# 第3章: 结构化查询语言

## Structured Query Language (SQL)

李东博

哈尔滨工业大学
计算学部
物联网与泛在智能研究中心
电子邮件: ldb@hit.edu.cn

2023年春

# 教学内容[1]

1. SQL数据定义
   - 基本数据类型
   - 创建关系模式
   - 修改关系模式
   - 删除关系模式
   - 定义视图
2. SQL数据更新
   - 插入数据
   - 修改数据
   - 删除数据
   - 数据完整性检查
3. SQL数据查询
   - 单关系查询
   - 连接查询
   - 嵌套查询

[1]课件更新于2023年2月16日

# 3.1 SQL数据定义

## SQL Data Definition

# SQL基本数据类型

## 数值型

- INT或INTEGER: 整数，取值范围取决于DBMS实现
- SMALLINT: 整数，取值范围比INT小
- BIGINT: 整数，取值范围比INT大
- NUMERIC(p, s)或DECIMAL(p, s): 定点数，$p$位有效数字，小数点后$s$位
- FLOAT(n): 浮点数，精度至少为$n$位数字
- REAL: 同FLOAT，但精度由DBMS确定
- DOUBLE PRECISION: 同FLOAT，但精度由DBMS确定，比REAL高

## 布尔型

- BOOLEAN: 真值TRUE或FALSE

# SQL基本数据类型(续)

## 字符串型

- CHAR(n)或CHARACTER(n): 定长字符串，长度为$n$
- VARCHAR(n)或CHARACTER VARYING(n): 变长字符串，最大长度为$n$
- CLOB或CHARACTER LARGE OBJECT: 超长字符串，长度超过VARCHAR(n)类型字符串的长度上限

## 二进制串型

- BINARY(n): 定长二进制串，长度为$n$个字节
- VARBINARY(n): 变长二进制串，最大长度为$n$个字节
- BLOB或BINARY LARGE OBJECT: 超长二进制串，长度超过VARBINARY(n)类型二进制串的长度上限

# SQL基本数据类型(续)

## 日期时间型

- DATE: 日期，格式YYYY-MM-DD
- TIME: 时间，格式HH:MM:SS
- TIME WITH TIME ZONE: 时间，包含时区
- TIMESTAMP: 日期和时间，格式YYYY-MM-DD HH:MM:SS
- TIMESTAMP WITH TIME ZONE: 日期和时间，包含时区

## 时间区间型

- INTERVAL YEAR TO MONTH: 时间区间，用年和月表示
- INTERVAL DAY TO SECOND: 时间区间，用日、时、分、秒表示

# MySQL基本数据类型

## 数值型

- 整型
  - INTEGER或INT: 4字节
  - SMALLINT: 2字节
  - TINYINT: 1字节
  - MEDIUMINT: 3字节
  - BIGINT: 8字节
- 定点型
  - DECIMAL(p,s): 有效数字共$p$位，小数点后$s$位
  - NUMERIC(p,s): 同DECIMAL(p,s)
- 浮点型
  - FLOAT: 单精度，4字节
  - DOUBLE: 双精度，8字节
- 二进制数型
  - BIT(b): $b$位二进制数$(1 \leq b \leq 64)$，$(b+7)/8$字节，输入时用b'101'表示5

# MySQL基本数据类型(续)

## 日期时间型

- YEAR: 年类型
  - 存储: 1字节整数YYYY
  - 显示和输入格式: 'YYYY'
- DATE: 日期型
  - 存储: 3字节整数$\text{YYYY} \times 16 \times 32 + \text{MM} \times 32 + \text{DD}$
  - 显示和输入格式: 'YYYY-MM-DD'
- TIME: 时间型
  - 存储: 3字节整数$\text{DD} \times 24 \times 3600 + \text{HH} \times 3600 + \text{MM} \times 60 + \text{SS}$
  - 显示和输入格式: 'HH:MM:SS'或 'HHH:MM:SS'
- DATETIME: 日期时间型(与时区无关)
  - 存储: 8字节整数，其中4字节存储整数YYYYMMDD，另外4字节存储整数HHMMSS
  - 显示和输入格式: 'YYYY-MM-DD HH:MM:SS'
- TIMESTAMP: 时间戳型(和UNIX时间戳相同，与时区相关)
  - 存储: 4字节整数，保存从1970-01-01 00:00:00 UTC以来的秒数
  - 显示和输入格式: 'YYYY-MM-DD HH:MM:SS'

# MySQL基本数据类型(续)

## 字符串型

- CHAR(n): 定长字符串型，最多存储$n$个字符($n \leq 255$)
  - 存储空间固定
  - 当字符串不足$n$个字符时，后面用空格补全
- VARCHAR(n): 变长字符串型，最多存储$n$个字符($n \leq 65535$)
  - 存储空间根据字符串的实际字节长度$L$变化
- TEXT: 文本型
  - TINYTEXT: $L < 2^8$
  - TEXT: $L < 2^{16}$
  - MEDIUMTEXT: $L < 2^{24}$
  - LONGTEXT: $L < 2^{32}$

| 类型 | 所需存储空间($L$: 字符串的实际字节长度) |
|---|---|
| CHAR(n) | $nw$，其中$w$是字符的最大长度 |
| VARCHAR(n) | 如果$0 \leq n \leq 255$，则需$L+1$字节；如果$n > 255$，则需$L+2$字节 |
| TINYTEXT | $L+1$字节 |
| TEXT | $L+2$字节 |
| MEDIUMTEXT | $L+3$字节 |
| LONGTEXT | $L+4$字节 |

# MySQL基本数据类型(续)

## 二进制串型

- BINARY(n): 定长二进制串型，最多存储$n$个字节($n \leq 255$)
  - 存储空间固定，$n$字节
  - 当二进制串不足$n$个字节时，后面用\0补全
- VARBINARY(n): 变长二进制串型，最多存储$n$个字节($n \leq 65535$)
  - 存储空间根据二进制串实际字节长度$L$变化
- BLOB: 二进制对象型(Binary Large OBject)
  - TINYBLOB: $L < 2^8$
  - BLOB: $L < 2^{16}$
  - MEDIUMBLOB: $L < 2^{24}$
  - LONGBLOB: $L < 2^{32}$

| 类型 | 所需存储空间($L$: 字符串的实际字节长度) |
|---|---|
| BINARY(n) | $n$字节 |
| VARBINARY(n) | 如果$0 \leq n \leq 255$，则需$L+1$字节；如果$n > 255$，则需$L+2$字节 |
| TINYBLOB | $L+1$字节 |
| BLOB | $L+2$字节 |
| MEDIUMBLOB | $L+3$字节 |
| LONGBLOB | $L+4$字节 |

# MySQL基本数据类型(续)

## 枚举型

- ENUM(值列表): 枚举型
  - 例: ENUM('Mercury', 'Venus', 'Earth', 'Mars')
  - 枚举型的值只能取自值列表
  - 值列表中最多包含65535个不同的值
  - 如果值列表中包含至多255个值，则占1字节；如果值列表中包含超过255个值，则占2字节

## 集合型

- SET(值列表): 集合型
  - 例: SET('Mercury', 'Venus', 'Earth', 'Mars')
  - 集合型的值只能是值集合的子集，如 'Mercury,Earth'
  - 值列表中最多包含64个不同的值
  - 存储为二进制数，列表中每个值对应一个二进制位。值在子集中存在，则相应位置为1；否则，置为0。例如，'Mercury,Earth'存储为0101
  - 占用空间大小取决于集合中元素的个数

# 创建关系模式

- 功能: 定义关系模式，包括关系名、属性名、属性类型、主键、外键、完整性约束等
- SQL语句: CREATE TABLE

## Example (创建关系模式)

创建Student关系，其模式如下:

| 属性名 | 属性类型 | 属性含义 |
|---|---|---|
| Sno | CHAR(6) | 学号 |
| Sname | VARCHAR(10) | 姓名 |
| Ssex | CHAR | 性别 |
| Sage | INT | 年龄 |
| Sdept | VARCHAR(20) | 所在系 |

