

# 数据库系统

## 课程资源

- CMU数据库公开课 [卡内基梅隆在线课程](#)
- 数据库人大在线课程 [人大数据库课程](#)

## 教材和参考资料

- 《数据库系统概论》(第6版) ---- 主要教材
- Database System Concepts (Abraham Silberschatz, Henry Korth, S. Sudarshan) ---- CMU参考教材

## 成绩组成

- 期末考试
- 平时成绩

## 1. 绪论

### 1.1 数据库系统概论

#### 1.1.1 数据库的4个基本概念

1. **数据(data):** 数据库中存储的基本对象，描述事物的符号记录。
2. 数据库
3. 数据库管理系统
4. 数据库系统

#### 数据(Data)

数据的含义称为**数据的语义**。

#### 数据库(DB)

- 数据库(database, 简称DB) 是长期存储在计算机内有**组织、可共享**的大量信息的集合。

#### 🔗 数据库基本特征

- 数据按一定的模型组织、描述和存储
- 较小的冗余
- 较高的数据独立性
- 可拓展性

#### 数据库管理系统(DMS)

- 位于用户与操作系统之间的一层数据管理软件
- 计算机基础软件

#### 🔗 主要功能

1. **数据定义功能**
  - 提供数据定义语言(DDL)
  - 定义数据库中的数据对象的组成和结构
2. **数据组织、存储和管理功能**
  - 分类组织、存储
  - 确定数据存储的文件结构和存取方式
  - 实现数据的联系
  - 提供多种存取方法提高存取效率
3. **数据的操纵功能**
  - 提供数据操纵语言(DML)
  - 实现数据库的基本操作
4. **数据库的事务管理和运行管理**
  - 实现与网络其他软件的通信
  - DBS之间或与文件系统之间的交换
  - 异构DB之间的互访和互操作

#### 数据库系统(DBS)

- 数据库
- 数据库管理系统
- 应用程序
- 开发人员

#### 🔗 数据库系统管理数据的特点

- 整体数据结构化
- 数据的共享性强，冗余度低且易于扩充
- 数据的独立性强
  - 物理独立性(绝对独立)
    - 程序与数据存储的物理存储相互独立
    - 物理存储发生改变，应用程序无需改变
  - 逻辑独立性(相对独立) ---- DBS采用三层模式结构
    - 用户的应用程序与数据库的逻辑结构相互独立
    - 数据的逻辑结构改变，用户的应用程序无需改变
  - 数据独立性由数据库管理系统的两级映射功能来保证

- 数据由数据库管理系统统一管理和控制
  - 数据的安全性(security)保护
  - 数据的完整性(integrity)检查
  - 数据的并发(concurrency)控制
  - 数据库的恢复(recovery)

## 1.2 数据模型

**数据模型(data model)** 是对现实世界数据特征的抽象，是现实世界的模拟。

### 📌 数据模型满足的要求

- 能比较真实的模拟现实世界
- 容易为人所理解
- 便于在计算机上实现

### 1.2.1 数据建模

把现实世界的具体事物抽象、组织为某一数据库管理系统支持的数据模型。

1. 建立概念模型
  - 现实世界抽象为信息世界
2. 将概念模型转化为数据模型
  - 信息世界转化为机器世界

### 1.2.2 概念模型

概念模型用于现实世界的建模。

#### 📌 基本要求

- 较强的语义表达能力
- 简单清晰、易于理解

### 信息世界的基本概念

- 实体(entity)
  - 客观存在的可区分的事物，可以是区分的人、事、物、抽象的概念和联系
- 属性(attribute)
  - 实体所具有的某一特性称为属性
  - 一个实体可以有若干个属性刻画
- 码(key)
  - 唯一标识实体的属性集称为码
- 实体型(entity type)
  - 用实体名及其属性名集合来抽象和刻画同类实体称为实体型
- 实体集(entity set)
  - 同一类型实体的集合
- 联系(relationship)
  - 现实世界的联系在信息世界中反映为实体内部的联系和实体集之间的联系
  - 实体内部的联系：实体的属性之间的联系
  - 实体之间联系：不同实体集之间的联系
  - 实体之间的联系有一对一、一对多和多对多等多种类型

### 实体联系方法(E-R模型)

- 用E-R图描述现实世界的概念模型。

### 1.2.3 数据模型的三要素----(逻辑模型)

1. 数据结构: 数据结构描述数据库组成对象以及对象之间的联系
2. 数据操纵: 是对数据库中各种对象的实例允许的操作的集合, 包含操作和操作规则
3. 完整性约束: 一组完整性规则

### 1.2.4 层次模型

层次模型是用树形结构表示数据模型。

数据操作：增删改查。

#### 优点

- 结构简单
- 查询效率高
- 良好的完整性约束

#### 缺点

- 很多层次不是层次结构
- 冗余性较高

### 1.2.5 网状模型

网状模型数据库采用网状模型作为数据的组织方式。

### 1.2.6 关系模型

#### 1. 关系模型的数据结构

- 关系(relation): 一个关系对应一个二维表

- 原组(tuple)
- 属性: 表中的一列。
- 域: 表中某一属性的取值范围
- 分量: 元组的一个属性值。
- 关系模型: 对关系的描述。

关系模型要求关系是规范化的

关系的每一个分量必须是一个不可分的数据项

## 2. 关系模型的数据操纵与完整性约束

- 数据的操作是集合操作，操作对象和操作结果都是关系
  - 查询
  - 插入
  - 删除
  - 更新
- 存取路径对用户屏蔽

## 3. 关系模型的优缺点

1. 建立在严格的数学概念的基础上
2. 概念单一
  - 实体与实体之间联系都用关系来表示
  - 对数据的检索和更新结果也是关系
3. 关系模型的存取路径对用户屏蔽

缺点

- 存取路径对用户屏蔽，查询效率往往不如层次模型和网状模型。
- 为提高性能，必须对用户的查询请求进行优化，增加了开发数据库管理系统的难度。

### 1.2.7 其他数据模型

- 面向对象数据模型、对象关系模型
- 半结构化的XML数据模型
- 键值对(KV)数据模型
- 文档数据模型
- 图数据模型
- 时序数据模型
- 时空数据模型
- 流数据模型

## 1.3 DBS的三级模式结构

DBS的三级模式结构是数据库系统内部的体系结构。

数据库系统的外部体系结构分为

- 集中式结构
- 客户机/服务器
- 并行结构
- 分布式结构
- 云结构

### 1.3.1 数据库系统中模式的概念

- 型
  - 对某一类数据的结构和属性描述
- 值
  - 型的一个具体赋值
- 模式(schema)
  - 数据库中全体数据的逻辑结构和特征的描述
  - 是对型的描述，不涉及具体值
  - 反应数据的结构及联系
  - 模式是相对稳定的
- 实例(instance)
  - 模式的具体值
  - 反应数据库某一时刻的状态
  - 一个模式可以有很多实例

### 1.3.1 数据库系统的三级模式结构

模式

- 数据库中全体数据的逻辑结构和数据特征
- 数据库系统模式结构的中间层
  - 与数据存储的物理存储细节无关
  - 与应用程序相对无关
- 定义
  - 数据的逻辑结构
  - 数据之间的联系
  - 数据有关的安全性、完整性
- DBMS提供DDL来管理模式  
模式为全局的数据系统

外模式(external schema)

- 外模式也称为子模式，是数据库用户能够看见和使用的局部数据的逻辑结构和特征的描述。
- 一个数据库可以有多个外模式
- 外模式的同一数据，在外模式中的结构、类型、长度、保密级别都可以不同。
- 同一外模式可以为某一用户的多个应用系统使用

用途

- 保证数据库安全性
- 每个用户只能看见和访问所对应的外模式中的数据

内模式(internal schema)

内模式也称为物理模式或存储模式。

- 是数据物理结构和组织方式的描述
- 是数据在数据库内部的表示方式
  - 记录的存储方式
  - 索引的组织方式(B+树或hash索引)
  - 数据是否压缩存储
  - 数据是否加密
  - 数据存储结构的规定

1.3.3 数据库的两级映射与数据独立性

- 保证数据的逻辑独立性
  - 模式改变时，数据库管理员对外模式/模式映像作相应改变，使外模式保持不变。

1.4 数据库系统的组成

- 硬件
- 软件
- 人员

软件平台

- 支持数据库的管理系统运行的操作系统
- 数据库管理系统
- 开发应用系统的高级语言以及编译系统
- 应用开发工具
- 为特定应用背景开发的数据库应用系统

人员

- 数据库管理员
- 系统分析员和数据库设计人员
- 应用程序猿
- 最终用户

数据库管理员(DBA)

- 设计与定义数据库
- 帮助最终用户使用数据库
- 负责数据库的运维工作

2. 关系模型

2.1 关系模型的数据结构及形式化定义

2.1.1 关系

- 单一的数据结构---关系
  - 现实世界实体以及实体之间的联系均用关系表示
- 逻辑结构 -- 二维表
- 域：一组相同类型的值的集合。
- 笛卡尔积：笛卡尔积是域上的集合运算。
  - 元组:笛卡尔积中的每个元素。
  - 分量
  - 基数：一个域的所有可能取值的个数
  - 关系模型中的笛卡尔积一般没有实际意义，只有某个真子集有意义



关系

$D_1 \times D_2 \times \cdots D_n$ 的一个子集称为在域 $D_1, D_2, \cdots D_n$ 的一个关系，表示为

$$R(D_1, D_2, \ldots, D_n)$$

- 属性：每一列所对应的域。
- 三类关系
  - 基本关系: 基本表
  - 查询关系: 查询执行产生的结果对应的表
  - 视图表: 由基本表或其他视图表导出的虚表，不存储实际属性



基本关系的性质

- 列是同质的

- 不同的列可以出自同一域
  - 其中的每一列成为一个属性
  - 不同的属性给予不同的属性名
- 列的顺序无所谓，列的次序可以任意交换
- 任意两个元组的码不能相同
- 行的顺序无所谓，行的次序可以任意交换
- 分量必须取原子值

## 2.2 关系模式

关系的描述称为**关系模式**。  
关系模式可以表示为

$$R(U, D, DOM, F)$$

- R: 关系名
- U: 组成给关系的属性名集合
- D: U中属性所来自的域
- DOM: 属性向域的映像集合
- F: 属性间数据的依赖关系集合

关系模式通常也可以简记为

$$R(U) \text{ 或 } R(A_1, A_2, \dots, A_n)$$

- R: 关系名
- $A_1, A_2, \dots, A_n$ : 属性名
- 候选码(candidate key): 关系模式中的某个属性或一组属性的值能够唯一地标识一个元组，而他的真子集不能，则称该属性或属性组为候选码
- 全码(all-key): 关系模式中的所有属性是这个关系模型的候选码，称为全码
- 主码: 若一个关系模式有多个候选码，则选定一个为主码。(主码通常用下划线标识)
- 主属性: 所有候选码的属性称为主属性(prime attribute)

### 2.1.3 关系数据库

- 支持关系模型的数据库系统
- 在一个关系数据库中，所有时刻的关系模式的集合称为关系数据库

### 2.1.4 关系模型的存储结构

## 2.2 关系操作

- 查询: 选择、投影、连接、除、并、差、交、笛卡尔积
  - 选择、投影、并、差、笛卡尔积是五种基本操作
- 更新: 插入、删除、修改

#### 🔗 操作的特点

- 集合的操作方式
- 关系操作的所有输入输出均是关系
- 关系代数语言
  - 用对关系的运算来表达查询要求
- 关系演算语言: 用谓词来表达查询要求
  - 元组关系演算语言
  - 域关系演算语言
  - 结构化查询语言(Structured Query Language, SQL): 具有关系代数和关系演算的双重特点

