第 1 章：引言

数据：描述事物的符号记录，数据库的基本对象。

数据库：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。

数据库系统：在计算机系统中引入数据库后的系统，由数据库、数据库管理系统、应用系统（及其开发工具）、数据库管理员（和用户）构成。

文件处理系统的弊端：数据的冗余和不一致、数据访问困难、数据孤立、完整性问题、原子性问题、并发访问异常、安全性问题

数据模型：数据库结构的基础。有关系模型、实体 - 联系模型、基于对象的数据模型、半结构化数据模型四种，网状模型和层次模型已经被淘汰。

存储管理器的数据结构：数据文件（存数据库）、数据字典（存元数据，是数据库的数据库）、索引
数据文件可以借助操作系统以文件形式存在本地，也可以不借助操作系统直接存入磁盘。

数据抽象的三个层次：**物理层**（描述数据真实的存储方式和底层复杂的数据结构）、**逻辑层**（描述数据间的关系和数据库中存储什么数据）、**视图层**（只描述整个数据库的某一部分）
根据描述数据库设计的层次，数据库系统可分为不同模式：**物理模式**、**逻辑模式**、**子模式**

物理数据独立性：应用程序不依赖物理模式，物理模式隐藏在逻辑模式下，可以在应用程序不受影响的情况下轻易更改。即使物理模式改变了，也无需重写应用程序。

数据库的层次结构：客户 / 服务器式数据库可以分为**两层或三层**。两层 - 前端 / 后端，前端直接和后端的数据库通信，客户机上的应用程序通过 SQL 来调用数据库（JDBC/ODBC）；三层 - 前端 / 应用服务器 / 数据库服务器，应用程序的业务逻辑嵌入到应用服务器内，客户端通过表单与应用服务器通信，应用服务器与数据库通信以访问数据。

数据库 DBA：模式定义、存储结构及存取方式定义、模式及物理组织修改、数据访问授权、日常维护（定期备份等）

数据操纵语言 DML（增删改查）、**数据查询语言** DQL（查询，select）、**数据定义语言** DDL（create）、**数据控制语句** DCL（grant, revoke 等）

第 2 章：关系模型

关系模型

关系模型中，唯一的数据结构是**关系**，即二维表。

元组：行；**属性**：列；**属性的域**：某一属性的取值集合。所有属性的域都必须是**原子的**。

数据库模式：数据库的逻辑设计；**数据库实例**：某一时刻数据库中的数据的快照。**关系模式**、**关系实例**与之类似。

超码：一个或多个属性的集合，这个集合可以唯一地区分出一个元组（一行）

候选码：最小超码，可能有多个；**主码**：人为选中，作为一行的区分标准的候选码。

外码：关系r_1的属性中可能包含了关系r_2的主码，这个属性在r_1上称作参照r_2的外码，r_1称作外码依赖的**参照关系**，r_2称作外码依赖的**被参照关系**

参照完整性约束：参照关系中的任意元组的特定属性的取值必须等于被参照关系中某个元组的特定属性的取值。

查询语言：用户用来从数据库中请求获取信息的语言

过程化语言：用户指导系统对数据库执行一系列操作来计算结果

非过程化语言：用户只描述所需信息，不给出获取该信息的具体过程

关系代数

过程化查询语言，基本运算：选择、投影、并、差、笛卡尔积、更名。**关系运算的结果也是关系。**

选择 选出满足给定谓词的元组。 $\sigma_{dept\_name="Physics"}(instructor) = \neq <>$
可以用 $\wedge$ (与), $\vee$ (或), $\neg$ (非) 合并多个谓词。 $\sigma_{dept\_name="Physics" \wedge salary>9000}(instructor)$

投影 返回修改过的关系，排除不符合要求的属性。$\Pi_{ID,name}(instructor)$

可以将选择和投影组合起来实现更复杂的运算。$\Pi_{name}(\sigma_{dept\_name="Physics"}(instructor))$

并 将两个集合并起来。$\Pi_{name}(\sigma_{dept\_}(instructor)) \cup \Pi_{name}(\sigma_{salary>9000}(instructor))$
要使并运算$r \cup s$有意义，必须满足：

1. r 和 s 是同元的，即属性数量相同；
2. 对所有的 i，r 中第 i 个属性的域与 s 中第 i 个属性的域相同。

差 找出一个集合中有而另一个没有的元组。$\Pi_{name}(\sigma_{dept\_name="Physics"}(instructor)) - \Pi_{name}(\sigma_{salary>9000}(instructor))$
与并运算相同，差运算也要求同元、相容。

笛卡尔积 将任意两个关系的信息组合在一起。$instructor \times student$

更名 给关系代数表达式的结果赋上名字。$\rho_x E$ 返回表达式 E 的结果，并为其命名为 x.
也可以同时为关系中的每个属性更名，假设 E 是 n 元的，$\rho_{x(A_1,A_2,...A_n)}(E)$

除了基本关系运算，还有一些附加的、简化查询的运算：交、自然连接、赋值、外连接

交 找出两个集合的交集。$\Pi_{name}(\sigma_{dept\_name="Physics"}(instructor)) \cap \Pi_{name}(\sigma_{salary>9000}(instructor))$
与差、并相同，要求同元、相容。

自然连接 对笛卡尔积的结果进行简化，要求两个关系中相同属性的值一致。$\Pi_{name,ID}(instructor \bowtie teacher)$

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

赋值 将关系存入一个临时变量。$temp1 \leftarrow R \times S$、$temp2 \leftarrow \Pi_{A_1}(temp1)$
赋值运算不会把结果展示给用户，而是将右侧表达式结果存入左侧变量，以备后续使用。

外连接 自然连接的扩展，它会在结果中创建带空值的元组，保留自然连接中可能丢失的元组。
左外连接、右外连接、全外连接

第 3~5 章：SQL

1. SQL 的概述

SQL（structured Query Language）结构化查询语句，是关系型数据库的标准语言，SQL 是一个通用的、功能极强的关系数据库语言。

2. SQL 的发展

1986 年发布第一个版本：SQL/86

> 目前，没有任何一个关系型数据库系统能支持所有标准的概念和特性

3. SQL 的特点

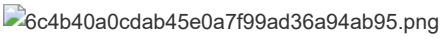
1. 综合统一
2. 高度非过程化
3. 面向结合的操作方式
4. 以同一种语法结构提供两种使用方式，SQL 语言既是自含式设计语言，又是嵌入式语言
5. 语言简洁，易学易用

4. SQL 的功能

SQL 功能	关键词
DDL	create,drop,alter
DML	insert,update,delete
DQL	select
DCL	grant,revoke

5. SQL 的结构

SQL 支持关系型数据库的三级模式结构：



6. SQL 的数据类型

数据类型	含义
char(n),character(n)	长度为 n 个字节的定长字符串
varchar(n),charactervarying(n)	最大长度为 n 个字节的变长字符串
CLOB	字符串大对象
BLOB	二进制大对象
int，integer	长整数（4 字节）
smallint	短整数（2 字节）
bigint	大整数（8 字节）
numeric(p,d), decimal(p,d),DEC(p,d)	定点数，由 p 位数字 (不包括符号、小数点）组成，小数点后面又 d 位数
real	取决于机器精度的单精度浮点数
double precision	取决于机器精度的双精度浮点数
float(n)	可选精度的浮点数，精度至少为 n 位数
boolean	逻辑布尔量
date	日期，包含年、月、日，格式为 YYYY-MMDD
time	时间，包含一日的时、分、秒，格式为 HH-MM-SS
timetamp	时间戳类型
interval	时间间隔类型

7. 数据定义语言 DDL

一个关系数据库管理系统的实例（instance）中可以建立多个数据库，一个数据库可以建立多个模式，一个模式下通常包含多个表、视图和索引等数据库对象。因此 SQL 的数据定义功能包括定义模式、定义表、定义视图和定义索引。视图式基于基本表的虚表，索引式依附于基本表， 因此 SQL 通常不提供修改视图定义和修改索引定义的操作。用户如果想修改视图或修改索引定义，只能先将原来的视图或索引删除，然后重新定义。不过也有存在提供修改视图或索引语句的关系型数据库管理软件。

表：SQL 数据定义语句

操作对象也	创建	删除	修改
模式	create schema	drop schema	–
表	create table	drop table	alter table
视图	create view	drop view	–
索引	create index	drop index	–

7.1 模式

7.1.1 模式定义

定义模式实际上是定义了一个命名空间，在这个空间中可以定义该模式包含的数据库对象，如基本表、视图、索引等。
语法格式：
create schema <模式名> authorization < 用户名 > [< 表定义子句 >|< 视图定义子句 >|< 授权定义子句 >];

实例：为用户 Tom 创建一个模式 TEST，并且在其中定义一个表 Student。
create schema TEST authorization Tom create table Student(

number int,
name varchar(20),
age int);

7.1.2 模式的删除

语法格式：
drop schema <模式名> <cascade|restrict>;

> cascade: 级联，删除模式等同时把该模式中的所有的数据库对象全部删除。
> restrict：限制，如果该模式中定义了下属的数据库对象（如基本表、视图、索引等），数据库管理系统拒绝该条删除模式等语句执行，仅当该模式中没有任何下属对象时才能被执行。

实例：删除模式 TEST，同时将该模式中的 Student 表一起删除。

```
drop schema TEST cascade；
```

7.2 基本表

7.2.1 基本表定义

建立数据库最重要的一步就是定义一些基本表，SQL 语言使用 create table 关键字进行基本定义。
语法格式：

```
create table <表名> (
<列名> < 数据类型 > [< 列级完整性约束条件 >]
[,< 列名>< 数据类型 >[< 列级完整性约束条件 >]
……
[, < 表级完整性约束条件>]
)；
```

实例 1：创建一个学生表 Student，学号是主码，姓名取值唯一。
先创建一个模式：

```
create schema T-S authorization Tom;
```

```
create table Student(
Sno char(9) primary key,
Sname char(20) unique,
Ssex char(2),
Sage smallint,
Sdept char(20)
);
```

实例 2: 创建一个课程表 Course，课程号是主码。

```
create table Course(
Cno char(4) primary key,
Cname char(40),
Cano char(4),
Ccredit int
);
```

实例 3: 创建一个学生选课表 SC，学生号和课程号组成主码。

```
create table SC(
Sno char(9);
Cno char(4)
Grade decimal(40,2),
primary key(Sno,Cno),
foreign key(Son) references Student(Sno),
foreign key(Cno) references Course(Cno)
);
```

** 模式与表 **
任何一个表都属于某一个模式，一个模式包含多个基本表，定义基本表所属模式有三种方式：

1. 在定义表是显示给出模式名：
   create table “S-T”.Student(…);
   crate table “S-T”.Cource(…);
   create table “S-T”.SC(…);
2. 在创建模式语句是同时创建表
3. 设置所属模式
   在数据库管理系统中创建基本表或其他对象时，若没有指定模式，系统会根据搜索路径来确定该对象所属的模式，关系数据库管理系统会使用模式里表中第一个存在的模式作为该数据库对象的模式名，若搜索路径中的模式名不存在，系统将给出错误。

7.3.3 基本表的修改

语法格式：

```
alter table <表名>
[add [column] <新列名> <数据类型> [列级完整性约束条件]]
[add <表级完整性约束条件>]
[drop [column] <列名> [cascade|restrict]]
[drop constraint <完整型约束条件> [cascade|restrict]]
[alter column <列名> <数据类型> ];
```

实例 1：向 Student 表增加 “入学时间” 列，其数据类型为日期类型
不管基本表中原来是否有数据，新增加的列一律为空

```
alter table Student
add S_entrance date;
```

实例 2：将年龄的数据类型字符型 (假设原来的数据类型是字符型）改为整数型。

```
alter table Student
alter column Sage int;
```

实例 3：增加课程名必须取唯一的约束条件

```
alter table Course
add unique(Cname);
```

7.2.3 基本表的删除

语法格式：

```
 drop talbe <表名> [cascade|restrict];
```

cascade(级联): 在删除基本表的同时，相关依赖这个表的对象一起删除；
restrict(限制): 在删除基本表时，会验证该表是否别其他对象依赖，若存在其他对象依赖，则删除该表失败

实例：删除 Student 表

```
drop table Student cascade;
```

7.3 视图

7.3.1 视图的定义

视图是一个虚拟表，其内容查询定义。但视图并不存储数据，视图数据来自定义视图的查询所引用的表。关系数据库管理系统执行 create view 语句时只是把视图定义存入数据字典，并不执行其中的 select 语句。在对视图查询时。视图对重构数据库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，适当的利用视图可以清晰的表达查询。
语法格式：

> create view <视图名> [(<>[,<>]…)] as
> <子查询> [with check option];
> with check option : 对视图进行 update，insert 和 delete 操作时要保证更新、插入或删除的行满足视图定义中的谓词条件（既子查询中的条件表达时）。子查询可以时任意的 select 语句，是否可以含有 order by 子句和 distinct 短语，则决定具体系统的实现。

实例 1: 建立信息系学生的视图

```
create view IS_Student as
select Sno,Sname,Sage
from Student
where Sdept='IS';
```

实例 2：建立信息系学生的视图，并要求进行修改和插入操作时任需保证该视图只有信息系的学生。

```
create view IS_Student as
select Sno,Sname,Sage
from Student
where Sdept = 'IS' with check option;
```

实例 2：建立信息系选修 1 号课程的学生的视图（包括学号，姓名、成绩）。

```
create view IS_S1(Sno,Sname,Grade) as
select Student.Sno,Sname,Grade
from Student,SC
where Sdept = 'IS'  and Student.Sno=SC.Sno and SC.Cno = '1';
```

实例 3：建立信息系选修 1 号课程且成绩在 90 分以上的学生视图。

```
create view IS_S2 as
select Sno,Sname,Grade from IS_S1 where Grade >= 90;
```

实例 4：定义一个反映学生出生年月的视图

```
create view BT_S(Sno,Sname,Sbirth) as
select Sno,Sname,2021-Sage from Student;
```

实例 5：将学生的学号及平均成绩定义为一个视图。

```
create view S_G(Sno,Gavg) as
select Sno,AVG(Grade)
from SC group by Sno;
```

7.3.2 视图的删除

```
drop view <视图名> [cascade];
```

> cascade：如果该视图还有其他视图进行引用，则把这个给视图同时一起删除。

实例 1：删除视图 BT_S。

```
drop view BT_S;
```

7.4 索引

7.4.1 索引定义

建立索引是加快查询速度的有效手段。 用户可以根据需要应用环境的需要，在**基本表上建立一个或多个索引**，以提供多种存取路径，加快查找速度。一般来说，建立与删除索引有数据库管理员 DBA 或表的属主负责完成。系统在存取数据是会自动选择适合的索引作为存取路径 用户不必也不能选择索引进行查找。

> 在 SQL 中，建立索引使用 create index 语句。索引可以建立在该表的一列或多列上，各列之间用逗号分隔。每个列名后面还可以用次序指定索引值的排序次序，可选 asc（升）或 desc（降）, 缺省时默认 asc。

