2))$
 - 等价规则使用的原则
 - 尽可能早地执行选择操作以减小被连接的关系的大小。
 - 尽可能早地执行投影操作以减小被连接的关系的大小。
 - 在多重连接中，尽量把产生较小结果的连接放在前面以产生较小的临时关系。

11.5.3表达式结果集统计大小的估计

一个操作的代价依赖于它的输入的大小和其他统计信息。我们来了解一下，系统提供了哪些统计信息，然后再看如何利用它们。**统计信息** + 目录信息 nr:关系r的元组数 br:包含关系r中元组的磁盘块数 lr:关系r中每个元组的字节数 fr:关系r的块因子，一个磁盘块能容纳的关系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，该值与A(r)的大小相同 假设关系r的元组物理上存储于一个文件中，则下面的等式成立： + 直方图 大多数数据库将每个属性的取值分布另存为一张直方图，如果没有直方图信息，优化器将假设数据分布是均匀的。注：一个直方图只占用很少的空间，因此不同的属性上的直方图可以存储在系统目录里。等宽直方图：把取值范围分成相等大小的区间。等深直方图：调整区间分解，以使落入每个区间的取值个数相等。**估计方法** + $\sigma A = v(r) nr / V(A, r)$ ：满足选择的记录数 + $\sigma A \leq v(r) + c$ 表示满足条件的元组的估计数 + 如果 min(A,r) 和 max(A,r) 可存储到目录上，当v小于记录的最小值时，c为0；当v大于记录的最大值时，c为nr。否则c为下式： 注：如果存在直方图，可以得到更精确的估计。不存在统计信息时，c被假设为 $nr / 2 + \text{选中率} \times \text{条件}\theta \text{的选中率} = \text{关系}r \text{上一个元组满足}\theta \text{的概率}$ 。 + 合取 $\sigma_{\theta 1 \wedge \theta 2 \wedge \dots \wedge \theta n}(r)$ + 析取 $\sigma_{\theta 1 \vee \theta 2 \vee \dots \vee \theta n}(r)$ + 取反 $\sigma_{\neg \theta}(r) = nr - \text{size}(\sigma_{\theta}(r))$ + 连接运算 + 笛卡尔积r x s包含nr.ns个元组，每个元组占用sr + ss个字节。 + 若R∩S =∅, 则r∩s与r x s结果一样。 + 若R∩S是R的码，则可知s的一个元组至多与r的一个元组相连接。因此，r∩s的元组数不会超过s 元组的数目。 + 若R∩S构成了S中参照R的外码，r ∩ s中的元组数正好与s中的元组数相等。 + 若R∩S既不是R的码也不是S的码，假定每个值等概率出现。我们假设 r 中的所有元组r ∩ s中产生的元组个数估计为下式：
注：上述估计是在各个值等概率出现的这一假设前提下做出的，如果这个假设不成立，则必须使用更复杂的估算方法。直方图可以改善上述结果，如果两个直方图有相似的区间，可以在每个区间中使用上述估计方法。

11.5.4执行计划选择

理论与实际结合 当选择执行计划时，必须考虑执行技术的相互作用。为每个操作独立地选择代价最小的算法可能不会产生最佳的整体算法。实际的查询优化器合并了以下两大方法中的元素： + 搜索所有的计划，基于代价选择最佳的计划 + 使用启发式方法选择计划 **基于代价的优化** + 基本理念： 从给定查询等价的查询计划执行空间进行搜索，并选择估计代价最小的一个。 + 缺点 对于复杂查询来说，搜索整个可能的空间代价太高。考虑为表达式 r1 ∗r2 ∗... ∗rn寻找最佳连接顺序，该表达式有 (2(n-1))/(n-1)! 个不同的连接顺序，对于n=7, 此数变为 665280, 对于 n=10, 此数大于 176 亿! **启发式优化** + 基本理念： + 启发式优化通过使用一系列规则转化查询树，这通常能改善执行性能。 + 尽早执行选择运算 (减少元组数目)。 + 尽早执行投影运算 (减少属性数目)。 + 在其他类似运算之前，执行能对关系进行最大限制的选择和投影运算（例如，能得到最少的结果的运算） + 查询优化器 + 许多优化器只考虑左深连接顺序，使用启发式规则在查询树中对选择和投影进行下推，减少优化的复杂性，生成适合流水线执行的计划。 + 一些查询优化器整合启发式选择和替代访问方法的生成。常用方法： + 启发式重写嵌套块结构和聚集 + 对每个块通过基于代价的连接顺序进行优化 + 如果优化代价甚至比计划代价还要高，提早停止优化的优化代价预算。 + 重用以前的计算计划的计划缓存，查询在短时间内被重新提交，节省反复优化的开销。

