

数据库系统



谷一滕

林子童

徐卫霞

杜泽林

张晓敏



吉鹏智库-让知识回归平凡

目录

| | | | |
|-----------------------------------|----|--------------------------------|----|
| 第一章 引言 | | 7.7 实体-联系设计问题 | 37 |
| 1.1 四个基本概念 | 3 | 7.8 扩展 E-R 特性 | 37 |
| 第二章 关系模型介绍 | | 第八章 关系数据库设计 | |
| 2.1 关系数据库的结构 | 4 | 8.1 好的关系设计的特点 | 41 |
| 2.2 数据库模式 | 5 | 8.2 原子域和第一范式 | 41 |
| 2.3 码 | 5 | 8.3 使用函数依赖进行分解 | 41 |
| 2.4 数据库完整性 | 6 | 8.4 函数依赖理论 | 41 |
| 2.5 关系数据语言概述 | 6 | 8.5 分解算法 | 45 |
| 第三章 SQL 结构化查询语言 | | 8.6 多值依赖 | 46 |
| 3.1 数据库客户端操作 | 7 | 第十章 数据存储和数据存取 | |
| 3.2 数据基本类型 | 7 | 10.1 物理存储介质 | 49 |
| 3.3 关于表格的操作 | 7 | 10.2 文件组织和记录组织 | 49 |
| 3.4 表格数据查询操作 | 9 | 10.3 数据字典存储 | 51 |
| 3.5 集合运算 | 10 | 10.4 数据缓冲区 | 51 |
| 3.6 聚集函数 | 11 | 10.5 索引基本概念 | 53 |
| 3.7 其他 | 12 | 10.6 顺序索引 | 53 |
| 3.8 例题 | 14 | 10.7 B+树索引文件 | 55 |
| 第四章 中级 SQL | | 10.8 散列文件组织和散列索引 | 57 |
| 4.1 连接表达式 | 16 | 第十一章 查询处理和查询优化 | |
| 4.2 视图 | 18 | 11.1 查询处理的基本步骤 | 60 |
| 4.3 事务 | 19 | 11.2 查询代价的度量 | 60 |
| 4.4 完整性约束 | 19 | 11.3 两种关系代数运算的执行 | 61 |
| 4.5 SQL 的数据类型与模式 | 20 | 11.4 两种表达式计算方法 | 65 |
| 4.6 授权 | 21 | 11.5 查询优化 | 66 |
| 第六章 形式化关系查询语言 | | 第十二章 事务 | |
| 6.1 关系代数 | 24 | 12.1 事务的基本概念 | 71 |
| 6.2 元组关系演算 | 29 | 12.2 事务调度 | 72 |
| 第七章 数据库设计和 E-R 模型 | | 12.3 并发控制 | 78 |
| 7.2 实体-联系模型 | 33 | 12.4 恢复系统 | 84 |
| 7.3 约束 | 34 | 吉鹏智库 | |
| 7.5 实体-联系图 | 35 | | |
| 7.6 转换为关系模式 | 35 | | |

第一章 引言

作者：谷一滕

前言：本章主要内容是将整本书所学知识进行了高度概括，因此截取 PPT 部分内容进行描述。

1.1 四个基本概念

1.1.1 数据(Data)

数据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：数值、字符、图形、图像、声音、日期

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)

数据库管理系统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 的目标：科学地组织和存储数据、高效方便地获取和维护数据

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构成。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：数据库管理系统（及其开发工具）、应用系统、数据库管理员（和用户）

第二章 关系模型介绍

作者：谷一滕

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

几个概念：

2.1.1 关系(relation)：代指表

1. 当关系作为关系数据库的数据结构时，需要对其进行限定，使其满足：

无限关系在数据库中是无意义的

通过为关系的每个列附加一个属性名的方法取消关系属性的有序性，即 $(d_1, d_2, \dots, d_i, d_j, \dots, d_n) = (d_1, d_2, \dots, d_j, d_i, \dots, d_n)$

2. 关系的性质：

列是同质的

不同的列可来自同一域，每列必须有不同的属性名

列的次序可以任意交换

任意两个元组不能完全相同

每一分量必须是不可再分的数据。满足这一条件的关系称作满足第一范式(1NF)的

2.1.2 元组(tuple)：代指行

2.1.3 属性(attribute)：代指列

2.1.4 关系实例(relation instance)：一个关系的特定实例

2.1.5 域(domain)：属性的取值集合，是原子的（取决于怎么使用域中元素）

2.1.6 空值(null)：表示值未知或不存在

空值的表现

参与算术运算：结果为 Null

参与比较运算：结果为 Null

参与逻辑运算：

1、Null or true=true

2、Null and false=false

3、其它情况结果为 null

2.1.7 笛卡尔积(Cartesian Product):

一组域 D_1, D_2, \dots, D_n 的笛卡尔积为：

$$D_1 \times D_2 \times \dots \times D_n = \{(d_1, d_2, \dots, d_n) \mid d_i \in D_i, i=1, \dots, n\}$$

笛卡尔积的每个元素 (d_1, d_2, \dots, d_n) 称作一个 n -元组 (n-tuple)

元组的每一个值 d_i 叫做一个分量 (component)

若 D_i 的基数为 m_i ，则笛卡尔积的基数为 $\prod_{i=1}^n m_i$

2.2 数据库模式

2.2.1 关系模式：关系的描述称作关系模式，包括关系名、关系中的属性名、属性向域的映象、属性间的数据依赖关系等。关系是某一时刻的值，是随时间不断变化的。

2.2.2 关系模型

三类关系：

基本关系（基本表或基表）实际存在的表，是实际存储数据的逻辑表示

查询表 查询结果对应的表

视图表 由基本表或其他视图表导出的表，是虚表，不对应实际存储的数据

2.2.3 老师的例子模型

D (dno, dname, dean)

S (sno, sname, sex, age, dno)

C (cno, cname, credit)

SC(sno, cno, score)

T (tno, tname, dno, sal)

TC(tno, cno)

要熟记于心，是老师讲课常用到的的基本模型。

2.3 码

2.3.1 超码(superkey)：是一个或多个属性的集合，这些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

2.3.2 候选码(Candidate Key)：任意真子集都不能够成为超码的超码，是最小的超码

2.3.3 主码(Primary Key)：进行数据库设计时，从一个关系的多个候选码中选定一个作为主码（应选择从不变化或极少变化的属性）

2.3.4 外部码(Foreign Key)：关系 R 中的一个属性组，它不是 R 的主码，但它与关系 S 的主码相对应，则称这个属性组为 R

的外部码(R 和 S 可以是同一关系)。关系 R 称为外码依赖的参照关系, 关系 S 叫做外码的被参照关系。

2.4 数据库完整性

定义：是指数据库中数据的正确性和相容性, 由各种各样的完整性约束来保证, 是数据库设计的重要组成部分, 完整性约束可以通过 DBMS 或应用程序来实现

2.4.1 关系模型完整性

实体完整性和参照完整性：关系模型必须满足的完整性约束条件, 称为关系的两个不变性, 由关系系统自动支持

用户定义的完整性：应用领域需要遵循的约束条件, 体现了具体领域中的语义约束, 用户定义后, 由关系系统自动支持

实体完整性(Entity Integrity)：关系的主码中的所有属性值不能为空值

参照完整性(Referential Integrity)：如果关系 R 的外部码 Fk 与关系 S 的主码 Pk 相对应(R 和 S 可以是同一个关系), 则 R 中的每一个元组的 Fk 值或者等于 S 中某个元组的 Pk 值, 或者为空值

用户定义的完整性(User-defined Integrity)：用户针对具体的应用环境定义的完整性约束条件

2.5 关系数据语言概述

2.5.1 关系数据语言的特点：一体化, 非过程化, 面相集合的存取方式

2.5.2 抽象的查询语言：关系代数, 关系演算

2.5.3 具体系统中的实际语言：SQL, QUEL, QBE

```
(sno CHAR (4) ,
cno CHAR (4) ,
score SAMLINT,
constraint pk_sc PRIMARY KEY (sno, cno),
constraint fk_scs FOREIGN KEY (sno)REFERENCES S(sno),
constraint fk_scc FOREIGN KEY (cno)REFERENCES C(cno),
CHECK((score IS NULL) OR score BETWEEN 0 AND 100))
```

常用完整性约束：

主码约束：PRIMARY KEY
 唯一性约束：UNIQUE
 非空值约束：NOT NULL
 参照完整性约束：FOREIGN KEY

删除表格 drop table 表名

注意：撤消基本表后，基本表的定义、表中数据、索引都被删除，由此表导出的视图将无法继续使用

插入语句 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，但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...)
```

示例：

```
insert into T (tno, tname, dno)
values ('t123','王明','d08', )
```

为了防止插入带有空值的元组，建议写列名

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

```
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

查询数据 select */属性名 from 表名

*的意思是全部的属性

删除数据 delete from 表名 [where...]**示例：**

删除王明老师所有的任课记录

```
delete from tc
where tno in
(select tno
from t
where tname = '王明')
```

注：在实际操作中不建议删除数据，因为会产生很多负面后果，而建议用一个属性来代表相关数据的状态，如有效，停用，取消，弃置等。

列数据修改：

| | |
|-------|---|
| 增加列 | <code>alter table 表名 add 属性名 类型(数据长度)</code> |
| 修改列定义 | <code>alter table 表名 modify 属性名 类型(数据长度)</code> |
| 删除列 | <code>alter table 表名 drop 属性名 [cascade/restrict]</code> |
| 删除约束 | <code>alter table 表名 drop constraint 约束名</code> |

此处 CASCADE 方式表示：在基本表中删除某列时，所有引用到该列的视图和约束也要一起自动地被删除。而 RESTRICT 方式表示在没有视图或约束引用该属性时，才能在基本表中删除该列，否则拒绝删除操作。

示例：

```

增加列          alter table T add location char (30) ;
修改基本表定义 alter table S modify sname varchar2 (30) ;
                (注意：修改列的长度时，只能改长，不能改短)
直接删除属性列 alter table B drop scome cascade ;
删除表中的约束 alter table S drop constraint pk_s

```

更新数据 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：

```

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

```

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

4. 表格数据查询操作

查询的基本结构 select,from,where

select 子句：指定要显示的属性列

目标列形式：可以为列名，*，算术表达式，聚集函数带+-*\的算术表达式

from 子句：指定查询对象(基本表或视图)，当目标列取自多个表时，在不混淆的情况下 可以不用显式指明来自哪个关系。相当于表的笛卡尔积，不是自然连接

where 子句：指定查询条件

去重关键字 select distinct...

多关系查询 属性名位于多关系时，加上前缀。写为：表名.属性名

| | |
|--------------|---|
| | 如：student.sno |
| 自然连接 | 表名 1 natural join 表名 2 |
| 构造自然连接 | 表名 1 join 表名 2 using (属性名 1, 属性名 2...) 意为，只要特定属性相等，就可以匹配 |
| 更名运算 (名) | select/where 旧表名 新表名 (注：SQL 中为：旧表名 as 新表名) |
| 所有属性 | * |
| 修改显示顺序 | order by 属性名 1 desc (降序)/ asc (升序)... |
| | (在查询的最后使用，默认升序) |
| | 当排序列含空值时： |
| | ASC：排序列为空值的元组最后显示 |
| | DESC：排序列为空值的元组最先显示 |
| 和，或，非 | and, or, not |
| 位于...和...中间 | where 属性名 between 值 1 and 值 2 |
| 不位于...和...中间 | where 属性名 not between 值 1 and 值 2 |
| 字典序多比较 | where (属性名 1, 属性名 2) 比较运算符 (属性名 1, 属性名 2) |
| 如： | where (s.sno,s.age) = (sc.sno,15) |

5. 集合运算

并，交，差(要求可包容) **union, intersect, except** (Oracle 中为 **minus**)

(每个集合都要用括号包裹，**except** 除外)

注意：集合操作自动去除重复元组，如果要保留重复元组的话，必须用 **all** 关键词

指明

示例：

查询选修了 001 或 002 号课程的学生号

```
(select sno
from SC
where cno = '001')
union
(select sno
from SC
where cno = '002');
```

查询选修了 001 和 002 号课程的学生号

```
(select sno
from SC
where cno = '001')
intersect
(select sno
from SC
where cno = '002');
```

查询选修了 001 而没有选 002 号课程的学生号

```
(select sno
from SC
where cno= '001')
except
```

```
(select sno
from SC
where cno = '002')
```

6. 聚集函数

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

它们默认是带 **all** 关键字（意味着保留重复元组）

| | |
|-----|------------|
| 平均值 | avg(属性名) |
| 最小值 | min(属性名) |
| 最大值 | max(属性名) |
| 总和 | sum(属性名) |
| 计数 | count(属性名) |

示例：

查询学生总人数

```
SELECT count (*)
FROM S;
```

查询选修了课程的学生人数

```
SELECT count(distinct sno)
FROM SC;
```

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；

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** 是在分组之后进行的选择运算

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

```
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

```
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]]
```

7. 其他

空值

空值测试：**is [not] null**

示例：

找出年龄值为空的学生姓名

```
select sname
from S
where age is null ;
```

注意事项

除 **is [not] null** 之外，空值不满足任何查找条件

如果 **null** 参与算术运算，则该算术表达式的值为 **null**

如果 **null** 参与比较运算，则结果可视为 **false**。

如果 **null** 参与聚集运算，则除 **count(*)** 之外其它聚集函数都忽略 **null**

嵌套子查询 **where 属性名 in/not in(select 属性名 子查询...)**

注：可以对应多个属性，此时用括号()将多个属性包裹，且多属性为一一对应，如：

where (属性名 1...) in/not in (select 属性名 1... 子查询...)

from 中也可以嵌套子查询：**from (子查询...)**

示例：

应用于枚举的场合

```
select sno
from s
where name in ('张三', '李四');
```

查询哪些学生没有选修哪些课程

```
select sno,cno
from s,c
where (sno,cno) not in
(select sno,cno
from sc);
```

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

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

示例：

查询其他系中比 **d1** 系中，指定学生年龄小的学生，列出其姓名和年龄

```
select sname, age
from S
where dno <> 'd1' and
age < some
(select age
from S
where dno= ' d1')
```

SOME 与 ALL 与聚集函数的对应关系

| | | | | | | |
|------|----|--------|------|--------|------|--------|
| | = | <>或!= | < | <= | > | >= |
| SOME | IN | -- | <MAX | <=MAX | >MIN | >= MIN |
| ALL | -- | NOT IN | <MIN | <= MIN | >MAX | >= MAX |

空关系测试 where **exists/not exists**(子查询...)

含义：测试子查询的结果中是否存在元组，对于 **exists**，若存在元组则返回 **true**

由 **EXISTS** 引出的子查询，传统意义认为其目标列表达式通常都用*，因为带 **EXISTS** 的子查询只关注是否有元组，给出列名无实际意义。但是*会带来查询性能问题，建议使用常数（如数字 1）。

示例：

exists 与 in 的区别

查询选修了 C1 号课程的学生的学号及姓名