语法格式：

```
create [unique] [cluster] index <索引名>
on <表名>(<列名> <次序>[,<列名> <次序>]...);
```

> 索引：可以建立在该表的一列或多列上，各列名之间用逗号分隔开。
> unique: 此索引的每一个索引值只对应唯一的数据记录
> cluster：表示建立的索引是聚簇索引。聚簇索引是指索引顶的顺序与表中记录的物理顺序一致的索引组织。用户可以在最经常查询的列上建立聚簇索引以提高查询效率。显然在一个基本表上最多只能建立一个聚簇索引。建立聚簇索引后，更新索引数据时，往往导致表中记录的物理顺序变更，代价较大，因此对于经常更新的列不宜建立聚簇索引。

实例：为 Student，Course，SC 三个表建立索引。Student 表按学号升序建立唯一索引，Course 表按课程号升序建唯一索引，SC 表按序号升序和课程好降序建唯一索引。

```
create unique index Stusno
on Student(Sno);
create unique index Coucno
on Course (Cno);
create unique index Scno
on SC(Sno asc,Cno desc);
```

7.4.2 名称的修改

语法格式：

```
alter index <旧索引名> rename to <新索引>;
```

实例：将表 SC 的 SCno 索引名改位 SCSno。

```
alter index SCno rename to SCSno;
```

7.4.3 索引的删除

删除索引时，系统会从数据字典中删去有关该索引的描述。
语法格式：

```
drop index <索引名> on <表名>;
```

实例：删除 Student 表的 Stusno 索引。

```
drop index Stusno on Student;
```

8. 数据查询语言（DQL）

8.1 单表查询

语法格式：

```
select [all|distinct] <目标列表达式>[,<目标列表达式>] ...
from <表名或视图名> [,<表名或视图名>] ... | （select 语句）[as] <别名>
[where <条件表达式>]
[group by <列名1> [having <表达式>]]
[order by <列名2> [asc|desc]];
```

> select 子句：指定要显示的属性列
> from 子句：指定查询对象（基本表、视图、查询子句）
> where 子句：指定查询条件
> group by 子句：**对查询结果**按**指定列的值分组**。 该属性列值相等的元组为一个组。通常会在每组中作业聚集函数。
> having 短语：只是 group by 子句列，满足指定条件的组才给予输出。
> order by 子句：对查询结果表按指定列值升序或降序排序。

常用的查询条件：

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
逻辑运算	and，or，not

8.1.1 简单查询

实例 1：查询全体学生学号与姓名。

```
select Sno，Sname from Student;
```

实例 2：查询全体学生的姓名、学号、所在系。

```
select Sno,Sname from Student;
```

实例 3：查询全体学生的详细记录。

```
select * from Student;
```

实例 4：查询全体学生的姓名及出生年份；

```
select Sname,2022-Sage from Student;
```

实例 5：查询全体学生的名字、出生年份和所在院系，要求小写字母表示系名。

```
select Sname,'Year of Birth',2022-Sage,LOWER(Sdept) from Student;
```

8.1.2 条件查询

实例 1：查询计算机科学系全体学生的名单。

```
select Sname from Student where Sdept='CS';
```

实例 2：查询所有年龄在 20 岁以下的学生及其年龄。

```
select Sname,Sage from Student where Sage<20;
```

实例 3：查询考试成绩有不及格的学生的学号。

```
select distinct Sno from SC where Grade<60;
```

实例 4：查询年龄在 20~23 岁（包括 20 岁到 23 岁）之间的学生的姓名、系别和年龄。

```
select Sname,Sdept,Sage from Student where Sage between 20 and 23;
```

实例 5：查询年龄不在 20~23 岁之间的学生姓名、系别和年龄。

```
select Sname,Sdept,Sage from Student where Sage not between 20 and 23;
```

8.1.3 IN 子句查询

实例 1：查询计算机科学系 CS、数学系 MA 和信息系 IS 学生的姓名和性别。

```
select Sname,Ssex from Student where Sdept in ('CS','MA','IS');
```

实例 2：查询既不是计算机科学系、数学系，也不是信息系的学生的姓名和性别。

```
select Sname,Ssex from Student where Sdept not in ('CS','MA','IS');
```

8.1.3 模糊查询

谓词：[not] like '< 匹配串 >' [escape '< 转码字符 >']

- %: 表示任意长度字符串（长度可为 0）
- _: 表示**任意单个字符**（一定要有一个字符）

实例 1：查询学号为 201215121 的学生详细情况。（没有 %、_时与 "=" 等价）。

```
select * from Student where Sno like '201215121';
select * from Student where Sno = '201215121';
```

实例 2： 查询所有姓刘学生的姓名、学号和性别。

```
select Sname,Sno,Ssex from Student where Sname like '刘%';
```

实例 3：查询姓 "欧阳" 且全名为四个汉字的学生的姓名。

```
select Sname from Student where Sname like '欧阳__';
```

实例 4：查询 DB_Design 课程的课程号和学分。

> 使用换码字符将通配符转义为普通字符

```
select Cno,Ccredit from Course where Cname like 'DB#_Design' escape '#';
```

实例 5：查询以 “DB_” 开头，并且倒数第三个字符为 i 的的详细情况。

```
select * from Course where Cname like 'DB#_%i__' escape '#';
```

8.1.5 空值查询

实例 1：某些学生选修课程后没有参加考试，所以有选课记录，但没有考试成绩，查询缺少成绩的学生序号和相应的课程号。

```
select Sno,Cno from SC where Grade is null;
```

实例 2：查询所有有成绩的学生学号和课程号。

```
select Sno,Cno from SC where Grade is not null;
```

8.1.6 与或查询

实例 1：查询计算机系年龄在 20 岁以下的姓名。

```
select Sname from Student where Sdept='CS' and Sage<20;
```

实例 2：查询计算机科学系、数学系、信息系学生的姓名和性别。

```
select Sname,Ssex from Student where Sdept='CS' or Sdept='MA' or Sdept='IS';
```

8.1.7 排序查询

实例 1：查询选修了 3 号课程的学生学号及成绩，查询结果按分数降序排序。

```
select Sno,Grade from SC where Cno='3' order by desc;
```

实例 2：查询全体学生情况，查询结果按所在系的系号升序排序，同一系中的学生按年龄降序排序。

```
select * from Student order by Sdept asc,Sage desc;
```

8.1.8 聚合查询

实例 1：查询学生总人数。

```
select count(*) from Student;
```

实例 2：查询选修了课程的学生人数。

```
select count(distinct Sno) from SC;
```

实例 3：计算 1 号课程的学生平均成绩

```
select avg(Grade) from SC where Cno='1';
```

实例 4：查询选修 1 号课程的学生最高分数

```
select max(Grade) from SC where Cno='1';
```

实例 5：查询学生 201215012 选修课程的总学分数。

```
select sum(Ccredit) from SC,Course Sno='201215012' and SC.Cno=Course.Cno;
```

8.1.9 分组查询

实例：求各个课程号及相应的选课人数

```
select  Cno,count(Sno) from SC group by Cno;
```

8.1.10 过滤查询

实例：查询选修 3 门以上课程的学生号

```
select Sno from SC group by Sno having count(*) >3;
```

实例：查询平均成绩大于等于 90 分的学生学号和平均成绩

```
select Sno,avg(Grade) from SC group by Sno having avg(Grade)>=90;
```

8.2 连接查询

8.2.1 等值连接

实例：查询每个学生及其选修课程的情况

```
select Student.*,SC.* from Student,SC where Student.Sno=SC.Sno;
```

8.2.2 非等值连接

实例：查询选修 2 号课程且成绩在 90 分以上的所有学生的学号和姓名。

```
select  Student.Sno,Student.name from Studnet,SC where SC.Grade >= 90 and SC.Cno='2' and Student.Sno=SC.Sno;
```

8.2.3 自身连接

实例：查询每一们课程的间接先选课 (先休的先休课程）。

```
select a.Cno,b.Cpno from Course a,Course b where a.Cpno=b.Cno;
```

8.2.4 外连接

实例：查询每一个学生及其选修课程的情况

```
select  a.*,b.Cno,b.Grade from Student a left join SC b on (Studnet.Sno=SC.Sno);
```

8.2.5 多表连接

实例：查询每一个学生的学号、姓名、选修的课程名及成绩。

```
select Student.Sno,Sname,Cname,Grade from Student SC，Course where Student.Sno=SC.Sno and SC.Cno=Course.Cno;
```

8.3 嵌套查询

一个 select-from-where 语句称为一个查询块，将一个查询块嵌套在另一个查询块的 where 字句或 having 短语的条件中的查询称为嵌套查询，上层的查询块称为外层查询或父查询，下层查询块称为内层查询或子查询，SQL 语言允许多层嵌套查询，既一个子查询还可以允许嵌套其他子查询，子查询不能使用 order by 字句。

- 不相关子查询：子查询的查询条件不依赖于父查询，由里向外逐层处理。既每一个子查询在上级查询处理之前求解，子查询的结果用于建立其子查询的查找条件。
- **相关子查询：子查询的查询条件依赖于父查询，首先取外层查询中的表的第一个元组，根据它与内层查询相关的属性值的属性值处理查询内层查询，若 where 子句返回的值为真，则取此元组放入查询结果表，然后再取外层表的下一个元组，重复这一过程，直至外层表全部检测完为止。**

8.3.1 带有比较运算符的子查询

实例：找出每个学生超过他选修课程平均成绩的课程号 (相关查询)

```
select  Sno,Cno from SC a where Grade >= (select avg(Grade) from SC b where a.Sno=b.Sno);
```

8.3.2 带有 IN 谓词的子查询

实例：查询与 “刘晨” 在同一个系学习的学生。

```
select Sno,Sname,Sdept from Student where Sdept in (select Sdept from Student where Sname = '刘晨');
```

8.3.3 带有 ANY 或 ALL 谓词的子查询

使用 ANY 或 ALL 谓词时必须同时使用比较运算符，语义为：

- > any: 大于子查询结果中的某个值
- > all: 大于子查询结果中的所有值
- < any: 小于子查询结果中的某个值
- < all: 小于子查询结果中的所有值
- …
- != all: 不等于子查询结果中的任何值

> 特定：子查询结果一般为一个属性多个值（集合）！

实例：查询非计算机科学系中比计算机科学系任意一个学生年龄小的学生姓名和年龄。

```
select Sname,Sage from Student where Sage < all (select Sage from Student where Sdept = 'SC') and Sdept !='SC';
```

8.3.4 带有 EXISTS 谓词的子查询

带有 exists 谓词的子查询不返回任何数据，只产生逻辑真值 “true” 或逻辑假值“false”。

- 若**内层查询结果非空**，则**外层的 where 字句**返回真值。
- 若**内层查询结果为空**，则**外层的 where 子句**返回假值。
- not exists 与 exists 语义相反。

> 由 exists 引出的子查询，其目标列表达式通常都用 *，因为带 exists 的子查询只关心返回值的真假，给出列名无实际意义。

实例 1：查询所有选修了 1 号课程的学生姓名。

```
select Sname from Student where exists  (select * from SC where SC.Sno=Student.Sno and SC.Cno='1');
```

实例 2：查询没有选修 1 号课程的学生姓名。

```
select * from Student where not exists  (select * from SC where SC.Sno=Student.Sno and Cno='1');
```

8.4 集合查询

8.4 .1 并操作 UNION

实例 1：查询计算机科学系学生及年龄不大于 19 岁的学生。（与 or 同效果）

```
select * from Student where Sdept='CS'
UNION
select * from Student where Sage<=19;
```

> union：将多个查询结果合并起来，系统自动去掉重复元组。
> all union：将多个查询结果合并起来时，保留重复元组。

8.4.2 交集作 intersect

实例：查询计算机科学系的学生并且年龄不大于 19 岁的学生的交集。(与 and 同效果）

```
select  * from Student where Sdept='CS'
intersect
select * from Student where Sage<=19;
```

8.4.3 差操作 except

实例：查询计算机科学系的学生与年龄不大于 19 岁的学生差集。

```
select * from Student where Sdept ='CS'
except
select * from Student where Sage <=19;
```

8.5 派生表查询

子查询不仅可以出现在 where 字句中，还可以出现在 from 字句中，这时子查询生成的临时派生表成为主查询的查询对象。如果子查询中没有聚集函数，派生表可以不指定属性列，子查询 select 字句后面的列名为其缺省属性。

实例 1：查找出每个学生超过他自己选修课程平均成绩的课程号。

```
select Sno,Cno
from SC,(select Sno,avg(Grade) from SC group by Sno) as Avg_sc(avg_sno,avg_grade)
where SC.Sno=Avg_sc.Sno and SC.Grade>=Avg_sc.avg_grade;
```

实例 2：查询所有选修了 2 号课程的学生姓名。

```
select Sname
from Student,(select Sno from SC where Cno='1') as SC1
where Student.Sno = SC1.Sno;
```

9. 数据操纵语言（DML）

9.1 插入

9.1.1 插入数据

语法格式：

```
insert into <表名> [(<列名1>[,<列名2>...)]
values (<常量1>[,<常量2>...]);
```

实例 1：将一个新学生元组 (学号：20220610；姓名：成成；性别：男；所在系：IS; 年龄：18 岁）插入表 student。(一条数据）

```
insert into Student (Sno,Sname,Ssex,Sdept,Sage)
values ('20220610','成成','男','IS','18');
```

实例 2：插入多条数据

```
insert into Student(Sno,Sname,Ssex,Sdept,Sage)
values ('20220610','成成','男','IS','18'),('20220609','田田','女','CS','17');
```

9.1.2 带字句的插入

语法格式：

```
insert into <表名> [<列1> [,<列2>....)]
子查询;
```

实例：对每一个系，求学生的平均年龄，并把结果存入数据库。

```
--建表
create table Dept_age (Sdept char(15), Avg_age int);
--插入数据
insert into Dept_age(Sdept,Avg_age)
select Sdept,avg(Sage) from Student group by Sdept;
```

9.2 修改

9.2.1 修改一条数据

语法格式：

```
update <表名>
set <列名>=<表达式>[,
        <列名>=<表达式>]...
where <条件> ;
--查询的条件筛选后只能有一个元组（即一条记录）,一般为主键或唯一属性值并且不为空。
```

实例：将学生 20220610 的年龄修改为 22 岁

```
update Student
set Sage=22
where Sno='20220610';
```

9.2.2 修改多条数据

语法格式：

```
update <表名>
set <列名>=<表达式>[,
        <列名>=<表达式>]...
where <条件> ;
--查询的条件筛选后为多个元组（即多条记录）
```

实例： 将计算机科学系的年龄都增加一岁。

```
update Student
set Sage=Sage+1
where Sdept='SC';
```

9.2.3 带子查询的修改

语法格式：

```
set <列名>=<表达式>[,
        <列名>=<表达式>]...
where <带子查询的条件>;
```

实例：将计算机科学系的全部学生成绩置零。

```
update <SC>
set Grade=0
where Sno in (select Sno from Student where Sdept='SC');
```