第十二章 事务

本章概述

构成单一逻辑工作单元的操作集合称作事务，为保证事务的正确执行，要么执行整个事务，要么属于该事务的操作一个也不执行。

12.1 事务的基本概念

12.1.1事务定义

事务是访问并可能更新各种数据项的一个程序执行单元。简单讲，这些操作要么都做，要么都不做，是一个不可分割的工作单位。比如现实生活中的银行转账，要么转账成功，要么没有转账，不可能出现转出去但对对方没收到的情况。注：**SQL中事务的定义**： Commit work表示提交，事务正常结束。Rollback work表示事务非正常结束，撤消事务已完成的操作，回滚到事务开始时状态。

12.1.2事务特性（ACID）

- 原子性 事务中包含的所有操作要么全做，要么全不做。注：破坏一致性的情况往往发生在系统故障时，会出现事务执行到一半被中断的情况，所以原子性由恢复系统实现。
- 一致性 事务的隔离执行必须保证数据库的一致性。一致性是指，事务开始前，数据库处于一致性的状态；事务结束后，数据库必须仍处于一致性状态；事务的执行过程中可以暂时的不一致。在串行调度的情况下，事务一个接一个地执行，每条事务都按照顺序不受干扰的执行完。然而在并发调度时，很可能两个事务交替访问一个数据项，此时如不加以控制，极有可能破坏一致性。因此，数据库的一致性状态由用户来负责，由并发控制系统实现。
- 隔离性 系统必须保证事务不受其它并发执行事务的影响。对任何一对事务T1，T2，在T1看来，T2要么在T1开始之前已经结束，要么在T1完成之后再开始执行。隔离性的强弱可以根据需要变动，隔离性通过并发控制系统实现。
- 持久性 一个事务一旦提交之后，它对数据库的影响必须是永久的。要保证系统发生故障不能改变事务的持久性，因此持久性通过恢复系统实现。注：可见并发控制系统和恢复系统是保证事务特性的两大重要工具，随后会有针对这两个系统的详细介绍。

12.2 事务调度

12.2.1什么是事务调度

- 事务的执行顺序称为一个调度(schedule)，表示事务的指令在系统中执行的时间顺序。
- 一组事务的调度必须保证：包含所有事务的操作指令；一个事务中指令的顺序必须保持不变。（完整且有序）

12.2.2事务调度的两种模式

- 串行调度 在串行调度中，属于同一事务的指令紧挨在一起。能保证事务的特性，但是极大的牺牲了系统的效率。
- 并行调度 在并行调度中，来自不同事务的指令可以交叉执行。不一定能保证事务的特性，当并行调度等价于某个串行调度时，则称它是正确的。

12.2.3并行与串行

基本比较 + 并行事务会破坏数据库的一致性。+ 串行事务效率高。

并行的优点 + 一个事务由不同的步骤组成，所涉及的系统资源也不同。这些步骤可以并发执行，以提高系统的吞吐量(throughput)。+ 系统中存在着周期不等的各种事务，串行会导致难于预测的延迟。如果各个事务所涉及的是数据库的不同部分，采用并行会减少平均响应时间(average response time)。

核心问题 在保证一致性的前提下最大限度地提高并发度。

并发操作面临的问题 + 丢失修改 (lost update) 丢失修改是指事务1与事务2从数据库中读入同一数据并修改，事务2的提交结果破坏了事务1提交的结果，导致事务1的修改被丢失。+ 不可重复读 (non-repeatable read) 不可重复读是指事务1读取数据后，事务2执行更新操作，使事务1无法再现前一次读 取结果。注：事务1读取某一数据后，可能的三类不可重复读：+ 事务2对其做了修改，当事务1再次读该数据时，得到与前一次不同的值。+ 事务2删除了其部分记录，当事务1再次读取数据时，发现某些记录神秘地消失了。+ 事务2插入了一些记录，当事务1再次按相同条件读取数据时，发现多了一些记录。

