ch3–5 SQL语言

3.2 SQL数据定义

结构化查询语言Structured Query Language

SQL 语言有以下几个部分：

- **数据定义语言**(Data-Definition Language，DDL)：SQL DDL 提供定义关系模式、删除关系以及修改关系模式的命令。
- **数据操纵语言**(Data-Manipulation Language，DML)：SQL DML 提供从数据库中查询信息，以及在数据库中插入元组、删除元组、修改元组的能力。
- **完整性**(integrity)：SQL DDL 包括定义完整性约束的命令，保存在数据库中的数据必须满足所定义的完整性约束。破坏完整性约束的更新是不允许的。
- **视图定义**(view definition)：SQL DDL 包括定义视图的命令。
- **事务控制**(transaction control)：SQL 包括定义事务的开始和结束的命令。
- **嵌入式 SQL 和动态 SQL**(embedded SQL and dynamic SQL)：嵌入式和动态 SQL 定义 SQL 语句如何嵌入到通用编程语言，如 C、C++ 和 Java 中。
- **授权**(authorization)：SQL DDL 包括定义对关系和视图的访问权限的命令。

创建表

我们用 **create table** 命令定义 SQL 关系。下面的命令在数据库中创建了一个 *department* 关系。

```
create table department
    (dept_name varchar (20),
    building varchar (15),
    budget numeric (12, 2),
    primary key (dept_name));
```

指定主码

主码属性必须唯一且非空

```
primary key (course_id, sec_id, semester, year),
foreign key (course_id) references course);
```

上面为主码和外码

如果不能为空可创建时候后缀not null

常用命令

```
insert into r values ( , , ) ;
--将数据添加到关系中，括弧里的数据遵循在关系模式中列出的顺序
delete from r;--保留关系r，但删除r中的所有元组
drop table r;--不仅删除元组，还删除r的模式
alter table r add A D;
--为已有关系添加属性，A是待添加属性的名字，D为添加属性的域
alter table r drop A;--从关系中去掉属性
```

3.3 SQL基本结构

单关系查询

```
--找出所有教师所在系名
SELECT dept_name
FROM instructor;

--强行删除重复
SELECT DISTINCT dept_name
FROM instructor;

--指明不删除重复
SELECT ALL dept_name
FROM instructor;

--用加减乘除+-*/来运算，显示给每位教师涨10%工资的结果
SELECT salary*1.1
FROM instructor;

--where子句，可以插入使用and，or，not
SELECT name
FROM instructor
WHERE dept_name='comp.sci'AND salary>7000;//注意不是==
```

多关系查询

```
--从多个关系中获取数据的查询
--找出教师姓名、系名、以及系的建筑名称
SELECT name,instructor,dept_name,buiding
FROM instructor,department
WHERE instructor.dept_name=department.dept_name;
```

运算顺序：

1. FROM, 在后面列出的关系产生笛卡尔积
2. WHERE, 在第一步骤的结果上应用WHERE的谓词
3. SELECT, 对步骤二的每个元组，选出SELECT的指定属性

自然连接

自然连接（natural join）运算作用于两个关系，并产生一个关系作为结果。它只考虑在两个关系模式中都出现的属性上取值相同的元组对。（共同属性相同）

```
--列出教师的名字以及他们所讲授课程的名称
SELECT name,title
FROM instructor NATURAL JOIN teaches, course
WHERE teaches.course_id=course.course_id

--不能这么写,这样会把“教的课不在自己的系”的老师忽略掉
SELECT name,title
```

```
FROM instructor NATURAL JOIN teaches NATURAL JOIN course

--using可以指明要自然连接的属性
SELECT name,title
FROM  (instructor NATURAL JOIN teaches) JOIN course USING (course_id);
```

3.4 附加运算

更名运算

```
--更换名字用as语句，可以用在SELECT和FROM里
SELECT T.name, S.name
FROM instructor AS T,teaches AS S
WHERE T.ID=S.ID;
```

下面这种情况也要换名

假设我们希望写出查询："找出满足下面条件的所有教师的姓名，他们的工资至少比 Biology 系某一个教师的工资要高"，我们可以写出这样的 SQL 表达式：

```
select distinct T. name
from instructor as T, instructor as S
where T. salary > S. salary and S. dept_name = 'Biology';
```