9.3 删除

9.3.1 删除一条数据

语法格式:

```
delete from <表名>
[where <条件>];
--where子句条件筛选后只能有一个元组（即一条记录）,一般为主键或唯一属性值并且不为空。
```

实例：删除学号为 20220610 的学生记录。

```
delete from Student
where Sno='20220610';
```

9.3.2 删除多条数据

语法格式：

```
delete from <表名>
[where <条件>];
--where子句条件筛选后为多个元组（即多条记录）。
```

实例：删除所有计算机科学系学生的记录

```
delete from Student
where Sdept='SC';
```

9.3.3 带子查询的删除

语法格式：

```
delete from <表名>
[where <带子句查询的条件>];
```

实例：删除计算机科学系所有学生的选课记录。

```
delete from SC
where Sno in (select Sno from Student where Sdept='SC');
```

数据库安全性

10. 数据库安全性概述

数据库安全：是指以保**护数据库系统、数据库系统服务器和数据库中的数据、应用、存储**，以及相关网络连接为目的，是防止数据库系统及其数据遭到泄露、篡改或破坏的安全技术。

10.1 不安全因素

- 非授权用户对数据库的恶意存取和破坏
- 数据库中重要或敏感的数据被泄露
- 安全环境的脆弱性

10.2 安全标准

安全级别	定义
A1	验证设计
B3	安全领域
B2	结构化保护
B1	标记安全保护
C2	受控的存取保护
C1	自主安全保护
D	最小保护

11 数据库安全性控制

11.1 用户身份鉴定

1. 静态口令鉴别
2. 动态口令鉴别
3. 生物特征鉴别
4. 智能卡鉴别

11.2 存取控制机制

1. 自主存取控制（Disctetionary Access Control，简称 DAC）
2. 强制存取控制（Mandatory Access Control，简称 MAC）

11.3 权限授权回收

11.3.1 权限授权

语法格式：

```
grant <权限>[,<权限>]...
on <对象类型> <对象名> [,<对象类型> <对象名>]...
to <用户1>[,<用户2>]...
[with grant option];
```

> with grant option : 所授予的权限后，该用户是否有该权限的授予权。

实例 1：把查询 Student 表权限授予用户 U1。

```
grant select
on table Student
to U1;
```

实例 2：把 Student 表和 Course 表的全部权限授予 U2 和 U3。

```
grant all priviliges
on table Student,Course
to U2,U3;
```

实例 3： 把对表 SC 的查询权限授予所有用户。

```
grant select
on table SC
to public;
```

实例 4：把查询 Student 表和修改学生学号的权限。

```
grant update(Sno),select
on table Student
to U4;
```

实例 5：把对表 SC 的 insert 权限授予 U5 用户，并且允许 其将该权限授予其他用户。

```
grant insert
on table SC
to U5
with grant option;
```

11.3.2 权限回收

语法格式：

```
revoke <权限>[,<权限>] ...
on <对象类型> <对象名> [,<对象类型><对象名>] ...
from <用户>[,<用户>] ... [cascade|restrict];
```

数据库完整性

12. 实体完整性

关系模型的实体完整性定义：create table 中用 primary key 定义那些列为主码。
单属性构成的码有两种说明方法：

- 定义为列级约束条件
- 定义为表级约束条件

多属性构成的码有一种说明方法：

- 定义为表级约束条件

实例 1：将 Student 表中的 Sno 属性定义为码。

12.1 在列级定义主码。

```
create table Student(
Sno char(9) primary key,
Sname char(20) not null,
Ssex char(2),
Sage int,
Sdept char(20)
);
```

12.2 在表级定义主码

```
create table Student(
Sno char(9),
Sname char(20) not null,
Ssex char(2),
Sage int,
Sdept char(20),
primary key (Sno)
);
```

实例 2：将 SC 表中的 Sno，Cno 属性定义为主码

```
create table SC(
Sno char(9) not null,
Cno char(4) not null,
Grade int,
primary key(Sno,Cno)
);
```

13. 参照完整性

关系模型的参照完整性定义：在 create table 中用 foreign key 定义那些列为外码。
实例 1：关系 CS(Sno,Cno) 是主码，Sno，Cno 分别参照 Student 表的主码和 Course 表的主码。

```
create table SC(
Sno char(9) not null,
Cno char(4) not null,
Grade int,
primary key (Sno,Cno),
foreign key (Sno) references Student(Sno),
foreign key (Cno) references Course(Cno)
);
```

用户定义完整性

关系模型的用户定义完整性：针对某一具体应用的数据必须满足的语义要求
关系数据库管理系统提供了定义和检验用户定义完整性的机制，不必应用程序承担。
create table 时定义属性上约束条件：

- 列值非空（not null）
- 列值唯一（unique）
- 检查列值是否满足一个条件表达式（check）

实例 1: 定义 SC 表时，说明 Sno、Cno、Grade 属性不允许取空

```
create table SC(
Sno char(9) not null,
Cno char(4) not null,
Grade int not null,
primary key (Sno,Cno),
);
```

实例 2: 建立部门表 DEPT, 要求部门名称 Dname 列取值唯一，部门编号 Dep

```
create table DEPT(
Deptno int,
Dname char(9) unique not null,
Location char(10),
primary key (Deptno)
);
```

实例 3： Student 表示 Ssex 只允许取 “男” 或“女”

```
create table Student（
Sno char（9）primary key,
Sname char(8) not null,
Ssex char(2) check (Ssex in ('男','女')),
Sage int,
Sdept char(20)
）;
```

> 在 create table 时使用 check 短语定义元组上的约束条件，即元组级限制。

实例 4：当学生的性别是男时候，其名字不能以 Ms. 打头。

```
create table Student (
Sno char(9),
Sname char(8) not null,
Ssex char(2),
Sdept int,
Sdept char(20),
primary key (Sno),
check (Sex ='男' and Sname not like 'Ms.%')
);
```

14. 触发器

触发器（trigger）是用户定义在关系表上的一类由事件驱动的特殊过程。触发器保存在数据库服务器中，任何用户对表的增、删、改操作均由服务器自动激活响应的触发器，触发器可以实施更为复杂的检查和操作，具有更精细和更强大的数据控制能力。
触发器又叫做事件 - 条件 - 动作（event-condition-action）规则，当特定的系统事件发生时，对规则的条件进行检查，如歌条件成立，则执行规则中的动作，否则不执行该动作，规则中的动作可以很复杂，通常是一段 SQL 存储过程。

14.1 定义触发器

语法格式：

```
create trigger<触发器名>
{before | after}<触发事件>
on <表名>
referencing new | old row as <变量>
for each{row | statement}
[when <触发条件>] <触发动作体>;
```

> 定义触发器语法说明：
>
> 1. 表的拥有者才可以在表上创建触发器
> 2. 触发器名：触发器名可以包含模式名，也可以不包括模式名；同一模式下，触发器名必须是唯一的；触发器名和表名必须在同一模式下。
> 3. 表名；触发器只能定义在基本表上，不能定义在视图上；当基本表的数据发送变化时，将激活定义在该表上响应触发器事件的触发器
> 4. 触发器：触发事件可以使 insert、delete 或 update 也可以是这几个事件的组合；可以是 update of <触发列，…>, 即进一步指明修改哪些列时激活触发器；after/before 是触发器的时机
>   5. 触发器类型：1. 行级触发器（for each row）；2. 语句级触发器（for each statement）
> 5. 触发器条件：触发器被激活时，只有但触发条件未真是触发动作体才执行；否则触发动作不执行； **如果省略 when 触发条件，则触发动作体在触发器激活后立即执行**
> 6. 触发动作体：1. 触发动作体可以是一个匿名 PL/SQL 过程块，也可以是对已经创建存储过程的调用；
>   2. 如果是行级触发器，用户都可以在过程体中使用 new 和 old 引用事件之后的新值和旧值；
>   3. 如果是语句级触发器，则不能在触发动作体中使用 new 和 old 进行引用
>   4. 如果触发器体执行失败，激活触发器的事件就会终止执行，触发器的目标表和触发器可能影响的其他对象不会发送任何变化。
>   实例: 当对表 SC 的 Grade 属性进行修改时，若分数增加了 10%，则将此次操作记录到下面表中；SC_U(Sno,Cno,Oldgrade,NewGrade), 其中 Oldgrade 是修改前的分数，Newgrade 是修改后的分数

```
create trigger SC_T
after update of Grade
on SC
referencing old row as oldTuple, new row as NewTuple
for each row
when (NewTuple.Grade>=1.1*OldTuple.Grade)
insert into SC_C(Sno,Cno,Oldgrade,NewGrade)
values(
OldTuple.Sno,
OldTuple.Cno,
OldTuple.Grade,
NewTuple.Grade
);
```

114.2 删除触发器

语法格式：

```
drop trigger <触发器名> on <表名>;
```

实例：删除触发器 SC_T

```
drop trigger SC_T on SC;
```

第 6、7 章数据库设计

E-R 模型

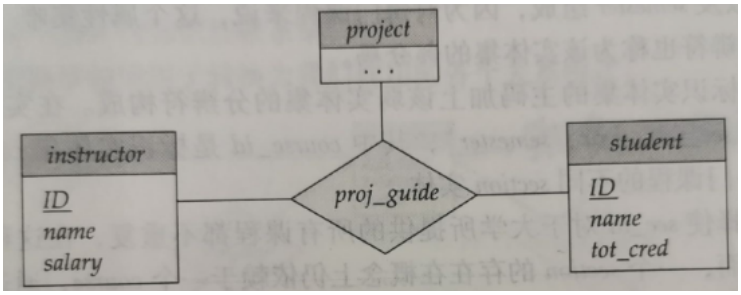
实体 - 联系模型用于表示**概念设计**。概念模式定义了实体、实体的属性、实体间的联系、实体和联系上的约束等。

数据库设计中要避免的两个缺陷是**冗余、不完整**

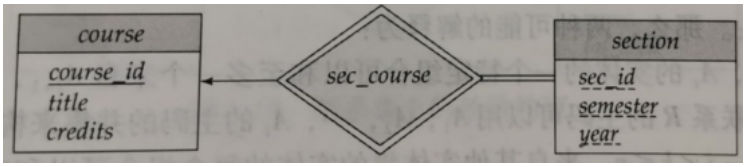
实体 是现实世界中有别于其他对象的一个 “事物” 或“对象”，通过一组属性表示
实体集 是相同类型的实体的集合
联系 是多个实体间的相互关联，联系中也可以有属性
联系集 是相同类型联系的集合。实体集$E_1, E_2, ...E_n$参与联系集R。

简单属性、复合属性、单值属性、多值属性、派生属性

非二元的联系集：



弱实体集：没有足够的属性形成主码的实体集。即实体集中可能会有完全相同的元组，无法区分。
弱实体集必须与另一个**属主实体集**关联，弱实体集依赖于强实体集。



E-R 图设计

范式

关系数据库设计的目的是得到一组合适的关系模式，使其不含冗余，结构良好，便于获取信息。

概念和数学基础

码

- 超码：一个或多个属性的集合，这个集合可以唯一地区分出一个元组（一行）
- 候选码：包含属性最少的超码，可能有多个
- 主码：人为地选中，作为一行的区分标准的候选码。本节中候选码更常用
- 另一个常用的码是外码，与本节无关，不详细介绍

函数依赖

函数依赖是一个形如 $\alpha \to \beta$ 的逻辑推理式，表示属性集 α 决定 (determine) 属性集 β ，或称 β 依赖 α . 同一模式中包含的多条函数依赖称为函数依赖集。

例如，R 上的两个属性 α 和 β ，如果关系实例中的所有元组对 t_1，t_2 都符合 若 $t_1[\alpha] = t_2[\alpha]$ ，则 $t_1[\beta] = t_2[\beta]$。也就是说，只要我们知道 α 的值，就能唯一确定 β 的值，称 α 决定 β 。特别地，如果$\beta \subseteq \alpha$，称为平凡的函数依赖。

由此，我们得出超码的新定义：对于关系 R 和函数依赖集 K，若 $K \to R$ 在 $r(R)$ 上成立，则 K 是 $r(R)$ 的一个超码。

闭包

有了函数依赖集 F，我们就可以由已知的函数依赖得出其他的函数依赖。如 $r(A, B, C)$中有$A \to B, B \to C$，则 $A \to C$（具体推理方法见下文 Armstrong 公理）。类似的推理被称为逻辑蕴涵。不断将这些新的函数依赖加入 F，最终能够得到一个被原 F 逻辑蕴涵的所有函数依赖的集合，称为 F 的闭包，记作 F^+.

Armstrong 公理系统

反复运用以下三条 Armstrong 公理，就能通过F求出F^+，保证结果是正确有效的。

- 自反律：若α为一属性集，$\beta \subseteq \alpha$，则$\alpha \to \beta$
- 增补律：若$\alpha \to \beta$，γ为一属性集，则$\gamma\alpha \to \gamma\beta$
- 传递律：若$\alpha \to \beta$且$\beta \to \gamma$，则$\alpha \to \gamma$

Armstrong 公理是完备的。下面还有一些推论规则，使用起来更方便。

- 合并律：若$\alpha \to \beta$且$\alpha \to \gamma$，则$\alpha \to \beta\gamma$
- 分解律：若$\alpha \to \beta\gamma$，则$\alpha \to \beta$且$\alpha \to \gamma$
- 伪传递律：若$\alpha \to \beta$且$\gamma\beta \to \delta$，则$\alpha\gamma \to \delta$

属性闭包

与函数依赖集的闭包相似，属性闭包是某一属性（或属性集）决定的所有属性的集合。函数依赖集 F 下被属性α决定的所有属性的集合称为F下α的闭包，记为α^+.