```
select sno, sname
from S
where exists
(select *
from SC
where cno = 'c1'
and sno = s.sno)
```

```
select sno,sname
from S
where sno in
(select sno
from SC
where cno = 'c1')
```

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。

关于示例的分析：

本查询涉及 **S** 和 **SC** 关系。

在 **S** 中依次取每个元组的 **sno** 值，用此值去检查 **SC** 关系。

若 **SC** 中存在这样的元组，其 **sno** 值等于此 **S.sno** 值，并且其 **cno**= 'c1'，则取此 **S.sname** 送入结果关系。

示例 (not exists)：

查询没有选修 C1 号课程的学生姓名

```
select sname
from S
where not exists
(select *
from SC
where sno = s.sno and cno = 'c1')
```

重复元组存在性测试 where **unique/not unique**(子查询...)

含义：测试子查询的结果中是否存在重复元组，对于 **unique**，若不存在相同元组，返回 **true**

示例：

找出所有只教授一门课程的老师姓名

```
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
select sname
from s
where not unique
      (select sno
       from sc
       where sc.sno = s.sno)

```

with 子句的使用

含义：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，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

```

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

字符串运算

% 匹配任意子串

_ 匹配任意一个字符

\ 转义字符，去掉特殊字符的特定含义，使其作为普通字符来看

例：列出姓张的教师的的所有信息

```

select *
from T
where tname like '张%'

```

8. 例题：

(1) “全部”的处理：

查询至少选修了学生002选修的全部课程的学生号码。

解题思路：

- 用逻辑蕴涵表达：查询学号为x的学生，对所有的课程y，只要002学生选修了课程y，则x也选修了y。

- 形式化表示：

用P表示谓词 “学生002选修了课程y”

用Q表示谓词 “学生x选修了课程y”

则上述查询为： $(\forall y) p \Rightarrow q$

等价变换：

$$\begin{aligned}
 (\forall y)p \Rightarrow q &\equiv \neg (\exists y (\neg (p \Rightarrow q))) \\
 &\equiv \neg (\exists y (\neg (\neg p \vee q))) \\
 &\equiv \neg \exists y (p \wedge \neg q)
 \end{aligned}$$

变换后语义：不存在这样的课程y，学生002选修了y，而学生X没有选。

此例题有三种解决方式：

①

用NOT EXISTS谓词表示：

```

SELECT DISTINCT sno
FROM sc scx
WHERE NOT EXISTS
  (SELECT 1
   FROM sc scy
   WHERE scy.sno = '002' AND
        NOT EXISTS
          (SELECT 1
           FROM sc scz
           WHERE scz.sno=scx.sno AND
                scz.cno=scy.cno))
  
```

②

用超集表示：

```

SELECT DISTINCT sno
FROM sc scx
WHERE NOT EXISTS
  (select cno
   from sc scy
   where scy.sno='002')
except
(select cno
 from sc scz
 where scx.sno=scz.sno)
  
```

③

- 使用基本关系代数运算，写出选修了002学生选修的全部课程的学生学号。

$$\Pi_{sno,cno}(sc) \div \Pi_{cno}(\sigma_{sno='002'}(sc))$$



$$\Pi_{sno}(s) - \Pi_{sno}(\Pi_{sno}(s) \times \Pi_{cno}(\sigma_{sno='002'}(sc)) - \Pi_{sno,cno}(sc))$$

即：

(2) exists :

— 查询同时选修了001号和002号课程的学生学号

```

select  sno
from    sc sc1
where   sc1.cno = '001'
and exists
  (select  1
   from    sc sc2
   where   sc2.cno = '002'
          and sc2.sno = sc1.sno)
  
```

第四章 中级 SQL

作者：谷一滕

4.1 连接表达式

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。

4.1.2 外连接

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：

| <i>R</i> | | | <i>S</i> | |
|----------|----|----|----------|----|
| A | B | C | C | D |
| a1 | b1 | c1 | c1 | d1 |
| a2 | b2 | c2 | c2 | d2 |
| a3 | b3 | c3 | c4 | d3 |

内连接：

R inner join S on R.C = S.C

| A | B | C | C | D |
|----|----|----|----|----|
| a1 | b1 | c1 | c1 | d1 |
| a2 | b2 | c2 | c2 | d2 |

左外连接：

R left outer join S on R.C = S.C

| A | B | C | C | D |
|----|----|----|------|------|
| a1 | b1 | c1 | c1 | d1 |
| a2 | b2 | c2 | c2 | d2 |
| a3 | b3 | c3 | null | null |

右外连接：

R nature right outer join S

| A | B | C | D |
|------|------|----|----|
| a1 | b1 | c1 | d1 |
| a2 | b2 | c2 | d2 |
| null | null | c4 | d3 |

全外连接：

R full outer join S on R.C = S.C

| A | B | R.C | S.C | D |
|------|------|------|------|------|
| a1 | b1 | c1 | c1 | d1 |
| a2 | b2 | c2 | c2 | d2 |
| a3 | b3 | c3 | null | null |
| null | null | null | c4 | d3 |

查询每个学生及其选修课程的情况包括没有选修课程的学生----用外连接操作

```
select S.sno, sname, sex, age, dno, cno, score
from S, SC
where S.sno = SC.sno(*);
```

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

| 连接类型 | 连接条件 |
|--|--|
| inner join left outer join right outer join full outer join | nature on <谓词> using (A ₁ , A ₂ ,..., A _n) |

4.2 视图

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

注意：

视图的属性名缺省为子查询结果中的属性名，也可以显式指明，在下列情况下，必须指明视图的所有列名：

某个目标列是**聚集函数**或者目标列**表达式**

多表连接时，选出了几个**同名列**作为视图的字段

需要在视图中为某个列启用**新的更合适的名字**

目标列是*

删除视图：drop view view_name

视图特点：视图不会要求分配存储空间，视图中也不会包含实际的数据。视图只是定义了一个查询，视图中的数据是从基表中获取，这些数据在视图被引用时动态的生成。

视图作用：

当视图中数据不是直接来自基本表时，定义视图能够简化用户的操作

视图能使不同用户以不同方式看待同一数据，适应数据库共享的需要

对不同用户定义不同视图，使每个用户只能看到他有权看到的数据

视图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

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。

1.from 子句中只有一个数据库关系

2.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

3.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）

4.查询中不含有 group by 或 having 子句

物化视图：

即实体化视图，它确实存放有物理数据。物化视图包含定义视图

的查询时所选择的基表中的行。对物化视图的查询就是直接从该视图中取出行。

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。

4.3 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

完整性定义：

数据的正确性和相容性

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

完整性检查：

DBMS 必须提供一种机制来检查数据库中的数据是否满足规定的条件，以保证数据库中数据是正确的。

4.4.1 单个关系上的约束

not null

Primary key

Unique

Check(P), P 是一个谓词

删除基本关系元组约束 (RESTRICT、CASCADE、SET NULL 方式)

参照三章内容

4.4.2 约束的操作

命名约束：CONSTRAINT 约束名 <约束条件>

示例：sno CHAR(4) CONSTRAINT S_PK PRIMARY KEY

age SMALLINT CONSTRAINT AGE_VAL

CHECK(age >= 15 AND age <= 25)

约束撤销：alter ...drop...

示例：alter table S drop constraint S_PK

约束添加：alter ...add...

示例：alter table SC add constraint SC_CHECK

check(sno in select sno from S)

4.4.3 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被暂时违反，但是后面的某一步也许就会消除这个违反，其实可以通过推迟完整性检查（不在事务的中间步骤上检查，而是在事务结束的时候检查）来达到目的。

4.5 SQL 的数据类型与模式

4.5.1 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

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`

两个 `date/time/timestamp` 类型值相减产生一个 `interval` 类型值

可以在 `date/time/timestamp` 类型的值上加减 `interval` 类型的值

4.5.2 默认值

为属性指定默认值

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，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

Clob (Character Large Object , 字符数据的大对象数据类型)

Blob (Binary Large Object 二进制数据的大对象数据类型)

典型的 BLOB 是一张图片或一个声音文件

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

LOB 存储实现：指针 + 文件

LOB 访问：一般使用专用语句访问

```
Oracle:
    SelectBlob doc into ...
    from book
    where cno='c1';
```

4.5.5 用户定义类型

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]

删除或修改：drop type alter type

4.5.6 域定义

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：

1.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。

2.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。

4.6 授权

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}
[with grant option]

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
```

转移权限：grant 权限 on 表名 to 用户名 with grant option

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
```

4.6.2 角色

概念：为了指明一类人应有的授权，提出了角色概念。在数据库中建立角色集，并授予每个角色一定的权限，然后将角色分配给用户。

创建角色：`create role` 角色名

授权给角色：`grant` 权限 `on` 表名 `to` 角色名

特点：角色可以授权给用户，也可授予给其他角色

示例：`grant teacher to Mark`

`grant student to teacher`

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 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

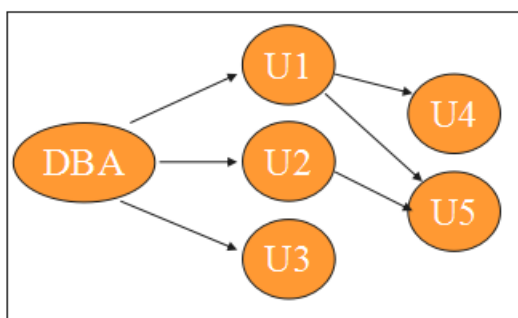
resource：`connect` 权限+建表、删除表及索引权利

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

权限图：结点为用户，根结点是 `DBA`，有向边 $U_i \rightarrow U_j$ ，表示用户 U_i 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在权限图中有一条从根结点到该用户结点的路径



第六章 形式化关系查询语言

作者：林子童

校验：谷一滕

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 σ Π \cup $-$ \times ρ

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 $\Pi \cup$ - 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 ： $\Pi_{Ai}(R)$ 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：

 $\Pi_{sno}(\sigma_{cno = 'c1' \vee cno = 'c2'}(SC))$

方案二：

 $\Pi_{sno}(\sigma_{cno = 'c1'}(SC)) \cup \Pi_{sno}(\sigma_{cno = 'c2'}(SC))$ 集合差(difference) $-$ ： $R - S$, 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 \neq (不等于)

例如：查询未选修 c1 号课程的学生号

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

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

方案二中，首先，没有将未选课的同学包含在内，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 ;

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

$\Pi_{\text{customer.name}}(\sigma_{\text{customer.street}=\text{s_add.street} \wedge \text{customer.city}=\text{s_add.city}}(\text{customer} \times \rho_{\text{s_add}(\text{street,city})}(\Pi_{\text{street,city}}(\sigma_{\text{name}='smith'}(\text{customer}))))))$

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： $\Pi_{\text{pid,name}}(S \cup T) \equiv \Pi_{\text{pid,name}}(S) \cup \Pi_{\text{pid,name}}(T)$

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

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：

$E_1 \cup E_2$
 $E_1 - E_2$
 $E_1 \times E_2$
 $\sigma_p(E_1)$
 $\Pi_p(E_1)$
 $\rho_X(E_1)$

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 $\cap \theta \bowtie \div \leftarrow$

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

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

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

$$R \bowtie S = \sigma_{r[A] \theta s[B]}(R \times S)$$

$$A \theta B$$

果

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 Y_x 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

| A | B | C |
|----|----|----|
| a1 | b1 | c2 |
| a2 | b3 | c7 |
| a3 | b4 | c6 |
| a1 | b2 | c3 |
| a4 | b6 | c6 |
| a2 | b2 | c3 |
| a1 | b2 | c1 |

a1的象集 $\{(b1,c2),(b2,c3),(b2,c1)\}$

a2的象集 $\{(b3,c7),(b2,c3)\}$

a3的象集 $\{(b4,c6)\}$

a4的象集 $\{(b6,c6)\}$

$R(X,Y) \div S(Y)$ ：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。

$$R(X,Y) \div S(Y) = \Pi_X(R) - \Pi_X(\Pi_X(R) \times \Pi_Y(S) - R)$$

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

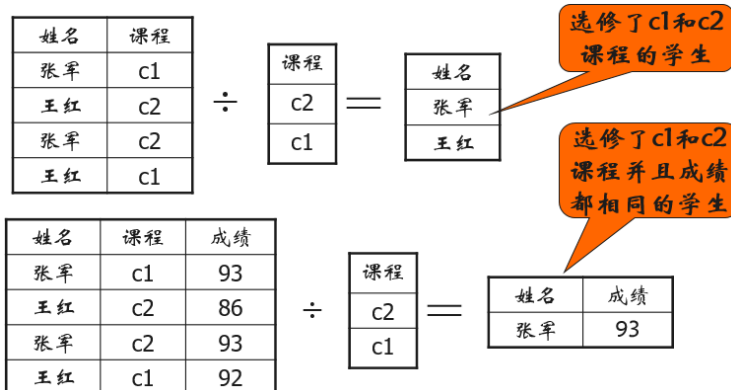
方案 1：

$$\Pi_{sno, cno}(SC) \div \Pi_{cno}(\sigma_{cno = 'c1' \vee cno = 'c2'}(C)) \vee$$

方案 2：

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

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 ← 关系代数表达式

示例：

$$R \div S = \Pi_X(R) - \Pi_X(\Pi_X(R) \times \Pi_Y(S) - R)$$

用赋值重写为：

$$\text{temp1} \leftarrow \Pi_X(R)$$

$$\text{temp2} \leftarrow \Pi_X(\text{temp1} \times \Pi_Y(S) - R)$$

$$\text{result} \leftarrow \text{temp1} - \text{temp2}$$

6.1.4 扩展关系代数

广义投影(generalized-projection)：

$\Pi_{F1, F2, \dots, Fn}(E)$ 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：

$$\Pi_{tno, sal*5/100 \text{ as income-tax}}(T) \text{ 和 } \rho_{TAX}(tno, INCOME-TAX)(\Pi_{tno, sal*5/100}(T))$$

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(\emptyset)=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

Gcount_distinct(sno) (S)

示例: 求全体教工的总工资 $G_{sum}(sal)(T)$

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: $G_1, G_2, \dots, G_n \quad G \quad F_1(A_1), F_2(A_2), \dots, F_m(A_m) \quad (E)$

解释: 将关系 E 按照 G_1, G_2, \dots, G_n 进行分组, 分别计算出每一组的 $F_1(A_1), F_2(A_2), \dots, F_m(A_m)$ 值并且满足:

同一组中所有元组在 G_1, G_2, \dots, G_n 上的值相同

不同组中元组在 G_1, G_2, \dots, G_n 上的值不同

示例: 查询每位学生的总成绩和平均成绩

$sno \quad G_{sum}(SCORE), avg(SCORE) \quad (SC)$

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

\bowtie 左外连接 = 自然连接 + 左侧表中失配的元组

\bowtie 右外连接 = 自然连接 + 右侧表中失配的元组

\bowtie 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:

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

其中 A_1, \dots, A_n 是 R 中的属性

6.1.5 关系代数例题解析:

1. 查询仅选修一门课程的学生学号

$\Pi_{sno}(SC) - \Pi_{a.sno}(\sigma_{a.sno = b.sno \wedge a.cno <> b.cno} (pa(SC) \times pb(SC)))$

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

2. 查询平均成绩高于 s2 平均成绩的学生学号

$\Pi_{sno}(\sigma_{as > bs} (pa(sno, as)(sno G_{avg}(score)(SC)) \times pb(bs)(G_{avg}(score)(\sigma_{sno = 's2'}(SC)))))$

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

3. 查询平均成绩最高的学生学号

$\Pi_{sno}(SC) - \Pi_{A.sno}(\sigma_{aa < ba} (pa(sno, aa)(sno G_{avg}(score)(SC)) \times pb(sno, ba)(sno G_{avg}(score)(SC))))$

记住一件事，avg 返回的是一个值，绝不可以
 $\text{snoGavg}(\text{avg})(\text{pa}(\text{sno}, \text{avg})(\text{snoGavg}(\text{score})))$ ，
 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

作者：徐卫霞

增删：谷一滕

校验：林子童

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 、 \Rightarrow 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 P_1, P_2 是公式，则 $P_1 \wedge P_2, P_1 \vee P_2, P_1 \Rightarrow P_2$ 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： $\exists x \in S(P(x))$

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)

- $P_1 \wedge P_2 \Leftrightarrow \neg (\neg P_1 \vee \neg P_2)$
- $\forall t \in R(P(t)) \Leftrightarrow \neg \exists t \in R(\neg P(t))$
- $P_1 \Rightarrow P_2 \Leftrightarrow \neg P_1 \vee P_2$

6.2.4 元组关系演算与关系代数的等价性

投影: $\Pi_A(R) = \{t \mid \exists s \in R(t[A] = s[A])\}$

选择: $\sigma_{F(A)}(R) = \{t \mid t \in R \wedge F(t[A])\}$

广义笛卡儿积: $R(A) \times S(B) = \{t \mid \exists u \in R \exists s \in S(t[A] = u[A] \wedge t[B] = s[B])\}$

并: $R \cup S = \{t \mid t \in R \vee t \in S\}$

差: $R - S = \{t \mid t \in R \wedge \neg t \in S\}$

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:

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

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 $\{t \mid \neg (t \in R)\}$

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 中出现的所有值的集合

$\text{dom}(t \in R \wedge t[\text{salary}] > 8000)$ 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 $\{t \mid P(t)\}$ 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:

R

| A | B |
|----|----|
| A1 | B1 |
| A1 | B2 |
| A2 | B3 |

dom($\{ t \mid \neg (t \in R) \}$)

| A | B |
|----|----|
| A1 | B3 |
| A2 | B1 |
| A2 | B2 |

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：

R

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

S

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 3 | 4 | 6 |
| 5 | 6 | 9 |

| A | B | C |
|---|---|---|
| 4 | 5 | 6 |
| 7 | 8 | 9 |

 $\{ t \mid t \in R \wedge \neg t \in S \}$

| A | B | C |
|---|---|---|
| 3 | 4 | 6 |
| 5 | 6 | 9 |

 $\{ t \mid t \in S \wedge t[A] > 2 \}$

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 3 | 4 | 6 |

| A | B | C |
|---|---|---|
| 4 | 5 | 6 |
| 7 | 8 | 9 |

$\{t \mid t \in S \wedge \exists u \in R(t[C] < u[B])\}$ $\{t \mid t \in R \wedge \forall u \in S(t[C] > u[A])\}$

| R.B | S.C | R.A |
|-----|-----|-----|
| 5 | 3 | 4 |
| 8 | 3 | 7 |
| 8 | 6 | 7 |
| 8 | 9 | 7 |

$\{t \mid \exists v \in S(\exists u \in R(u[A] > v[B] \wedge t[A] = u[B] \wedge t[B] = v[C] \wedge t[C] = u[A]))\}$

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

$\{t \mid t \in S \wedge t[dno] = 'd1'\}$

• t 的属性通过表达式定义

示例：查询 d1 院系的学生的姓名

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

因为结果集是一元的，关系 S 是五元的，因此不能直接用

“t ∈ S”，需要引入约束变量 s

注意：

• 查询 d1 院系的学生的姓名

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

注：运算符 ∧ 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

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

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：∀ 课程，002 选之 ⇒ 所查询同学选之

$\{t \mid (\exists x \in S(x[sno] = t[sno])) \wedge$

$\forall u \in SC \exists v \in SC(u[sno] = '002' \Rightarrow (u[cno] = v[cno] \wedge t[sno] = v[sno]))\}$

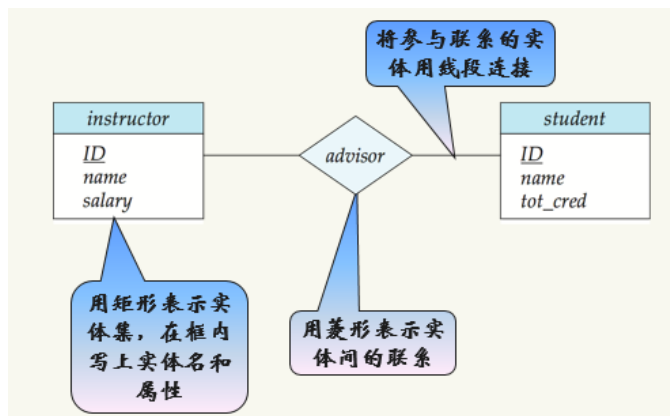
⇒ 经常与 ∀ 一起使用

第七章 数据库设计和 E-R 模型

作者：林子童

增删：谷一滕

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），实体集可相交

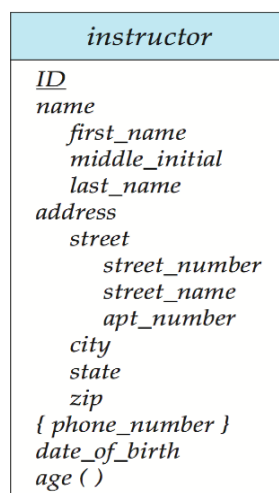
属性 (attribute)：实体集中每个成员具有的描述性性质，是将实体集映射到域的函数，在 E-R 图中如右图所示

注：有阴影的部分包含实体集的名字，无阴影部分包含实体集的所有属性名字

- **简单属性**：不可再分
- 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），一般不存在数据库中，只存储定义或依赖关系，用到时再从基属性中计算出来

基属性：基础属性

域 (domain)：属性的取值范围



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 E_1, E_2, \dots, E_n 为实体集，那么**联系集** R 是：

$\{(e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$ 的一个子集。

而 (e_1, e_2, \dots, e_n) 是一个联系

元或度(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

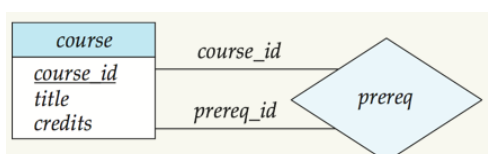
示例：如王军选修“数据库系统”，表示实体“王军”与实体“数据库系统”参与了联系“选修”

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)：指实体在联系中的作用，由于参与一个联系的实体集通常是互异的，角色是隐含的一般不需要指定。但同一个实体集不止一次参与一个联系集时，为区别各实体的参与联系的方式，需要显式指明其角色

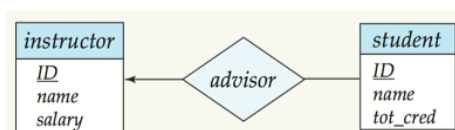
注：在 E-R 图中，通过在菱形和矩形之间的连线上进行标注来表示角色如下图



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



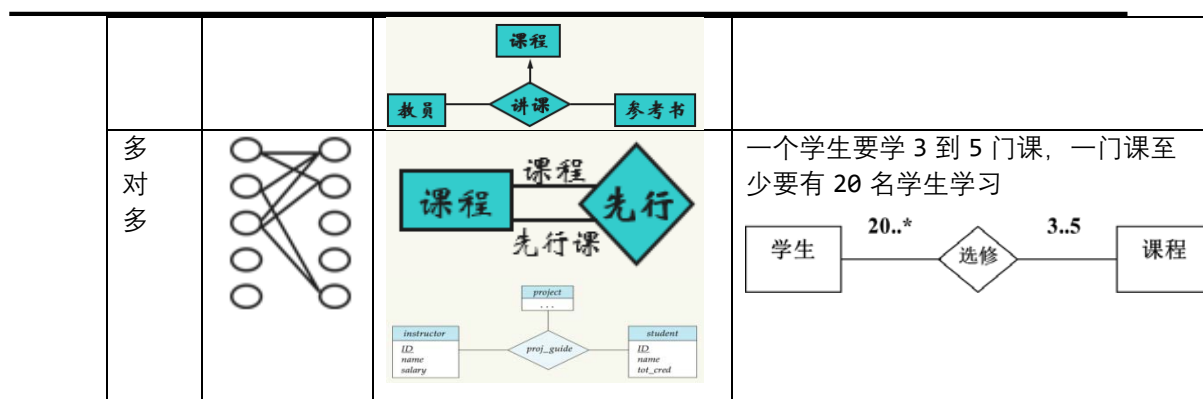
对于 ER 图，有箭头的一方代表一，左图表示一对多，一对一是两边都有箭头，多对多是都没有箭头。

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 $1..h$

$0..*$ 代表“多”， $0..1$ 代表“1”

| 关系 | 示例图 | E-R 图（以一个实体集和多个实体集为例） | 基数约束 |
|-----|-----|-----------------------|------|
| 一对一 | | | |
| 一对多 | | | |



7.3.3 码

一个联系的主码可以由联系的所有实体的主码并上联系的所有属性的集合构成。其中，当实体的属性名重名的时候，用实体名.属性来区分；实体不止一次的参与某个联系的时候，用角色名代替实体名

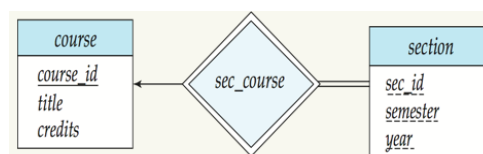
7.5 实体-联系图

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

1) 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

2) 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

1) 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， $3 \rightarrow 2$

示例：

E-R 图如下(假设每个实体都有属性编号和姓名)

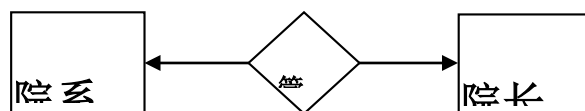
转化成的表：

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)

2) 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， $3 \rightarrow 2$

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)

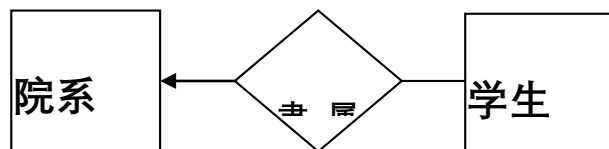
Student(sno, sname)

SD(sno, dno, time)

