COMP9311 Database Systems

Author : Yunqiu Xu

`Database` `SQL` `PostgreSQL` `UNSW`

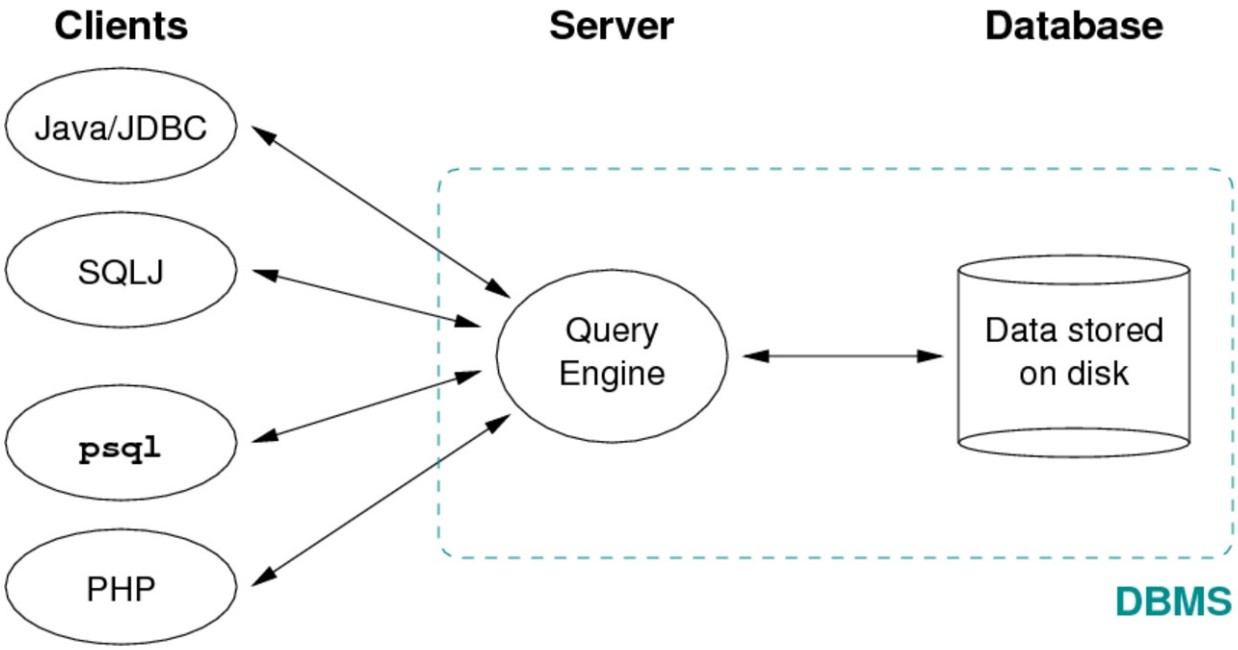
Lecture 1 Introduction, Data Modelling, ER Notation

- Some basic knowledge about database
- Entity-relationship model

1.1 Home Computing

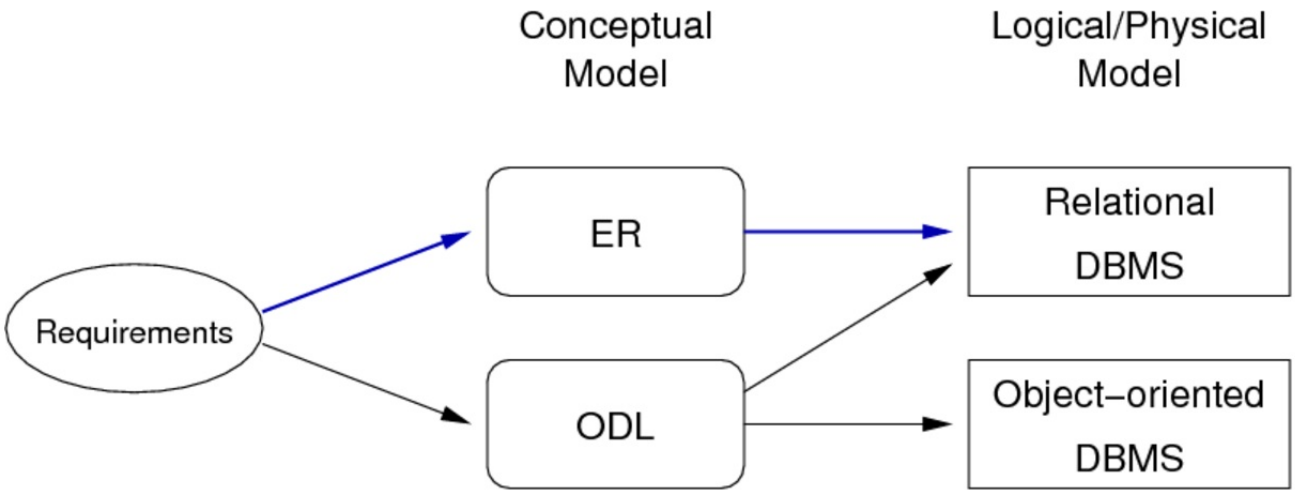
- Software needed: PostgreSQL 9.0 , PHP 5.3 , aPACHE 2
- Alternative: run them on CSE servers(grieg), need VPN

1.2 Typical environment for a modern DBMS



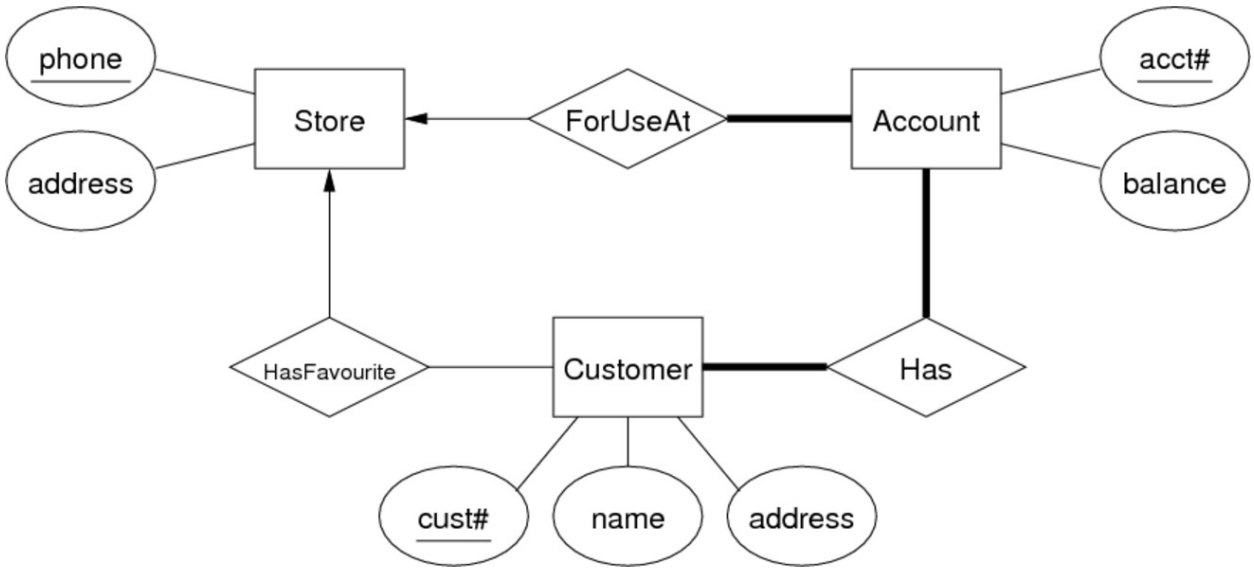
1.3 Data Modelling

- Describe information/relationships/constraints
- Kinds of modellig : logical&physical
  - logical: abstract, for conceptual design (e.g. ER, ODL)
  - physical: record-based, for implementation (e.g. relational)
- Strategy: design using abstract model; map to physical model