利用属性闭包，我们可以判断属性α是否为超码，通过所有属性集的闭包求F^+，求候选码等。

无关属性

一条函数依赖中可能含有不必要的属性，“不必要” 指的是去掉这些属性不会改变函数依赖集的闭包。下文将举例说明，并提供找出无关属性的方法。

关系模式 $student(sid, name, birthday, dept\_name)$，显然有$sid \to name$，$sid \to dept\_name$... 根据 Armstrong 公理能写出很多函数依赖，但我们只用几条为例。以这两条函数依赖作为 F。结合实际生活经验，我们还能得出两条函数依赖：$sid, name \to dept\_name$，$sid \to name, dept\_name$ ，尽管它们看起来有些臃肿。

“臃肿” 是因为这些函数依赖不必要单独列出，利用 F 中的函数依赖，运用 Armstrong 公理就能推导。
F 中的函数依赖告诉我们，属性 sid 决定 dept_name。根据增补律，sid 与任意属性构成的集合都能决定 dept_name，因此这里的 name 是无关属性。同样，根据合并律也能得出 sid→name,dept_name，因此这里的 name 或 dept_name 其中一个是无关属性。

> 在函数依赖$\alpha \to \beta$中：
>
> 如果$A \in \alpha$且F逻辑蕴涵 $(F - \{\alpha \to \beta\} \cup \{(\alpha - A) \to \beta\})$，则属性$A$在$\alpha$中是无关的。
>
> 如果$A \in \beta$且 $(F - \{\alpha \to \beta\} \cup \{\alpha \to (\beta - a)\})$逻辑蕴涵$F$，则属性$A$在$\beta$中是无关的。

通俗来说：

> 在函数依赖$\alpha \to \beta$中：
>
> 要证明α中的属性 A 是无关的，就看能否利用 F 推出 $(\alpha - A) \to \beta$. 如果能，A 就是无关属性。
>
> 要证明β中的属性 A 是无关的，就从 F 中去掉$\alpha \to \beta$，加上$\alpha \to (\beta - a)$，看能否推出 F. 如果能，A 就是无关属性。

正则覆盖

F 的正则覆盖F_c是最小的、与 F 等价的函数依赖集。同一函数依赖集对应的F_c可能不唯一。F_c具有如下性质：

- F_c中的任何函数依赖都不含无关属性
- F_c中函数依赖的左半部分α都是唯一的

F_c的计算方法如下：

> 使用合并律将F_c中所有形如$\alpha_1 \to \beta_1, \alpha_1 \to \beta_2$的依赖替换为$\alpha_1 \to \beta_1, \beta_2$
>
> 验证F_c中的每一条函数依赖，消除其中的无关属性，同时不断更新F_c

多值依赖

在了解多值依赖前，建议读者先阅读下面的 1NF, 2NF, 3NF, BCNF 部分。

函数依赖理论能够将关系模式分解为 BCNF，但它仍有不足。考虑教师模式 $teacher(ID, child, phone)$，它符合 BCNF，但仍然会出现冗余，因为一个教师可以有多个手机号码，也可以有多个孩子，导致同一教师在表中有多个对应元组。为了消除冗余，可以将其分解为 $r_1(ID, child), r_2(ID, phone)$. 这个分解虽然看起来合理，但在函数依赖中并没有理论依据。为了解决这类问题，引入多值依赖。

根据函数依赖的定义，它说明了哪些元组不能存在于关系中；与之相对，多值依赖规定了哪些元组应当存在于关系中。

> $\alpha \to\to \beta$ 称为α多值决定β. 它在 R 上成立的条件是：
>
> 在关系$r(R)$中任意一对满足$t_1[\alpha] = t_2[\alpha]$的元组对$t_1, t_2$，$r$中都存在元组$t_3, t_4$，使得：
>
> $t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$
> $t_3[\beta] = t_1[\beta], t_4[\beta] = t_2[\beta]$
> $t_3[R - \beta] = t_2[R - \beta], t_4[R - \beta] = t_1[R - \beta]$

多值依赖具有以下两条性质：

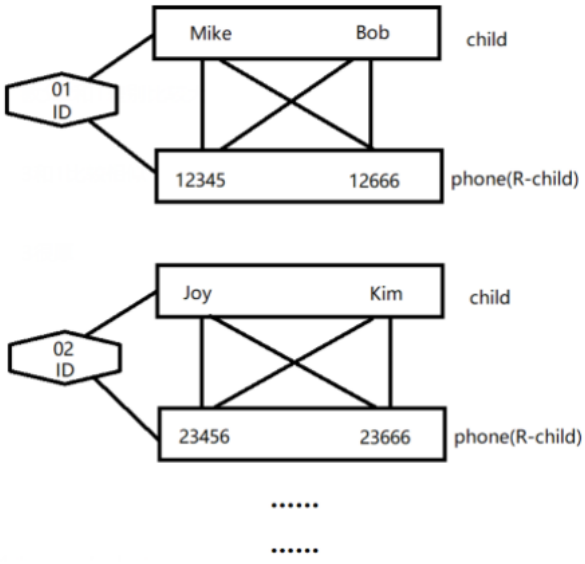
> 对于$\alpha, \beta \subseteq R$:
>
> 若$\alpha \to \beta$，则$\alpha \to\to \beta$； 若$\alpha \to\to \beta$，则$\alpha \to\to R-\beta$

只看定义有些难以理解。依然以模式$teacher(ID, child, phone)$为例，假设该表如下所示：

ID	child	phone
01	Mike	12345
01	Bob	12666
02	Joy	23456
02	Kim	23456
02	Kim	23666

目前，该模式不符合$ID \to\to child$，也不符合$ID \to\to phone$（由性质 2 可知这两条依赖是等价的）。

要满足多值依赖，属性（集）间必须形成完全二分图（也是笛卡尔积）。如图所示：



由图可知，上表还需要添加$(01, Mike, 12666), (01, Bob, 12345), (02, Joy, 23666)$三个元组，以满足多值依赖。由此看来，满足多值依赖就能够保证数据的完整性（但仍会出现冗余）

我们用 D 表示函数依赖和多值依赖的集合，D 的闭包D^+是由 D 逻辑蕴涵的所有函数依赖和多值依赖的集合。

范式

第一范式（1NF）

如果关系 R 的所有域都是原子的，则称其属于第一范式。

例如 $r(dept\_name, location, phone)$ 这个关系模式，如果我们规定 phone 是系公用电话（只能有一个），它就属于第一范式；如果 phone 中保存了系内所有老师的电话，这个属性的域就不是原子的（因为可以继续拆分为单个老师的电话），就不属于第一范式。

第二范式（2NF）

第二范式要求数据库表中的每个元组必须可以被唯一地区分，且主属性只依赖于主码，而不能依赖于主码的一部分。

例如$r(ID, name, course, credit)$不属于第二范式，因为$ID$，$course \to name, credit$，主码为 $(ID, course)$，但又存在$ID \to name, course \to credit.$

第二范式只有历史意义，一般题目中不作考虑。

第三范式（3NF）

在 2NF 的基础上，消除了非主属性对码的传递函数依赖。准确条件是：

> 对于F^+中所有形如$\alpha \to \beta$的函数依赖，以下至少一项成立：
>
> $\alpha \to \beta$是平凡的函数依赖；α是 R 的一个超码；$\beta - \alpha$中的每个属性 A 都包含于 R 的一个候选码中。

注意$\beta - \alpha$中的每个属性 A 可以分别属于不同的候选码，只要它是候选码的一部分即可。

在工程上，一般认为满足第三范式的数据库设计就已经足够好了。

BC范式（BCNF）

在 3NF 的基础上，消除了主属性对码的部分函数依赖和传递依赖。准确条件是：

> 对于F^+中所有形如$\alpha \to \beta$的函数依赖，以下至少一项成立：
>
> $\alpha \to \beta$是平凡的函数依赖；α是 R 的一个超码。

用函数依赖进行数据库设计的目标是：BCNF、无损、保持依赖。有时候无法达到所有目标，必须在 BCNF 和 3NF 中权衡。

第四范式（4NF）

在 BCNF 的基础上，消除了上述范式没能解决的冗余现象。准确条件是：

> 对于D^+中所有形如$\alpha \to\to \beta$的多值依赖，以下至少一项成立：
>
> $\alpha \to\to \beta$是一个平凡的多值依赖；α是 R 的一个超码。

分解

分解是为了使我们设计的模式符合要求的范式，分解方式是本节的重点。

- 有损分解
  对于分解后的两个模式进行自然连接后，无法完整地得到原有的信息。
  例如表 $student(ID, name, dname)$，如果分解为$stu1(ID, name)$ 和 $stu2(name, dname)$，一旦出现重名而不同专业的学生，如 $(01, Ming, CS)$ 和 $(02, Ming, SE)$，将其分解为 stu1 和 stu2 再自然连接后会出现对应的四行：
  $(01, Ming, CS)$、$(01, Ming, SE)$、$(02, Ming, CS)$、$(02, Ming, SE)$，此时就无法区分 01 和 02 所属的专业了。
  有损分解是有害的，应当避免这样的分解。
- 无损分解
  对于分解后的两个模式进行自然连接后，能够完整地得到原有的信息。
  无损分解是我们希望进行的分解。在下文将介绍几种分解算法，它们都能求出无损分解。

若R_1, R_2是R的无损分解，则 $r(R)$上的函数依赖集闭包F^+中至少存在以下依赖中的一个：

$R_1 \cap R_2 \to R_1$； $R_1 \cap R_2 \to R_2$ 即$R_1 \cap R_2$是R_1或R_2的超码。

3NF 分解

> 求出 F 的正则覆盖F_c
> 对于F_c中的每一条函数依赖$\alpha \to \beta$，令$R_i = \alpha\beta$
> 如果所有R_i中都不包含 R 的候选码，就令R_{i+1}= 候选码
> 遍历所有R_i，如果发现R_j包含于另一个模式R_k中，就将R_j删去
>
> 得到新的模式集（$R_1, R_2, ..., R_n$），且每个模式都符合 3NF

BCNF 分解

> 检查 F 中的每条函数依赖是否符合 BCNF 的要求（F_c更好用）
> 对不符合要求的函数依赖$\alpha \to \beta$，借助这条依赖将 R 分解为以下两个：
> （$R_i - \beta$），（α, β）

4NF 分解（与 BCNF 类似）

4NF 分解

> 检查 F 中的每条函数依赖是否符合 4NF 的要求（F_c更好用）
> 对不符合要求的函数依赖$\alpha \to\to \beta$，借助这条依赖将 R 分解为以下两个：
> （$R_i - \beta$），（α, β）

第 12~14 章 物理、数据存储以及索引

存储介质

- 高速缓冲存储器 Cache：最昂贵 容量小 易失
- 主存储器：较小 较贵 易失
- 快闪存储器：容量大 价格低 非易失
- 磁盘存储器：长期联机数据存储

以上存储器速度递减，容量递增。下文主要讨论磁盘。

磁盘

扇区是从磁盘读出和写入信息的最小单位，**块**包含固定数目的连续扇区，数据在磁盘和主存之间以块为单位传输。

磁盘性能的**指标**主要是容量、访问时间、数据传输率、可靠性

访问时间是从发出读写请求到数据开始传输之间的时间。**访问时间 = 寻道时间 + 旋转等待时间**（找磁道→找扇区→读取）

寻道时间是磁盘臂重定位的时间，依赖于目的磁道距离磁盘臂的初始位置有多远。**平均寻道时间**是寻道时间的平均值，大约是最长寻道时间的 1/2，依赖于磁盘模式。

旋转等待时间是读写头到达所需磁道后，等待访问的扇区出现在读写头下需要的时间。**平均旋转等待时间**是平均值，即磁盘旋转一周的时间的 1/2.

数据传输率是从磁盘获得数据或者向磁盘存数据的速率。

平均故障时间是平均来说期望系统无故障连续运行的时间量（是动态变化的），硬盘寿命一般大于五年。

磁盘块访问的**优化**：缓冲、预读、调度、文件组织、非易失性写缓冲区、日志磁盘

为了尽量减少磁盘和存储器之间传输的块的数目，可以建立**缓冲区**，在主存储器中保留尽可能多的块，从而最大化要访问的块已经在主存储器中的几率。

缓冲区替换策略：尽量减少对磁盘的访问，策略有 LRU, MRU 等

被钉住的块：限制一个块写回磁盘的时间，例如一个块上的更新正在进行时，不允许写回磁盘

块的强制写出：为减少崩溃时丢失的内容，有时会强制将缓冲区内的块强制写回磁盘以更新内容

数据存储

文件中记录的组织

- 堆文件组织：一条记录放在文件的任何地方，没有顺序，通常一个关系对应一个单独的文件
- 顺序文件组织：记录根据搜索码的值顺序存储，搜索码是任意属性或属性的集合（不必是超码）
- 散列文件组织：在每条记录的某些属性上计算散列函数，从而确定记录应放到文件的哪个块中
- 多表聚簇文件组织：在每一块中存储两个或更多关系的相关记录，能高效处理连接运算。
  何时使用多表聚簇取决于数据库设计者认为的最频繁的查询类型，合理使用能明显提高性能。

数据字典

存储元数据：关系的名字、属性的名字、属性的域和长度、视图的名字和定义、完整性约束，等等
也会记录关系的组织方式、存储位置、索引的名字、被索引的关系、定义索引的属性、索引的类型，等等

索引

基本概念

数据库系统中的索引和图书馆中的索引类似，可以节省查询的时间

两种基本的索引类型

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

我们主要介绍顺序索引，顺序索引页数数据库中最常用的索引，散列索引提一下，带过就行

顺序索引

几个概念

1. 主索引（聚集索引）：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（聚集索引）
2. 辅助索引（非聚集索引）：与顺序不同