## 2.3 关系的完整性

- 实体的完整性和参照的完整性
- 用户定义的完整性

### 2.3.1 实体完整性

实体的完整性规则

- 若属性A是基本关系R的主属性，则A不能取空值

### 2.3.2 参照完整性

关系间的引用关系，参照引用

#### 🔗 外码

- 设F是基本关系R的一个或一个组属性，但不是关系R的码， $K_s$ 是基本关系S的主码。如果F与 $K_s$ 相对应，则称F为R的外码(foreign key)。
- 基本关系R称为参照关系
- 基本关系S称为被参照关系或目标关系
- R和S不一定是不同的关系
- 目标关系S的主码 $K_s$ 。

### 2.3.3 用户定义的完整性

- 满足某些业务需求的数据要求

## 2.4 关系代数

- 关系代数是一个抽象的查询语言，用关系的运算来表达查询。
- 运算对象为关系
- 结果是关系

- 关系代数运算符：集合运算符和准们的运算符

### 集合运算

#### Union

- 具有相同的目n(两个关系有n个属性)
- 相应的属性来自同一域

#### Difference

- 相同的目
- 相应属性来自同一域

#### Intersection

$$R \cap S = \{t \in R, t \in S\}$$

#### Cartesian production

- R: n目关系, k\_1个元组
- S: m目关系, k\_2个元组
- R×S
  - n+m列元素
  - k\_1×k\_2个元组

### 专门的关系运算符

R表示关系,  $t \in R, t[A_i]$

#### 选择(selection)

选择又称为限制, 是在关系R中满足给定条件的元组, 记为

$$\sigma_F(R) = \{t | t \in R \wedge F(t) = true\}$$

#### 投影

关系R上的投影是从R中选择若干属性组成新的关系

$$\Pi_A(R) = \{t[A] | r \in R\}$$

- 结果会自动取消相同的元组

#### 连接

连接也称为θ连接, 指从两个关系的笛卡尔积中选取其属性间满足一定条件的元组。记作

$$R \bowtie S = \{t_r \hat{t}_s | t_r \in R \wedge t_s \in S \wedge t_r[A] \theta t_s[B]\}$$

- A和B: 分别为R和S上度数相等且可比的属性组
- 等值连接
- 自然连接

## 3. 数据库标准语言SQL

### 数据定义

#### 模式定义与删除

定义模式的语句如下

```
CREATE SCHEMA [<模式名>] AUTHORIZATION <用户名>
```

 表示可选参数, <>表示为必须参数

如果没有指定模式名, 则模式名默认为用户名。

删除模式的语句为

```
DROP SCHEMA <模式名> <CASCADE|RESTRICT>
```

**CASCADE** 和 **RESTRICT** 二者必选其一, 如果选择了 **CASCADE**, 表示在删除该模式的同时把在该模式中定义的所有数据库对象都删除; 而如果选择了 **RESRTICT**, 表示如果在该模式下没有任何数据对象时, 才会执行删除指令, 否则会拒绝执行指令。

### 定义表

SQL 使用 **CREATE TABLE** 语句定义基本表

基本格式如下

```
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

数据类型	含义
CHAR(n), CHARACTER(n)	长度为n的字符串
VARCHAR(n), CHARACTERVARING(n)	最大长度为n的可变长字符串
CLOB	字符串大对象
BLOB	二进制大对象
INT, INTEGER	整数(4字节)
SMALLINT	短整型(2字节)
BIGINT	大整型(8字节)
NUMERIC(p, d)	定点数，由p位数(不包含符号和小数点)组成，小数点之后有d位数
DECIMAL(p, d), DEC(p, d)	同NUMERIC，但是数值精度不受p和d的关系
REAL	基于机器精度的单精度浮点数
DOUBLE PRECISION	基于机器精度的双精度浮点数
FLOAT(n)	可选精度的浮点数，精度至少位n位数字
BOOLEAN	布尔值
DATE	日期，格式为YYYY-MM-DD
TIME	时间，格式为HH: MM: SS
TIMESTAMP	时间戳类型
INTERVAL	时间间隔类型

修改基本表

SQL使用 **ALTER TABLE** 来修改已经定义的表

```
ALTER TABLE<表名>
    [ADD[COLUMN] <新列名><数据类型>[完整性约束]]
    [ADD<表级完整性约束>]
    [DROP[COLUMN] <列名>[CASCADE|RESTRICT]]
    [DROP CONSTRAINT<完整性约束名>[CASCADE|RESTRICT]]
    [RENAME COLUMN <列名> TO <新列名>]
    [ALTER COLUMN <列名> TYPE <数据类型>];
```

删除基本表

```
DROP TABLE <表名> [RESTRICT|CASCADE];
```

索引的建立与删除

当数据库的内容较多时，建立索引可以有效的加快查询效率。  
常用的索引类型有

- 顺序表
- B+树索引
- 哈希索引
- 位图索引

建立索引

SQL中使用语句 **CREATE INDEX** 语句建立索引

```
CREATE[UNIQUE][CLUSTER]INDEX <索引名>
ON <表名>(<列名>[次序][, <列名>[<次序>]]...)
```

索引可以建立在表的一列或者多列上，列之间用逗号隔开。每个列名后面可以指定排列次序，  
可选**ASC**(升序排列)或者**DESC**(降序排列)，默认为**ASC**。

修改索引

对于已经建立的索引，可以使用语句 **ALTER INDEX** 语句对其重命名，其用法如下

```
ALTER INDEX <旧索引名> RENAME TO <新索引名>;
```

删除索引

如果表中数据的查询频率比较小，反而增删改的频率比较大，则每次修改都需要维护已经建立的索引，从而影响效率，就需要删除索引。

```
DROP INDEX <索引名>;
```

数据查询

数据查询的基本语法如下

```
SELECT [ALL|DISTINCT] <目标列表达式> [别名][, <目标列表达式>[别名]]...
FROM <表名或视图名>[别名][, <表名或视图名>[别名]] ... | (<SELECT语句>[AS] <别名>)
[WHERE <条件表达式>]
[GROUP BY <列名1>[HAVING <条件表达式>]]
[ORDER BY <列名2>[ASC|DESC]]
[LIMIT <行数1>[ OFFSET <行数2>]];
```

基本查询

```
SELECT * FROM <表名>
```

**SELECT**是关键字，\* 表示所有列，**FROM**表示 从哪个表来查询  
以下语法可以去除重复的行，默认情况下**SELECT**等价于**SELECT ALL**，会保留所有的行

```
SELECT DISTINCT * FROM <表名>
```

条件查询

```
SELECT * FROM <表名> WHERE <条件表达式>
```

条件表达式可以表示为 **<条件1> AND <条件2>**

- 使用 **<>** 判断不相等
  - Example: score <> 80, name <> 'abc'
- 使用 **LIKE** 判断相似
  - Example: name LIKE 'ab%', name LIKE '%bc%'
  - % 为通配符，匹配任意字符， 例如 **ab%** 匹配'ab', 'abc', 'abcd'

比较	=, >, <, >=, <=, !=, <>, !>, !<; NOT+上述比较符
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(逻辑运算)	AND, OR, NOT

通配符

- %通配符匹配任意个数的字符
- \_匹配单个字符

正则表达式

.	任意一个字符
^	开头
\$	结尾
[abc]	其中任意一个字符
[a-z]	范围内的任意一个字符
A   B	A或者B

**Example** 查找姓王的玩家

```
SELECT * FROM player WHERE name REGEXP '^王.$';
```

SQL

投影查询

```
SELECT id, score, name FROM students;
```

```
SELECT id, score points, name FROM students;
```

将列名 **score** 重命名为 **points**。  
使用投影查询的同时也可以使用条件查询



```
SELECT id, score points, name FROM students WHERE gender = 'M';
```

## 排序

当我们使用 **SELECT** 查询时，查询结果通常默认按照 **id** 也就是主值进行排序，但是也可以我们指定其排序方式。

```
SELECT id, name, gender, score FROM students ORDER BY score;
```

默认是按照由小到大进行排序，如果要从大到排列，可以使用 **DESC**

```
SELECT id, name, gender, score FROM students ORDER BY score DESC;
```

如果 **score** 列具有相同的数据，则可以再指定排序的列名，如

```
SELECT id, name, gender, score FROM students ORDER BY score DESC, gender;
```

排序的默认规则为升序排列，为 **ASC**，可以省略。

如果排序和选择排序混合使用，即如果句子中有 **WHERE**，则 **ORDER BY** 需要放在 **WHERE** 之后。

```
SELECT id, name, gender, score
FROM students
WHERE class_id = 1
ORDER BY score DESC;
```

## 聚合查询

聚合函数

- **COUNT(\*)** -- 统计元组的个数
- **COUNT([DISTINCT|ALL]<列名>)** -- 统计一列中值的个数
- **SUM([DISTINCT|ALL]<列名>)** -- 计算一列值的总和
- **AVG([DISTINCT|ALL]<列名>)** -- 计算一列值的平均值
- **MAX([DISTINCT|ALL]<列名>)** -- 求一列值中的最大值
- **MIN([DISTINCT|ALL]<列名>)** -- 求一列值中的最小值

△ **WHERE** 子句不能直接用聚合函数作为条件表达式。聚合函数只能用于 **SELECT** 子句和 **GROUP BY** 子句中的 **HAVING** 短语。

### GROUP BY子句

**GROUP BY** 子句将查询结果按查询的某一列或多列的值分组，值相等的为一组。

对查询结果分组的主要目的是细化聚合函数的作用对象，分组之后每组都对应着一个聚合函数值。

### LIMIT子句

**LIMIT** 子句用于限制 **SELECT** 语句查询结果的数量，一般形式为

```
LIMIT <行数1> [OFFSET<行数2>];
```

SQL

上述语句含义为取<行数1>，忽略前<行数2>行，作为查询结果数据。**OFFSET** 为可选参数。

**Example** 查询平均成绩排名在第3-7名的学生的学号和平均成绩

```
SELECT Sno, AVG(Grade)
FROM SC
GROUP BY Sno
ORDER BY AVG(Grade) DESC
LIMIT 5 OFFSET 2;
```

SQL

## 连接查询

### 等值连接查询和非等值连接查询

使用运算符"="的连接查询称为等值连接查询，使用其他运算符的连接查询称为非等值连接查询。

**连接字段**：连接谓词中的列名称为连接字段。

### 自然连接查询

把查询结果中重复的属性列去掉的等值连接查询称为**自然连接查询**

**Example** 查询每个学生的学号、姓名、性别、出生日期、主修专业及该学生选修课程的课程号与成绩

SQL

```
SELECT Student.Sno, Sname, Ssex, Sbirthday, Smajor, Cno, Grade
FROM Student, SC
WHERE Student.Sno = SC.Sno;
```

### 复合条件连接查询

**WHERE**的子句是由连接谓词和选择谓词组成的复合条件。

**Example** 查询选修81002号课程且成绩在90分以上的所有学生的学号和姓名

SQL

```
SELECT Student.Sno, Sname
FROM Student, SC
WHERE Student.Sno = SC.Sno AND SC.Cno='81002' AND SC.Grade > 90
```

### 自身连接查询

连接操作将一个表自身与自身连接，称为自身连接查询。

**Example** 查询每一门课的间接先修课

SQL