//dno 非空

表的合并：

Student+SD→Student(sno, sname, dno, time)//dno 可以为空



3) 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

4) 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）

5) 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：

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

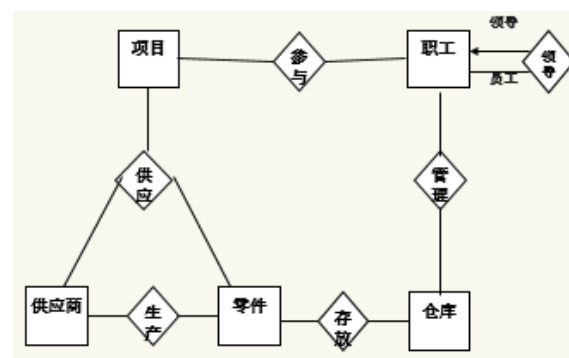
联系转化为表：

```
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

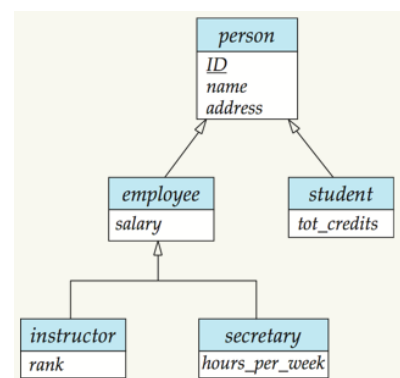
实体集可能包含一些子集，子集中的实体在某些方面区别于实体集中其他实体，这种关系被称作 IS-A 关系。对于属性继承，子类可以继承其超类的属性及超类所参与的实体集，如右图：

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）

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）



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：不可以属于多个

重叠：可以属于多个

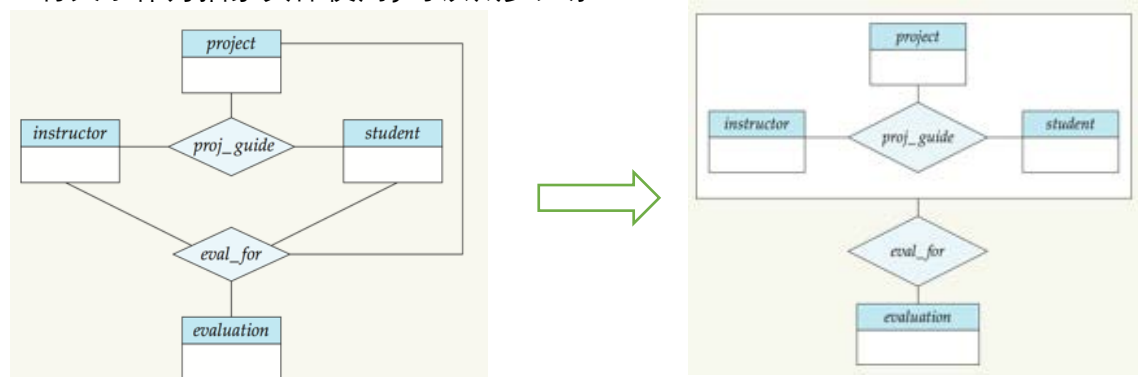
完全性约束：

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



聚集的模式表示：聚集关系的主

码、相关实体集外码和任何描述属性

7.8.6 用模式表示特化

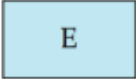


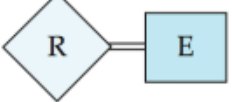
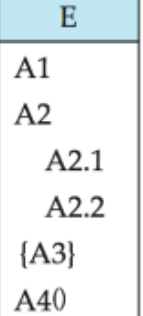
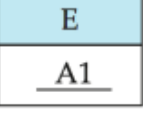
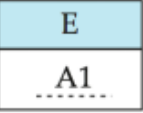


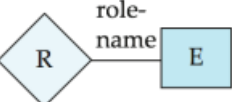
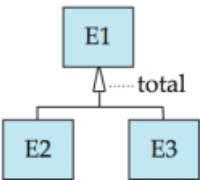
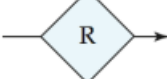
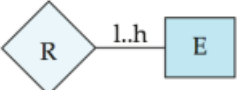
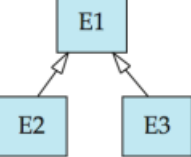
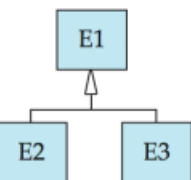
1) 为高层实体集创建一个模式，底层实体集属性中包括自己的特有属性和高层的主码

缺点：获得底层信息需要访问两个关系与低层关系模式有关的关系和与高层关系模式有关的关系

2) 底层实体集的属性对应自己和高层的所有属性

缺点：出现冗余

附：E-R 图表示法中使用的符号

| | | | |
|---|-------------|---|---|
|     | 实体集 |    | 属性 简单属性(A1) 复合属性(A2)和多值属性(A3) 派性属性(A4) |
| | 联系集 | | |
| | 弱实体集的标识性联系集 | | |
| | 实体集在联系中全部参与 | | |
|     | 多对多联系 |     | 主码 弱实体集的分辨属性 多对一联系 基数限制 ISA: 概化或特化 不相交概化 |
| | 一对一联系 | | |
| | 角色标识 | | |
| | 全部(不相交)概化 | | |

第八章 关系数据库设计

作者：林子童，谷一滕

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 $\forall t, s \in r$ ，若 $t[\alpha] = s[\beta]$ ，则 $t[\alpha] = s[\beta]$

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： $\alpha \xrightarrow{f} \beta$

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： $\alpha \xrightarrow{p} \beta$

传递函数依赖： $\alpha \rightarrow \beta, \beta \rightarrow \gamma$ ，且 $\beta \not\subseteq \alpha$ ，则称 γ 对 α 传递函数依赖

8.3.2 码

超码：设 K 为 $R\langle U, F \rangle$ 的属性或属性组，若 $K \rightarrow U$ ，则称 K 为 R 的超码

候选码：设 K 为 $R\langle U, F \rangle$ 的超码，若 U 完全依赖于 K，则称 K 为 R 的候选码

主码：若 $R\langle U, F \rangle$ 有多个候选码，则可以从中选定一个作为 R 的主码

主属性：包含在任一个候选码中的属性，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 α 、 β 、 γ 、 δ 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 $\alpha \rightarrow \beta$ ， $\gamma\beta \rightarrow \delta$ ，则 $\gamma\alpha \rightarrow \delta$

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： $\alpha^+ = \{A \mid \alpha \rightarrow A \text{ 能由 } F \text{ 根据 Armstrong 公理导出}\}$

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 类都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 $F^+ = G^+$ ，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 F_c)

- F_c 与 F 等价
- F_c 中任何函数依赖都不含无关属性
- F_c 中函数依赖的左半部都是唯一的, 即不存在两个依赖 $\alpha_1 \rightarrow \beta_1$, $\alpha_2 \rightarrow \beta_2$ 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 F_c 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 F_m : 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 $R\langle U, F \rangle$ 的一个分解是指

$\rho = \{R_1\langle U_1, F_1 \rangle, R_2\langle U_2, F_2 \rangle, \dots, R_n\langle U_n, F_n \rangle\}$

其中 $U = U_1 \cup U_2 \dots \cup U_n$, 并且没 $U_i \subseteq U_j$, $1 \leq i, j \leq n$

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 $\rho\{R_1, R_2, R_3, \dots\}$ ）

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_i ， 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_i ，如果只有 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 $R\langle U, F \rangle$ ， $U = \{A, B, C, D, E\}$ ， $F = \{A \rightarrow C, B \rightarrow C, C \rightarrow D, DE \rightarrow$

$C,$

$CE \rightarrow A\}$ ， R 的一个分解为 $R_1(AD)$ ， $R_2(AB)$ ， $R_3(BE)$ ，

$R_4(CDE)$ ， $R_5(AE)$

| 模式 \ 属性 | A | B | C | D | E |
|------------|----------|----------|----------|----------|----------|
| $R_1(AD)$ | a_1 | b_{12} | b_{13} | a_4 | b_{15} |
| $R_2(AB)$ | a_1 | a_2 | b_{23} | b_{24} | b_{25} |
| $R_3(BE)$ | b_{31} | a_2 | b_{33} | b_{34} | a_5 |
| $R_4(CDE)$ | b_{41} | b_{42} | a_3 | a_4 | a_5 |
| $R_5(AE)$ | a_1 | b_{52} | b_{53} | b_{54} | a_5 |

$A \rightarrow C$ ，所有 A 中有 a 的都没有对应的 a_3 ，那么就把所有 a_1 对应的 C 都改成相同的

| 模式 \ 属性 | A | B | C | D | E |
|------------|----------|----------|----------|----------|----------|
| $R_1(AD)$ | a_1 | b_{12} | b_{13} | a_4 | b_{15} |
| $R_2(AB)$ | a_1 | a_2 | b_{13} | b_{24} | b_{25} |
| $R_3(BE)$ | b_{31} | a_2 | b_{33} | b_{34} | a_5 |
| $R_4(CDE)$ | b_{41} | b_{42} | a_3 | a_4 | a_5 |
| $R_5(AE)$ | a_1 | b_{52} | b_{13} | b_{54} | a_5 |

$B \rightarrow C$ ，同理，将 a_2 对应的 C 改成相同的

| 模式 \ 属性 | A | B | C | D | E |
|------------|----------|----------|----------|----------|----------|
| $R_1(AD)$ | a_1 | b_{12} | b_{13} | a_4 | b_{15} |
| $R_2(AB)$ | a_1 | a_2 | b_{13} | b_{24} | b_{25} |
| $R_3(BE)$ | b_{31} | a_2 | b_{13} | b_{34} | a_5 |
| $R_4(CDE)$ | b_{41} | b_{42} | a_3 | a_4 | a_5 |
| $R_5(AE)$ | a_1 | b_{52} | b_{13} | b_{54} | a_5 |

$C \rightarrow D$ ，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：

| 模式 \ 属性 | A | B | C | D | E |
|------------|-------|----------|----------|-------|----------|
| $R_1(AD)$ | a_1 | b_{12} | b_{13} | a_4 | b_{15} |
| $R_2(AB)$ | a_1 | a_2 | b_{13} | a_4 | b_{25} |
| $R_3(BE)$ | a_1 | a_2 | a_3 | a_4 | a_5 |
| $R_4(CDE)$ | a_1 | b_{42} | a_3 | a_4 | a_5 |
| $R_5(AE)$ | a_1 | b_{52} | a_3 | a_4 | a_5 |

结果表中无一行全为 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 $R\langle U, F \rangle$ 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 $R\langle U, F \rangle$ 中，若不存在这样的码 X ，属性组 Y 及非主属性 $Z (Z \notin Y)$ ，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 F_c

按照 F_c 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 $R\langle U, F \rangle$ 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) $R(X, Y, Z)$ ， $F = \{XY \rightarrow Z\}$

(2) $R(X, Y, Z)$ ， $F = \{Y \rightarrow Z, XZ \rightarrow Y\}$

(3) $R(X, Y, Z)$ ， $F = \{Y \rightarrow Z, Y \rightarrow X, X \rightarrow YZ\}$

(4) $R(X, Y, Z)$ ， $F = \{X \rightarrow Y, X \rightarrow Z\}$

(5) $R(W, X, Y, Z)$ ， $F = \{X \rightarrow Z, WX \rightarrow Y\}$

(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， $X \rightarrow Z$ 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

$A \rightarrow BCD$ $BC \rightarrow DE$ $B \rightarrow D$ $D \rightarrow A$

1) 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 $B \rightarrow D$ 加入 D ，根据 $D \rightarrow A$ 加入 A ，根据 $A \rightarrow BCD$ 加入 C ，
根据 $BC \rightarrow DE$ 加入 E 没有依赖关系了，得到最终的结果

$B^+ = \{ABCDE\}$

2) 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

$A \rightarrow BCD, BC \rightarrow DE, \therefore A^+ = \{ABCDE\}, AF^+ = r$

8.6 多值依赖 $\alpha, \beta, \gamma, \delta$

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 γ 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：

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

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

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

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 t 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 $U - \alpha - \beta = \emptyset$ 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 $\alpha \rightarrow \beta$ 在 $R(U)$ 上成立, 则对于任何 $\beta' \subseteq \beta$, 均有 $\alpha \rightarrow \beta'$ 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 $\beta' \subseteq \beta$, 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^+ ：

$$R_1 \cap R_2 \twoheadrightarrow R_1$$

$$R_1 \cap R_2 \twoheadrightarrow R_2$$

D 在 R_i 上的限定是集合 D_i ，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：

— 关系模式 R ， $U = \{A, B, C, D, E, G\}$ ， $D = \{A \twoheadrightarrow BCG, B \rightarrow AC, C \rightarrow G\}$ 请将关系模式分解成为 4NF

— $R_1 = \{A, D, E\}$

平凡的多值依赖

违反 4NF

— $R_2 = \{A, B, C, G\}$ ， $D_2 = \{A \twoheadrightarrow BCG, C \rightarrow G, B \rightarrow AC\}$

— $R_{21} = \{C, G\}$ $D_{21} = \{C \rightarrow G\}$

— $R_{22} = \{A, B, C\}$ ， $D_{22} = \{B \rightarrow AC\}$

候选码

公理系统：考虑关系模式 (U, D)

- 1: 若 $\beta \subseteq \alpha \subseteq U$ ，则 $\alpha \rightarrow \beta$
- 2: 若 $\alpha \rightarrow \beta$ ，且 $\gamma \subseteq U$ ，则 $\alpha\gamma \rightarrow \beta\gamma$
- 3: 若 $\alpha \rightarrow \beta$ ， $\beta \rightarrow \gamma$ ，则 $\alpha \rightarrow \gamma$
- 4: 若 $\alpha \twoheadrightarrow \beta$ ， $V \subseteq W \subseteq U$ ，则 $\alpha W \twoheadrightarrow \beta V$
- 5: 若 $\alpha \twoheadrightarrow \beta$ ，则 $\alpha \twoheadrightarrow U - \alpha - \beta$
- 6: 若 $\alpha \twoheadrightarrow \beta$ ， $\beta \twoheadrightarrow \gamma$ ，则 $\alpha \twoheadrightarrow \gamma - \beta$
- 7: 若 $\alpha \rightarrow \beta$ ，则 $\alpha \twoheadrightarrow \beta$
- 8: 若 $\alpha \twoheadrightarrow \beta$ ， $W \rightarrow \gamma$ ， $W \cap \beta = \emptyset$ ， $\gamma \subseteq \beta$ ，则 $\alpha \rightarrow \gamma$

推理规则：

合并规则：若 $\alpha \twoheadrightarrow \beta$ ， $\alpha \twoheadrightarrow \gamma$ ，则 $\alpha \twoheadrightarrow \beta\gamma$

伪传递规则：若 $\alpha \twoheadrightarrow \beta$ ， $W \beta \rightarrow \gamma$ ，则 $W \alpha \twoheadrightarrow \gamma - W \beta$

混合伪传递规则：若 $\alpha \twoheadrightarrow \beta$ ， $\alpha\beta \rightarrow \gamma$ ，则 $\alpha \rightarrow \gamma - \beta$

分解规则：若 $\alpha \twoheadrightarrow \beta$ ， $\alpha \rightarrow \gamma$ ，则 $\alpha \twoheadrightarrow \beta \cap \gamma$ ， $\alpha \twoheadrightarrow \beta - \gamma$ ， $\alpha \twoheadrightarrow \gamma - \beta$

第十章 数据存储和数据存取

作者：杜泽林

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

①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。

②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

③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

①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；

②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：

| | | | | |
|-----------|-------|------------|------------|-------|
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | 12121 | Wu | Finance | 90000 |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 3 | 22222 | Einstein | Physics | 95000 |