注意：主索引的搜索码不一定是主码

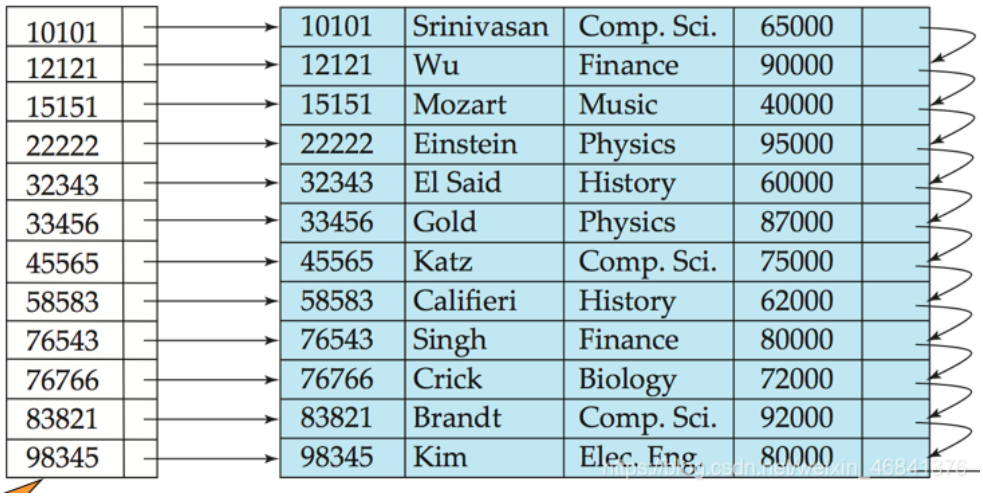
这种在搜索码上由聚集索引的文件称为**索引顺序文件**。

稠密索引和稀疏索引

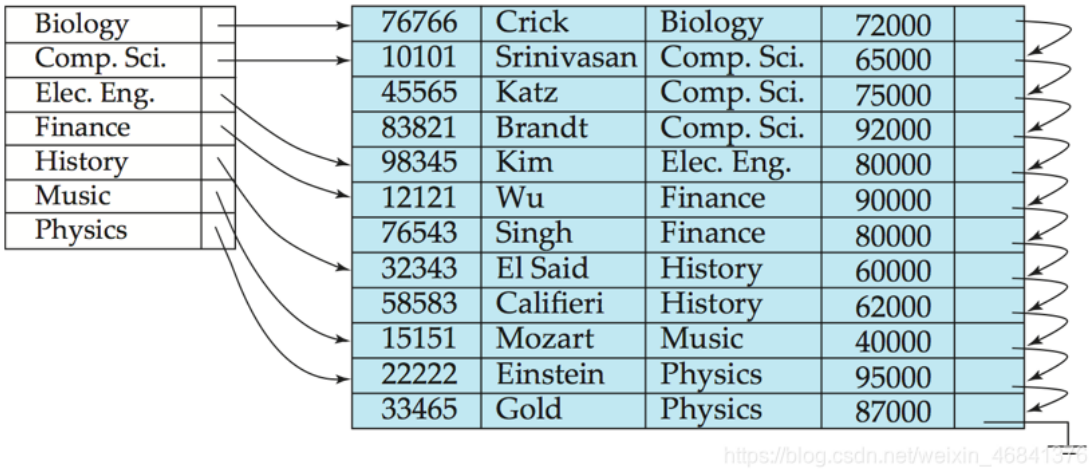
稠密索引

在稠密索引中，文件中的每个搜索码值都有一个索引项。在稠密聚集索引中，索引项包括搜索码值以及指向具有该搜索码值的**第一条数据**的指针。

instructor 的 id 属性的主索引



以 dept_name 为搜索码的 instructor 文件的稠密聚集索引

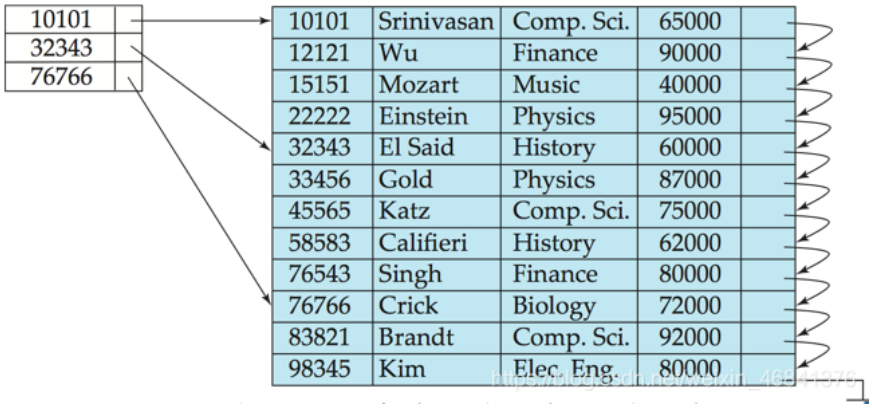


稀疏索引

在稀疏索引中，只为搜索码的某些值建立索引项。只有当关系按搜索码排列顺序存储时才可以使用稀疏索引

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



两种索引的比较

稀疏索引占用空间小，但查找时间比稠密索引慢一些

多级索引

理解即可

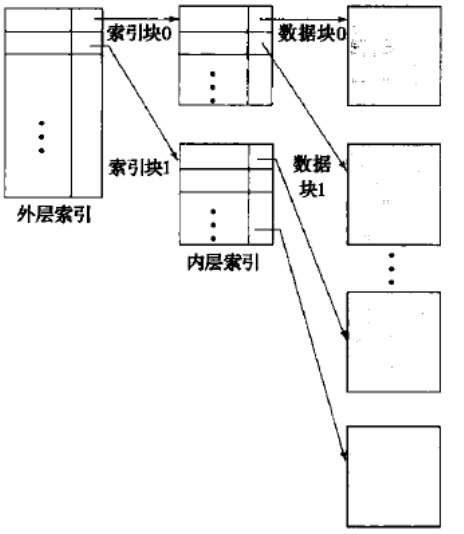
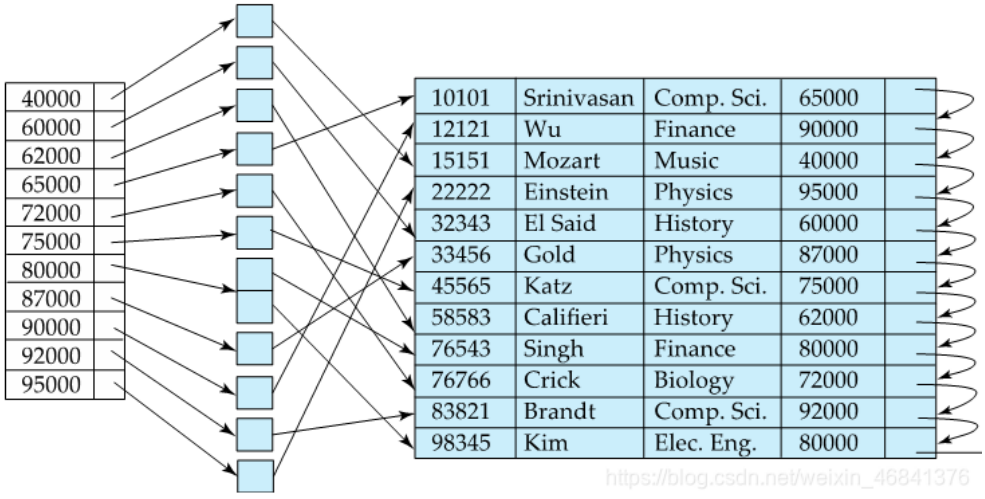


图 11-5 二级稀疏索引

辅助索引

辅助索引必须是稠密索引，每个搜索码都有一个索引项，对文件中的每条记录都有一个指针



这张图中，80000 工资的有两人，因此辅助索引中有两个桶

多码上的索引

之前的搜索码都只含有单个属性，一般来说一个搜索码可用有多个属性

一个包含多个属性的搜索码被称为**复合搜索码**

这个索引的结构和其它索引一样，唯一不同的地方是搜索码不是单个属性，而是一个属性列表

搜索码按照**字典序**排序

B^+ 树索引文件

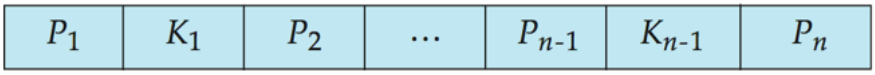
索引顺序文件组织最大的缺点在于，随着文件的增大，索引查找性能和数据顺序扫描性能都会下降。

B + 树索引结构是在数据插入和删除的情况下仍能保持其执行效率的几种使用最广泛的索引结构之一。

B + 树索引采用平衡树结构，树根到树叶的每条路径的长度相同

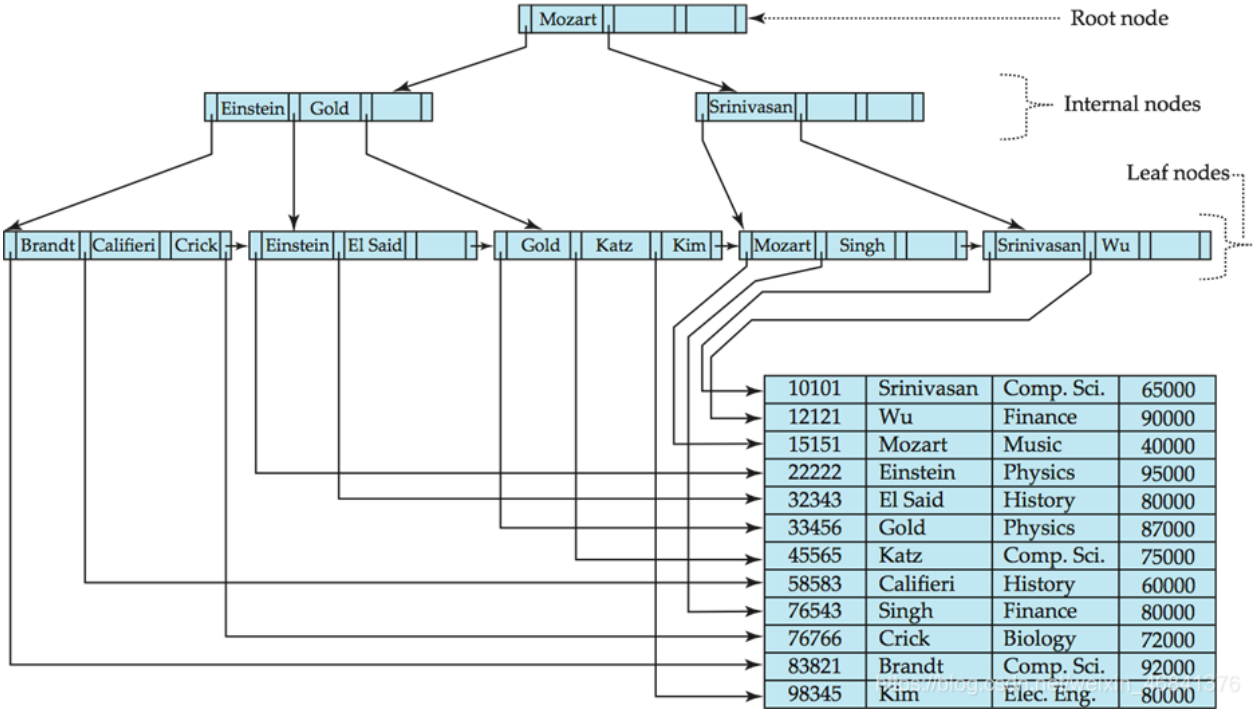
B^+ 树的结构

关于 B^+ 树的具体结构，上学期数据结构已经学过了，这里不再赘述

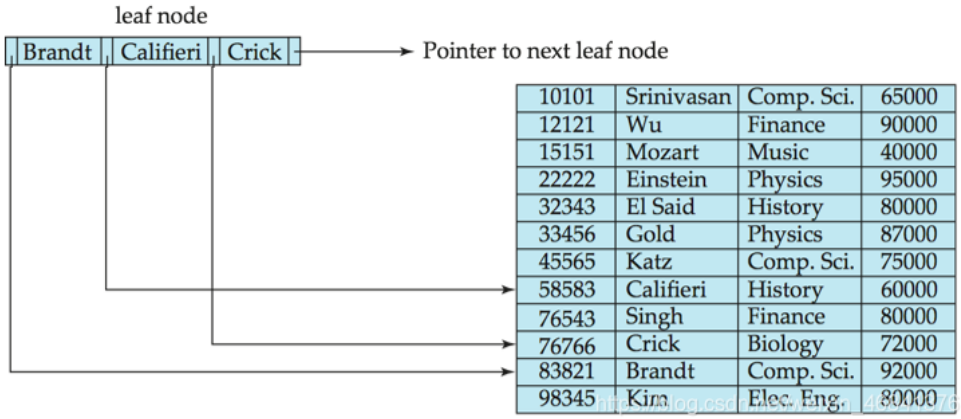


- K_i 是搜索码值
- P_i 是指向孩子结点的指针(对于非叶结点)或者指向记录或记录桶的指针(对于叶子结点)