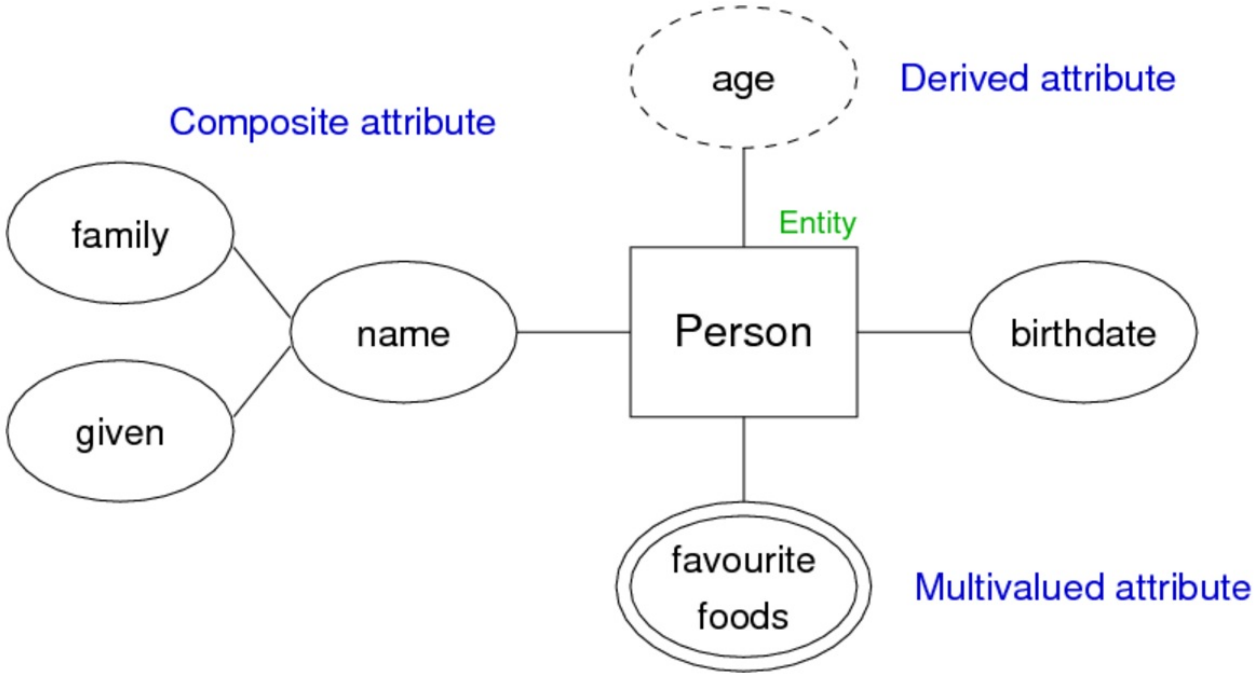


1.3 ER Data Modelling

- attribute+entity+relationship
- ER diagram consist of: entity set + relationship set + attributes + connections



1.3.1 Attribute notations



- rectangle: entity
- oval attribute:
  - dashed oval: derived attribute
  - double oval: multivated attribute

1.3.2 Entity sets

- defination:
  - extensional view: a set of entities with same set of attributes
  - intensional view: a class of entities
- an entity may belong to many entity sets

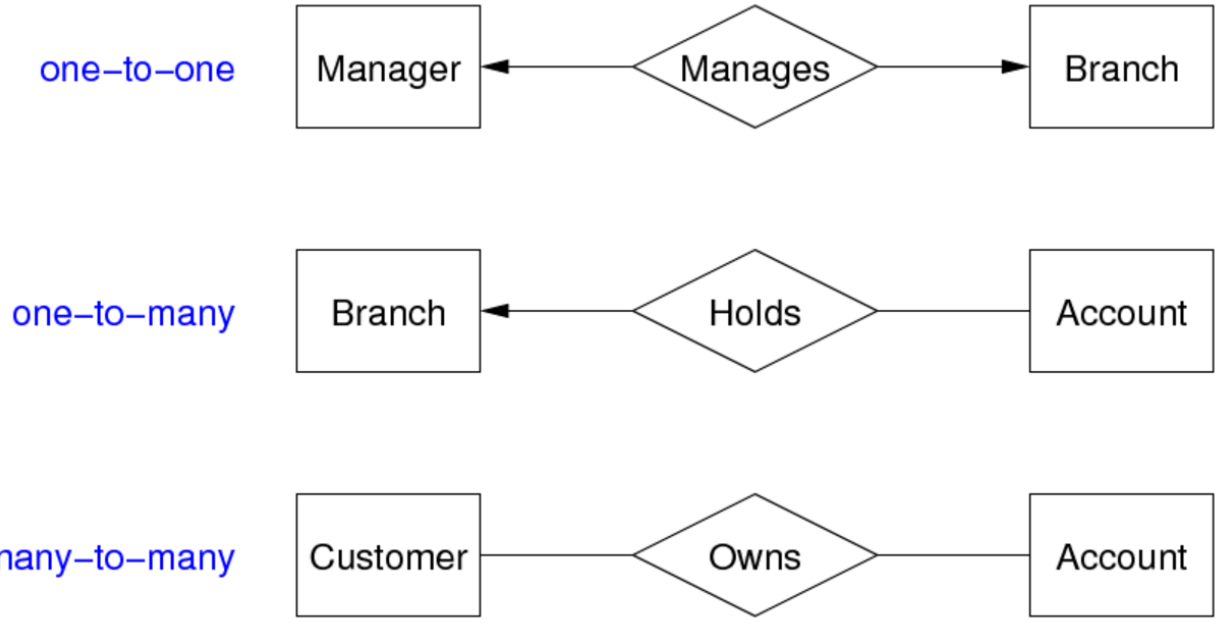
1.3.3 Keys (set of attributes)

- superkey: set of values are distinct over entity set
- candidate key: special superkey , no subset of attributes is also a superkey
- primary key: special candidate key, chosen by designer
- 使用下划线指定主键