| record 4 | 32343 | El Said | History | 60000 |
| record 5 | 33456 | Gold | Physics | 87000 |
| record 6 | 45565 | Katz | Comp. Sci. | 75000 |
| record 7 | 58583 | Califieri | History | 62000 |
| record 8 | 76543 | Singh | Finance | 80000 |
| record 9 | 76766 | Crick | Biology | 72000 |
| record 10 | 83821 | Brandt | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim | Elec. Eng. | 80000 |

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

(1) 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。

(2) 缺点

①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

②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 $i + 1, \dots, n$ 到 $i, \dots, n-1$ ；

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

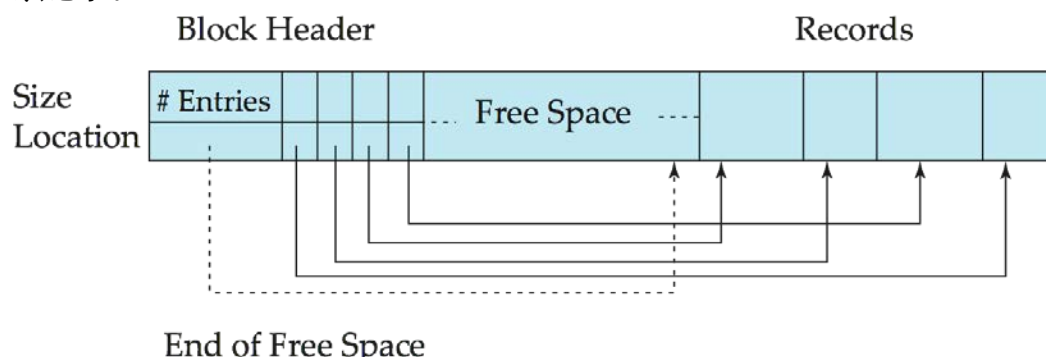
(1) 可变长度记录的几种方式

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)

(2) 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。



①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组成的数组。

②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

①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。

②顺序文件

记录根据“搜索码”的值顺序存储。

③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：在多表聚簇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：

①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。

②用户和账号信息，包括密码

③统计和描述数据

④文件的组织信息

比如：关系的存储组织、关系的存储位置

④ 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

(1) 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。

(2) 缓冲区替换策略

①LRU 策略——系统替换掉那些最近最少使用的块

1、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。

2、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。

②MRU 策略——替换时替换最近最常使用的块

（一）为了介绍 MRU 策略，我们先来了解以下几个概念。

1、被钉住的块——不允许写回磁盘的块。

2、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。

3、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。

（二）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。

（三）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

①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

②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

①顺序索引

基于搜索码值的顺序排序

②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

①能有效支持的访问类型

②访问时间

③插入时间

④删除时间

⑤ 空间开销

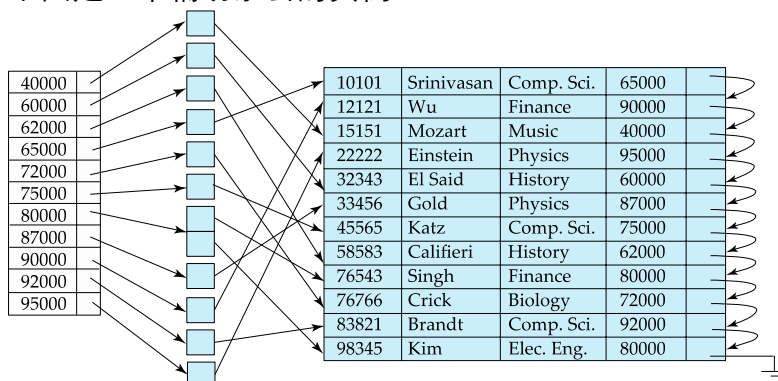
10.6 顺序索引

10.6.1 顺序索引的几个重要概念（容易混淆，注意区分）

1、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。

2、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：



3、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。

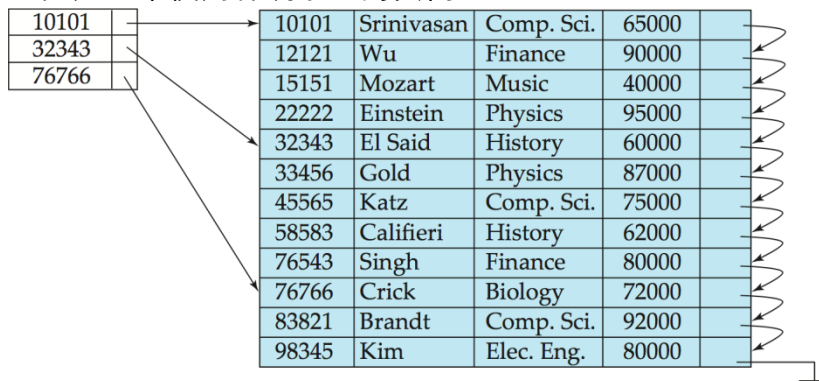
4、稀疏索引：在稀疏索引中，只为搜索码的某些值建立索引项。只有索引是聚集索引时才能使用稀疏索引。

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：



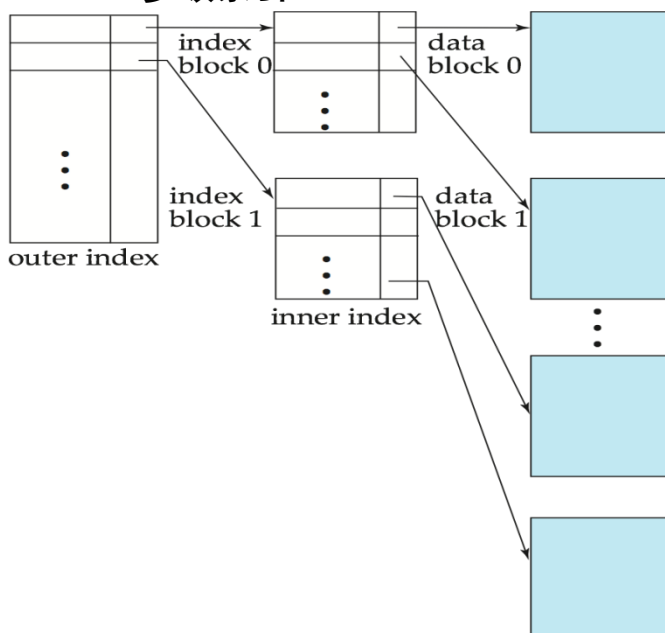
10.6.2 稀疏索引与稠密索引比较

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



①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。

②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

(1) 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。

①稠密索引

搜索码的删除与文件记录的删除类似。

②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。

(2) 单级索引插入

用被插入记录的搜索码值进行一次检索。

①稠密索引

如果搜索码值没有出现在索引中，将其插入。

②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 (a_1, \dots, a_n) 的一组值，其中 a_1, \dots, a_n 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

(1) 索引顺序文件的缺点（主要在结构方面）

①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。

⑥ 插入和删除时，频繁重组整个文件。

(2) B+树索引文件的优点

①在数据插入和删除时，能够通过小的自动调整来保持平衡。

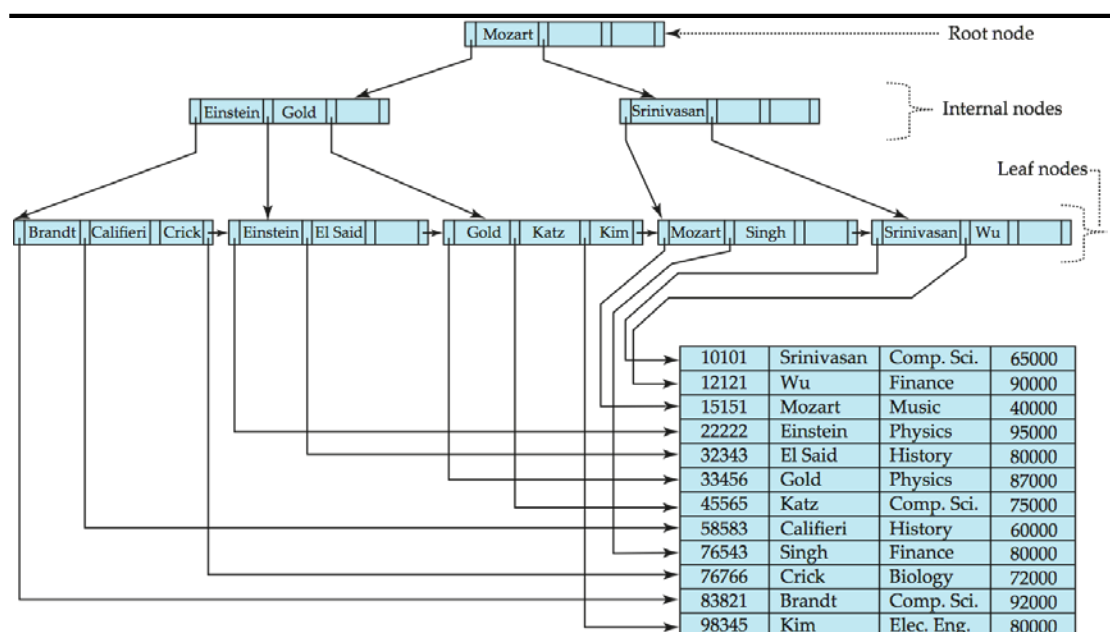
②不需要重组文件来维持性能。

(3) 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



(1) 基本特征

- ①从根结点到叶结点的所有路径长度是一致的。
- ②每一个非根且非叶结点有 $n/2$ 到 n 个孩子结点。
- ③一个叶结点有 $(n-1)/2$ 到 $n-1$ 个值。
- ④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

(2) B+树结点结构

| | | | | | | |
|-------|-------|-------|-----|-----------|-----------|-------|
| P_1 | K_1 | P_2 | ... | P_{n-1} | K_{n-1} | P_n |
|-------|-------|-------|-----|-----------|-----------|-------|

结构特征：

K_i 是搜索码值。 P_i 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。 $K_1 < K_2 < K_3 < \dots < K_{n-1}$ (假设目前没有重复的码值)

(3) B+树的叶子结点

- ①指针 P_i ($i = 1, 2, \dots, n-1$) 指向搜索码值为 K_i 的文件记录
- ②如果 L_i, L_j 是叶结点并且 $i < j$, L_i 的搜索码值小于或等于 L_j 的搜索码值
- ③ P_n 指向按搜索码排序的下一个叶结点

(4) 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：

- ① P_1 指针所指子树上的所有搜索码值小于 K_1
- ②对 $2 \leq i \leq n-1$, P_i 指针所指子树上的所有搜索码值大于或等于 K_{i-1} 且小于 K_i
- ③ P_n 指针所指子树上的所有搜索码值大于或等于 K_{n-1}

10.7.3 B+树的特性

- ①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。

②B+树的一层非叶结点形成一级稀疏索引。

③B+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 $\log_{\lceil n/2 \rceil} (K)$

从而可以有效地进行检索

④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。

10.8 散列文件组织和散列索引

10.8.1 为什么引入散列

(1) 顺序索引的缺点（主要在查询数据方面）

顺序文件的缺点在于我们必须通过索引定位数据，每查找一个搜索码对应的记录，都要对索引进行一次从前往后的遍历。

(2) 散列索引的优点

在散列文件组织中，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的磁盘块地址。因此，基于散列（hash）技术的文件组织使我们能够避免访问索引结构。

10.8.2 散列文件组织的特点

①数据结构支持：桶表示能存储一条或多条记录的一个存储单位，通常一个桶就是一个磁盘块。

②操作算法支持：令 K 表示所有搜索码值的集合，令 B 表示所有桶地址的集合，散列函数 h 是一个从 K 到 B 的函数。为了插入一条搜索码为 K_i 的记录，我们计算 $h(K_i)$ ，结果既是应该存放该记录的桶地址。如果查询搜索码 K_i 查询，需要计算 $h(K_i)$ ，然后搜索具有该地址的桶。比如，有两个搜索码 K_5 和 K_7 具有相同的散列值， $h(K_5) = h(K_7)$ 。如果我们执行对 K_5 的查找，则桶 $h(K_5)$ 包含的搜索码是 K_5 和 K_7 的记录，我们需要检查桶中每一条记录。

下面是一个散列文件组织的例子：

bucket 0

| | | | |
|--|--|--|--|
| | | | |
| | | | |
| | | | |
| | | | |

bucket 1

| | | | |
|-------|--------|-------|-------|
| 15151 | Mozart | Music | 40000 |
| | | | |
| | | | |
| | | | |

bucket 2

| | | | |
|-------|-----------|---------|-------|
| 32343 | El Said | History | 80000 |
| 58583 | Califieri | History | 60000 |
| | | | |
| | | | |

bucket 3

| | | | |
|-------|----------|------------|-------|
| 22222 | Einstein | Physics | 95000 |
| 33456 | Gold | Physics | 87000 |
| 98345 | Kim | Elec. Eng. | 80000 |
| | | | |

bucket 4

| | | | |
|-------|-------|---------|-------|
| 12121 | Wu | Finance | 90000 |
| 76543 | Singh | Finance | 80000 |
| | | | |
| | | | |

bucket 5

| | | | |
|-------|-------|---------|-------|
| 76766 | Crick | Biology | 72000 |
| | | | |
| | | | |
| | | | |

bucket 6

| | | | |
|-------|------------|------------|-------|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 83821 | Brandt | Comp. Sci. | 92000 |
| | | | |

bucket 7

| | | | |
|--|--|--|--|
| | | | |
| | | | |
| | | | |
| | | | |

①instructor 文件的散列文件组织,使用 dept_name 作为码, 8 个桶

②散列算法：假设字母表中的第 i 个字母用整数 i 表示；最初把散列函数值设为 0，从首字符开始迭代，直到最后一个字符为止，每一步迭代都把散列函数值乘以 31 再加上下一个字符的值，结果再取桶数的模得到的值就可以用作索引。比如：

$h(\text{Music}) = 1$ 、 $h(\text{History}) = 2$ 、 $h(\text{Physics}) = 3$ 、 $h(\text{Elec.Eng.}) = 3$

10.8.3 理想的散列函数和桶溢出

(1) 理想的散列函数

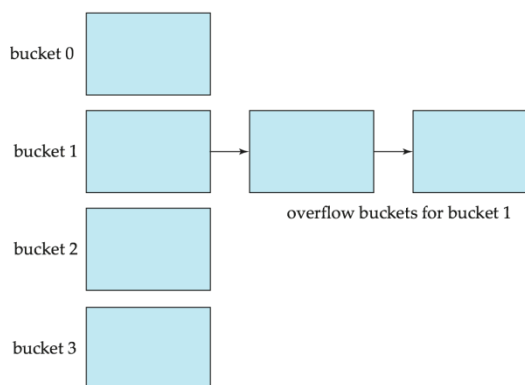
①理想的散列函数是均匀的。理想的散列函数把存储的码均匀地分布到所有桶中，使每个桶含有相同数目的记录。

②理想的散列函数是随机的。即在一般情况下，不管搜索码值实际怎么分布，每个桶应分配到的搜索码值数目几乎相同。

但是，实际应用中所用的散列函数并不是绝对理想的散列函数，往往会随着数据量的加大，出现个别桶被分配到过多的记录，导致桶溢出。

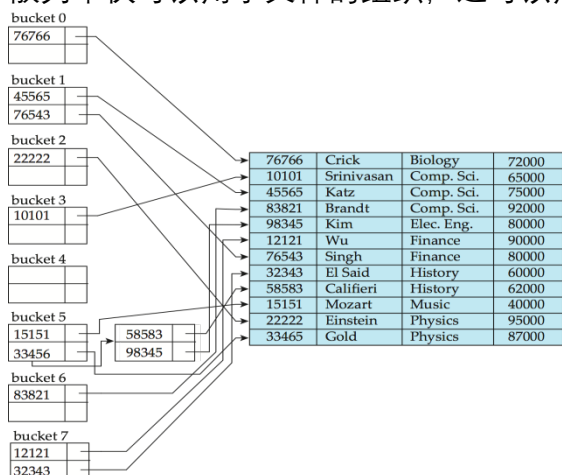
(2) 桶溢出处理

数据库中采用溢出链的方式处理桶溢出所谓溢出链，就是一个给定桶的所有溢出桶用一个链接列表链接在一起。



10.8.4 散列索引

散列不仅可以用于文件的组织，还可以用于索引结构的创建。



①散列索引将搜索码及其相应的指针组织成散列文件结构。

②严格的说，散列索引只是一种辅助索引结构：如果一个文件自身是按散列组织的，就不必在其上另外建立一个独立的索引结构；使用散列索引来表示散列文件结构，同时也用它表示辅助散列索引。

10.8.5 静态散列和动态散列

(1) 静态散列的不足

静态散列技术要求固定桶地址集合 B ，但是大多数数据库都会随时间而变大，这会带来很严重的问题：

①数据增长：

如果初始桶的数目太小，随着文件的增长，由于产生太多溢出，导致性能下降。

②数据收缩：

如果预期增长的空间提前被分配，大量空间将被浪费掉。如果数据库收缩，空间将会被再次浪费。

(2) 针对静态散列不足的解决方案

一种解决方案：

周期性的对散列结构进行重组（规模大，耗时，扰乱正常操作）

另一种解决方案：

允许桶的数量被动态的修改（动态散列）

(3) 动态散列

能够适应数据库增长和收缩的需要，允许散列函数动态改变。

可扩充散列——动态散列的一种形式

①散列函数产生的值范围相对较大，是 b 位二进制整数。

②定义了一个前缀，让前缀的长度为 i 位， $0 \leq i \leq b$ 。再定义一个桶地址表，桶地址表大小为 2^i ，最初 $i = 0$ 。 i 随着数据库的大小变化而增大或减小。

④不同的桶地址目录可能指向同一个桶。

⑤以此模拟桶的结合和分裂，桶数动态变化。

10.8.6 顺序索引和散列索引的比较

(1) 散列的特点

散列的优点在于，它优化了顺序索引中访问一条记录时的最坏访问时间，在访问单个记录时，散列的效率更稳定、更便捷。

散列也有明显的缺点，如果是静态散列，需要付出周期性重组的巨大代价；如果是动态散列，也不可避免的有额外开销。

(2) 针对查询

①确定值的查询：对于具有指定码值的记录检索，散列是一个更好的选择。

②确定范围的查询：散列面对这种查询，基本退化成了全表扫描，顺序索引是更好的选择。

(3) 实际应用

顺序索引被更加广泛的采用，而散列索引只作为辅助

①PostgreSQL 支持散列索引，但是由于变现不佳而不鼓励使用。

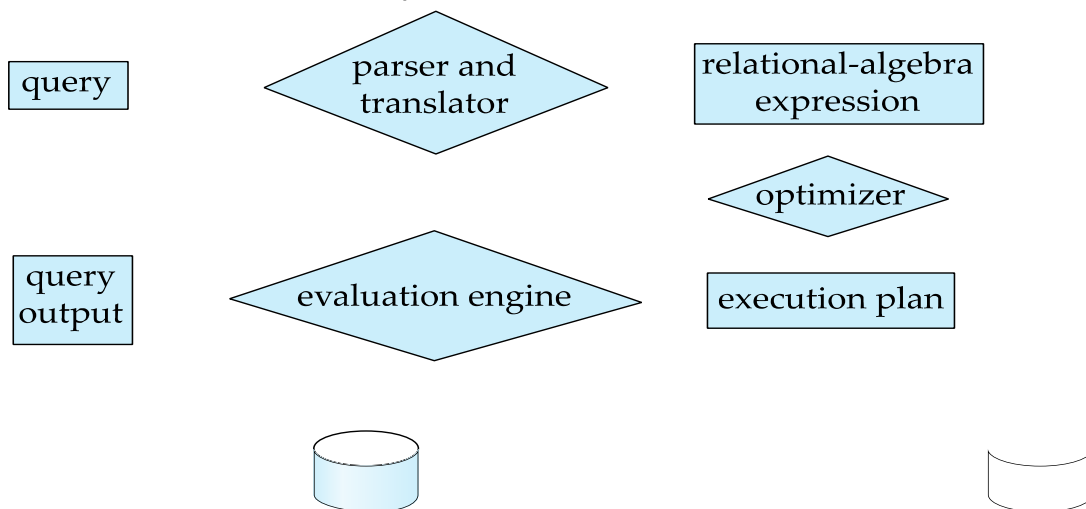
②Oracle 支持静态散列组织，但不支持散列索引。

③SQLServer 仅支持 B+树。

第十一章 查询处理和查询优化

作者：杜泽林

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

- ①语法分析器检查语法，验证关系。
- ②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

(1) 关于优化的几个重要概念

- ①一个关系代数表达式可能有許多等价的表达式。
- ②可以用多种不同的算法来执行每个关系代数运算。
- ③用于执行一个查询的原语操作序列称为查询执行计划。

(2) 查询优化

- ①使用来自数据库目录的统计信息来评估代价。
- ② 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：

①搜索磁盘次数 * 平均寻道时间

②读取的块数 * 平均块读取时间

③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：

①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：

$$b * tT + s * tS$$

②忽略 CPU 时间（实际应用中 CPU 时间应被考虑）。

③没有包括将操作的最终结果写回磁盘的代价。

④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

(1) 线性搜索

①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。

②时间代价： $Cost = br * tT + tS$

③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。

④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 $[(br / 2) \text{ 次磁盘块传输} + 1 \text{ 次磁盘搜索}]$ 。

⑤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。

(2) 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。

a. 主索引，码属性等值比较

①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。

②时间代价： $Cost = (h_i + 1) * (tT + tS)$

③时间代价分析：索引使用 B+树结构（ h_i 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。

b. 主索引，非码属性等值比较

①因为是非码属性，所以允许重复，有可能检索多条记录。

②时间代价： $Cost = h_i * (tT + tS) + tS + tT * b$

③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。

c. 辅助索引，等值比较

1、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

①时间代价： $Cost = (h_i + 1) * (t_T + t_S)$

②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。

2、若索引字段是非码属性，则可检索到多条记录

①时间代价： $Cost = (h_i + n) * (t_T + t_S)$

②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。

d. 主索引，比较

①建立主索引的文件记录是按搜索码的顺序排序的，对于 $\sigma A \geq V(r)$ ，使用索引找到 $\geq v$ 的第一个元组，从这里开始顺序扫描关系。

②时间代价： $Cost = h_i * (t_T + t_S) + t_S + t_T * b$

③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。

④特别的：对于 $\sigma A \leq V(r)$ ，只是顺序扫描关系找到 $\geq v$ 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。

e. 辅助索引，比较

①对于 $\sigma A \geq V(r)$ ，使用索引找到第一个 $\geq v$ 的索引项，从这里开始依次扫描索引，找到指向记录的指针。

②对于 $\sigma A \leq V(r)$ ，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。

③时间代价： $Cost = (h_i + n) * (t_T + t_S)$

④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

(1) 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 加到结果中
    end
end
end
```