注：后两种不可重复读有时也称为幻影现象。+ 读“脏”数据 (dirty read) 事务1修改某一数据，并将其写回磁盘，事务2读取同一数据后，事务1由于某种原因被撤消，这时事务1已修改过的数据恢复原值，事务2读到的数据就与数据库中的数据不一致，是不正确的数据，又称为“脏”数据。注：并行操作面临的问题将由并发控制系统解决。+ 并行调度的原则 并行调度应该在某种意义上等价于一个串行调度。+ 数据库系统的调度应该保证任何调度执行后数据库总处于一致状态。+ 通过保证任何调度执行的效果与没有并发执行的调度执行效果一样，可以保证数据库的一致性。+ 冲突可串行化 + 指令顺序 考虑一个调S中的两条连续指令（仅限read与 write操作）Ii与Ij，分别属于事务Ti与Tj

Ii = read(Q), Ij = read(Q); Ii = read(Q), Ij = write(Q); Ii = write(Q), Ij = read(Q); Ii = write(Q), Ij = write(Q); 在①情况下，Ii与Ij的次序无关紧要。其余情况下，Ii与Ij的次序不同，其执行结果也不同，数据库最终状态也不同。+ 冲突指令 当两条指令是不同事务在相同数据项上的操作，并且其中至少有一个是write指令时，则称这两条指令是冲突的。如在②、③、④情况下，Ii与Ij是冲突的。特别的：非冲突指令交换次序不会影响调度的最终结果。+ 冲突等价 如果调度S可以经过一系列非冲突指令交换转换成调度S'，则称调度S与S'是冲突等价的(conflict equivalent)。+ 冲突可串行化 当一个调度S与一个串行调度冲突等价时，则称该调度S是冲突可串行化的(conflict serializable)。

下面是一个冲突可串行化的例子： 注：也存在结果相同，单非冲突等价的例子。我们不能把这种极个别现象当做普遍存在的。下面就是一个这样的例子： **冲突可串行化的判定——优先图** + 优先图构造方法： 一个调度S的优先图是这样构造的：它是一个有向图G

= (V, E)，V是顶点集，E是边集。顶点集由所有参与调度的事务组成，边集由满足下述条件之一的边Ti→Tj组成：

①在Ti执行read(Q)之前，Ii执行write(Q) ②在Tj执行write(Q)之前，Ti执行read(Q) ③在Tj执行write(Q)之前，Ti执行write(Q) 注：有向边从先执行的事务出发，除非两个事务都执行read，否则都形成一条边。+ 并行转串行准则： 如果优先图中存在边Ti→Tj，则在任何等价于S的串行调度S'中，Ti都必须出现在Tj之前。+ 冲突可串行化判定准则： 如果调度S的优先图中有环，则调度S是非冲突可串行化的。如果图中无环，则调度S是冲突可串行化的。

12.2.4可恢复性

可恢复调度 对于每对事务T1与T2，如果T2读取了T1所写的的数据，则T1必须先于T2提交。注：事务的恢复：一个事务失败了，应该能够撤消该事务对数据库的影响。如果有其它事务读取了失败事务写入的数据，则该事务也应该撤消。**无级联调度** 对于每对事务T1与T2，如果T2读取了T1所写的的数据，则T1必须在T2读取之前提交。注：无级联调度比可恢复调度的要求更高，它不仅是可恢复的，而且还避免了写数据回滚可能造成的一系列事务的回滚。