SQL语句:

```
CREATE TABLE Student (
  Sno CHAR(6),
  Sname VARCHAR(10),
  Ssex CHAR,
  Sage INT,
  Sdept VARCHAR(20)
);
```

# 声明主键

- 功能: 声明关系的主键(一个关系仅有一个主键)，有两种声明方法
- SQL关键词: PRIMARY KEY

**Example (声明主键)**

将Sno属性声明为Student关系的主键

方法1:

```
CREATE TABLE Student (
  Sno CHAR(6) PRIMARY KEY,
  Sname VARCHAR(10),
  Ssex CHAR,
  Sage INT,
  Sdept VARCHAR(20)
);
```

方法2:

```
CREATE TABLE Student (
  Sno CHAR(6),
  Sname VARCHAR(10),
  Ssex CHAR,
  Sage INT,
  Sdept VARCHAR(20),
  PRIMARY KEY (Sno)
);
```

声明多个属性构成的主键时，只能用第2种方法

# 声明外键

- 功能: 声明关系的外键(一个关系可以有多个外键)
- SQL关键词: FOREIGN KEY

## Example (声明外键)

创建SC关系，其模式如下:

| 属性名 | 属性类型 | 属性含义 |
|---|---|---|
| Sno | CHAR(6) | 学号 |
| Cno | CHAR(4) | 课号 |
| Grade | INT | 成绩 |

{Sno, Cno}是SC的主键，Sno是SC的外键，参照Student关系的主键Sno

```
CREATE TABLE SC (
  Sno CHAR(6),
  Cno CHAR(4),
  Grade INT,
  PRIMARY KEY (Sno, Cno),
  FOREIGN KEY (Sno) REFERENCES Student(Sno));
```

# 声明外键(续)

## Example (声明外键)

因为SC中的外键{Sno}参照Student的主键{Sno}，所以可以省略FOREIGN KEY子句中被参照关系Student中的被参照属性列表(Sno)

```
CREATE TABLE SC (
  Sno CHAR(6),
  Cno CHAR(4),
  Grade INT,
  PRIMARY KEY (Sno, Cno),
  FOREIGN KEY (Sno) REFERENCES Student
);
```

# 声明用户定义完整性约束

- 功能:
  1. 规定属性值非空: NOT NULL
  2. 规定属性值不重复: UNIQUE
  3. 定义属性的缺省值: DEFAULT 缺省值
  4. 规定属性值必须满足表达式给出的条件: CHECK (表达式)[2]

**Example (声明用户定义完整性约束)**

```
CREATE TABLE Student (
  Sno CHAR(6),
  Sname VARCHAR(10) NOT NULL,
  Ssex CHAR NOT NULL CHECK (Ssex IN ('M', 'F')),
  Sage INT DEFAULT 0 CHECK (Sage >= 0),
  Sdept VARCHAR(20),
  PRIMARY KEY (Sno)
);
```

[2]MySQL只解析CHECK，但存储引擎并不处理

# 删除关系

- 功能: 删除关系
- SQL语句: DROP TABLE 关系名1，关系名2，...，关系名$n$;
- 删除关系时会连同关系中的数据一起删除!

# 修改关系模式

- 功能: 修改关系模式的定义
  1. 修改关系名
  2. 增加、修改、删除属性
  3. 增加、删除约束
- SQL语句: ALTER TABLE

**Example (修改关系名)**

将Student关系改名为XueSheng

```
ALTER TABLE Student RENAME TO XueSheng;
```

# 修改关系模式(续)

**Example (增加属性)**

在Student关系中增加属性Mno，记录学生的班长的学号

```
ALTER TABLE Student ADD Mno CHAR(6);
```

**Example (删除属性)**

删除Student关系的Mno属性

```
ALTER TABLE Student DROP Mno;
```

# 修改关系模式(续)

## Example (增加表约束)

将属性Mno声明为Student关系的外键，参照Student的主键Sno

```
ALTER TABLE Student
ADD CONSTRAINT fk_mno
FOREIGN KEY (Mno) REFERENCES Student(Sno);
```

## Example (删除表约束)

删除Student关系中Mno属性上的外键约束fk_mno

```
ALTER TABLE Student DROP CONSTRAINT fk_mno;
```

# 修改关系模式(续)

不同DBMS中用于修改属性定义的语法不同

- MySQL中修改属性名的方法

  ALTER TABLE Student CHANGE 旧属性名 新属性名 属性定义;

- MySQL中修改属性定义的方法

  ALTER TABLE Student MODIFY 属性名 属性定义;

- PostgreSQL和openGauss中修改属性定义的方法见下面的例子

**Example (修改属性名)**

将Student关系的Sno属性改名为XueHao

```
ALTER TABLE Student RENAME Sno TO XueHao;
```

**Example (修改属性类型)**

将Student关系中Sname属性的类型修改为VARCHAR(20)

```
ALTER TABLE Student ALTER Sname TYPE VARCHAR(20);
```

# 修改关系模式(续)

**Example (给属性设置非空约束)**

将Student关系的Sname属性设置为非空

```
ALTER TABLE Student ALTER Sname SET NOT NULL;
```

**Example (取消属性的非空约束)**

取消Student关系中Sname属性的非空约束

```
ALTER TABLE Student ALTER Sname DROP NOT NULL;
```

# 修改关系模式(续)

**Example (设置属性缺省值)**

将Student关系的Sage属性的缺省值设置为18

```
ALTER TABLE Student ALTER Sage SET DEFAULT 18;
```

**Example (取消属性缺省值)**

取消Student关系中Sage属性的缺省值

```
ALTER TABLE Student ALTER Sage DROP DEFAULT;
```

# 定义视图(View)

讲完SQL查询后再回来讲 ▸ SQL数据查询

- 基本关系: 关系数据库中真实存储的关系
- 视图: 从用户的视角所看到的数据
- 功能: 创建、修改、删除视图
- SQL语句: CREATE VIEW、ALTER VIEW、DROP VIEW

# 创建视图

- 语法: CREATE VIEW 视图名 [(属性名列表)] AS 子查询;[3]

## Example (创建视图)

为选修了3006号课的计算机系(CS)的学生建立视图，列出学号、姓名和成绩

```
CREATE VIEW CS_Student_on_DB AS
  SELECT Sno, Sname, Grade
  FROM Student NATURAL JOIN SC
  WHERE Sdept = 'CS' AND Cno = '3006';
```

[3] MySQL对CREATE VIEW中的子查询有很多限制条件，参考https://dev.mysql.com/doc/refman/5.5/en/create-view.html

# 修改视图

不同的DBMS使用的ALTER VIEW的语法差别很大，下面给出的是MySQL的ALTER VIEW语法

- 语法: ALTER VIEW 视图名 [(属性名列表)] AS 子查询;

**Example (修改视图)**

- 修改视图CS_Student_on_DB，增加性别属性

```
ALTER VIEW CS_Student_on_DB AS
  SELECT Sno, Sname, Ssex, Grade
  FROM Student NATURAL JOIN SC
  WHERE Sdept = 'CS' AND Cno = '3006';
```

# 删除视图

- 语法: DROP VIEW 视图名;

# 查询视图

- 语法: 与SQL查询的语法相同

**Example (查询视图)**

查询计算机系(CS)没有通过“Database Systems”课考试的学生

```
SELECT Sno, Sname
FROM CS_Student_on_DB
WHERE Grade < 60;
```

# 视图的作用

学习视图的概念有助于具体理解第1章讲过的“三层模式结构”