```
SELECT FIRST.Cno, SECOND.Cpno
FROM Course FIRST, Course SECOND
WHERE FIRST.Cpno = SECOND.Cno AND SECOND.Cpno IS NOT NULL;
```

### 外连接查询

为了保留悬空元组，需要利用外连接查询

SQL

```
SELECT Student.Sno, Sname, Ssex, Sbirthday, Smajor, Cno, Grade
FROM Student LEFT OUTER JOIN SC ON (Student.Sno = SC.Sno);
```

左外连接查询列出FROM左边关系的所有元组，右外连接查询列出FROM右边关系的所有元组。

### 多表连接查询

大于两个表连接查询称为 多表连接查询。

**Example** 查询每个学生的学号、姓名、选修的课程名及成绩

SQL

```
SELECT Student.Sno, Sname, Cname, Grade
FROM Student, SC, Course
WHERE Student.Sno = SC.Sno AND SC.Cno = Course.Cno;
```

### 嵌套查询

SQL中一个SELECT-FROM-WHERE称为一个 查询块，将一个查询块嵌入到另一个查询块的WHERE或者HAVING语句的条件中称为嵌套查询(nested query)。

上层查询称为 外层查询或者父查询，下层查询称为 内层查询或者子查询。

△ 子查询的SELECT语句中不能使用ORDER BY子句，只能对最终的结果进行排序操作。

- 子查询的查询条件不依赖于父查询，这类子查询称为不相关子查询。
- 子查询的查询条件依赖于父查询，这类子查询称为相关子查询，整个语句称为相关嵌套语句。

**Example** 找出每个学生超过他自己选修课程平均成绩的课程号

SQL

```
SELECT Sno, Cno
FROM SC x
WHERE Grade >= (SELECT AVG(Grade)
                  FROM SC y
                  WHERE y.Sno = x.Sno);
```

SQL

```
SELECT Sno, Cno
FROM SC x
WHERE Grade >= (SELECT AVG(Grade)
                  FROM SC y
                  WHERE x.Cno = y.Cno);
```

### 带有ANY(SOME)和ALL谓词的子查询

- ANY 大于查询结果的某个值

- ALL 大于查询结果的所有值
- = ANY 等于子查询的结果的某个值
- = ALL 等于查询结果中的所有值
- != ALL 不等于查询结果的某个值
- != ANY 不等于查询结果的任何值

ANY、ALL谓词与聚集函数、IN谓词的等价转换关系

谓词	=	<>或者!=	<	<=	>	>=
ANY	IN	--	<MAX	<=MAX	>MIN	>=MIN
ALL	--	NOT IN	<MIN	<=MIN	>MAX	>=MAX

带有EXISTS谓词的子查询

EXISTS表示逻辑与, 这个谓词不返回任何数据，只返回布尔值。

**Examble**查询选修了81001号课程的学生姓名

SQL

```

SELECT Sname
FROM Student
WHERE EXISTS
    (SELECT *
     FROM SC
     WHERE Sno = Student.Sno AND Cno='81001'
    );

```

SQL中不存在全称量词(for all)，但是可以利用存在量词取反实现，即

!与等价于forall

**Example** 查询选修了全部课程的学生姓名

SQL

```

SELECT Sname
FROM Student
WHERE NOT EXISTS
    (SELECT *
     FROM Course
     WHERE NOT EXISTS
        (SELECT *
         FROM SC
         WHERE SC.Cno = Course.Cno AND SC.Sno = Student.Sno
        )
    );

```

**Example** 查询至少选修了学生20180002选修的全部课程的学生学号

SQL

```

SELECT Sno
FROM Student
WHERE NOT EXISTS
    (SELECT *
     FROM SC SCX
     WHERE SCX.Sno='20180002' AND NOT EXISTS
        (SELECT *
         FROM SCY SC
         WHERE SCY.Sno=Student.Sno AND SCY.Cno=SCX.Cno
        )
    );

```

集合查询

注：参与集合查询的查询结果的列数必须相同，对应数据类型也必须相同。

集合操作主要包含UNIN，INTERSECTION，EXCEPT操作。

基于派生表的查询

子查询也可以出现在FROM子句中，这时子查询生成 临时派生表。

**Example**

SQL

```

SELECT Sname
FROM Student, (SELECT Sno FROM SC WHERE Cno='81001') AS SC1
WHERE Student.Sno=SC1.sno;

```

AS 可以省略，但是必须有一个别名。

## 数据更新

### 插入数据

插入数据使用INSERT语法，通常分为两种形式，一种为插入一个元组，另一种为插入子查询的结果。

#### 插入一个元组

SQL

```
INSERT INTO <表名> [(<属性列1>, [<属性列2>], ...)]
VALUES (<常量1>[, <常量2>]...);
```

#### 插入子查询的结果

SQL

```
INSERT INTO <表名> [(<属性列1>[, <属性列2>...])]
子查询;
```

**Example** 对每个专业求学生的平均年龄，并把结果存入数据库

1. 创建表

SQL

```
CREATE TABLE Smajor_age
(
  Smajor VARCHAR(20),
  Avg_age SMALLINT
);
```

2. 利用子查询插入数据

SQL

```
INSERT INTO Smajor_age
SELECT Smajor, AVG(extract(year from current_date) - extract(year from Sbirthdate))
FROM Student
Group By Smajor;
```

### 修改数据

修改数据就是我们经常所说的“增删改查”中的‘改’，其一般表达式为

SQL

```
UPDATE <表名>
SET <列名> = <表达式>[, <列名>=<表达式>]...
[WHERE <条件>];
```

其功能为修改满足WHERE条件的元组，WHERE可省略，代表修改所有元组。

#### 修改多个元组的值

**Example** 将2020年第一学期选修81002课程的所有学生成绩减少5分

SQL

```
UPDATE SC
SET Grade=Grade-5
WHERE Cno='81002' AND Semester='20201';
```

#### 带有子查询的修改语句

**Example** 将计算机科学技术专业同学的成绩置0

```
UPDATE SC
SET Grade=0
WHERE Sno IN
(
  SELECT Sno
  FROM Student
  WHERE Smajor='计算机科学与技术'
);
```

### 删除数据

删除语句的一般格式为

```
DELETE FROM <表名>
[WHERE <条件>];
```

**Example** 删除一个元组的值

```
DELETE FROM Student
WHERE Sno='20180007';
```

**Example** 删除多个元组

```
DELETE FROM SC;
```

此语句删除了所有的选课记录.

**Example** 带有子查询的删除: 删除计算机科学与技术所有学生的选课记录

```
DELETE FROM SC
WHERE Sno IN
(
    SELECT Sno
    FROM Student
    WHERE Smajor='计算机科学与技术'
);
```

处理空值NULL

空值就是"不知道""不存在"或"无意义"的值。SQL中的空值有以下几种情况

- 该属性应该有一个值，但当前不知道
- 该属性不应该有值
- 由于某种原因不便与填写，如涉及隐私的信息

空值的判断

判断一个属性是否为空值，使用 **IS NULL** 和 **IS NOT NULL** 语句

**Example** 从Student表中找出漏填了数据的学生信息

```
SELECT *
FROM Student
WHERE Sname IS NULL OR Ssex IS NULL OR Sbirthdate IS NULL OR Smajor IS NULL;
```

由于主码不可能为空值，故 **Sno** 无需判断。

空值约束

- 如果在创建基本表时，属性定义为 **NOT NULL** 约束，则不可以取空值。
- 主码不可以取空值。

空值的运算

- 空值与任何数值进行计算的结果都为NULL
  - 空值与另一个值的比较运算结果为UNKNOWN
- 加入了UNKONOW后的三值逻辑运算真值表

x	y	x AND y	x OR y	NOT x
T	T	T	T	F
T	U	U	T	F
T	F	F	T	F
U	T	U	T	U
U	U	U	U	U
U	F	F	U	U
F	T	F	T	T
F	F	F	F	T
F	U	F	U	T

视图

定义视图

创建视图的基本语法为

SQL

```
CREATE VIEW <视图名>[(<列名>[, <列名>]...)]
AS <子查询>
[WITH CHECK OPTION];
```

子查询为任意包含SELECT的语句

**WITH CHECK OPTION** 表示对视图进行 **UPDATE**、**INSERT** 和 **DELETE** 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。

组成视图的属性列名只能**全部省略**或者**全部指定**，以下情况必须明确指定所有的列名

- 某个目标不是单纯的属性名，而是聚集函数或者表达式
- 多表连接时出现了几个同名列作为视图的字段
- 需要在视图中为某个列启用新的名字

**Example** 建立信息管理与信息系统专业学生的视图

SQL

```
CREATE VIEW IS_Student
AS
SELECT Sno, Sname, Ssex, Sbirthdate, Smajor
FROM Student
WHERE Smajor='信息管理与信息系统';
```

**CREATE VIEW** 执行时，只是把视图的定义存入数据字典，并不执行 **SELECT** 语句。只有对视图查询时，才会按照视图表的定义从基本表中查询。

**Example** 建立信息管理与信息系统专业学生的视图, 且保证在插入数据时，视图中仍然只有信息管理与信息系统的学生

SQL

```
CREATE VIEW IS_Student
AS
SELECT Sno, Sname, Ssex, Sbirthdate, Smajor
FROM Student
WHERE Smajor='信息管理与信息系统'
WITH CHECK OPTION;
```

行列子集视图: 若一个视图从单个基本表导出，并且只是去掉了基本表的某些列和某些行，但保留了主码。

**Example** (由多个表创建视图) 建立信息管理与信息系统专业选修81001号课程的学生的视图(包含学号、姓名、成绩属性)

SQL

```
CREATE VIEW IS_C1(Sno, Sname, Grade)
AS
SELECT Student.Sno, Sname, Grade
FROM Student, SC
WHERE Student.Sno=SC.Sno AND SC.Cno='81001' Smajor="信息管理与信息系统";
```

**Example** (在已有视图的基础上创建视图) 建立信息管理与信息系统专业选修了81001课程且成绩在90分以上的学生视图

SQL

```
CREATE VIEW IS_C2
AS
SELECT Sno, Sname, Grade
FROM IS_C1
WHERE Grade>90;
```

虚拟列:由其他列经过计算所得出的列，系统不存储其数值，而是经过计算得出。

- 带虚拟列的视图称为带表达式的视图

**Example** 将学生的学号姓名年龄定义为一个视图

SQL

```
CREATE S_AGE(Sno, Sname, Sage)
AS
SELECT Sno, Sname, (extract(year from current_date)-extract(year from Sbirthdate))
FROM Student;
```

！最好在修改基本表的结构之后删除由该表导出的视图，然后重建这些视图

## 删除视图

删除视图的基本语句为

SQL

```
DROP VIEW <视图名>[CASCADE];
```

视图删除后其定义将从数据字典中删除，如果该视图还导出了其他视图，则需要使用 **CASCADE** 级联删除语句将其一并删除。

### 查询视图

视图定义之后就可以像基本表一样对视图进行查询。

### 视图消解(view resolution)

关系数据库管理系统在执行对视图的查询时,进行完有效性查询之后,从数据字典中取出视图的定义,把定义中的子查询和用户的查询结合起来,转化为等价的基本表的查询,这一过程称为视图消解。

注:定义视图之后对视图查询和直接对派生表查询还是有区别的。视图一旦定义,该视图保存在数据字典里,之后所有的查询都可以引用该视图(类似于函数),但是派生表只是在语句执行时临时定义的,语句结束之后就被删除。

### 更新视图

由于视图不是实际存储数据的虚表,因此对视图的更新最终都会转化为对基本表的更新。

**Example** 将视图IS\_Student中的2018005号学生的姓名改为"刘新奇"

SQL