12.2.5事务隔离性级别

事务隔离性的实质： 事务的隔离性实质上是数据库的并发性与一致性的函数。随着事务隔离级别的上升，数据库的一致性随之上升，而并发性反而下降。事务隔离的这种特性实际上会影响一个应用的性能和数据完整性，例如，对于性能要求较高的应用比如信用卡处理等，您可以适当降低其事务隔离级别，以提高整个应用的并发性（但是会降低数据的完整性）；对于并发量较小的应用比如财务处理等，您可以适当提高其事务隔离级别，以提高数据的完整性（但是会降低应用的性能）。**事务隔离级别，按照隔离级别从低到高的顺序：** ①未提交读②已提交读③可重复读④可串行化 + 未提交读 + 允许读取未提交数据。（当事务A更新某条数据时，不容许其他事务来更新该数据，但可以读取。） + 结合实例理解未提交读： 在t2时刻，事务A对数据库进行了一次更新操作，而它提交之前，事务B就可以在t3时刻观察到这种变化，读取到更新后的价格（94.23）；也就是说，事务A中的更新操作完全没有被隔离。如果事务A因为异常回滚，那么事务B中读取的数据就是脏数据。这一隔离级别违反了最基本的ACID特性，因此很多数据库都不支持（包括Oracle）。+ 已提交读 + 只允许读取已提交数据，但不要求可重复读。（当事务A更新某条数据时，不容许其他事务进行任何操作包括读取，但事务A读取时，其他事务可以进行读取、更新） + 结合实例理解已提交读： 当事务A在t2时刻更新了价格（94.23）之后，事务B在t3时刻仍然看不到该更新，此时读取价格仍然是90.00。这是一种使用较多的隔离级别，它既允许了事务B获取数据（支持并发性），同时又隐藏了其它事务（事务A）对该数据的更新，直到（事务A）提交的那一刻为止。几乎所有的数据库都支持“读已提交”的隔离级别，并且大部分将其作为默认的隔离级别。

- 可重复读
 - 只允许读取已提交数据，而且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，但是该事务不要求与其他事务可串行化。
 - 结合实例理解可重复读： 尽管在事务B执行期间，事务A插入了一条数据（QRS），但是在事务B在t2时刻查询所得的结果依然和t0时刻一样（不包含QRS），即使到了t4时刻，事务A已经提交了也是如此（这一点不同于“读已提交”）。只有在事务B也提交了，它才会看见事务A对数据库所作的修改。值得注意的是，该隔离级别下，会在被查询或修改的数据上加上读写锁，因此任何想要修改该数据的其它事务会等待（或失败），直到“可重复读”的事务提交为止。
- 可串行化
 - 保证可串行化调度
 - 结合实例理解可串行化 在该隔离级别下，所有同时到达的事务将会“排队进入”，保证每次只允许一个事务操作数据。使用“串行化”的隔离级别，应用的并发性明显下降，而数据完整性则显著提高。

12.3 并发控制

为了使并行系统中的事务符合数据一致性，人们做出了很多巧妙的并发控制方案。这里将为大家介绍以下三种：+ 基于锁的协议 + 基于时间戳的协议 + 基于有效性检查的协议

12.3.1基于锁的协议

锁 + 两种封锁类型 + 排它锁 (exclusive lock，简记为X锁) + 共享锁 (Share lock，简记为S锁) + 排他锁 + 排它锁又称为写锁。+ 若事务T对数据对象Q加上X锁，则事务T既可以读又可以写Q，其它任何事务都不能再对Q加任何类型的锁，直到T释放A上的锁。+ 共享锁 + 共享锁又称为读锁。+ 若事务T对数据对象Q加上S锁，事务T可读但不能写Q，其它事务只能再对Q加S锁，而不能加X锁，直到T释放Q上的S锁。注：可以看出排他锁的封锁级别更高，共享锁允许多个事务同时读取。如果这两种锁的优先级不加约束，会出现严重的后果。+ 饥饿 + 饥饿产生的原因 假设系统中有一系列事务Ai读一项数据，还有一个事务B需要更新同一项数据。当A1已经加上共享锁后，B只能等待其完成后再加排他锁，然而其余的Ai型事务可能一个接一个的前来对数据加共享锁，B只好一直等待。概括一下：不断出现的申请并获得S锁的事务，使申请X锁的事务一直处在等待状态。+ 饥饿的防止 规定两种锁的优先级，排他锁的优先级高于共享锁。对申请S锁的事务，如果有先于该事务且等待的加X锁的事务，令申请S锁的事务等待。

封锁协议 + 在运用X锁和S锁对数据对象加锁时，需要约定一些规则：封锁协议（Locking Protocol） 何时申请X锁或S锁 持锁时间、何时释放 + 不同的封锁协议，在不同的程度上为并发操作的正确调度提供一定的保证。+ 封锁协议限制了可能的调度数目，这些调度组成的集合是所有可能的可串行化调度一个真子集。