- 关系模式就是概念模式(conceptual schema)，视图就是外模式(external schema)
- 一个关系数据库只有一个概念模式，但可以有多个外模式
- CREATE VIEW定义了从外模式(视图)到概念模式的映射
- 若概念模式发生变化，只需用ALTER VIEW重新定义外模式(视图)到概念模式的映射，而无需修改基于视图编写的查询，从而实现了逻辑数据独立性(logical data independence)

# 物化视图(Materialized Views)

- 视图本质上是虚拟的表，但某些DBMS允许在物理上存储视图中的数据，这样的视图称作物化视图(materialized view)
- 如果用于定义物化视图的关系发生改变，则物化视图必须被更新
- 优点: 查询效率高
- 缺点: 产生视图维护代价

**Example (PostgreSQL和openGauss中的物化视图)**

- 创建物化视图: `CREATE MATERIALIZED VIEW`
- 删除物化视图: `DROP MATERIALIZED VIEW`
- 修改物化视图: `ALTER MATERIALIZED VIEW`
- 刷新物化视图: `REFRESH MATERIALIZED VIEW`

# 3.2 SQL数据更新

## SQL Data Modification

# SQL数据更新

- 插入数据
  - 直接插入元组
  - 插入子查询结果(讲完SQL查询后再讲)
- 修改数据
  - 基于本关系的修改
  - 基于其他关系的修改(讲完SQL查询后再讲)
- 删除数据
  - 基于本关系的删除
  - 基于其他关系的删除(讲完SQL查询后再讲)
- 数据完整性检查
  - 实体完整性检查
  - 参照完整性检查
  - 用户定义完整性检查
- 更新视图$\star$

# 直接插入元组

- 功能: 向一个关系中直接插入元组
- 语法: INSERT INTO 关系名 [(属性名1, ..., 属性名n)] VALUES (表达式1, ..., 表达式n);
- 说明:
  - 列出的属性名与表达式个数相同
  - 表达式$i$的值是该元组属性$i$的值
  - 若没给出属性名列表，则表示插入一条完整的元组，其属性顺序与关系模式中定义的属性顺序相同
  - 若只列出部分属性名，则该元组的其他属性值为空

**Example (插入元组)**

1. ```
   INSERT INTO Student
   VALUES ('MA-002', 'Cindy', 'F', 19, 'Math');
   ```
2. ```
   INSERT INTO Student (Sno, Sname, Sage, Ssex)
   VALUES ('MA-002', 'Cindy', 19, 'F');
   ```

# 基于本关系的数据修改

- 功能: 修改一个关系中的数据
- 语法: UPDATE 关系名 SET 属性名1 = 表达式1, ..., 属性名n = 表达式n [WHERE 修改条件];
- 说明:
  - 若WHERE子句存在，则将满足修改条件的元组中属性$i$的值修改为表达式$i$的值
  - 若WHERE子句不存在，则将所有元组中属性$i$的值修改为表达式$i$的值
  - 修改条件仅涉及该关系本身

**Example (修改数据)**

1. 将学号为MA-002的学生的年龄修改为20岁

```
UPDATE Student SET Sage = 20 WHERE Sno = 'MA-002';
```

2. 将所有学生的年龄增加1岁

```
UPDATE Student SET Sage = Sage + 1;
```

# 基于本关系的数据删除

- 功能: 删除一个关系中的元组
- 语法: DELETE FROM 关系名 [WHERE 删除条件];
- 说明:
  - 若WHERE子句存在，则将满足删除条件的元组从关系中删除
  - 若WHERE子句不存在，则将所有元组从关系中删除(这与DROP TABLE不同，因为它并不删除关系本身)
  - 删除条件仅涉及该关系本身

**Example (修改数据)**

1. 将学号为MA-002的学生元组删除

```
DELETE FROM Student WHERE Sno = 'MA-002';
```

# 数据完整性检查

- 目的: 数据修改可能导致数据库违反完整性约束，需要对数据完整性进行检查
- 功能: 检查数据库是否满足完整性约束
  1. 实体完整性约束检查
  2. 参照完整性约束检查
  3. 用户定义完整性约束检查

# 实体完整性约束检查

## 检查时机

- 插入元组时
- 修改元组的主键属性值时

## 处理方法

- 如果插入或修改的元组的某个主键属性值为空，则拒绝插入或修改元组
- 如果插入或修改的元组的主键值与现有某个元组相同，则拒绝插入或修改元组

# 用户定义完整性约束检查

## 检查时机

- 插入元组时
- 修改元组时

## 检查方法

- 若更新后的数据库不满足用户定义完整性约束，则拒绝插入或修改元组

# 参照完整性检查

有四种情况可能会破坏参照完整性

## 情况1

在参照关系(如SC)中插入一个元组，该元组的外键(如Sno)的值在被参照关系(如Student)的主键(如Sno)值中找不到，即悬空(dangling)。若发生这种情况，则拒绝插入该元组。

SC

| Sno | Cno | Grade |
|---|---|---|
| PH-001 | 1002 | 92 |
| PH-001 | 2003 | 85 |
| PH-001 | 3006 | 88 |
| CS-001 | 1002 | 95 |
| CS-001 | 3006 | 90 |
| CS-002 | 3006 | 80 |
| MA-001 | 1002 | |
| MA-003 | 1002 | |

拒绝插入

Student

| Sno | Sname | Ssex | Sage | Sdept |
|---|---|---|---|---|
| PH-001 | Nick | M | 20 | Physics |
| CS-001 | Elsa | F | 19 | CS |
| CS-002 | Ed | M | 19 | CS |
| MA-001 | Abby | F | 18 | Math |
| MA-002 | Cindy | F | 19 | Math |

# 参照完整性检查(续)

## 情况2

修改参照关系(如SC)中的一个元组，造成该元组的外键(如Sno)的值在被参照关系(如Student)的主键(如Sno)值中找不到。若发生这种情况，则拒绝修改该元组。

SC

| Sno | Cno | Grade |
|---|---|---|
| PH-001 | 1002 | 92 |
| PH-001 | 2003 | 85 |
| PH-001 | 3006 | 88 |
| CS-001 | 1002 | 95 |
| CS-001 | 3006 | 90 |
| CS-002 | 3006 | 80 |
| ~~MA-001~~ MA-003 | 1002 | |

拒绝修改

Student

| Sno | Sname | Ssex | Sage | Sdept |
|---|---|---|---|---|
| PH-001 | Nick | M | 20 | Physics |
| CS-001 | Elsa | F | 19 | CS |
| CS-002 | Ed | M | 19 | CS |
| MA-001 | Abby | F | 18 | Math |
| MA-002 | Cindy | F | 19 | Math |

# 参照完整性检查(续)

## 情况3

从被参照关系(如Student)中删除一个元组，造成参照关系(如SC)中某些元组的外键(如Sno)的值在被参照关系(如Student)的主键(如Sno)值中找不到。如果发生这种情况，则采取下列处理方法之一:

1. 当参照关系(如SC)中没有任何元组的外键值与被参照关系(如Student)中待删除元组的主键值相对应时，方可删除元组；否则拒绝删除该元组。
2. 级联(cascade)删除参照关系(如SC)中相关联的元组
3. 将参照关系(如SC)中相关联元组的外键(如Sno)值置为空(NULL)

SC

| Sno | Cno | Grade |
|---|---|---|
| PH-001 | 1002 | 92 |
| PH-001 | 2003 | 85 |
| PH-001 | 3006 | 88 |
| CS-001 | 1002 | 95 |
| CS-001 | 3006 | 90 |
| CS-002 | 3006 | 80 |
| MA-001 | 1002 | |

Student

| Sno | Sname | Ssex | Sage | Sdept |
|---|---|---|---|---|
| PH-001 | Nick | M | 20 | Physics |
| CS-001 | Elsa | F | 19 | CS |
| CS-002 | Ed | M | 19 | CS |
| ~~MA-001~~ | ~~Abby~~ | ~~F~~ | ~~18~~ | ~~Math~~ |
| MA-002 | Cindy | F | 19 | Math |