注意我们不能使用 *instructor. salary* 这样的写法，因为这样并不清楚到底是希望引用哪一个 *instructor*。

在上述查询中，*T* 和 *S* 可以被认为是 *instructor* 关系的两个拷贝，但更准确地说是被声明为 *instructor* 关系的别名，也就是另外的名字。像 *T* 和 *S* 那样被用来重命名关系的标识符在 SQL 标准中被称作**相关名称**(correlation name)，但通常也被称作**表别名**(table alias)，或者**相关变量**(correlation variable)，或者**元组变量**(tuple variable)。

注意用文字表达上述查询更好的方式是："找出满足下面条件的所有教师的姓名，他们比 Biology 系教师的最低工资要高"。我们早先的表述更符合我们所写的 SQL，但后面的表述更直观，事实上它可

字符串运算

在字符串上可以使用 **like** 操作符来实现模式匹配。我们使用两个特殊的字符来描述模式：

- **百分号(%)**：匹配任意子串。
- **下划线(_)**：匹配任意一个字符。

模式是大小写敏感的，也就是说，大写字符与小写字符不匹配，反之亦然。为了说明模式匹配，考虑下列例子：

- 'Intro%' 匹配任何以"Intro"打头的字符串。
- '% Comp%' 匹配任何包含"Comp"子串的字符串，例如 'Intro. to Computer Science' 和 'Computational Biology'。
- '___' 匹配只含三个字符的字符串。
- '___%' 匹配至少含三个字符的字符串。

在 SQL 中用比较运算符 **like** 来表达模式。考虑查询"找出所在建筑名称中包含子串 'Watson' 的所有系名"，该查询的写法如下：

```
select dept_name
from department
where building like '%Watson%';
```

为使模式中能够包含特殊模式的字符(即% 和_)，SQL 允许定义转义字符。转义字符直接放在特殊字符的前面，表示该特殊字符被当成普通字符。我们在 **like** 比较运算中使用 **escape** 关键词来定义转义字符。为了说明这一用法，考虑以下模式，它使用反斜线(\)作为转义字符：

- **like** 'ab\%cd%' **escape** '\' 匹配所有以"ab%cd"开头的字符串。

*表示所有属性

排列元组的显示次序

```
--最后加一句
ORDER BY a;
--可以使得SELECT的结果
ORDER BY a desc;
--desc表示降序，asc表示升序
```

3.5 集合运算

并运算UNION

并运算自动去除重复，UNION ALL表示保留所有重复

交运算INTERSECT

并运算自动去除重复，INTERSECT ALL表示保留所有重复

为了找出在 2009 年秋季和 2010 年春季同时开课的所有课程的集合，我们可写出：

```
(select course_id
 from section
 where semester = 'Fall' and year = 2009)
intersect
(select course_id
 from section
 where semester = 'Spring' and year = 2010);
```

差运算EXCEPT

差运算自动去除输入中的所有重复

3.6 空值

涉及空值的比较问题更多。例如，考虑比较运算“1 < null”。因为我们不知道空值代表的是什么，所以说上述比较为真可能是错误的。但是说上述比较为假也可能是错误的，如果我们认为比较为假，那么“not (1 < null)”就应该为真，但这是没有意义的。因而 SQL 将涉及空值的任何比较运算的结果视为 unknown(既不是谓词 is null，也不是 is not null，我们在本节的后面介绍这两个谓词)。这创建了除 *true* 和 *false* 之外的第三个逻辑值。

由于在 where 子句的谓词中可以对比较结果使用诸如 and、or 和 not 的布尔运算，所以这些布尔运算的定义也被扩展到可以处理 unknown 值。

- and：*true* and *unknown* 的结果是 *unknown*，*false* and *unknown* 结果是 *false*，*unknown* and *unknown* 的结果是 *unknown*。
- or：*true* or *unknown* 的结果是 *true*，*false* or *unknown* 结果是 *unknown*，*unknown* or *unknown* 结果是 *unknown*。
- not：not *unknown* 的结果是 *unknown*。

