数据库概论

信息与数据

数据是信息的**载体**，信息是数据的**内涵。**

数据库

所谓数据库是长期储存在计算机内的、有组织的、可共享的数据集合。

数据库中的数据按一定的**数据模型**组织、存储和描述，由**DBMS**统一管理，多用户共享。

数据管理发展阶段

数据管理的三个阶段：人工管理阶段、文件系统阶段、数据库系统阶段。

- 人工管理：不存储数据、数据不共享、无独立性
- 文件系统：数据存储为文件、有一定的共享性、有一定的物理独立性、无逻辑独立性
- 文件系统：数据冗余与不一致、数据独立性整体较差
- 数据库系统：**面向全组织的结构化**，**数据集成与共享**、**可控冗余度**，**数据独立性好**，**统一的控制机制**

数据库系统

数据库系统是指一个**计算机存储记录**的系统。

它是一个计算机系统，该系统的目标是**存储**信息，并支持用户**检索**和**更新**所需要的信息。

数据库系统的特点：**面向全组织的结构化**，**数据集成与共享**、易扩充性、**可控冗余度**，**数据独立性好**（包括逻辑独立性与物理独立性），**统一的控制机制**（安全性与完整性、并发控制、数据库恢复）。

数据库系统与文件系统的本质差别：**面向全组织的结构化。**

数据库系统的组成部分：**数据库**、**软件**（DBMS,支持DBMS的OS,高级语言及其Compiler,应用开发工具与应用系统）、**硬件**（内存\存取设备\IO）、**用户**（DBA、偶然用户、...）

DBA：数据库管理员，决定和监控数据库

数据库管理系统——DBMS

数据库管理系统（DBMS）是一个**通用的软件系统**，由一组计算机程序构成。

DBMS：**数据库定义**功能（提供DDL，数据描述语言）、**数据存取**功能（提供DML，数据操作语言，分为宿主型与自含型）、**数据库运行**管理、**数据组织存储与管理**、数据库的**建立和维护**功能

数据模型

分类：概念数据模型、逻辑数据模型、物理数据模型

组成：数据结构、数据操作、数据的约束条件

数据结构的组成：数据本身、数据之间的联系

概念数据模型

实体（值）、属性、域、实体型、实体集

能唯一标识实体的属性集：码

联系的种类

E-R图

逻辑数据模型

层次模型：树结构，结点是数据集合，边是多个指针，不可表达多对多，对用户要求较高。

网状模型：有向图，结点是数据集合，边是多个指针，支持多对多。表达能力强。易用性差。

关系模型：二维表。将数据本身与数据之间的联系统一形式。联系的形式：将两个实体的码（唯一标识）放在同一个表项中。建立在严格的数学概念基础上（集合论）。简单直观。数据独立性、安全保密性强。性能相对更差。

面向对象模型：增添方法，以实现对属性值的操作。

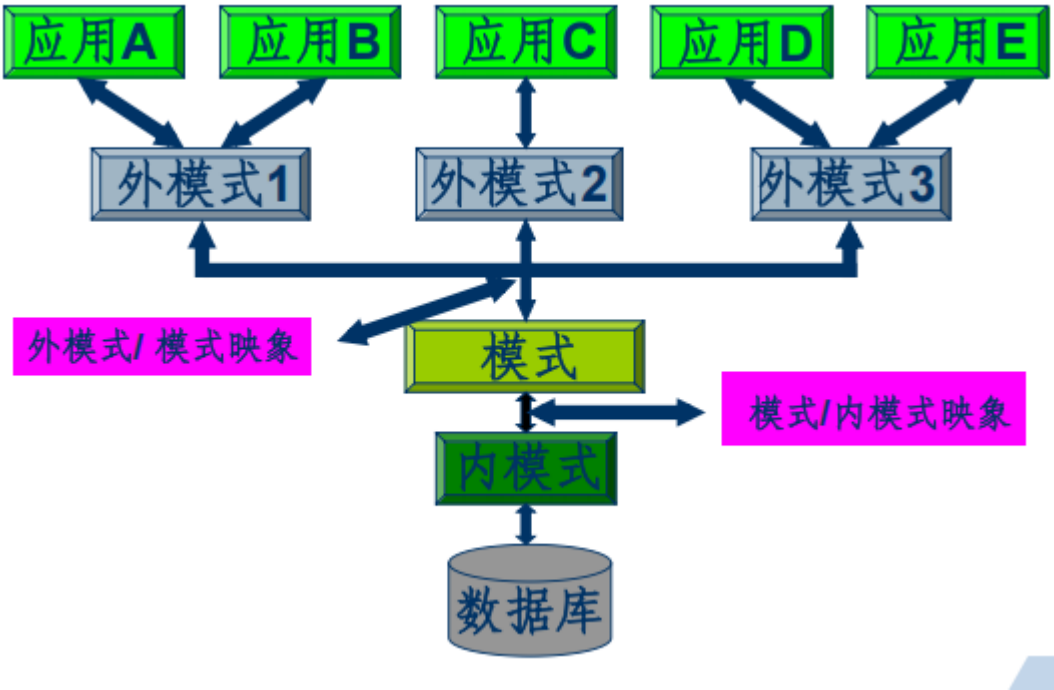
数据库系统模式

模式：数据的结构与联系的描述

型是对一类属性结构和属性的说明，值是型的具体赋值

模式也有其对应的实例。

数据库系统**三级模式与两级映像**



内模式：存储模式

模式：逻辑模式（所有数据的逻辑结构与属性）

外模式：模式的子集，是用户的数据视图（为用户定义外模式）

意义：某模式改变时，**DBMS自动调整映象**，使得其他模式无需改变。

数据独立性：**逻辑独立性**与**物理独立性**

逻辑独立性：外模式/模式映象，模式改变，外模式不变

数据独立性：模式/内模式映象，内模式改变，模式不变

关系数据库

关系模型的三个组成要素：

数据结构：关系

> 实体型和联系用同样的二维表结构表示。
>
> 联系转为表时：可以单独成表，在某些条件下可以与相连的实体合并

数据操作：关系代数、SQL、~~关系演算（不学）~~

> 关系操作的特点：1.面向**集合操作**，操作的**对象与结果**都是**集合** 2.非过程化，只需给出做什么，无需给出怎么做

数据的约束条件：实体完整性约束、参照完整性约束、用户定义完整性约束

数据结构

域：某属性的值的集合（值域）

域之间做**笛卡尔积**得到一张大表，表中的一行称为n-元组或**元组**，元组的每一个值叫做一个**分量。**

笛卡尔积的基数为**每个域的基数的积。**

> D1(1,2) ×D2 (3,4) ---- (1,3) (1,4) (2,3) (2,4)

笛卡尔积的**子集**叫做**关系**，写作**R(D1,D2,...,Dn)**,为n元关系，n为关系的度/目。

由于域有可能相同，为了区分不同的域，为每一列起名，称为**属性。**n元关系即有n个属性。

关系模型对关系的限定和扩充：

限定：关系必须是**有限的、有意义的**

扩充：利用属性名的唯一性，**取消关系的有序性。**

> 同一关系在不同的属性名顺序下可有不同的表达顺序

关系的性质：

1.列是同质的：每一列的分量来自同一域，有同样类型

2.每列属性有不同的属性名，但可出自同一域

3.列的顺序与行的顺序都无关紧要 （这个算两点性质）

4.集合内任意两个元组**不可完全相同**

5.每一分量必须是**不可再分的数据**，满足**1NF**

> 二维表无法表示`对分量进行再分`。例如住址这一分量不能是`河南省`、`南阳市`两个数据，而应该为`河南省南阳市`单个数据。或者直接设定为`省`、`市`两个分量。

关系模式

用关系模型设置出来的一种结构就是关系模式。

关系模式还能确定为**关系实例**，关系实例是关系模式在**某一时刻**的状态或内容。

R（U，D，dom，F， I），R为关系名，U

为组成该关系的**属性名集合**，D为属性集U**所来自的域**，dom为**属性**向**域**的**映象集合**、F为属性间的**数据依赖关系**集合，I为**完整性约束**集合。关系模式通常可以简记作R(A1 , A2 ,..., An)，只写**R,U。**

关系数据库

关系数据库的型是关系模式的集合。（各个表的结构）

关系数据库的值是关系实例的集合。（具体的取值）

完整性约束

候选码：唯一性、最小性（删一个都不行）

> 候选码可能只有一个属性，也有可能包含所有属性。所有属性作为候选码时，称为**全码**。

主码：候选码中选一个。

主属性：**所有候选码**的属性的并。

非主属性：不出现在任何候选码的属性。

外部码：F是关系R的一个/一组属性，但**不是R的码**，而**与关系S**的主码**相对应**（F与S的主码必须定义在一个域）。**R为参照关系，S为被参照关系（目标关系）。**

> R和S未必是不同的关系。比如学生表，可有主码`学号`，也可有外码`班长学号`。`班长学号`可以与学生表的主码`学号`相对应。

实体完整性

关系R**主码**中所包含的**任意属性**都不能取空值（未知或无意义）。 reason: 主码要用来区分不同实体

参照完整性

若关系R的外码F与关系S的主码P相对应，那么关系R中每个元组的**F值必须为空**，或者**等于**关系S中**某个元组的P值。**
reason：可以尚未参照，若有参照，参照对象必须存在。

用户定义的完整性

用户针对具体的应用环境定义的完整性约束条件。如数值范围等。

系统自动支持**实体完整性**和**参照完整性**，并提供**定义和检验**用户定义的完整性的机制。

数据操作——关系代数

集合论中传统的集合操作：**集合并、集合交、集合差、广义笛卡尔积**

> 不能处理条件，比如分数低于60的元组
>
> 不能选择属性，比如只看元组的学号、课程名、成绩，不显示学生名字

> 不能选择性拼接多个表，比如课程表里的课程名与学生表里的学生名
>
> 不能做笛卡尔积的逆运算

专门的关系运算：**选择、投影、连接、商**

由于要处理条件，所以还会有比较运算符与逻辑运算符

基本运算

集合并、集合差、广义笛卡尔积、选择、投影

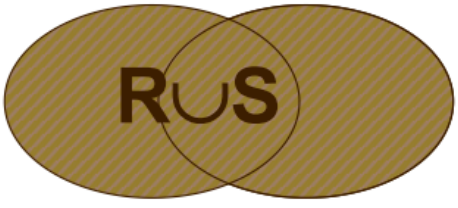
其他运算都可以由基本运算得出。

运算符		含义	运算符		含义
集合运算符	∪	并	比较运算符	>	大于
	∩	交		≥	大于等于
	−	差		<	小于
	×	广义笛卡尔积		≤	小于等于
				=	等于
				≠	不等于
专门的关系运算符	σ	选择	逻辑运算符	¬	非
	Π	投影		∧	与
	⋈	连接		∨	或
	÷	除			

运算的结果都是表，也就是**元组的集合。**

并运算

- 设R和S是n元关系，并且两者各对应属性的数据类型也相同。则R和S的并运算定义为：
  $R \cup S = \{ t \mid t \in R \vee t \in S \}$

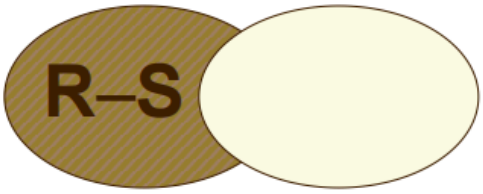


会去重。

差运算

- 设R和S是n元关系，并且两者各对应属性的数据类型也相同。则R和S的差运算定义为：

$$R-S=\{t \mid t\in R \wedge t\notin S\}$$

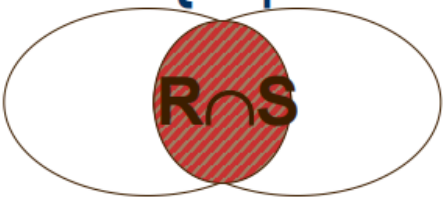


前面的去掉共有的。R-S与S-R不同。

交运算

- 设R和S是n元关系，并且两者各对应属性的数据类型也相同。则R和S的交运算定义为：

$$R\cap S=\{t \mid t\in R \wedge t\in S\}$$



- 其结果仍为n元关系，由既属于R又属于S的元组构成。
- 交运算可以通过差运算来重写：

$$R\cap S = R - (R - S)$$

广义笛卡尔积

$$R\times S=\{t \mid t=<r, s> \wedge r\in R \wedge s\in S\}$$

R

A	B	C
a	b	c
d	a	f
c	b	d

S

A	B	C
b	g	a
d	a	f

R×S

R.A	R.B	R.C	S.A	S.B	S.C
a	b	c	b	g	a
a	b	c	d	a	f
d	a	f	b	g	a
d	a	f	d	a	f
c	b	d	b	g	a
c	b	d	d	a	f

笛卡尔积的元组个数是原来两个关系的元组个数的乘积。

笛卡尔积的属性个数是原来两个关系的属性个数之和。

选择运算

在关系R中选择满足给定条件的元组。

$$\sigma_F(R)=\{t \mid t \in R \wedge F(t) = \text{‘真’}\}$$

F是选择的条件，取逻辑值“真”或“假”。

F的形式：由逻辑运算符连接算术表达式而成。

逻辑运算符：$\wedge$，$\vee$，$\neg$

算术表达式：$X\ \theta\ Y$

X，Y是属性名、常量、或简单函数。

θ是比较算符，$\theta \in \{>, \geq, <, \leq, =, \neq\}$

R

A	B	C
a	b	c
d	a	f
c	b	d

$\sigma_{B=b}(R)$

A	B	C
a	b	c
c	b	d

$\sigma_{B=b \wedge C=c}(R)$

A	B	C
a	b	c

投影运算

- 从关系R中取若干属性列组成新的关系。

$$\Pi_A(R) = \{ t[A] \mid t \in R \}, A \subseteq R$$

- 投影的结果中要去掉重复的行。

R

A	B	C
a	b	c
d	e	f
c	b	c

$\Pi_{B,C}(R)$

B	C
b	c
e	f

要去重。

连接运算

- 从两个关系的广义笛卡儿积中选取给定属性间满足θ操作的元组。

$$R \underset{A\theta B}{\bowtie} S = \{ t \mid t = <r, s> \wedge r \in R \wedge s \in S \wedge r[A]\ \theta\ s[B] \}$$

θ为算术比较符，当θ为等号时称为等值连接，其他依此类推。

- 连接运算可以用笛卡尔积和选择运算重写为

$$R \underset{A\theta B}{\bowtie} S = \sigma_{A\theta B}(R \times S)$$

笛卡尔积是强硬的拼接。经过选择即为连接。

若**θ为'='**，则为**等值连接。**

拼接后属性的个数等于原关系的属性数之和。

自然连接

若不写条件，则为自然连接。

自然连接相对于等值连接的区别：

1.不强调是哪个属性，而是取**属性名相同**且**取值相等**拼接为新元组

2.同名属性只保留一份

因此自然拼接后属性的个数未必等于原关系的属性数之和。

r

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

s

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	∈

$r \bowtie s$

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

> 对于多处属性名相同的情况，必须**都对应相等**才能视为取值相等

复合连接

对于自然连接的结果，直接**删去连接属性列**。

半连接

连接运算后只保留R或者S的属性列。

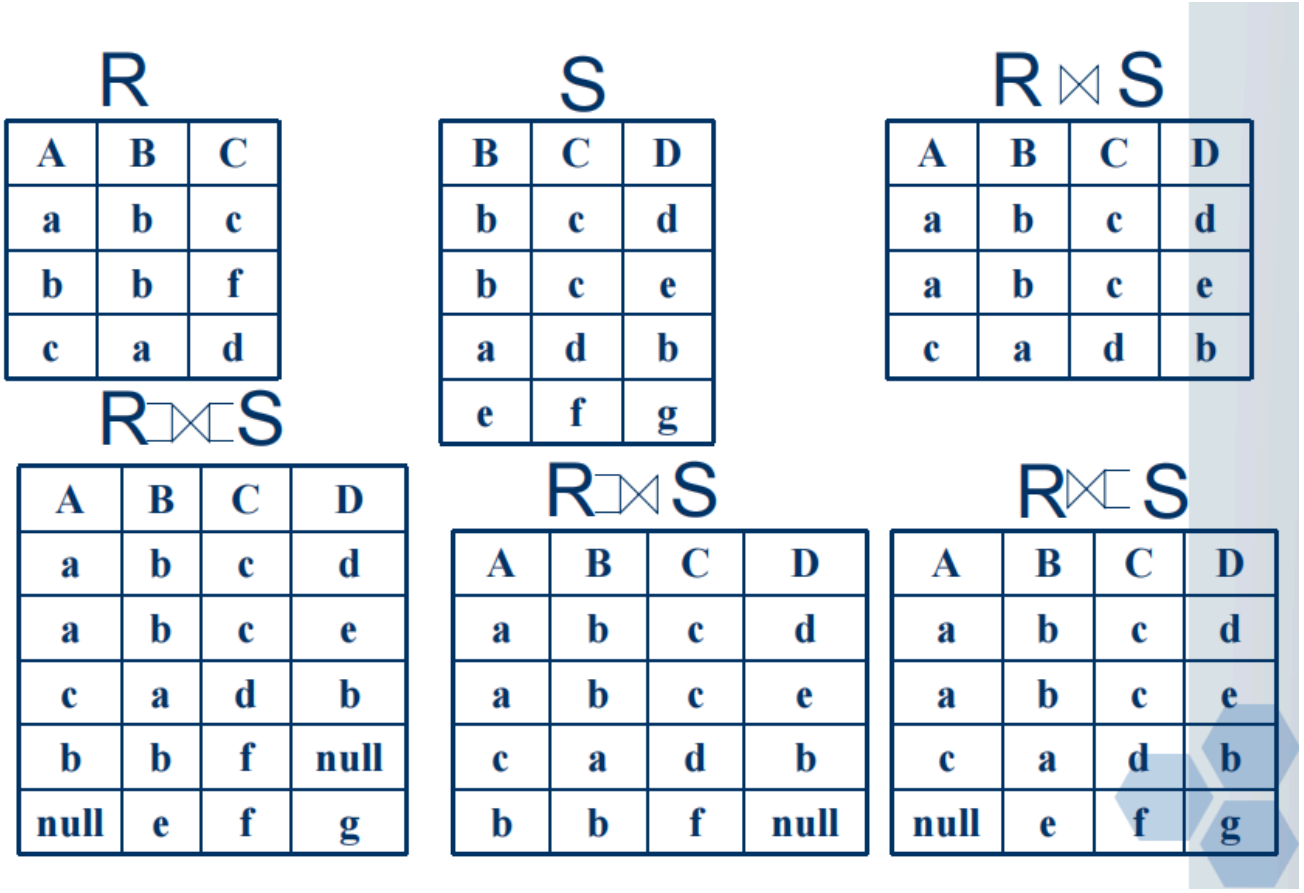
> 用于节省数据传输，仅提取数据用于条件判断，不必传整张表

外连接

目的：避免自然连接时因失配而发生的信息丢失。

未匹配的元组也仍存在于表中，只是多出来的属性取null。

> 左外连接 = 自然连接 + 左侧表中失配的元组
>
> 右外连接 = 自然连接 + 右侧表中失配的元组
>
> 全外连接 = 自然连接 + 两侧表中失配的元组