① r 被称为连接的外层关系，而 s 称为连接的内层关系。

② 无需索引，并且不管连接条件是什么。

③ 代价很大，因为算法逐个检查两个关系中的每一对元组 $A \theta B$

④ 代价分析：

1、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

$nr * bs + br$ 次块传输

$nr + br$ 次磁盘搜索

2、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

3、最坏的可用内存情况下的成本估算，用 $student$ 作为外层关系：

$5000 * 400 + 100 = 2,000,100$ 次块传输

$5000 + 100 = 5100$ 次磁盘搜索

4、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

(2) 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 $tr \cdot ts$ 加入到结果中

end

end

end

end

① 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形成组对，从而得到全体组对。

② 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。

③ 代价分析：

1、最坏情况

$br * bs + br$ 次块传输

$2 * br$ 次磁盘搜索

2、最好情况

$br + bs$ 次块传输

2 次磁盘搜索

5、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： $100 \times 400 + 100 = 40100$

块搜索： $2 \times 100 = 200$

补充：改进嵌套循环与块嵌套循环算法

①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。

$Cost = \lceil br / (M-2) \rceil * bs + br$ 次块传输 + $2 \lceil br / (M-2) \rceil$ 次磁盘搜索

②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。

③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）。

③ 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。

(3) 索引循环嵌套连接

①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。

②代价分析：

1、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。

2、连接的时间代价： $br (tT + tS) + nr * c$ (c 是使用连接条件对关系 s 进行单次选择操作的代价)

3、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。

(4) 归并连接

①概述：首先，在连接属性上对全部关系进行排序（如果之前并非有序的）。然后，为了连接它们，归并有序关系。连接步骤类似于归并排序算法中的归并阶段，主要不同在于处理连接属性上的重复值，每对具有相同值的连接属性的元组必须被匹配。

②适用场景：可用于计算自然连接和等值连接。

③特点：每个块只需被读取一次。（当然，需要先付出排序的代价）

④代价分析：

$br + bs$ 次块传输 + $\lceil br / bb \rceil + \lceil bs / bb \rceil$ 次磁盘搜索 + 排序代价
(bb 是为每个关系分配的缓冲块数量)

⑤特殊的：混合归并-连接

如果一个关系已排序，并且另一关系有一个连接属性上的 B+树辅助索引。

1、把已排序关系和另一个关系的 B+树辅助索引叶结点进行归并

2、将归并生成的文件按照未排序关系元组的地址进行排序

3、对相关元组按照物理存储顺序进行有效的检索，最终完成连接操作

注：2 和 3 是因为：顺序扫描比随机查找更有效。

(5) 散列连接

①适用场景：等值连接和自然连接。

②原理：用散列函数 h 来划分两个关系的元组， h 是将 $JoinAttrs$ 值映射到 $\{0, 1, \dots, n\}$ 的散列函数，其中 $JoinAttrs$ 表示自然连接中 r 与 s 的公共属性。具体方式为：

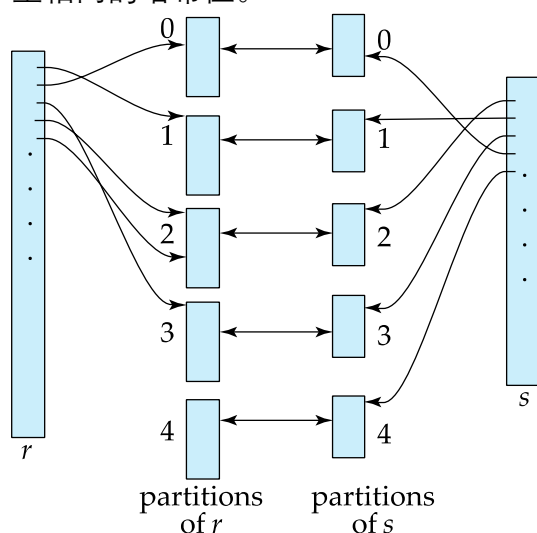
1、 r_0, r_1, \dots, r_n 表示关系 r 的元组划分

每个元组 $tr \in r$ 被放入划分 r_i 中，其中 $i = h(tr[JoinAttrs])$

2、 s_0, s_1, \dots, s_n 表示关系 s 的元组划分

每个元组 $ts \in s$ 被放入划分 s_i 中，其中 $i = h(ts[JoinAttrs])$

关系 r_i 中的元组 r 只需要与关系 s_i 中的元组 s 相比较，而没有必要与其他任何划分里的元组 s 相比较。因为相同属性值经过同样的哈希函数运算必定会产生相同的哈希值。



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：

①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）

②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

(1) 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。

(2) 特点：

①任何情况下，物化计算都是永远适用的。

②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

(1) 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。

(2) 特点：

①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。

②比实体化代价小很多。

③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

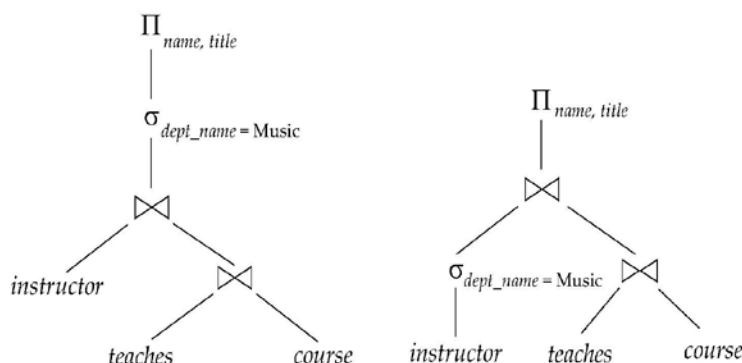
11.5.1 查询优化概述

①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。

②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。

③结合实例体会查询优化

$\Pi_{name, title}(\sigma_{dept_name = 'music'}(instructor \bowtie (teaches \bowtie \Pi_{courseid, title}(course))))$



左侧的表达式树将产生很大的中间关系， $instructor \bowtie (teaches \bowtie \Pi_{courseid, title}(course))$ ，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。

④基于代价的优化步骤

- 1、使用等价规则产生逻辑上的等价表达式
- 2、注解结果表达式来得到替代查询计划
- 3、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

(1) 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。

(2) 等价规则

a. 等价规则的具体内容

①合取选择运算可以被分解为单个选择运算的序列

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

②选择运算满足交换律

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

③一系列投影中只有最后一个运算是必需的，其余的可省略

④选择操作可与笛卡尔积以及 θ 连接相结合

$$\sigma_{\theta}(E1 \times E2) = E1 \bowtie_{\theta} E2$$

⑤ θ 连接运算满足交换律

$$E1 \bowtie_{\theta} E2 = E2 \bowtie_{\theta} E1$$

⑥自然连接运算满足结合律

$$(E1 \bowtie E2) \bowtie E3 = E1 \bowtie (E2 \bowtie E3)$$

⑦ θ 连接具有以下方式的结合律

$$(E1 \bowtie_{\theta1} E2) \bowtie_{\theta2 \wedge \theta3} E3 = E1 \bowtie_{\theta1 \wedge \theta3} (E2 \bowtie_{\theta2} E3)$$

⑧选择运算在下面两个条件下对 θ 连接运算具有分配率

(a) 当选择条件 $\theta\theta$ 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：

$$\sigma_{\theta\theta}(E1 \bowtie_{\theta} E2) = (\sigma_{\theta\theta}E1) \bowtie_{\theta} E2$$

(b) 当选择条件 $\theta1$ 只涉及 $E1$ 的属性，选择条件 $\theta2$ 只涉及 $E2$ 的属性时，满足分配率：

$$\sigma_{\theta1 \wedge \theta2}(E1 \bowtie_{\theta} E2) = (\sigma_{\theta1}(E1)) \bowtie_{\theta} (\sigma_{\theta2}(E2))$$

⑨投影运算在下列条件下对 θ 连接运算具有分配率

(a) 假设连接条件 θ 只涉及 $L1 \cup L2$ 中的属性

$$\pi_{L1 \cup L2}(E1 \bowtie_{\theta} E2) = (\pi_{L1}(E1)) \bowtie_{\theta} (\pi_{L2}(E2))$$

(b) 考虑连接 $E1 \bowtie_{\theta} E2$

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 $L1 \cup L2$ 中的属性

令 $L4$ 是 $E2$ 中出现在连接条件 θ 中但不在 $L1 \cup L2$ 中的属性

$$\pi_{L1 \cup L2}(E1 \bowtie_{\theta} E2) = \pi_{L1 \cup L2}((\pi_{L1 \cup L3}(E1)) \bowtie_{\theta} (\pi_{L2 \cup L4}(E2)))$$

⑩集合的并与交满足交换律

$$E1 \cup E2 = E2 \cup E1$$

$$E1 \cap E2 = E2 \cap E1$$

⑪集合的并与交满足结合律

$$(E1 \cup E2) \cup E3 = E1 \cup (E2 \cup E3)$$

$$(E1 \cap E2) \cap E3 = E1 \cap (E2 \cap E3)$$

⑫选择运算对 \cup , \cap 和 $-$ 运算具有分配率

$$\sigma_{\theta}(E1 - E2) = \sigma_{\theta}(E1) - \sigma_{\theta}(E2)$$

上述规则将“-”替换成 \cup 或 \cap 时也成立。

$$\sigma_{\theta}(E1 - E2) = \sigma_{\theta}(E1) - E2$$

上述规则将“-”替换成 \cap 时成立，替换成 \cup 时不成立

⑬投影运算对并运算具有分配率

$$\pi_L(E1 \cup E2) = (\pi_L(E1)) \cup (\pi_L(E2))$$

b. 等价规则使用的原则

①尽可能早地执行选择操作以减小被连接的关系的大小。

②尽可能早地执行投影操作以减小被连接的关系的大小。

③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。

(1) 统计信息

①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：

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

②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。

(2) 估计方法

① $\sigma A=v(r)$

nr / $V(A, r)$: 满足选择的记录数

② $\sigma A \leq V(r)$

1、c 表示满足条件的元组的估计数

2、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：

$$n_r \cdot \frac{v - \min(A, r)}{\max(A, r) - \min(A, r)}$$

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 $nr / 2$

③选中率

条件 θ_i 的选中率是关系 r 上一个元组满足 θ_i 的概率。

1、合取 $\sigma \theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n (r)$

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

2、析取 $\sigma \theta_1 \vee \theta_2 \vee \dots \vee \theta_n (r)$

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

3、取反 $\sigma \neg \theta(r)$

$$nr - \text{size}(\sigma \theta(r))$$

④连接运算

- 1、笛卡尔积 $r \times s$ 包含 $n_r \cdot n_s$ 个元组，每个元组占用 $s_r + s_s$ 个字节。
- 2、若 $R \cap S = \emptyset$ ，则 $r \bowtie s$ 与 $r \times s$ 结果一样。
- 3、若 $R \cap S$ 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， $r \bowtie s$ 的元组数不会超过 s 元组的数目。
- 4、若 $R \cap S$ 构成了 S 中参照 R 的外码， $r \bowtie s$ 中的元组数正好与 s 中的元组数相等。
- 5、若 $R \cap S$ 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 $r \bowtie s$ 中产生的元组个数估计为下式：

$$\frac{n_r * n_s}{V(A, s)}$$

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，则必须使用更发杂的估算方法。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