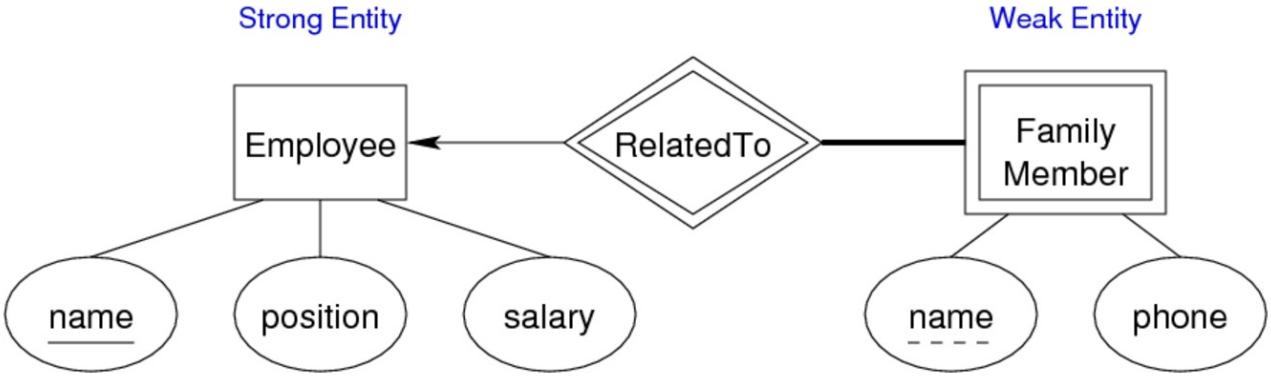
1.3.4 Relationship sets

- Partial(thin line)&Total(thick line)
  - all Loan are taken out by Customer
  - Customer will not take out all Loan

- Cardinality(arrow line)



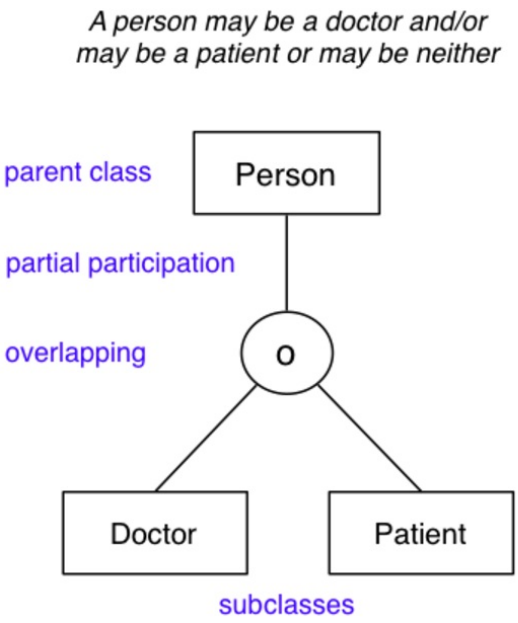
1.3.5 Weak Entity Sets: 需要依赖其他实体才能存在的实体

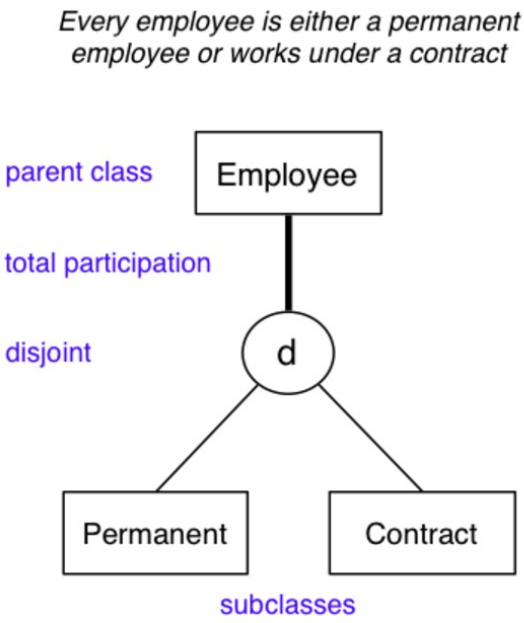


- Analysis:
  - Arrow:一个家庭成员只能有一种工作， 而可能有不止一个家庭成员是同行
  - Thick line: 所有家庭成员都有工作,但不是所有工作都由该家庭成员担任
  - Weak entity: 家庭成员依赖于其他实体集(仅仅靠自有属性无法标识主码)
  - dashed line: 弱实体没有主键, 'name' is discriminator