R

A	B	C
a	b	c
b	b	f
c	a	d

S

B	C	D
b	c	d
b	c	e
a	d	b
e	f	g

R ⋈ S

A	B	C	D
a	b	c	d
a	b	c	e
c	a	d	b

R⟕S

A	B	C	D
a	b	c	d
a	b	c	e
c	a	d	b
b	b	f	null
null	e	f	g

R⟖S

A	B	C	D
a	b	c	d
a	b	c	e
c	a	d	b
b	b	f	null

R⟗S

A	B	C	D
a	b	c	d
a	b	c	e
c	a	d	b
null	e	f	g

除运算

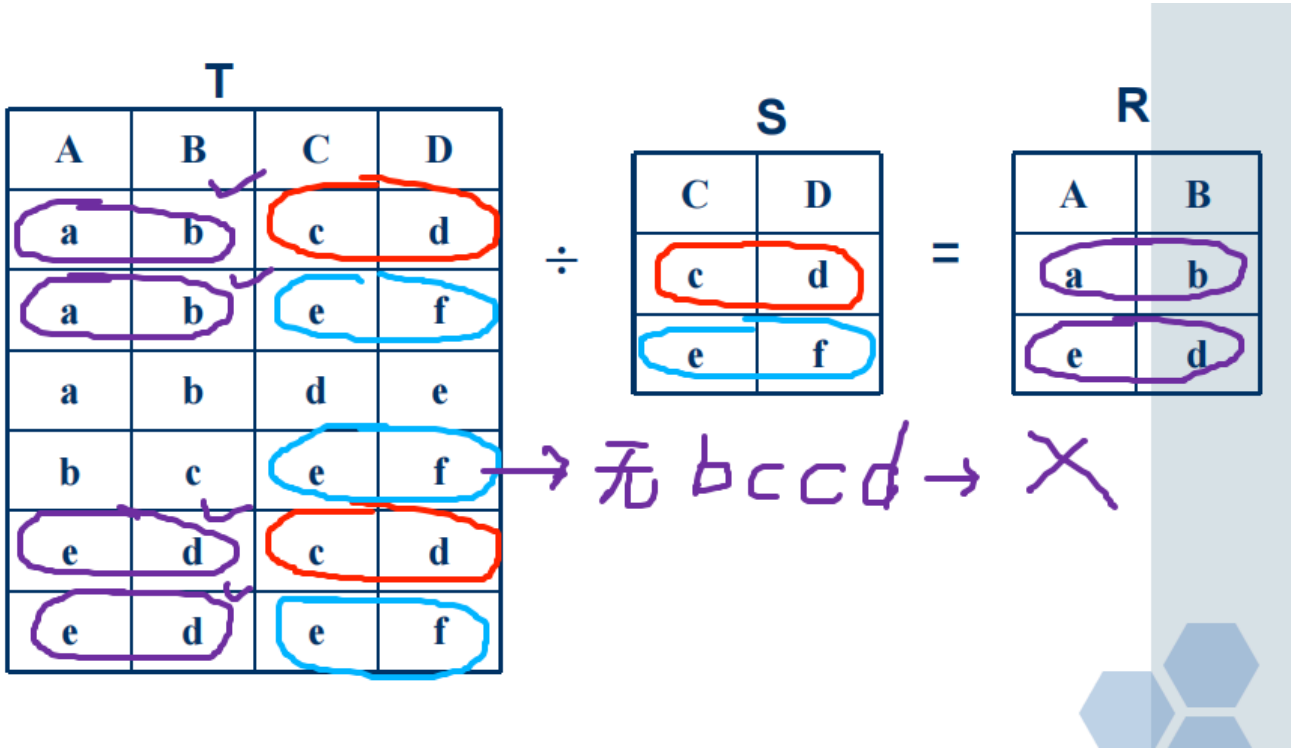
A÷B=C……D

则B×C+D=A

也即C的属性应为 A的属性**去掉**B的属性

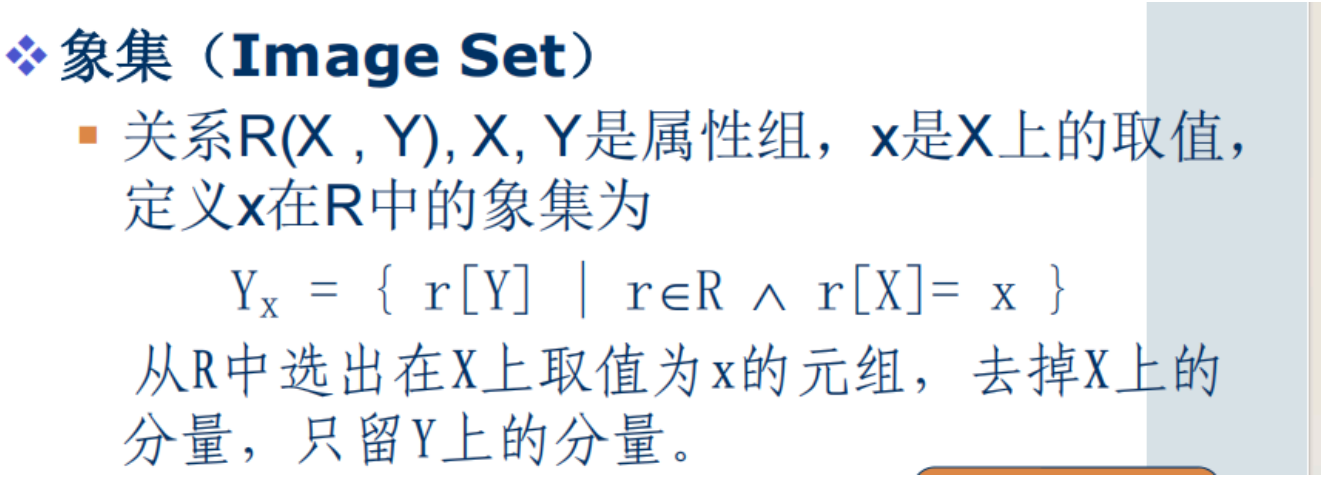
C应该填什么值？C中的值和B中的值一定能组合出A中的值！

那么需要考虑A中**与B中元组取值相同**的元组。



可以大胆先得到所有可能元组ab,bc,ed，然后再逐元组验证。

也可以利用象集：



对于T(CD,AB),

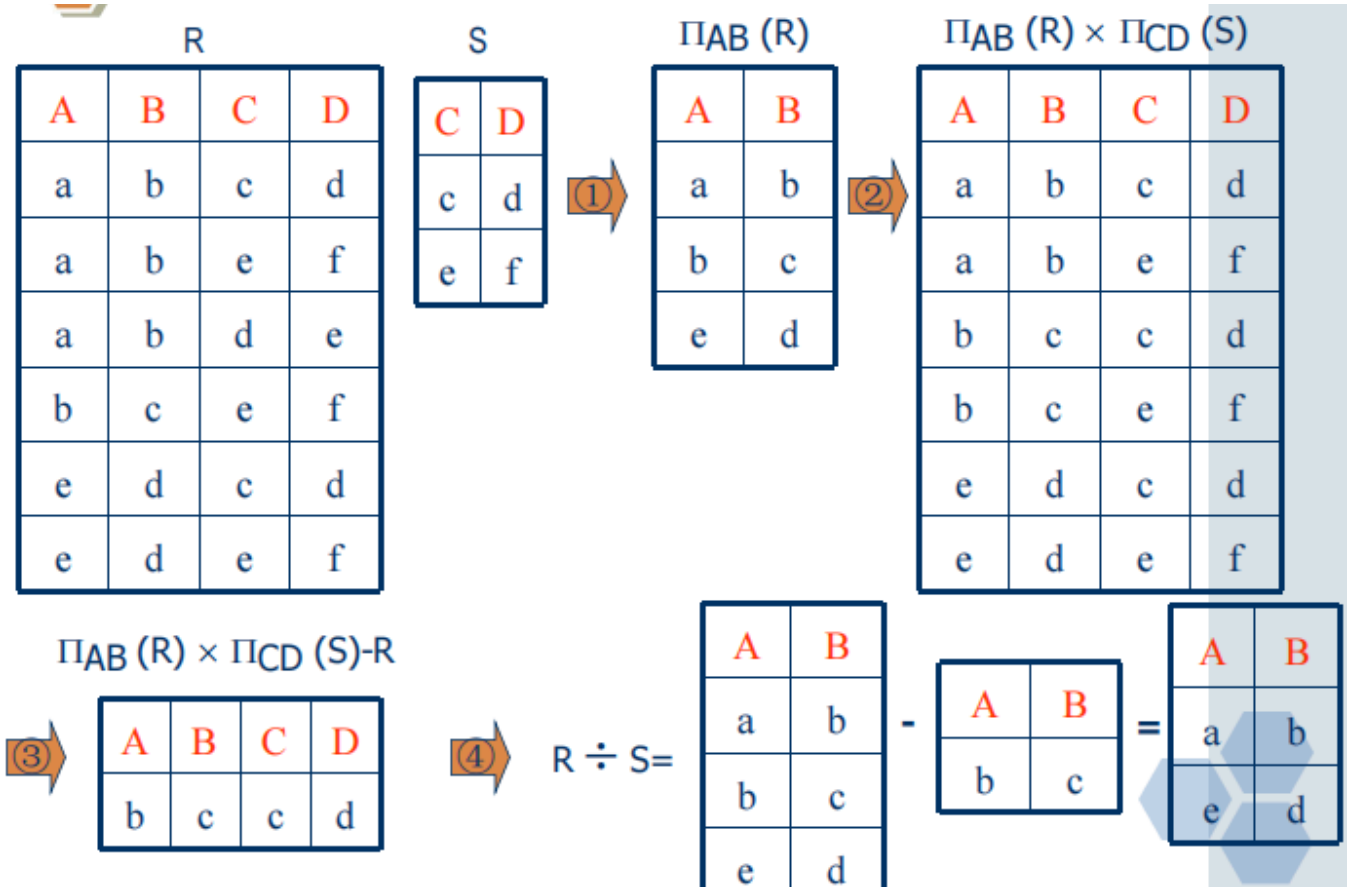
CD=cd时，象集为ab,ed

CD=ef时，象集为ab,bc,ed

对所有象集做广义交（在所有象集中都出现的元组）。

下面也是一种思路：

先列出所有AB的组合，并做笛卡尔积，得到理想被除数，然后减去真实被除数，可以得到欠缺的组合，从中提取欠缺的AB。最后并从所有AB组合中减去欠缺的AB，就是真正的商。



这也就得到了除法的这一定义：

$$R \div S=\{r[X] \mid r \in R \wedge \Pi_y(S) \subseteq Y_x\}$$

其中，

$$\Pi_y(S) \subseteq Y_x$$

就代表找到的AB组合一定能满足和除数做笛卡尔积还在R中，也就是除数包含于这个组合的象集。

象集的特殊含义：若把X作为对象A，那么X的象集就表示与对象A有关系R的**所有**对象B。

除法的特殊含义：得到的数据满足**S中所有要求**，也即与S中**所有数据**都满足关系R。

关系代数练习

简单的选择+投影

查找选修了1号课程或3号课程的学生的学号

$$\Pi_{Sno}\left(\sigma_{Cno=1 \vee Cno=3}(SC)\right)$$

或：$\Pi_{Sno}(\sigma_{Cno=1}(SC)) \cup \Pi_{Sno}(\sigma_{Cno=3}(SC))$

求选修了1号而没有选2号课程的学生号。

$\Pi_{Sno}(\sigma_{Cno=1}(SC)) - \Pi_{Sno}(\sigma_{Cno=2}(SC))$

求选修了1号和2号课程的学生号。

$\Pi_{Sno}(\sigma_{Cno=1}(SC)) \cap \Pi_{Sno}(\sigma_{Cno=2}(SC))$

多表连接后选择+投影

求选修了1号课程的学生号和姓名。

$\Pi_{Sno,\ Sname}(\sigma_{Cno=1}(Student \bowtie SC))$

可只选择表的一部分（关联部分+使用部分）后进行连接。

比如STUDENT预先只投影出关联部分Sno与使用部分Sname

> Cpno=5也可以最后再选择，但是连接操作cost很大，预处理后再连接比较好。
>
> Course甚至也可以先投影只留下Cno和Cpno

查询至少选修了一门其直接先行课为5号课程的学生姓名。

$\Pi_{Sname}(\sigma_{Cpno=5}(Course) \bowtie SC \bowtie \Pi_{Sno,\ Sname}(STUDENT))$

除法！

先求学号，之后关联就能求姓名。

学号怎么求？ 满足和**所有课程**都有**选修关系**的**学生学号**，用**除法**！

需要注意**属性的选择。**这里被除数不应包含其他属性。例如被除数包含`成绩`属性，那么就会要求`（学号，成绩）`的二元组与所有课程都有选修关系，也就是选了所有课的**且成绩都相同**选课记录，这是错误的。

查询选修了全部课程的学生号和姓名

$\Pi_{Sno,\ Cno}(SC) \div \Pi_{Cno}(Course) \bowtie \Pi_{Sno,\ sname}(STUDENT)$

数据操作——SQL语言

特点

1.综合统一，集**数据定义**，**数据操纵**，**数据查询**和**数据控制**于一体

2.高度非过程化（关系操作的特点）

3.面向集合的操作方式（关系操作的特点）

4.以同一种语法结构提供**两种使用方式**，**SQL既是自含式语言，又是嵌入式语言**

5.语言简洁，易学易用

SQL功能	操作符
数据查询	SELECT
数据定义	CREATE，DROP，ALTER
数据操纵	INSERT，UPDATE，DELETE
数据控制	GRANT，REVOKE

外模式、模式和内模式都可以用SQL管理。

模式：**基本表**是本身独立存在的表，一个（或多个）基本表对应一个存储文件。

内模式：**存储文件**由数据块构成的存储空间，用于存储**基本表、索引**等。

外模式：**视图**是从一个或几个基本表中导出的表，其本身不独立存储于数据库中。

数据查询

Select [ALL|DISTINCT] <目标表达式>[{,<目标表达式>}]
From <表名或视图名>[{,<表名或视图名>}]
[Where <条件表达式>]
[Group By <分组列> [{,<分组列>}] [HAVING <条件表达式>]]
[Order By <排序列> <排序方式>[{,<排序列> <排序方式>}]

ALL|DISTINCT

SQL不会自动去重，如果需要去重，需要加上DISTINCT

> 一般在筛选结果为非主码时，都加一下DISTINCT

目标表达式

目标表达式是**表达式**，也可直接**缺省为***表示所有属性，也可以不是/不包含表中的属性

select * from Student;

select **'北航'**, toLower(Sname), **2025-Sage** from Student;

加空格后面可**附上属性别名**:

select '北航' **school**, 2025-Sage **birth_date** from Student;

聚集函数也是目标表达式。

> Count() 求元组个数
>
> Sum() 对数值列求总和
>
> Avg() 求数值列的平均值
>
> Max() 求最大值
>
> Min() 求最小值

Select Count(*) student_count From Student;

Select Count(**Distinct** Sno) From SC;

条件表达式

where + 条件表达式

查询条件	谓词
比　　较	>, >=, <, <=, =, !=, <>, !>, !<; NOT +上述比较运算符
确定范围	BETWEEN AND, NOT BETWEEN AND
确定集合	IN, NOT IN
字符匹配	LIKE, NOT LIKE
空　　值	IS NULL, IS NOT NULL
多重条件	AND, OR

LIKE: 支持通配符`%`(通配零到多个字符)，`_`(仅通配一个字符)

IN可以作为同类OR的简写，如A=1 OR A=2可写为A IN (1,2)

> 注意A IN (1,2)可不是A BETWEEN 1 AND 2的意思

`IS NULL` 而非 `= NULL` ，因为说 `NULL = NULL` 是不合理的

> 注意若NULL参与计算，要把NULL转为其他值再进行计算，避免数据污染

Order By

可选择**升序ASC**与**降序DESC**

顺序代表排序优先级

Group By

聚集函数的结果总是只有一行。因为默认大家是一个大group.

我们可以先对某变量进行group，聚集函数就会根据新的group进行成批运算。

Select **Cno, Count(Sno)** From SC

Group By Cno;

这样就会算出每个Cno Group的Count(Sno)

> 要求：若使用group,目标表达式只能包含聚集函数与group的对象。因为只有这两类表达式能保证以group为单位，而不会出现一个group对应多值。

查询选课至少为三门的学生的学号及其选课门数：

Select Sno, Count(Sno) From SC

Group By Sno **Having Count(Sno)>=3**;

对聚集函数统计结果进行筛选**只能使用HAVING**!

因为**WHERE子句比GROUP BY先执行。**

查询 **至少有三门课选课并及格** 的学生的学号及其**及格选课**门数：

Select Sno, Count(Sno) From SC

WHERE Grade >= 60

Group By Sno **Having Count(Sno)>=3**;

多表查询

直接写from多个表即可。

若有属性名相同，需要强调 `表名.属性名`

直接在WHERE子句应填写**连接条件**，不填写则默认**笛卡尔积。**

连接条件：[<表名> .] <列名> <比较运算符> [<表名> .] <列名>

> 推荐新版写法：JOIN ON语句，把连接条件单列出来
>
> from 表1
>
> JOIN 表2 ON .连接条件

查询每个学生及其所选课程的情况

- Select Student.*, SC.* From Student, SC
  Where Student.Sno = SC.Sno;

表名后也可以加空格进行更名。

查询每一门课的间接先修课（即先修课的先修课）

- Select FIRST.Cno, SECOND.Cpno
  From Course FIRST, Course SECOND
  Where FIRST. Cpno = SECOND. Cno

外连接：所有学生

查询所有学生的学号、姓名、所选修课程的课程号和成绩

- Select Student.Sno, Sname, Cno, Grade
  From Student, SC
  Where Student.Sno=SC.Sno(*)

新版本表达（推荐）：

from Student **LEFT (OUTER) JOIN** SC ON ...

同理还有RIGHT与FULL

嵌套查询

不相关子查询：由内向外，将**查询语句的返回值**作为**查询条件的条件值。**因为查询语句的返回值是集合，我们往往使用**IN**运算符而非等号，或者**< ANY()**，**<ALL()**这种，any为存在，all为任意，也可以是**EXIST()**，若返回值为空集则为**FALSE。**

相关子查询：子查询进行时需要父查询提供信息。

`查询与刘晨在同一个系学习的学生`：先找到刘晨在哪个系

> 这里是特殊情况，使用=

```
Select Sno, Sname, Sdept
From Student
Where Sdept =
                (Select Sdept
                 From Student
                 Where Sname = '刘晨' );
```

这是一个不相关子查询。

查询选修了课程名为‘信息系统’的学生的学号和姓名。

这是一个典型的连接查询，但我们也可以用嵌套查询。

先求出课程名为'信息系统'的课程号，

再求选修了该课程号的学生的学号，

最后求该学号对应的学号和姓名。

```
Select Sno, Sname From Student
  Where Sno IN
      (Select Sno From SC
       Where Cno IN
           (Select Cno From Course
           Where Cname = '信息系统' ));
```

查询选择了1号课程的学生姓名。

```
Select Sname From Student
Where Exists
      (Select * from SC
      Where Sno=Student.Sno And Cno='1')
```

这是一个**相关子查询。**

外层Student表也参与了子查询。执行逻辑是遍历父查询的每一个元组，对每个元组计算Student.Sno，这个值参与子查询的运算，从而决定父查询是否包含该元组。

查询选修了全部课程的学生姓名

除法操作，表示与某属性所有值均有联系：

```
Select Sname From Student
  Where Not Exists
    (Select * from Course
      Where Not Exists
        (Select * from SC
          Where Sno=Student.Sno
            And Cno=Course.Cno));
```

这是一个相关子查询。

双重相关子查询：找的是学生，所以遍历student，再对每个student遍历course，看是否每个都存在。

不存在 没匹配上的课程，就是选修了所有课程。

如果改为EXIST NOEXIST，则表示未选修全部课程。

如果改为EXIST EXIST，则表示选修过课程。

如果改为NOEXIST EXIST，则表示未选修过课程。

集合查询

查询选修了1号或2号课程的学生

```
Select Sno From SC Where Cno='1'
Union
Select Sno From SC Where Cno='2'
```

对语句做**并**操作。要求查询结果字段结构一致。

交操作与差操作没有集合查询功能。

查询选修了1号课程的学生和选修了2号课程的学生的交集

```
Select Sno From SC
   Where Cno='1'
      And Sno In (Select Sno From SC
         Where Cno='2')
```

差集就是排除，只需要把第二个条件取反，或者用NOT。

查询计算机系的学生与年龄不大于19岁的学生的差集

```
Select * from Student
   Where Sdept = 'CS' And Sage > 19;
```

数据更新

Insert -- Into

> 不是add

```
Insert Into Student
  Values ('95020', '陈冬' , 'IS', 18);

Insert Into SC (Sno, Cno)
  Values('95020', '1');

Insert Into Dept_Age (Sdept, Avgage)
   Select Sdept, AVG(Sage) From Student
   Group By Sdept;
```

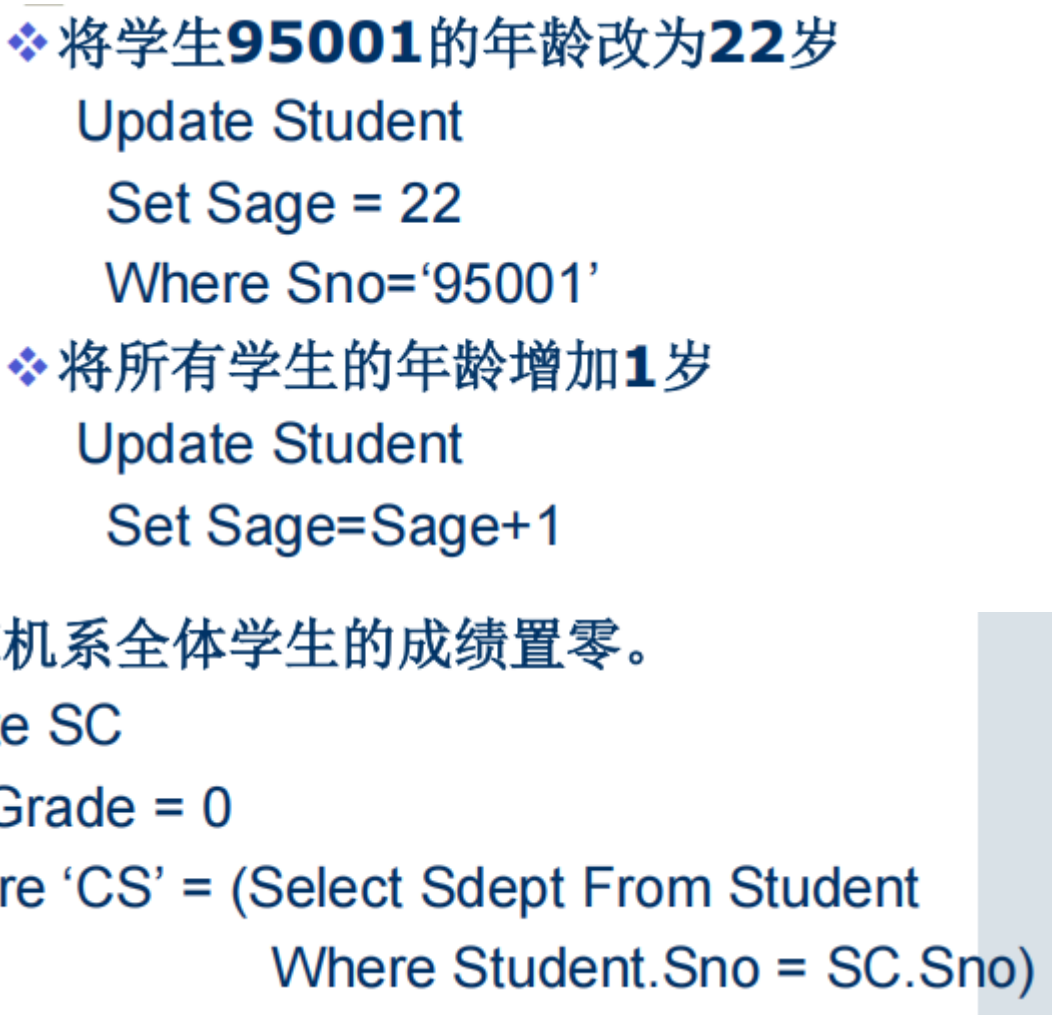
如果只insert了一部分属性，其余属性值**默认为null。**

Update -- Set

> 不是modify

支持原值引用。

子查询的值可以用在**WHERE**语句与**SET**语句。



Delete

delete from 表名 会直接删掉该表所有元组。

数据定义

操作对象	操作方式		
	创建	删除	修改
表	Create Table	Drop Table	Alter Table
视图	Create View	Drop View	
索引	Create Index	Drop Index	

基本表——定义、删除、修改

create

Create Table <表名>

(<列名><**数据类型**>[<**列级**完整性约束>]

[{, <列名><数据类型>[<**列级**完整性约束>]}]

[{, [<**表级**完整性约束>]}]**)**

数据类型常用：char(n)、int、DECIMAL(总位数, 小数位数)、date、time

完整性约束：NULL/NOT NULL（列级）、UNIQUE（**列级**、候选码，可以取空值）、PRIMARY KEY（**表级**、主码，不可取空值）、FOREIGN KEY（**表级**）、CHECK（**表级**）

单一属性的非外码的表级完整性约束也可以只写在对应列级上。

```
Create table student
 (Sno char(5) not null,
 Sname char(20) not null unique,
 Ssex char(2),
 Sage Int,
 Sdept char(15),
 Primary key(Sno),
 Check (sage >=18 and sage <=45));
```

外码的书写需要标明references 表(主码)

Create table SC

(Sno char(5) not null,
Cno char(5) not null,
Grade number(4,2),
Primary key (Sno, Cno),
Foreign key (Sno) references Student(Sno)
Foreign key (Cno) references Course(Cno));

alter

Alter Table <表名>

Add(<新列名><数据类型>[<完整性约束>])
Drop <完整性约束名>
Modify(<列名><数据类型>)

drop尽量不要删除属性列，其实是可以删的。

modify尽量不要改列名，虽然其实也可以改。

向Student表增加“入学时间”列

- Alter Table Student Add (Scome Date);

将Sage的数据类型改为半字长整数

- Alter Table Student Modify (Sage Smallint)