(1) 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：

- ①搜索所有的计划，基于代价选择最佳的计划
- ②使用启发式方法选择计划

(2) 基于代价的优化

a. 基本理念：

从给定查询等价的所有查询计划执行空间进行搜索，并选择估计代价最小的一个。

b. 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 $r_1 \bowtie r_2 \bowtie \dots \bowtie r_n$ 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！

(3) 启发式优化

a. 基本理念：

- ①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。
- ②尽早执行选择运算（减少元组数目）。
- ③尽早执行投影运算（减少属性数目）。
- ④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）

b. 查询优化器

①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。

②一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：

- 1、启发式重写嵌套块结构和聚集

2、对每个块通过基于代价的连接顺序进行优化

③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。

④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

作者：杜泽林

12.1 事务的基本概念

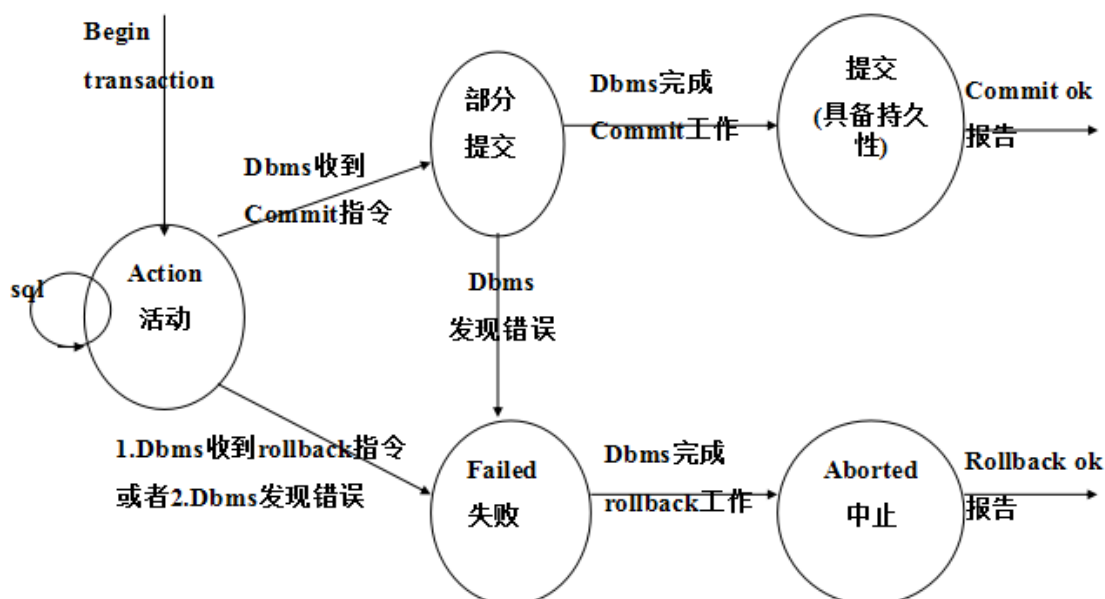
12.1.1 事务定义

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)

①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。

②一致性

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。

③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。

④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

①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。

②一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

①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。

②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

(1) 基本比较

①并行事务会破坏数据库的一致性。

②串行事务效率低。

(2) 并行的优点

①一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)。

②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)。

(3) 核心问题

在保证一致性的前提下最大限度地提高并发度。

(4) 并发操作面临的问题

①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。

②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：

1、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。

2、事务 2 删除了其中部分记录，当事务 1 再次读取数据时，发现某些记录神秘地消失了。

3、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。

③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。

(5) 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。

①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。

②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。

(6) 冲突可串行化

a. 指令顺序

考虑一个调度 S 中的两条连续指令（仅限 read 与 write 操作） I_i 与 I_j ，分别属于事务 T_i 与 T_j

① $I_i = \text{read}(Q), I_j = \text{read}(Q);$

② $I_i = \text{read}(Q), I_j = \text{write}(Q);$

③ $I_i = \text{write}(Q), I_j = \text{read}(Q);$

④ $I_i = \text{write}(Q), I_j = \text{write}(Q);$

在①情况下， I_i 与 I_j 的次序无关紧要。其余情况下， I_i 与 I_j 的次序不同，其执行结果也不同，数据库最终状态也不同。

b. 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， I_i 与 I_j 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。

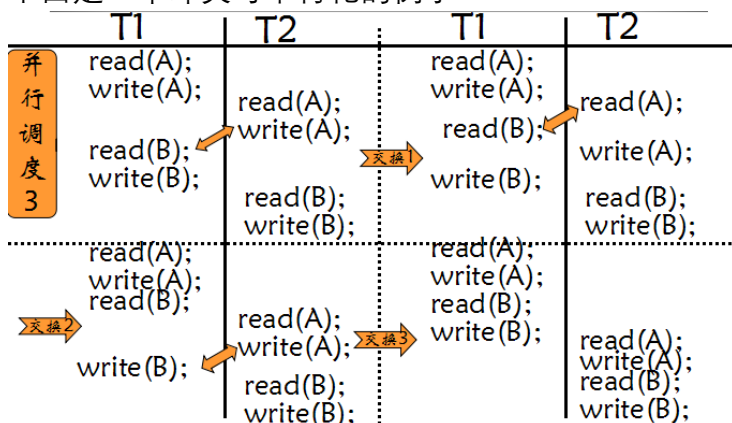
c. 冲突等价

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)。

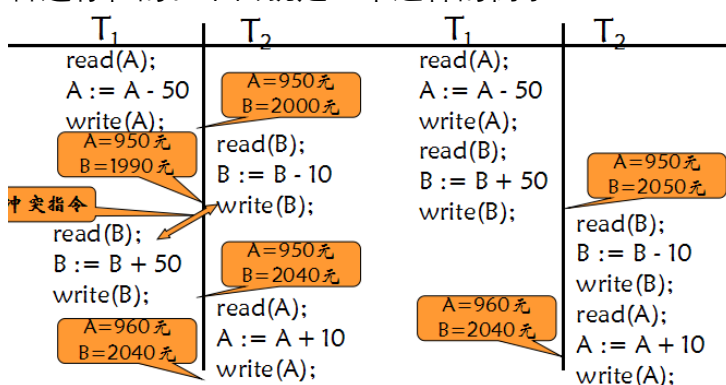
d. 冲突可串行化

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

①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 $T_i \rightarrow T_j$ 组成：

- ①在 T_j 执行 $read(Q)$ 之前， T_i 执行 $write(Q)$
- ②在 T_j 执行 $write(Q)$ 之前， T_i 执行 $read(Q)$
- ③在 T_j 执行 $write(Q)$ 之前， T_i 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。

②并行转串行准则：

如果优先图中存在边 $T_i \rightarrow T_j$ ，则在任何等价于 S 的串行调度 S' 中， T_i 都必须出现在 T_j 之前。

③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

(1) 可恢复调度

对于每对事务 T_1 与 T_2 ，如果 T_2 读取了 T_1 所写的的数据，则 T_1 必须先于 T_2 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。

(2) 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

(1) 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）。

(2) 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

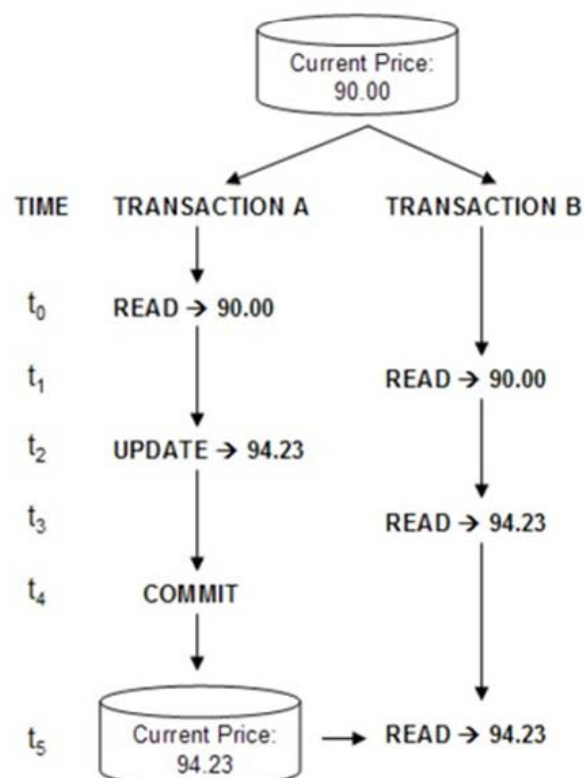
a. 未提交读

①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。）

②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）。



b. 已提交读

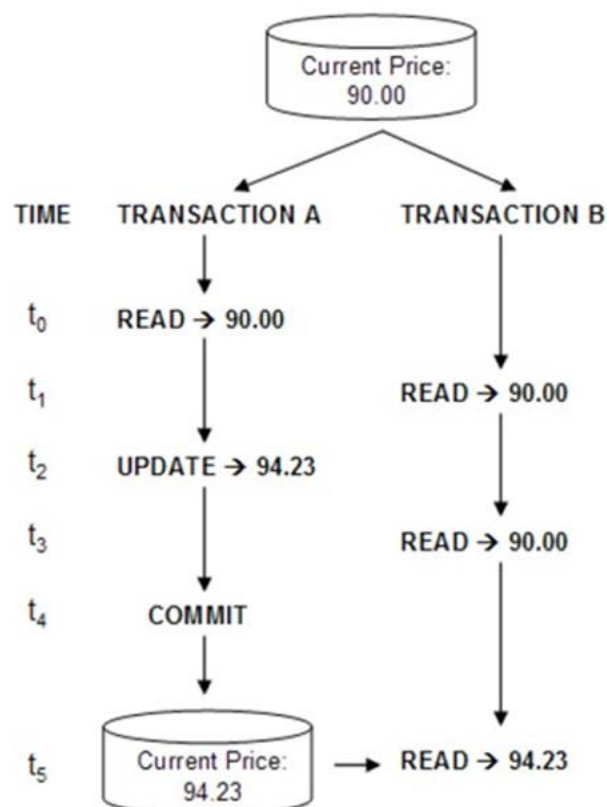
①只允许读取已提交数据，但不要求可重复读。（当事务 A 更新某条数据时，不容许其他事务进行任何操作包括读取，但事务 A 读取时，其他事务可以进行读取、更新）

②结合实例理解已提交读：

当事务 A 在 t_2 时刻更新了价格

（94.23）之后，事务 B 在 t_3 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。



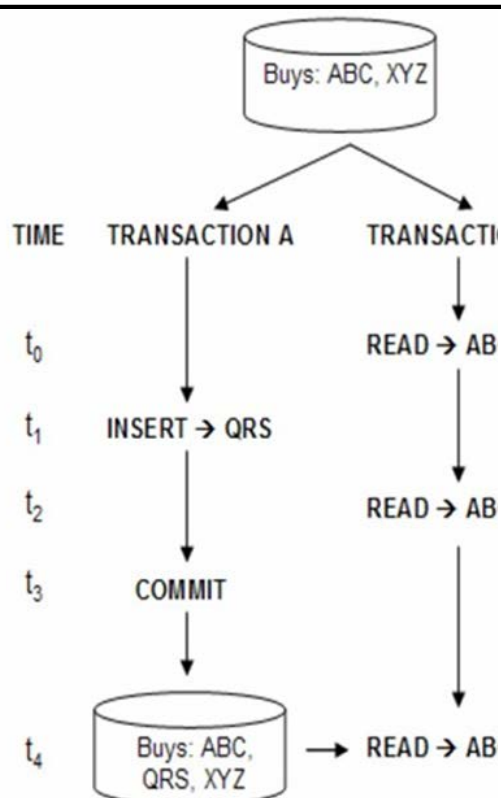
c. 可重复读

①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。

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。

②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 t_2 时刻查询所得的结果依然和 t_0 时刻一样（不包含 QRS），即使到了 t_4 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

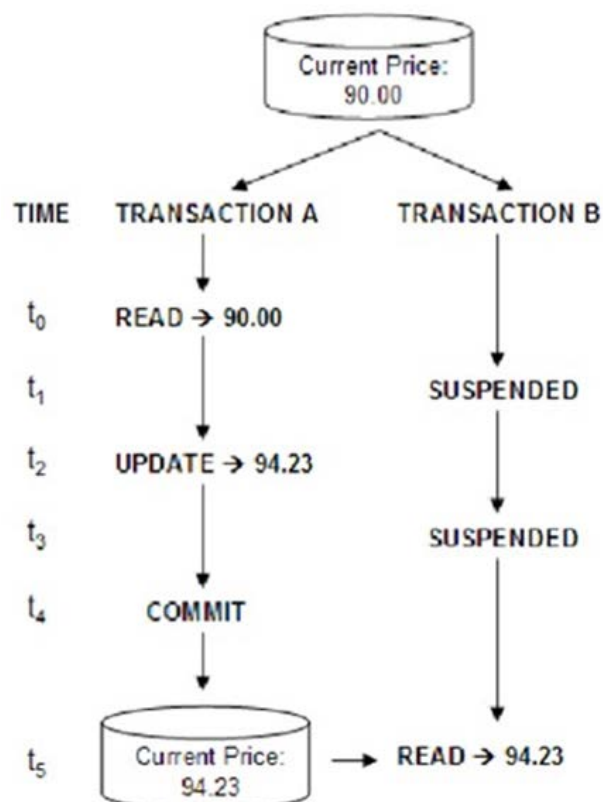


d. 可串行化

①保证可串行化调度

②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：

- ①基于锁的协议
- ②基于时间戳的协议
- ③基于有效性检查的协议

12.3.1 基于锁的协议

(1) 锁

a. 两种封锁类型

- ①排它锁 (eXclusive lock, 简记为 X 锁)
- ②共享锁 (Share lock, 简记为 S 锁)

b. 排他锁

- ①排它锁又称为写锁。
- ②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。

c. 共享锁

- ①共享锁又称为读锁。
- ②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。

d. 饥饿

①饥饿产生的原因

假设系统中有一系列事务 A_i 读一项数据，还有一个事务 B 需要更新同一项数据。当 A_1 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 A 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。