拒绝删除

# 参照完整性检查(续)

## 情况4

修改被参照关系(如Student)中一个元组，造成参照关系(如SC)中某些元组的外键(如Sno)的值在被参照关系(如Student)的主键(如Sno)值中找不到。如果发生这种情况，则采取下列处理方法之一:

1. 当参照关系(如SC)中没有任何元组的外键值与被参照关系(如Student)中待修改元组的主键值相对应时，方可修改元组；否则拒绝修改该元组。
2. 级联(cascade)修改参照关系(如SC)中相关联的元组
3. 将参照关系(如SC)中相关联元组的外键(如Sno)值置为空(NULL)

SC

| Sno | Cno | Grade |
|---|---|---|
| PH-001 | 1002 | 92 |
| PH-001 | 2003 | 85 |
| PH-001 | 3006 | 88 |
| CS-001 | 1002 | 95 |
| CS-001 | 3006 | 90 |
| CS-002 | 3006 | 80 |
| ~~MA-001~~ | 1002 | |

Student

| Sno | Sname | Ssex | Sage | Sdept |
|---|---|---|---|---|
| PH-001 | Nick | M | 20 | Physics |
| CS-001 | Elsa | F | 19 | CS |
| CS-002 | Ed | M | 19 | CS |
| ~~MA-001~~ MA-003 | Abby | F | 18 | Math |
| MA-002 | Cindy | F | 19 | Math |

# 参照完整性检查(续)

- 在FOREIGN KEY子句中声明，当违反参照完整性约束时，如何处理
- 语法: 在FOREIGN KEY子句的末尾加上

  ON DELETE NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT
  ON UPDATE NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT
  - RESTRICT或NO ACTION: 拒绝删除或修改，缺省处理方式
  - CASCADE: 级联删除或修改
  - SET NULL: 置为空值
  - SET DEFAULT: 置为被参照属性的缺省值

**Example (参照完整性检查)**

1. FOREIGN KEY (Sno) REFERENCES Student(Sno)
   ON DELETE RESTRICT
   ON UPDATE RESTRICT

# 插入查询结果

讲完SQL查询后再回来讲 ▸ SQL数据查询

- 功能: 将一个查询的全部结果插入到另一个关系中
- 语法: INSERT INTO 关系名 子查询;

## Example (修改数据)

1. 将计算机系学生的学号、姓名、选课数、平均分插入到关系CS_Grade(Sno, Sname, Amt, AvgGrade)中

```
INSERT INTO CS_Grade
  SELECT Sno, Sname, COUNT(*), AVG(Grade)
  FROM Student NATURAL JOIN SC
  WHERE Sdept = 'CS'
  GROUP BY Sno, Sname;
```

# 基于外部关系的数据修改

- 待修改的元组在一个关系中，而修改条件涉及其他的关系

**Example (修改数据)**

将计算机系(CS)全体学生的成绩置零

1. 
```
UPDATE SC SET Grade = 0 WHERE 'CS' =
  (SELECT Sdept FROM Student WHERE Student.Sno = SC.Sno);
```
2. 
```
UPDATE SC SET Grade = 0 WHERE Sno IN
  (SELECT Sno FROM Student WHERE Sdept = 'CS');
```
3. 
```
UPDATE SC SET Grade = 0 WHERE EXISTS
  (SELECT * FROM Student
   WHERE Student.Sno = SC.Sno AND Sdept = 'CS');
```

# 基于外部关系的数据删除

- 待删除的元组在一个关系中，而删除条件涉及其他的关系

**Example (修改数据)**

删除计算机系所有学生的选课记录

1. 
```
DELETE FROM SC WHERE 'CS' =
  (SELECT Sdept FROM Student WHERE Student.Sno = SC.Sno);
```
2. 
```
DELETE FROM SC WHERE Sno IN
  (SELECT Sno FROM Student WHERE Sdept = 'CS');
```
3. 
```
DELETE FROM SC WHERE EXISTS
  (SELECT * FROM Student
   WHERE Student.Sno = SC.Sno AND Sdept = 'CS');
```

# 更新视图(View)[4]

- 和基本关系不同，不是所有视图都可以更新
- 视图可修改的(updatable)条件
  - 视图中的元组和基础关系中的元组存在1对1关系
- 如果视图的定义符合以下情况之一，则视图不可修改(non-updatable)
  - 包含聚集函数
  - 包含DISTINCT
  - 包含GROUP BY
  - 包含UNION
  - FROM子句中包含子查询
  - FROM子句中包含不可更新的视图
  - 包含特定类型连接
  - WHERE子句包含相关子查询
- 视图可插入的(insertable)条件
  - 视图可修改
  - 视图的属性不能定义为表达式
  - 视图中不能包含基本关系中同一属性的多个副本

[4] https://dev.mysql.com/doc/refman/5.5/en/view-updatability.html

# 3.3 SQL数据查询

## SQL Data Query

# SQL数据查询

1. 单关系查询
2. 连接查询
3. 嵌套查询

# 3.3.1 单关系查询

# 示例数据库College

```
INSERT INTO Student VALUES
('PH-001', 'Nick', 'M', 20, 'Physics'),
('CS-001', 'Elsa', 'F', 19, 'CS'),
('CS-002', 'Ed', 'M', 19, 'CS'),
('MA-001', 'Abby', 'F', 18, 'Math'),
('MA-002', 'Cindy', 'F', 19, 'Math');
```

```
INSERT INTO SC VALUES
('PH-001', '1002', 92),
('PH-001', '2003', 85),
('PH-001', '3006', 88),
('CS-001', '1002', 95),
('CS-001', '3006', 90),
('CS-002', '3006', 80),
('MA-001', '1002', NULL);
```

Student

| Sno | Sname | Ssex | Sage | Sdept |
|---|---|---|---|---|
| PH-001 | Nick | M | 20 | Physics |
| CS-001 | Elsa | F | 19 | CS |
| CS-002 | Ed | M | 19 | CS |
| MA-001 | Abby | F | 18 | Math |
| MA-002 | Cindy | F | 19 | Math |

SC

| Sno | Cno | Grade |
|---|---|---|
| PH-001 | 1002 | 92 |
| PH-001 | 2003 | 85 |
| PH-001 | 3006 | 88 |
| CS-001 | 1002 | 95 |
| CS-001 | 3006 | 90 |
| CS-002 | 3006 | 80 |
| MA-001 | 1002 | |

# 投影查询

- 功能: 从一个关系中选出指定的列
- 语法: SELECT [DISTINCT] 属性名列表 FROM 关系名;
- 说明:
  - 查询结果中属性的顺序与属性名列表给出的属性顺序相同
  - 不加DISTINCT，则不去除结果中的重复元组；加上DISTINCT，则去除结果中的重复元组
  - 若要返回关系中所有的列，可将属性名列表可简写为*

**Example (投影查询)**

1. 查询学生的学号和姓名

```
SELECT Sno, Sname FROM Student;
```

2. 查询所有系名

```
SELECT DISTINCT Sdept FROM Student;
```

3. 查询全部学生信息

```
SELECT * FROM Student;
```

# 投影查询(续)

- 可将投影查询中的属性名替换为表达式，做更复杂的投影操作
- 语法: SELECT [DISTINCT] 表达式列表 FROM 关系名;
- 说明:
  - 表达式可以是常量、算术表达式、函数表达式、逻辑表达式
  - 查询结果中表达式列的名称就是该表达式的字符串，可以用“表达式 AS 属性名”将表达式列重命名

**Example (扩展的投影查询)**

1. 查询学生的学号和姓名(姓名全大写)

```
SELECT Sno, UPPER(Sname) FROM Student;
```

2. 查询学生的姓名和出生年份

```
SELECT Sname, (2022 - Sage) AS bd FROM Student;
```

# 选择查询

- 功能: 从一个关系中选择满足给定条件的元组
- 语法: SELECT [DISTINCT] 表达式列表 FROM 关系名 WHERE 选择条件;