可以验证，如果 *r.A* 为空，那么“1 < *r.A*”和“not (1 < *r.A*)”结果都是 unknown。

3.7 聚集函数

以值的一个集合（或多重集）为输入，返回单个值的函数

平均值：AGV

最小值：MIN
最大值：MAX
总和：SUM（输入必须是数字）
计数：COUNT（输入必须是数字）

基本聚集

```
--查询计算机系教师的平均工资
--AS是重新定义名字
SELECT AVG(salary) AS avg_salary
FROM instructor
WHERE dept_name='Comp.Sci'

--找出2010年春季讲授一门课程的教师总数
--DISTINCT表示去除重复，只计算一次
SELECT COUNT (DISTINCT ID)
FROM teaches
WHERE semester='spring' AND year=2010;

--COUNT (*)表示计算一个关系中元组的个数
```

分组聚集GROUP BY

```
--找出每个系的平均工资
SELECT dept_name, AVG(salary)AS avg_salary
FROM instructor
GROUP BY dept_name

--找出每个系在2010年春季有教课的教师人数
SELECT dept_name,COUNT(DISTINCT ID)AS instr_count
FROM instructor NATURAL JOIN teaches
WHERE semester='spring' AND year=2010
GROUP BY dept_name;
```

dept_name	instr_count
Comp. Sci.	3
Finance	1
History	1
Music	1

图 3-16　查询“找出每个系在 2010 年春季学期讲授一门课程的教师人数”的结果关系

注意：任何没有出现在GROUP BY子句中的属性如果出现在SELECT子句中，它只能出现在聚集函数内部。

HAVING子句

针对GROUP BY构成的分组进行的WHERE查询（HAVING是分组后过滤，WHERE是分组前过滤）

```
--找出系平均工资超过42000美元的那些系中教师的平均工资
SELECT dept_name,AVG(salary) AS avg_salary
FROM instructor
GROUP BY dept_name
HAVING AVG(salary)>4200;
```

SQL查询顺序：

1. 根据FROM计算出一个关系
2. WHERE计算出的谓词用在FROM上
3. GROUP BY进行分组
4. HAVING语句应用
5. SELECT

```
--对于2009年的每一个课程，如果该课程至少有两个学生选课，找出该课程段所有学生的总学分（tot_cred的平均值
SELECT course_id,semester,year,sec_id,AVG(tot_cred)
FROM takes NATURAL JOIN student
WHERE year=2009
GROUP BY course_id,semester,year,sec_id
HAVING COUNT(ID)>2;
```

对空值和布尔值的聚集

除了COUNT(*）外的所有聚集函数都忽略输入集合中的空值（count对空值的运算结果为0），其他运算在输入为null的情况下返回null

3.8 嵌套子查询

子查询：嵌套在WHERE中

集合成员资格IN/NOT IN

IN/NOT IN测试集合成员资格，即可以用于子查询，也可以用于枚举集合如（'mary','tom'）

```
--找出选修了ID为10101教师所讲授的课程段
SELECT COUNT (DISTINCT ID)
FROM takes
WHERE (course_id,sec_id,semester,year)
IN
(SELECT course_id,sec_id,semester,year
FROM teaches
WHERE teaches.ID=10101)
```

集合的比较SOME/ALL

SOME表示至少一个，ALL表示全部

```
--找出工资至少比Biology系某一个教师工资高的教师
SELECT name
FROM instructor
WHERE salary>SOME(SELECT salary
FROM instructor
WHERE dept_name='Biology');

--找出平均工资最高的系
SELECT dept_name
FROM instructor
GROUP BY dept_name
HAVING AVG(salary)>=ALL(SELECT AVG(salary)
FROM instructor
GROUP BY dept_name);
```

空关系测试EXISTS

EXISTS在子查询结果非空时返回true

```
--找出在2009和2010年春季学期都开设的所有课程
SELECT course_id
FROM section AS s
WHERE semester='Fall'AND year=2009 AND
EXISTS (SELECT *
FROM section AS T
WHERE semester='Spring'AND year=2010 AND
S.course_id=T.course_id);
```

NOT EXISTS (B EXCEPT A)表示A包含B，也就是说B-A=空集

```
--找出选修了Biology系开设的所有课程的学生
SELECT S.ID,S.name
FROM student AS S
WHERE NOT EXISTS((SELECT course_id
FROM course
WHERE dept_name='Biology')
EXCEPT
(SELECT T.course_id
FROM takes AS T
WHERE S.ID=T.ID));
--上述查询就是：生物系开的课程-该学生修的课=空集
```

