数据库系统概念-中

第三章 SQL

3.1 SQL查询语言概览

- SQL语言组成部分
  - 数据定义语言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DML： SQL DML提供从数据库中查询信息，以及在数据库中插入元祖、删除元祖、修改元祖的能力。

- 完整性 `Integrity`：`SQL DDL` 包括定义完整性约束的命令，保存在数据库中数据必须满足定义的完整性约束，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。
- 事物控制 `Transaction Control`：`SQL` 包括定义事物的开始和结束的标志。
- 嵌入式 `SQL` 的动态 `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。

3.2 `SQL` 数据定义

数据库中的关系集合必须由数据定义语言指定给系统。`SQL` 的 `DDL` 既能定义一组关系，又能定义每个关系的信息，包括：

- 关系的模式；
- 属性的取值类型；
- 完整性约束；
- 关系维护的索引集合；
- 关系的安全性和权限信息；
- 关系在磁盘上的物理存储结构；

3.2.1 基本类型

`SQL` 标准支持多种固有类型，包括：

- `char`(n)：固定长度的字符串，用户指定长度n，也可以使用全称character;
- `varchar`(n)：可变长度字符串，用户指定最大长度为n，等价于全称character varying;
- `int`：整数类型（和机器相关的整数的有限子集），等价于全称integer;
- `smalint`：小整数类型（和机器相关的整数类型的子集）；
- `numeric`(p,d)：定点数，精度由用户指定。p：代表数含p位数字（加上符号位），其中d位数字在小数点右边;
- `real, double precision`：浮点数与双精度浮点数，精度与机器有关;
- `flaot`：精度至少为n位的浮点数;

每种类型都可能包含一个被称作空值的特殊值。空值表示一个缺失的值，该值可能存在但并不为人所知，或者可能根本不存在。在可能的情况下，一般禁止加入空值。

**注意：`char`与`charvar`的区别；

3.2.2 基本模式定义

- **create table**命令定义relation：
  - create table department (dept_name `varchar`(20),building `varchar`(15),budget `numeric`(12,2),*primary key*(dept_name));
  - 通用形式：

```
create table table_name(
    A_1 D_1 not null,
    A_2 D_2,
    ...,
    A_n D_n,
    <完整性约束_1>,
    ...,
    <完整性约束_1>
);
```

    - SQL部分完整性约束：
        - primary key (A_1, A_2,... , A_n): 指定关系主码，主码属性必须非空(不存在一个元祖在主码属性上取空值)且唯一(不存在两个元祖在所有主码属性上取值相同)
        - foreign key (A_1, A_2,..., A_n) references r:
            - 表示关系中任何元祖在属性(A_1, A_2, ..., A_n)上的取值必须对应于关系r中某元祖在主码属性上的取值；
        - not null: 属性上的 not null 约束表明在该属性上不允许空值；
- insert 命令: 添加元祖
    - insert r values (a_1, a_2, ..., a_n);
    - insert r(A_1, A_2, ..., A_n) values (a_1, a_2, ..., a_n);
- delete 命令: 删除元祖
- drop table 命令: 从数据库中删除关于被去掉关系的所有信息
    - The difference between "delete table r" and "drop table r":
        - "delete table r" 保留关系r, 但删除r的所有元祖；
        - "drop table r" 不仅删除r的所有元祖, 还删除r的模式；
- alter table 命令: 为已有关系增加或去掉属性
    - alter table r add A D;(r: relation A: attribute D: Domain)
    - alter table r drop A;(Not supported by many databases)

3.3 SQL查询的基本结构

- SQL查询的基本结构由三个子句构成：select、from、where
    - select子句用于列出查询结果中所需要的属性；
    - from子句是一个查询求值中需要访问的关系列表(查询的输入)；
    - where子句是一个作用在from子句中关系的属性上的谓词；

3.3.1 单关系查询

- select A from r:
    - To force the elimination of duplicates, insert the keyword **distinct** after select:
        - select distinct A from r;
    - The keyword **all** specifies that duplicates not be removed:
        - select all A from r;
- The **select** clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples:
    - select ID, name, dept_name, salary*1.1 from instructor;
- An **asterisk** in the select clause denotes “all attributes”:
    - select * from instructor;
- **where** 子句允许我们选择那些在from子句的结果关系中满足特定谓词的元祖：
    - SQL允许where子句中使用逻辑连词and、or和not；
    - 逻辑连词的运算对象可以是包含比较运算符、算术表达式以及特殊类型；

3.3.2 多关系查询

- 典型的SQL查询形式：
    - ```
      select A_1, A_2, ..., A_n
      from r_1, r_2, r_3
      where P;
      ```
      ```

- **理解顺序：from——where——select**；
  - *事实上，`SQL`的实际实现不会执行这种形式的查询，它会通过（尽可能）只产生满足where子句谓词的笛卡尔积元素来进行执行优化*；
- from子句定义了一个在该子句中所列出关系上的笛卡尔积：

```
for each tuple_1 in r_1:
    for each tuple_2 in r_2:
        ...
            for each tuple_n in r_n:
                insert (tuple_1, tuple_2, ..., tuple_n) in r;
```