删除Student表的主键

- Alter Table Student Drop Primary Key

drop

直接 drop table 表名即可删除整个基本表。分配的空间也回收了。

索引——定义、删除

```
Create [Unique][Cluster] Index <索引名>
  On <表名>
  (<列名>[次序][, <列名>[次序]] ...);
```

unique是**唯一性约束**，cluster要求索引与基本表在**物理上连续存储**，便于查询。

cluster索引只能建一个。因为**物理上**只能有一个连续存储。

索引可以**提升查询速度**，需要**单独创建文件**，占**空间小**、元组**有序。**

但是会**降低更新速度**，每次更新还要更新索引表，还要重新排序。

> 不能说索引**越多越好**，也不能说索引会提高**操作**效率

也是create和drop

```
Create Unique Index Stusno
   On Student (Sno);
Create Unique Index Scno
   On Sc (Sno ASC, Cno DESC)
Drop Index Stusno;
```

index的drop**不会丢失数据，不必备份。**

索引的**排序顺序对我们没有意义**，是DBMS操作的。

索引与where子句、聚合函数高度相关，索引**便于筛选**，并且**预先计算好统计数据。**

双关键字排序时，第二个关键字整体还是乱序。

消除碎片：把所有index删掉再重建，DBMS重新集中分配空间。

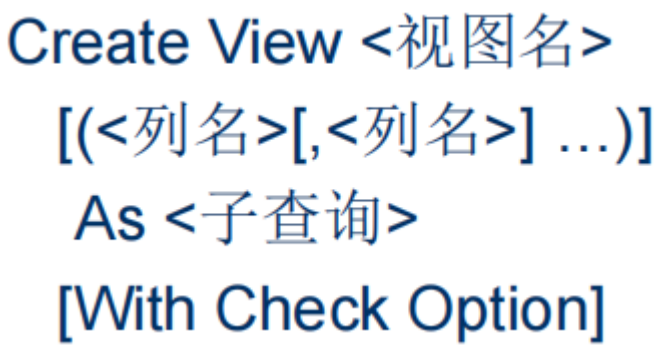
视图——定义、删除

视图被称为**虚表**，**没有存储数据**，而是存储**select语句。**

> 能够**简化用户操作**, 使用户能够以**多种角度**看待同一数据, 对重构数据提供了一定程度的**逻辑独立性**, 能够对机密数据提供**安全保护**。

直接利用视图**封装**好复杂的select语句。

我们只提供视图，就既能提供功能，还能保护数据。



手写列名可用于更名。

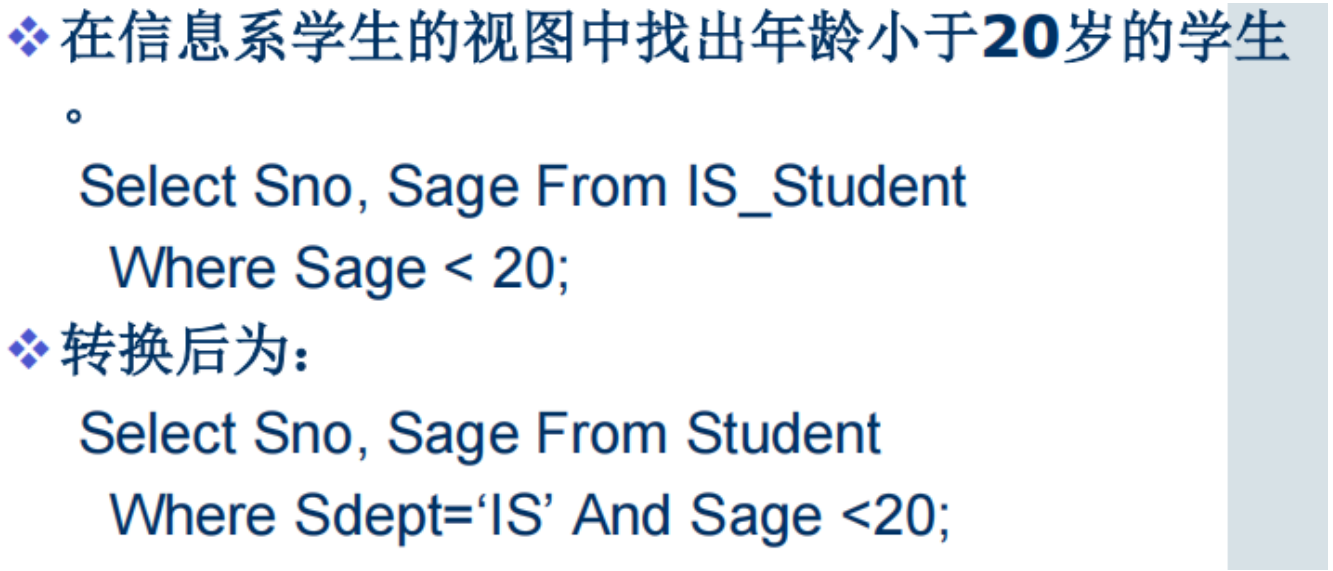
子查询可以是单表查询，多表查询，嵌套查询，集合查询……

视图可以嵌套！子查询语句可以是对另一视图的select。

drop view 视图名即可直接删除，**不会丢失数据，不必备份。**

视图消解

DBMS会**尝试**把程序员发起的SQL指令中的视图拆为select，并进行语句合并，融合为一个select进行。



视图更新

视图也可以充当**范围受限的基本表**使用。

`insert into 视图 values(...)` 也能**实现修改基本表。**

values可以包含视图中没有提供的属性，这可能触发错误。可开启**With Check Option**，使其在Insert与Update的时候也会做检查。

视图更新约束

要借助视图进行insert时，视图自身需满足一定要求。

1. select子句中的目标列不能包含**聚集函数**
2. select子句中不能使用**distinct关键字**
3. 不能包括**group by**子句

4. 不能包括经**算术表达式**计算出来的列
5. 必须包含基本表的**主码**（否则插入的元组主码为空）

对于行列子集视图可以更新（视图是从**单个基本表使用选择、投影操作导出**的，并且**包含了基本表的主码**）

数据库安全性与完整性

计算机系统的安全性是指为计算机系统建立和采取的各种安全保护措施，以保护计算机系统中的硬件、软件和数据，防止其因偶然或恶意的原因使系统遭到破坏，数据遭到更改或泄漏等。

技术安全类（计算机系统及其数据受到攻击，如增添丢失泄露）、管理安全类（意外故障、管理不善、物理破坏丢失）、政策法律类

安全性

安全威胁

1. 造成对数据库内存储数据的**非授权访问-读取**，或**非授权的写入-增加、删除、修改**
2. 影响**授权用户以正常方式**使用数据库系统的数据服务

具体分类：

1.**偶然地、无意地**接触或修改DBMS管理下的数据

2.**蓄意的侵犯和敌意的攻击**

定义

数据库的安全性是指**保护数据库**以防止**不合法的**使用所造成的**数据泄漏、更改和破坏。**它包括两个方面的含义：向**授权用户提供可靠**的信息服务。同时，**拒绝非授权的**对数据的**存取访问**请求，保证数据库管理下的数据的**可用性、完整性和一致性**，进而保护数据库**所有者和使用者**的**合法权益。**

安全控制

外部安全控制

外部安全控制解决内部安全控制**不能解决**的问题。

> 偏管理

实体安全控制、**人员、组织安全**控制、**过程安全**控制

内部安全控制

内部安全控制由计算机系统的**软硬件**实现。它必须与管理系统物理存取的适当的外部安全控制相配合。

> 偏技术

数据库本身提供的安全机制：

用户标识与鉴别

采用用户名与口令，是**系统提供的最外层安全保护措施**

存取控制

合法用户能按照指定权限使用，非法用户、无权用户不能

- 定义用户权限
- 合法权限检查

存取控制可以分为

自主存取控制（DAC)：更自由，用户对不同数据对象有不同权限，可转授权限。

用户权限分为**数据对象**和**操作类型**两个要素。

	数据对象	操作类型
数据库	模式	建立、修改、检索
	基本表	建立、修改、检索
	视图	建立、修改、检索
	索引	建立、删除
数据	表或视图	查找、插入、删除、修改
	属性列	查找、插入、删除、修改

用户级权限：一般由**数据库管理员**为每个用户授权，与整个数据库相关。

关系级权限：由**数据库管理员或数据库对象拥有者**授权，基本是表义级权限。

> 对权限与用户打包：
>
> **角色**：**权限**的集合，可以授予给**用户**或其他**角色**。
>
> **用户组**：**用户**的集合，可以**以用户组为单位**授权与收回权限

用户级权限与角色

授权与收回权限：

```
Grant <用户级权限>|<角色> [{,<用户级权限>|<角色>}]
    To <用户名>|<角色>|public [{,<用户名>|<角色>}]
    [With Grant Option]

Revoke <用户级权限>|<角色> [{,<用户级权限>|<角色>}]
    From <用户名>|<角色>|public [{,<用户名>|<角色>}]
```

public 指数据库中的**全部用户**。

With Grant Option则允许被授权的用户将指定的用户级权限或角色**授予其他用户**。

多重授权：A->C B->C 若A被revoke,C仍有权限。

循环授权：A->B B->C 若C主动->B，会导致A被revoke,C仍有权限。现在已被禁止。

Oracle的角色：

拥有的权限	操作			
	Create User	Create Schema	Create Table	登录数据库，执行查询和更新
DBA	√	√	√	√
RESOURCE	×	×	√	√
CONNECT	×	×	×	需要授权才可以执行

没有授权CONNECT甚至没有登录权。

DBA拥有全部权限。

关系级权限

对于具体的表格与视图的权限控制。

多了**ON子句针对具体的表/视图。**

```
Grant ALL| <权限> [{,<权限>}]
  On <表名> | <视图名> [{,<表名> | <视图名>}]
  To {<用户> [{, <用户>}] | public}
  [With Grant Option]
```

可以**利用视图**实现**安全控制。**

强制存取控制（MAC）：中央集中控制，用户与数据对象被赋予固定的许可证级别与密级。

主体的敏感度标记称为**许可证级别**，客体敏感度标记称为**密级。**

- **许可证级别 >= 密级**：可以**读**
- **许可证级别 == 密级**：可以**写**，避免滥用权力

审计

把用户对数据库的**所有操作都自动记录**下来放入审计日志中。DBA可以利用审计跟踪的信息，**重现**导致数据库现有状况的一系列事件，找出非法存取数据的人、时间和内容等。

数据加密

防止数据库中数据在**存储和传输中失密**的有效手段。加密的基本思想是根据一定的算法将**原始数据（明文）变换为不可识别的格式（密文）**，从而使得不知道解密算法的人无法获知数据的内容。

统计数据库安全性

统计数据库中的数据分为两类，一类是**微数据**描述现实世界的实体、概念或事件的数据；另一类是**统计或综合数据**，是对微数据进行综合处理而得到的结果数据。

统计数据库只为用户提供统计数据，不允许用户访问微数据。但微数据有时可以通过一组统计数据推导出来。统计数据库安全性的目的就是防止用户**访问或推导**出统计数据库的微数据。

解决方案：

> 先用where语句访问count，count为1时的统计数据就是微数据！

1. 控制统计结果的大小

> 访问A+其他人，B+其他人，相减得到AB差值。

2. 禁止重复执行统计查询
3. 统计结果加入噪声 （越查询误差越大）

完整性

数据库的完整性是指数据的**正确性**和**相容性。**

正确性：合法的类型、有效的取值范围

相容性：同一事实的两个数据应当自洽。

完整性措施的防范对象是**不合语义的数据。**

完整性约束条件

施加在数据库数据之上的**语义约束条件**称为数据库完整性约束条件。

作用的对象可以是**关系、元组、列**三种。

- 列约束主要是列的**类型、取值范围、精度**等约束条件。
- 元组的约束是元组中**各个字段间联系**的约束。
- 关系的约束是若干**元组间、关系集合上以及关系之间**的联系的约束。

又可分为**静态约束**和**动态约束。**

静态约束

静态约束是指数据库**每一确定状态**时，数据对象所应满足的约束条件，它是反映数据库状态合理性的约束。

静态列级约束是对一个列的**取值域**的说明。

静态元组级约束是对**一个元组的各个列之间的约束关系。**

> 属性A为x的元组需要满足属性B大于100

静态关系约束：**实体完整性约束、参照完整性约束、函数依赖、统计约束**

> 统计约束：对于属性A为x的元组，属性B的值必须高于其平均值。

还可分为：

固有约束：数据**模型**固有约束。如属性原子化

隐含约束：数据**模式**隐含约束，DDL语句说明（如实体完整性约束等建表时就确定的约束）

显式约束：人为显示定义的语义与应用层约束

隐含约束

```
Create table SC
(Sno char(5) not null,
Cno char(5) not null,
Grade number(4,2),
Primary key (Sno, Cno),
Foreign key (Sno) references Student(Sno)
Foreign key (Cno) references Course(Cno));
```

- **实体完整性与其他DDL定义**约束：

  Insert 或者Update时检查，违反则取消操作

- **参照完整性约束**：

  Insert、Update、**Delete**时检查

被参照表	参照表	违约处理
可能破坏参照完整性	插入元组	拒绝
可能破坏参照完整性	修改外码值	拒绝
删除元组	可能破坏参照完整性	拒绝/级联删除/设为空值
修改主码值	可能破坏参照完整性	拒绝/级联更新/设为空值

级联删除/更新：直接把参照表的对应记录也删除/更新

显示完整性约束

- **过程**（程序员为每个约束编制一个验证过程）
- **断言**
- **触发器**（不同DBMS写法不同 `:new.属性名` `:old.属性名`）

> 断言表示数据库状态**应当满足**的条件，
>
> 而触发器中的条件却是**违反约束**的条件

动态约束

动态约束是指数据库从一种状态转变为另一种状态时，**新、旧值之间**所应满足的约束条件，它是反映数据库状态变迁的约束。

动态列级约束是修改**列定义**或**列值**时应满足的约束条件。

> 某属性的值只能增加不能减少
>
> 不能在已包含null的列上增加not null约束

动态元组级约束指修改元组值时元组中各个字段间需要满足的约束。

> 属性A为x的元组需要满足属性B只能增加

动态关系约束：多元组之间等等。

完整性控制机制

三个功能：**定义功能、检查功能、违约响应。**

检查时机：

立即执行约束是指在执行用户事务的过程中，在**一条语句执行完**后立即进行完整性约束的检查。若违背了完整性约束，系统将**拒绝该操作。**

延迟执行约束是指在**整个用户事务**执行完毕后，再进行完整性约束的检查。若违背了完整性约束，系统将**拒绝整个**事务。

一条完整性规则可以用一个**五元组（D，O，A，C，P）**来描述，其中：

D（Data）约束所作用的**数据对象**

O（Operation）触发完整性检查的**数据库操作**，即当用户发出**什么操作**请求时需要检查，是**立即检查还是延迟检查。**

A（Assertion）数据对象必须满足的断言或语义**约束。**

C（Condition）选择A作用的数据对象值的**谓词**（条件）

P（Procedure）违反完整性规则时**触发的过程。**

关系数据理论

如何判断关系模式的优劣？如何优化关系模式?

关系数据理论借助于**数学工具**规定了一套关系数据库设计的理论和方法。

基于**属性之间的联系**进行设计，这种联系叫做**数据依赖。**

函数依赖

数据依赖中最重要的就是**函数依赖。**

在某关系模式中，若某属性组A的值确定后，另一属性组B的值也被唯一确定，则称A**函数决定**B，B**函数依赖**于A。

函数依赖：设R(U)是属性集U上的关系模式，X，Y 是U的子集。若对于R(U)的任意一个可能的关系r，r中不可能存在两个元组在X上的属性值相等，而在Y上的属性值不等，则称X函数决定Y，或Y函数依赖于X，记作X→Y。

- 函数依赖不随时间而变，永远成立。（不能仅凭当前的数据判断函数依赖）
- 函数依赖是**语义**决定的，是一种语义完整性约束，是一切关系均需满足的约束条件。

属性之间的联系分为1对1，1对多，多对多：

其中**X与Y**为**1对1**或**1对多**时，有**X函数依赖于Y**，**Y函数决定X**

> 学号与身份证号是1对1，名字与学号是1对多（存在重名）
>
> 都有学号**函数决定**身份证号和名字

多对多时，则不存在函数依赖。

根据表格初步判断函数依赖时，找**左端重复项**，对应**右端是否相同**即可。

分类

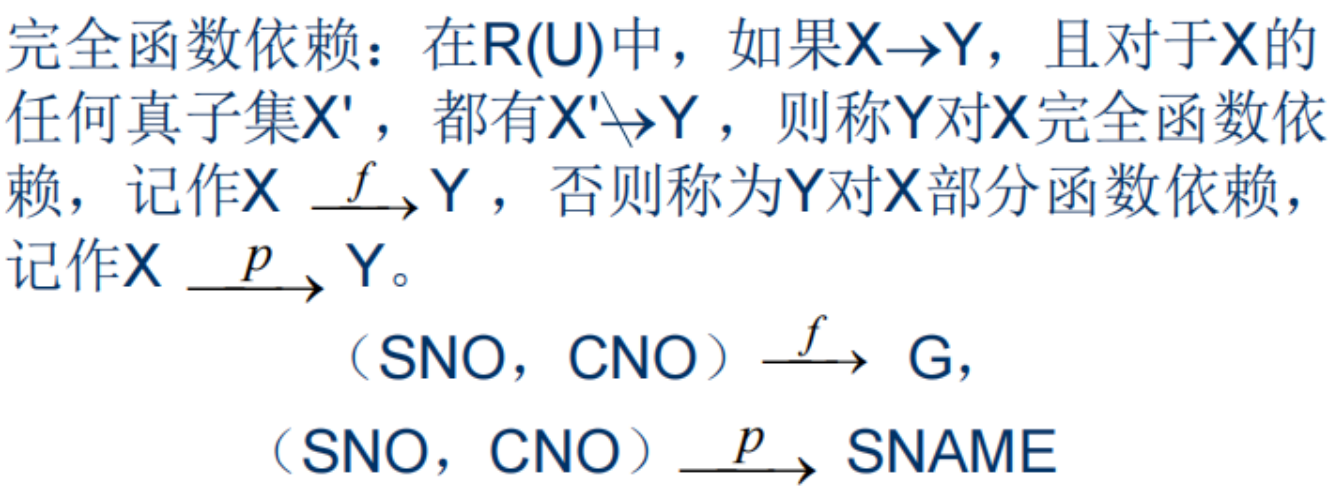
平凡函数依赖与非平凡函数依赖

平凡函数依赖：如果X → Y，但Y⊆X，则称其为非平凡的函数依赖，否则称为平凡的函数依赖。如（SNO，SNAME）→ SNAME是平凡的函数依赖

自己决定自己就是平凡的函数依赖。

平凡的函数依赖是**永真的**，意义不大。

完全函数依赖与部分函数依赖

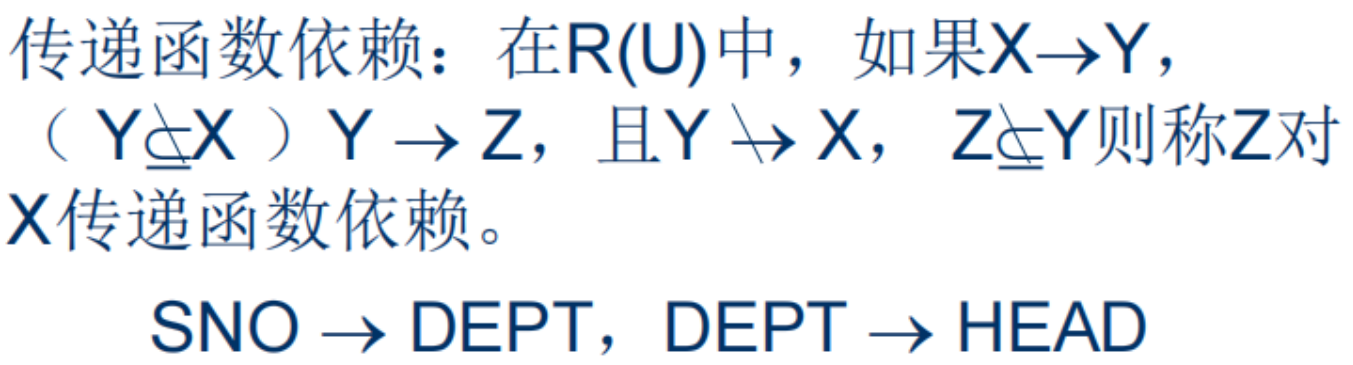


完全：**full** 部分：**part**

完全的函数依赖有**最小性**，决定者**不包含多余属性**，或者说删去任一属性就无法再决定。

> 上图中的例子就很经典：
>
> 完全函数依赖属性缺一不可，部分函数依赖则有冗余

传递函数依赖



1. X决定Y是**非平凡**的函数依赖
2. Y决定Z是**非平凡**的函数依赖
3. X与Y不是**一一对应**（1:1的属性联系，比如学号和身份证号）

码的规范定义

码有 *唯一性* 和 **最小性**。其实对应的就是 **完全** *函数依赖*。

候选码：设K为R< U , F >的属性或属性组合， 若K $\xrightarrow{f}$ U，则称K为R的一个候选码。若候选码多于一个，则选定其中一个作为主码。

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

超码不一定是码，而是广义的码。只要求函数依赖，不强调**完全**。

同样，主属性依旧是所有候选码中出现的属性的并集。

非主属性不包含在任何一个候选码。

全码是指**码由整个属性组构成**，也强调最小性。

> 不要混淆超码和全码。

范式

范式是对关系的不同**数据依赖程度**的要求。

> 范式即表示条件，也表示满足条件的关系构成的集合。
>
> 我们用 属于符号 来描述关系与范式。

1NF 2NF 3NF BCNF 4NF 5NF

> 本课程仅讲述1NF到BCNF。

一个低级范式的关系模式，通过**模式分解**可以转换为**若干个**高级范式的关系模式的集合，这一过程称作**规范化**。

范式等级越高越好？(X)

> 表数量增多，增加了关联操作的次数，开销也会更大。

1NF

关系中每一分量必须是**原子的，不可再分**。即不能以**集合、序列**等作为属性值。

这是关系六大性质之一。所以**只要是关系模式，都满足1NF**。

2NF

若R$\in$1NF，且每个非主属性完全依赖于码，则称R$\in$2NF

也就是**所有候选码**都**完全决定 所有非主属性**。

要成为2NF，关键是消除**非主属性**对码的**部分**依赖。

- 关系R的全体属性都是R的主属性时，也就是没有非主属性，没有反例，所以一定属于2NF
- R的所有候选码都只含一个属性时，一定是最小的，所以一定都是完全的函数依赖，一定属于2NF

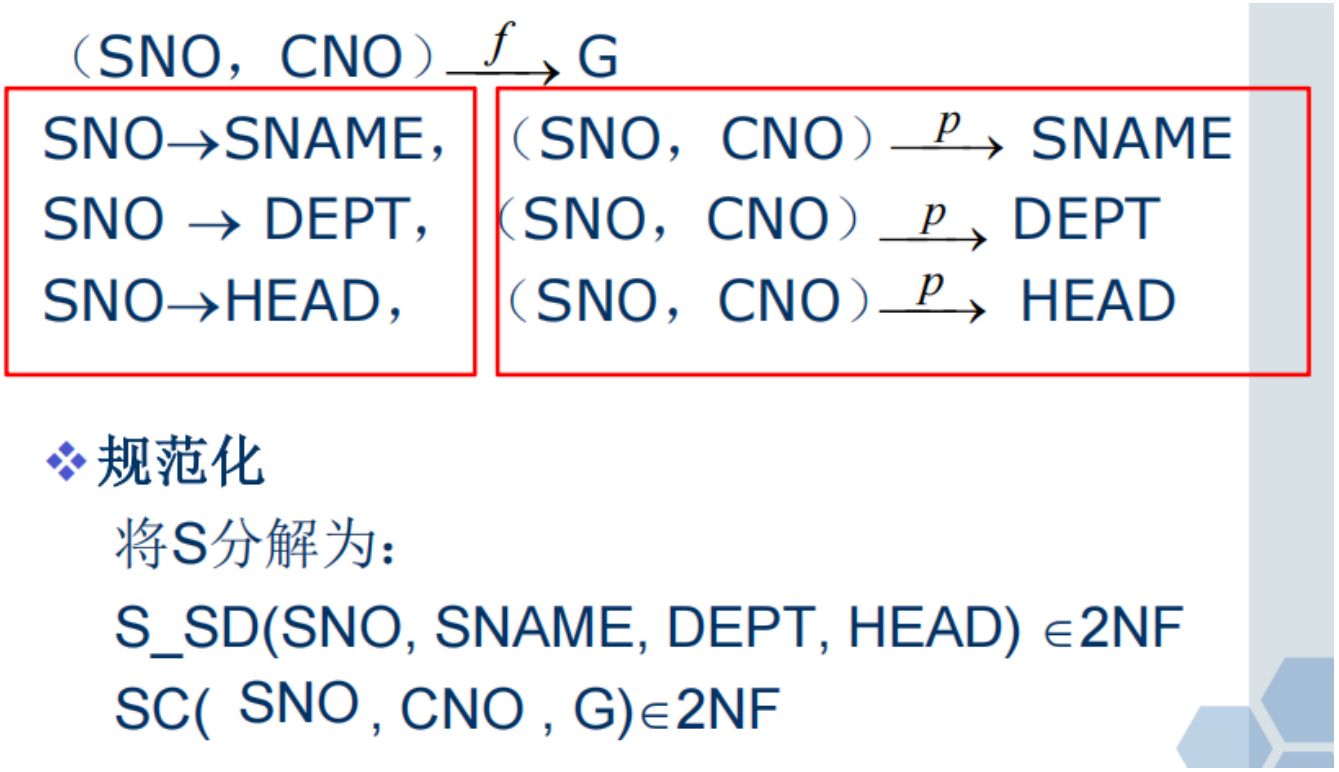
没有2NF时存在捆绑问题：

1. 插入异常：码中存在**多余属性**，在插入**不依赖于多余属性**的信息时，**多余属性没有值**，而主码不能为空，导致插入失败。
2. 删除异常：要想删除学生的**多余属性信息**，**整个元组**都需随之删除。
3. 数据冗余：在导入**多余属性相关信息**时，会重复插入相同的**不依赖于多余属性**的信息。
4. 更新异常：3的逆过程，数据冗余会导致修改时需要多次重复来保证数据一致性。