两阶段封锁协议 + 定义：每个事务分两个阶段提出加锁和解锁申请。增长阶段(growing phase): 事务可以获得锁，但不能释放锁。缩减阶段(shrinking phase): 事务可以释放锁，但不能获得新锁。+ 两阶段封锁协议的特性： + 并行执行的所有事务均遵守两阶段协议，则对这些事务的所有并行调度策略都是可串行化的。也就是说，所有遵守两阶段锁协议的事务，其并行执行的结果一定是正确的。+ 事务遵守两阶段锁协议是可串行化调度的充分条件，而不是必要条件。即可串行化的调度中，不一定所有事务都必须符合两阶段协议。 + 普通的两阶段封锁协议下不能避免死锁。+ 普通的两阶段封锁协议下不能避免级联回滚。+ 两阶段封锁协议的变体 调度除了应该是可串行化的以外，还需要是无级联的。因此对两阶段封锁协议做一些约定，以实现无级

联。+ 严格两阶段封锁协议 事务持有的所有排他锁必须在事务结束后，方可释放。+ 强两阶段封锁协议 事务提交之前，不得释放任何锁。

多粒度封锁协议 + 概述： 到目前为止，我们讨论的并发控制只是将一项数据作为控制单元，在很多情况下，我们需要同时操作许多项数据，将它们组合在一起作为同步单元。假如有一项事务要访问整个数据库，要是对每一项数据逐一加锁，加锁将会成为巨大的负担。因此我们可以在整个数据库上加锁。+ 单一粒度的缺点 + 封锁粒度大：并发性低 + 封锁粒度小：访问大粒度数据加锁量巨大 + 多粒度的优点 根据访问数据的粒度，确定封锁的粒度。以求加锁量有限，并可获得最大的并发性 + 多粒度封锁的基本原则 + 大粒度数据由小粒度数据组成。+ 允许对不同粒度数据进行封锁。+ 事务对大粒度数据加锁，隐含地对组成大粒度数据的所有小粒度数据加锁。+ 多粒度封锁判定授予锁 + 申请小粒度锁的判定 + 判定在申请数据上有没有不相容锁。+ 判定在申请数据相关大粒度数据上，有没有不相容锁。+ 粒度的层次有限，本判定不困难 + 申请大粒度锁的判定 + 判定在申请数据上有没有不相容锁。+ 判定在申请数据相关小粒度数据上，有没有不相容锁；如：封锁表，要判定每个元组上有没有不相容锁 + 小粒度的数据量可能巨大，本判定困难。+ 优化申请大粒度锁 + 意向锁：如果一个节点加上了意向锁，则意味着要在树的较低层进行显示加锁。+ 意向锁添加时机：在一个节点显式加锁之前，该结点的全部祖先均加上了意向锁。+ 意向锁的作用：事务判定是否能够成功地给一个结点加锁时，不必搜索整棵树。相当于一个标志，当节点上有意向锁时，表明它的下一级数据有一个或多个正在被其他事务访问。此时，只用观察当前节点有无意向锁，即可知道能否对当前节点的所有后代加锁，不用逐一检测下一级数据的锁。+ 三种意向锁：共享意向锁（IS）/排他意向锁（DO）/共享排他意向锁（SDO）+ 多粒度封锁协议 + 遵从锁的相容矩阵 + 根结点必须首先加锁，可以加任何类型的锁 + 仅当Ti对Q的父结点持有IX或IS锁时，Ti对于结点Q加S锁或者IS锁 + 仅当Ti对Q的父结点持有IX或SDX锁时，Ti对于结点Q加X、SDX、IX锁 + 仅当Ti未曾对任何结点解锁时，Ti可以对结点加锁（两阶段的）+ 仅当Ti当前不持有Q的子节点的锁时，Ti可以对节点Q解锁

下图是多粒度封锁相容矩阵：

12.3.2基于时间戳的协议

另一种解决事务可串行化的次序的方法是事先选定事务的次序。**概述** + 时间戳排序协议的目标： 令调度冲突等价于按照事务开始早晚次序排序的串行调度。+ 时间戳排序协议的基本思想： 开始早的事务不能读开始晚的事务写的数据。 开始早的事务不能写开始晚的事务已经读过或写过的数据。

事务的时间戳 对于系统中的每一个事务Ti，将唯一的时间戳与它相联系，记为TS(Ti)。+ 时间戳的两种简单方法：系统时钟+逻辑计数器 + 事务的时间戳决定了串行化顺序。时间戳的大小标志着事务发生的早晚。

数据项时间戳 + W-timestamp(Q)：表示成功执行write(Q)的所有事务的最大的时间戳。+ R-timestamp(Q)：表示成功执行read(Q)的所有事务的最大的时间戳。注：不是最后执行Read(Q)的事务的时间戳。 例如：TS(T1)=1;TS(T2)=2; T2:read(Q) //r-ts(Q)=2 T1:read(Q) //r-ts(Q)=2 (≠1!)