  - 通过笛卡尔积得到的关系一般过于庞大，可以通过where子句限制：
    - select name,course_id from instructor, teaches where instructor. id = teaches. id

3.3.3 自然连接

- select name,course_id from instructor, teaches where instructor. id = teaches. id ，通过`SQL`的自然连接运算可以简洁的写作：
  - select name, course_id from instructor **natural join** teaches;
- 一般形式：
  - ```
    select A_1, A_2, ..., A_n
    from r_1 natural join r_2 natural join ... natural r_n
    ```

    更为一般的，from子句可以为"from E_1, E_2, ..., E_n", 其中每个E_i可以是单一关系或一个包含自然连接的表达式；
  - Example：列出教师的名字及其所讲授课程的名称：

    ```
    select name，title
    from instructor natural join teaches，course
    where teaches.course_id=course.course_id
    ```

    查询结果与以下结果不同，以下查询会忽略“教师讲授的可不是其所在系的课程”的对(teaches. dept_name与course. dept_name造成)：

    ```
    select name，title
    from instructor natural join teaches natural join course
    ```

    通过`SQL`的自然连接的构造形式（允许用户指定需要哪些列相等）改为(**join...using**)：

    ```
    select name，join
    from (instructor natural join teaches) join course using(course_id)
    ```

3.4 附加的基本运算

3.4.1 更名运算

- old-name **as** new-name:
  - 重命名属性：
    ```

```
select name as instructor_name,course_id
from instructor,teaches
where instructor.ID=teaches.ID;
```

- 重命名关系及其原因：
  - 为了使用方便

```
select T.name, S.course
from instructor as T, teaches as S
where T.ID=S.ID;
```

  - 为了适用于需要比较同一个关系中的元祖的情况：这种情况下，需要把一个关系同它自身进行笛卡尔积运算，如果不重命名的话，就不可能把一个元祖与其它元祖区分开

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

  - **说明**：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`，或元祖变量`tuple variable`；

3.4.2 字符串运算

- `SQL`使用一对单引号来标示；如果单引号是字符串的组成成分，就用双引号标示；
- `SQL`标准中，字符串的相等运算值大小写敏感的（`MySQL`与`SQL Server`不区分）；
- 字符串函数：串联，提取子串，计算字符长度，大小写转换，去掉字符串后面的空格（字符串函数集与名称具体参阅使用的数据库系统）；
- 模式匹配：
  - 百分号`percent`(%)：匹配任意子串；
  - 下划线`underscore`(_)：匹配任意一个字符；
  - 模式是大小写敏感的；
  - Example：
    - 'Intro%' 匹配任何以"Intro"打头的字符串；
    - '%Comp%' 匹配包含"Comp"子串的字符串；
    - '___' 匹配只包含三个字符的字符串；
    - '___%' 匹配至少包含三个字符的字符串；
  - `SQL`：使用**like**表达模式
    - select dept_name from department where building like '%Watson';
    - `SQL`允许定义**转义字符**(使用**escape**)：
      - `like 'ab\%c%' escape '\'` 匹配所有以"ab%c"开头的字符串；
      - `'like' 'ab\\c%' escape '\'` 匹配所有以"ab\c"开头的字符串；
    - `SQL`允许使用**not like**比较运算符搜寻不匹配项

3.4.3 排列元祖的显示次序

- **order by**：
  - select name from instructor where dept_name='Physics' order by name;(默认使用升序)

- `desc` 降序；`asc` 升序；
- 排序可在多个属性进行：
  - `select * from instructor order by salary desc, name asc;`

3.4.5 where子句谓词

- between ... and ...
  - select name from instructor where salary between 9000 and 10000;
- not between
- ```
  select name,  course_id
  from instructor, teaches
  where instructor.ID = teaches.ID and depat_name = "Biology";
  ```

  ```
  select name, course_id
  from instructor, teaches
  where (instructor.ID, dept_name) = (teaches.ID, 'Biology')
  ```

3.5 集合运算

`SQL`作用在关系上的**union、intersect、except**运算对应于数学集合论中的∪、∩、–运算。

3.5.1 并运算

- 找出2009年秋季开课或在2010年春季学期开课或均开课的所有课程

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

与**select**子句不同，**union**运算自动去除重复；如果我们想保留所有重复，就必须用**union all**代替**union**；

3.5.2 交运算

- 2009年秋季和2010年春季学期同时开课的所有课程

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

与**select**子句不同，**intersect**运算自动去除重复；如果我们想保留所有重复，就必须用**intersect all**代替**intersect**；
```

3.5.3 差运算

- 找出2009年秋季开课，而不在2010年春季学期开课的所有课程

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

except运算从其第一个输入中输出所有不出现在第二个输入中的元祖，也即它执行集合差操作，此运算在执行集合差操作之前自动去除输入中的重复；如果我们想保留所有重复，就必须用**except all**代替**except**；

