数据库系统概念

Chapter 3 & 4 【SQL语句】

SQL查询的操作序列：

- 为 `from` 子句中列出的关系产生笛卡尔积；
- 在上一步的结果上应用 `where` 子句中指定的谓词；
- 对于上一步结果中的每个元组，输出 `select` 子句中指定的属性。

包含聚集、group by或having子句的SQL查询的操作序列：

- 先根据 `from` 子句来计算出一个关系；
- 如果出现了 `where` 子句， `where` 子句中的谓词应用在上一步产生的结果关系上；
- 如果出现了 `group by` 子句，满足 `where` 谓词的元组通过 `group by` 子句形成分组；如果没有 `group by` 子句，满足 `where` 谓词的整个元组集将被当做一个分组；
- 如果出现了 `having` 子句，它将应用到每个分组上；不满足 `having` 子句谓词的分组将被抛弃。
- `select` 子句利用剩下的分组产生出查询结果中的元组，即在每个分组上应用聚集函数来得到单个结果元组，**输出元组个数等于分组个数。**

单表查询

1. 从表中选择c1、c2列，并以condition为条件过滤得到选中的行

- condition可以用逻辑连接词 `and` ， `or` ， `not` 连接，包含比较运算符 `>` ， `>=` ， `=` ， `<` ， `<=` ， `<>` （等同于 `!=` ），也可以是 `between ... and ...` 或者 `(a,b) = (value1, value2)`

```
select c1, c2 from t
where condition;
```

2. 以升序/降序排列所选行

```
select c1, c2 from t
order by c1 asc/desc;
```

3. 在排好序后，忽略前k行，然后从剩下的行中选择最多n行作为结果

```
select c1, c2 from t
order by c2
limit n offset k;
```

4. 去重/不去重

```
select distinct/all c1 from t;
```

5. 聚集（分组聚集）

- aggregate 包含 avg ， min ， max ， sum ， count
- 除了 count(*) 外所有的聚集函数都**忽略输入集中的空值**

```
select c1, aggregate(c2)
from t
group by c1
having condition;
```

多表查询

1. 笛卡尔积连接

```
select a.c1, b.c2 from a, b;
```

2. 自然连接

- 自然连接只考虑那些在两个关系模式中**都出现**的属性上**取值相同**的元组对。自然连接列出的属性顺序为：先是两个关系模式中的**共同属性**，然后是那些只出现在**第一个**关系模式中的属性，最后是那些只出现在**第二个**关系模式中的属性。

```
select a.c1, b.c2 from a natural join b;
```

3. 在指定属性上做自然连接，使用 join ... using (...) 运算

```
select p1, p2
from (a natural join b) join c using (p3);
```

4. 内连接

```
SELECT c1, c2
FROM t1
INNER JOIN t2 ON condition;
```

5. 左外连接，只保留出现在左外连接运算之前的关系中的元组

- 计算方式如下：
    - 1. 先计算内连接的结果
    - 2. 对于在内连接左侧关系中任意一个与右侧关系中任何元组都不匹配的元组t，向连接结果中加入一个元组r，r的构造如下：
        - 元组r从左侧关系得到的属性被赋为t中的值
        - r的其他属性被赋为空值
- 注意属性的顺序

```
SELECT c1, c2
FROM t1
NATURAL LEFT OUTER JOIN t2 ON condition;
```

6. 右外连接，只保留出现在右外连接运算之后的关系中的元组

- 计算方式与左外连接相反
- 注意属性的顺序

```
SELECT c1, c2
FROM t1
NATURAL RIGHT OUTER JOIN t2 ON condition;
```

7. 全外连接，保留出现在两个关系中的元组 **MySQL不支持**

```
SELECT c1, c2
FROM t1
FULL OUTER JOIN t2 ON condition;
```

附加的基本运算

1. 更名运算 `... as ...`

```
select c1, c2 as p2
from t1 as a, t2 as b
where a.c3 = b.c3;
```

2. 模式匹配 `like '...'`

- 百分号 `%` ： 匹配任意子串
- 下划线 `_` ： 匹配任意一个字符
- 定义转义字符 `escape '\'` ：

```
select c1 from t where c2 like '% Hello%';

select c1 from t where c2 like 'ab\%cd%' escape '\';
```

集合运算

1. 并集

- union 运算**自动去除重复**，如果想保留所有重复，则**必须**用 `union all` 代替 `union`

```
(select c1, c2 from t1 where condition1)
union
(select c1, c2 from t2 where condition2);
```

2. 交集 **MySQL不支持**

- intersect 运算**自动去除重复**，如果想保留所有重复，则**必须**用 `intersect all` 代替 `intersect`

```
(select c1, c2 from t1 where condition1)
intersect
(select c1, c2 from t2 where condition2);
```