重复元组存在性测试UNIQUE

```
--找出所有在2009年最多开设一次的课程
SELECT T.course_id
FROM course AS T
WHERE UNIQUE(SELECT R.course_id
FROM section AS R
WHERE T.course_id=R.course_id AND
R.year=2009);
```

FROM子句中的子查询

```
--找出系平均工资超过4200美元的那些系中教师的平均工资
SELECT dept_name,avg_salary
FROM(SELECT dept_name,AVG(salary)
FROM intructor
GROUP BY dept_name)
WHERE avg_salary>4200;
```

LATERAL

我们注意到在 **from** 子句嵌套的子查询中不能使用来自 **from** 子句其他关系的相关变量。然而 SQL: 2003 允许 **from** 子句中的子查询用关键词 **lateral** 作为前缀，以便访问 **from** 子句中在它前面的表或子查询中的属性。例如，如果我们想打印每位教师的姓名，以及他们的工资和所在系的平均工资，可书写查询如下：

```
select name, salary, avg_salary
from instructor I1, lateral (select avg(salary) as avg_salary
                             from instructor I2
                             where I2.dept_name = I1.dept_name);
```

没有 **lateral** 子句的话，子查询就不能访问来自外层查询的相关变量 *I1*。目前只有少数 SQL 实现支持 **lateral** 子句，比如 IBM DB2。

WITH子句

提供定义临时关系的方法，这个定义只对包含WITH子句的查询有效

```
--找出所有工资总额大于所有系平均工资总额的系
--此处一个WITH引导两个表查询
WITH
  dept_total(dept_name,value)AS
  (SELECT dept_name,SUM(salary)
  FROM instructor
  GROUP BY dept_name),
  dept_total_avg(value)AS
  (SELECT AVG(value)
  FROM dept_total)
SELECT dept_name
FROM dept_total,dept_total_avg
WHERE dept_total.value>=dept_total_avg.value;
```

标量子查询

返回值是一个标量（常量）的查询，如查询所有系以及它们拥有的教师数

```
select dept_name,
    (select count(*)
     from instructor
     where department.dept_name = instructor.dept_name)
     as num_instructors
from department;
```

3.9 数据库的修改

删除DELETE

```
--删除特定元组
DELETE FROM r
WHERE P;
--删除关系r的所有元组
DELETE FROM instructor;
```

插入INSERT

```
INSERT INTO r
VALUES('','','');
--让Music系所有修满144学分的学生成为Music系的老师，工资为18000元
INSERT INTO instructor
  SELECT ID,name,dept_name,18000
  FROM student
  WHERE dept_name='Music' AND tot_cred>144;
```

更新UPDATE

```
--给工资低于7000的教师工资增长1.05倍
UPDATE instructor
SET salary=salary*1.05
WHERE salary < 7000;
```

4.1 连接表达式

连接条件

select *
from *student* **join** *takes* **on** *student. ID = takes. ID*;

上述 **on** 条件表明：如果一个来自 *student* 的元组和一个来自 *takes* 的元组在 *ID* 上的取值相同，那么它们是匹配的。在上例中的连接表达式与连接表达式 *student* **natural join** *takes* 几乎是一样的，因为自然连

第一个ID出现两次，第二个出现一次（一列）

外连接

常规连接叫做内连接，inner可以省略
在结果中创建包含空值元组（例如找到学生和他们修的课，想要把没有选课的同学也显示出来）

```
SELECT *
FROM student NATURAL LEFT OUTER JOIN takes;
```

实际上有三种形式的外连接：

- **左外连接**(left outer join)只保留出现在左外连接运算之前(左边)的关系中的元组。
- **右外连接**(right outer join)只保留出现在右外连接运算之后(右边)的关系中的元组。
- **全外连接**(full outer join)保留出现在两个关系中的元组。

4.2 视图

视图定义

SQL中使用视图

一种“虚关系”，并不预先计算，当用到的时候才计算

我们在 SQL 中用 **create view** 命令定义视图。为了定义视图，我们必须给视图一个名称，并且必须提供计算视图的查询。**create view** 命令的格式为：

create view *v* **as** < query expression > ;