```
UPDATE IS_Student
SET Sname='刘新奇'
WHERE Sno='2018005';
```

对其视图消解之后所对应的SQL语句为

SQL

```
UPDATE Student
SET Sname='刘新奇'
WHERE Sno='2018005' AND Smajor='信息管理与信息系统';
```

**Example** 向视图IS\_Student中插入一个新的学生记录(2018207, 赵新, 男, 2001-7-19)

SQL

```
INSERT INTO IS_Student
VALUES('2018207', '赵新', '男', '2001-7-19', '信息管理与信息系统');
```

某些视图是无法更新的,例如视图中的某些列是由其他基本表属性计算而来的

一般行列子集视图是可更新的

### 视图的作用

1. 视图能够为机密数据提供安全保护
2. 视图为重构数据库提供了一定程度的逻辑独立性
3. 视图可以简化用户的操作
4. 视图可以使用户能以多种角度看待同一数据

## 4. 数据库安全性

**数据库的安全性:** 指保护数据库,以防不合法使用所造成的数据泄露、篡改或破坏。

### 4.1数据库的安全性控制

#### 用户身份鉴别

每个用户标识由用户名(user name)和用户标识号(user identification number, UID)组成。

##### 用户鉴别方法

1. 静态口令鉴别
2. 动态口令鉴别: 验证码、动态令牌
3. 生物特征鉴别
4. 智能卡鉴别
5. 入侵检测

#### 存取控制

存取控制主要包括定义用户权限和合法检查。

1. 定义用户权限并将用户权限登记到数据字典
  - 定义用户权限经过编译之后存储在数据字典中,被称为安全规则或授权规则
2. 合法权限检查

△ 定义用户权限和合法权限检查机制一起组成了DBMS的存取控制子系统

#### 自主存取控制方法

大型数据库均支持自主存取控制。SQL主要通过GRANT和REVOKE语句来实现。

用户权限主要由两个部分组成:

- 数据库对象
  - 操作类型
- 定义用户权限就是定义用户能够在那些数据库对象上进行那些操作。

授权: 在数据库中定义存取权限称为授权(authorization)。

## 授予与收回对数据的操作权限

### GRANT语句

GRANT语句的一般格式为

```
GRANT <权限>[, <权限>]...
ON <对象类型> <对象名> [, <对象类型> <对象名>]...
TO <用户>[, <用户>]...
[WITH GRANT OPTION]
```

SQL

该语句的主要作用是将对指定对象的指定操作权限授予指定的用户。  
执行该语句的用户:

- 数据库管理员(DBA)
- 数据库的创建者
- 已经拥有该权限的用户

如果指定了**WITH GRANT OPTION**子句, 则获得某种权限的用户可以把权限授予其他用户。

△ 数据库不允许循环授权!

循环授权就是被授权者又把权限授权给祖先

**Example** 把查询Student表的权限授给用户U1

```
GRANT SELECT
ON TABLE Student
TO U1;
```

SQL

**Example** 把对Student表和Course表的全部操作权限授予用户U2和U3

```
GRANT ALL PRIVILEGES
ON TABLE Student, Course
TO U2, U3;
```

SQL

**Example** 把对表SC的查询权限授予所有用户

```
GRANT SELECT
ON TABLE SC
TO PUBLIC;
```

SQL

### REVOKE 语句

REVOKE用户回收权限, 授予的权限可以由数据库管理员或者其他授权者使用REVOKE语句进行回收。其一般格式为

```
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];
```

SQL

如果语句指定了CASCADE, 则级联收回授予的权限; 如果指定了RESTRICT, 则转授权限后不能收回。默认为RESTRICT。

**Example** 把用户U4修改学生学号的权限收回

```
REVOKE UPDATE(Sno)
ON TABLE Student
FROM U4;
```

SQL

**Example** 收回所有用户对表SC的查询权限

```
REVOKE SELECT
ON TABLE SC
FROM PUBLIC;
```

SQL

**Example** 把用户U5对SC表的INSERT权限收回



SQL

```
REVOKE INSERT
ON TABLE SC
FROM US CASCADE;
```

**自主存取控制:** 用户可以"自主"地将数据的存取权限授予何人，并决定是否也将"授权"的权限授予别人。

### \*创建数据库的权限

#### 数据库角色

数据库角色是被命名的一组数据库操作的相关权限，**角色是权限的集合**。

SQL中，采用CREATE ROLE语句创建角色，并且使用GRANT和REVOKE语句授予和收回权限。

#### 1. 角色的创建

SQL

```
CREATE ROLE <角色名>;
```

#### 2. 给角色授权

SQL

```
GRANT <权限>[, <权限>]...
ON <对象类型>对象名
TO <角色>[, <角色>]...
```

#### 3. 将一个角色权限授予其他角色或用户

SQL

```
GRANT <角色1>[, <角色2>]...
TO <角色3>[, <用户1>]...
[WITH ADMIN OPTION];
```

该语句可以将角色权限授予一个其他角色或者一个用户。

授予者可以是角色的创建者或者获得了WITH ADMIN OPTION 的授权者。

一个角色权限

- 直接授予这个角色的权限
- 由其他角色授予这个角色的权限

#### 4. 角色权限的收回

SQL

```
REVOKE <权限>[, <权限>]...
ON <对象类型> <对象名>
FROM <角色>[, <角色>]...
```

**Example** 通过角色实现将一组权限授予一个用户

1. 创建一个角色R1

SQL

```
CREATE ROLE R1;
```

2. 使用GRANT语句，使得角色R1拥有Student表的SELECT、UPDATE、INSERT权限。

SQL

```
GRANT SELECT, UPDATE, INSERT
ON TABLE Student
TO R1;
```

3. 将这个角色的权限授予王平、张明、赵玲

SQL

```
GRANT R1
TO 王平, 张明, 赵玲;
```

4. 一次性通过R1收回王平的这三个权限

SQL

```
REVOKE R1
FROM 王平;
```

强制存取控制方法

在强制存取控制方法中，数据库管理系统的全部实体被分为主体的客体两大类。  
主体: 主体是系统中的活动实体，既包括数据库管理系统所管理的实际用户，也包括代表用户的各进程。  
客体: 客体是系统中的被动实体，是受主体操纵的，包括文件、基本表、索引、视图等。

对于主体和客体，DBMS为他们的每一个实例都会指派一个敏感度标记(label)。

敏感度标记等级

1. 绝密级(top secret, TS)

2. 机密级(secret, S)

3. 秘密级(confidential, C)

4. 公开(public, P)

许可证级别: 主体的敏感度标记

密级: 客体的密感度标记

当用户以敏感度标记注册进入系统时，系统要求其对于客体的存取必须遵循如下规则

1 主体许可证级别大于或等于客体密级，主体才能读取相应的客体

2 当主体的许可证级别小于或等于客体密级，主体才能写相应的客体

规则2保证了数据只能由密级低到高写入，这样防止了密级高的数据被写入低密级。

强制存取控制是对数据本身进行密级标记，无论数据如何复制，标记和数据都是一个不可分割的整体

视图机制

可以利用视图控制用户的权限，建立视图将部分权限授予用户，从而对视图外的数据看不见。  
Example 建立计算机科学与技术专业的学生的视图，把对该视图的SELECT权限授予王平，把该视图上的所有操作授权于张明

```
CREATE VIEW CS_Student
AS
SELECT *
FROM Student
WHERE Smajor='计算机科学与技术'
WITH CHECK OPTION;

GRANT SELECT
ON CS_Student
TO 王平;

GRANT ALL PRIVILEGES
ON CS_Student
TO 张明;
```

审计(audit)

审计功能把用户对数据库的所有操作自动记录下来，存入审计日志(audit log)。

1. 审计事件
- 服务器事件

系统权限

语句事件: 审计SQL语句

模式对象事件
2. 审计功能
- 基本功能

提供多套审计规则

提供审计分析和报表功能

提供审计日志管理功能

AUDIT和NOAUDIT语句

Example 对修改SC表结构或修改SC表数据的操作进行审计

```
SHOW AUDIT_TRAIL;
SET AUDIT_TRAIL TO ON;

AUDIT ALTER, UPDATE ON SC BY ACCESS;
```

Example 取消对SC表的ALTER和UPDATE操作审计

SQL

```
NOAUDIT ALTER, UPDATE ON SC;
```

数据加密

存储加密

- 数据在底层以密文形式存储

传输加密

将数据库通过网络传输的数据、SQL 语句、登陆系统信息等在传输时进行加密。

安全套接层协议(secure socket layer protocol, SSL protocol)

- 确认通信双方端点的可靠性
- 协商加密算法和密钥
- 可信数据传输

5. 数据库的完整性

🔗 数据库的完整性是指数据库中数据的正确性(correctness)和相容性(compatibility).

- 正确性: 数据库的数据符合现实世界语义且反应实际情况
- 相容性: 数据库的同一对象在不同的关系表中的数据是一致的

为了维护数据库的完整性，关系数据库管理系统必须能够实现以下功能

1. 提供定义完整性约束的机制
2. 提供完整性约束的方法
3. 提供完整性的违约处理方法

1. 实体完整性

实体的完整性检查

- 检查主码值是否唯一，如果不唯一则拒绝插入或修改
- 检查主码的各个属性是否为空，只有一个为空，就拒绝插入或修改

2. 参照完整性

关系模型的参照完整性在CREATE TABLE 中使用 FOREIGN KEY 短语定义哪些列为外码，并且使用REFERENCE 短语指明外码所参照的哪些表的主码

Example 定义SC中的参照完整性

SQL

```
CREATE TABLE SC(  
    Sno CHAR(8),  
    Cno CHAR(5),  
    Grade SMALLINT,  
    Semester CHAR(5),  
    Teachingclass CHAR(8),  
    PRIMARY KEY(Sno, Cno),  
    FOREIGN KEY(Sno) REFERENCES Student(Sno),  
    FOREIGN KEY(Cno) REFERENCES Course(Cno)  
);
```

当不一致发生时，系统可以采取以下策略解决

1. 拒绝执行(默认策略)
2. 级联操作
3. 设置为空值

一般的，系统默认采取默认策略拒绝执行，要想采用其他策略，必须显式的声明。

```
CREATE TABLE SC(
    Sno CHAR(8),
    Cno CHAR(5),
    Grade SMALLINT,
    Semester CHAR(5),
    Teachingclass CHAR(8),
    PRIMARY KEY(Sno, Cno),
    FOREIGN KEY(Sno) REFERENCES Student(Sno),
    ON DELETE CASCADE,
    /*
    当删除Student中的元组时，会级联的删除SC表中的Sno与之相同的元组
    */
    ON UPDATE CASCADE,
    /*
    当修改Student中的元组时，会级联的修改SC表中的Sno与之相同的元组
    */
    FOREIGN KEY(Cno) REFERENCES Course(Cno),
    ON DELETE CASCADE,
    /*
    当删除Student中的元组时，会级联的删除SC表中的Sno与之相同的元组
    */
    ON UPDATE CASCADE,
    /*
    当修改Student中的元组时，会级联的修改SC表中的Sno与之相同的元组
    */
);
```

### 3. 用户定义的完整性

#### 1. 属性上的控制

在**CREATE TABLE**中定义属性的同时，可以根据需求添加属性上的约束，即 **属性限制**

1. 列值非空(NOT NULL)
2. 列值唯一(UNIQUE)
3. 检查列值是否满足一个条件表达式(CHECK 短语)

#### NOT NULL

**Example** 定义SC表时，限制Sno、Cno、Grade属性不为空值