## Example (选择查询)

1. 查询计算机系(CS)全体学生的学号和姓名

```
SELECT Sno, Sname FROM Student WHERE Sdept = 'CS';
```

2. 查询计算机系(CS)全体男同学的学号和姓名

```
SELECT Sno, Sname FROM Student
WHERE Sdept = 'CS' AND Ssex = 'M';
```

3. 查询计算机系(CS)和数学系(Math)全体学生的学号和姓名

```
SELECT Sno, Sname FROM Student
WHERE Sdept = 'CS' OR Sdept = 'Math';
```

# 选择查询条件

## 表达式比较

- 语法: 表达式1 比较运算符 表达式2
- 比较运算符: =，<，<=，>，>=，!=，<> (不等于)

## 范围比较

- 语法: 表达式1 [NOT] BETWEEN 表达式2 AND 表达式3
- 功能: 判断表达式1的值是否(不)在表达式2和表达式3之间

## 集合元素判断

- 语法: 表达式1 [NOT] IN (表达式2，...，表达式n)
- 功能: 判断表达式1的值是否(不)在表达式2, ..., 表达式$n$的值构成的集合中

# 选择查询条件(续)

## 字符串匹配

- 语法: 字符串表达式 [NOT] LIKE 模式 [ESCAPE 转义字符]
- 功能: 判断字符串表达式的值是否匹配给定的含有通配符的模式
  - 通配符_: 匹配单个字符
  - 通配符%: 匹配任意长度的字符串(含空串)
  - 可通过ESCAPE子句指定转义字符，默认转义字符为\

## Example (字符串匹配)

1. 查询姓名首字母为E的学生的学号和姓名

```
SELECT Sno, Sname FROM Student WHERE Sname LIKE 'E%';
```

2. 查询姓名为4个字母且首字母为E的学生的学号和姓名

```
SELECT Sno, Sname FROM Student WHERE Sname LIKE 'E___';
```

# 选择查询条件(续)

## 字符串正则表达式匹配

- 语法: 字符串表达式 [NOT] 表示正则表达式匹配的关键词 模式
  - PostgreSQL和openGauss使用关键词SIMILAR TO
  - MySQL使用关键词REGEXP或RLIKE
  - Oracle使用关键词REGEXP_LIKE
  - SQL Server使用关键词LIKE
- 功能: 判断字符串表达式的值是否匹配给定的正则表达式

## Example (正则表达式)

1. 查询姓名首字母为E或F的学生的学号和姓名

```
SELECT Sno, Sname FROM Student
WHERE Sname REGEXP '^[EF].*';
```

# 选择查询条件(续)

## 空值(NULL)判断

- 语法: 属性名 IS [NOT] NULL
- 功能: 判断属性值是否(不)为空
- 错误写法: 属性名 = NULL，属性名 <> NULL，属性名 != NULL。这些错误写法的结果均为UNKNOWN (既非真，也非假)

## Example (空值判断)

查询选了课但还未取得成绩的学生

```
SELECT Sno FROM SC WHERE Grade IS NULL;
```

## Question

下面的*SQL*语句的结果是什么?

```
SELECT Sno FROM SC WHERE Grade = NULL;
```

# 选择查询条件(续)

## 逻辑运算

- 逻辑运算符: AND, OR, NOT
- 逻辑运算真值表

| AND | TRUE | FALSE | UNKNOWN |
|---|---|---|---|
| TRUE | TRUE | FALSE | UNKNOWN |
| FALSE | FALSE | FALSE | FALSE |
| UNKNOWN | UNKNOWN | FALSE | UNKNOWN |

| OR | TRUE | FALSE | UNKNOWN |
|---|---|---|---|
| TRUE | TRUE | TRUE | TRUE |
| FALSE | TRUE | FALSE | UNKOWN |
| UNKNOWN | TRUE | UNKOWN | UNKNOWN |

| NOT | TRUE | FALSE | UNKNOWN |
|---|---|---|---|
| | FALSE | TRUE | UNKNOWN |

- 只有使查询条件为真的元组才能出现在查询结果中
- UNKNOWN只能表示逻辑运算结果，不能用作布尔型属性的值

# 集合操作

- 功能：求两个查询语句结果的并、交、差
- 语法：
  1. 求并：查询语句1 UNION [ALL] 查询语句2
  2. 求交：查询语句1 INTERSECT 查询语句2[5]
  3. 求差：查询语句1 MINUS/EXCEPT 查询语句2[6]
- 说明：
  - 查询语句1的结果的属性名将作为集合操作结果的属性名
  - 若使用关键词ALL，则并集不去重
  - 即使两个查询语句的结果都是有序的，集合操作的结果也未必有序

[5] Oracle、SQL Server、PostgreSQL和openGauss支持INTERSECT，MySQL不支持INTERSECT

[6] Oracle用MINUS；SQL Server和PostgreSQL用EXCEPT；openGauss既可以用MINUS，也可以用EXCEPT；MySQL两者都不支持

# 集合操作

## Example (集合操作)

1. 查询选修了1002号或3006号课的学生的选课信息

```
SELECT * FROM SC WHERE Cno = '1002' UNION ALL
SELECT * FROM SC WHERE Cno = '3006';
```

2. 查询选修了1002号或3006号课的学生的学号

```
SELECT Sno FROM SC WHERE Cno = '1002' UNION
SELECT Sno FROM SC WHERE Cno = '3006';
```

## Question

⋆⋆⋆ 在*MySQL*中如何实现INTERSECT和EXCEPT?

# 查询结果排序

- 功能: 按照指定的顺序，对查询结果关系中的元组进行排序
- 语法: 在查询语句的后面加上ORDER BY 属性名1 [ASC|DESC], ..., 属性名n [ASC|DESC]
  - 按照(属性1，...，属性$n$)的字典序进行排序
  - ASC表示升序(默认)，DESC表示降序
- 若排序属性含空值，ASC: 属性值为空的元组排在最前；DESC: 属性值为空的元组排在最后
- 通常用ORDER BY子句对最终查询结果排序，而不对中间结果排序

## Example (结果排序)

1. 查询计算机系(CS)全体学生的学号和姓名，并按学号升序排列

```
SELECT Sno, Sname FROM Student WHERE Sdept = 'CS'
ORDER BY Sno;
```

2. 查询全体学生的信息，结果按所在系升序排列，同一个系的学生按年龄降序排列

```
SELECT * FROM Student ORDER BY Sdept ASC, Sage DESC;
```

# 限制查询结果数量

- 功能: 限制查询结果中元组的数量(不是标准SQL的内容)
- 语法: 在查询语句的后面加上LIMIT [偏移量,] 结果数量或者LIMIT 结果数量 [OFFSET 偏移量][7]
- 从偏移量(默认是0)位置的元组开始，返回指定数量的元组

**Example (限制查询结果数量)**

1. 查询3006号课得分最高的前2名学生的学号和成绩

```
SELECT Sno, Grade FROM SC WHERE Cno = '3006'
ORDER BY Grade DESC LIMIT 2;
```

[7] Oracle和SQL Server使用不同的语法

# 聚集(Aggregation)查询

- 功能: 计算一个关系上某表达式所有值的聚集值(值的个数COUNT、最大值MAX、最小值MIN、总和SUM、平均值AVG)[8]
- 语法: SELECT 聚集函数([DISTINCT] 表达式) FROM...WHERE...

| | |
|---|---|
| COUNT(*) | 所有元组的数量 |
| COUNT(表达式) | 非空表达式值的数量 |
| COUNT(DISTINCT 表达式) | 不同的非空表达式值的数量 |
| MAX([DISTINCT] 表达式) | 表达式的最大值 |
| MIN([DISTINCT] 表达式) | 表达式的最小值 |
| SUM(表达式) | 表达式值的和 |
| SUM(DISTINCT 表达式) | 不同的表达式值的和 |
| AVG(表达式) | 表达式值的平均值 |
| AVG(DISTINCT 表达式) | 不同的表达式值的平均值 |