1.3.6 Subclasses and Inheritance

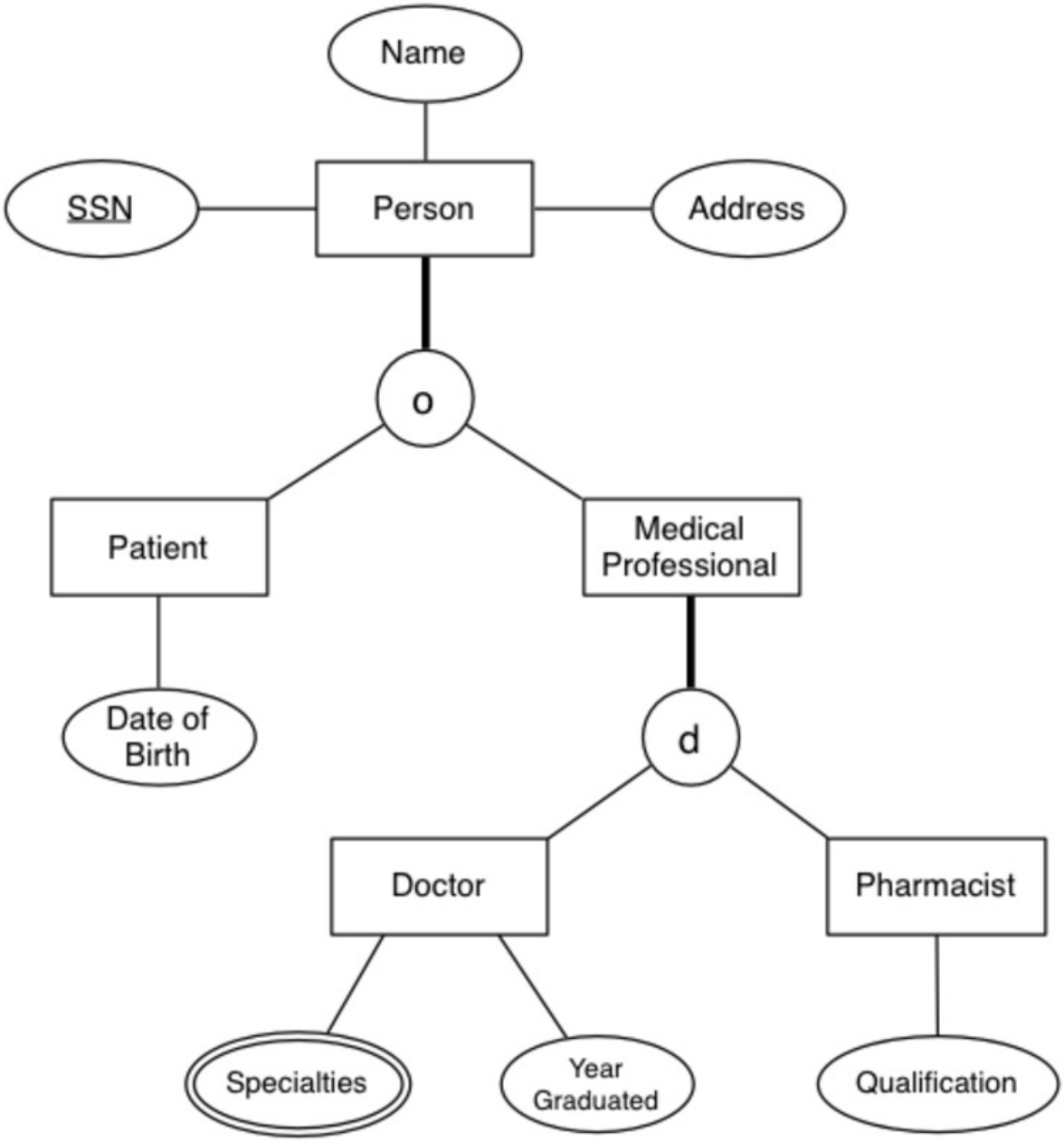
- overlapping/disjoint
- total/partial





1.3.7 最后举一个比较复杂的栗子

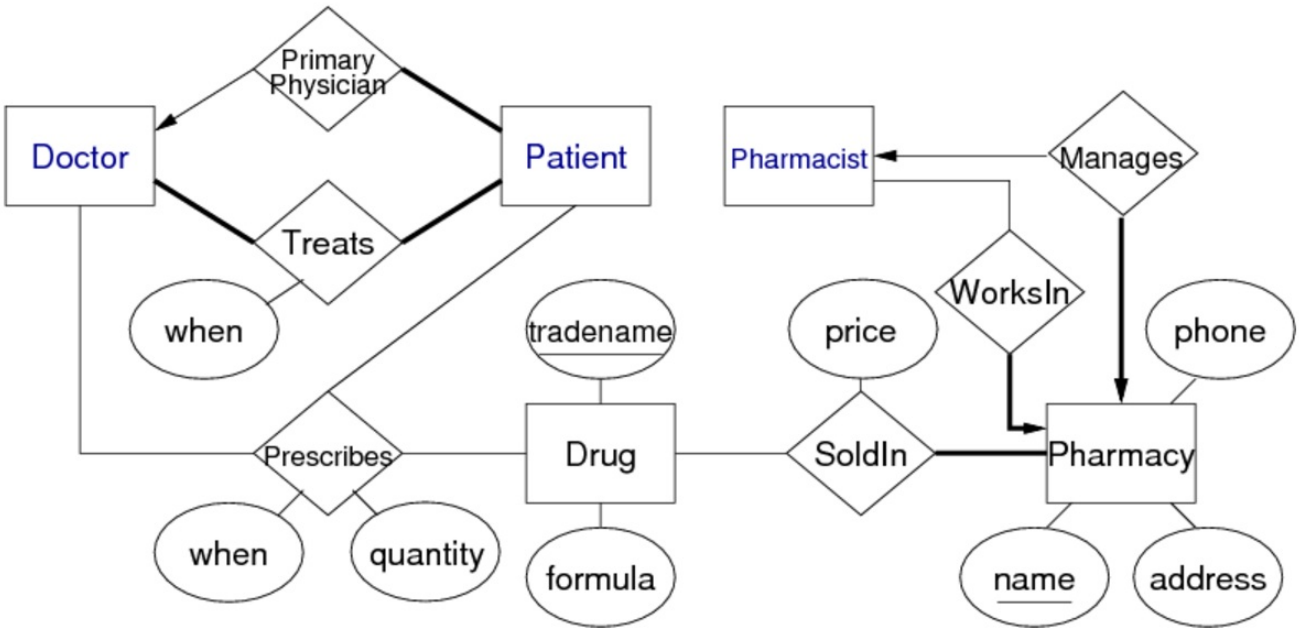
People subclasses



- **Person** has 3 attributes, **SSN** is primary key
- **Patient** and **Medical Professional** are subclasses of **Person** with property **total participation , overlapping**
  - 这里是医院,不是医学专家就是病人
  - 医生可能为病人
- **Doctor** and **Pharmacist** are subclasses of **Medical Professional** with property **total participation, disjoint**

- 不可同时担任医生和药剂师
- **Doctor** has multivariate attribute **Specialities**

Relationships



- **Doctor** and **Patient**:
  - partial + one-to-many:
    - 所有病人都有主治医生,不是所有医生都是主治医生
    - 每个病人一个主治医生,一个医生主治多个病人
  - total+ many-to-many:
    - 所有的医生都会治病人,所有的病人都被医生治
    - 一个医生治多个病人,一个病人被多个医生治
- **Drug** and **Pharmacy**: partial + many-to-many
  - 所有的药店都买药,但不是所有的药都可以在药店买到
  - 药店卖多种药,药在多个药店出售
- **Pharmacist** and **Pharmacy**:
  - partial + one-to-one:
    - 所有的药店都由药剂师管理, 但不是所有的药剂师都管理药店
    - 每个药剂师只能管理一个药店, 每个药店只能被一个药剂师管理
  - partial + one-to-many:
    - 所有的药店都由药剂师工作,但不是所有的药剂师都在药店工作
    - 每个药剂师只能在一家药店工作, 但在一家药店工作的可能有多个药剂师

Lecture 2 Relational Model, ER-Relational Mapping, SQL Schemas

2.1 Summary of ER

ER model is popular for doing conceptual design

- high-level, models relatively easy to understand
- good expressive power, can capture many details

Basic constructs: entities, relationships, attributes

Relationship constraints: total / partial, n:m / 1:n / 1:1

Other constructs: ineritance hierarchies, weak entities

Many notational variants of ER exist
(especially in the expression of constraints on relationships)