规范化为2NF的示例：

将**导致部分函数依赖**的**完全函数依赖**相关的**主属性与非主属性**拉出去单独成表。

原表若缺主码可以再填回主属性。



3NF

- 一种民间的定义：

若**R属于2NF**，并且不存在**非主属性对码的传递函数依赖**，则R属于3NF。

- **正式定义**：

不要求2NF的前提。

关系模式R< U , F >中，若不存在这样的码X，属性组Y及非主属性Z(Z⊈ Y)，使得下式成立，

$$X \to Y , Y \to Z , Y \not\to X$$

则称R∈3NF。

对比一下传递依赖的定义：

传递函数依赖：在R(U)中，如果X→Y，（ Y⊈X ）Y → Z，且Y ↛ X， Z⊈Y则称Z对X传递函数依赖。

$$SNO \to DEPT，DEPT \to HEAD$$

3NF去除了Y不是X的子集的限制。

- 如果Y是X的子集，那么Y->Z的意思就是**码的子集**能函数决定非主属性，这就是**部分函数依赖**的定义。因此可以排除部分函数依赖。
- 如果不是，那就是**传递函数依赖**的定义，可以排除传递函数依赖。

要成为3NF，关键是要消除2NF的关系模式中**非主属性对码**的**传递函数依赖。**

> 所有非主属性对码都有依赖，如果非主属性对码有了**传递函数依赖**，那就说明非主属性**内部存在函数依赖**。

- 关系R的只有一个非主属性时，内部必无非平凡依赖，一定属于3NF

没有3NF时存在捆绑问题：

同样是那四条。主要原因是非主属性内部存在函数依赖，而这种关系可以提取出来单独成表。

例如S_SD(SNO, SNAME, DEPT, HEAD)，DEPT与HEAD之间的函数依赖就应当单独成表，否则由于**SNO的主码限制**，不能单独插入、修改DEPT与HEAD的信息。

规范化为3NF的示例：

将**导致传递依赖**的**非主属性内部函数依赖**相关的**非主属性**拉出去单独成表。

新表的码放回原表当外码。

- $SNO \rightarrow SNAME, SNO \rightarrow DEPT$
- $SNO \xrightarrow{t} HEAD$
- 原因：$SNO \rightarrow DEPT$，$DEPT \rightarrow HEAD$

规范化

- 将S分解为：
  - DEPT(DEPT , HEAD)
  - STUDENT(DEPT , SNAME , SNO)

BCNF

R(A,B,C)有以下函数依赖：A->B (B,C)->A

候选码：(B,C) (A,C)

问题：

- 如果没有C，就无法描述A->B的关系。
- 因为C的不同产生的不同元组也会存入A->B的冗余信息。

若关系模式$R<U, F> \in 1NF$，若$X \rightarrow Y$，且$Y \not\subseteq X$时，X必含有码，则$R<U, F> \in BCNF$。

所有**非平凡函数决定者**必须是**码**（也就是候选码）。

> 也就是不允许码之外存在任何函数依赖。

- 只包含**两个属性**时一定是BCNF。（四种函数依赖情况都没有反例）

解决示例：

A->B，但是A不是候选码。

改造为R1(A,B)与R2(B,C)。只有两个属性一定是BCNF。

数据依赖的公理系统

对于满足一组函数依赖F的关系模式R< U, F >，若函数依赖X->Y成立，则称F**逻辑蕴涵**X->Y，也即F|=X->Y。

Armstrong公理

- 自反律。**平凡的**函数依赖为F所蕴含。
- 增广律。X->Y为F所蕴含，则(X,Z)->(Y,Z)也为F所蕴含。
- 传递律。X->Y，Y->Z为F所蕴含，则X->Z也为F所蕴含。

为F所蕴含的**所有函数依赖**的全体称为**F的闭包**，记作F^+。

- 有效性：由F出发根据Armstrong公理推导出来的函数依赖一定在F的闭包中。
- 完备性： F的闭包中的每一个函数依赖都可以由F出发根据Armstrong公理从F中导出。

由Armstrong公理导出的推理规则：

- 合并律。若X->Y，X -> Z，则X -> YZ。

  > 增广律：XX->XY, XY->YZ

- 分解律。若X -> YZ ，则X -> Y，X -> Z。

  > 自反律：YZ->Y，YZ->Z

- 伪传递律。若X -> Y，WY -> Z，则XW -> Z。

  > 也就是XW->WY->Z

求属性集的闭包

F有闭包，属性集也有闭包！

属性集X的闭包X_F^+是它在 F^+ 中**能函数决定**的属性的集合。

引理一：

$X \to A_1 A_2 \ldots A_k$成立 $\Leftrightarrow X \to A_i$成立$(i=1, 2, \ldots, k)$

引理二：

设F为属性集U上的一组函数依赖，$X, Y \subseteq U$，$X \to Y$能由F根据Armstrong公理导出的充分必要条件是$Y \subseteq X_F^+$

引理一就是合并律与分解律。

引理二显然，总之要判定一个函数依赖是否可推导、是否成立，我们需要求出决定者的闭包中有没有依赖者，总之要会算X_F^+。

Input：X，F

Output：X_F^+

步骤：

(1)令$X^{(0)}=X$，i=0

(2)求B，$B=\{A|(\exists V)(\exists W)(V\to W\wedge V\subseteq X^{(i)}\wedge A\in W)\}$

(3) $X^{(i+1)}=B\cup X^{(i)}$

(4)判断$X^{(i+1)}=X^{(i)}$？

(5)若相等，或$X^{(i+1)}=U$，则$X^{(i+1)}$就是X_F^+，算法终止。

(6)若否，则i=i+1，返回第二步。

把X放进闭包，然后从闭包里找F^+中的左端项，把所有对应右端项放进去。迭代直到闭包结果不变或达到全集。

逻辑很直观很简单。下面是一个例子。

R< U, F >, U = (A, B, C, D, E), F = {AB→C, B→D, C→E, CE→B, AC→B},计算$(AB)_F^+$。

所用依赖	$(AB)_F^+$
AB→C	ABC
B→D	ABCD
C→E	ABCDE

$(AB)_F^+$ = ABCDE

同理，$(AC)_F^+$= ABCDE。

又因为$(A)_F^+$=A,$(B)_F^+$=BD, $(C)_F^+$=BCDE，所以AB和AC还满足最小性，都是码。

更便捷地求码

求码有什么用？**范式的判断**需要知道**码、主属性、非主属性。**

我们按照求属性集闭包的方法来验证是不是码。但如果一个一个试就太麻烦了。

我们可以预先得到一些信息，来判断某属性一定是/不是码。

- L、N：只出现在左侧或不出现。或者直接说：**不出现在右侧。**这说明他一定不能被推导出来，所以**必须是**码。
- R：只出现在右侧。它不决定任何属性，**而且能被推导出来。**因此**一定不是**码，否则加入它会破坏最小性。
- LR：无法初步直接判定是否是码。

如上题中，A是L类，D是R类，BCE是LR类。

所以只需要**依次**考虑：A、AB、AC、AE、ABC、ABE、ACE、ABCE。
首先$(A)_F^+$=A。

然后考虑$(AB)_F^+$=ABCDE、排除ABC,ABE,ABCE。

$(AC)_F^+$=ABCDE、排除ACE。

$(AE)_F^+$=AE。

所有情况考虑完毕。

所以候选码就是AB和AC。

最小依赖集

首先函数依赖集有**等价性**，等价性的定义就是其**闭包结果相同。**

很显然，我们要利用等价性，找到最好最方便的一个函数依赖集。

如果有这样一个规范的函数依赖集F，我们称之为**最小依赖集（最小覆盖）**：

- F中所有函数依赖的**右部只有一个属性。**
- F有最小性：F中去掉任何一个函数依赖就不能与F等价。

  > 排除传递函数依赖。
  >
  > A->B,B->C,A->C就不满足最小性。

- F中函数依赖的左侧也要满足最小性。

  > 排除部分函数依赖。
  >
  > 如果A->C，就不能有AB->C。

如何求出最小覆盖?

1. 用分解律分解为**右侧单一属性。**
2. 验证每一条函数依赖**是否可以去掉。**
3. 验证每一条函数依赖的**决定者的子集**是否也能推出**依赖者。**

最小覆盖不是唯一的，和我们的验证（删除）顺序有关。

$F = \{A \to B，B \to A，A \to C，B \to C，C \to A\}$，求$F_m$。

$F_m = \{A \to B，C \to A，B \to C\}$

或者

$F_m = \{A \to B，B \to A，A \to C，C \to A\}$

模式分解

设R是一个关系模式，R的属性集合U={SNO，DEPT，HEAD），R的函数依赖集合F={SNO→DEPT，DEPT→HEAD}，由于R中存在传递函数依赖，因此存在异常，需要进行模式分解，其方式可以有：

分解一：（SNO），（DEPT），（HEAD）

分解二：（SNO，DEPT），（SNO，HEAD）

分解三：（SNO，DEPT），（DEPT，HEAD）

模式分解可以提高范式等级，但是并不是所有的模式分解都能够无损地保存原有信息。分解一损坏了SNO->DEPT与DEPT->HEAD，分解二损坏了DEPT->HEAD。

分解的目标：

- **无损**连接分解

  > 可以通过自然连接得到原表

- 保持**函数依赖**
- 达到**更高级**范式

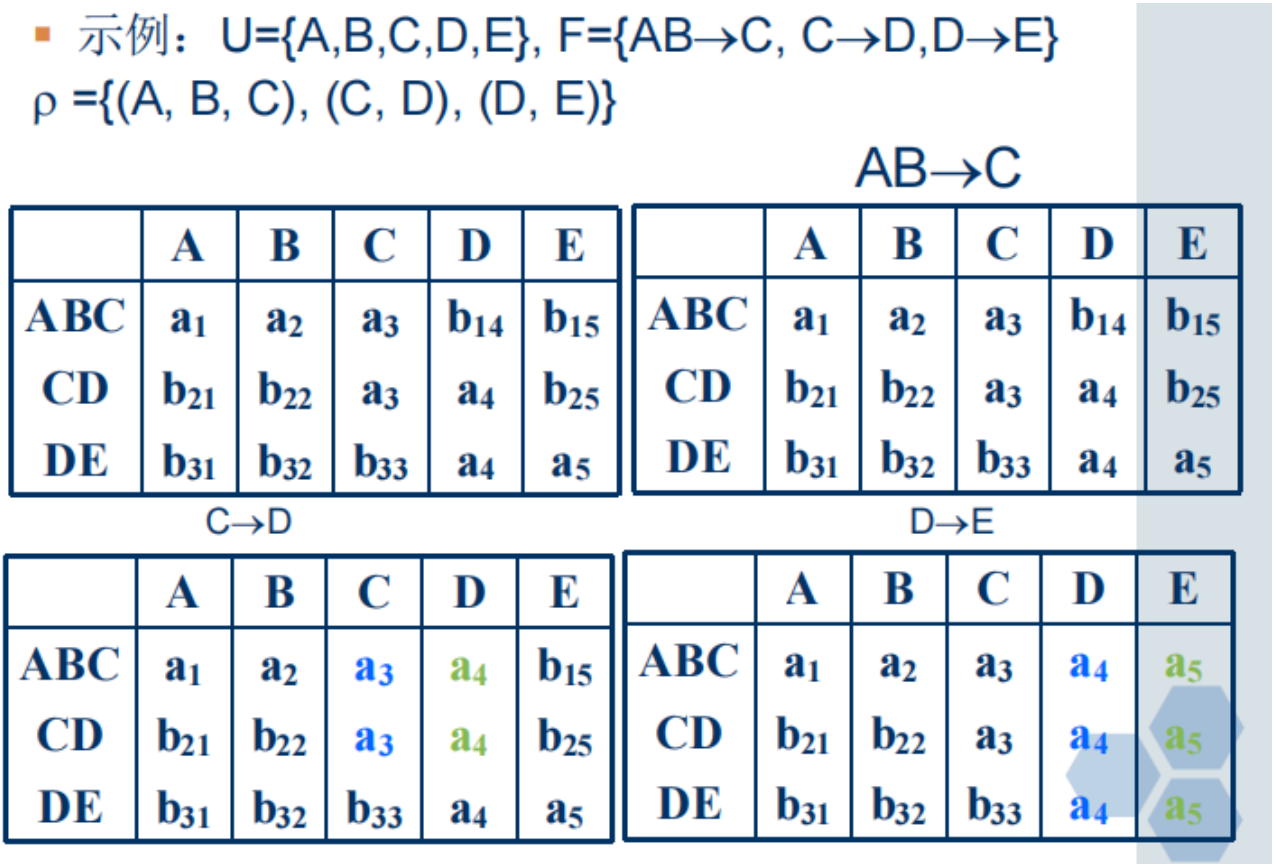
分解的基本要求：

- 表之间的**属性集没有包含关系**，且**并集为原表属性集。**
- 函数依赖需要**投影**到各个表中。

  > 投影也就是左侧有点

判别模式分解的无损连接性

填表法



	A	B	C	D	E
ABC	a_1	a_2	a_3	b_{14}	b_{15}
CD	b_{21}	b_{22}	a_3	a_4	b_{25}
DE	b_{31}	b_{32}	b_{33}	a_4	a_5

	A	B	C	D	E
ABC	a_1	a_2	a_3	b_{14}	b_{15}
CD	b_{21}	b_{22}	a_3	a_4	b_{25}
DE	b_{31}	b_{32}	b_{33}	a_4	a_5

	A	B	C	D	E
ABC	a_1	a_2	a_3	a_4	b_{15}
CD	b_{21}	b_{22}	a_3	a_4	b_{25}
DE	b_{31}	b_{32}	b_{33}	a_4	a_5

	A	B	C	D	E
ABC	a_1	a_2	a_3	a_4	a_5
CD	b_{21}	b_{22}	a_3	a_4	a_5
DE	b_{31}	b_{32}	b_{33}	a_4	a_5

第一张表：

如果**i列**的属性在**j行的属性集**中，填写a_i，如果不在就填写b_{ji}

后续的表：

验证每一条函数依赖是否被违反。如果被违反，也即决定者相同、依赖者不同，就将依赖者统一，如果**有a_i就定为a_i**，否则为**其中行数最小**的值。

> 注意！
>
> - 相等的值要永远保持相等，也就是如果A和B统一过，那么B和C统一时应该将ABC一起统一。
> - 用过的依赖关系需要全程发挥作用。

如果某行达成了$a_1a_2……a_n$，说明是无损连接。

定理法

$$若U_1 \cap U_2 \rightarrow U_1(或U_2)，则r = \prod_{U_1}(r) \bowtie \prod_{U_2}(r)$$

如果拆为了U1,U2两个属性集。如果U1与U2的交集能够**函数决定**U1或U2，那么就是无损连接。

原理如下：U1交U2->U1，那么可以将b_{22}变为a_2，从而形成$a_1a_2a_3$行。

解释起来也不难。

用于自然连接的公共属性是分表的候选码，说明是无损连接。

模式分解中函数依赖的保持

判定方法很简单。如果投影过去的各个依赖集**并起来**能和原依赖集**等价**（也就是闭包相等），就说该分解保持函数依赖。

达到3NF且保持函数依赖的分解算法

这个算法仅满足两个条件，不保证无损连接性。

1. 求出最小依赖集F
2. 找到不在F中出现的属性，单独构成一个关系模式，剩余属性记为U。
3. 若有X->A且XA=U，说明我已找到**满足3NF**的码X, 非主属性集A。算法终止。
4. 否则对F根据**相同左部**的原则分组：每一组函数依赖所涉及的全部属性记为U_i。若有包含关系，取最大的组合。最后得到的组合即为分解结果。

不在F中出现的属性（N类属性）不能进行自然连接。因此不能保证无损分解。

达到3NF且同时保持无损连接与函数依赖的分解算法

针对上一步求出的结果，观察是否有某个关系模式保留了**原表的候选码**。

如果有，该结果即为无损连接。

如果没有，就**新增一个关系模式**，其属性集为**原表**的一个**候选码**，其不含函数依赖（候选码内部肯定没有函数依赖，否则违反最小性）。大家可以依靠这个表实现无损连接。

达到BCNF且保持无损连接的分解算法

该算法**可能无法保证**函数依赖。

1. 检查每个关系模式是不是BCNF。
2. 如果存在某表不是BCNF，则是因为存在函数依赖X->A且X不是码。所以将该表U拆为XA与U-A。（这一步保证了无损连接）

算法必能终止，因为只含两个属性的关系模式一定是BCNF。

> 推荐的做法：
>
> 先用3NF的，针对3NF的结果再去做这个算法，更有可能保存函数依赖。

数据库设计

数据库设计的关键是**构造合理的数据模型**。

数据库设计的特点：

- 数据库设计与**硬件、软件**等紧密相关。
- 数据库设计要把**结构**（数据）设计和**行为**（处理）设计密切结合起来。

设计方法：手工试凑方法、规范化设计方法。

规范化设计的特点：**分步进行**、**反复性**、**试探性**。

数据库的生命周期

- **需求分析**
- **概念结构设计**
- **逻辑结构设计**
- **物理结构设计**
- **数据库的实施**
- **数据库运行与维护**

需求分析

需求分析的任务是通过详细调查现实世界要处理的对象（组织、部门、企业等），充分了解原系统（手工系统或计算机系统）的工作情况，明确用户的各种需求，并预测系统今后可能的扩充和改变，然后在此基础上确定新系统的功能。

调查的重点是**数据**和**处理**，包括：

- **信息要求**，指用户需要从数据库中获得的信息的内容与性质。从中可以导出数据要求。
- **处理要求**，指用户要完成什么处理功能，对处理的响应时间和处理方式的要求。
- **安全性与完整性**的要求

数据字典是系统中各类**数据描述**的集合，是进行详细的数据收集和数据分析所获得的主要成果。包括：数据项、数据结构、数据流、数据存储、处理过程。

概念结构设计

将需求分析得到的**用户需求**抽象成为**信息结构**即**概念模型**的过程就是概念结构设计。

应用定义的目的是确定最终数据库**支持哪些应用**系统。应用领域的逻辑模型是应用定义的基础。

信息定义的目的是确定最终数据库**需要存储哪些**信息。信息的定义也以应用领域的逻辑模型为基础。

设计策略：

- 自顶向下

首先定义**全局概念**结构的框架，然后逐步细化。

- 自底向上（常用）

首先定义**各局部应用**的概念结构，然后将它们集成起来，得到全局概念结构。

- 逐步扩张

首先定义**最重要的核心概念**结构，然后向外扩充，以滚雪球的方式**逐步生成**其他概念结构，直至总体概念结构。

- 混合策略

将自顶向下和自底向上相结合，用**自顶向下**策略设计一个全局概念结构的框架，以它为骨架集成用**自底向上**策略设计的各局部概念结构。

视图综合设计方法分为两步：

1. 设计局部概念结构

2. 把局部概念结构合并为一个全局概念结构。

局部概念结构

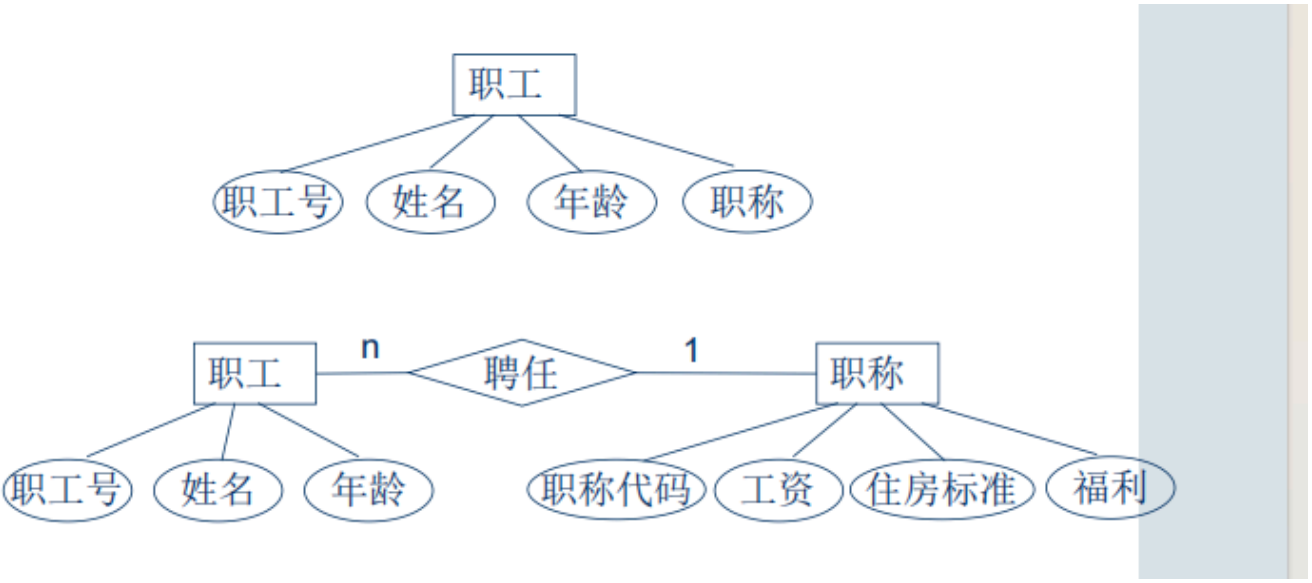
局部视图设计步骤

1. 选择局部应用
2. 逐一设计分E-R图
3. 确定局部实体
4. 确定局部实体之间的联系
5. 构造局部E-R图

局部视图设计的关键在于**实体和属性**的正确划分。其主要手段为：**分类**、**聚集**、**概括**。

- 属性不能再具有需要描述的性质。属性必须是不可分的数据项，**不能包含其他属性。**
- **属性不能与其他实体**具有联系，即E-R图中所表示的联系都是实体之间的联系。

职称由**属性**升级为**实体。**



全局概念模式合成

E-R图之间的集成：

- 一次集成
- 逐步集成（两两合并）

集成的步骤：

- 合并：**解决冲突**并合并
- 修改和重构：**消除冗余**

冲突的种类：

- **命名冲突**
    - 同名异义
    - 异名同义

- **结构冲突**。结构冲突是模式结构的冲突。
    - 在一个局部ER图中被表示为一个**实体**，而在另一个局部ER中却被表示为一个**属性或联系。**
    - 同一实体在不同的局部ER图中所包含的**属性个数和属性的排列次序**不完全相同。可以取全集来解决。
- **值域冲突**。属性域冲突，即属性的**类型**、**取值范围**或**取值集合**不同。（如取值单位）
- **约束冲突**。两个局部模式在同一个概念上**定义了不同的约束**。

消除冗余：

在**初步ER图**中，可能存在一些冗余的数据和实体间冗余的联系。

> 冗余的数据是指可**由基本数据导出**的数据。
>
> 冗余的联系是指可**由其他联系导出**的联系。

消除了冗余后的初步ER图称为**基本ER图。**

消除冗余主要采用**分析**方法，即以**数据字典**和**数据流图**为依据，根据数据项之间的逻辑关系来消除冗余。

事务设计

数据库设计的目的是**支持各种事务的运行。**

事务可以分为三类：**数据查询型事务，数据更新型事务，混合型事务。**

逻辑结构设计

任务：**E-R图转关系模式**

> 把概念结构设计阶段设计好的**基本E-R图**转换为与选用的DBMS所支持的数据模型相符合的**逻辑结构**。

目标：

- 满足用户的**完整性和安全性**要求。
- 动态关系至少满足**第三范式**，静态关系至少满足**第一范式。**

    > 动态：更新型，静态：查询型

- 能够在**逻辑级上高效率**地支持各种数据库事务的运行。
- **存储空间利用率**高。

设计步骤：

- 形成初始关系数据库模式
- 关系模式规范化（3NF、1NF）
- 关系模式优化
- 定义关系上的完整性和安全性约束
- **子模式**定义（为每一类用户定义外模式）
- 性能估计