[8]DBMS还支持其他聚集函数，PostgreSQL中的聚集函数参考http://postgres.cn/docs/12/functions-aggregate.html；MySQL中的聚集函数参考https://dev.mysql.com/doc/refman/5.5/en/group-by-functions.html

# 聚集(Aggregation)查询(续)

## Example (聚集查询)

1. 查询计算机系全体学生的数量

```
SELECT COUNT(*) FROM Student WHERE Sdept = 'CS';
```

2. 查询计算机系学生的最大年龄

```
SELECT MAX(Sage) FROM Student WHERE Sdept = 'CS';
```

- 聚集函数不能出现在WHERE子句中! (重要的事情说三遍)
- 例: 查询年龄最大的学生的学号

```
SELECT Sno FROM Student WHERE Sage = MAX(Sage);
```

写法错误，正确的做法是使用嵌套查询(第3.3.3节讲)

## Question

下面两个*SQL*语句的结果有什么区别?

1. `SELECT COUNT(*) FROM SC;`
2. `SELECT COUNT(Grade) FROM SC;`

# 分组(Group By)查询

- 功能: 关系代数的分组操作
- 语法: SELECT 分组属性列表, 聚集函数表达式列表 FROM 关系名 WHERE 选择条件 GROUP BY 分组属性列表;
- 说明:
  1. 根据指定的分组属性, 对一个关系中的元组进行分组, 分组属性值相同元组的分为一组
  2. 对每个组中元组的非分组属性的值进行聚集
  3. 分组查询语句中不能包含分组属性及聚集函数表达式以外的其他表达式(为什么?)

## Example (分组查询)

1. 统计每门课的选课人数和平均成绩

```
SELECT Cno, COUNT(*), AVG(Grade) FROM SC GROUP BY Cno;
```

2. 统计每个系的男生人数和女生人数

```
SELECT Sdept, Ssex, COUNT(*) FROM Student
GROUP BY Sdept, Ssex;
```

# 分组(Group By)查询(续)

- 分组完成后，经常需要按照组内元组的统计信息对分组进行筛选
- 语法: SELECT 分组属性列表, 聚集函数表达式列表 FROM 关系名 WHERE 选择条件 GROUP BY 分组属性列表 HAVING 分组筛选条件;

**Example (分组查询)**

1. 查询选修了2门以上课程的学生的学号和选课数

```
SELECT Sno, COUNT(*) FROM SC
GROUP BY Sno HAVING COUNT(*) >= 2;
```

2. 查询2门以上课程得分超过80的学生的学号及这些课程的平均分

```
SELECT Sno, AVG(Grade) FROM SC WHERE Grade >= 80
GROUP BY Sno HAVING COUNT(*) >= 2;
```

# 分组(Group By)查询(续)

## 注意事项

- SELECT子句的目标列中只能包含分组属性和聚集函数[9]
- WHERE子句的查询条件中不能出现聚集函数
- HAVING子句的分组筛选条件中可以使用聚集函数
- WHERE、GROUP BY和HAVING的执行顺序
  1. 按照WHERE子句给出的条件，从关系中选出满足条件的元组
  2. 按照GROUP BY子句指定的分组属性，对元组进行分组
  3. 按照HAVING子句给出的条件，对分组进行筛选

[9]单纯出于性能考虑，MySQL允许某些非分组属性出现在目标列中https://dev.mysql.com/doc/refman/5.5/en/group-by-handling.html，PostgreSQL也类似

# 窗口函数(Window Function)

- 一个窗口函数在一组与当前行有某种关联关系的行上执行一种计算
- 窗口函数与聚集函数有相似之处，但窗口函数不会将多个元组聚集成一个元组

## Example (窗口函数)

列出每名已选课学生的单科成绩以及在课程中的排名

```
SELECT Sno, Cno, Grade,
  RANK() OVER (PARTITION BY Cno ORDER BY Grade DESC) AS Rank
  FROM SC;
```

| Sno | Cno | Grade | Rank |
|---|---|---|---|
| PH-001 | 1002 | 92 | 2 |
| PH-001 | 2003 | 85 | 1 |
| PH-001 | 3006 | 88 | 2 |
| CS-001 | 1002 | 95 | 1 |
| CS-001 | 3006 | 90 | 1 |
| CS-002 | 3006 | 80 | 3 |
| MA-001 | 1002 | 91 | 3 |

# 单关系查询习题 I

在PostgreSQL、openGauss或MySQL上创建Products数据库(Database Systems The Complete Book - Exercise 2.4.1)，并使用SQL完成下列查询

`SELECT ... FROM ...`

1. What are the manufacturers?
2. Find the model numbers and the hard disk size (in GB) of all PC's

`SELECT ... FROM ... WHERE ...`

1. What PC models have a speed of at least 3.00 and ram of at least 1024MB?
2. What PC models have a speed of at least 3.00 or ram of at least 1024MB?
3. What models does the manufacturer A produce?
4. Find the model numbers of all color laser printers

`UNION`

# 单关系查询习题 II

1. Find the model numbers and price of all PC's and all laptops
2. What manufacturers make all types of products (PC, laptop, and printer)?

`SELECT ... FROM ... GROUP BY ...`

1. How many models does every manufacturer have?
2. How many models does every manufacturer have for every type of products?

`SELECT ... FROM ... GROUP BY ... HAVING ...`

1. Find those hard-disk sizes that occur in two or more PC's
2. What manufacturers make all types of products (PC, laptop, and printer)?

`SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ...`

1. Find those hard-disk sizes of at least 100GB that occur in two or more PC's

# 3.3.2 连接查询

# 笛卡尔积

- 功能: 计算多个关系的笛卡尔积
- 语法1: SELECT ... FROM 关系名1, 关系名2, ..., 关系名n
- 语法2: SELECT ... FROM 关系名1 CROSS JOIN 关系名2 CROSS JOIN ... CROSS JOIN 关系名n

**Example (笛卡尔积)**

1. 查询学生及其选课情况，列出学号、姓名、课号、得分

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

# 内连接

- 功能: 按照给定的连接条件，对两个关系做内连接
- 语法: SELECT ... FROM 关系名1 [INNER] JOIN 关系名2 ON 连接条件

**Example (内连接)**

1. 查询学生及其选课情况，列出学号、姓名、课号、得分

```
SELECT Student.Sno, Sname, Cno, Grade
FROM Student JOIN SC ON (Student.Sno = SC.Sno);
```

# 内连接(续)

- 当内连接是等值连接，且连接属性同名时，可使用如下语法
- 语法: 关系名1 [INNER] JOIN 关系名2 USING (连接属性列表)
- 连接属性只会保留一个副本

**Example (内连接)**

1. 查询学生及其选课情况，列出学号、姓名、课号、得分

```
SELECT Student.Sno, Sname, Cno, Grade
FROM Student JOIN SC USING (Sno);
```

# 自然连接

- 功能: 两个关系做自然连接
- 语法: 关系名1 NATURAL JOIN 关系名2

**Example (自然连接)**

1. 查询学生及其选课情况，列出学号、姓名、课号、得分

```
SELECT Student.Sno, Sname, Cno, Grade
FROM Student NATURAL JOIN SC;
```

# 自连接

- 功能: 一个关系与其自身进行连接
- 语法: 与其他连接操作相同
- 注意:
  - 参与连接的关系在物理上是同一个关系，但在逻辑上看作两个关系，因此用AS必须重命名
  - 当为关系取了别名后，但凡用到该关系时都必须使用其别名，不能再使用原关系名
  - 属性名前必须加别名做前缀