一个 B^+ 树索引的示例

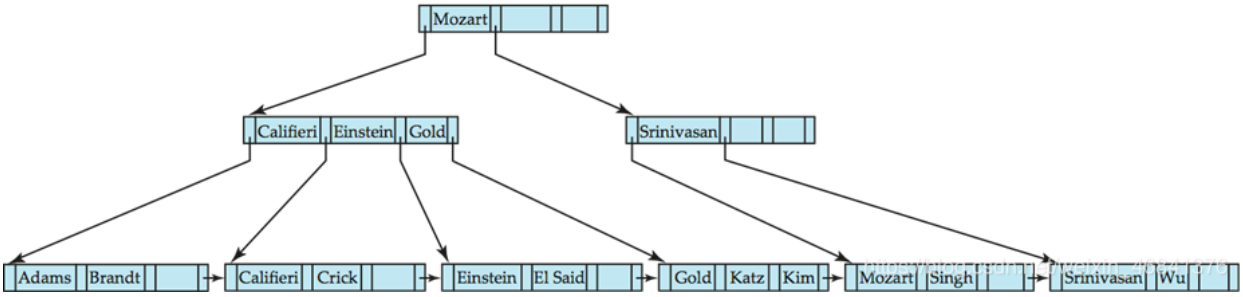


对于叶节点



B^+ 树的查询

查询的流程在上学期数据结构课已经学过了，这里不再赘述

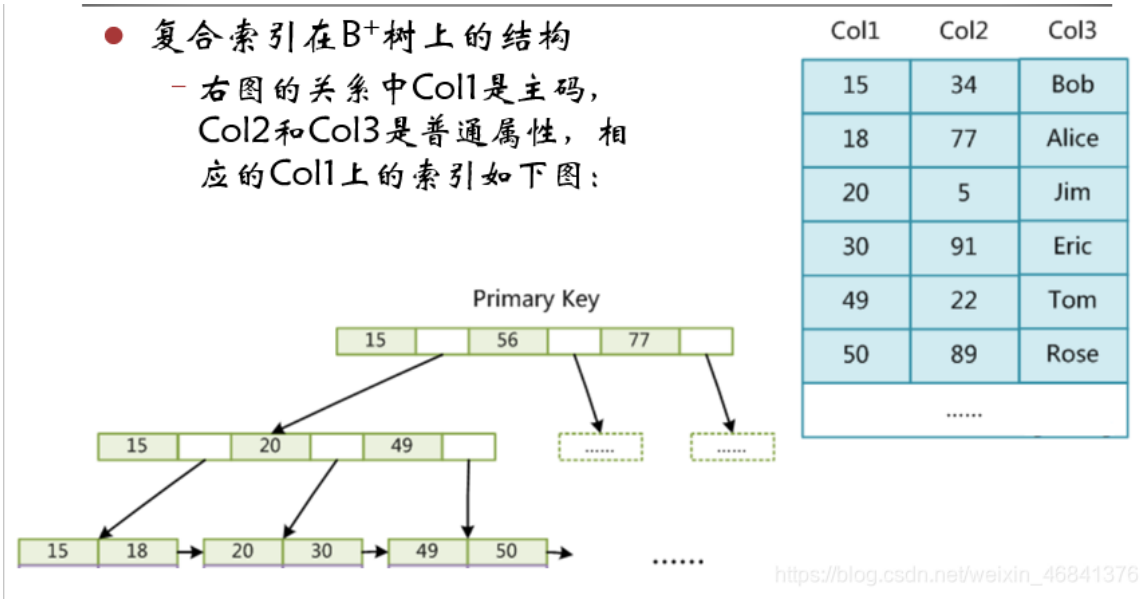


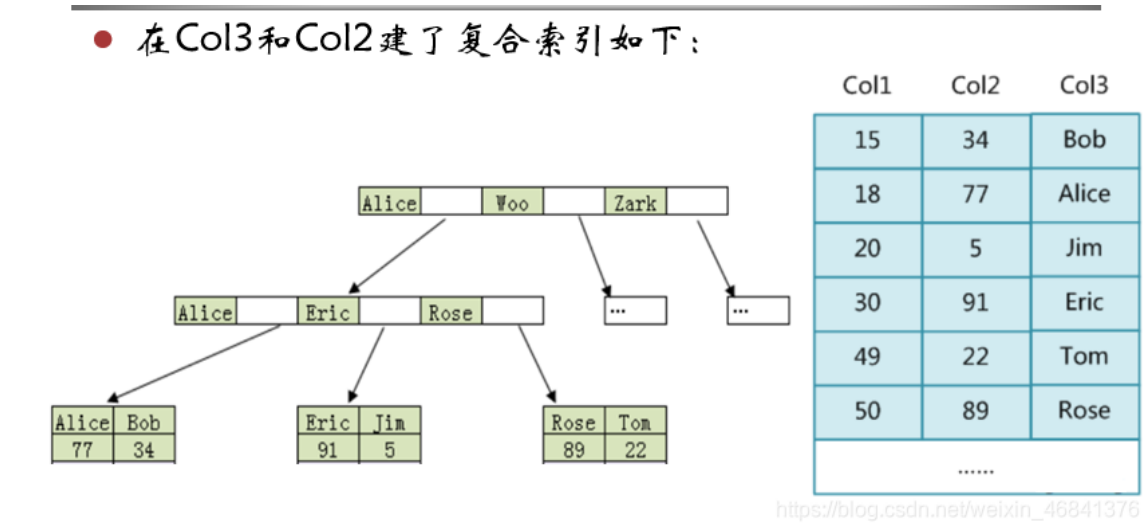
关于 B^+ 树的更新，插入，删除

上学期数据结构课学过，这一块不是重点，不再赘述

B^+树上实现符合索引

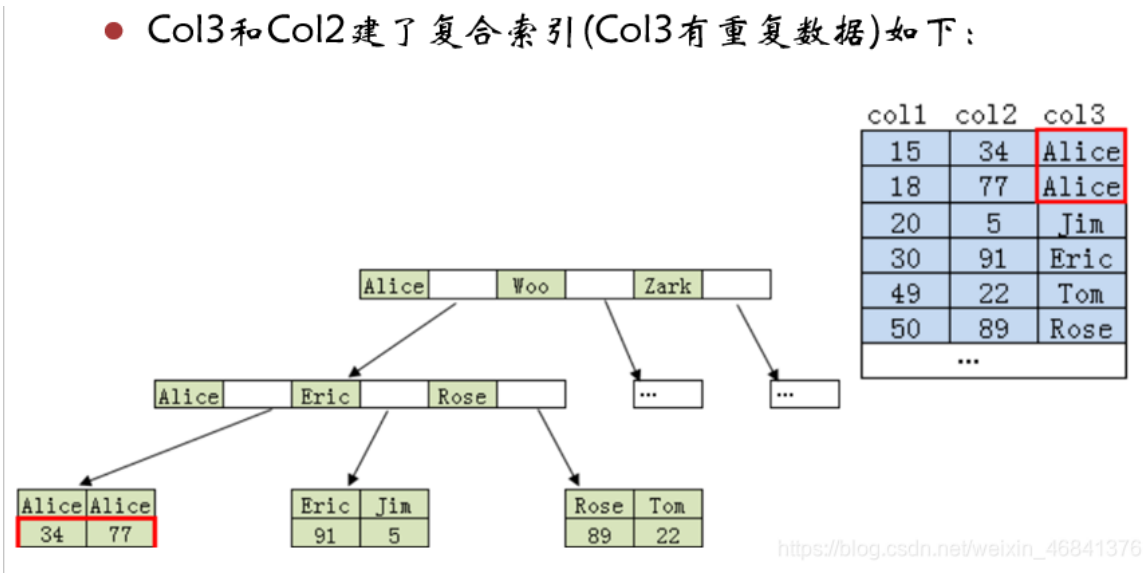
如图





索引中有重复数据的情况

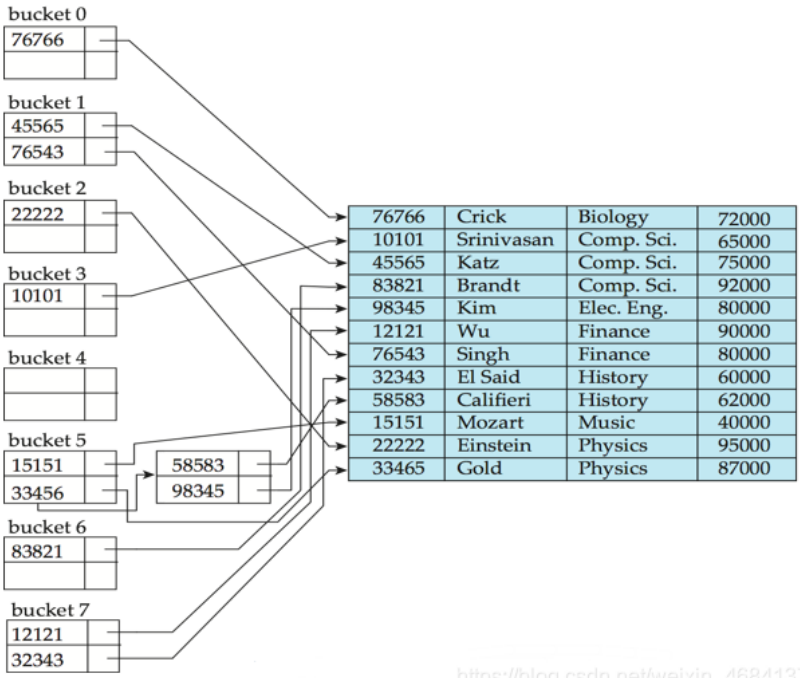
按照字典序进行排序



静态散列

关于散列的文件组织方式，我们只需要了解即可

散列索引的例子



SQL 中的索引定义

创建索引

```
create index <index-name> on <relation-name>(<attribute-list>)
```

删除索引

```
drop index <index-name>
```

第 15 章查询处理

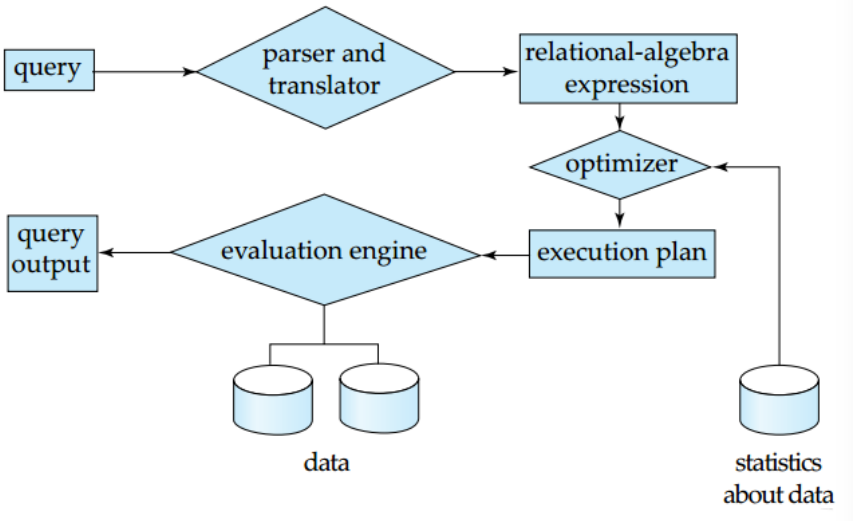
查询处理指从数据库中提取数据时涉及的一系列活动。

这些活动包括：将用高层数据库语言表示的查询语句翻译为能在文件系统的物理层上使用的表达式，为优化查询而进行各种转换，以及查询的实际执行。

概述

查询处理步骤如下图：
基本步骤包括：

1. 语法分析与翻译。
2. 优化。
3. 执行。



要全面说明如何执行一个查询，不仅要提供关系代数表达式，还要对表达式加上注释来说明如何执行每个操作。

注释可以声明某个具体操作所采用的算法，或将要使用的一个或多个特定的索引。

加了 “如何执行” 注释的关系代数运算称为**计算原语**。
用于执行一个查询的原语操作序列称为**查询执行计划**或**查询计算计划**。

下图表示对所举查询例子的一个执行计划。

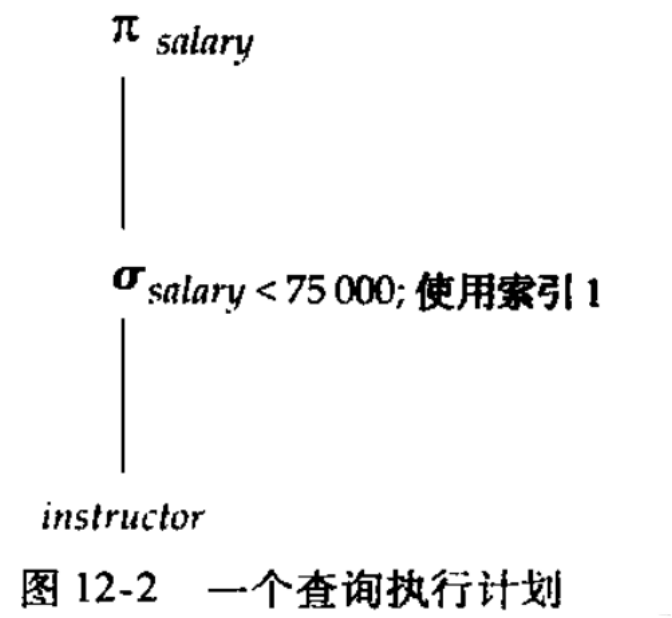


图 12-2　一个查询执行计划

查询执行引擎接受一个查询执行计划，执行该计划并把结果返回给查询。

构造具有最小查询执行代价的查询执行计划应当是系统的责任。这项工作叫作**查询优化**。

某些操作可以组合成**流水线**，其中的每个操作都同时在输入元组上开始执行，即使某个操作的输入元组由另一个操作产生。

查询代价的度量

在磁盘上存取数据的代价常是最主要的代价。
响应时间依赖于主存中缓冲区的大小。
多张磁盘系统响应时间依赖于访问在磁盘的分布。

优化器通常努力去尽可能降低查询计划总的**资源消耗**，而不是尽可能缩低响应时间。

选择运算

在查询处理中，**文件扫描**是存取数据最低级的操作。
文件扫描是用于定位，检索满足选择条件的记录的搜索算法。

使用文件扫描和索引的选择

考虑所有元组都保存在单个文件中关系上的一个选择运算。

执行一个选择最简单的方式如下。

- **A1（线性搜索）**
  在线性搜索中，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。
  开始时需做一次磁盘搜索来访问文件的第一个块。
  索引结构称为存取路径，因为它们提供了定位和存取数据的一条路径。
  主索引（也称为聚集索引）允许文件记录可以按与其在文件中的物理顺序一致的顺序进行读取。
  不是主索引的索引称为**辅助索引**。
  使用索引的搜索算法称为**索引扫描**。
- **A2（主索引，码属性等值比较）**
  对于具有主索引的码属性的等值比较，可以使用索引检索到满足相应等值条件的唯一一条记录。
- **A3（主索引，非码属性等值比较）**
  当选择条件是基于非码属性 A 的等值比较时，可以利用主索引检索到多条记录。
  与前一种情况唯一不同的是，这种情况下需要取多条记录。
  然而，因为文件是依据搜索码进行排序的，所以这些记录在文件中必然是连续存储的。
- **A4（辅助索引，等值比较）**
  使用等值条件的选择可以使用辅助索引。
  若等值条件是码属性上的，则该策略可检索到唯一一条记录；若索引字段是非码属性，则可能检索到多条记录。

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

图 12-3　选择算法代价估计

使用 h_i表示 B^+ 树的高度。

涉及比较的选择

- **A5（主索引，比较）**
  在选择条件是比较时，可用顺序主索引（如 B^+ 树主索引）。
  对于 $A >= v$，在索引中寻找值 V ，以检索出满足条件 $A = v$ 的首条记录。
  从该元组开始到文件末尾进行一次文件扫描就返回所有满足该条件的元组。
  对于 $A > v$，文件扫描从第一条满足 $A > v$ 的记录开始。
  对形如 $A < v$ 或 $A <= v$ 的比较式，没有必要查找索引。
  对于 $A < v$，简单地从文件头开始扫描，直到遇上（但不包含）首条满足 $A = v$ 的元组为止。
  $A <= v$ 类似，这两种情况下，索引没什么用处。
- **A6（辅助索引，比较）**
  可使有序辅助索引指导涉及 $<$, $<=$, $>$, $>=$ 的比较条件的检索。

对 “<” 及“<=”情形，扫描最底层索引块是从最小值开始直到 v 为止；
对于 “>” 及“>=”情形，扫描从 v 开始直到最大值为止。

辅助索引提供了指向记录的指针，但我们需要使用指针以取得实际的记录。
由于连续的记录可能存在于不同的磁盘块中，因此每取一条记录可能需要一次 I/O 操作。
辅助索引应该仅在选择得到的记录很少时使用。

连接运算

用等值连接这个词来表示形如 r ⋈ r . A = s . B s r ⋈_{r.A = s.B} s r⋈r.A=s.Bs 的连接，其中 A、B 分别为关系 r 与 s 的属性或属性组。

嵌套循环连接

一个计算 r ⋈ θ s r ⋈_{θ} s r⋈θs 的简单算法。
由于该算法主要由两个嵌套的 for 循环构成，因此它称为**嵌套循环连接**。
由于算法中有关 r 的循环包含有关 s 的循环，因此关系 r 称为连接的**外层关系**，而 s 称为连接的**内层关系**。

```
for each 元组t_r in r do begin
                for each 元组t_s in s do begin
                        测试元组对(t_r, t_s)是否满足连接条件θ
                        如果满足，把t_r*t_s加到结果中
                end
        end
```

块嵌套循环连接

```
for each 块B_r of r do begin
        for each 块B_s of s do begin
                for each 元组t_r in B_r do begin
                        for each 元组t_s in B_s do begin
                                测试元组对(t_r,t_s)是否满足连接条件
                                如满足，加入到结果
                        end
                end
        end
end
```

第 17~19 章事务管理、并发控制以及故障恢复

事务是访问并可能更新各种数据项的一个程序执行单元

事务的特性

原子性 A、一致性 C、隔离性 I、持久性 D

- 原子性
  由于事务执行出现故障，导致系统的状态不再能反映数据库所描述世界的真实状态，而是**不一致状态**。
  数据库在磁盘上记录数据项的旧值，如果事务没能完成执行，数据库就从**日志**中恢复旧值，消除不一致。
- 隔离性
  事务的隔离性确保事务并发执行后的系统状态与这些事务以某种次序一个接一个地执行后的状态等价。
- 一致性
  对于 “从账户 A 转账到账户 B” 这一操作而言，一致性要求 A, B 的账户之和不变。如果原子性或隔离性遭到破坏，一致性也会被破坏。
- 持久性
  一旦事务成功地完成执行，系统必须保证任何故障都不会引起这次操作的数据丢失。

事务的执行状态：中止（没能成功执行）、已回滚（执行一个**补偿事务**，撤销变更）、已提交（成功完成执行）
当事务执行完最后一条语句，会进入**部分提交**状态。此时实际输出可能仍驻留在主存中，硬件故障可能会导致中止。