②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。

(2) 封锁协议

- ①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

- ②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。
- ③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。

(3) 两阶段封锁协议

- a. 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。

b. 两阶段封锁协议的特性：

①并行执行的所有事务均遵守两段锁协议，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，所有遵守两段锁协议的事务，其并行执行的结果一定是正确的。

②事务遵守两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段锁协议。

| T1 | T2 | T1 | T2 | T1 | T2 |
|----------|----------|----------|----------|----------|----------|
| Lock-S B | | Lock-S B | | Lock-S B | |
| 读B=2 | | 读B=2 | | 读B=2 | Lock-S A |
| Lock-X A | | Y=B | | Unlock B | 读A=2 |
| | Lock-S A | Unlock B | | Lock-X A | Unlock A |
| A=B+1 | 等待 | Lock-X A | | A=B+1 | Lock-X B |
| 写回A=3 | 等待 | | Lock-S A | 写回A=3 | 等待 |
| Unlock B | 等待 | | 等待 | Unlock A | Lock B |
| Unlock A | 等待 | A=Y+1 | 等待 | | B=A+1 |
| | 等待 | 写回A=3 | 等待 | | 写回B=3 |
| | Lock-S A | Unlock A | 等待 | | Unlock B |
| | 读A=3 | | Lock-S A | | |
| | Lock-X B | | 读A=3 | | |
| | B=A+1 | | X=A | | |
| | 写回B=4 | | Unlock A | | |
| | Unlock B | | Lock-X B | | |
| | Unlock A | | B=X+1 | | |
| | | | 写回B=4 | | |
| | | | Unlock B | | |

(a) 遵循两阶段封锁

(b) 不遵循两阶段封锁

(c) 不遵循两阶段封锁

③普通的两阶段封锁协议下不能避免死锁。

④普通的两阶段封锁协议下不能避免级联回滚。

c. 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。

①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。

②强两阶段封锁协议

事务提交之前，不得释放任何锁。

(4) 多粒度封锁协议

a. 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，要是对于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。

b. 单一粒度的缺点

①封锁粒度大：并发性低

②封锁粒度小：访问大粒度数据加锁量巨大

c. 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

d. 多粒度封锁的基本原则

- ①大粒度数据由小粒度数据组成。
- ②允许对不同粒度数据进行封锁。
- ③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。

e. 多粒度封锁判定授予锁

1、申请小粒度锁的判定

- ①判定在申请数据上有没有不相容锁。
- ②判定在申请数据相关大粒度数据上，有没有不相容锁。
- ③粒度的层次有限，本判定不困难

2、申请大粒度锁的判定

- ①判定在申请数据上有没有不相容锁。
- ②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
- ③小粒度的数据量可能巨大，本判定困难。

3、优化申请大粒度锁

- ①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。
- ②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。
- ③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。

- ④三种意向锁：共享意向锁（IS）/排他意向锁（IX）/共享排他意向锁（SIX）

f. 多粒度封锁协议

- ①遵从锁的相容矩阵
- ②根结点必须首先加锁，可以加任何类型的锁
- ③仅当 T_i 对 Q 的父结点持有 IX 或 IS 锁时， T_i 对于结点 Q 加 S 或者 Is 锁
- ④仅当 T_i 对 Q 的父结点持有 IX 或 SIX 锁时， T_i 对于结点 Q 加 X、SIX、IX 锁
- ⑤仅当 T_i 未曾对任何结点解锁时， T_i 可以对结点加锁（两阶段的）
- ⑥仅当 T_i 当前不持有 Q 的子节点的锁时， T_i 可以对节点 Q 解锁

下图是多粒度封锁相容矩阵：

| | IS | IX | S | S IX | X |
|------|----|----|---|------|---|
| IS | ✓ | ✓ | ✓ | ✓ | × |
| IX | ✓ | ✓ | × | × | × |
| S | ✓ | × | ✓ | × | × |
| S IX | ✓ | × | × | × | × |
| X | × | × | × | × | × |

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。

(1) 概述

①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。

②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。

(2) 事务的时间戳

对于系统中的每一个事务 T_i ，将唯一的时间戳与它相联系，记为 $TS(T_i)$ 。

①时间戳的两种简单方法：系统时钟&逻辑计数器

②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。

(3) 数据项时间戳

①W-timestamp(Q)：表示成功执行 write(Q)的所有事务的最大的时间戳。

②R-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： $TS(T_1)=1; TS(T_2)=2;$

$T_2: read(Q) \quad //r-ts(Q)=2$

$T_1: read(Q) \quad //r-ts(Q)=2 \quad (\neq 1!)$

(4) 时间戳排序协议

1、假设事务 T_i 发出 read(Q)

①如果 $TS(T_i) < W\text{-timestamp}(Q)$ ，则 T_i 需读入的 Q 值已被覆盖。因此，read 操作被拒绝， T_i 回滚。

②如果 $TS(T_i) \geq W\text{-timestamp}(Q)$ ，则执行 read 操作，R-timestamp(Q) 被设为 R-timestamp(Q) 和 $TS(T_i)$ 两者的最大值。

2、假设事务 T_i 发出 write(Q)

①如果 $TS(T_i) < R\text{-timestamp}(Q)$ ，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被产生。因此，write 操作被拒绝， T_i 回滚。

②如果 $TS(T_i) \geq R\text{-timestamp}(Q)$ ，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。

③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 | T4 | T5 | R_ (A) | W_ (A) | R_ (B) | W_ (B) |
|--------|--------------|--------------|--------|--------|--------|--------|
| TS(T4) | *start* | | | | | |
| TS(T5) | Read(B) | *start* | | | TS(T4) | |
| Time1 | | Read(B) | | | TS(T5) | |
| Time2 | | B=B-50 | | | | |
| Time3 | | Write(B) | | | | TS(T5) |
| Time4 | Read(A) | | TS(T4) | | | |
| Time5 | | Read(A) | | TS(T5) | | |
| Time6 | Display(A+B) | | | | | |
| Time7 | | A=A+50 | | | | |
| Time8 | | Write(A) | | TS(T5) | | |
| Time9 | | Display(A+B) | | | | |

(5) 时间戳排序协议的特性

- ①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。
- ②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。
- ③存在饥饿现象。事务可能被反复回滚、重启。
- ④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。

(6) 时间戳排序协议的优化

- ①与两阶段封锁协议和多版本协议相结合。

②调度可恢复方法：（下列之一）

- 1、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。
- 2、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
- 3、事务 Ti 读取了其他事务所写的数据，只有在其他事务提交之后，Ti 才能提交

③Thomas 写规则

假如事务 Ti 发出 write(Q)

1、如果 $TS(Ti) < R\text{-timestamp}(Q)$ ，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。

2、如果 $TS(Ti) < W\text{-timestamp}(Q)$ ，则 T1 试图写入的值已过时，因此，忽略这个写操作

3、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。

(1) 划分事务阶段

每个事务 T_i 在其生存期中按两个或三个阶段执行：

- 1、读阶段：各数据项值被读入，并保存在事物 T_i 的局部变量中。
- 2、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。
- 3、写阶段：若事务 T_i 已经通过有效性检查，进行实际的数据库更新，否则，回滚。

(2) 按照阶段设置时间戳

- 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阶段的时间。
- 3、 $Finish(T_i)$ ：事务 T_i 完成写阶段的时间。

(3) 有效性检查协议

①利用时间戳 $Validation(T_i)$ 的值，通过时间戳排序技术决定可串行化顺序。

$TS(T_i) = Validation(T_i)$

- 1、事务完成读之后即更改其时间戳的值。
- 2、之所以选择 $Validation(T_i)$ 的值作为事务 T_i 的时间戳，而不使用 $Start(T_i)$ ，是为了在冲突频度低的情况下，可以拥有更快的响应时间。

②事务 T_j 的有效性测试要求任何满足 $TS(T_i) < TS(T_j)$ 的事务 T_i 必须满足下列条件之一：

1、 $Finish(T_i) < Start(T_j)$

2、 T_i 所写的数据项集与 T_j 所读数据项集不相交，并且 T_i 的写阶段在 T_j 开始其有效性检查阶段之前完成 ($start(T_j) < finish(T_i) < validation(T_j)$)，此条件保证 T_i 和 T_j 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。

(1) 死锁预防

预防死锁的发生就是要破坏产生死锁的条件。

a. 一次封锁法

①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。

②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。

b. 顺序封锁法

①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。

②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。

c. 抢占与事务回滚

①在抢占机制中，当事务 T_i 所申请的锁被事务 T_j 所持有时，授予 T_j 的锁可能通过回滚事务 T_j 被抢占，并将锁授予 T_i 。

②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，破除了饥饿。

③两种技术：

1、Wait-die(非抢占技术)：当事务 T_i 申请的数据项当前被事务 T_j 持有时，仅当 T_i 的时间戳小于 T_j 的时间戳时，允许 T_i 等待，否则 T_i 回滚。

2、Wound-die（抢占技术）：当事务 T_i 申请的数据项当前被事务 T_j 持有时，仅当 T_i 的时间戳大于 T_j 的时间戳时，允许 T_i 等待，否则 T_j 回滚。

注：

①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。

②二者的共同问题是：发生不必要的回滚

(2) 死锁的诊断与解除

a. 概述

①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。

②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。

b. 检测死锁

1、超时法

①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。

②优点：实现简单。

③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。

2、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 $G=(V, E)$

① V 为结点的集合，每个结点表示正运行的事务

② E 为边的集合，每条边表示事务等待的情况

③若 T_i 等待 T_j ，则 T_i, T_j 之间划一条有向边，从 T_i 指向 T_j

④事务 T_j 不再持有事务 T_i 所需要的数据项时，边从等待图中删除

c. 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

①事务故障：逻辑故障&系统错误

②系统崩溃

③磁盘故障

12.4.2 恢复算法（养兵千日，用兵一时）

①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。

②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。

①RAID（独立冗余磁盘阵列，Redundant Array of Independent Disk）

②归档备份保存至磁带

12.4.4 基于日志的恢复

(1) 系统日志

1、日志是日志记录的序列，记录数据库中所有的更新活动。

2、先写日志，后写数据库。

3、日志的组成：事务标识符、数据项标识符、旧值、新值

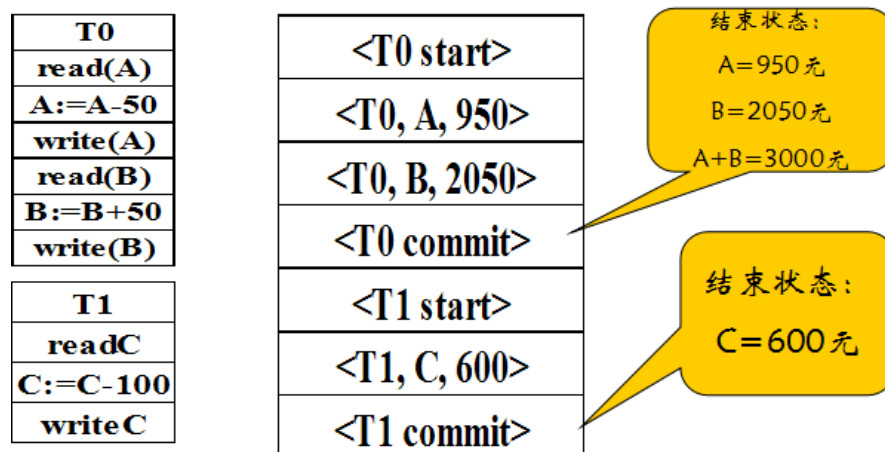
下面是几种常见的日志：

$\langle T_i \text{ start} \rangle$; $\langle T_i, X_j, V_1, V_2 \rangle$; $\langle T_i, \text{commit} \rangle$; $\langle T_i, \text{abort} \rangle$

(2) 延迟的数据库修改的恢复机制

a. 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。

b. 恢复机制

①基础操作

Redo(T_i): 将事务 T_i 更新的所有数据项的值设为新值。

②操作原则

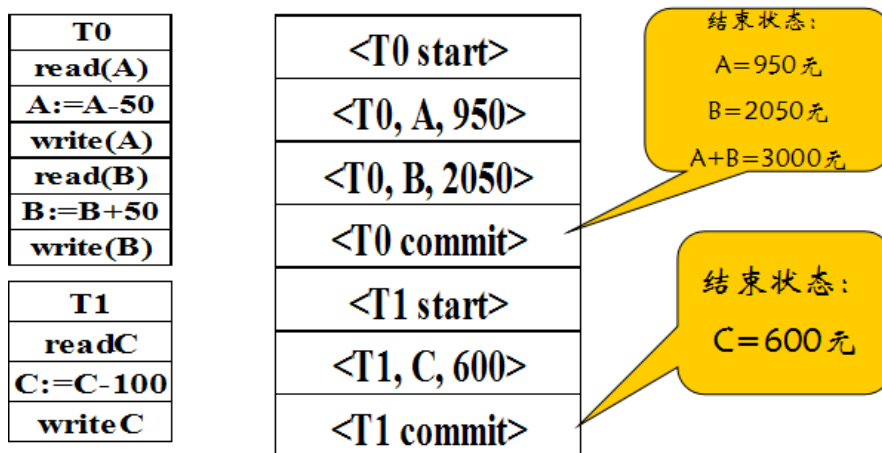
事务 T_i 需要 Redo 操作，当且仅当日志中既包含记录 $\langle T_i, \text{start} \rangle$ 又包含记录 $\langle T_i, \text{commit} \rangle$ 。

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。

(3) 立即的数据库修改的恢复机制

a. 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。

b. 恢复机制

① 基础操作

1、Undo(Ti)：将事务 Ti 所有更新的所有数据项的值恢复成旧值。

2、Redo(Ti)：将事务 Ti 所有更新的所有数据项的值置为新值。

② 操作原则

1、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>

2、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。

c. 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。

c1. 检查点

由系统周期性地执行检查点，需要执行下列操作：

① 将当前位于主存的所有日志记录输出到稳定存储器上。

② 将所有修改了的缓冲块输出到磁盘上。

③ 将一个日志记录 <checkpoint> 输出到稳定存储器。

④ 检查点执行过程中，不允许事务执行更新操作。

c2. 优化操作

1、基本原则：

① 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 <checkpoint> 之前，这表示系统故障前 Ti 已经提交，Ti 的操作有效。

②确定最近的检查点发生前开始执行的最近的一个事务 T_i ，对于 T_i 和 T_i 之后的开始执行的事务 T_j 执行 redo 和 undo 操作。

2、具体过程

①系统由后向前扫描日志，直至发现第一个<checkpoint>：

Redo-list：对每一个形如< T_i commit>的记录，将 T_i 加入 Redo-list

Undo-list：对每一个形如< T_i start>的记录，如果 T_i 不属于 Redo-list，将 T_i 加入 undo-list

②Redo-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<checkpoint>记录。

系统由最近一条<checkpoint>记录由前向后扫描日志，并且对 redo-list 中事务 T_i 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。