时间戳排序协议 + 假设事务Ti发出read(Q) + 如果TS(Ti)< W-timestamp(Q)，则Ti需读入的Q值已被覆盖。因此，read操作被拒绝，Ti回滚。+ 如果TS(Ti)>= W-timestamp(Q)，则执行read操作，R-timestamp(Q)被设为R-timestamp(Q)和TS(Ti)两者的最大值。+ 假设事务Ti发出write(Q) + 如果TS(Ti)< R-timestamp(Q)，则Ti产生的Q值是先前所需要的值，且系统已假定该值不会被产生。因此，write操作被拒绝，Ti回滚。+ 如果TS(Ti)< W-timestamp(Q)，则Ti试图写入的Q值已过时。因此，write操作被拒绝，Ti回滚。+ 否则，执行write操作，将W-timestamp(Q)设为TS(Ti)。下面是一个时间戳排序协议的示例：

时间戳排序协议的特性 + 保证冲突可串行化，冲突可串行化的调度不一定能被时间戳排序协议调度出来。+ 无死锁。事物如不满足协议即回滚，不会出现事务间相互等待的情况。+ 存在饥饿现象。事务可能被反复回滚、重启。+ 不能保证可恢复性。可以扩展协议以保证可恢复性，如跟踪提交依赖等。

时间戳排序协议的优化 + 与两阶段封锁协议和多版本协议相结合。+ 调度可恢复方法：（下列之一）+ 所有的写操作都在事务末尾执行，在写操作正在执行时，任何事务都不允许访问已写好的任何数据项。+ 对未提交数据项的读操作，被推迟到更新该数据项的事务提交之后 + 事务Ti读取了其他事务所写的的数据，只有在其他事务提交之后，Ti才能提交 + Thomas写规则 假如事务Ti发出write(Q) + 如果TS(Ti)<R-timestamp(Q)，则Ti产生的Q值是先前所需要的值，且系统已假定该值不会被产生。因此，write操作被拒绝，Ti回滚。+ 如果TS(Ti)<W-timestamp(Q)，则T1试图写入的值已过时，因此，忽略这个写操作 + 否则，执行write操作，将W- timestamp(Q)设为TS（Ti）

注：Thomas写规则尽量减少数据被反复修改，注重保护当前有效的数据。因此在第二种情况下，它会选择忽略这个老的写操作。这样做减少了回滚。Thomas写规则通过删除事务发出的过时的write操作产生视图等价于串行调度。

12.3.3基于有效性检查的协议

经过前面的介绍，大家会发现只读事务的并发性很好。在大部分事务是只读事务的情况下，事务发生冲突的频率较低。并且我们想要一种开销尽可能小的并发控制协议，减少开销面临的困难是我们事先不知道哪些事务将陷入冲突中。为了获得这些知识，需要一种监控系统的机制。**划分事务阶段** 每个事务Ti在其生存期中按两个或三个阶段执行：+ 读阶段：各数据项值被读入，并保存在事物Ti的局部变量中。+ 有效性检查阶段：判断是否可以将write操作所更新的临时局部变量值复制到数据库而不违反可串行性。+ 写阶段：若事务Ti已经通过有效性检查，进行实际的数据库更新，否则，回滚。

按照阶段设置时间戳 + Start(Ti)：事务Ti开始执行的时间。+ Validation(Ti)：事务Ti完成读阶段并开始其有效性检查阶段的时间。+ Finish(Ti)：事务Ti完成写阶段的时间。