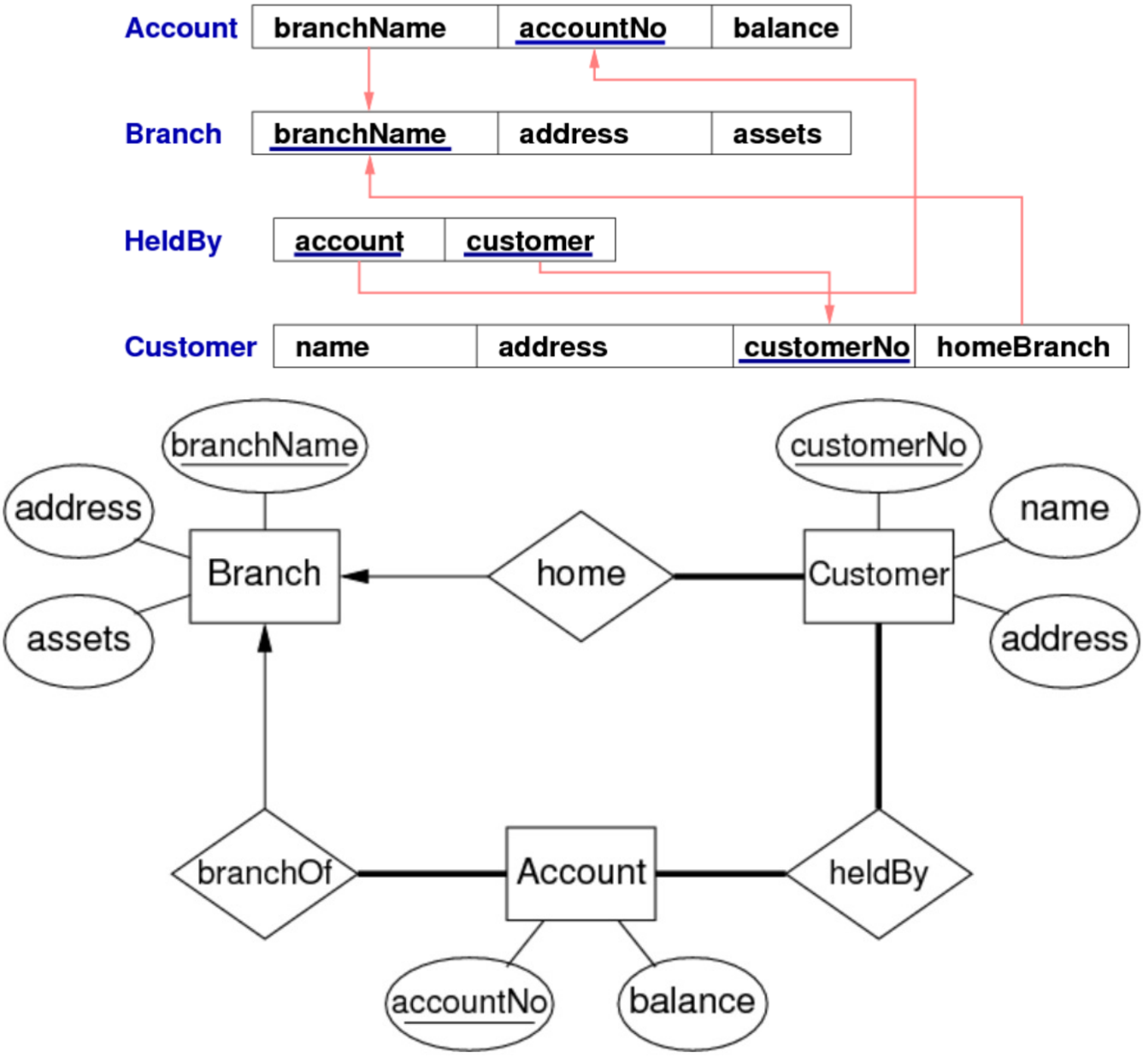
2.2 Relational Data Model

- relation: table consist of a **name** and a set of **attributes**
- attribute: column consist of a **name** and associated **domain** (allowed values)
- contraints : logic expressions
- no multi-valued attributes
- each relation has a key
- tuples in relation R: list of values
- instances in relation R: set of tuples
- Database schema: a collection of relation schemas
- Database : a collection of instances

2.3 Terminology

- different namespaces:
  - DBMS level: unique database
  - database level: unique schema
  - schema level: unique table
  - table level: unique attribute

Schema with 4 relations:



2.4 Contraints

- Domain constraints: AGE(15

- Key constraints: Student(id,...)
- Entity integrity

2.5 Referential integrity

- reference between tables
- foreign key : 某个表的外键指向另一个表的主键
- 构成外键的条件:
  - 该attribute指向另一个表的主键
  - 该attribute的值或者与主键表的值相同,或者为NULL

2.6 Relational Database

```
CREATE TABLE TableName (
    attrName1 domain1 constraints1 ,
    attrName2 domain2 constraints2 ,
    ...
    PRIMARY KEY (attri,attrj,...)
    FOREIGN KEY (attrx,attry,...)
                REFERENCES
                OtherTable (attrm,attrn,...)
);
```

- 每个表不一定有外键但一定有主键

2.7 Mapping ER Designs to Relational Schemas

2.7.1 ER Model VS Relational Model

Correspondences between relational and ER data models:

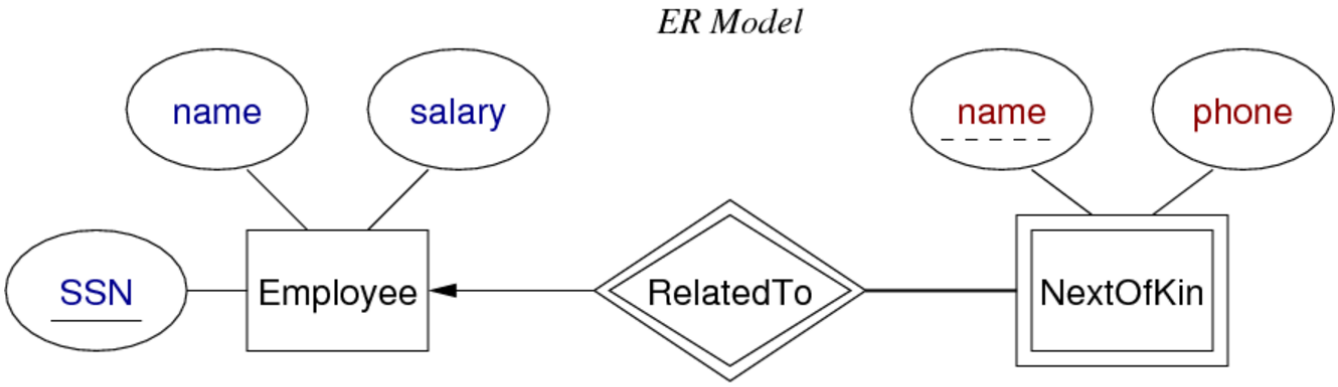
- attribute(ER) ≅ attribute(Rel), entity(ER) ≅ tuple(Rel)
- entity set(ER) ≅ relation(Rel), relationship(ER) ≅ relation(Rel)

Differences between relational and ER models:

- Rel uses relations to model entities *and* relationships
- Rel has no composite or multi-valued attributes (only atomic)
- Rel has no object-oriented notions (e.g. subclasses, inheritance)

2.7.2 实体

- 对于强实体: 直接将attributes 变为列名即可
- 对于弱实体: 需要有两个主键(自己的和与之联系的强实体的)