事务隔离性与串行化

允许并发的理由：提高吞吐量和资源利用率、减少等待时间和平均响应时间。

并发控制机制：数据库控制事务之间的交互，使其并发执行，又不破坏数据库的一致性。

可串行化：并发执行的调度在某种意义上等价于一个串行调度，效果与没有并发时的效果一样。

- 冲突可串行化
  对于两条连续指令 I, J，考虑以下 4 种情形：
  i. I=read(Q), J=read(Q)，因为都是读，I 与 J 的次序无所谓
  ii. I=read(Q), J=write(Q)，因为 J 修改了 Q 的值，所以 I 与 J 的顺序是重要的
  iii. I=write(Q), J=read(Q)，与 2 类似，I 与 J 的顺序是重要的
  iv. I=write(Q), J=write(Q)，I 与 J 的顺序对 I 和 J 无所谓，因为它们都写入新的值，但会对下一条 read 指令有影响
  综上所述，当 I 与 J 是不同事务对相同数据项（Q）的操作，且其中至少一个是 write 指令时，I 与 J 是**冲突**的
  如果调度 S 可以经一系列非冲突指令转换成 S'，称 S 与 S'是**冲突等价**的。若调度 S 与一个串行调度冲突等价，则称调度 S 是**冲突可串行化**的。
- 视图可串行化

视图等价

考虑关于某个事务集的两个调度 S，S'，若调度 S，S'满足以下条件，则称它们是视图等价的：

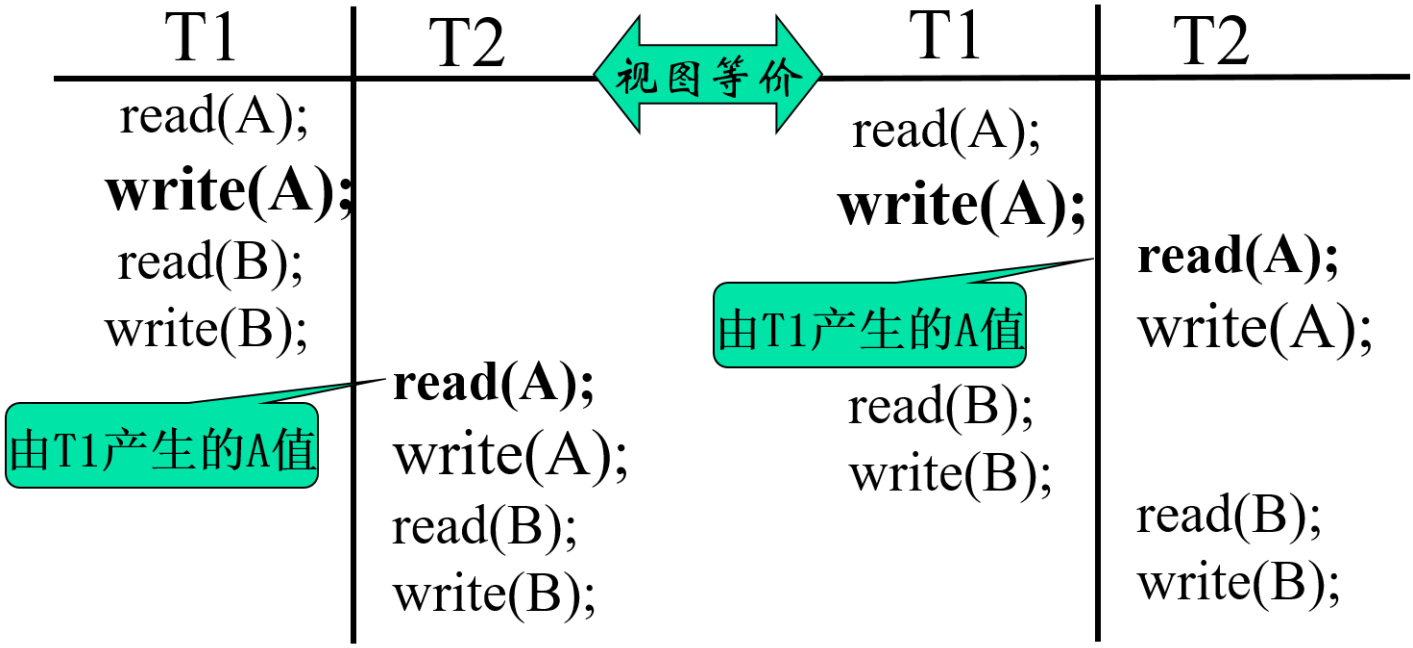
①对于每个数据项 Q，若事务 T_i 在调度 S 中读取了 Q 的初始值，那么 T_i 在调度 S'中也必须读取 Q 的初始值

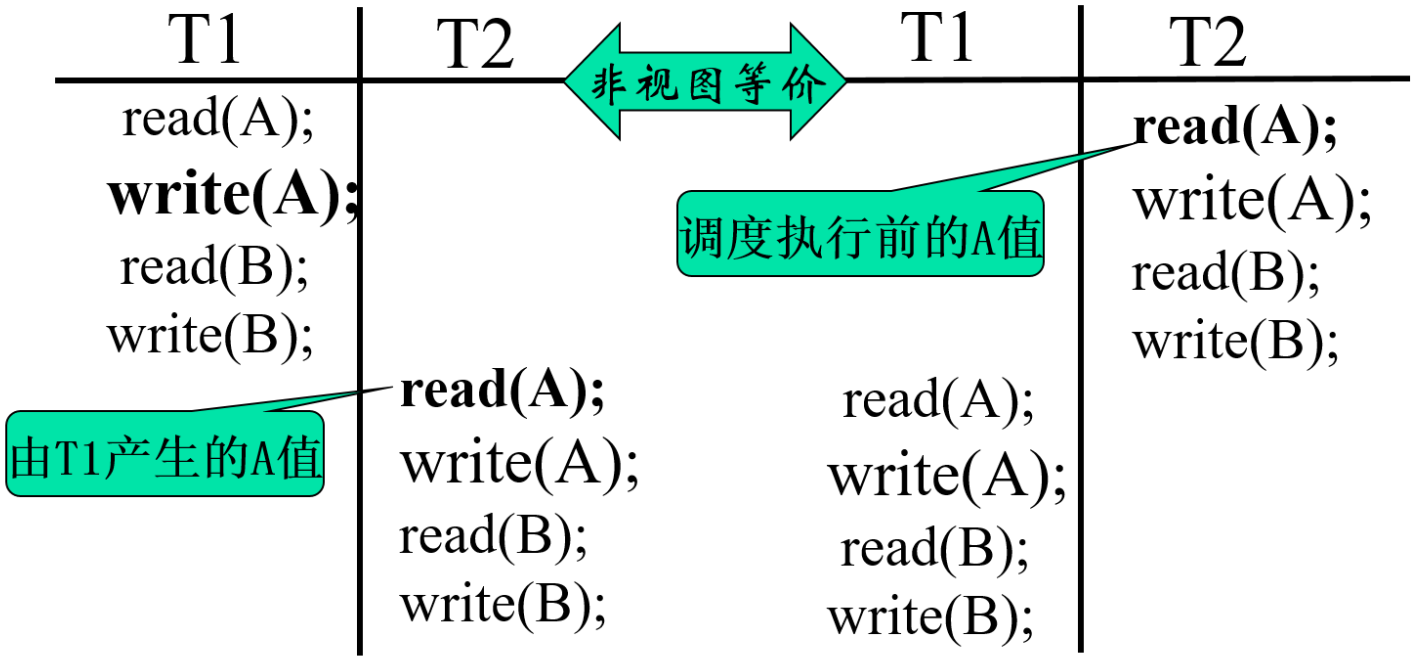
②对于每个数据项 Q，若事务 T_i 在调度 S 中执行了 read(Q)，并且读取的值是由 T_j 产生的，那么 T_i 在调度 S'中读取的 Q 值也必须是由 T_j 产生的

③对于每个数据项 Q，若在调度 S 中有事务执行了最后的 write(Q)，则在调度 S'中该事务也必须执行最后的 write(Q)

注：条件①、②保证两个调度中的每个事务都读取相同的值，从而进行相同的计算

条件①、②、③保证两个调度得到最终相同的系统状态





视图可串行化

如果某个调度视图等价于一个串行调度，则称该调度是视图可串行化的

冲突可串行化调度一定是视图可串行化的

存在视图可串行化但非冲突可串行化的调度![20190314201751622 (1)](/Users/rong/SynologyDrive/Dropbox/Sysu/AnyShare/ShareCache/潘嵘/DB@2022Fall/pics/20190314201751622 (1).png)

可恢复性

- 可恢复调度
  对于每对事务T_i和T_j，如果T_j读取了先前T_i所写的数据项，则称T_j依赖于T_i，T_i应先于T_j提交。
  如果T_j先提交，随后T_i产生错误必须回滚，T_j却无法回滚，导致无法正确恢复。
- 无级联调度
  对于每对事务T_i和T_j，如果T_j读取了先前T_i所写的数据项，则T_i应先于T_j的读操作提交。
  如果T_j和其他多个事件读操作后，T_i产生错误，会导致T_j和一系列事件回滚，产生级联回滚
- 事务隔离级别
  非可串行化、可串行化、可重复读、已提交读、未提交读。都不允许脏写。

并发控制

- 基于锁的协议（悲观）
  共享锁：lock-S(Q)，获得该锁的事务能读 Q 但不能写 Q，允许其他共享锁并存
  排他锁：lock-X(Q)，获得该锁的事务能读写 Q，不允许其他共享锁或排他锁并存
  要访问数据 Q，事务必须先根据自己对 Q 进行的操作**申请**锁，在并发控制管理器**授予**所需锁后继续操作。
  如果访问的数据项已经被其他事务加上了排他锁，当前事务只能等待，直到排他锁全部释放。
  如果两个事务都在等待对方释放锁而无法进行，称为**死锁**，此时系统必须回滚两个事务中的一个。
  封锁协议：规定事务何时对数据进行加锁、解锁，从而限制可能的调度数目。
  两阶段封锁协议：能保证可串行性，不能防止死锁。要求每个事务分两阶段提出加锁和解锁申请：
  增长阶段，事务可以获得锁，但不能释放锁。增长阶段的结束点称为事务的封锁点。
  缩减阶段，事务可以释放锁，但不能获得锁。
  严格两阶段封锁协议：除上述要求外，还要求事务持有的排他锁必须在事务提交之后才能释放
  强两阶段封锁协议：要求事务提交之前不得释放任何锁
  多粒度：将数据项组织成树形结构，加锁时通过封锁一个点来隐式地封锁这个节点的全部后代。由此引入了意向锁，当一个节点加上意向锁，其所有祖先节点都会被显式加锁。
- 基于时间戳的协议（悲观）
  预先选定事务的顺序，从而实现事务可串行化。协议同时保证无死锁，但可能导致饥饿。
  对于每个事务T_i，为其分配唯一固定的时间戳$TS(T_i)$. 越早进入系统的事务，时间戳越小。可以利用**系统时钟**或**逻辑计数器**实现时间戳机制。
  对于每个数据项 Q，W-timestamp(Q) 表示成功执行 write(Q) 的所有事务的最大时间戳，下称 *WTS(Q)*；R-timestamp(Q) 表示成功执行 read(Q) 的所有事务的最大时间戳，下称 *RTS(Q)*；这些时间戳随事务的执行不断更新。

时间戳排序协议保证任何有冲突的读写都按时间戳顺序执行，具体如下：

> 情形一：事务T_i发出$read(Q)$.
>
> - 若$TS(T_i) < WTS(Q)$，拒绝操作，事件回滚
> - 若$TS(T_i) \geq WTS(Q)$，执行操作，$RST(Q)$被更新为$RST(Q), TS(T_i)$间的最大值
>
> 情形二：事务T_i发出$write(Q)$.
>
> - 若$TS(T_i) < WTS(Q)$，或$TS(T_i) < RTS(Q)$，拒绝操作，事件回滚
> - 其他情况，执行操作，$WTS(Q)$被更新为$TS(T_i)$.

如果事务被回滚，系统会赋予它新的时间戳并重新启动。

- 基于有效性检查的协议（乐观）
  将每个事务的生命周期依次分为三个阶段：读阶段、有效性检查阶段、写阶段。不同事务的三阶段可以交叉进行。
  有效性测试使用时间戳：$Start(T_i), TS(T_i), Finish(T_i)$，分别对应三个阶段，其具体测试如下：

  > 任何满足$TS(T_k) < TS(T_i)$的事务T_k必须满足以下两者之一：、
  >
  > - $Finish(T_k) < Start(T_i)$
  > - T_k所写数据项集与T_i所读数据项集不相交，且$Start(T_i) < Finish(T_k) < TS(T_i)$.

  有效性检查机制自动预防级联回滚，但可能导致饥饿。

乐观机制得名于事务乐观地执行，假定它们能完成执行且最终有效；相反，**悲观机制**在检测到一个冲突时就会强制回滚，即使该调度还存在冲突可串行化的可能。

故障

故障的原因包括：磁盘故障、电源故障、软件错误、人为破坏等。

数据库系统必须预先采取措施，以保证即使发生故障，也能保证事务的原子性，持久性。

恢复机制用于将数据库在遭遇故障时，将其恢复到故障发生前的一致性状态。

恢复机制还提供**高可用性**，即：它必须将数据库崩溃后不能使用的诗句缩减到最短。

故障分类

- **事务故障**
  有两种错误可能造成事务执行失败：
  逻辑错误。 事务由于某些内部条件而无法继续正常执行，这样的内部条件如非法输入、找不到数据、溢出或超出资源限制。
  系统错误。 系统进入一种不良状态（如死锁），结果事务无法继续正常执行。但该事务可以在以后的某个时间重新执行。
- **系统崩溃**
  硬件错误，或者是数据库软件或操纵系统的漏洞，导致易失性存储器内容的丢失，并使得事务处理停止。而非易失性存储器仍完好无损。
  硬件错误和软件漏洞致使系统终止，而不破坏非易失性存储器内容的假设称为**故障 - 停止假设**。
- **磁盘故障**
  在数据传送操作过程中由于磁头损坏或故障造成磁盘块上的内容丢失。
  其他磁盘上的数据拷贝，或三级介质（如 DVD 或磁带）上的归档备份可用于从这种故障中恢复。

故障发生后仍保证数据库一致性以及事务原子性的算法，称为**恢复算法**。由两部分组成：

- １．在正常事务处理时采取措施，保证有足够信息用于故障恢复。
- 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。

存储器

存储器分为以下三类：

- **易失性存储器**
- **非易失性存储器**
- **稳定存储器**

稳定存储器，或更准确地说是接近稳定的存储器，在恢复算法中起到至关重要的作用。

稳定存储器的实现

在多个非易失性存储介质（通常是磁盘）上以独立的故障模式复制信息，并且以受控的方式更新信息，以保证数据传送过程中发生的故障不破坏所需信息。

最简单并且最快的 RAID 形式是**磁盘镜像**，即在不同的磁盘上为每个磁盘块保存两个拷贝。

更安全的系统远程为文档存储器的每一个块保存一份拷贝，除在本地磁盘系统进行块存储外，还通过计算机网络写到远程去。
由于在往本地存储器输出块的同时也要输出到远程系统，一旦输出操作完成，即使发生火灾或洪水这样的灾难，输出结果也不会丢失。即**远程备份系统**。