有效性检查协议 + 利用时间戳Validation(Ti)的值，通过时间戳排序技术决定可串行化顺序。 TS(Ti)=Validation(Ti) + 事务完成读之后即更改其时间戳的值。+ 之所以选择Validation(Ti)的值作为事务Ti的时间戳，而不使用Start(Ti)，是为了在冲突频度低的情况下，可以拥有更快的响应时间。+ 事务Tj的有效性测试要求任何满足TS(Ti) < TS(Tj)的事务Ti必须满足下列条件之一：+ Finish(Ti)< Start(Tj) + Ti所写的的数据项集与Tj所读数据项集不相交，并且Ti的写阶段在Tj开始其有效性。检查阶段之前完成(start(Tj)<finish(Ti)<validation(Tj))，此条件保证Ti和Tj的写不重叠。

12.3.4死锁处理

在基于锁的协议里曾经提到过死锁的问题。**死锁预防** 预防死锁的发生就是要破坏产生死锁的条件。+ 一次封锁法 + 概述：要求每个事务必须一次将所有要使用的数据全部加锁，否则就不能继续执行。+ 问题：降低并发度 将以后要用到的全部数据加锁，势必扩大了封锁的范围，从而降低了系统的并发度。+ 顺序封锁法 + 概述：顺序封锁法是预先对数据对象规定一个封锁顺序，所有事务都按这个顺序实行封锁。+ 问题：维护成本高 数据库系统中可封锁的数据对象极其众多，并且随数据的插入、删除等操作而不断地变化，要维护这样极多而且变化的资源的封锁顺序非常困难。+ 抢占与事务回滚 + 在抢占机制中，当事务Ti所申请的锁被事务Tj所持有时，授予Tj的锁可能通过回滚事务Tj被抢占，并将锁授予Ti。+ 通过时间戳确定事务等待还是回滚，事务重启时，保持原有的时间戳。注：为何保持原有时间戳？ 因为一个被反复回滚的事务很可能处于饥饿状态，让它保持原有时间戳相当于一种老化机制。早产生的事务在不断回滚时一定会成为最“老”的那个，此时它在抢占锁时一定有最高的优先级，被除了饥饿。+ 两种技术：+ Wait-die(非抢占技术)：当事务Ti申请的数据项当前被事务Tj持有时，仅当Ti的时间戳小于Tj的时间戳时，允许Ti等待，否则Ti回滚。+ Wound-die(抢占技术)：当事务Ti申请的数据项当前被事务Tj持有时，仅当Ti的时间戳大于Tj的时间戳时，允许Ti等待，否则Tj回滚。

注：+ 上述两种机制均避免“饿死”：任何时候均存在一个时间戳最小的事务。在这两种机制中，这个事务都不允许回滚。由于时间戳总是增长，并且回滚的事务不被赋予新的时间戳，被回滚的事务最终变成最小时间戳事务，从而不会再次回滚。+ 二者的共同问题是：发生不必要的回滚

死锁的诊断与解除 + 概述 + 在操作系统中广为采用的预防死锁的策略并不很适合数据库的特点。DBMS在解决死锁的问题上更普遍采用的是诊断并解除死锁的方法。+ 由DBMS的并发控制子系统定期检测系统中是否存在死锁。一旦检测到死锁，就要设法解除。+ 检测死锁 + 超时法 + 判断方法：如果一个事务的等待时间超过了规定的时限，就认为发生了死锁。+ 优点：实现简单。+ 缺点：时限若设置得太短，有可能误判死锁；时限若设置得太长，死锁发生后不能及时发现。+ 等待图法 用事务等待图动态反映所有事务的等待情况，并发控制子系统周期性地（比如每隔1 min）检测事务等待图，如果发现图中存在回路，则表示系统中出现了死锁。注：事务等待图是一个有向图G=(V, E) + V为结点的集合，每个结点表示正运行的事务 + E为边的集合，每条边表示事务等待的情况 + 若Ti等待Tj，则Ti, Tj之间划一条有向边，从Ti指向Tj + 事务Tj不再持有事务Ti所需要的数据项时，边从等待图中删除 + 解除死锁 选择牺牲者，回滚事务。