## Example (自连接)

1. 查询和Elsa在同一个系学习的学生的学号和姓名

```
SELECT S2.Sno, S2.Sname
FROM Student AS S1 JOIN Student AS S2
  ON S1.Sdept = S2.Sdept AND S1.Sno != S2.Sno
WHERE S1.Sname = 'Elsa';
```

# 外连接

- 功能：两个关系做外连接
- 语法：
  1. 左外连接：关系名1 LEFT [OUTER] JOIN 关系名2 ON 连接条件
  2. 右外连接：关系名1 RIGHT [OUTER] JOIN 关系名2 ON 连接条件
  3. 全外连接：关系名1 FULL [OUTER] JOIN 关系名2 ON 连接条件[10]
  4. 自然外连接：关系名1 NATURAL LEFT|RIGHT [OUTER] JOIN 关系名2
- 当外连接是等值连接，且连接属性同名时，可使用USING (连接属性列表)声明连接条件，但连接属性只会保留一个副本（如果是左（右）外连接，则保留左（右）关系中的连接属性）

## Example (外连接)

1. 查询没有选课的学生的学号和姓名

```
SELECT Student.Sno, Sname
FROM Student LEFT JOIN SC ON (Student.Sno = SC.Sno)
WHERE Cno IS NULL;
```

[10] Oracle和SQL Server支持FULL OUTER JOIN，MySQL不支持FULL OUTER JOIN

# 连接执行顺序

- 因为自然连接操作满足交换律和结合律，所以$R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$有$n!$种执行顺序
- 连接执行顺序通常是由DBMS的查询优化器决定的
- 在PostgreSQL和openGauss中，可以通过设置配置参数`join_collapse_limit = 1`来强制查询优化器采用SQL语句中明确给出连接顺序

# 连接查询习题 I

在PostgreSQL、openGauss或MySQL上创建Products数据库(Database Systems The Complete Book - Exercise 2.4.1)，并使用SQL完成下列查询

内连接

1. What manufactures make laptops with a hard disk of at least 100GB?
2. What PC models with a price less than $500 does the manufacturer A produce?
3. Find the manufacturers of PC's with at least three different speeds

自连接

1. Find those pairs of PC models that have both the same speed and RAM (a pair should be listed only once)
2. Find the model numbers of all printers that are cheaper than the printer model 3002
3. Find those hard-disk sizes that occur in two or more PC's

# 连接查询习题 II

外连接

1. ⋆⋆ Find the PC model with the highest available speed

# 3.3.3 嵌套查询

# 嵌套查询

- 查询块: 一个SELECT-FROM-WHERE语句称为一个查询块(block)
- 嵌套查询: 将一个查询块嵌套在另一个查询块中得到的查询称为嵌套查询(nested query)，内层查询块称为子查询(subquery)
- 子查询的类型
  - 不相关子查询(independent subquery): 子查询不依赖于外层查询
  - 相关子查询(correlated subquery): 子查询依赖于外层查询

## Example (子查询的类型)

查询和Elsa在同一个系学习的学生的学号和姓名(含Elsa)。似曾相识?

1. 使用不相关子查询

```
SELECT Sno, Sname FROM Student WHERE Sdept =
  (SELECT Sdept FROM Student WHERE Sname = 'Elsa');
```

2. 使用相关子查询

```
SELECT Sno, Sname FROM Student AS S WHERE EXISTS
  (SELECT * FROM Student AS T
   WHERE T.Sname = 'Elsa' AND T.Sdept = S.Sdept);
```

# 嵌套查询的写法

1. 在集合判断条件中使用子查询：使用[NOT] IN
2. 在比较条件中使用子查询：使用比较运算符
3. 在存在性测试条件中使用子查询：使用[NOT] EXISTS
4. 在FROM子句中使用子查询
5. 在WITH子句中使用子查询：使用WITH

# 嵌套查询写法1: 在集合判断条件中使用子查询

- 功能: 判断表达式的值是否(不)属于子查询的结果
- 语法: 表达式 [NOT] IN (子查询)
- 先执行子查询，后执行父查询

## Example (嵌套查询的写法)

1. 查询和Elsa在同一个系学习的学生的学号和姓名(含Elsa)

```
SELECT Sno, Sname FROM Student WHERE Sdept IN
  (SELECT Sdept FROM Student WHERE Sname = 'Elsa');
```

2. 查询选修了“Database Systems”的学生的学号和姓名

```
SELECT Sno, Sname FROM Student WHERE Sno IN
  (SELECT Sno FROM SC WHERE Cno IN
    (SELECT Cno FROM Course
     WHERE Cname = 'Database Systems'));
```

很显然，这个嵌套查询既不易读，也不高效。有更好的做法吗?

# 嵌套查询写法2: 在比较条件中使用子查询

- 功能: 将表达式的值与子查询的结果进行比较
- 语法: 表达式 比较运算符 [ALL|ANY|SOME] (子查询)
  - ALL: 当表达式的值与子查询结果中任意的值都满足比较条件时，返回真；否则，返回假
  - ANY或SOME: 当表达式的值与子查询结果中某个值满足比较条件时，返回真；否则，返回假
  - 如果子查询结果应仅包含单个值，则无需在比较运算符后面加ALL、ANY或SOME
- 先执行子查询，后执行父查询

**Example (嵌套查询的写法)**

1. 查询和Elsa在同一个系学习的学生的学号和姓名(含Elsa)

```
SELECT Sno, Sname FROM Student WHERE Sdept =
  (SELECT Sdept FROM Student WHERE Sname = 'Elsa');
```

# 嵌套查询写法2: 在比较条件中使用子查询(续)

## Example (嵌套查询的写法)

1. 查询年龄最大的学生的学号(第2次出现，上次是什么时候出现的?)

```
SELECT Sno FROM Student WHERE Sage =
  (SELECT MAX(Sage) FROM Student);
```

2. 查询比计算机系(CS)全体学生年龄都大的学生的学号

```
SELECT Sno FROM Student WHERE Sage > ALL
  (SELECT Sage FROM Student WHERE Sdept = 'CS');
```

3. 查询学生平均年龄比全校学生平均年龄大的系

```
SELECT Sdept FROM Student
GROUP BY Sdept
HAVING AVG(Sage) > (SELECT AVG(Sage) FROM Student);
```

# 嵌套查询写法3: 在存在性测试条件中使用子查询

- 功能: 判断子查询结果是否(不)为空
- 语法: [NOT] EXISTS (子查询)
- 子查询的SELECT后无需列出目标列，只需用SELECT *，因为我们只判断子查询结果是否为空，并不需要使用子查询结果

## Example (嵌套查询的写法)

1. 查询和Elsa在同一个系学习的学生的学号和姓名(含Elsa)

```
SELECT Sno, Sname FROM Student AS S
WHERE EXISTS (SELECT * FROM Student AS T
  WHERE T.Sname = 'Elsa' AND T.Sdept = S.Sdept);
```

## 查询执行过程

1. 从Student关系中依次取出每条元组$t$，设$t$的Sdept属性值为$t[Sdept]$
2. 将$t[Sdept]$代入子查询后，执行子查询
3. 若子查询的结果不为空，则将$t$投影到Sno和Sname属性上，并输出投影后的元组

# 用EXISTS实现全称量词∀功能

- SQL不支持全称量词∀ (for all)
- 用EXISTS实现全称量词，因为$\forall x(P(x)) = \nexists x(\neg P(x))$

Course

| Cno |
|---|
| 1002 |
| 2003 |
| 3006 |

## Example (全称量词)

1. 查询选修了全部课程的学生的学号(思考一下用关系代数怎么做?)