Relational Version

Employee

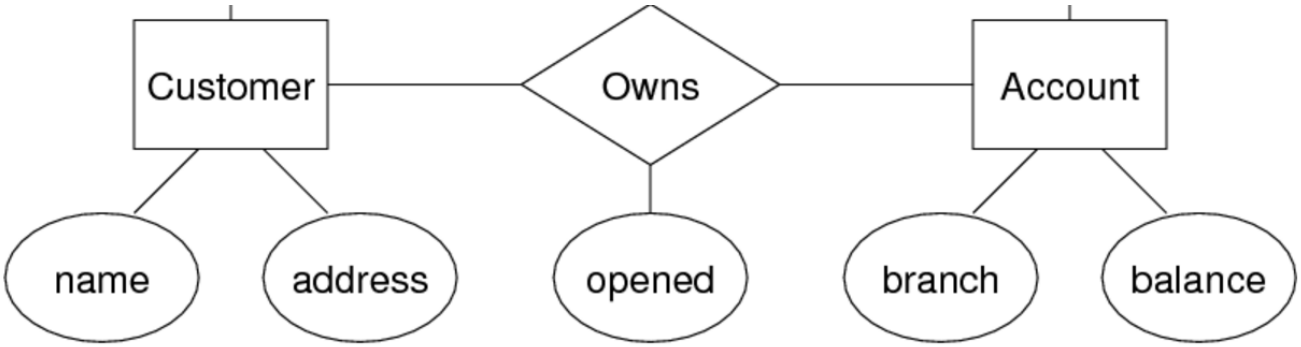
SSN	name	salary

NextOfKin

SSN	name	phone

2.7.3 Cardinality

- many-to-many : 新增一个relation, attributes为两个表的主键(既是主键又是外键)+关系属性



Relational Version

Customer

custNo	**name**	**address**

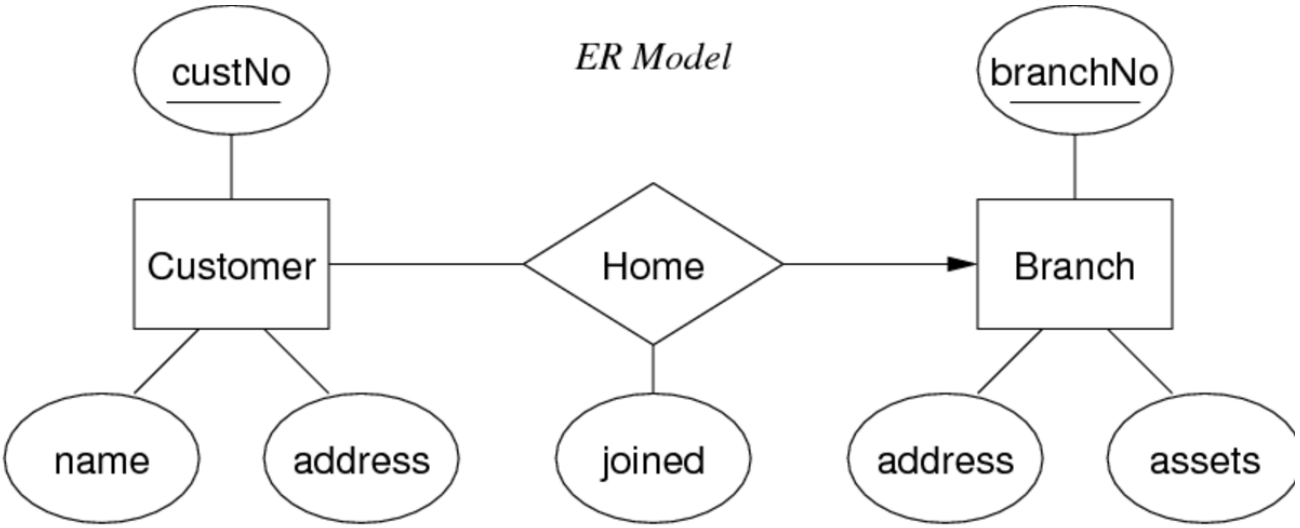
Account

acctNo	**branch**	**balance**

Owns

custNo	**acctNo**	**opened**

- one-to-many : 增加箭头指向的表的主键(外键)+关系属性



Relational Version

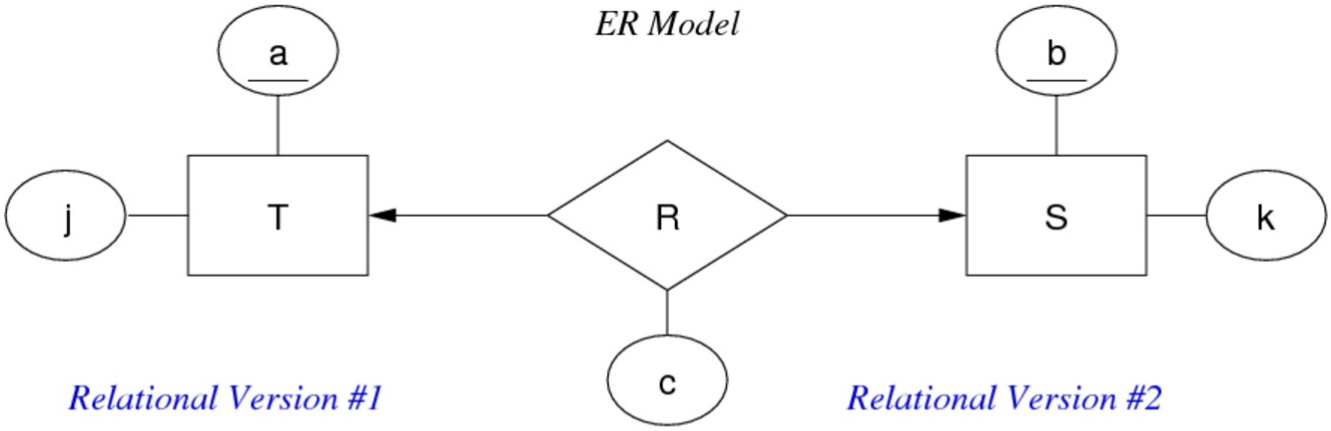
Customer

custNo	**name**	**address**	**branchNo**	**joined**

Branch

branchNo	**address**	**assets**

- one-to-one : 其中一个表需要加入另外一个表的主键(因为是唯一的)



Relational Version #1

T

a	**j**	**S**	**c**

S

b	**k**

Relational Version #2

T

a	**j**

S

b	**k**	**T**	**c**

Lecture 3 DBMSs, Databases, Data Modification

Lecture 4 SQL Queries

Lecture 5 More SQL Queries, Stored Procedures, PLpgSQL

5.1 SQL functions

```
CREATE OR REPLACE
    funcName(arg1type, arg2type, ....)
    RETURNS rettype
AS $$
    SQL statements
$$ LANGUAGE sql;
```

- 在函数中参数表示为$1, $2, ...
- 若要返回多列数据: `returns setof TupleType`
  - 注意需要先创建新数据类型
  - `CREATE TYPE TupleType AS (Name Text, Age Intger,...)`
- 栗子 1: 输入啤酒名字返回该啤酒的最高价格

```
-- max price of specified beer
create or replace function
    maxPrice(text) returns float
as $$
select max(price) from Sells where beer = $1;
$$ language sql;

-- usage examples
select maxPrice('New');
 maxprice
----------
      2.8

select bar,price from sells
where beer='New' and price=maxPrice('New');
    bar     | price
------------+-------
 Marble Bar |   2.8
```

- 栗子 2: 返回地区名返回该地区的所有酒吧

```
-- set of Bars from specified suburb
create or replace function
    hotelsIn(text) returns setof Bars
as $$
select * from Bars where addr = $1;
$$ language sql;

-- usage examples
select * from hotelsIn('The Rocks');
      name        |   addr    | license
------------------+-----------+---------
 Australia Hotel | The Rocks |  123456
 Lord Nelson     | The Rocks |  123888
```

5.2 PLpgSQL: 几个栗子

- 栗子 1:

```
create type IntVal as ( val integer ); #创建一个数据类型
create or replace function #函数名(参数名及类型) 返回类型
    iota(_lo int, _hi int, _step int) returns setof IntVal
as $$
declare #声明变量
 i integer;
 v IntVal;
begin #函数段
 i := _lo;
 while (i <= _hi)
 loop
  v.val := i;
  return next v;
  i := i + _step;
 end loop;
 return;
end;
$$ language plpgsql; #语言格式
```

- 栗子 2:

```
#Input : the name of a brewer and the name of a beer)
#Output : style of the beer
create or replace function BeerStyle(brewer text, beer text) returns
text
as $$
select s.name
from   Beer b, Brewer br, BeerStyle s
where  lower(br.name) = lower($1) and lower(b.name) = lower($2)
  and b.brewer = br.id and b.style = s.id
$$ language sql
;
```

- 栗子 3:

```
create or replace function
    iota(_lo int, _hi int) returns setof IntVal
as $$
declare
 v IntVal;
begin
 for v in select * from iota(_lo, _hi, 1)
 #若select得到的结果为多行，使用循环返回
 loop
  return next v; #返回下一个v
 end loop;
 return; #循环结束返回null
end;
$$ language plpgsql;
```