3. 差集 **MySQL不支持**

- except/minus 运算**自动去除重复**，如果想保留所有重复，则**必须**用 `except/minus all` 代替 `except/minus`

```
(select c1, c2 from t1 where condition1)
except/minus
(select c1, c2 from t2 where condition2);
```

空值处理

1. 如果算术表达式的**任一输入**为空 `null` ，则该表达式（涉及诸如 +, −, ×, 或÷）结果为空。SQL将涉及空值的任何比较运算的结果视为 `unknown` （既不是 `is null` 也不是 `is not null` ）

- **and**：
  - `true and unknown` → `unknown`
  - `false and unknown` → `false`
  - `unknown and unknown` → `unknown`
- **or**：
  - `true or unknown` → `true`
  - `false or unknown` → `unknown`
  - `unknown or unknown` → `unknown`
- **not**：
  - `not unknown` → `unknown`
- 如果 `where` 子句谓词对一个元组计算出 `false` 或 `unknown` ，那么该元组不能被加入到结果集中

嵌套子查询

1. 集合成员资格 `in` ， `not in`

```
select c1
from t
where c2 = value2 and c3 not in (
    select c3 from t
    where c2 = value2 and c4 = value4
)
```

2. 集合的比较 `some` ， `all` （ `any` 同义于 `some` ， `in` 等同于 `exists` ）

- `=some` 等价于 `in`
- `<>all` 等价于 `not in`

3. 空关系测试

- 关系A包含关系B： `not exists (B except A)`

4. 重复元组存在性测试 `unique`

- 当作为参数的查询结果中没有重复的元组时， `unique` 返回 `true` 值
- `unique` 谓词在空集上返回 `true` 值
- 当一个元组有多个副本，但存在一个属性为空，则 `unique` 测试有可能返回 `true` 值（因为空值的比较返回 `unknown` ）

5. from子句中的子查询

- temp是临时关系的名称，包含tc1和tc2两个属性

```
select c1, c2
from (select c1, avg(c2)
    from t
    group by c1)
    as temp (tc1, tc2)
where condition;
```

6. 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

7. 标量子查询

- SQL允许子查询出现在单个值的表达式可以出现的任何地方，只要子查询只返回单个属性的单个元组；这样的子查询称为标量子查询。select、where、having子句都可以使用标量子查询。

数据库的修改

1. 删除元组

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

2. 删除关系

```
drop table r;
```

3. 插入

```
insert into tablename (c1name, c2name, c3name)
values (value1, value2, value3);


insert into tablename
    select c1, c2, value3
    from t
    where condition;
```

4. 更新

```
update tablename
set c1 = c1 * 1.05;


update tablename
set c1 = c1 *1.05
where condition;


update tablename
set c1 = case
        when condition then c1 * 1.05
        else c1 * 1.03
    end;
```

5. 添加属性

- A是添加属性的名称，D是属性的域

```
alter table r add A D;
```

视图

1. 创建视图

- 视图关系在概念上包含查询结果中的元组，但**并不进行预计算和存储**，而是存储与视图关系相关联的查询表达式（**即定义**）。
- 当与视图关系有关的实际关系改变时，视图关系随之改变。这样的视图称为**物化视图** (materialized view)。

```
CREATE VIEW v(c1,c2)
AS
SELECT c1, c2
FROM t;
```

2. 视图更新

- 视图更新存在的问题
    - 1. 元组在插入到实际关系时属性不全
    - 2. 元组在插入到实际关系时不满足外键约束

    - ……
- 除了一些有限的情况外，一般不允许对视图关系进行修改。

> 一般说来，如果定义视图的查询对于下列条件都能满足，则称SQL视图是**可更新的** (updatable)（即视图上可以执行插入、更新或删除）：
>
> - `from` 子句中只有一个数据库关系。
> - `select` 子句中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**声明，也不构成主码的一部分。
> - 查询中不含有 `group by` 或 `having` 子句。

- 满足上述条件的SQL视图**仍然**存在下面这个问题：
    - 创建视图

```
create view his_ins as
select * from instructor where
dept_name='History';
```

    - 插入元组

```
insert into his_ins values('25566','Brown','Biology',100000);
```

    - 该插入被允许，但由于新插入的记录不满足 `where` 子句的条件，所以不会显示在视图中。
- 为了解决这个问题，可以在创建视图的末尾包含 `with check option` 子句来定义视图。这样，如果向视图中插入一条不满足视图 `where` 子句的条件的元组，数据库系统将拒绝该插入操作。

事务 (Transaction)

- 事务由查询和（或）更新语句的序列组成。SQL标准规定当一条SQL语句被执行，就隐式地开始了一个事务。下列SQL语句之一会结束一个事务：
- Commit work
- Rollback work
- 一个事务或者在完成所有步骤后提交其行为，或者在不能成功完成其所有动作的情况下回滚其所有操作，通过这种方式，数据库提供了对事务具有**原子性** (atomic)的抽象。

完整性约束 (Integrity Constraints)