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

思路: 设$t \in Student$是某个选修了全部课程的学生的元组，
则$\forall c \in Course$，必$\exists s \in SC$，使$s[Sno] = t[Sno] \wedge s[Cno] = c[Cno]$
这等价于:
$\nexists c \in Course$，使得$\nexists s \in SC$，使$s[Sno] = t[Sno] \wedge s[Cno] = c[Cno]$

# 用EXISTS实现逻辑蕴含→功能

- SQL不支持逻辑蕴含→ (implication)
- 可以用EXISTS实现逻辑蕴含，因为$x \to y = \neg x \vee y$

## Example (全称量词)

1. 查询至少选修了CS-001号学生选修的全部课程的学生的学号

```
SELECT Sno FROM Student WHERE NOT EXISTS (
  SELECT * FROM SC AS SC1
  WHERE SC1.Sno = 'CS-001' AND NOT EXISTS (
    SELECT * FROM SC AS SC2
    WHERE SC2.Sno = Student.Sno AND SC2.Cno = SC1.Cno));
```

# 嵌套查询写法4: 子查询结果作为派生关系

- 派生表(derived table): 将子查询的结果当作关系放在外层查询的FROM子句中使用，称为派生表
- 语法: FROM (子查询)
- 子查询必须是独立子查询
- 派生表必须重命名

## Example (嵌套查询的写法)

1. 查询选修了2门以上课程的学生的学号和选课数

```
SELECT Sno, T.Amt FROM
  (SELECT Sno, COUNT(*) AS Amt
   FROM SC GROUP BY Sno) AS T
WHERE T.Amt >= 2;
```

(第2次出现，上一次是怎么做的? 还能怎么做?)

# 嵌套查询写法5: 在WITH子句中使用子查询

- WITH子句提供了一种定义临时关系的方式
- 语法: WITH 临时关系名(属性列表) AS (子查询)
- WITH子句定义的临时关系只在包含WITH子句的查询中有效

**Example (嵌套查询的写法)**

1. 查询选修了2门以上课程的学生的学号和选课数

```
WITH T AS
  (SELECT Sno, COUNT(*) AS Amt
   FROM SC GROUP BY Sno)
SELECT Sno, Amt FROM T
WHERE Amt >= 2;
```

(第3次出现，上一次是怎么做的? 还能怎么做?)

# 嵌套查询的执行

- 嵌套查询的执行计划通常是由DBMS的查询优化器决定的
- 查询优化器可能会将一个嵌套查询“扁平化”为一个执行效率更高的连接查询
- 在PostgreSQL和openGauss中，可以通过设置配置参数`from_collapse_limit = 1`来强制查询优化器采用SQL语句中明确给出子查询执行顺序

# 嵌套查询习题 I

在PostgreSQL、openGauss或MySQL上创建Products数据库(Database Systems The Complete Book - Exercise 2.4.1)，并使用SQL完成下列查询

用IN

1. Find the manufacturers that sell laptops but not PC's

用比较运算符

1. Find the model numbers of all printers that are cheaper than the printer model 3002
2. Find the PC model with the highest available speed

用EXISTS

1. Find the model numbers of all printers that are cheaper than the printer model 3002
2. Find the PC model with the highest available speed

用派生关系或WITH子句

1. Find the manufacturers of PC's with at least three different speeds

# SQL视图定义和数据更新

讲完SQL查询了，该回头把SQL视图定义和数据更新讲完了

- SQL视图定义 ▸ SQL视图定义
- SQL数据更新 ▸ SQL数据更新

# 总结

1. SQL数据定义
   - 基本数据类型: 数值型、日期时间型、字符串型、枚举型
   - 创建关系模式: CREATE TABLE
   - 修改关系模式: ALTER TABLE
   - 删除关系模式: DROP TABLE
   - 定义视图: CREATE VIEW, ALTER VIEW, DROP VIEW
2. SQL数据更新
   - 插入数据: INSERT
   - 修改数据: UPDATE
   - 删除数据: DELETE
3. SQL数据查询
   - 单关系查询: SELECT ... FROM ... WHERE
   - 集合查询: UNION, INTERSECT, MINUS/EXCEPT
   - 连接查询: INNER JOIN, NATURAL JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN
   - 嵌套查询: IN, 比较运算符, EXISTS, 派生表

# 习题 I

1. 身份证号用CHAR(13)和VARCHAR(13)哪种类型表示更合适?
2. 什么情况下需要使用UNION ALL? 什么情况下使用UNION?
3. MySQL不支持集合交操作INTERSECT，如何实现集合交操作?
4. MySQL不支持集合差操作EXCEPT，如何实现集合差操作?
5. 为什么分组查询语句的目标属性只能包含分组属性和聚集函数?
6. 相关子查询和不相关子查询有何区别?
7. 为什么子查询中通常不使用DISTINCT和ORDER BY?
8. 视图和基本表有什么区别?
9. 判断对错
    1. SQL语句DELETE FROM TABLE R从数据库中删除关系$R$
    2. 将属性声明为PRIMARY KEY和UNIQUE NOT NULL作用是一样的
    3. ORDER BY A, B DESC将查询结果按照属性$A$和$B$的值降序排列
    4. SQL语句SELECT A FROM R与关系代数表达式$\Pi_A(R)$的结果相同
    5. 若关系$R$的属性$A$被声明为UNIQUE，则SQL语句SELECT COUNT(A) FROM R的结果是$|R|$

# 习题 II

10. 设$R(a,b)$和$S(a)$是两个关系，将关系代数表达式$R \div S$用SQL语句表示
11. 在openGauss或MySQL上创建Product数据库(Database Systems The Complete Book - Exercise 2.4.1)，然后使用SQL表达下列数据库查询与更新，并在DBMS上验证
    1. Find the manufacturers that sell laptops but not PC's. (使用集合差运算)
    2. Find the manufacturers that sell laptops but not PC's. (使用含有IN的嵌套查询)
    3. Find the manufacturers that sell laptops but not PC's. (使用含有EXISTS的嵌套查询)
    4. Find the model numbers of all printers that are cheaper than the printer model 3002. (使用内连接查询)
    5. Find the model numbers of all printers that are cheaper than the printer model 3002. (使用含有比较运算符的嵌套查询)
    6. Find the model numbers of all printers that are cheaper than the printer model 3002. (使用含有EXISTS的嵌套查询)

# 习题 III

7. Find the PC model with the highest available speed. (使用外连接查询)
8. Find the PC model with the highest available speed. (使用含有IN的嵌套查询)
9. Find the PC model with the highest available speed. (使用含有=的嵌套查询)
10. Find the PC model with the highest available speed. (使用含有>=的嵌套查询)
11. Find the PC model with the highest available speed. (使用含有EXISTS的嵌套查询)
12. Find the manufacturers of PC's with at least three different speeds. (使用内连接查询)
13. Find the manufacturers of PC's with at least three different speeds. (使用分组查询)
14. Find the manufacturers of PC's with at least three different speeds. (使用派生关系)
15. Decrease the price of all PC's made by maker A by 10%. (使用含有=的更新条件)

# 习题 IV

16. Decrease the price of all PC's made by maker A by 10%. (使用含有IN的更新条件)
17. Decrease the price of all PC's made by maker A by 10%. (使用含有EXISTS的更新条件)

12. 前一题(g)中的查询可以用多种SQL语句表示。尝试从语句的易读性和执行效率两方面对(g)–(k)的SQL语句进行分析和比较。在做效率分析时，我们假定每个关系上只有主索引，而没有其他索引(请自学索引的概念)。

# 资源

- MySQL Community Server下载: https://dev.mysql.com/downloads/mysql/
- MySQL文档: https://dev.mysql.com/doc/
- openGauss下载: https://opengauss.org/zh/download.html
- openGauss文档: https://opengauss.org/zh/docs/2.1.0/docs/Quickstart/Quickstart.html
- 在线SQL练习: https://dbis-uibk.github.io/relax[11]
- W3Schools SQL教程: https://www.w3schools.com/sql/

[11] 该系统功能尚不完善，请在openGauss或MySQL上练习