12.4 恢复系统

计算机系统可能在实际应用的过程中出现各种各样的问题，一旦有故障发生就可能破坏数据。因此恢复系统就是要保证，即使发生故障也可以保持事物的原子性和持久性。

12.4.1故障分类

- 事务故障：逻辑故障&系统错误
- 系统崩溃
- 磁盘故障

12.4.2恢复算法（养兵千日，用兵一时）

- 在正常事务处理时采取措施，保证有足够的信息用于故障恢复。
- 故障发生后采取措施，将数据库内容恢复到某个保证数据库一致性、事务原子性及持久性的状态。

12.4.3数据备份

理论上不可能得到稳定存储器，可以通过技术手段使数据极不可能丢失。+ RAID（独立冗余磁盘阵列，Redundant Array of Independent Disk）+ 归档备份保存至磁带

12.4.4基于日志的恢复

系统日志 + 日志是日志记录的序列,记录数据库中所有的更新活动。+ 先写日志，后写数据库。+ 日志的组成：事务标识符、数据项标识符、旧值、新值 下面是几种常见的日志：； <Ti, Xj, V1,V2>; <Ti, commit>; <Ti, abort>

延迟的数据库修改的恢复机制 + 延迟的数据库修改 事务中所有的write操作，在事务部分提交时才修改数据库的执行，日志中只记录新值。
在上图所示的示例中，T0和T1只有commit时才对数据库执行了A、B、C的写操作。+ 恢复机制 + 基础操作 Redo(Ti): 将事务Ti更新的所有数据项的值设为新值。+ 操作原则 事务Ti需要Redo操作，当且仅当日志中既包含记录<Ti, start>又包含记录<Ti, commit>。注：当且仅当，事务完整提交后，数据库中的数据才被修改了。因此必须有commit标志的事务才可以被redo。

立即的数据库修改的恢复机制 + 立即的数据库修改 立即的数据库修改：允许数据库修改在事务处于活动状态时就输出到数据库中。
在上图所示的示例中，T0和T1在执行对应的write()时，就在数据库中将A、B、C重写了。+ 恢复机制 + 基础操作 + Undo(Ti)：将事务Ti所有更新的所有数据项的值恢复成旧值。+ Redo(Ti)：将事务Ti所有更新的所有数据项的值置为新值。+ 操作原则 + 事务Ti需要Redo操作，当且仅当日志中既包含记录<Ti, start>又包含记录<Ti, commit> + 事务Ti需要Undo操作，当且仅当日志中既包含记录<Ti, start>不包含记录<Ti, commit> 注：只有commit的事务才是有效的。因此即使事务修改了数据库，但是它没有commit，它新写的数据也不能具有持久性。+ 优化 系统发生故障时，检查日志，决定哪些事务需要Redo，哪些事务需要Undo，原则上需要搜索整个日志。但是搜索过程太耗时，并且大多数需要Redo的事务已经写入了数据库，此时Redo不会产生不良后果，但是会使得恢复过程太长。+ 检查点 由系统周期性地执行检查点，需要执行下列操作：+ 将当前位于主存的所有日志记录输出到稳定存储器上。+ 将所有修改了的缓冲块输出到磁盘上。+ 将一个日志记录输出到稳定存储器。+ 检查点执行过程中，不允许事务执行更新操作。+ 优化操作 + 基本原则：+ 在检查点之前提交的事务，不予考虑。记录<Ti,commit>在日志中，出现在之前，这表示系统故障前Ti已经提交，Ti的操作有效。+ 确定最近的检查点发生前开始执行的最近的一个事务Ti，对于Ti和Ti之后的开始执行的事务Tj执行redo和undo操作。+ 具体过程 + 系统由后向前扫描日志，直至发现第一个：Redo-list: 对每一个形如的记录，将Ti加入Redo-list Undo-list: 对每一个形如的记录，如果Ti不属于Redo-list，将Ti加入undo-list + Redo-list和undo-list构造完毕后：从最后一个记录开始由后至前从新扫描日志，并且对undo-list中的每一个日志记录执行Undo操作。忽略redo-list中的事务。找到最近一条记录。系统由最近一条记录由前向后扫描日志，并且对redo-list中事务Ti的每一个日志记录执行redo操作。注：从后向前undo，从前往后redo。因为undo是还原数据的操作，应该从最后一个无效数据逐步还原回最近记录的有效数据。而redo是重写操作，有可能多个事务对同一数据单元进行过更新，应该按照这些事务写的顺序来重写数据，否则会出现最后有效的数据反而是原先“最老”的事务提交的结果。