```
CREATE TABLE SC(
    Sno CHAR(8) NOT NULL,
    Cno CHAR(5) NOT NULL,
    Grade SMALLINT NOT NULL,
    Semester CHAR(5),
    Teachingclass CHAR(8),
    PRIMARY KEY(Sno, Cno)
);
```

#### UNQIE

**Example** 建立学院表School，要求学院名称SHname列取值唯一，学院编号SHno列为主码

```
CREATE TABLE School(
    SHno CHAR(8) PRIMARY KEY,
    SHname VARCHAR(40) UNIQUE,
    SHfounddate Date
);
```

#### CHECK

**Example** Student表中的Ssex表属性只允许取“男”或“女”

```
CREATE TABLE Student(
    Sno CHAR(8) PRIMARY KEY,
    Sname CHAR(20) NOT NULL,
    Ssex CHAR(6) CHECK(Ssex IN('男', '女')),
    Sbirthdate Date,
    Smajor VARCHAR(40)
);
```

🔗 往表中插入或者修改属性时，若不满足属性上的约束，将拒绝执行操作。

## 2. 元组上的约束

元组级约束可以添加不同属性之间的相互约束

**Example** 当学生的性别是“男”时，其姓名不能以“Ms.”打头

SQL

```
CREATE TABLE Student(  
    Sno CHAR(8),  
    Sname CHAR(20) NOT NULL,  
    Ssex CHAR(6),  
    Sbirthdate Date,  
    Smajor VARCHAR(40),  
    PRIMARY KEY(Sno),  
    CHECK(Ssex='女' OR Sname NOT LIKE 'Ms.%')  
);
```

## 完整性约束命名子句

SQL还可以在创建TABLE时，使用完整性约束命名子句**CONSTRAINT**从而对完整性约束命名，从而做到灵活的添加和删除一个完整性约束。

SQL

```
CONSTRAINT <完整性约束名> <完整性约束>
```

<完整性约束>包括

- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- CHECK

**Example** 建立“学生”表Student，要求学号在1000000到29999999之间，姓名不能取空值，出生日期在1980年之后，性别只能是“男”或“女”

SQL

```
CREATE TABLE Student(  
    Sno CHAR(8)  
    CONSTRAINT C1 CHECK(Sno BETWEEN '1000000' AND '29999999'),  
    Sname CHAR(20)  
    CONSTRAINT C2 NOT NULL,  
    Sbirthdate Date  
    CONSTRAINT C3 CHECK(Sbirthdate > '1980-1-1'),  
    Ssex CHAR(6)  
    CONSTRAINT C4 CHECK(Ssex IN('男', '女')),  
    Smajor VARCHAR(40),  
    CONSTRAINT StudentKey PRIMARY KEY(Sno)  
);
```

**Example** 建立“教师”表Teacher，要求每个教师的应发工资(每月)不低于3000元。应发工资是工资列Sal和扣除项Deduct之和。

SQL

```
CREATE TABLE Teacher(  
    Eno CHAR(8) PRIMARY KEY,  
    Ename VARCHAR(20),  
    Job CHAR(8),  
    Sal NUMERIC(7, 2),  
    Deduct NUMERIC(7,2)  
    Schoolno CHAR(8),  
    CONSTRAINT TeacherFKey FOREIGN KEY(Schoolno),  
    CONSTRAINT CHECK(Sal + Deduct >= 3000)  
);
```

## 修改表中的完整性限制

SQL

```
/*删除对出生日期的限制*/  
ALTER TABLE Student  
    DROP CONSTRAINT C3;
```

**Example** 修改表Student中的约束条件，要求学号在900000到999999之间，出生日期改为1985年之后

SQL

```
ALTER TABLE Student
    DROP CONSTRAINT C1;
ALTER TABLE Student
    ADD CONSTRAINT C1 CHECK(Sno BETWEEN '900000' AND '999999');
ALTER TABLE Student
    DROP CONSTRAINT C3;
ALTER TABLE Student
    ADD CONSTRAINT C3 CHECK(Sbirthdate > '1985-1-1');
```

## 域的完整性约束

**Example** 建立一个性别域，并声明性别域的取值范围

SQL

```
CREATE DOMAIN GenderDomain CHAR(6)
CHECK(VALUE IN ('男', '女'));
```

这样约束属性时就可以这样声明

SQL

```
Ssex GenderDomain
```

**Example** 建立一个性别域，并对限制命名

SQL

```
CREATE DOMAIN GenderDomain CHAR(6)
CONSTRAINT GD CHECK(VALUE IN ('男', '女'));
```

**Example** 删除域GenderDomain的限制条件GD

SQL

```
ALTER DOMAIN GenderDomain
DROP CONSTRAINT GD;
```

SQL

```
ALTER DOMAIN GenderDomain
ADD CONSTRAINT GGD CHECK(VALUE IN ('0', '1'));
```

## 触发器

### 1. 定义触发器

#### ✎ 触发器

触发器又叫做 **事件-条件-动作规则**(ECA rule).

当特定的系统事件发生时，对规则的条件进行检查，如果条件成立则执行规则中的动作，否则不执行该动作。

SQL使用**CREATE TRIGGER**命令创建触发器

SQL

```
CREATE TRIGGER <触发器名>
| BEFORE | AFTER | <触发事件> ON <表名> /* 指明触发器激活时间是在事件之前还是之后 */
REFERENCING NEW | OLD AS <变量> /* 指出引用的变量 */
FOR EACH { ROW | STATEMENT } /* 定义触发器的类型，指明动作体执行的频率 */
[ WHEN <触发条件> ] <触发动作体> /* 仅当触发条件为真时执行触发体 */
```

MySQL创建触发器

```

CREATE
[DEFINER = user]
TRIGGER [IF NOT EXISTS] trigger_name trigger_time trigger_event

ON tbl_name FOR EACH ROW [trigger_order]

    trigger_body

trigger_time_: { BEFORE | AFTER }

trigger_event_: { INSERT | UPDATE | DELETE }

trigger_order_: { FOLLOWS | PRECEDES } _other_trigger_name_

```

1. 创建触发器: 只有表的拥有者可以在表上创建触发器
2. 表名: 触发器只能定义在基本表上, 不能定义在视图上。该表也被称为触发器的目标表。
3. 触发事件: 触发事件可以是 **INSERT**、**DELETE**、**UPDATE**, 也可以是这几个事件的组合。
  - **AFTER/BEFORE** 是定义触发时机。
4. 触发器类型: 触发器根据所触发的动作的间隔尺寸可以分为行级触发器(**FOR EACH ROW**)和语句级触发器(**FOR EACH STATEMENT**)
5. 触发动作体: 触发动作体可以是一个匿名PL/SQL过程块, 也可以是对已存储的过程的调用。

**Example** 当对表SC的Grade属性进行修改时, 若分数增加了10%, 则将此操作记录到另一个表 **SC\_U(Sno CHAR(8), Cno CHAR(5), Oldgrade SMALLINT, Newgrade SMALLINT)** 中。

SQL

```

CREATE TRIGGER SC_T
AFTER UPDATE ON SC

REFERENCING
    OLD AS OldTuple
    NEW AS NewTuple
FOR EACH ROW
WHEN(NewTuple.Grade >= 1.1*OldTuple.Grade)

BEGIN
    INSERT INTO SC_U(Sno, Cno, OldGrade, NewGrade)
    VALUES(OldTuple.Sno, OldTuple.Cno, OldTuple.Grade, NewTuple.Grade)

END;

```

**Example** 将每次对表Student的插入操作所增加的学生个数记录到表 **Student InsetLog(numbers INT)** 中, 运行触发器之前须初始化次表

SQL

```

CREATE TRIGGER Student_Count
AFTER INSERT ON Student
REFERENCING
    NEW TABLE AS Delta
FOR EACH STATEMENT

BEGIN
    INSERT INTO StudentInsertLog (Numbers)
    SELECT COUNT(*) FROM Delta

END;

```

**Example** 定义一个 **BEFORE** 行级触发器, 为教师表Teacher定义完整性规则: 教授的工资不得低于4000, 如果低于4000, 自动改为4000

SQL

```

CREATE TRIGGER Update_Sal
BEFORE UPDATE ON Teacher
REFERENCING NEW AS newTuple
FOR EACH ROW
BEGIN
    IF(newTuple.job='教授') AND (newTuple.sal < 4000)
    THEN newTuple.sal := 4000;
    END IF;

END;

```

## 2. 执行触发器

当一个表上定义了多个BEFORE和多个AFTER触发器时, 其执行次序为

- 执行该表上的BEFORE触发器
- 激活触发器的SQL语句
- 执行该表上的AFTER触发器

对于同一个表上的多个BEFORE/AFTER语句, 按照“先创建先执行”的规则执行

3. 删除触发器

删除触发器的SQL语句一般为

SQL