3.6 空值

- 空值给关系运算带来了特殊的问题，包括算术运算、比较运算和集合运算；
- 如果算术表达式的任一输入为空(null)则该算术表达式(涉及诸如+、-、*或/)结果为空；
- `SQL`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**)；
- 由于where子句的谓词可以对比较结果使用诸如and、or和not的布尔运算，所以这些布尔值运算的定义也被扩展到可以处理unknown值：(相关定义见Chapter 2)；
    - *如果where子句谓词对一个元祖计算出false或unknown，那么该元祖不能被加入到结果集中*;
- `SQL`使用关键词**null(is null\is not null)**测试空值；某些`SQL`实现允许使用子句**is unknown\is not unknown**来测试一个表达式的结果是否为unknown，而不是true或false；
- 如果元祖在所有属性上的取值相等，那么它们就会被当作相同元祖，即使某些值为空；

3.7 聚集函数

> 聚集函数是以值的一个集合(集或多重集)为输入、返回单个值的函数。`SQL`提供了五个固有聚集函数：
>
> - 平均值：avg
> - 最小值：min
> - 最大值：max
> - 总和：sum
> - 计数：count
>
> sum和avg的输入必须是数字集，但其它运算符还可以作用在非数字数据类型的集合上；

- 基本聚集
    - 查找"Computer science"系的平均工资

```
select avg(salary) as avg_salary
from instructor
where dept_name='Comp.Sci.';
```

    - 计算平均值时注意保留重复元祖；
    - 某些情况需要删除重复元祖以进行聚集，删除重复元祖可以使用关键词**distinct**：

```sql
select count(distinct ID)
from teaches
where semeter='Spring' and year=2010;
```

  - 计算关系中元祖的个数：count(*)

- 分组聚集
  - group by：子句给出的一个或多个属性是用来构造分组的，group by 子句中的所有属性上取值相同的元祖将被分在同一组；
  - Example：查找每个系的平均工资：

```sql
select dept_name,avg(salary) as avg_salary
from instructor
group by dept_name
```

  - *当 SQL 查询使用分组时，注意保证出现在select语句中但没有被聚集的属性只能是出现在group by子句中的那些属性(即，任何没有出现在group by子句中的属性如果出现在select子句的话，它只能出现在聚集函数内部，否则这样的查询是错误的)*；
- having子句
  - having：**针对分组限定条件**
  - Example：找出系平均工资超过42000美元的那些系中教师的平均工资

```sql
select dept_name,avg(salary) as avg_salary
from instructor
group by dept_name
having avg(salary)>42000
```

    - *任何出现在having语句中但没有被聚集的属性只能是出现在group by子句中的那些属性*
- 包含聚集、group by或having子句的查询的含义可通过下述操作序列来理解：
  - 首先，from子句计算出关系；
  - 如果出现where子句，则where子句筛选关系元祖；
  - 如果出现group by子句，对经筛选的元祖分组；
  - 如果出现having子句，筛选分组；
  - 最后，执行select；
- 对空值和布尔值的聚集：
  - 假设instructor关系中某些元祖在salary上取空值，考虑以下查询:
    - select sum(salary) from instructor

      > 故而求和的值存在空值，SQL标准不认为为0，而忽略空值；
  - 聚集函数根据以下原则处理空值：除了count(*)外所有的聚集函数都忽略输入集合中的空值；
    - 由于空值被忽略，可能造成参加聚集函数运算的输入集合为空，故规定空集count运算值为0，其它聚集运算在输入为空集时返回空值；
  - boolean：
    - 使用some与every聚集函数；

3.8 嵌套子查询

> `SQL`提供嵌套子查询机制：子查询是嵌套在另一个查询中的select-from-where表达式。子查询嵌套在where子句中，通常用于对集合的成员资格、集合的比较及集合的技术进行检查；子查询也可以嵌套from子句中；除此之外，还有一类子查询就是标量子查询。

- 集合成员资格
  - `SQL`允许测试元祖在关系中的成员资格：
    - 连接词**in**测试元祖是否是集合中的成员；
    - 连接词**not in**测试元祖是否不是集合中的成员；
  - Example：
    - 找出2009年秋季和2010年春季学期同时开课的所有课程(交运算可)

```
select distinct course_id
from section
where semester='Fall' and year=2009 and
    course_id in (select course_id
                  from section
                where semeter='Spring' and year=2010)
```

    - 找出2009年秋季开课，而不在2010年春季学期开课的所有课程(差运算可)

```
select distinct course_id
from section
where semester='Fall' and year=2009 and
course_id not in (select course_id
                      from section
                    where semeter='Spring' and year=2010)
```

    - in与not in可用于枚举集合:
      - 找出既不叫"Mozart"，也不叫"Einstein"的教师的姓名；

```
select distinct name
from instructor
where name not in ('Mozart', 'Einstein')
```

    - 找出选修了ID为'10101'的教师所讲授的课程段的学生总数：

```
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、<some、<=some、>=some、=some(等价于in)、<>(!=)some(不等价于not in)
  - >all、<all、<=all、>=all、=all(不等价于in)、<>(!=)all(等价于not in)
  - Example：
    - 找出工资至少比Biology系某一教师的工资高的所有教师姓名(更名运算也可)