形成初始关系数据库模式

直接由概念结构设计的结果生成关系数据库模式。

直接把E-R图中的实体、实体间的联系等变换为关系模式。

实体型的转换

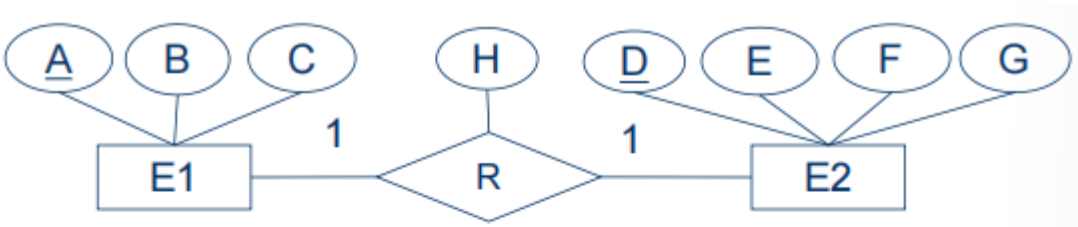
直接将一个实体型转化为一个关系模式。

实体的属性就是关系的属性，实体的码就是关系的码。

实体间联系的转换

1：1联系

- 独立关系模式：属性为与该联系相连的各实体的码以及联系本身的属性。**每个实体的码**均是该关系的候选码。
- 合并进其中一个关系模式：该关系模式的属性中加入**另一个关系模式的码**和**联系本身的属性**。



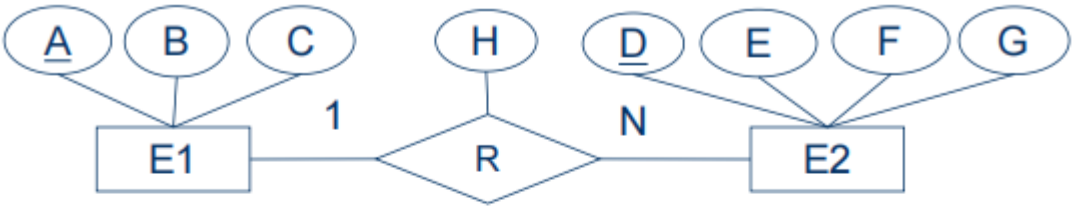
- 可转换为以下关系模式
  - E1（A，B，C）
  - E2（D，E，F，G）
  - R（A，D，H） 或者R（A，D，H）
- 或：
  - E1 (A，B，C，D，H)
  - E2（D，E，F，G）

或：

- E1（A，B，C）
- E2（D，E，F，G，A，H）

1：N联系

- 独立关系模式：属性为与该联系相连的各实体的码以及联系本身的属性。**N端实体的码**均是该关系的候选码。
- 合并进**N端**关系模式：N端关系模式的属性中加入**1端关系模式的码**和**联系本身的属性**。

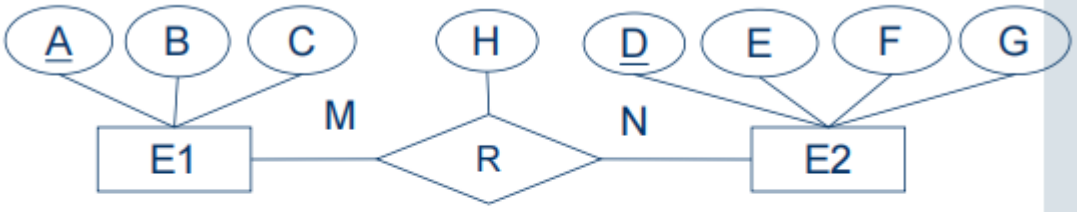


- 可转换为以下关系模式
  - E1（A，B，C）
  - E2（D，E，F，G）
  - R（A，D，H）
- 或：
  - E1（A，B，C）
  - E2（D，E，F，G，A，H）

M：N联系

只能转换为**一个独立的关系模式。**

与该联系相连的各实体的码以及联系本身的属性均转换为关系的属性，而**关系的码为各实体码的组合。**

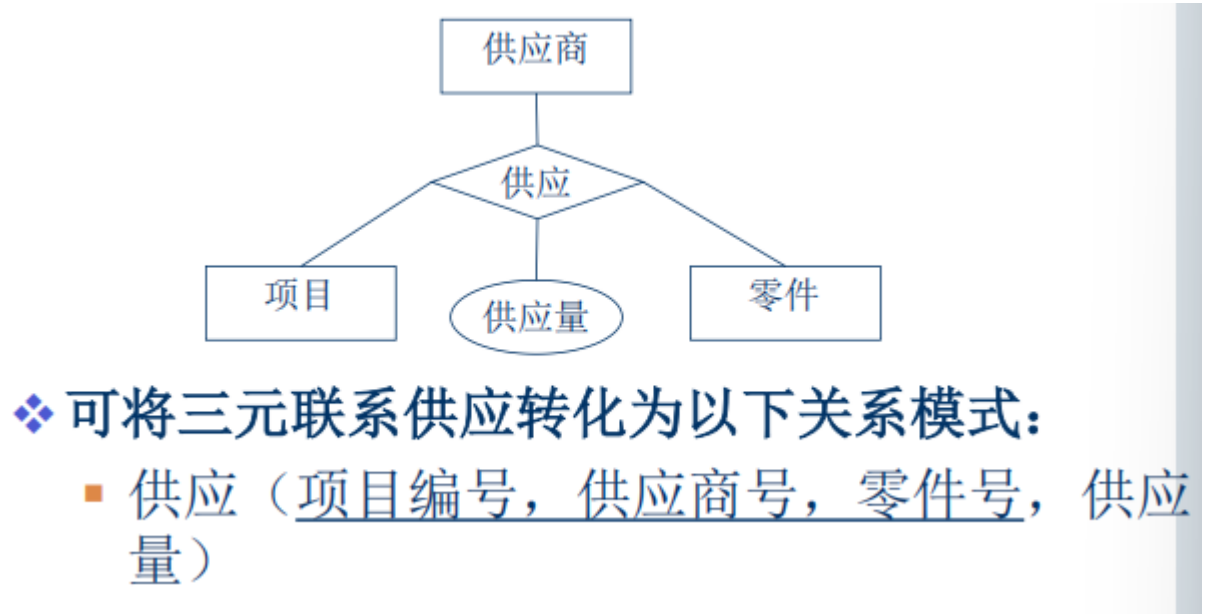


- 只能转换为以下关系模式
  - R（A，D，H）

三个或三个以上实体间的一个多元联系

只能转换为**一个独立的关系模式。**

与该联系相连的各实体的码以及联系本身的属性均转换为关系的属性，而**关系的码为各实体码的组合。**



关系模式的规范化

确定函数依赖。消除冗余联系。进行规范化处理。

分析这些关系模式对于这样的应用环境**是否合适**，是否要对某些模式进行**合并或分解**。

关系模式的优化

在数据库设计中，**并非关系的规范化程度越高越好。**

规范化需要**模式分解**，查询时**连接操作更多**，影响**系统效率。**因此必须根据应用特点对数据库设计进行优化，对关系模式进行必要的分解，提高**数据操作的效率**和**存储空间的利用率。**

常用的两种分解方法是**水平分解**和**垂直分解。**

水平分解 减少行数

是把**关系的元组**分为若干子集合，每个子集合定义为一个子关系。

- 高频数据：根据**80/20原则**， 可以把经常使用的那一部分**数据**分解出来作为一个关系，其他数据作为另一个关系。
- 事务专用数据：如果关系R上具有n个**存取数据不相交**的事务，则R可以分解为少于或等于n个子关系。

垂直分解 减少列数

是把关系模式**R的属性**分解为若干子集合，形成若干子关系模式。

原则：**经常在一起使用的属性**从R中分解出来形成一个子关系模式。

> 如学生表信息很多，可以把学号与姓名单独分出来。

垂直分解必须确保**无损连接性和保持函数依赖。**

逆规范化 合并关系

连接操作代价很高，是造成关系数据库**低效的主要原因之一。**

如果**经常需要**对多个关系进行连接操作，且**大多数操作为查询操作，更新很少**，则可以考虑将这些关系**合并**为一个关系。

设计子模式

设计用户外模式。由于用户外模式与全局模式是**相对独立**的，因此定义用户外模式时可以注重考虑用户的习惯和方便。

数据库物理设计

设计**内模式。**

数据库在**物理设备上的存储结构和存取方法**称为数据库的物理结构。

方法：

- 了解应用的类型和特点
- 了解所用的RDBMS提供了哪些**存取方法和存储结构**

数据库的存取方法

索引存取方法

应为其建立索引：

- 经常在**查询条件**中出现的属性
- 经常作为最大值和最小值等**聚集函数参数**的属性
- 经常在**连接条件**中出现的属性

聚簇存取方法

聚簇存取方法将**相关的数据存放在连续的物理块**中。

HASH存取方法

如果一个关系的属性主要出现在**相等比较条件**中，且满足以下两个条件之一，则可以使用HASH存取方法：

- 一个关系的**大小可以预知**，而且不变。
- 关系的大小动态改变，但数据库管理系统提供了**动态HASH存取**方法

数据库的存储结构

- 确定数据的存放位置

  > 根据应用情况将数据的**易变部分与稳定部分**、**存取频率**高与低的部分分开存放。

- 确定系统配置

数据库的实施

- 数据的载入和应用程序的调试
- 数据库的试运行（功能和性能）

数据库的运行和维护

- 数据库的转储和恢复
- 数据库的安全性、完整性控制
- 数据库的性能监督、分析和改造

- 数据库的重组织和重构造

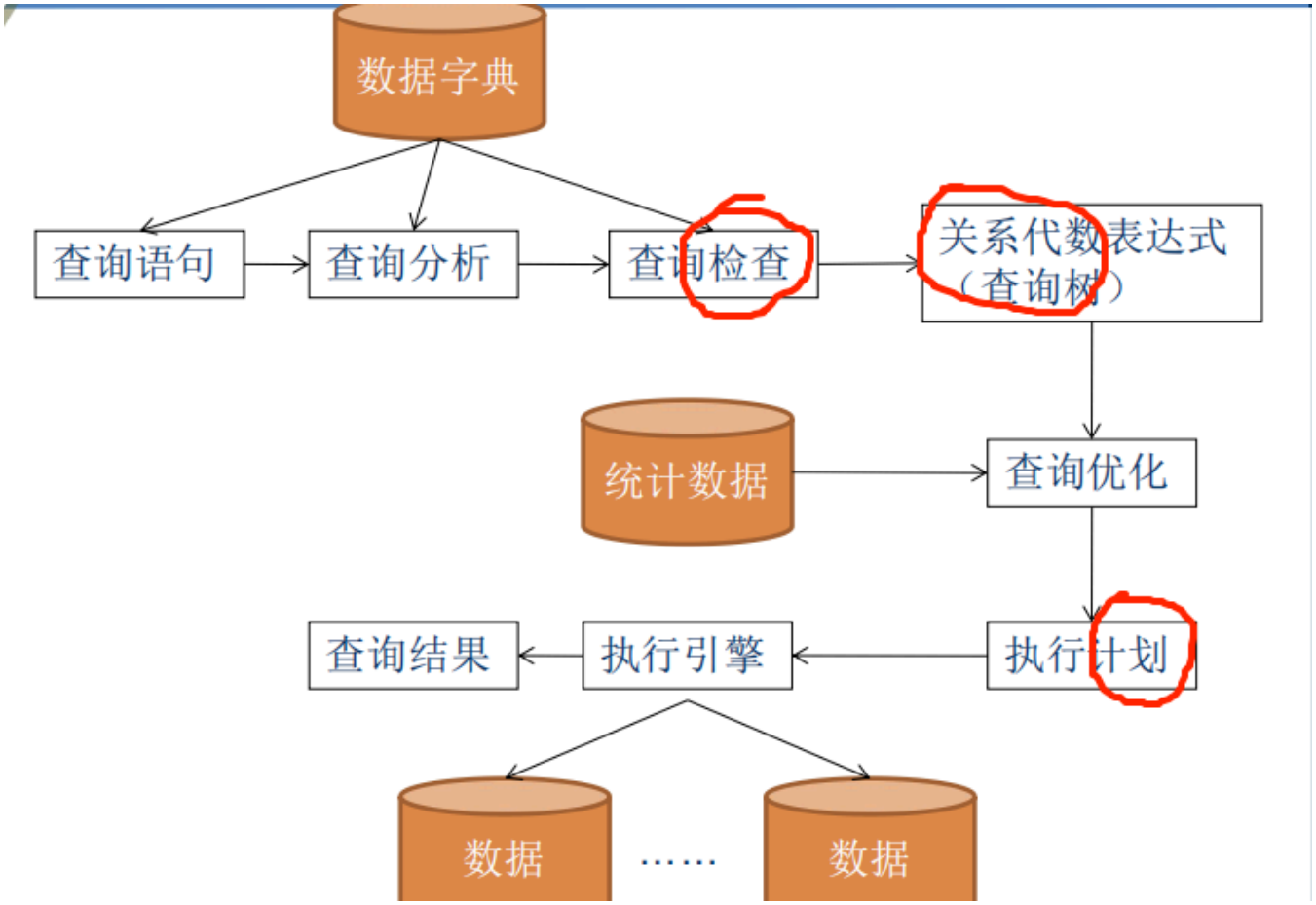
查询优化

本单元视角引向：作为**DBMS的设计者**，从数据库实现而非应用的角度，我们应该关注的内容。我们要实现关系操作的“非过程化”，本单元则对该过程进行简述。

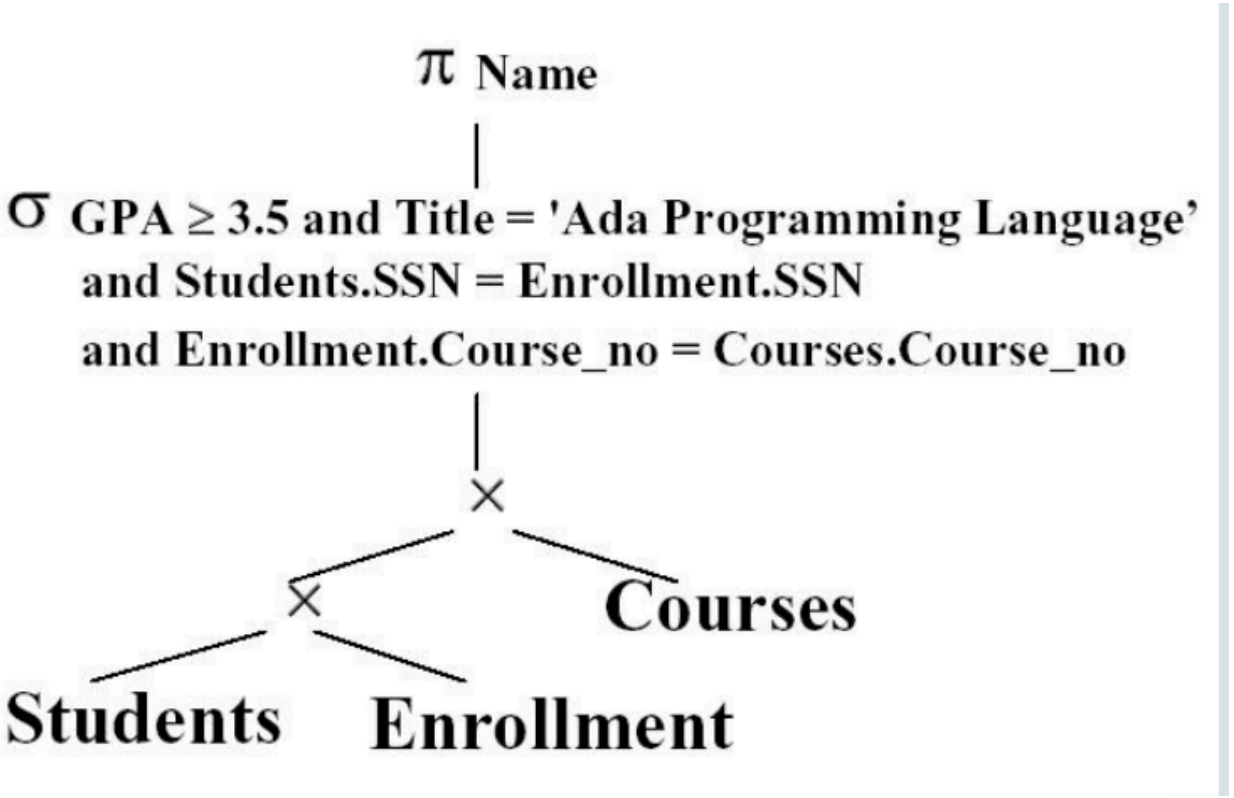
- 降低对用户的要求，方便了用户的使用
- 能够取得更好的优化效果

> DBMS掌握的信息更丰富，经验更丰富，优化更到位

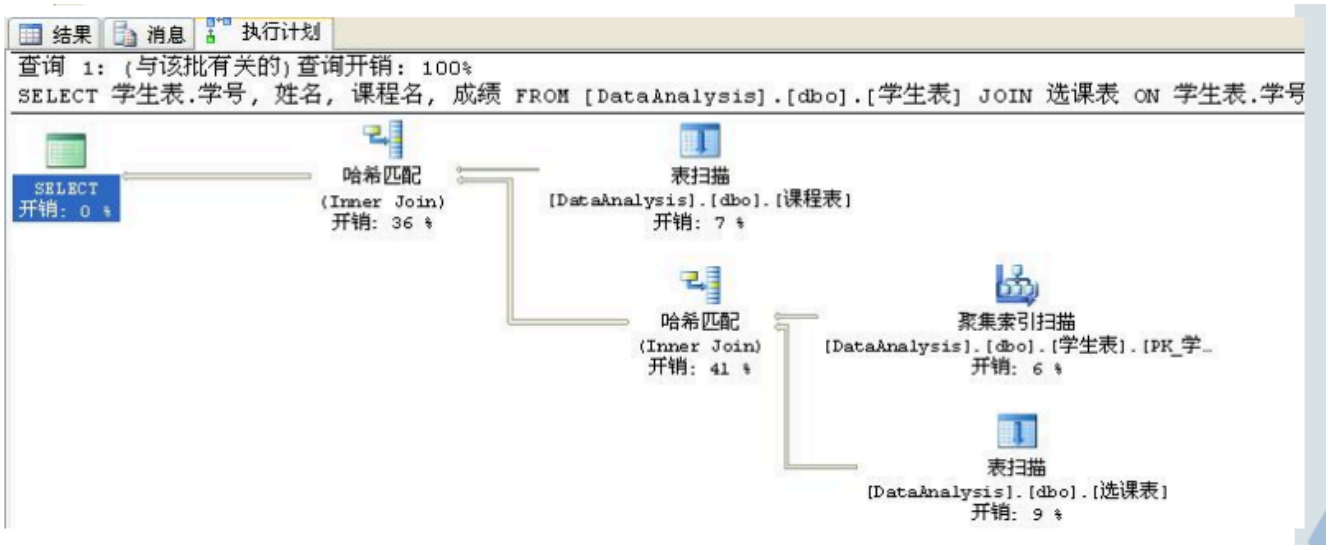
查询处理的基本步骤：查询分析、查询检查、查询优化、查询执行。



查询树：



执行计划：



基本运算的实现

每一基本的代数运算都有**多种不同的实现算法。**

适用于不同的情况，其执行代价不同。

选取

全表扫描方法：依次访问表的每一个块，对于每一个元组，测试它是否满足选择条件。

> 效率低，但对关系的存储方式**没有要求，不需要索引**。适用于任何选择条件。

折半扫描: 对于按某一属性排序的文件，且选择条件是该属性上的**等值比较**方法，可以使用折半的方法扫描文件。

> 效率高，但**需要有序文件。**

索引扫描：对于在选择条件的属性上建有索引的表，可以采用访问索引，根据索引项的指示去访问数据元组的方法。

- 无序索引：访问满足**等值条件**的元组
- 有序索引：访问**满足范围**查找条件的一系列元组

> 效率高，**必须建立对应索引。**

连接

- **嵌套循环方法**
- **排序-合并方法**

  > 先排序再折半查找

- **索引连接方法**
- **HASH join方法**

查询代价的度量

总代价=**I/O代价+**CPU代价+通信开销

I/O代价占绝大部分。

其度量方式为：I/O块数或者I/O的次数。

> 不是时间：影响时间的相关因素太多了，不适合作为查询本身的代价度量。

一个重要因素：主存中**缓冲区的大小M**

- 最好，所有的数据可以读入到缓冲区中
- 最坏，缓冲区只能容纳少量数据块，大约每个关系一块

连续测试两次，第二次更快很有可能是缓冲区导致的！

代数优化与物理优化

代数优化：**关系代数表达式**的优化，即按照一定的规则，改变代数表达式中操作的次序和组合
物理优化：**存取路径和底层操作**算法的选择。

关系代数等价变换

若产生的结果关系具有相同的属性集和元组集，则称**两个表达式等价。**

1. 连接、笛卡尔积的交换律
2. 连接、笛卡尔积的结合律
3. 投影的串接定律
4. 选择的串接定律
5. 选择与投影的交换律
6. 选择与笛卡尔积的交换律
7. 选择与并的分配律
8. 选择与差的分配律

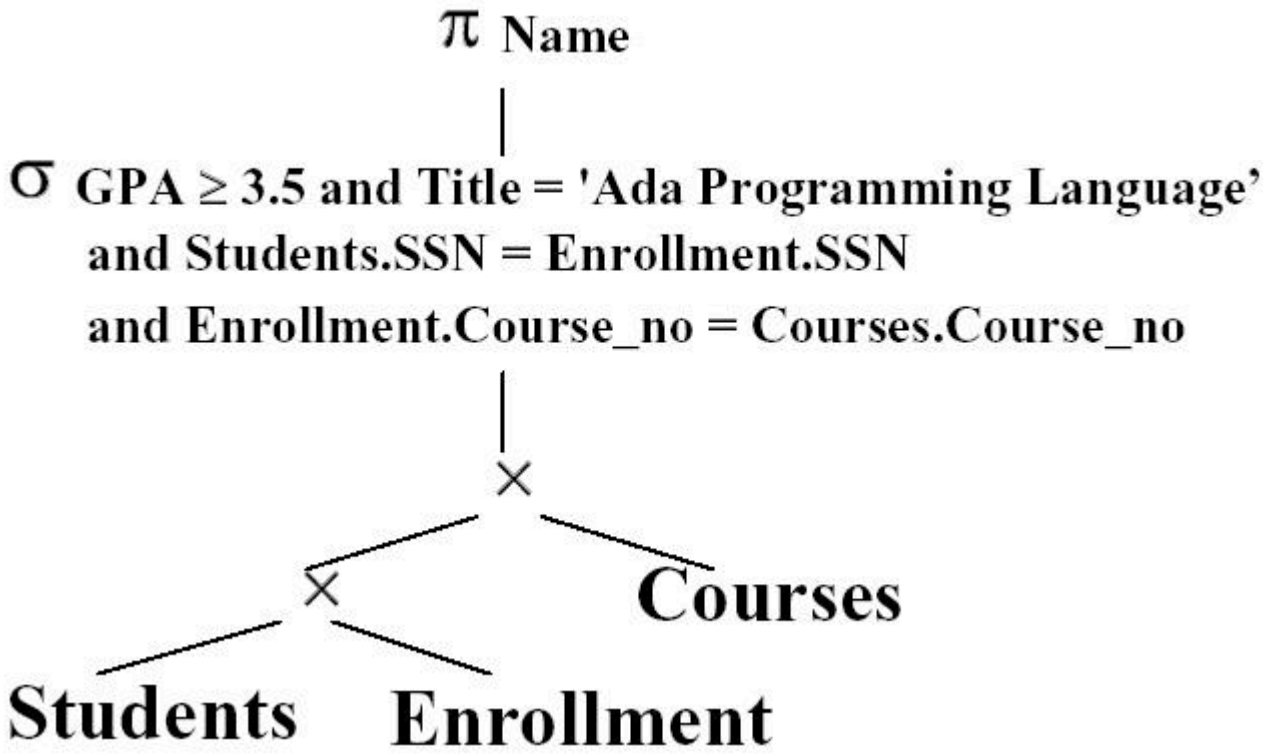
9. 投影与笛卡尔积的分配律
10. 投影与并的分配律

查询优化的一般准则

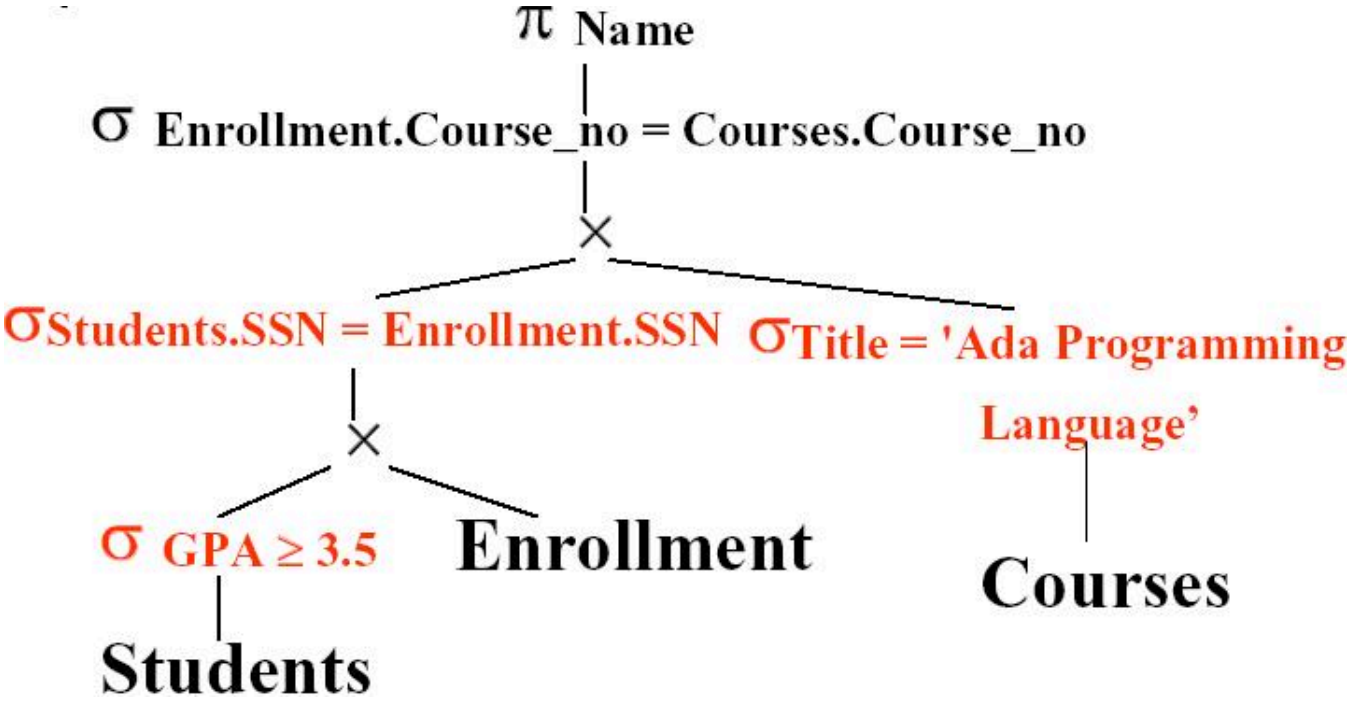
- **选择运算应尽可能先做。**目的：减小**中间关系。**
- 在执行连接操作前对文件**适当进行预处理。**如排序、在连接属性上建立索引。
- **投影运算和选择运算**同时做。目的：避免**重复扫描**关系。
- 把**投影运算与其前面或后面的双目运算**结合起来。目的：减少扫描关系的遍数。
- 选择 + 笛卡尔积->连接
- 找出**公共子表达式**