Lecture 6 Extending SQL: Queries, Functions, Aggregates, Triggers

6.1 PostgreSQL扩展

- 创建新定义域和数据类型

```
create domain Positive as integer check (value > 0);
create type Rating as enum ('poor','ok','excellent');
create type Pair as (x integer, y integer);
```

- 创建新函数(见上一章)

```
create function
  f(arg1 type1, arg2 type2, ...) returns type
as $$ ... function body ... $$
language language [ mode ];
```

  - immutable mode: does not access database (fast)
  - stable mode: does not modify the database
  - volatile mode: may change the database (slow, default)

6.2 高级查询

6.2.1 Window functions

- 之前已经学过了`GROUP BY`
- 另一种窗口函数为`OVER (PARTITION BY xxx)`
  - 与`GROUP BY`的区别在于, 使用聚合函数后不会合并项目
  - 依旧保持原来的所有行
  - 只是多了一列聚合函数结果
- 栗子: 后者计算平均分后不合并

```
select student,avg(mark) ... group by student

 student  |  avg
----------+-------
 46000936 | 64.75
 46001128 | 73.50


select *,avg(mark) over (partition by student) ...

 student  | course | mark | grade | stueval |  avg
----------+--------+------+-------+---------+-------
 46000936 |  11971 |   68 | CR    |       3 | 64.75
 46000936 |  12937 |   63 | PS    |       3 | 64.75
 46000936 |  12045 |   71 | CR    |       4 | 64.75
 46000936 |  11507 |   57 | PS    |       2 | 64.75
 46001128 |  12932 |   73 | CR    |       3 | 73.50
 46001128 |  13498 |   74 | CR    |       5 | 73.50
 46001128 |  11909 |   79 | DN    |       4 | 73.50
 46001128 |  12118 |   68 | CR    |       4 | 73.50
```

6.2.2 `WITH`语句

- `WITH`相当于一个暂时性的视图, 创建后只在当前语句中生效

```
with V as (select a,b,c from ... where ...),
     W as (select d,e from ... where ...)
select V.a as x, V.b as y, W.e as z
from   V join W on (v.c = W.d);
```

- 等价于

```
select V.a as x, V.b as y, W.e as z
from   (select a,b,c from ... where ...) as V,
       (select d,e from ... where ...) as W
where  V.c = W.d;
```

- 此外`WITH`语句可以接递归

6.2.3 递归查询

- 计算1-100的累加

```
1.  with recursive nums(n) as (
```

```
    select 1 #base
  union
    select n+1 from nums where n < 100 #recursion
)
select sum(n) from nums; #
>>> 5050

#若只是想展示数字
with recursive nums(n) as (
    select 1 #base
  union
    select n+1 from nums where n < 100 #recursion
)
select * from nums;
>>> 1 2 3 4 5 ... 100
```

6.2.4 条件查询

```
select case
when x=2 then 'x is 2!'
when x<>2 then 'x is not 2!'
end
from myTable;
```

- 栗子: 返回指定品酒人的地址

```
create or replace function TasterAddress(text) returns text
as $$
      select case
             when loc.state is null then loc.country
             when loc.country is null then loc.state
             else loc.state||', '||loc.country
             -- concat the state and country with ','
             end
      from   Taster t, Location loc
      where  t.given = $1 and t.livesIn = loc.id
$$ language sql
;
```

6.3 Aggregates

- 之前已经用过了诸如`COUNT(*), SUM()`等聚集函数

- 聚集函数的机理

```
AggState = initial state
for each item V {
    # update AggState to include V
    AggState = newState(AggState, V)
}
return final(AggState)
```

- 聚集函数与窗口函数结合

```
R                    select a,sum(b),count(*)
 a | b | c           from R group by a
---+---+---
 1 | 2 | x            a | sum | count
 1 | 3 | y           ---+-----+-------
 2 | 2 | z            1 |   5 |     2
 2 | 1 | a            2 |   6 |     3
 2 | 3 | b
```

6.3.1 定义聚集函数

- 基本构造
  - BaseType : 输入数据类型
  - StateType : type of intermediate states
  - sfunc(state,value): mapping function
    - 每次输入数据调用sfunc
  - initcond(type StateType): 可选, 初始值, 默认为null
  - finalfunc: 可选, 默认为identity function
    - 所有数据输入完毕,准备输出时调用finalfunc
    - 如果没有则就直接输出sfunc执行完毕的结果

```
CREATE AGGREGATE AggName(BaseType) (
    sfunc     = NewStateFunction,
    stype     = StateType,
    initcond  = InitialValue,
    finalfunc = FinalResFunction,
    sortop    = OrderingOperator
);
```

- 栗子 1: 重写count函数
    - 注意在定义oneMore函数时的参数x

```
create aggregate myCount(anyelement) (
    stype    = int,    -- the accumulator type
    initcond = 0,      -- initial accumulator value
    sfunc    = oneMore -- increment function
);

create function
    oneMore(sum int, x anyelement) returns int
as $$
begin return sum + 1; end;
$$ language plpgsql;
```

- 栗子 2: 两列数字的加和
    - 创建了一个新数据类型IntPair

```
create type IntPair as (x int, y int);

create function
    AddPair(sum int, p IntPair) returns int
as $$
begin return p.x + p.y + sum; end;
$$ language plpgsql;

create aggregate sum2(IntPair) (
    stype     = int,
    initcond  = 0,
    sfunc     = AddPair
);
```

- 栗子 3: 将某列的所有内容以逗号联结,输出为一列

```
# 该函数用于将给定的两个字符串用逗号联结
create or replace function
    appendNext(_state text, _next text) returns text
as $$
begin
 return _state||','||_next;
end;
$$ language plpgsql;

# 该函数用于将给定字符串的首字符删除
create or replace function
    finalText(_final text) returns text
as $$
begin
 return substr(_final,2,length(_final));
end;
$$ language plpgsql;


create aggregate concat (text)
(
    stype     = text,
    initcond  = '',
    # 初始值为''因此第一次执行sfunc的结果为',xxx'
    sfunc     = appendNext,
```

```
        # 所有输入数据都处理完毕后执行finalfunc
        # 删除一开始的逗号
        finalfunc = finalText
    );
```

6.4 Constraints

- 之前已经用过的
  - attribute (column) constraints
  - relation (table) constraints
  - referential integrity constraints

  Examples:

```
create table Employee (
   id      integer primary key,
   name    varchar(40),
   salary  real,
   age     integer check (age > 15),
   worksIn integer
              references Department(id),
   constraint PayOk check (salary > age*1000)
);
```

6.4.1 Assertions

- 若不满足断言, 抛出错误
- 栗子 1: 不存在选课人数超过10000的课
  - Courses或是CourseEnrolments发生变化时执行断言检查

```
create assertion ClassSizeConstraint check (
   not exists (
      select c.id from Courses c, CourseEnrolments e
      where  c.id = e.course
      group  by c.id having count(e.student) > 9999
   )
);
```

- 栗子 2: assets of branch = sum of its account balances
  - Branches或是Accounts发生变化时执行断言检查

```
create assertion AssetsCheck check (
   not exists (
      select branchName from Branches b
      where  b.assets <>
             (select sum(a.balance) from Accounts a
              where a.branch = b.location)
   )
);
```