其中 < query expression > 可以是任何合法的查询表达式，*v* 表示视图名。

重新考虑需要访问 *instructor* 关系中除 *salary* 之外的所有数据的职员。这样的职员不应该授予访问 *instructor* 关系的权限(我们将在后面 4.6 节介绍如何进行授权)。相反，可以把视图关系 *faculty* 提供给职员，此视图的定义如下：

```
create view faculty as
select ID, name, dept_name
from instructor;
```

查询每个系中教师工资的总和

视图的属性名可以按下述方式显式指定：

```
create view departments_total_salary(dept_name, total_salary) as
  select dept_name, sum (salary)
  from instructor
  group by dept_name;
```

物化视图

视图更新

4.3 事务

4.4 完整性约束

保证用户对数据库所做的修改不会破坏数据的一致性

- **not null**。
- **unique**。
- **check(<谓词>)**。

NOTNULL约束

禁止插入空值

```
name varchar(20) not null
budget numeric(12, 2) not null
```

UNIQUE约束

unique ($A_{j1}, A_{j2}, \cdots, A_{jm}$)

unique 声明指出属性 $A_{j1}, A_{j2}, \cdots, A_{jm}$ 形成了一个候选码；即在关系中没有两个元组能在所有列出的属性上取值相同。然而候选码属性可以为 *null*，除非它们已被显式地声明为 **not null**。回忆一下，空值不等

CHECK子句

例如在创建关系department的create table命令中的check（budget>0）就是保证取值为正

参照完整性

在一个关系中给定属性集上的取值也在另一个关系的特定属性集中出现。
就是**外码**
例如学生里面有外码专业代号，那么这个表就是参照表（子表或从表）
而专业表就是被参照表（父表或主表）

复杂CHECK条件与断言

对于student的每一个元组，它在属性tot_cred上的取值必须等于该学生所成功修完课程的学分总和。

SQL 中的断言为如下形式：

create assertion < assertion-name > **check** < predicate > ;

图 4-9 给出了我们如何用 SQL 写出第一个约束的示例。由于 SQL 不提供“for all X, $P(X)$”结构（其中 P 是一个谓词），我们只好通过等价的“not exists X such that not $P(X)$”结构来实现此约束，这一结构可以用 SQL 来表示。

```
create assertion credits_earned_constraint check
    (not exists (select ID
      from student
      where tot_cred < > (select sum(credits)
                          from takes natural join course
                          where student. ID = takes. ID
                          and grade is not null and grade < > 'F' ));
```

图 4-9　一个断言的例子

4.5 SQL的数据类型与模式

SQL的日期和时间类型

- **date**：日历日期，包括年（四位）、月和日。
- **time**：一天中的时间，包括小时、分和秒。可以用变量 **time**(p) 来表示秒的小数点后的数字位数（这里默认值为 0）。通过指定 **time with timezone**，还可以把时区信息连同时间一起存储。
- **timestamp**：**date** 和 **time** 的组合。可以用变量 **timestamp**(p) 来表示秒的小数点后的数字位数（这里默认值为 6）。如果指定 **with timezone**，则时区信息也会被存储。

默认值

创建表的时候：
tot_cred numeric（3,0） default 0

用户定义的类型

可以用 **create type** 子句来定义新类型。例如，下面的语句：

```
create type Dollars as numeric (12, 2) final;
create type Pounds as numeric (12, 2) final;
```

CREATETABLE扩展

应用常常要求创建与现有的某个表的模式相同的表。SQL 提供了一个 **create table like** 的扩展来支持这项任务：

```
create table temp_instructor like instructor;
```

5.2 函数和过程

声明和调用SQL函数和过程

表函数

```
select *
from table (instructor_of ('Finance'));
```

这个查询返回'金融'系的所有教师。在上面的简单情况下直接写这个查询而不用以表为值的函数也是很直观的。但通常以表为值的函数可以被看作**带参数的视图**(parameterized view)，它通过允许参数把视图的概念更加一般化。

```
create function instructor_of (dept_name varchar(20))
    returns table (
        ID varchar (5),
        name varchar (20),
        dept_name varchar (20),
        salary numeric (8,2))
return table
    (select ID, name, dept_name, salary
    from instructor
    where instructor.dept_name = instructor_of.dept_name);
```