优化实操：

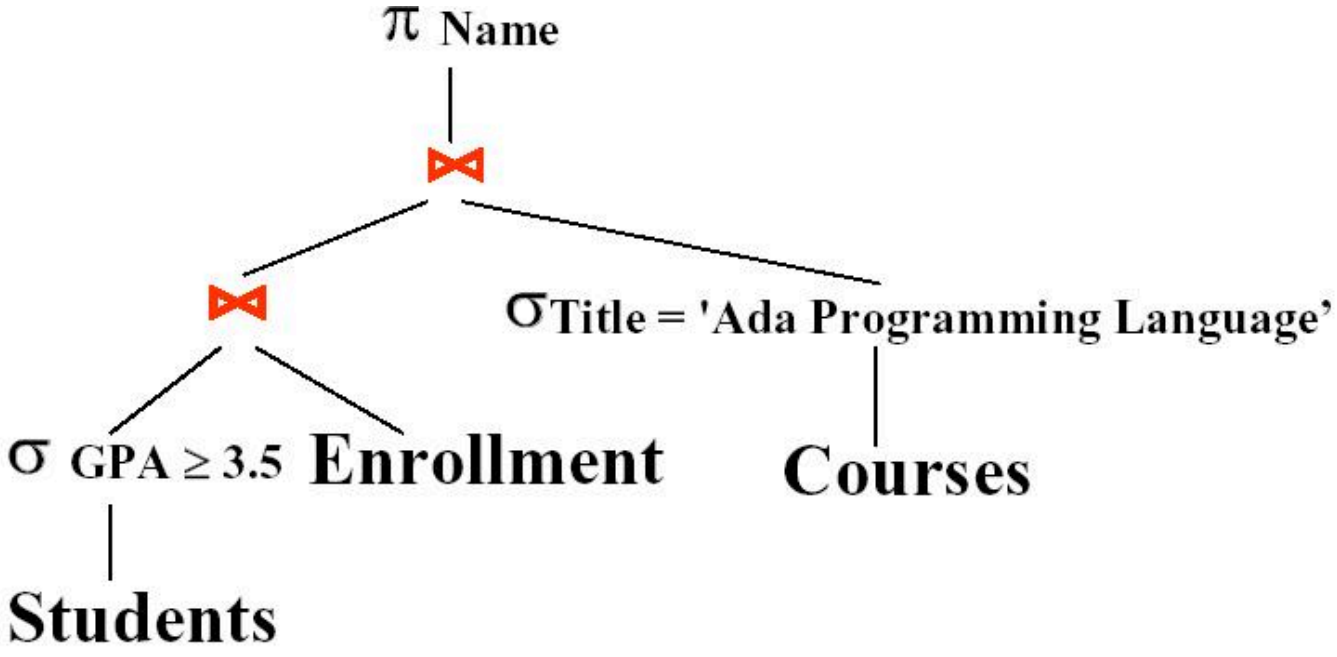
初始查询树



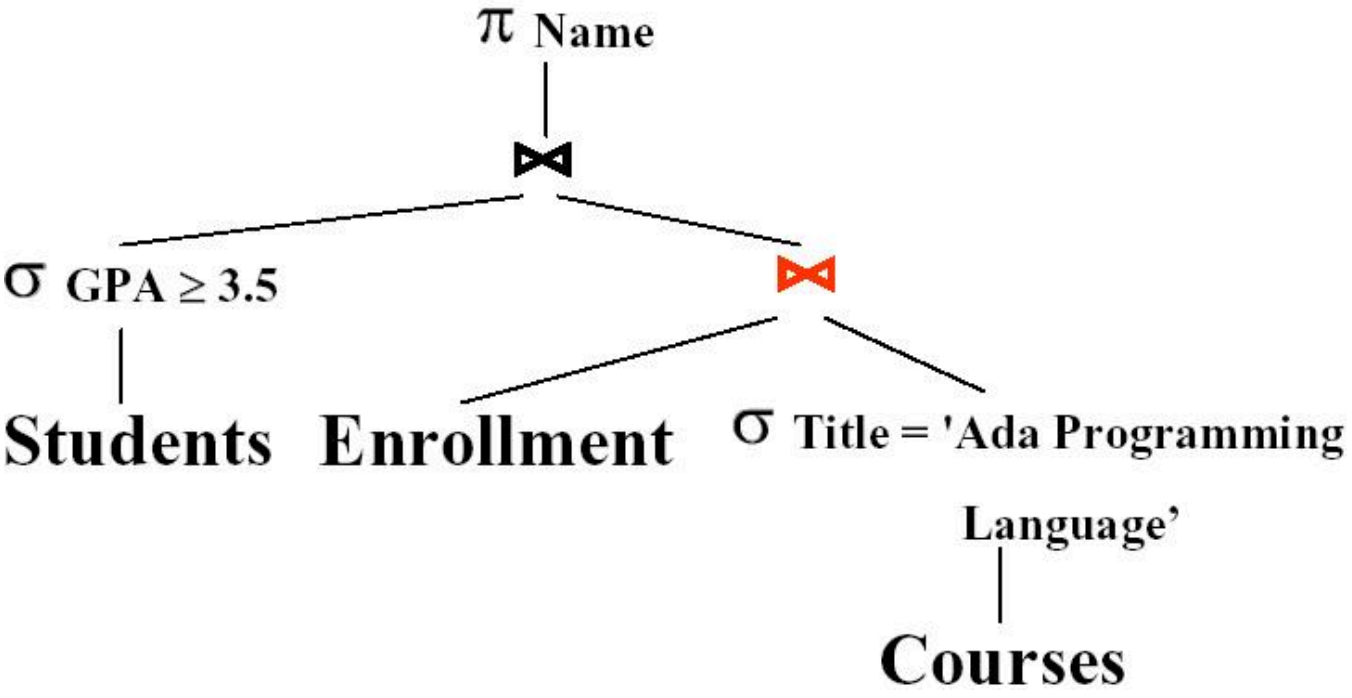
优先选取



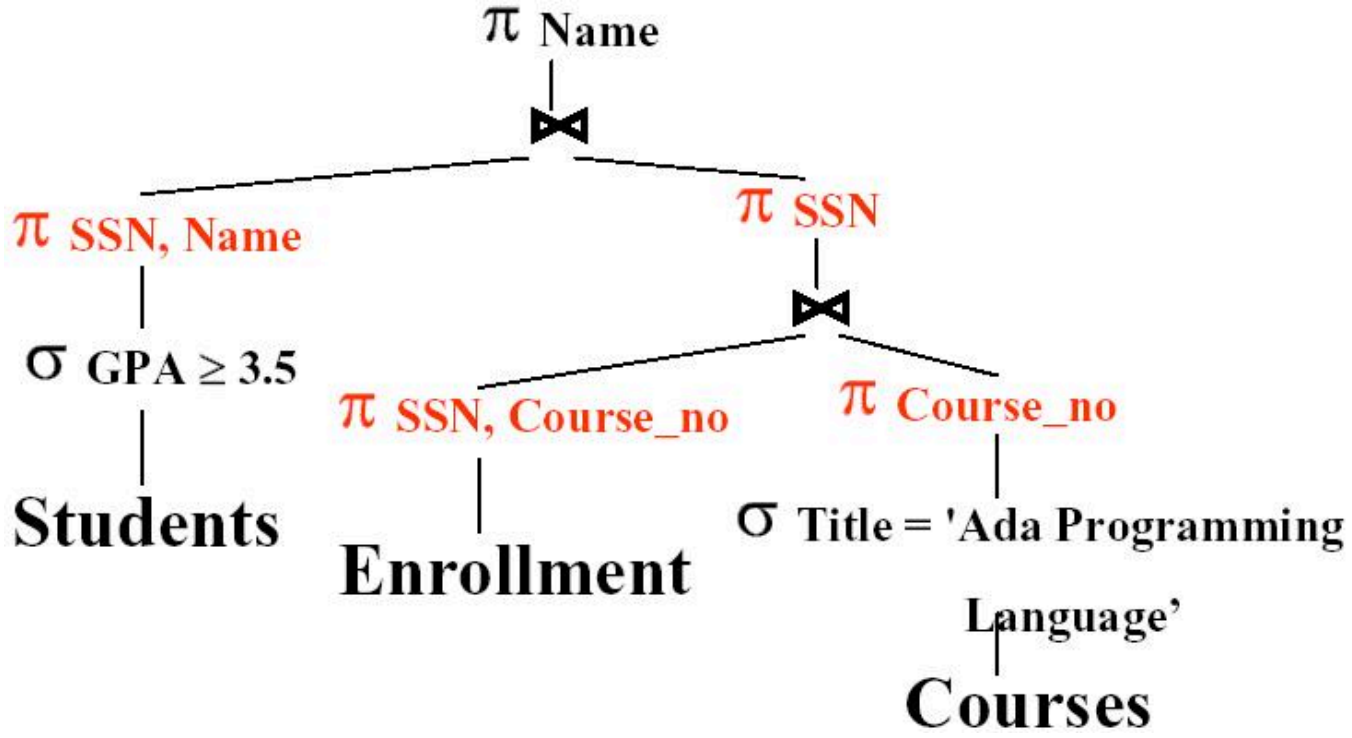
连接替代笛卡尔积



优先执行小连接



投影去掉无关属性



数据库恢复

事务

事务是用户定义的一个数据库**操作序列**构成，这些操作**要么全做，要么全不做**，是一个**不可分割**的工作单位。

> 事务与应用程序是两个概念，一般来说，一个应用程序可以包含多个事务，事务是更底层的概念。

事务显式定义：事务的**开始与结束可以由用户显式控制。**

事务隐式定义：如果用户没有显式定义，则由DBMS**自动划分事务。**

事务的基本流程

事务以**Begin transaction开始**，以 **Commit work 或 Rollback work** 结束。

Commit work表示提交，**事务正常结束。**

Rollback work表示事务非正常结束，**撤消事务已做的操作，回滚到事务开始时状态。**

事务的特性 ACID

原子性(Atomicity)

事务是数据库的逻辑工作单位，事务中包括的诸操作**要么全做，要么全不做。**

一致性(Consistency)

事务执行的结果应当使数据库从一个**一致性状态**转变为另一个**一致性状态。**

隔离性(Isolation)

一个事务的执行**不能被其它事务干扰。**即一个事务内部的操作及使用的数据对其它并发事务是隔离的，**并发执行**的各个事务之间不能相互干扰。

持久性(Durability)

一个事务一旦提交之后，它对数据库的影响**必须是永久**的。事务提交后，系统**发生故障不能改变事务的持久性。**也就是发生故障了必须恢复。

数据库恢复子系统

数据库管理系统必须具有把数据库从**错误状态**恢复到**某一已知**正确状态的功能，这就是数据库的恢复。

数据库恢复是通过**数据库恢复子系统**完成的。

核心目标：

1. 保护事务的原子性。
2. 发生故障后，数据库仍能恢复到正确状态。

数据库故障

事务内部的故障包括可预期的和不可预期的。

- 可预期的是指可以通过**事务程序本身**发现和处理的故障。
- 不可预期的错误是指那些不能由应用程序处理的**事务故障**，如死锁，运算溢出，违反完整性规则等。

系统故障

系统故障是指造成**系统停止运行**的任何事情，使得系统要重新启动。如硬件错误，操作系统故障，停电等。

这类故障影响**正在运行的所有事务**，**但不会破坏数据库。**

介质故障

介质故障指**外存**故障，如磁盘损坏，瞬时强磁场干扰等。

这类故障将**破坏全部或部分数据库**，并**影响正在存取这部分数据的所有事务。**

计算机病毒

计算机病毒是一种**人为的破坏或故障**，已成为数据库系统的主要威胁之一。

恢复技术

数据库恢复的基本原理为冗余。冗余包括**数据**的冗余与**操作**的冗余。

如果数据库中任何一部分数据被破坏或处于不正确的状态，则可以通过存储在系统别处的冗余数据来重建。

建立冗余

数据转储——数据冗余

DBA定期地将整个数据库**复制到磁带或其它存储设备上**保存起来的过程。

这些备用的数据文本称为**后备副本或后援副本。**

转储状态分为**静态转储**和**动态转储。**

静态转储——冷备份

静态转储是在系统中**无事务运行时进行**的转储操作。

新的事务必须等待转储结束才能执行，降低了数据库的可用性。

动态转储——热备份

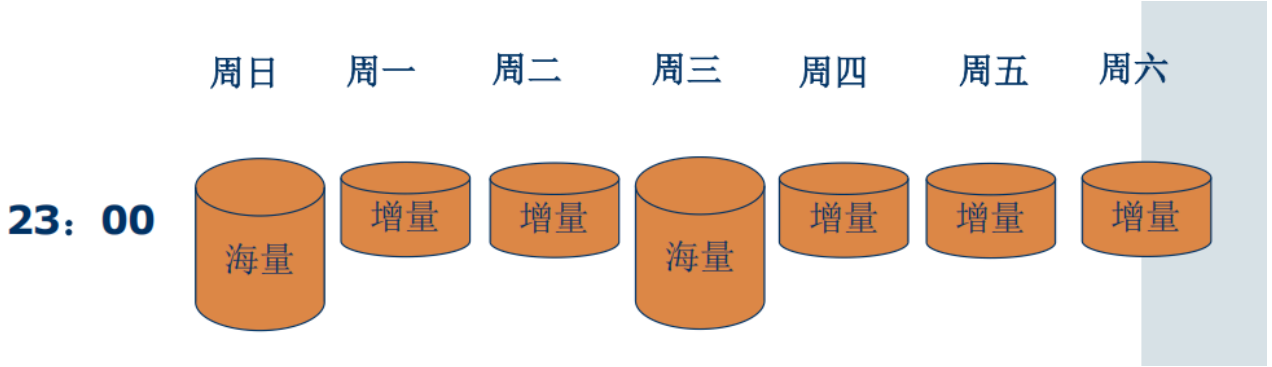
动态转储是指**转储期间允许**对数据库进行存取或修改，即转储和用户事务可以**并发执行。**

必须把转储期间各事务**对数据库的修改记录**下来，这样才能保证把数据库恢复到某一时刻的一致状态。

转储方式又分为**海量转储**和**增量转储。**

- 海量转储指每次转储**全部数据库。**
  - 慢，但是每次的数据都能直接用。
- 增量转储指每次只转储上一次转储后更新过的数据。
  - 快，但是需要严格按次序恢复，不能直接保证正确性。

从而有：**动态海量转储 静态海量转储 动态增量转储 静态增量转储。**



登记日志文件——操作冗余

日志文件是用来记录事务对数据库的**更新**操作的文件。

> 不是记录所有操作！查询不会导致数据库变化。

日志文件两种格式：以**记录为单位**和以**数据块为单位。**

以**记录**为单位：内容包括各个事务的**所有更新操作、开始标记、结束标记。**

- 事务标识、操作的类型、操作对象、更新前数据的旧值、更新后数据的新值

以**数据块**为单位：内容包括**事务标识**以及**更新前和更新后的数据块。**

> 为什么不仅记录操作还要记录值?
>
> 只依赖逆操作可能有问题：可能操作做了，日志记录了，但值还没改就故障了。

事务故障和系统故障恢复必须使用日志文件。

在动态转储方式中必须建立日志文件。

在静态转储方式中，也**可以建立**日志文件。当数据库发生故障（**介质故障**）时，用后援副本恢复过去版本的数据，然后利用日志文件重做已完成的事务，把数据库恢复到正确状态。

原则：

- 登记的次序**严格按并发事务执行**的时间顺序。
- 必须**先写日志文件，后写数据库。**

恢复策略

事务故障

事务故障的恢复是由**系统自动完成**的。

反向扫描日志文件，依次将日志记录中的**“更新前的值”**写入数据库，直到读到该事务的**开始标志。**

系统故障

未完成的事务对数据库的更新**可能（所以用值而非逆操作）**已经写入数据库。

已提交事务对数据库的更新**可能（所以用值而非操作）**还留在缓冲区还没来得及写入数据库。

要**撤销未完成的事务，重做已完成的事务。**

1. **正向扫描**整个日志文件，故障前已完成的事务放入**REDO**，故障后已完成的事务放入**UNDO。**
2. 反向执行UNDO，依次将日志记录中的**“更新前的值”**写入数据库。
3. 正向执行REDO，依次将日志记录中的**“更新后的值”**写入数据库。

介质故障

介质故障将全部或部分地**破坏数据库甚至是日志文件。**

1. 装入最新的数据库后备副本，将数据库恢复到一致状态。

   > 对于动态转储的副本，还需要装入转储开始时刻的日志文件副本。

2. 装入转储以后的日志文件副本，重做已经完成的事务。

具有检查点的恢复技术

事务太多了。常常REDO了大量已经写到数据库中的事务。所以已完成的事务无需管理。

在日志文件中增加一类新的记录：**检查点记录**，增加一个**重新开始文件**。

> 检查点记录也是日志的一条。

检查点记录的内容包括：

- 建立检查点时刻所有**正在执行的事务**清单。
- 这些事务最近一个**日志记录**的地址。

这两个信息表示了恢复时建议处理的**事务**和建议开始扫描的**日志**地址。

重新开始文件用来记录**各个检查点**记录本身在日志文件中的地址。

检查点记录操作

将当前日志缓存中的所有**日志记录写入磁盘**的日志文件上。

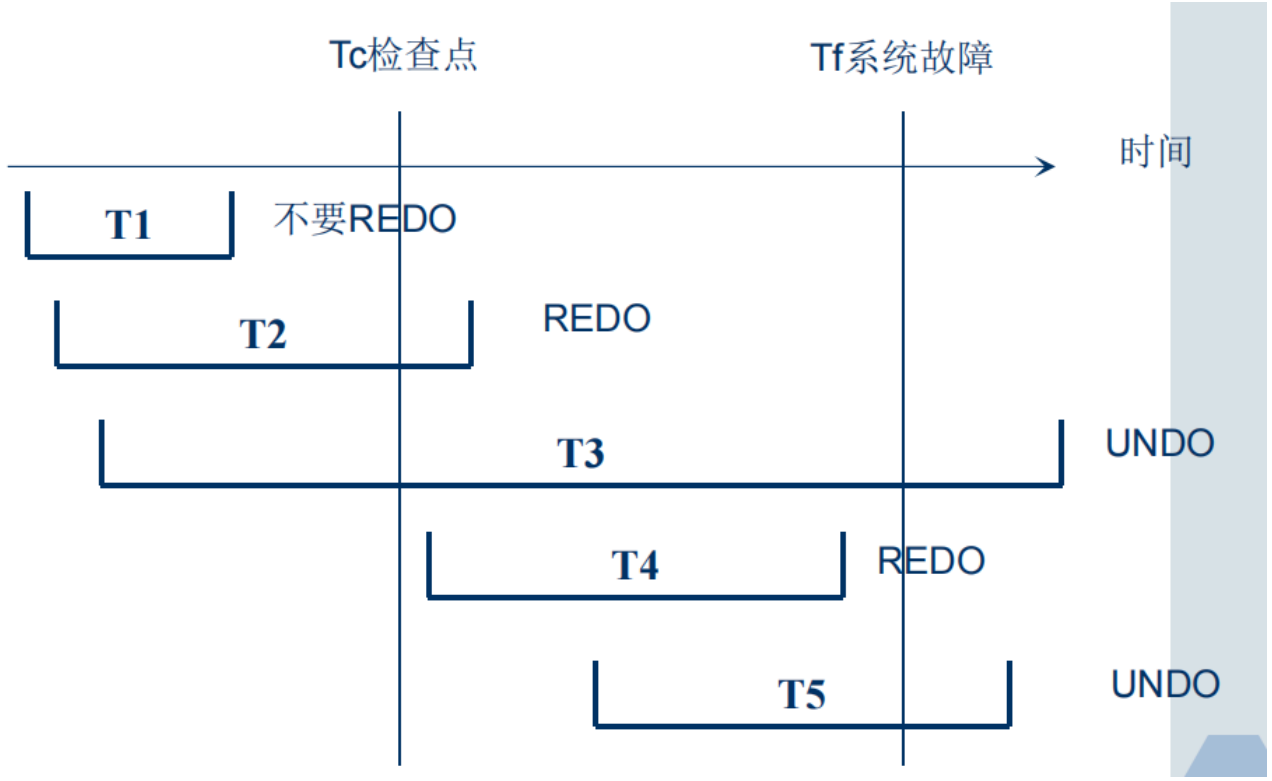
在日志文件上**写入一个检查点**记录，其内容包括多个事务与一个日志地址。

将当前数据缓存的所有**数据记录写入磁盘**的数据库中。**（保证数据已存入）**

把**检查点记录在日志文件中地址**写入一个重新开始文件。

> 当事务T在一个检查点之前提交，T对数据库所做的修改一定已经写入数据库。
>
> 这样在进行恢复处理时，没有必要对事务T进行REDO操作。

恢复



1. 在**重新开始文件**中找到**最后一个检查点记录**在日志文件中的地址，由该地址找到最后一个检查点记录。
2. 由该检查点记录得到**检查点建立时刻所有正在运行的事务**清单，放入UNDO-LIST
3. 从**检查点开始正向扫描**日志文件
    - 如果有新开始的事务，放入UNDO-LIST
    - 如果有提交的事务 ，**从UNDO-LIST队列移入到REDO-LIST队列**