```
DROP TRIGGER <触发器名> ON <表名>;
```

6. 关系数据理论

2. 规范化

函数依赖

函数依赖

给定关系模式R(U, F), X, Y是U的子集。若对于R(U,F)的任意一个关系r, r中不可能存在在两个元组在属性X上相等, 而在Y属性值不相等, 则称 **X函数确定Y或Y函数依赖于X**, 记做  $X \rightarrow Y$

- 即属性X到Y的映射为单射
- $X \rightarrow Y$ , 但  $Y \not\subseteq X$ , 则称  $X \rightarrow Y$  是**非平凡**的函数依赖(non-trival functional dependency)
- $X \rightarrow Y$ , 但  $Y \subseteq X$ , 则称  $X \rightarrow Y$  是**平凡**函数依赖(trival functional dependency)
- 若  $X \rightarrow Y$ , 则X称为这个函数依赖的决定属性组, 也称为**决定因素**(determinant)
- 若  $X \rightarrow Y, Y \rightarrow X$ , 则记做  $X \leftrightarrow Y$
- 若Y不函数依赖于X, 则记做  $X \nrightarrow Y$

完全函数依赖

在R(U, F)中, 如果  $X \rightarrow Y$ , 并且对于X的任何一个真子集  $X'$ , 都有  $X' \nrightarrow Y$ , 则称Y对X**完全函数依赖**(full functional dependency), 记作  $X \xrightarrow{F} Y$ .  
若  $X \rightarrow Y$ , 但Y不完全函数依赖于X, 则称Y对X**部分函数依赖**(partial functional dependency), 记作  $X \xrightarrow{P} Y$

传递函数依赖

在R(U, F)中, 如果  $X \rightarrow Y (Y \not\subseteq X), Y \rightarrow Z, Z \not\subseteq Y$ , 则称Z对X**传递函数依赖**(transitive functional dependency). 记为  $X \xrightarrow{T} Z$

码

候选码

设K为R(U,F) 中的属性或者属性组

- 若  $K \xrightarrow{F} U$ , 则K为R的候选码
- 若  $K \xrightarrow{P} U$ , 则称K为U的超码

外码

关系模式R中的属性组X并非R的码, 但X是另一个关系模式的码, 则称X为R的**外码**。

范式

范式分为1NF, 2NF, 3NF, BCNF, 4NF

$$4NF \subset BCNF \subset 3NF \subset 2NF \subset 1NF$$

一个低级范式的关系模式可以通过模式分解(schema decomposition)可以转化为若干个更高级别的关系模式的集合, 这个过程称为规范化。

2NF

若  $R \in 1NF$ , 且每一个非主属性完全依赖于任何一个候选码, 则  $R \in 2NF$ .

- 投影分解法: 将1NF分解为多个关系模式, 使每个模式均满足2NF

3NF

设关系模式  $R(U, F) \in 1NF$ , 若R中不存在这样的码, 属性组Y及非主属性  $Y \not\subseteq Z$ , 使得  $X \rightarrow Y, Y \rightarrow Z$ 成立,  $Y \nrightarrow X$ , 则称  $R(U, F) \in 3NF$

- 投影分解法: 将2NF分解为多个关系模式, 使每个模式均满足3NF, 即没有非主属性对码的传递函数依赖
- Properties**
- 若  $R \in 3NF$ , 则R的每一个非主属性既不函数依赖也不传递依赖于另一组码。

BCNF (Boyce Codd Normal Form)

关系模式  $R(U, F) \in 1NF$ , 若  $X \rightarrow Y$  且  $Y \not\subseteq X$  时必含有码, 则  $R \in BCNF$ .  
即关系模式  $R(U, F)$  中, 若每一个决定因素都包含码, 则  $R(U, F) \in BCNF$ .

- BCNF 消除了插入异常和删除异常



多值依赖

多值依赖

给定关系模式R(U, F),  $X, Y, Z$ 是U的子集, 并且 $Z = U - X - Y$ 。关系模式中多值依赖 $X \twoheadrightarrow Y$ 成立, 当且仅当对R(U,F)的任一关系r给定的一对 $(x, z)$ 值, 有一组Y的值, 这组值仅仅决定于 $x$ 值而与 $z$ 值无关。

4NF

关系模式R(U, F)∈ 1NF, 若果对于R的每个非平凡多值依赖 $X \twoheadrightarrow Y (Y \not\subseteq X)$ , X都含有码, 则称R(U, F) ∈ 4NF

- 不包含“真正的”多值依赖

7.数据库设计

3.概念结构设计

描述概念模型的E-R模型

E-R模型

1. 实体间的联系

两个实体型之间的联系有

- 一对一联系(1:1)
- 一对多联系(1:n)
- 多对多联系(m:n)

两个以上的实体型之间的联系与两个相同。

单个实体型内的联系: 同  
如先修课与其自身为单个实体型的联系。

2. E-R图

E-R图提供了表示实体型、属性和联系的方法。

- 实体型用矩形表示, 矩形内写明实体名。
- 属性用椭圆表示, 并用无向边将其与相应的实体型连接起来。
- 联系用菱形表示, 菱形框内写明联系名, 并用无向边与有关实体型连接起来, 同时在无向边旁注联系的类型。

3. 扩展的E-R模型

ISA联系(is a)

ISA联系用三角形表示。

基数约束

Part-of联系

部分联系, 表示某个实体型是另一个实体型的一部分。

- 独占Part-of联系
- 非独占Part-of联系

强实体型和弱实体型

如果一个实体型依赖于其他的实体型存在, 称为弱实体型。  
如果一个实体型不依赖于其他的实体型而独立存在, 称为强实体型。

8.数据库编程

扩展SQL的功能

WITH子句

WITH子句用来建立一个临时结果表, 该结果尽在SQL语句执行时有效, 不长期存储。  
WITH 语句的语法结构

SQL

```
WITH RS1[(<目标列>, <目标列>)] AS (SELECT 语句1) [,
      RS2[(<目标列>, <目标列>)] AS (SELECT 语句2), ...]
SQL语句;
```

RS1为临时结果集的命名, 对应着 SELECT 语句1的结果。

Example 求81001-01和81001-02两个教学班之间学生选课的平均成绩的差异

```
WITH RS1(grade) AS
  (SELECT AVG(grade) FROM SC WHERE Teachingclass='81001-01'),
  RS2(grade) AS
  (SELECT AVG(grade) FROM SC WHERE Teachingclass='81001-02')
SELECT RS1.grade - RS2.grade FROM RS1, RS2;
```

## WITH RECURSIVE子句

WITH RECURSIVE 子句是WITH子句的特殊情况，用于查找有结构层次的数据。  
语法格式：

```
WITH RECURSIVE RS AS
(
    SEED QUERY
    UNION [ALL]
    RECURSIVE QUERY
)
SQL语句;
```

**Example** 打印“数据库系统概论”课程的所有先修课信息

```
WITH RECURSIVE RS AS
  (SELECT Cjno FROM Course WHERE Cname='数据库系统概论'
   UNION
   SELECT Course.Cjno FROM Course, RS WHERE RS.Cjno=Course.Cno
  )
SELECT Cno, Cname FROM Course WHERE Cno IN (SELECT Cjno FROM RS);
```

## 新的内置函数

**Example** 打印一周内将过生日的学生信息

```
SELECT Sno, Sname, Ssex, Sbirthdate, Smajor
FROM Student
WHERE to_date(to_char(current_date, 'yyyy') || '-' || to_char(Sbirthdate, 'mm-dd'))
BETWEEN current_date AND current_date + interval '7' day;
```

## 通过高级程序设计语言实现复杂功能

### 基于嵌入式SQL的方式

嵌入式SQL就是将SQL语言嵌入到高级编程语言中，一般通过预编译编译为相应的语言  
数据库工作单元与源程序的通信

1. SQL通信区  
SQL通信区(SQL communication area, SQLCA) 主要实现向主语言传递SQL语句的执行状态信息，使得主程序可以控制程序流。
2. 主变量  
主语言主要用主变量(host variable)向SQL语句提供参数。  
SQL语句中的主变量名前要加冒号(:)作为标志。
3. 游标  
游标是关系数据库管理系统为应用程序开设的一个数据缓冲区。

## 过程化SQL

过程化SQL程序的基本结构是块

```
/* 定义部分 */
DECLARE
变量、常量、游标、异常等

/* 执行部分 */
BEGIN
    SQL语句、过程话SQL流程控制语句
    EXCEPTION
    异常处理
END;
```

## 变量和常量的定义

SQL

```
/* 变量 */
变量名 数据类型 [[NOT NULL] :=初值表达式]
变量名 数据类型 [[NOT NULL] 初值表达式]

/* 常量 */
常量名 数据类型 CONSTANT:=常量表达式

/*赋值语句*/
变量名:=表达式
```

## 流程控制

### 条件控制语句

**IF** 语句分为 **IF-THEN**, **IF-THEN-ELSE**  
语句格式:

SQL

```
IF condition THEN
    Sequence_of_statement;
END IF;
```

**IF** 语句可以嵌套。

### 循环控制语句

过程化SQL有三种循环控制语句

- **LOOP**
- **WHILE-LOOP**
- **FOR-LOOP**

SQL

```
LOOP
    Sequence_of_statements;
END LOOP;
```

多数服务器SQL都有 **EXIT**、**BREAK** 或者 **LEAVE** 结束循环体的语句  
**WHILE-LOOP** 语法

SQL

```
WHILE condition LOOP
    Sequence_of_statements
END LOOP;
```

**FOR-LOOP** 语法

SQL

```
FOR count IN [REVERSE] bound1 .. bound2 LOOP
    Sequence_of_statements
END LOOP;
```

- 默认过程为设置count初始值为bound1, 然后依次递增到bound2-1
- 如果设置 **REVERSE**, 则初始值被设置为bound2-1, 依次递减至bound1

## 游标的定义与使用

### 声明游标

SQL

```
DECLARE 游标名[(参数1数据类型, 参数2数据类型, ...)]
CURSOR FOR
SELECT语句;
```

### 打开游标

用 **OPEN** 语句打开游标

SQL

```
OPEN 游标名[(参数1数据类型, 参数2数据类型, ...)]
```

打开游标实际上是在执行 **SELECT** 语句, 把查询结果取到缓存区。

使用游标

推进游标指针并取当前的记录

SQL

```
FETCH 游标名 INTO 变量1 [, 变量2, ...];
```

其中变量必须与 **SELECT** 语句的目标列表达式一一对应。

关闭游标

SQL

```
CLOSE 游标名;
```

**Example** 根据给定学号20180001，使用游标输出该学生的全部选课记录

SQL

```
DECLARE
    CnoOfStudent CHAR(10);
    GradeOfStudent INT;
    mycursor CURSOR FOR
    SELECT Cno, Grade FROM SC WHERE Sno="20180001";

BEGIN
    OPEN mycursor;
    LOOP
        FETCH mycursor INTO CnoOfStudent, GradeOfStudent;
        EXIT WHEN mycursor%NOTFOUND;
        RAISE NOTICE 'Sno:20180001, Cno: %', CnoOfStudent, GradeOfStudent;
    END LOOP;
    CLOSE mycursor;
END;
```

存储过程

过程化SQL也可以将程序命名和编译并保存在数据库中，被称为**存储过程**或**存储函数**。

创建存储过程

SQL

```
CREATE OR REPLACE PROCEDURE 过程名(
    [[IN|OUT|INOUT] 参数1数据类型,
    [[IN|OUT|INOUT] 参数2数据类型, ...]
)
AS <过程化SQL块>;
```

**Example** 给定学生学号，计算学生的平均学分绩点

```

CREATE OR REPLACE PROCEDURE compGPA(
    IN inSno CHAR(10),
    OUT outGPA FLOAT)
AS
DECLARE
    courseGPA INT;
    totalGPA INT;
    totalCredit INT;
    grade INT;
    credit INT;
    mycursor CURSOR FOR
    SELECT Ccredit, Grade FROM SC, Course
        WHERE Sno=inSno and SC.Cno=Course.Cno;

BEGIN
    totalGPA := 0;
    totalCredit := 0;
    OPEN mycursor;
    LOOP
        FETCH mycursor INTO credit, grade;
        EXIT WHEN mycursor%NOTFOUND;
        IF grade BETWEEN 90 AND 100 THEN courseGPA := 4.0;
        ELSEIF grade BETWEEN 80 AND 89 THEN courseGPA := 3.0;
        ELSEIF grade BETWEEN 70 AND 79 THEN courseGPA := 3.0;
        ELSEIF grade BETWEEN 60 AND 69 THEN courseGPA := 3.0;
        ELSE courseGPA :=0;
        END IF;
        totalGPA := totalGPA + courseGPA*credit;
        totalCredit := totalCredit + credit;
    END LOOP;
    CLOSE mycursor;
    outGPA := 1.0 * totalGPA/totalCredit;
END;
)

```

### 执行存储过程

```
CALL/PERFORM [PROCEDURE] 过程名 ([参数1, 参数2, ...]);
```

**Example** 查询学号为20180001的学生平均学分绩点

```

DECLARE outGPA FLOAT;
BEGIN
    CALL compGPA('20180001', outGPA);
    RAISE NOTICE 'GPA: %', outGPA;
END;

```

### 修改存储过程

可以使用 **ALTER PROCEDURE** 重命名一个存储过程

```
ALTER PROCEDURE 过程名1 RENAME TO 过程名2;
```

可以重新编译存储过程

```
ALTER PROCEDURE 过程名 COMPILE;
```

### 删除存储过程

```
DROP PROCEDURE 过程名;
```

### 存储函数

存储函数也称为自定义函数。

存储函数与存储过程类似，不同点为存储函数必须指定返回类型。

创建存储函数

SQL