```
select name
from instructor
where salary>some(select salary
                  from instructor
                  where dept_name = 'Biology');
```

        - 找出工资比Biology系任一教师的工资都高的所有教师姓名

```
select name
from instructor
where salary>all(select salary
                  from instructor
                  where dept_name = 'Biology');
```

        - 找出平均工资最高的系

```
select 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：**exist\not exist**
    - exists 结构在作为参数的子查询非空时返回true值：
        - Example：找出2009年秋季和2010年春季学期均开课的所有课程(in 可)

```
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
)
```

    - 来自外层查询的一个相关名称可以用在where子句的子查询中，**使用了来自外层查询相关名称的子查询被称作相关子查询(correlated sub_query)**；
    - not exists 结构测试子查询结果集中是否不存在元祖：
        - 包含操作模拟：A包含B ------ not exists (B except A);
        - Example：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
    )
)
```

- 重复元祖存在性测试
  - unique\not unique 测试子查询的结果是否存在重复元祖
    - Example：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)
);//unique谓词在空集上计算出真值
```

```
select T.course_id
from course as T
where 1>=(
    select count(R.course_id)
    from section as R
    where T.course_id = R.course_id and R.year = 2009
);
```

    - Example：找出所有在2009年最少开设二次的课程

```
select T.course_id
from course as T
where not unique (select R.course_id
              from section as R
              where T.course_id = R.course_id and R.year = 2009)
);//unique谓词在空集上计算出真值
```

- from子句中的子查询
  - 由于任何select-from-where表达式返回的结果都是关系，故可被插入到另一个select-from-where中任何关系可以出现的位置；
  - Example：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
from (
    (select dept_name,avg(salary)
    from instructor
    group by dept_name)
    as dept_avg(dept_name,avg_salary)
)
where avg_salary>42000;
```

    - 上述查询使用as子句给子查询的结果关系起名，并对属性进行重命名
  - Example：找出所有系中工资总额最大的系的工资总额

```
select max(tot_salary)
from (select dept_name,sum(salary)
      from instructor
      group by dept_name) as dept_total(dept_name,tot_salary);
```

- with子句
  - 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
  - Example：
    - 找出具有最大运算值的系：(以下查询定义了临时关系max_budget)

```
with max_budget(value) as
(select max(budget) from department)
select budget
from department,max_budget
where department.budget=max_budget.value;
```

  - 找出系工资总额大于所有系平均工资总额的系：

```
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
  - `SQL`允许只返回包含单个属性的单个元祖的子查询出现在返回单个值的表达式的能够出现的任何地方，这样的子查询称为**标量子查询(scalar sub_query)**；
  - Example: 列出所有的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

3.9.1 删除

- 删除请求的表达与查询类似

```
delete from r
where P;
```

其中P代表一个谓词，r代表一个关系，delete语句首先从r中找出所有使P(t)为真的元祖t，然后把它们从r中删除；若省略where子句，则r的所有元祖将被删除。

- delete命令只能作用于一个关系：若想从多个关系删除元祖，必须要在每个关系上使用一条delete命令；where子句谓词可以与select命令的where子句谓词一样复杂；
- Example：
  - delete from instructor where dept_name ='Finance';
  - delete from instructor where salary between 13000 and 15000;
- delete请求可以包含嵌套的select；

3.9.2 插入

- 插入数据的方式：
  - 指定待插入的元祖；
  - 写一条查询语句来生成待插入的元祖集合；
- 待插入元祖的属性值必须在相应属性的域，且分量数必须正确；
- Example:
  - insert into course *values* ('CS-437', 'Database System', 'Comp Sci', 4);(元祖属性值的插入顺序和关系模式的属性序列排序一致)；
  - insert into course(title, course_id, credits, dept_name) values course('Database System', 'CS-437', 4, 'Comp Sci');(指定属性顺序)
  - 在查询结果基础上插入元祖：
    - 执行插入之前执行完select语句；
    - ```
      insert into instructor
          select ID,name, dept_name,18000
          from student
          where dept_name = 'Music' and tot_cred>144;
      ```
  - insert into student values ('3003', 'Green', 'Finance', null);
- bulk loader;

3.9.3 更新

- update：在不改变整个元祖的情况下改变其部分属性的值；
- Example：
      ```

- update instructor set salary = salary*1.05;
- update instructor set salary = salary*1.05 where salary<7000;(update语句的where子句可以包含select语句的where子句的任何合法结构);
- update instructor set salary = salary*1.05 where salary<(select avg(salary) from instructor);
- 工资超过10000的教师工资涨3%，其余教师工资涨5%(注意update语句顺序, 避免涨8.15%的情况发生)

```
update instructor
set salary = salary * 1.03
where salary > 10000;
update instructor
set salary = salary * 1.05
where salary <= 10000;
```