- 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。因此，完整性约束防止的是对数据的意外破坏。

1. 单个关系上的约束
   - **not null**

```
create table t(
    name varchar(20) not null,
    budget numeric(12,2) not null
);
```

   - **unique**

     unique 声明指出属性(A, B, C, D, ...)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。（候选码属性可以为 null ）

```
unique(A, B, C, D, ...)
```

   - **check(<predicate>)**

     通常用 check 子句来保证属性值满足指定的条件，如：

```
create table section(
    course_id varchar(8),
    sec_id varchar(8),
    semester varchar(6),
    year numeric(4,0),
    building varchar(15),
    room_number varchar(7),
    time_slot_id varchar(4),
    PRIMARY KEY (course_id, sec_id, semester, year),
    check (semester in ('Fall', 'Winter', 'Spring', 'Summer'))
);
```

2. 参照完整性(referential integrity)

> **参照完整性约束**要求：在参照关系中任意元组在特定属性上的取值必然等于被参照关系中某个元组在特定属性上的取值。

   - 参照完整性约束也称为子集依赖
   - 使用外码保证参照完整性：
     - cascade 表示"级联"操作，可选的其他操作有 set null 或 set default 。
     - 外码中的属性允许为 null

```
...
foreign key (dept_name) references department
    on delete cascade
    on update cascade,
...
```

3. 事务中对完整性约束的违反
    - 考虑如下SQL语句：

```
create table person (
    ID  char(10),
    name char(40),
    mother char(10),
    father  char(10),
    primary key ID,
    foreign key father references person,
    foreign key mother references person);
```

    - 插入第一条数据时违反了参照性约束！
    - 使用 `set constraints` $constraint-list$ `deferred` 可以延缓约束检查
4. 复杂 `check` 条件与断言
    - 一个**断言**(assertion)就是一个谓词，可以添加在 `check()` 条件中
    - 创建一个断言：

```
create assertion <assertion-name> check <predicate>
```

SQL的数据类型与模式

1. 日期和时间
    - date，日历日期
      2005-7-27
    - time，一天中的时间，包括小时、分和秒，**可以使用** `with timezone` **指定时区**
      09:00:30
      09:00:30.75
    - timestamp， `date` 和 `time` 的组合，**可以使用** `with timezone` **指定时区**
      2005-7-27 09:00:30.75
    - interval，时间间隔
2. 默认值
    - 在 `create table` 中使用 `default ...` 为属性设定默认值
3. 索引
    - 在关系的属性上所创建的**索引**(index)是一种数据结构
    - 创建索引：

```
create index <index name> on table_name(column_name)
```

4. 用户自定义类型

- SQL支持两种形式的用户定义数据类型第一种称为独特类型(distinct type)，另一种称为结构化数据类型(structured data type)
- 独特类型的创建：

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

- 尝试为Pounds类型的变量赋予一个Dollars类型的值会导致一个编译时错误。
- 可以使用 `drop type` 和 `alter type` 子句来删除或修改以前创建过的类型。
- 创建**域**：

```
create domain person_name char(20) not null
```

- 类型和域的区别：
  1. 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可以。
  2. 域不是强类型的。因此一个域类型的值可以被赋给另一个域类型，只要他们的基本类型是相容的。

5. `create table` 的扩展
   - SQL提供了一个 `create table like` 的扩展来支持创建一个与现有某个表的模式相同的表：

```
create table table_name2 like table_name1;
```

授权 (authorization)

- 我们可能会给一个用户在数据库的某些部分授予几种形式的权限。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

1. 权限的授予与收回
   - SQL标准包括 `select` 、 `insert` 、 `update` 、 `delete` 权限
   - 授予权限的命令为：

```
grant < privileges list >
on < table_name / view_name >
to < user_name / role_name >
```

   - 用户名 `public` 用来表示系统所有当前用户和将来的用户。因此对 `public` 的授权隐含着对所有当前用户和将来用户的授权。
   - 收回权限的命令为：

```
revoke < privileges list >
on < table_name / view_name >
from < user_name / role_name >
```

2. 角色
    - 任何可以授予给用户的权限都可以授予给角色。
    - 创建角色：

```
create role role_name;
```

    - 角色可以像用户一样被授予权限：

```
grant select on table_name to role_name;
```

    - 角色可以授予给用户，也可以授予给其他角色
3. 模式的授权
    - SQL提供了一种 `references` 权限，允许用户在创建关系时声明外码。
4. 权限的转移
    - 获得了某些形式授权的用户可能被允许将此授权传递给其他用户。在默认的方式下，被授予权限的用户/角色**无权**把得到的权限再授予给另外的用户/角色。
    - 使用 `with grant option` 子句，可以在授权时允许接受者把得到的权限再传递给其他用户
5. 权限的收回
    - 默认的权限收回是采用级联机制，如果要防止级联收回，可以增加 `restrict` 关键字，当存在任何级联收回时，系统就返回一个错误，并且不执行收回动作。