```
CREATE OR REPLACE FUNCTION 函数名([参数1数据类型, 参数2数据类型, ...])
RETURNS <类型>
AS <过程化SQL块>;
```

执行存储函数

SQL

```
CALL/SELECT 函数名([参数1, 参数2, ...]);
```

修改存储函数

SQL

```
/*重命名*/
ALTER FUNCTION 函数名1 RENAME TO 函数名2;
/*重新编译*/
ALTER FUNCTION 函数名 COMPILE;
```

9. 关系数据库存储管理

数据组织

记录表示

- 定长记录存储
- 变长记录存储

关系表的组织

常见的关系表组织方式:

- 堆存储
- 顺序存储
- 多表聚簇存储
- B+树存储
- 哈希存储

堆存储

表中的一条记录可以存储在该表的任何块中。

顺序存储

表中的各条记录按照指定的属性或属性组的取值大小顺序存储

多表聚簇存储

某些情况下，不同表的记录也可以聚簇存储在同一组块中，可以减少连接操作的开销。

B+树存储

与B+树索引的区别是:

B+树存储方式中的叶节点的数据块存放的不是索引项，而是数据记录。

哈希存储

哈希存储就是将表中指定的属性值建立与存储块之间的哈希函数。

索引结构

常见的索引结构

- 顺序表索引
- 辅助索引
- B+树索引
- 哈希索引
- 位图索引

顺序表索引

- 稠密索引
- 稀疏索引
- 多级索引

辅助索引

辅助索引是建立在表的非排序属性上的索引。

△ Attention

由于辅助索引建立在非排序属性上，则辅助索引一定为稠密索引

B+树索引

课本P 288

哈希存储

哈希存储的两个关键因素:

- 哈希函数
- 哈希表


静态哈希索引

- 哈希函数: 哈希索引借助哈希函数将索引映射到不同的哈希桶中存储
- 哈希表: 哈希表由一组哈希桶组成, 一个桶对应着一个或者多个物理块  
哈希桶的空间是有限的, 当某个桶的存储空间不足时将其称为桶溢出(bucket overflow)  
导致桶溢出的原因:
  - 哈希桶的数量不足
  - 属性取值不均匀, 某些属性值过多
  - 哈希函数设计不合理, 导致索引无法均匀的映射到每个桶
- 哈希表的查找与维护
  - 哈希表适用于等值查找
  - 向基本表中插入元组时, 需要同时向哈希索引中插入相应的索引项, 如果哈希桶已满, 则申请溢出桶并插入。

动态哈希索引

动态哈希索引是随着表数据的增大而定期逐渐增大桶的数目。

位图索引

 位图索引是长度为n的位向量的集合

每个位向量对应于索引属性中的一个可能取值。

10. 关系查询处理和查询优化

查询优化一般分为

- 代数优化: 优化关系代数表达式
- 物理优化: 通过存取路径和底层操作算法选择进行优化

关系数据库管理系统的查询处理

关系数据库管理系统的查询处理可分为:

1. 查询分析
2. 查询检查
3. 查询优化
4. 查询执行

实现查询操作的算法

选择操作的实现

如果选择操作只涉及一个关系, 一般采用**全表扫描(full table scan)** 算法或者**索引扫描(index scan)** 算法,

1. 全盘扫描算法  
假设可以使用的内存为 $M$ 块, 全盘扫描的算法思想为:
  - 1 按照物理次序读Student表的 $M$ 块内存
  - 2 检查内存的每个元组 $t_i$ , 如果 $t_i$ 满足选择条件, 则输出 $t_i$ .
  - 3 如果Student表还有其他块未处理, 则重复1、2
2. 索引扫描法  
如果在选择条件的属性上有索引, 则通过索引先找到满足条件的元组指针, 再通过元组指针在查询的基本表中找到对应元组。

连接操作的实现

Example

SQL