> 废话一堆，其实就是还是已提交的REDO，未提交的UNDO。只不过从检查点开始。

4. 直到文件结束，然后对UNDO-LIST中的执行UNDO，对REDO-LIST中的执行REDO。

数据库镜像

根据DBA要求，自动把整个数据库或其中的关键数据**复制到另一个磁盘上。**每当主数据库更新时，DBMS**自动把更新后的数据复制过去**，即DBMS自动保证镜像数据与主数据的一致性。

这样，一旦发生介质故障：

- 可以**继续由镜像磁盘提供服务。**
- 可以**直接通过镜像磁盘进行数据库恢复。**

在没有故障时，镜像磁盘还可以**提供一定的并行性。**

并发控制

优势

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，改善系统的资源的利用率。
2. 系统中存在着周期不等的各种事务，串行会导致难以预测的时延。采用并发会减少**平均响应时间**，特别是改善短事务的响应时间。

问题

当多个事务并发执行的时候，有可能会**相互影响**，从而读取或者存储不正确的数据，破坏**数据库的一致性。**

造成并发执行事务问题的**原因**是：

- 多个事务同时存取**同一个数据集合**
- 并且其中至少有一个事务对该数据集合**进行了更新**操作

丢失修改

两个事务T1和T2读入同一数据并修改， T2提交的结果**破坏了T1提交**的结果，导致T1的修改被丢失。

> 写后写，前一次写的结果被覆盖。

不可重复读

事务T1读取某一数据后，事务T2对其做了修改，当事务T1**再次读取**该数据时，得到与前一次不同的值。

破坏了事务的**隔离性**，一个事务进行过程中遭遇了其他事务的干扰。

> 读后写，再读就不一样了。

不可重复读之幻影行

事务T1按照一定条件从数据库中**读取了某些数据记录**后，

事务T2删除/增加了部分记录，

当T1再次按照**相同条件**读取数据时，发现某些记录神秘的消失/出现了。

读“脏”数据

事务T1修改某一数据，并将其写回磁盘，事务T2读取同一数据后，

T1由于某种原因被撤销，这时T1已修改过的数据恢复为原值，

T2读到的数据就与数据库中的不一致，为“脏”数据。

> 写后读，写撤回导致读错误。

并发控制措施——封锁

事务T在对某个数据对象操作之前，先向系统**发出请求，对其加锁**，从而对该数据对象有了一定的控制，在事务T释放它的锁之前，其他事务不能更新此数据对象。

> 不能更新的原因是其他事务加锁失败。这也就是一级封锁协议中**读直接无锁**导致**读脏数据**无法阻拦的原因。

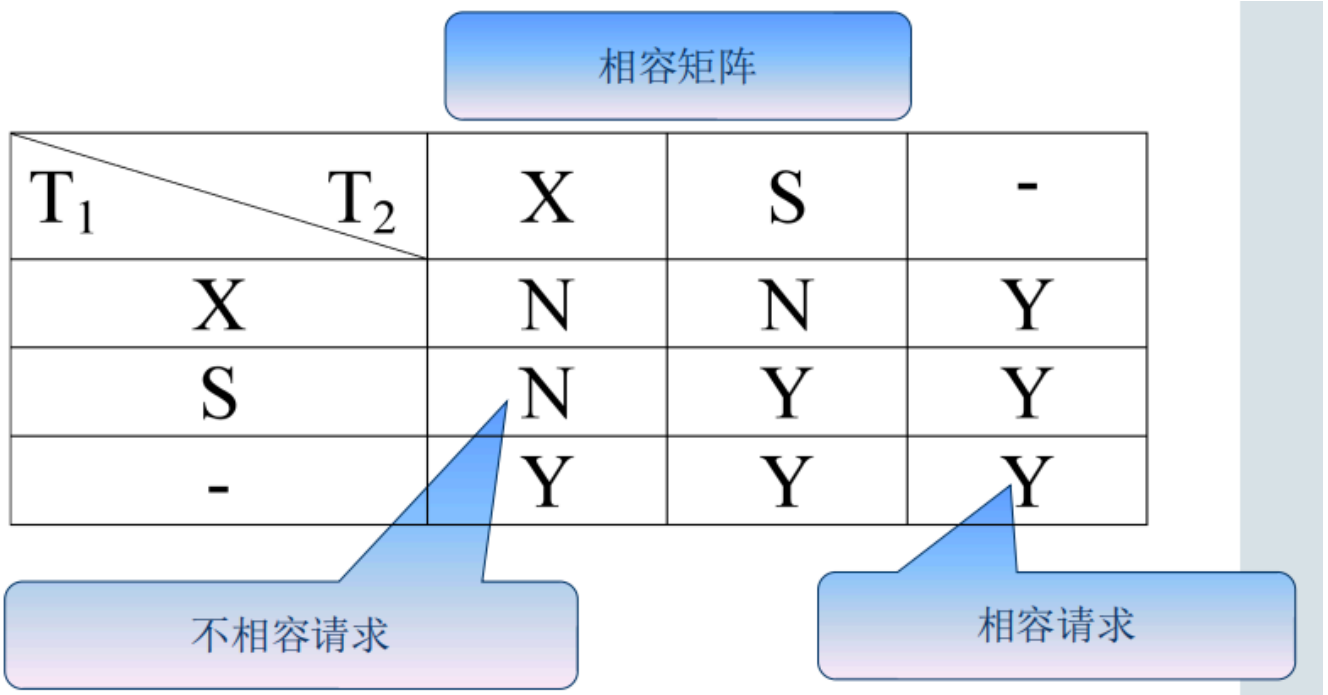
封锁的类型

排它锁（**X锁**，Exclusive lock）：事务T对数据对象A加上X锁，则**只允许T读取和修改A**，其它事务对A的任何**封锁请求**都不能成功（**因而不能**读取和修改R），直至T释放A上的X锁。

共享锁（**S锁**，Share lock）：事务T对数据对象A加上S锁，则**事务T可以读取但不能修改A**，其它事务**只能对A加S锁**（因而可以读取A），而**不能对A的加X锁**（因而不能修改A），直到T释放A上的S锁。

> 注意封锁机制是靠**锁**实现的，锁只能阻拦锁！

相容矩阵：（读读或者数据不同为True）



封锁协议

一级封锁协议（Read Uncommitted）

事务T在修改数据R之前**必须对其加X锁**，直到**事务结束**才释放。

> 事务结束包括正常结束(COMMIT)和非正常结束(ROLLBACK)。

对**更新**提出了很严格的要求，但对于**读取，根本没有锁。**

可解决：丢失修改

不可解决：读脏数据、不可重复读

二级封锁协议（Read Committed）

一级封锁协议 +

事务T在**读取**数据R之前必须先对其**加S锁**，**读完后即可释放**S锁。

> 读完即可释放，而非事务结束！

可解决：丢失修改、读脏数据（读有锁了，肯定写彻底结束后才能读）

不可解决：**不可重复读**（同一事务内，每次读取都单独上锁开锁，读取中间存在缝隙）

三级封锁协议（Repeatable Read）

一级封锁协议 +

事务T在读取R之前必须对其加S锁，直到**事务结束才释放。**

可解决：丢失修改、读脏数据、不可重复读

不可解决：不可重复读之**幻影行**之insert，因为当前是针对于数据的上锁，insert的新的数据未能和读的数据产生冲突。

> 四级封锁协议（Serializable 可串行化）解决了此问题

封锁的危害

封锁级别

封锁级别并非越高越好，极度封锁虽然安全，但会**让并行趋于串行化**，违背初衷。

此外，不安全的行为，尤其**在读操作上**，一定程度上是可允许的。

> 人口统计数据一直在更新，每次数据都是不准的，也都是可接受的。

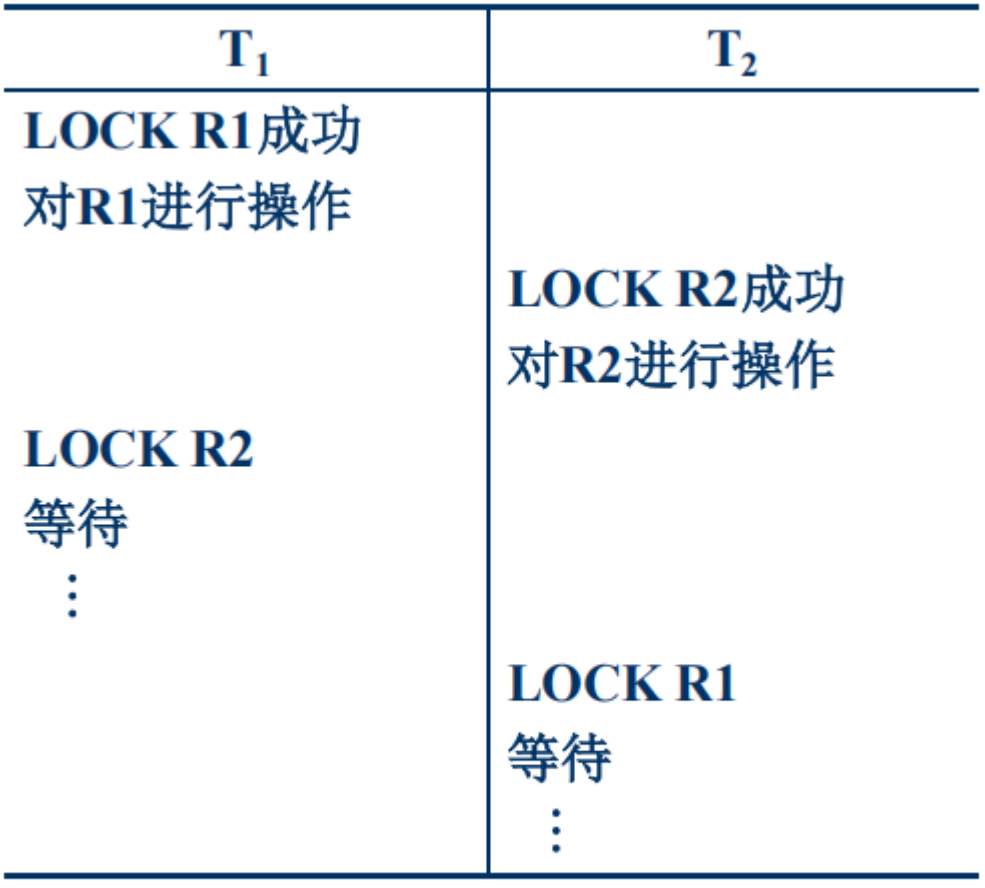
活锁与死锁

活锁是指**某事务**长时间无法申请到锁，从而长期阻塞的问题。**解决方法**：与等待时间相关的**调度**方案。

死锁：

在数据库运行期间，如果存在**一个事务集合{T0，T1，…，Tn}**，使得T0等待T1持有的数据项锁，…，Tn-1等待Tn持有的数据项锁，Tn等待T0持有的数据项锁，则称**系统处于死锁状态**，{T0，T1，…，Tn }称为**死锁事务集合。**

最经典的死锁：



死锁

死锁的解决

预防死锁

一次封锁法

一次封锁法要求每个事务**必须一次**将其所有要使用的数据**全部加锁，否则就不能执行。**

由于需要扩大加锁的范围，因此**降低了系统的并发度。**

T_1	T_2
LOCK R1，R2成功	
对R1，R2进行操作	
	LOCK R1,R2
⋮	等待
COMMIT	等待
UNLOCK R1, R2	等待
	LOCK R1,R2
	对R1,R2进行操作
	⋮
	COMMIT
	UNLOCK R1, R2

顺序封锁法

顺序封锁法是预先对数据对象**规定一个封锁顺序**，所有的事务都要按照这个顺序实行封锁。

由于数据库中数据的不断变化和事务封锁要求的动态提出，**难度往往很大。**

T_1	T_2
LOCK R1成功	
对R1进行操作	
	LOCK R1
LOCK R2成功	等待
对R2进行操作	等待
COMMIT	等待
UNLOCK R1,R2	等待
	等待
	LOCK R1成功
	对R1进行操作
	LOCK R2成功
	对R2进行操作
	COMMIT
	UNLOCK R1，R2

死锁检测与恢复

死锁检测

超时法

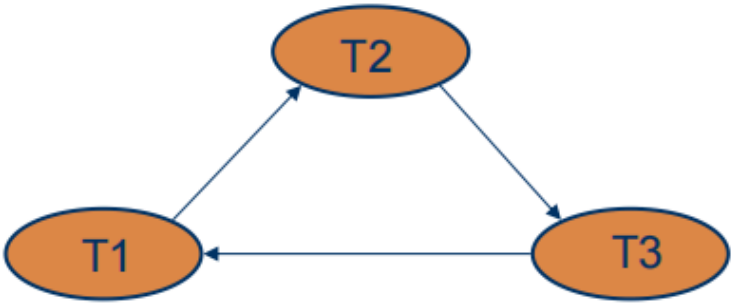
如果一个事务的等待时间超过了规定的期限，就认为发生了死锁。

等待图法

事务等待图是一个有向回路G=(T, U)。T为结点的集合，每个结点表示正在运行的事务；U为边的集合，每条边表示事务等待的情况。**若T1等待T2，则T1，T2之间画一条有向边，从T1指向T2。**事务等待图动态地反映了所有事务的等待情况。

如果发现**图中存在回路**，则表示系统出现死锁。

事务号	占有资源号	请求资源号
T1	R1	R2
T2	R2	R3
T3	R3	R1



死锁恢复

选择一个**处理死锁代价最小**的事务，将其**撤销**，**释放**此事务持有的所有锁，使其他事务得以继续运行下去。

对于所撤销的事务已作的操作必须都加以撤销。死锁结束后再REDO。

事务的调度与可串行性

串行调度，属于**同一事务的指令紧挨**在一起。对于有n个事务的事务组，可以有**n!**个有效调度。

并行调度，来自不同事务的指令**可以交叉执行**，可以有大于**n!**个有效调度。

调度的正确性

T1:读B;A=B+1;写回A;

T2:读A;B=A+1;写回B;

对其进行串行调度：

T_1	T_2
SLOCK B成功	
Y=B= 2	
UNLOCK B	
XLOCK A	
A=Y+1	
写回A(=3)	
UNLOCK A	
	SLOCK A
	X=A=3
	UNLOCK A
	XLOCK B
	B=X+1
	写回B(=4)
	UNLOCK B

T_1	T_2
	SLOCK A
	X=A=2
	UNLOCK A
	XLOCK B
	B=X+1
	写回B(=3)
	UNLOCK B
SLOCK B成功	
Y=B=3	
UNLOCK B	
XLOCK A	
A=Y+1	
写回A(=4)	
UNLOCK A	

串行调度**一定是正确**的，尽管图中两个结果不同，我们也都称之为正确的。

并发执行是正确的，当且仅当**其结果与按某一次序串行的执行它们时的结果相同。**

这种调度策略称为**可串行化调度。**

一个给定的并发调度，当且仅当它是可串行化的，才认为是正确调度。

T_1	T_2
SLOCK B	
Y=B=2	
	SLOCK A
	X=A=2
UNLOCK B	
	UNLOCK A
XLOCK A	
A=Y+1	
写回A(=3)	
	XLOCK B
	B=X+1
	写回B(=3)
UNLOCK A	
	UNLOCK B

T_1	T_2
SLOCK B成功	
Y=B=2	
UNLOCK B	
XLOCK A	
	SLOCK A
A=Y+1	等待
写回A(=3)	等待
UNLOCK A	等待
	X=A=3
	UNLOCK A
	XLOCK B
	B=X+1
	写回B(=4)
	UNLOCK B

图中的左侧调度就不是可串行化调度，不是正确的。而右侧调度则是正确调度、可串行化调度。

如何判断是否是可串行化调度?

冲突可串行化调度

这是一个很硬核的方法：

> 有这样一些指令(组)的**相邻**交换操作，可以不影响运行的结果，并把该调度转为某个串行调度，那么该调度可串行化。

哪些指令的交换操作不影响运行的结果?

- 对**不同**数据对象进行的操作
- 对同一数据对象的**读操作之间**

交换操作可能影响运行的结果的操作叫做**冲突操作。**

冲突操作包括不同事务对**同一数据**的**读写操作和写写操作。**

- Ri（x）与 Wj（x）
- Wi（x）与 Wj（x）

例子：R1(A) W1(A) **R2(A) W2(A)** `R1(B) W1(B)` R2(B) W2(B)

------> R1(A) W1(A) `R1(B) W1(B)` **R2(A) W2(A)** R2(B) W2(B) 变为了串行调度。

这个方法只是一个充分条件。

- 事务T1: W1(Y)W1(X)
- 事务T2: W2(Y)W2(X)
- 事务T3: W3(X)

调度1(串行调度)	调度2（并发调度）
W1(Y) W1(X) W2(Y) W2(X) W3(X)	W1(Y) W2(Y) W2(X) W1(X) W3(X)

上图的例子就是一个典例。当前语境下W1(X)和W2(X)其实可交换，但是属于冲突操作。

两段锁协议

1. 在对任何数据进行读、写操作之前，事务首先要获得对该数据的封锁。
2. 在释放一个封锁之后，事务不再获得任何其它封锁。

也就是两个阶段，加锁阶段与解锁阶段，不存在解锁后再加锁。

第一阶段是获得封锁，也称**扩展阶段**。事务不能释放任何锁。

第二阶段是释放阶段，也称**收缩阶段**。事务不能申请任何锁。

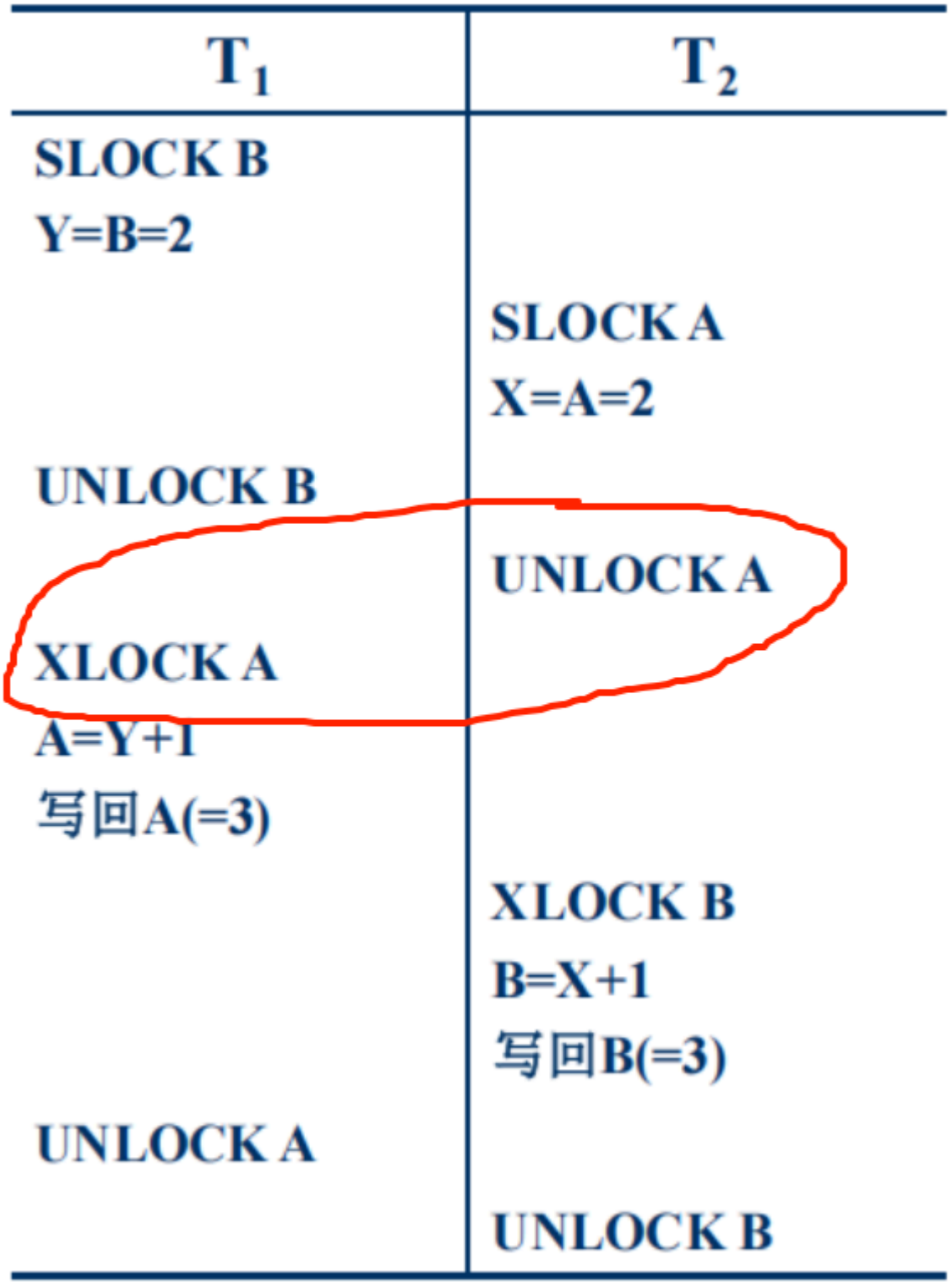
定理：若所有事务均遵从两段锁协议，则这些事务的所有并行调度都是可串行化的。

这个方法也只是一个充分条件，但是他要更加直观简单。

> 串行调度并不满足两段锁协议，本身就是一个反例。

不过我们至少知道：一个调度如果是错误的，那他一定不满足两段锁协议。

比如之前的例子，就显然没有遵循两段锁协议：



两段锁协议**不能用于预防死锁。**

因为只是在加锁阶段就可以直接死锁。

多粒度封锁

封锁对象的大小称为**封锁粒度**。

封锁对象包括**逻辑单元**（属性值、元组、关系……）和**物理单元**（物理页、块）。

封锁粒度大，则**并发度低**，封锁机构简单，**开销小**。

多粒度封锁：在一个系统中同时**支持多种封锁粒度**供不同的事务选择。

选择封锁粒度时应同时考虑**封锁开销和并发度**两个因素，适当选择封锁粒度以达到最优效果。

多粒度树

多粒度树的**根结点**是**整个数据库**，表示最大的粒度。**叶结点**表示最小的粒度。

多粒度树中的每个结点可被独立地加锁，此外：

对一个结点加锁意味着**这个结点的所有后裔结点**也被加以同样类型的锁。

> - 显式封锁是应事务的要求直接加到数据对象上的封锁。
> - 隐式封锁是该数据对象没有独立加锁，是由于其上级结点加锁而使该数据对象加上了锁。

意向锁

一般的，对某个数据对象加锁，系统要

1. 检查该数据对象上**有无显式封锁与之冲突**；
2. 还要检查其所有上级结点的显式封锁，也是否有**隐式封锁**与之冲突
3. 还要检查其所有**下级结点的显式封锁**是否与本次加锁产生的隐式封锁冲突。