- **case**:
  - 格式:

```
case
    when pred_1 then result_1
    when pred_2 then result_2
    ...
    when pred_n then result_n
    else result_0
end
```

  - Example:工资超过10000的教师工资涨3%，其余教师工资涨5%

```
update instructor
set salary =
    case
        when salart <=10000 than salart*1.05
        else salary*1.03
    end
```

-

```
update student S
set tot_cred = (
    select sum(credit)
    from takes natural join course
    where S.ID=takes.ID and
    takes.grade<>'F' and takes.grade is not null
);
```

End

第四章 高级 SQL

4.1 连接表达式

4.1.1 连接条件

- **natural**
- **using**：一种自然连接(不保留未匹配元祖的连接，也称作**内连接(inner join)**)的形式，字需要在指定属性上的取值匹配
- **on**：允许在参与连接的关系上设置通用的谓词，谓词写法与where子句类似
  - ```
    select *
    from student join takes on student.ID = takes.ID;
    ```

    注：查询结果几乎与**natural join**一致，但是需要注意查询结果ID出现两次，与自然连接有别；结果与where替换掉on的查询结果等价。
  - ```
    select student.ID as ID, name, dept_name, tot_cred, course_id,
            sec_id, semester, year, grade
    from student join takes on student.ID = takes.ID;
    ```

    注：查询结果ID出现一次。
  - on条件可以表示任何 SQL 谓词，使用on可以表示比自然连接更为丰富的连接条件

4.1.2 外连接

- **左外连接**：只保留出现在左外连接运算之前(即左边)的关系中的元祖;
  - 如何操作?
    - 计算自然连接(内连接)的结果；
    - 对于内连接左侧关系的任意一个与右侧关系任意元祖不匹配的元祖t，结果添加元祖r
      - 元祖r从左侧关系得到的属性被赋为t中的值；
      - r的其它属性被赋为空；
  - Example：
    - ```
      select *
      from student natural left outer join takes;
      ```

      ```
      select *
      from student left outer join takes on student.ID = takes.ID
      ```

添加"where course_id is null"可以查询"一门课程都没选修的学生";

- on语句是外连接声明的一部分，结果与where替换掉on的查询结果不等价，此结果包含未选课程的学生；

* **右外连接**：只保留出现在右外连接运算之后(即右边)的关系中的元祖;

  * 与左外连接类似

  * Example(结果与左外连接例子结果一致，只是属性顺序存在差异)：

    * ```mysql
      select *
      from takes natural right outer join student;
      ```

- **全外连接**：保留出现在两个关系的元祖;
    - ```
      select *
      from (select *
            from student
            where dept_name = 'Comp.Sci')
            natural full outer join
            (select *
            from takes
            where semester = 'Spring' and year = 2009
            )
      ```

> 为把常规连接和外连接区分开来，`SQL`中常规连接称作内连接(使用"inner join"，其中inner可选，不使用outer则默认为inner)

4.2 视图

> 目前的所有例子一直都在逻辑模型层操作(即，假定了给定的集合中的关系都是实际存储在数据库中的)，然而让所有用户看到整个逻辑模型并不合适(例如，出于安全考虑，可能需要向用户隐藏特定的数据，或是希望创建一个比逻辑模型更符合特定用户直觉的个人化的关系集合)。

- 虚关系：
    - 并不预先计算并存储关系，而是在使用虚关系是才通过执行查询被计算出来 (概念上包含查询的结果)
    - 任何不是逻辑模型的一部分，但作为虚关系对用户可见的关系称为视图(**view**)
- 使用**create view**命令定义视图：
    - create view v as
    - 访问除salary之外所有数据的职员：

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

    - ```
      create view all_customer as
      (select branch_name, customer_name
      from depositor, account
      where depositor.account_number = account.account_number )
      union
      (select branch_name, customer_name
      from borrower, loan
      where borrower.loan_number = loan.loan_number)
      ```

- 一旦定义了一个视图，就可以使用视图名来指代该视图生成的需关系 (视图名可以出现在关系名可以出现的任何地方) (Once a view is defined, the view name can be used to refer to the virtual relation that the view generates)
- 视图的属性名指定：
    - create view view_name(A_1,A_2,...,A_n) as
- 视图实现：

    - 当定义一个视图是，数据库系统存储视图的定义本身，而不存储定义该视图的查询表达式的执行结果；
    - 一旦视图出现在查询中，它就被已存储的查询表达式代替，故无论何时执行查询，视图关系读背重新计算；
- One view may be used in the expression defining another view.
    - A view relation v_1 is said to **depend directly** on a view relation v_2 if v_2 is used in the expression defining v_1.
    - A view relation v_1 is said to **depend on** view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2.
    - A view relation *v* is said to be **recursive** if it depends on itself
- View Expansion
- 物化视图(materialized view)***：
    - 特定数据库系统允许存储视图关系，但它们保证：如果用于定义视图的实际关系改变，视图也跟着修改，这样的视图被称作物化视图；
    - 保持物化视图一直正在最新状态的过程称为物化视图维护(materialized view maintenance)
    - 频繁使用视图的应用将会从视图的物化中获益，那些需要快速响应基于大关系上聚集计算的特定查询也会从创建与查询相对应的物化视图中受益良多 (避免了读取大的底层关系)
- 视图更新
    - 使用视图来表达数据库修改(更新、插入或删除)必须被翻译为对数据库逻辑模型中实际关系的修改：