```
SELECT * FROM Student, SC WHERE Student.Sno=SC.Sno;
```

1. 嵌套循环连接(nested loop join)算法  
假设可以使用的内存为 $M$ 块, nested loop join 算法思想为:
  - 1 将Student表的 $M-1$ 块数据读入内存中
  - 2 读入SC表的1块数据
  - 3 对Student表在内存中的每一个元组, 检索SC表在内存中的每一个元组, 如果满足条件, 则串接后作为结果输出
  - 4 继续读入SC表的下一个数据块, 重复3, 直到SC表处理一遍
  - 5 继续读区 $M-1$ 块Student数据, 重复2、4, 直到Student表处理完毕
2. 顺序-合并链接(sort-merge join)算法  
排序合并算法是等值连接常用的算法, 算法步骤为:
  - 1 如果参与连接的Student表和SC表没有排好序, 首先让这两个表按照连接属性排序。
  - 2 将排好序的Student和SC表中的数据块读入到内存中。
  - 3 依次扫描SC表中与Student表具有相同的Sno值的元组, 把他们连接起来
  - 4 当扫描到与Sno不相同的第一个SC元组时, 返回Student表并扫描它的下一个元组, 在扫描SC表中具有相同Sno的元组, 把它们连接起来
  - 5 在内存中的Student表与SC表扫描完后, 继续将相应表余下的数据块读入内存, 重复3、4, 直到Student和SC表全部扫描完。

3. 索引连接(index join)算法

索引连接的步骤如下:

1. 在SC表上建立属性Sno的索引, 若Sno上已有索引则跳过此步
2. 对Student中的每一个元组, 由Sno值通过SC的索引查找相应的SC元组
3. 把这些SC元组和该Student元组连接起来。
4. 循环步骤2、3直到Student表中的元组均处理完毕为止

4. 哈希(hash join)算法

哈希算法将连接属性作为哈希码, 用同一个hash函数对Student和SC表的元组计算哈希值。具体步骤为:

1. 划分(创建hash表)。将包含较少元组的表(Student), 进行一边处理, 把它的元组按hash函数分散到hash表的桶中。
2. 试探阶段, 也称为连接阶段。对另一个表(SC)进行一边处理, 把SC表中的元组按同一个hash函数进行散列, 映射到相应的Student表的hash桶中, 检索该桶中的Student表元组, 将SC表元组与该桶中的与之匹配的Student元组连接起来。

[解读Hash join 算法](#)

hash算法就像是“种萝卜”, 拿以上的连接语句举例, 对于Student中的每一个元组, 根据它的Sno值在内存中“挖一些坑”, 这些坑是有序的, 就比如说一个Sno

为“20180001”的学生, 我就把这个元组种在编号为“00000001”号的坑里, 这样对应的Sno为“20180002”的学生元组就被种在“00000002”号坑中, 依次类推。这个将学生编号Sno对应坑编号的过程便是一种映射, 这个映射就是所谓的**Hash函数**。“挖好坑”之后, 我们只需要对SC表中的元组根据其Sno的值, 就完全可以根据**Hash函数**得出它应该属于哪个萝卜坑, 而这个坑里放着与其对应的Student的元组, 将它们连起来就可以。

## 关系数据库查询优化

查询优化的总目标是:

选择有效策略, 求得给定关系式的值, 使得查询代价最小。

## 代数优化

代数优化是指通过关系表达式的等价变换来提高查询效率。

### 1. 连接运算、笛卡尔积运算的交换律

设 $E_1$ 和 $E_2$ 是关系代数表达式, 则有

$$\begin{aligned}E_1 \times E_2 &\equiv E_2 \times E_1 \\E_1 \bowtie E_2 &\equiv E_2 \bowtie E_1\end{aligned}$$

### 2. 连接运算、笛卡尔积运算的交换律

设 $E_1, E_2, E_3$ 是关系代数表达式,  $F_1$ 和 $F_2$ 是连接运算的条件, 则有

$$\begin{aligned}(E_1 \times E_2) \times E_3 &\equiv E_1 \times (E_2 \times E_3) \\(E_1 \bowtie E_2) \bowtie E_3 &\equiv E_1 \bowtie (E_2 \bowtie E_3) \\(E_1 \bowtie_{F_1} E_2) \bowtie_{F_2} E_3 &\equiv E_1 \bowtie_{F_1} (E_2 \bowtie_{F_2} E_3)\end{aligned}$$

### 3. 投影连接的串接律

$$\Pi_{A_1, A_2, \dots, A_n}(\Pi_{B_1, B_2, \dots, B_n}(E)) \equiv \Pi_{A_1, A_2, \dots, A_n}(E)$$

其中  $\{A_1, A_2, \dots, A_n\}$  是  $\{B_1, B_2, \dots, B_n\}$  的子集。

### 4. 选择运算的串接律

$$\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_1 \wedge F_2}(E)$$

### 5. 选择运算与投影运算的交换律

$$\sigma_F(\Pi_{A_1, A_2, \dots, A_n}(E)) \equiv \Pi_{A_1, A_2, \dots, A_n}(\sigma_F(E))$$

## 11. 数据库恢复技术

## 事务

[事务](#)

事务是用户定义的一个数据库操作序列, 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。

### 事务的ACID特性

- 原子性(atomicity)
- 一致性(consistency)
- 隔离性(isolation)
- 持续性(durability)

## 恢复的实现技术

### 数据转储

建立冗余数据最常用的技术是数据转储和登记日志文件

数据转储根据数据库的状态分为:

- 静态转储: 在系统无事务运行时进行转储
- 动态转储: 在转储过程中允许允许对数据库进行存取或修改, 转储和用户事务可以**并发执行**

数据转储还可以分为:

- 海量转储: 每次转储全部数据库。
- 增量转储: 每次只转储上次转储后更新的数据。



登记日志文件

日志文件是用于记录事务对数据库的更新操作的文件

日志文件的内容和格式

日志文件:

- 1. 各个事务的开始标记(BEGIN TRANSACTION)
- 2. 各个事务的结束标记(COMMIT 或 ROLLBACK)
- 3. 各个事务的所有更新操作  
上面三个过程称为一个日志记录，日志记录包括:
- 4. 事务标识
- 5. 操作的类型
- 6. 操作对象
- 7. 更新前的数据的旧值
- 8. 更新后数据的新值

登记日志文件

登记日志文件必须遵守的原则:

- 1. 登记的次序严格按并发事务执行的时间次序
- 2. 必须先写日志文件，后写数据库

恢复策略

- 1. 事务故障的恢复  
事务故障是指在事务正常运行结束之前被终止。这时可以利用日志文件UNDO此事务实现恢复。  
恢复过程:
  - 反向扫描日志文件，找到此事务的更新操作
  - 对该该事务的更新操作执行逆操作，将日志记录中更新前的值写入
  - 继续反向扫描，执行以上两步
  - 直至扫描到事务的开始标识
- 2. 系统故障恢复  
系统故障造成数据库不一致的原因:
  - 1. 未完成事务对数据库的更新已经写入数据库
  - 2. 已提交的事务对数据库的更新可能还留在缓冲区，未来及写入数据库恢复过程:
  - 正向扫描日志文件，找出故障发生前已经提交的事务(既有BEGIN TRANSACTION又有COMMIT)，将这些事务放入REDO\_LIST; 将发生故障时尚未完成的事务(只有BEGIN TRANSACTION 没有COMMIT)加入UNDO\_LIST.
  - 对UNDO\_LIST中的事务进行撤销操作
  - 对REDO\_LIST中的事务进行重做操作。REDO的方法: 正向扫描日志文件，对每个重做事务重新执行日志文件登记操作，将日志文件中的"更新后的值"写入数据库。
- 3. 介质故障的恢复  
介质故障指的是磁盘上的物理数据和日志文件被破坏。  
其恢复方式为重装数据库，然后重做已完成的事务。  
恢复过程:
  - 1.装入最新的数据库后备副本，将数据库恢复到最近一次转储时的一致性状态。
  - 2.装入相应的日志文件副本(转储结束时刻的日志文件副本)，重做已完成的事务。

具有检查点的恢复技术

1. 检查点记录

- 1. 建立检查点时刻所有的正在执行的事务清单
- 2. 这些事务的最近一个日志记录的地址

2. 重新开始文件

重新开始文件用于记录各个检查点记录在日志文件中的地址。

3. 动态维护日志文件

动态维护日志文件是指周期性的执行 **建立检查点** 和 **保存数据库** 的操作。  
具体步骤:

- 1. 将当前日志缓冲区中的所有日志文件写入磁盘的日志文件中。
- 2. 在日志文件中写入一个检查点记录。
- 3. 将当前数据缓冲区的所有数据记录写入磁盘的数据库中。
- 4. 把检查点记录在日志文件中的地址写入一个重新开始文件。

4. 恢复子系统的恢复策略

系统使用检查点方法进行恢复的步骤:

- 1. 从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，由该地址在日志文件中找到最后一个检查点记录
- 2. 由检查点记录得到检查点建立时刻所有正在执行的事务清单ACTIVE-LIST

 Note

建立两个事务队列: UNDO-LIST和REDO-LIST。  
将ACTIVE-LIST暂时放入UNDO-LIST队列，REDO-LIST暂时为空

- 3. 从检查点开始正向扫描日志文件
  - 1. 如果有新开始的事务，则将其放入UNDO-LIST
  - 2. 如果有已提交的事务，则将其从UNDO-LIST放入UNDO-LIST
- 4. 对UNDO-LIST中的事务执行UNDO操作，对REDO-LIST中的事务执行REDO操作。

12. 并发控制

并发操作所带来的数据不一致:

- 1. 丢失修改: 两个事务 $T_1, T_2$ 读入同一数据, 各自进行修改, 一条修改将另一条修改破坏。
  - 2. 脏读: 事务 $T_1$ 将某一数据修改后写回磁盘, 事务 $T_2$ 读取数据之后, 事务 $T_1$ 撤回修改。此时 $T_2$ 读取的就是"脏"数据。
  - 3. 不可重复读
  - 4. 幻读
- 并发控制的主要技术:
- 封锁(locking)
  - 时间戳(timestamp)
  - 乐观方法(optimistic scheduler)
  - 多版本并发控制(multi-version concurrency control, MVCC)

封锁

封锁就是在某个事务对数据 $A$ 进行操作时, 其他事务无法对数据 $A$ 进行读取和修改 $A$ 。

基本的封锁类型:

- 排他型锁(exclusive lock, 简称 $X$ 锁)
  - 共享型锁(shared lock, 简称 $S$ 锁)
- 排他型锁又称为write lock, 当事务 $T$ 对数据对象 $A$ 加上了 $X$ 型锁, 则指允许 $T$ 对 $A$ 进行读取和修改。
- 共享型锁有称为read lock, 当事务 $T$ 对数据对象 $A$ 加上了 $S$ 型锁, 则事务 $T$ 只能读取 $A$ , 但无法修改 $A$ , 其他事务只能再对 $A$ 加 $S$ 型锁, 而不能加 $X$ 锁, 直到 $T$ 释放 $A$ 上的 $X$ 型锁。

封锁协议

封锁协议(locking protocol)

- 如何申请 $S$ 、 $X$ 型锁
- 持锁时间
- 何时释放

一级封锁协议

🔗 一级封锁协议

一级封锁协议指事务 $T$ , 在修改数据 $R$ 之前必须先对其加 $X$ 锁, 直到事务结束才释放。

一级封锁协议可以解决丢失修改的问题。

二级封锁协议

🔗 二级封锁协议

二级封锁协议在一级封锁协议的基础上, 增加事务 $T$ , 在读取数据 $R$ , 之前必须先对其加 $S$ 型锁, 读取完之后可释放 $S$ 型锁。

可以解决丢失修改和脏读问题。

三级封锁协议

🔗 三级封锁协议

三级封锁协议在一级封锁协议的基础上, 增加事务 $T$ , 在读取数据 $R$ , 之前必须先对其加 $S$ 型锁, 事务完成之后可释放 $S$ 型锁。

可以解决丢失修改和脏读以及不可重复读。

活锁和死锁

避免活锁的简单方法为先来先服务。先进先出

🔗 死锁(deadlock)

假设有事务 $T_1, T_2$ , 事务 $T_1$ 封锁了数据 $R_1$ , 事务 $T_2$ 封锁了数据 $R_2$ , 这时事务 $T_1$ 请求封锁 $R_2$ , 同时事务 $T_2$ 请求封锁 $R_1$ , 这时就形成了事务 $T_1$ 需要等待 $T_2$ 结束, 而事务 $T_2$ 需要等待 $T_1$ 结束的局面, 形成死锁

死锁的解决方法:

- 避免死锁的产生
- 或允许死锁产生, 采取手段检测死锁, 将其解除

预防死锁

- 1. 一次封锁法  
一次封锁法就是指的事务必须一次性对所有要使用的数据全部加锁, 否则不能继续执行。

⚠ 缺点

- 一次性将所有需要数据加锁, 扩大了封锁范围, 降低了系统的并发性
- 数据库中的数据是变化的, 有些数据可能执行过程中由不要求封锁变为封锁, 这种变化无法预测

- 2. 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, 所有事务按照顺序封锁数据。

#### △ 缺点

- 数据库中的数据极多，随着数据的变化维护封锁顺序代价很高
- 事务的封锁请求 可以随着事务的执行而动态决定，无法事先预知事务要封锁的数据对象

### 死锁的诊断与解除

数据库一般采用诊断与解除方法解决死锁问题

1. 超时法  
如果一个事务的等待时间超过了规定的时限，则认为发生了死锁。
2. 事务等待图法  
事务等待图法是建立一个有向图G(T, U), T表示正在执行的事务的集合，U表示边的集合，所有的边为有向边，如果事务 $T_1$ 等待事务 $T_2$ ， 则一条有向边从 $T_1$ 指向 $T_2$ 。

✍ 如果等待图中出现了回路，则证明出现了死锁

### 并发调度的可串行性

#### 可串行化调度

多个事务的并发执行是正确的，当且仅当其结果与按某一次串行执行这些事务的结果相同。

#### 冲突可串行化调度

##### ✍ 冲突操作

冲突操作是指不同的事务对同一个数据的读写操作和写写操作:

$$R_i(x) \text{ 与 } W_j(x)$$

$$W_i(x) \text{ 与 } W_j(x)$$

##### ✍ 冲突可串行调度

如果一个调度 $S_c$ 在保证冲突操作次序不变的情况下，通过交换两个不冲突事务的操作次序得到一个串行调度, 则称调度 $S_c$ 为冲突可串行的调度。

若一个调度是冲突可串行化的，则一定是可串行化的调度

### 两段锁协议

#### ✍ 两段锁协议

所有事务必须分为两个阶段对数据项加锁和解锁。

- 在对任何数据进行读、写操作之前，首先要申请并获得对该数据的封锁
- 在释放一个封锁之后，事务不再申请任何其他封锁

○ 若并发执行的所有事务都遵循两段锁协议，则对这些事务的任何并发调度都是可串行化的。

- 上述条件为并发调度可串行化的充分条件

### 封锁的粒度

封锁对象的大小称为 封锁力度(lock granularity)

#### 多粒度封锁

## 13. 数据库管理系统

### 数据库管理系(DBMS)的基本功能

1. 数据库定义和创建
2. 数据的组织、存储和管理
3. 数据存取
4. 数据库事务管理和运行管理
5. 数据库的建立和维护
6. 其他功能

### 数据库管理系统的层次结构

1. 最上层为应用层
2. 第二层为语言处理层
3. 第三层为数据存取层
4. 第四层为数据存储层

### 语言处理层

主要任务: 把用户在各种方式下提交的数据库语句和数据控制语句转化为对关系数据库系统内层可执行的基本存取模块的调用序列。

数据库语言包括:

- 数据定义语言
- 数据操纵语言
- 数据控制语言

对数据操纵语句的处理过程

对数据库操纵语句的处理步骤如下:

- 1. 对数据库操纵语句进行词法检查和语法分析
- 2. 根据数据字典中的内容进行查询检查, 包括语义检查、审核用户的存取权限、完整性检查和视图消解
- 3. 对查询进行优化

🔗 束缚(binding)

将数据操纵语言语句转化为一串可执行的存取动作的这一过程被称为一个逐步束缚的过程。

数据存取层

数据存取层所涉及的主要数据结构为:

- 逻辑数据记录
- 逻辑块
- 逻辑存取路径

数据存取层的主要任务

- 1. 提供一次一个元组的查找、插入、删除、修改等基本操作
- 2. 提供元组查找所循的存取路径以及对存取路径的维护操作
- 3. 对记录和存取路径的封锁、解锁操作
- 4. 其他辅助操作, 如扫描、合并/排序等

数据存取层的功能子系统

1. 记录存取、事务管理子系统

记录存取子系统:

- 在某个存取路径上按属性值找元组(FIND)
  - 按相对位置找元组(NEXT, PRIOR, FIRST, LAST)
  - 给某个关系增加一个元组(INSERT)
  - 从找到的元组中取某个属性值(GET)
  - 从某关系中删除一个元组(DELETE)
  - 把修改完的元组写回关系中(REPLACE)
- 事务管理子系统:
- 定义事务的开始(BEGIN TRANSACTION)
  - 事务提交(COMMIT)
  - 事务回滚(ROLLBACK)
- 事务管理系统提供的这些操作将登记入日志文件

2. 日志登记子系统

日志登记系统和事务管理子系统密切配合, 完成关系数据库对事务和数据库的恢复任务。  
对日志文件的主要操作有:

- 写日志记录(WROTELOG)
- 读日志记录(READLOG)
- 扫描日志文件(SCANLOG)
- 撤销尚未结束的事务(UNDO)
- 重做已经结束的事务(REDO)

3. 控制信息管理模块

该模块利用内存中专门的数据区登记不同记录类型以及不同存取路径的说明信息和控制信息。  
该模块提供对数据字典中说明信息的读取、增加、删除和修改操作。

4. 排序/合并子系统

排序的主要用途

- 1. 输出有序结果
- 2. 数据预处理  
对于进行一些关系运算时, 当参与运算的关系无法全部读取到内存中时, 对其进行预处理, 对其排序, 在进行操作, 可以将算法时间复杂度由 $O(n^2)$ 降为  $O(n\lg n)$
- 3. 支持动态建立索引结构
- 4. 减少数据块的存取次数

存取路径维护子系统

对数据执行插入、删除、修改操作的同时, 要对相应的存取路径进行维护。  
例如, 如果采用B+树作为存取路径, 则对元组进行插入、删除、修改操作时要对该表上的所有B+树索引进行动态维护。

封锁子系统

封锁子系统完成并发控制。

缓冲区管理

数据存取层的下面是**数据存储层**。

🔗 主要功能

存储管理:

- 缓冲区管理
- 内外存交换管理
- 外存管理