这样效率**效率很低**，因此引入了意向锁。

如果对一个结点加意向锁，则说明**该结点的下层结点正在被加锁。**

因为对任一结点加锁时，必须**先对它的上层结点加意向锁。**

意向共享锁（Intent Share Lock，简称IS锁）

表示它的后裔结点拟（意向）加S锁。

意向排它锁（Intent Exclusive Lock，简称IX锁）

表示它的后裔结点拟（意向）加X锁。

意向共享排它锁（ Share Intent Exclusive Lock，简称SIX锁）

表示对它加S锁，再加IX锁，即SIX=S+IX。

相容矩阵

S 只和 S IS 相容。

X一定不相容。

IS 只和 X 不相容。

IX 只和 IS IX 相容。

SIX 只和 IS 相容。

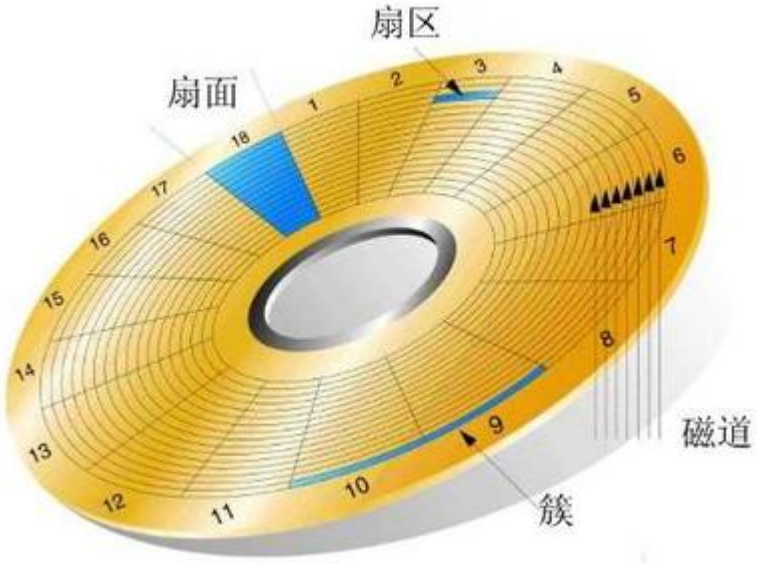
T_1 \ T_2	S	X	IS	IX	SIX	—
S	Y	N	Y	N	N	Y
X	N	N	N	N	N	Y
IS	Y	N	Y	Y	Y	Y
IX	N	N	Y	Y	N	Y
SIX	N	N	Y	N	N	Y
—	Y	Y	Y	Y	Y	Y

数据库存储

存储介质

CPU寄存器-----高速缓存 L1/L2/L3-----主存储器-----闪存-----磁盘-----光盘-----磁带

在此着重讲解磁盘。



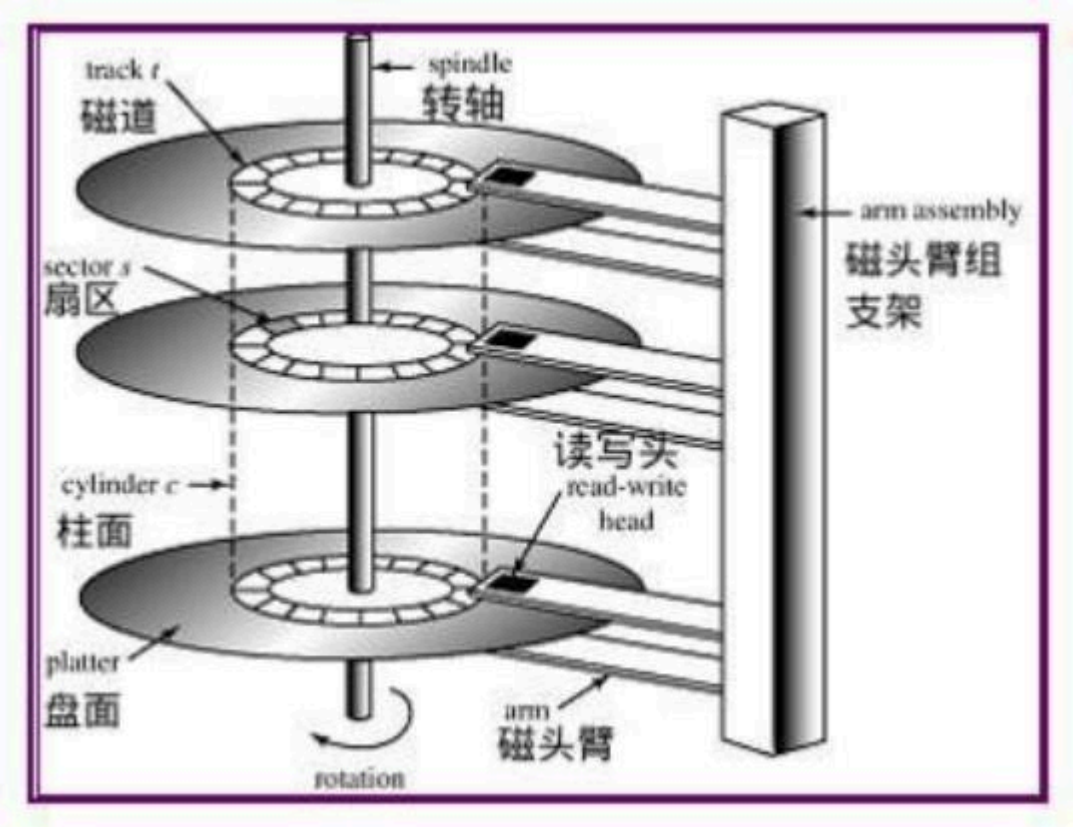
磁道：盘片的表面被逻辑地划分为磁道。

扇区：磁道又被逻辑地划分为扇区；扇区是从**磁盘读出和写入数据的最小单位**，通常大小为512字节。

> 磁头可同时移动扫过多个跨磁道扇区构成扇面。

物理块：一个盘片的一条磁道内**几个连续的扇区**构成的序列称为物理块，一般也简称块；

> 数据在磁盘和主存储器之间**以块为单位传输**。



RAID

磁盘可以构成**磁盘阵列**。

RAID即独立磁盘冗余阵列。将多个单独的物理硬盘**以不同的方式组合成一个逻辑硬盘**，从而提高了硬盘的读写性能和数据安全性。

提升容量、提升性能、提升可靠性

根据不同的组合方式可以分为不同的RAID级别。

RAID 0	数据条带化，无校验
RAID 1	数据镜像，无校验
RAID 2	海明码错误校验及校正
RAID 3	数据条带化读写，校验信息存放于专用硬盘
RAID 4	单次写数据采用单个硬盘，校验信息存放于专用硬盘
RAID 5	数据条带化，校验信息分布式存放
RAID 6	数据条带化，分布式校验并提供两级冗余

RAID0：至少两个盘。提升了性能，没有任何安全性防护。

RAID1：至少两个盘。空间利用率低，读写性能无提升。

RAID2：第1个、第2个、第4个……第2的n次幂个硬盘都是校验盘。

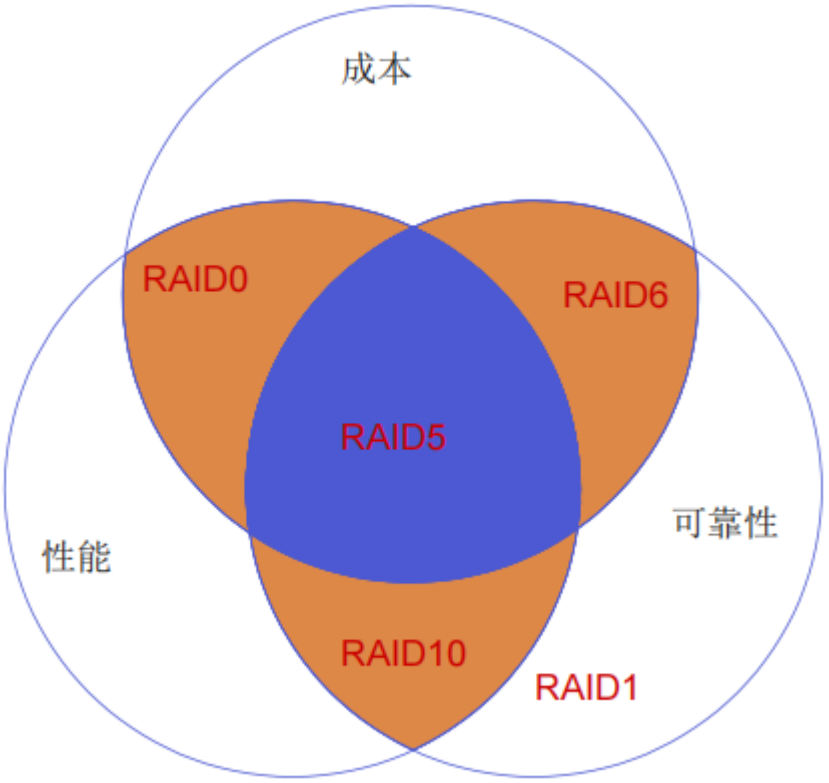
RAID3：专门的校验盘。

RAID5：至少三个盘，校验信息分散在不同磁盘。

RAID6：允许两个硬盘同时失效。

常用 RAID0，RAID1，RAID5，及其组合：

- RAID 0+1：先做RAID 0，后做RAID 1，同时提供数据条带化和镜像
- RAID 10：类似于RAID 0+1，区别在于先做RAID 1，后做RAID 0
- RAID 50：先做RAID 5，后做RAID 0，能有效提高RAID 5的性能



缓冲区调度

OS常用**最近未使用** LRU，但是 LRU 对某些**涉及重复扫描数据**的存取模式来说是**很差**的策略！

对于DB，要根据不同情况选择方案。一般地，如果需要替换，则最好替换**最近使用的块**。

- 支持**钉住**不允许写回磁盘的内存块。如循环连接中的**外层循环**中的数据块。
- 支持**立即替换**策略。完成某任务后直接替换所有相关块。
- 支持**最近使用**策略。如循环连接中的内层循环数据块，只会用到一次，用完直接替换。
- 支持**某些数据**优先存放到缓冲区中，如尽可能将**数据字典块**保存在缓冲区中。

数据库文件结构

数据文件：数据库数据的物理存储。

记录文件

定长记录文件：记录的长度是固定的。

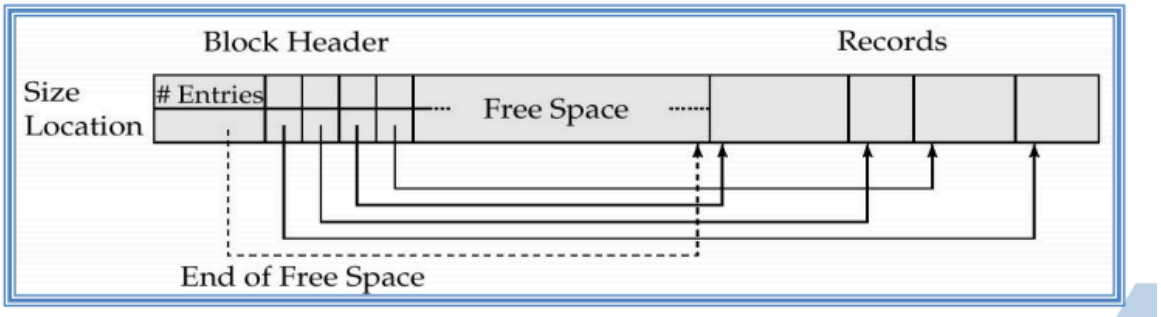
- 优势：记录存取简单。
- 删除记录困难：不应把被删除的记录的后面的记录都往上移，而是应该用链表把被删除的记录串起来。

变长记录文件：文件中能够容纳不同长度的、不同类型的记录。

- 支持变长数据类型：如varchar
- 支持多种记录类型
- 记录存取灵活，空间利用率高
- **Slotted Page结构**：

❖ **Slotted page 页头包含：**
- 该块中记录个数
- 该块中自由空间末端
- 每条记录的位置和大小

❖ **记录可以在页内移动以便保持记录间的连续存储；页头中的记录的位置信息必须更新.（移动代价不高）**



- **定长表示法**：使用一个或多个定长记录表示一个变长记录
  - 预留空间法：利用具有**已知最大长度**的定长记录
  - 指针法：**若干定长**记录通过指针链在一起

记录文件的组织

- 堆：记录可置于文件中的**任何有空间**的地方
- 顺序：记录**按顺序存储**，基于每条记录在搜索码上的值
- 散列（hash）：对记录的某属性计算hash函数，计算结果决定该记录应该置于文件的哪个块中

每个关系的记录可存储在单独的文件中。

但在**聚簇文件组织**下, 单个和多个关系的记录**都可存储于同一文件**中。

> 语义关联密切的记录存储在同一块中**可减少磁盘 I/O**

因为物理顺序只能有一个，聚簇应该用在最需要的位置。

顺序文件组织

适用于需要顺序地高效处理**按照某个搜索码排列**的整个文件的记录。

通过**指针**，同时尽可能保证**逻辑次序和物理次序**保持一致，减少可能的磁盘操作。

维护记录的物理顺序代价高：虽然用指针维护顺序，但还是需要时常**重组文件**以恢复物理存储顺序。

关系数据库的文件选择

存储关系到一个文件

- **充分利用OS**所提供的功能，减少DMBS的处理代价
- DBMS不支持某些优化策略

存储多个关系到一个文件

- DBMS支持某些优化策略，可以在文件中设计复杂的结构
- 数据库的**大小和复杂性**增大

数据字典

数据字典，也称为系统目录，保存元数据，即**关于数据的数据**，也是需要存储的。

内容包括**关系**的信息、**用户及账号**信息与口令、**统计与描述**数据、**物理文件组织**信息、**索引**信息。

索引

基本概念

索引机制用于**加速对所需数据的存取。**

更新文件时，该文件上的**每个索引都必须更新**，有一定开销。

搜索码：用来在文件中查找记录的**属性或属性集合。**

索引文件一般比原始文件小的多。

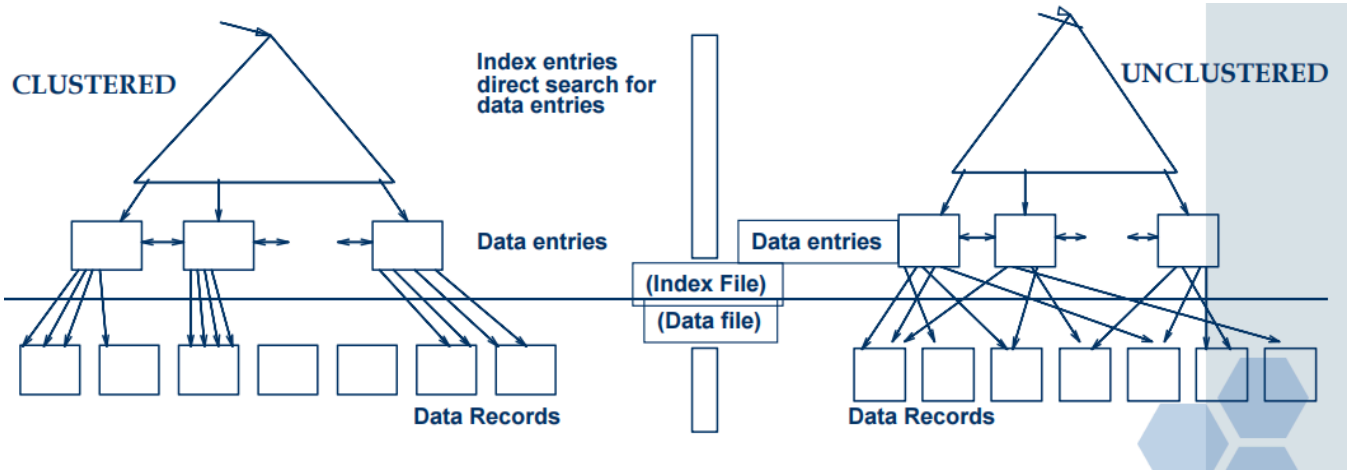
两种基本索引

- 有序索引：搜索码**按顺序**存储
- 散列索引：搜索码**被散列函数分配**到若干桶中.
- new：位图索引。实现高效的**多键查询。**将数据转储为二进制01串，利用交并补运算实现查询。

评价手段:

- 有效支持的存取类型：**Point Query**（属性值为某值的记录）、**Range Query**（属性值位于某范围的记录）
- 存取时间、插入时间、删除时间
- 空间开销

索引分为**主索引**与**辅助索引**。



主索引

主索引：顺序文件的**记录顺序**正是索引**搜索码的顺序**，也称为**聚簇索引**。

- 一个关系只能有一个主索引。
- 主索引的搜索码通常是主码，但并非必要。
- 索引顺序文件：带有主索引的顺序文件。

多级索引

如果主索引不能一次放入内存，存取代价就会很大。

将主索引视为**存储在磁盘上的顺序文件**并为它**建立一个稀疏索引**。

- 外索引：主索引的稀疏索引
- 内索引：主索引文件

如果外索引仍太大而不能放入内存，还可再为它创建另一层索引。

> 当插入或删除文件记录时，**各层索引**都必须更新。

辅助索引

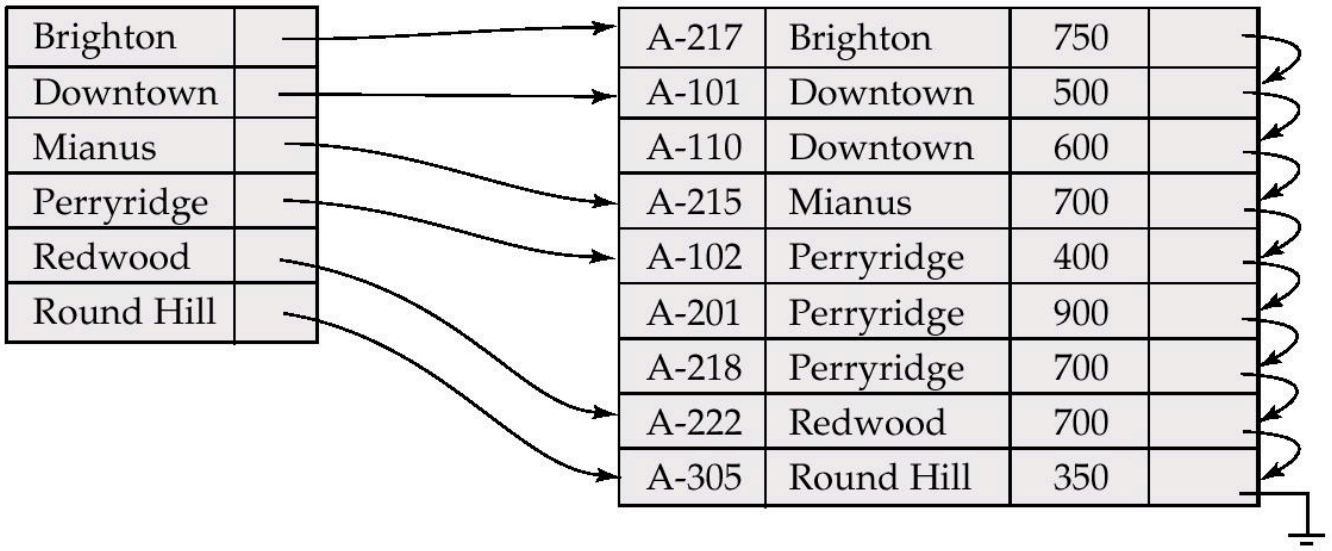
辅助索引：索引**搜索码的顺序与文件的记录**顺序不同，也称为非聚簇索引。

- 对每个搜索码值都可有一辅助索引记录。
- **辅助索引必须是稠密的。**

利用**主索引做顺序扫描**很高效，但利用辅助索引做顺序扫描则代价昂贵，因为按照辅助索引搜索码进行扫描，其对应的**物理位置并不紧密连接**，对每个记录的存取都可能导致存取一个新的磁盘块。

有序索引

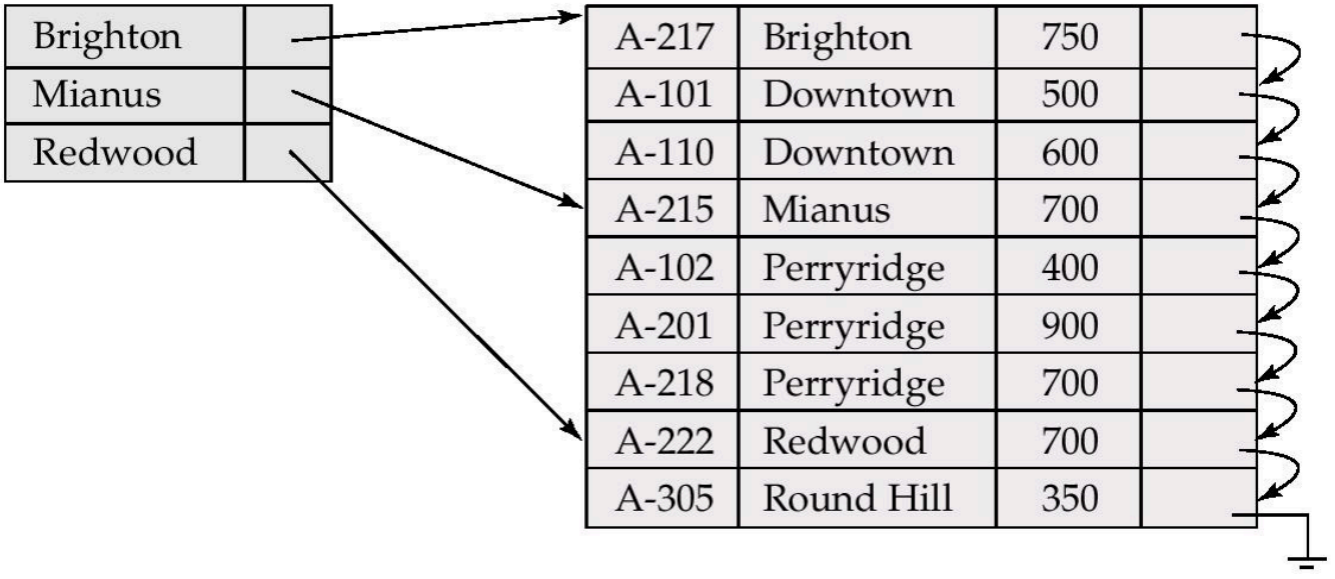
稠密索引文件



文件中的**每个搜索码值都有索引**记录。

稠密索引的数据可以不按照索引数据物理存放。

稀疏索引文件



只对某些搜索码值有索引记录。

文件记录必须**按搜索码排序**才可用，原因是：定位具有搜索码值K的记录时，需要

- 找到具有**比K小**的**最大**搜索码值的索引记录，从该索引记录指向的文件记录**开始顺序搜索**文件。

特点：好建不好用。

- 插入和删除需要**较少空间与维护开销**.。
- 查找记录**一般比稠密索引慢。**

B+树索引文件

索引顺序文件如果采用指针维护，需要定期重建（删除所有索引并重新创建），开销很大。

B+树索引结构是**使用最广泛**的在数据**插入和删除**情况下能够**保持执行效率**的索引结构之一。

优点：插入与删除时仅以**较少的局部的变化**来自动重组，不需要整个文件重组来维持性能。

缺点：额外的**插入与删除开销**，**空间开销**。

- **B+-树是满足下列性质的树：**
  - 从根到叶的所有路径长度相同
  - 每个非根非叶节点有$\lceil n/2 \rceil$到n 个子节点.
  - 叶节点有$\lceil (n-1)/2 \rceil$到$n-1$ 个值
  - 特殊情况:
    - 若根非叶, 则至少有2 个子节点.
    - 若根是叶(即树中没有其他节点), 则可有0 到$(n-1)$ 个值.
- **典型节点**

  - K_i 是搜索码值
  - P_i 是子节点指针 (非叶节点中)或记录/记录桶指针(叶节点).
- **每个节点中搜索码是有序的**

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

从树根到叶节点的**路径长度相同**，所有叶节点都在同一层，也就是访问不同数据**所花费的时间是相对稳定**的。

处理查询时，需遍历一条从根到叶的路径.
若文件中有K 个搜索码值，则路径长度不超过$\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.

Hash数据索引

一个桶是一个存储单元，包含一个或者多个记录 (桶的大小一般是磁盘块大小)

我们根据搜索码的值，利用Hash函数**直接得到搜索码所在的桶**，需要**顺序扫描**该桶来定位到一个记录。

Hash索引通常是**辅助索引**。

尽管桶溢出的可能性会减少，但是绝不可能完全消除，通过溢出桶来解决.。

> 溢出桶：一个桶的溢出桶集合通过链连接起来

- 静态索引技术。可能需要周期性重组。
- 动态Hash表，使用可扩展hash函数，通过桶的分解或合并来适应数据库大小的变化

总结

给定数值进行确定性查询：hash

range查询：有序索引

B+树索引对于给定**数值的查询**和**范围查询**都是比较有效的，但在数据的**插入、删除**时动态调整代价高。

多属性索引中，第二关键字意义远弱于第一关键字。