      例如，如下插入必须表示为对instructor关系的插入

      ```
      insert into faculty values('30765','Green','Music')
      ```

      如何处理：

      拒绝插入，返回错误信息；

      插入('30765', 'Green', 'Music', null)到instructor中

      又如，

      ```
      create view instructor_info as
      select ID,name,building
      from instructor,department
      where instructor.dept_name = department.dept_name

      insert into instructor_info values ('69987','White','Taylar')
      ```

      假设没有标识为69987的教师与位于Taylor楼的系，则向instructor与department关系插入元祖的唯一可能方法为：分别插入('69987', 'White', null, null)、('null', Taylor, null)，然而这一更新并未产生出所需结果。因此，通过利用空值来更新instructor与department以得到instructor所需更新不可行。
    - 一般来说，如果定义视图的查询对下列条件都能满足，我们称 `SQL` 视图是可更新的:
        - from 子句只有一个数据库关系；
        - select 子句只包含关系的属性名，不包含任何表达式、聚集或distinct声明；
- 任何没有出现在select子句中的属性可以取空值：即这些属性上没有not null约束，也不构成主码的一部分；
    - 查询中不含有 group by 或 having 子句；

      注：即使可更新，依然存在某些问题；**with check option** 子句

4.3 事务

- 事务`Transaction`由查询和(或)跟新语句的序列组成，`SQL`标准规定当一条`SQL`语句被执行，就隐式的开始了一个事务。下列`SQL`语句之一会结束一个事务：
  - **Commit work**：提交当前事务(把当前事务所做的更新在数据库中持久保存)，事务提交后，一个新的事务自动开始；
  - **Rollback work**：回滚当前事务(撤销当前事务中所有`SQL`语句对数据库的更新)，数据库恢复到执行该事务第一条语句之前的状态；

    注：

    1. work 均是可选的
    2. 事务提交类比于编辑文档的变化存盘，而回滚这是不保存变化退出编辑
    3. 一旦某市务执行**commit work**，就不能使用**rollback work**撤销
- 当事务执行过程中检测到错误时，事务回滚是有用的；
- 数据库系统保证在发生诸如某条`SQL`语句错误、断电、系统崩溃的情况下，如果一个事务还没有完成commit work，其影响将被回滚；
- 一个事务或者在所有步骤提交后提交其行为，或者在不能成功完成其所有动作的情况下回滚其所有动作，通过这种方式数据库提供了对事务具有原子性的抽象；
- **begin atomic … end**：所有在关键字之间的语句构成了一个单一的事务；

4.4 完整性约束

> 完整性约束保证授权用户对数据库所做的修改不会破坏数据的一致性，故而完整性约束防止的是对数据的意外破坏。
>
> 完整性约束通常被看成是数据库模式设计过程的一部分，它作为用于创建关系的create table命令的一部分被声明，而可以通过使用alter table table-name add constraint命令施加到已有关系上，其中constraint可以是关系上的任意约束(执行上述命令时，系统首先保证关系满足指定的约束，满足则施加到关系上，不满足则拒绝执行)。

- 单个关系上的约束：
  - **not null**
    - 空值是所有域的成员，在默认情况下是`SQL`中每个属性的合法值，然而空值与一些属性来首并不合适；
    - **not null** 声明禁止在该属性上插入空值；
    - 主码不必显示的声明为**not null**;
  - **unique**
    - unique(A_1, A_2, …, A_m)；
    - unique声明指出属性A_1, A_2, …, A_m形成一个候选码，即在关系中没有两个元祖能在所有列出的属性上取值相同 (候选码属性可为null，除非被显式声明为not null)；
  - **check(<谓词>)**
    - check(P)子句指定一个谓词P，关系中每个元祖都必须满足谓词P；
    - 通常用check子句来保证属性值满足指定的条件，实际上创建了一个强大的类型系统；
    - 根据`SQL`标准，check子句的谓词可以是包括子查询在内的任意谓词；

- 参照完整性约束`Referancial integrity constraint`或子集依赖`subset dependency`：
  - 定义：保证在一个关系中给定属性集上的取值也在另一关系的特定属性集的取值中现的情况；
  - 更一般地：