在内存和磁盘存储器间进行块传送有以下几种可能结果：

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

要求，如果**数据传送故障**发生，系统能检测到且调用恢复过程将块恢复为一致的状态。

系统需为每个逻辑数据库块维护两个物理块。
若是镜像磁盘，则两个块在同一个地点；若是远程备份，则一个块在本地，另一个在远程节点。

输出操作的执行如下：

1. 将信息写入第一个物理块。
2. 当第一次写成功完成时，将相同信息写入第二个物理块。
3. 只有第二块写成功完成时，输出才算完成。

恢复过程：
对于每一个块，系统需要检查它的两个拷贝。如果他们相同并且没有检测到错误存在，则不需要采取进一步动作。
如果系统检测到一个块中有错误，则可以用两一个块的内容替换这一块的内容。如果两个块都没有检测出错误，但它们的内容不一致，则我们用第二块的值替换第一块的内容，或者用第一块的值替换第二块的内容。

上述方法可保证写操作，要么完全成功，要么完全失败。

数据访问

假设没有数据项跨越多个块。

事务由磁盘向主存输入信息，然后再将信息输出回磁盘。
输入和输出操作以块为单位完成。
位于磁盘上的块称为**物理块**，临时位于主存的块称为**缓冲块**。
内存中用于临时存放块的区域称为**磁盘缓冲区**。

磁盘和主存间的块移动是由下面两个操作引发的：

- **input(B)**
  将物理块 B 送至内存。
- **output(B)**
  将缓冲块 B 送至磁盘，并替换磁盘上相应的物理块。

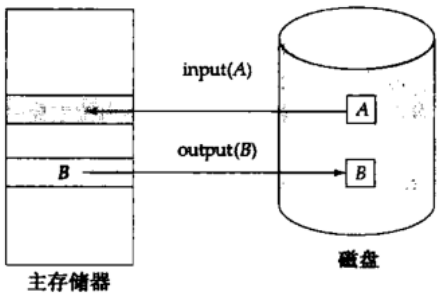


图 16-1　块存储操作

每个事务有一个私有工作区，用于保持事务访问和更新的数据项的拷贝。
事务提交或中止时，由系统删除。

1. **read(X)**
   a. 若 X 所在的块 B_x 不在主存中，执行 $input(B_x)$，将包含 x 的块读入主存。
   b. 将缓冲块中 X 的值赋予 x_i。

2. **write(X)**
   a. 若 X 所在的块 B_x 不在主存中，执行$input(B_x)$，将包含 x 的块读入主存。
   b. 将 x_i 的值赋予缓冲块 B_x 中的 X。

如果数据库系统发指令执行 $output(B_x)$，称对缓冲块 B 执行**强制输出**。

数据库系统执行额外的动作来保证，即使发生了系统崩溃，由提交的事务所做的更新也不会丢失。

恢复与原子性

修改数据库本身前，先向稳定存储器输出信息，描述要做的修改。
输出的信息能帮助我们确保已提交事务做的所有修改都反映到数据库中（或者在故障后的恢复过程中反映到数据库中）。
这种信息还能帮助我们确保中止事务所做的任何修改都不会持久存在于数据库中。

日志记录

日志是日志记录的序列，它记录数据库中的所有更新活动。

日志记录有几种。
更新日志记录描述一次数据库写操作，它具有如下几个字段：

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

日志记录类型：

- $< T_i start >$
  表示事务的开始
- $< T_i commit >$
  表示事务的提交
- $< T_i abort >$
  表示事务的中止

每次事务执行写操作时，必须在数据库修改前建立该次写操作的日志记录并把它加到日志中。再实际执行写数据库。

日志记录让我们有能力进行操作的撤销、重做。
日志需放在稳定存储器中。

数据库修改

事务在对数据库进行修改前创建了一个日志记录。
日志记录使得系统在事务必须中止的情况下能够对事务所做的修改进行撤销；并且在事务已经提交但在修改已存放到磁盘上的数据库中之前系统崩溃的情况下能够对事务所做的修改进行重做。

事务执行数据项修改的步骤：

1. 事务在主存中子句私有的部分执行某些计算。
2. 事务修改主存的磁盘缓冲区中包含该数据项的数据块。
3. 数据库系统执行 output 操作，将数据块写到磁盘中。

如果事务对磁盘缓冲区内的块或磁盘自身进行了更新，则称其修改了数据库。
如果仅在主存中对事务私有部分进行修改，则不算对数据库的修改。
事务提交时，若仍然没修改数据库，称其采用了**延迟修改**技术。
这种情况下，事务对修改的数据项，先拷贝一份到事务私有部分，之后指向和修改其私有部分。
事务执行中，即修改了数据块，称其采用了**立即修改**技术。

恢复算法必须考虑多种因素，包括：

- 有可能事务已经提交，但其修改仍然在磁盘缓冲区，而非磁盘上数据库中。
- 有可能一个事务已经修改了数据库，但执行了某一步需要中止。

由于所有的数据库修改之前必须建立日志记录，因此系统有数据项修改前的旧值和要写给数据项的新值可以用。

- **undo**
  使用一个日志记录，将指明的数据项设为旧值
- **redo**
  使用一个日志记录，将指明的数据项设为新值

并发控制和恢复

一般，数据项 X 被事务 A 修改了，则在 A 提交或中止前，不允许其他事务修改 X。
对更新数据项 X 申请排他锁，事务提交时才释放锁。

事务提交

当一个事务的 commit 日志记录为该事务的最后一个日志记录，输出到文档存储器后，就称**事务提交**了。

此后，即使系统崩溃，事务所做的更新也可重做。

如果在事务的 commit 日志记录输出到稳定存储器前，崩溃，则事务回滚。

包含 commit 日志记录的块的输出是当原子动作，它导致了事务的提交。

事务提交时事务修改的缓冲区块，可以立即写入文档存储器，也可后续写入。

使用日志来重做和撤销事务

- **redo(T)**
  将事务 T 更新过的所有数据项的值都设为新值。
  扫描日志，对扫描范围的每个日志记录逐个处理。
- **undo(T)**
  将事务 T 更新过的所有数据项的值都设为旧值。
  - 撤销过程也产生日志记录（redo-only 日志记录）。
  - 对事务 T 的 undo 操作完成后，写一个日志记录，表示撤销完成。
    每个事务在日志中，最终要么有一个 commit，要么有一个 abort。

发生系统崩溃之后，系统查阅日志以确定为保证原子性需要对哪些事务进行重做，对哪些事务进行撤销。

- 如果日志中含 $< Tstart >$ 记录，但既不含 $< Tcommit >$ 又不含记录，则需对 T 执行撤销。
- 如果日志中含 $< Tstart >$ 记录，以及 $< Tcommit >$ 或 $< Tabort >$ 记录，需对 T 进行重做.
  如果在日志中有 $< Tabort >$，意味着日志中有 undo 操作所产生的 redo-only 日志记录；对其重做意味着对 T 的撤销。

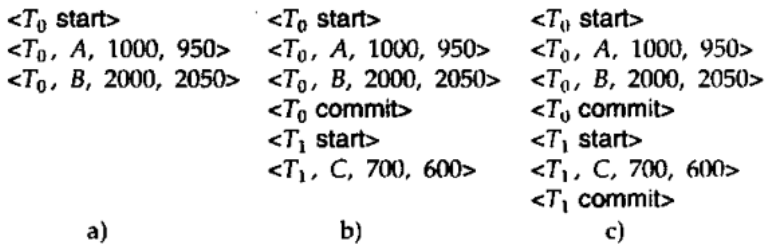


图 16-4　在三个不同时间显示的同一个日志

若崩溃重启后，日志如 a 所示，则对 T_0 执行撤销，撤销后 A 为 1000, B 为 2000。
若崩溃重启后，日志如 b 所示，则对 T_1执行撤销，对 T_0 执行重做．处理后 A 为 950,B 为 2050,C 为 700。
若崩溃重启后，日志如 c 所示，则对 T_0， T_1执行重做，处理后 A 为 950,B 为 2050,C 为 600。

检查点

当系统故障发生时，我们必须检查日志，决定哪些事务需要重做，哪些需要撤销。
原则上，需要搜索整个日志来确定该信息。这样做有两个主要的困难：

1. 搜索过程太耗时。
2. 根据算法，大多数需要重做的事务已把其更新写入数据库中。会使恢复过程变得更长。

为降低这种开销，引入检查点。

一个简单的检查点：它在执行检查点操作的过程中不允许执行任何更新，在执行检查点的过程中将所有更新过的缓冲块都输出到磁盘。

检查点具体执行过程如下：

- 将当前位于主存的所有日志记录输出到稳定存储器。
- 将所有修改的缓冲块输出到磁盘。
- 将一个日志记录 <checkpoint L> 输出到文档存储器，其中 L 是执行检查点时正活跃的事务的列表。

检查点执行过程，不允许事务执行任何更新动作，如往缓冲块写入，写日志记录等。

若某事务 A 的\<A commit>记录在日志中位于记录前，则恢复时，不必再对 A 的相关日志记录进行 redo。

系统崩溃后，系统检查日志，以找到最后一条 <checkpoint L> 记录。

只需对 L 中的事务，及 记录写到日志中后才开始的事务执行 undo 或 redo。把此事务集合记为 T。

对 T 中任一事务设为 A，
１．若日志中无 $< Acommit >$ 和 $< Aabort >$ 记录，则执行 undo(A)。
２．若日志中有 $< Acommit >$ 或 $< Aabort >$ 记录，则执行 redo(A)。

一旦检查点完成了，就不再需要 L 对应事务集合关联的。
$< Xstart >$ 集合中在日志中最先出现的那个 $< Pstart >$ 之前的所有日志记录，对后续恢复不再有意义，可以从日志文件删除掉这部分日志记录。

恢复算法

使用日志记录从事务故障中恢复的完整恢复算法，以及将最近的检查点和日志记录集合起来从系统崩溃中进行恢复的算法。

事务回滚

正常操作时的事务 T 的回滚。

1. 从后往前扫描日志
   对发现的每个形如 $< T_i, X_j, V_1, V_2 >$ 的日志记录：
   ａ．值 V_1 被写到数据项 X_j
   ｂ．往日志中写一个特殊的日志记录 $< T_i, X_j, V_1 >$，其中 V_1 是在本次回滚中数据项 X_j恢复成的值。
   有时这种日志记录称为**补偿日志记录**。
   这样的日志记录只会在崩溃恢复时被执行 redo, 不会被执行 undo
2. 一旦发现 $< Tstart >$ 日志记录，就停止从后往前扫描，并往日志中写一个 $< Tabort >$。

这样对事务 T 所做的和对其撤销所做的每个动作都记录到了日志．

系统崩溃后的恢复

崩溃发生后当数据库系统重启时，恢复动作分两阶段进行：

- １．**在重做阶段**
  系统从最后一个检查点日志记录处开始正向扫描日志来重做所有每个扫描到日志记录所做操作。
  这些被重做的事务包含崩溃前已经回滚的事务，崩溃时尚未提交的事务。
  扫描过程采取的步骤如下：
  ａ．将要回滚的事务列表 undo-list 初始设定为 日志记录中的 L 列表。
  ｂ．一旦遇到形如 $< T_i, X_j, V_1, V_2 >$ 的正常日志记录或形如 $< T_i, X_j, V_2 >$ 的 redo-only 日志记录，就对其执行重做；即将值 V_2写给数据项 X_j。
  ｃ．一旦遇到形如 $< T_istart >$ 的日志记录，就把 T 加到 undo-list。
  ｄ．一旦遇到形如 $< T_iabort >$ 或 $< T_icommit >$ 的日志记录，就把 T_i 从 undo-list 中去掉。
  redo 阶段在扫描并处理完崩溃发生时日志中的最后一个日志记录后结束。
  此时 undo-list 包括在崩溃前尚未完成的所有事务。
- ２．**在撤销阶段**
  系统回滚 undo-list 中的所有事务。
  从日志尾端开始反向扫描日志来执行回滚。
  ａ．一旦发现属于 undo-list 中的所有事务，就执行 undo 操作。就像在一个失败事务的回滚过程中发现了该日志记录一样。
  ｂ．系统发现 undo-list 中事务 T_i的 $< T_istart >$ 日志记录时，
  就从 undo-list 中删除 T_i，且产生一条 $< T_iabort >$ 形式的日志记录，写入日志末尾。
  ｃ．一旦 undo-list 变为空表，即系统已经找到了开始时位于 undo-list 中的所有事务的 $< T_istart >$ 日志记录， 则撤销阶段结束。
  撤销阶段结束后，即可开始正常的事务处理。

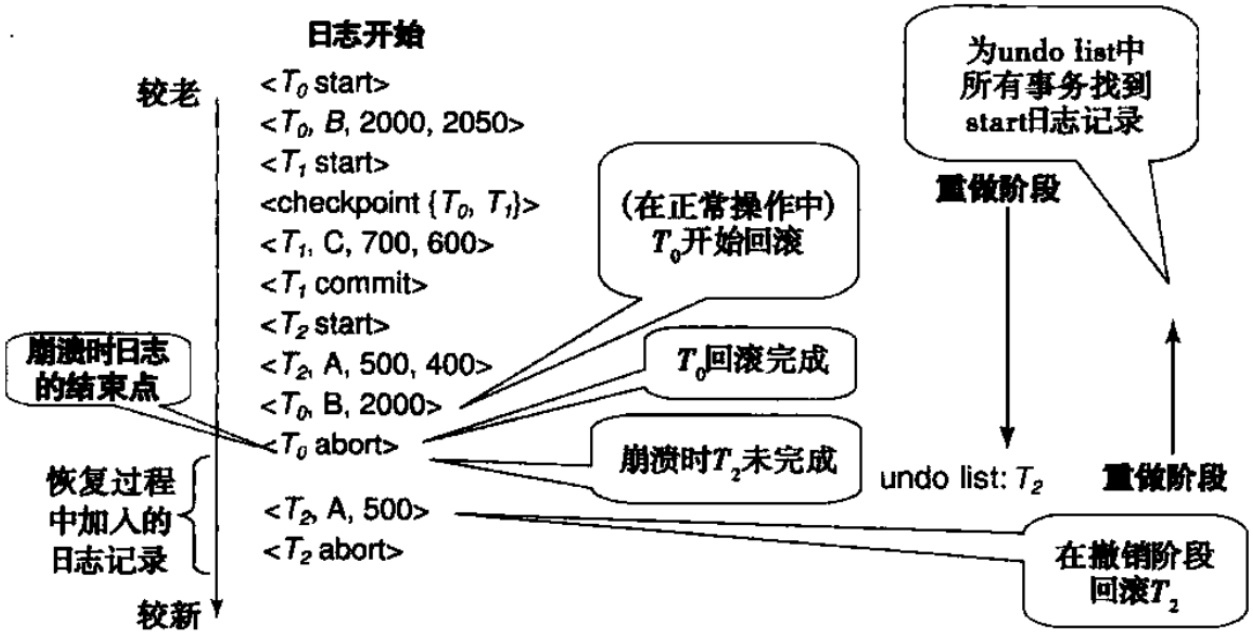


图 16-5　记录在日志中的动作和恢复中的动作的例子