图 5-6　SQL 中定义的表函数

过程函数

```
——在确保教室能容纳下的前提下注册一个学生
——如果成功注册，返回0，如果超过教室容量则返回 -1
create function registerStudent(
        in s_id varchar(5),
        in s_courseid varchar (8),
        in s_secid varchar (8),
        in s_semester varchar (6),
        in s_year numeric (4, 0),
        out errorMsg varchar(100)
returns integer
begin
    declare currEnrol int;
    select count( * ) into currEnrol
    from takes
    where course_id = s_courseid and sec_id = s_secid
          and semester = s_semester and year = s_year;
    declare limit int;
    select capacity into limit
    from classroom natural join section
    where course_id = s_courseid and sec_id = s_secid
          and semester = s_semester and year = s_year;
    if (currEnrol < limit)
      begin
        insert into takes values
            (s_id, s_courseid, s_secid, s_semester, s_year, null);
        return(0);
      end
    ——否则，已经达到课程容量上限
    set errorMsg = 'Enrollment limit reached for course ' || s_courseid
        || ' section ' || s_secid;
    return( -1);
end;
```

图 5-7　学生注册课程的过程

5.3 触发器

触发器(trigger)是一条语句，当对数据库作修改时，它自动被系统执行。要设置触发器机制，必须满足两个要求：

- 指明什么条件下执行触发器。它被分解为一个引起触发器被检测的事件和一个触发器执行必须满足的条件。
- 指明触发器执行时的动作。

一旦我们把一个触发器输入数据库，只要指定的事件发生，相应的条件满足，数据库系统就有责任去执行它。

示例一

设计一个触发器当进行Teacher表更新元组时，使其工资只能升不能降

```
create trigger teacher_chgsal before update of salary
    on teacher
    referencing new x, old y
    for each row when (x.salary < y.salary)
        begin
    raise_application_error(-20003, 'invalid salary on update');
            //此条语句为Oracle的错误处理函数
        end;
```

before：在update之前触发

示例二

假设student(S#, Sname, SumCourse), SumCourse为该同学已学习课程的门数，初始值为0，以后每选修一门都要对其增1。设计一个触发器自动完成上述功能。

```
create trigger sumc after insert on sc
    referencing new row newi
    for each row
        begin
            update student set SumCourse = SumCourse + 1
            where S# = :newi.S# ;
        end;
```

where是指学生的学号等于当前新增表中的学生学号（newi的学生学号）

*SQL*语句中“<>”含义 在*SQL*语句中,“<>”代表的是不等于,和 ” != ”是一个意思

下面的触发器实现了“已修并且取得及格成绩的课程不可被新成绩覆盖”

tot_cred 属性，使其保持实时更新。只有当属性 *grade* 从空值或者 'F' 被更新为代表课程已经完成的具体分数时，触发器才被激发。除了 *nrow* 的使用，update 语句都属于标准的 SQL 语法。

181
182

```
create trigger credits_earned after update of takes on (grade)
    referencing new row as nrow
    referencing old row as orow
    for each row
    when nrow.grade <> 'F' and nrow.grade is not null
        and (orow.grade = 'F' or orow.grade is null)
    begin atomic
        update student
        set tot_cred = tot_cred +
            (select credits
             from course
             where course.course_id = nrow.course_id)
        where student.id = nrow.id;
    end;
```

新成绩不为空格&不为空
又(旧为不及格/空)

图 5-9　使用触发器来维护 *credits_earned* 值

当插入值为’ ’时，自动更新为null的触发器

```
create trigger setnull before update of takes
    referencing new row as nrow
    for each row
    when (nrow.grade = ' ')
    begin atomic
        set nrow.grade = null;
    end;
```

图 5-10　使用 set 来更改插入值的例子

何时不用触发器

5.22　一个触发器的执行可能会引发另一个被触发的动作。大多数数据库系统都设置了嵌套的深度限制。解释为什么它们要设置这样的限制。

因为程序员在写触发器时，运行期间一个触发器的错误不仅仅会导致该触发器语句失败，而且该触发器的动作还可能影响另一个触发器，甚至于导致一个无线的触发连。因此有些数据库中会限制这种触发器链的长度，把超过这种长度的触发器看成一种错误。从而可以减少开销，提高执行效率。