    令关系r_1, r_2的属性集分别为R_1和R_2, 主码分别为K_1和K_2, 如果要求对r_2中任意的元祖t_2, 均存在r_1中的元祖t_1使得$t_1.K_1 = t_2.a$, 称R_2的子集a为参照关系r_1中K_1的外码；(为使参照完整性约束有意义，a和K_1必须是相容的属性集)
  - **references**子句：显示指定被参照关系的属性列表(必须为被参照关系的候选码：**unique**或**primary key**约束) 的**references**子句
  - ```
    create table course
    (...
    foreign key (dept_name) references department
        on delete cascade
        on update cascade
    ...);
    ```
  - 注：1.由于有了与外码声明相关联的`on delete casade`，如果删除department中的元祖而导致违反了参照完整性约束，删除虽不被系统拒绝，但是会对course关系作“级联”删除，即删除参照了被删除关系的元祖；

    2.如果更新被参照字段违反了约束，则更新也不被拒绝，而是将course中参照的元祖的dept_name字段也改为新值

- 断言`Assertion`
  - `SQL`中断言为如下形式：
    - create assertion check
  - 由于`SQL`不提供“for all X, P(X)”结构，故通过等价的“not exists X such that not P(X)”实现
  - 约束创建例子:

    ```
    create assertion credits_earned_constraint ckeck
    (not exists (select ID
                 from student
                 where tot_cred<>(select sum(credits)
                                  from takes natural join course)
    )
    )
    ```
  - 当创建断言时，系统检测其有效性：若有效，则以后只有不破坏断言的数据库修改才被允许；

> 注：
>
> 目前，没有一个广泛使用的数据库系统支持check子句的谓词使用子查询或create assertion

4.5 `SQL`的数据类型与模式

- 日期和时间类型
  - `date`：日历日期，包括年、月、日`yyyy-mm-dd`
    ```

- `time`：时间，包括小时、分钟、秒 `hh:mm:ss`
  - 可以使用**time(p)**表示秒的小数点后的数字位数(默认为0)；
  - 通过指定**time with timezone**，可以把时区信息连同时间一起存储；
- `timestamp`：时间戳，date 和 time 的组合
  - 可以使用`timestamp(p)`表示秒的小数点的数字位数(默认为6)；
  - 指定`with time zone`，时区信息也会被存储；
- **cast e as t**：使用该形式可以将一个字符串(或字符串表达式)e转换为类型t(date\time与时间戳)，注意e符合正确的格式；
- **extract** (field from d) :
  - 从date或time类型的值d中提取单独的域(year\mouth\day\hour\minute\second等)
- current_date : 返回当前时间(带时区)
- `localtime`：返回当前的本地时间(不带时区)
- `current_tiemstamp`\`localtimestamp`.
- **interval**: 时间间隔，允许在日期、时间和时间间隔上进行计算

- 默认值：

  `SQL`允许为属性指定任意值：

  ```
  create table student
  (ID varchar(5),
   name varchar(20) not null,
   dept_name varchar(20),
   tot_cred numeric(3,0) default 0,
   primary key(ID)
  )
  ```

  由于tot-credit的值默认值被声明为0，插入student元祖时可省略该属性的值：

  ```
  insert into student(ID,name,dept_name) values('12789','Newman','Comp.Sci.')
  ```

- 创建索引***：
  - 在关系的属性上所创建的索引是一种数据结构，它允许数据库系统高效的找到关系中那些在索引属性上取给定值的元祖，而不用扫描关系中的所有元祖；
  - create index student_id_index on student(ID);
  - 如果用户提交的`SQL`查询可以从索引中获益，那么`SQL`查询处理器就会自动使用索引；
- 大对象`Large Object`类型：
  - `clob`：字符数据的大对象数据类型
  - `blob`：二进制数据的大对象数据类型
  - Example:

    ```
    book_review clob(10KB);
    image blob(10MB);
    movie blob(2GB);
    ```

    对于包含大对象的结果元祖而言，把整个大对象放入内存中是非常低效和不现实的。相反，一个应用通常用一个`SQL`查询来检索出一个大对象的“定位器”，然后在宿主语言中用这个定位器来操作对象，应用本身也是用宿主语言书写的。
- 用户定义的类型：

- 结构化数据类型`structured data type`：
  - 允许创建具有嵌套记录结构、数组和多重集的复杂数据类型；
- 独特类型`distinct type`：
  - 检测诸如把一个教师姓名赋给一个系名的赋值、把美元表示的货币值与英镑表示的比较等情况，并避免；
  - 类型：create type 子句定义：

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

  - 新创建的类型可以用作关系属性的类型：

```
create table department
(dept_name varchar(20),
building varchar(15),
budget Dollars
);
```

  - 强类型检查：department. budget + 20 false ------ 转换 cast
    - **cast** (department. budget to numeric(12,2))
  - `SQL`提供了**drop type**和**alter type**子句来删除或修改以前创建过的类型;
  - 域：可以在基本类型的基础上施加完整性约束，create domain 定义

    create domain Dollars as numeric(12, 2) not null;
  - 类型和域之间的差别：
    - 在域上可以声明约束，也可以为域类型变量定义默认值，而用户定义类型上不能声明约束或默认值;
    - 域不是强类型的：一个域类型的值可以被赋给另一个域类型，只要它们的基本类型是相容的;
  - 当把check子句应用到域时，允许模式设计者指定一个谓词，被声明为来自该域的任何变量都必须满足这个谓词：

```
create domain YearlySalary numeric(8,2)
    constraint salary_value_test check(value>=29000.00)
```

    约束命名：constraint salary_value_test 可选，系统使用这个名字指出违反哪个约束

```
create domain degree_level varchar(10)
    constraint degree_level_test
        check (value in ('Bachelors','Masters','Doctorate'))
```

- create table 的扩展***
  - 创建与现有某个表模式相同的表：create table table_name_2 like table_name_1
  - 书写复杂查询把查询的结果存储成一个新表通常很有用，这个表通常是临时的

```
create table t as
    (select *
    from instructor
    where dept_name = "Music";
    )
with data;
```

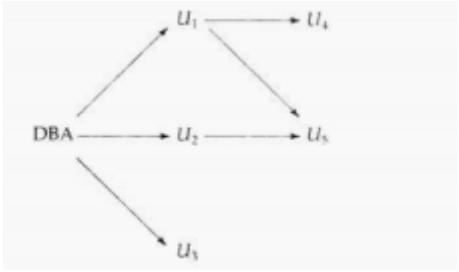
        - SQL 2003 标准：如果省略 with data 子句，表会被创建，但不会载入数据；
- 模式、目录与环境***

4.6 授权

> - 数据的授权包括：
>   - 授权读取数据
>   - 授权插入新数据
>   - 授权更新数据
>   - 授权删除数据
>
>   > - 每种类型的授权都称为一个权限(privilege)
>   > - 在数据库的某些特定部分上授权给用户所有这些类型的权限，或者完全不授权，或者这些权限的一个组合
> - 数据库模式上的授权：
>   - 创建、修改或删除关系等
> - 数据库管理员：被授予最大的授权形式

- 权限的授予与撤回
  - 一个创建了新关系的用户将被自动被授予该关系上的所有权限；
  - 授予权限：
    - **grant** 语句：grant <权限列表> on <关系/视图名> to <用户/角色名>
      - select: grant select on department to User_1
      - update: grant update (budget) on department to User_1
        - update 权限既可以在关系的所有属性书上授予又可以只在某些属性上授予；
      - insert:
        - insert 权限也可以指定属性列表：对关系的任何插入必须只针对这些属性，系统将其它属性要么赋默认值，要么赋null;
      - delete
      - all privileges
    - 用户名**public**指系统的所有当前用户和将来用户，对public的授权隐含着对所有当前用户和将来用户的授权；
    - SQL 授权机制可以对整个关系或一个关系的指定属性授予权限，而不允许对任一关系的指定元祖授予权限；
  - 权限收回：
    - **revoke** 语句：revoke <权限列表> on <关系/视图名> from <用户/角色名>
      - revoke select on department form User_1;
      - revoke update (budget) on department from User_1;
    - 如果被收回权限的用户已经把权限授予其它用户，权限的收回会更加复杂；

- 权限的转移
  - 默认情况下，被授予权限的用户/角色无权把此权限授予其它角色；
  - 允许授权转移：**grant option**
    - grant select on department to User_1 with **grant option**；
  - 授权图`authorization graph`：



    - 用户具有权限的充分必要条件是：**当且仅当存在从授权图的根(即代表数据库管理员的顶点)到代表给用户定点的路径**；
- 权限的收回
  - `DBA`收回用户U_1的授权，则U_4权限被收回，U_5则由于U_2的关系权限保留；
  - 用户相互授权破坏权限收回规则：不可行(注意：用户具有权限的充分必要条件)；
  - 级联收回：在大多数书籍库系统中，级联是默认行为
    - 防止级联收回：**restrict**
      - revoke select on department from User_1 restrict;
    - 允许级联收回：**cascade**
      - 可省略；
  - revoke grant option for select on department from User_1;

4.7 使用程序设计语言访问数据库

- JDBC
- ODBC
- 嵌入式`SQL`
  - `SQL`查询所嵌入的语言被称为宿主语言，宿主语言中使用的`SQL`结构被称为嵌入式`SQL`；
  - 使用宿主语言写出的程序可以通过嵌入式`SQL`的语法访问和修改数据库中的数据：
    - 使用嵌入式`SQL`的程序在编译前必须由一个特殊的预处理器进行处理：
      - 嵌入式的`SQL`请求被宿主语言的声明以及运行时刻执行数据库访问的过程调用调用所代替，然后所产生的程序由宿主语言编译器编译；
  - 使用嵌入式`SQL`时，`SQL`错误可以在编译过程中被发现；
  - 预处理器识别`SQL`请求：
    - 格式：EXEC `SQL` <嵌入式`SQL`语句>;
    - 确切语法取决于宿主语言(Coble ; ---> END_EXEC; Java # `SQL` {...})
  - 应用程序插入`SQL INCLUDE SQLCA`语句：表示预处理器在此处插入特殊变量以用于程序和数据库系统间的通信；
  - 执行`SQL`语句之前，程序必须首先连接到数据库：
    - EXEC `SQL` connect server user user-name using password;
      - server 标识要建立连接的服务器；
  - 嵌入式`SQ`语句可以使用宿主语言的变量，注意添加冒号(:)以区别；