- 在更新过程中查询被触发的断言效率不高: 可以使用Triggers代替断言

6.5 Triggers

- 触发器是存储在数据库中的进程, 在特定动作发生时被激活

  Triggers provide event-condition-action (ECA) programming:

  - an event activates the trigger
  - on activation, the trigger checks a condition
  - if the condition holds, a procedure is executed (the action)

6.5.1 触发器标准格式

- Event: INSERT / UPDATE / DELETE
- FOR EACH {ROW|STATEMENT}
  - 每当有tuple发生改变则检查一次
  - 若没有该语句则在所有改变都发生后一起检查, 检查完成后COMMIT
  - 注意BEFROE函数一定要有返回语句RETURN NEW/OLD, AFTER没限制
    - RETURN OLD或是检查触发异常: 回退原来的状态(不发生任何变化)

```
CREATE TRIGGER TriggerName
{AFTER|BEFORE}  Event1 [OR Event2 ...]
ON TableName
[ WHEN ( Condition ) ]
FOR EACH {ROW|STATEMENT}
EXECUTE PROCEDURE FunctionName(args...);
```

6.5.2 Event与Before/After的关系

- INSERT: 假设向原有{2,3}的表中插入1
  - INSERT 中只存在NEW变量, 这里NEW指向1;
  - 首先X被激活, 检查NEW 1;
  - 注意在NEW变量中可以再次改变插入值如将插入1改成插入666;
  - X检查完成后, DBMS在插入数据时进行约束检查(检查无误就插入);
  - 前面检查若出错了则回退到插入前的状态;
  - 插入完成后, 因为发生了插入动作, Y被激活,检查被插入的元素NEW (如果没更改插入元素的话还是1);

```
create trigger X before insert on T Code1;
create trigger Y after insert on T Code2;
insert into T values (a,b,c,...);
```

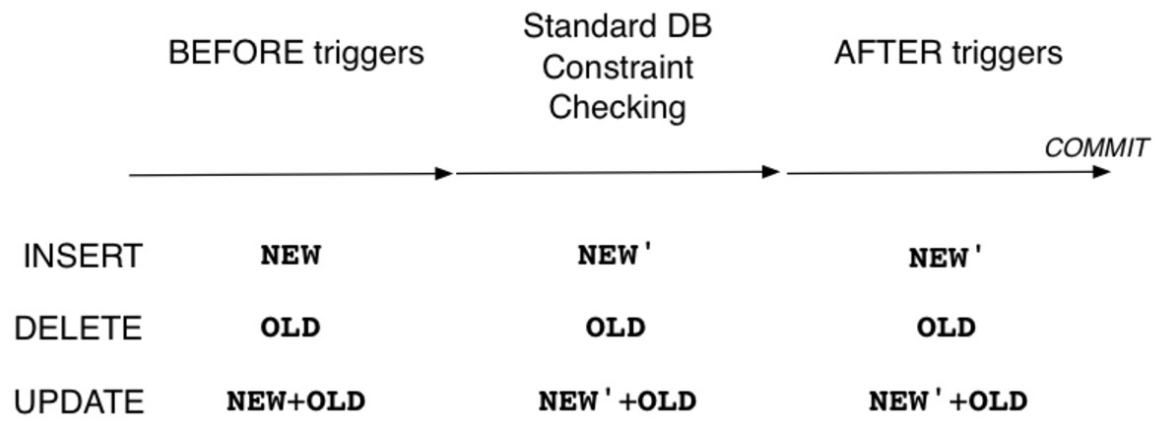
- UPDATE: 假设原表为{1,2,3},更新为{666,2,3}
  - 更新前X被激活, NEW-->666, OLD-->1;
  - X先检查NEW后检查OLD,此时NEW的值还是可以再变的;
  - X检查完成后, DBMS在更新数据时对NEW进行约束检查;
  - 更新完成后Y被激活, 检查NEW(666);

```
create trigger X before update on T Code1;
create trigger Y after update on T Code2;
update T set b=j,c=k where a=m;
```

- DELETE: 假设删除{1,2,3}中的1
  - DELETE 中只存在OLD, 这里OLD-->1;
  - 删除前X被激活,检查OLD;
  - X检查完成后, DBMS在删除数据时对OLD进行约束检查;
  - 删除完成后Y被激活,再次检查OLD(1);

```
create trigger X before delete on T Code1;
create trigger Y after delete on T Code2;
delete from T where a=m;
```

- 总结下:
  - 注意这里NEW'指NEW发生改变的情况(本来打算插入1后来在检查时又改成插入666了
  - 若未发生改变则始终就是NEW, NEW, NEW



6.5.3 举几个触发器的栗子

- 栗子 1: 插入或更新后检查
  - NEW州名格式是否正确
  - NEW该州是否存在
  - 若前两个检查未抛出异常, 返回NEW

```
create trigger checkState before insert or update
on Person for each row execute procedure checkState();

create function checkState() returns trigger as $$
begin
   -- normalise the user-supplied value
   new.state = upper(trim(new.state));
   if (new.state !~ '^[A-Z][A-Z]$') then
      raise exception 'Code must be two alpha chars';
   end if;
   -- implement referential integrity check
   select * from States where code=new.state;
   if (not found) then
      raise exception 'Invalid code %',new.state;
   end if;
   return new;
end;
$$ language plpgsql;
```

  - 测试结果

```
insert into Person
   values('John',...,'Calif.',...);
-- fails with 'Statecode must be two alpha chars'

insert into Person
   values('Jane',...,'NY',...);
-- insert succeeds; Jane lives in New York

update Person
   set town='Sunnyvale',state='CA'
         where name='Dave';
-- update succeeds; Dave moves to California

update Person
   set state='OZ' where name='Pete';
-- fails with 'Invalid state code OZ'
```

- 栗子 2:

```
Employee(id, name, address, dept, salary, ...)
Department(id, name, manager, totSal, ...)
```

  - Case 1: 在加入新职员后检查
    - 若dept非空(已被指向部门id)
    - 更新总工资
    - 检查完成后返回NEW

```
create trigger TotalSalary1
after insert on Employees
for each row execute procedure totalSalary1();

create function totalSalary1() returns trigger
as $$
begin
    if (new.dept is not null) then
        update Department
        set    totSal = totSal + new.salary
        where  Department.id = new.dept;
    end if;
    return new;
end;
$$ language plpgsql;
```

- Case 2: 在职员信息(departments/salaries)发生变化后检查
  - 职员被调到新部门后, 该部门总工资加上该职员工资
  - 职员被调离的老部门总工资减去该职员工资
  - 检查完成后返回NEW

```
create trigger TotalSalary2
after update on Employee
for each row execute procedure totalSalary2();

create function totalSalary2() returns trigger
as $$
begin
    update Department
    set    totSal = totSal + new.salary
    where  Department.id = new.dept;
    update Department
    set    totSal = totSal - old.salary
    where  Department.id = old.dept;
    return new;
end;
$$ language plpgsql;
```

- Case 3: 在职员离职后检查
  - 职员离职后, 若其dept非空(在旧部门的信息未更改)

- 更新旧部门信息, 总工资减去该职员原来的工资
- 检查完成后返回OLD

```
create trigger TotalSalary3
after delete on Employee
for each row execute procedure totalSalary3();

create function totalSalary3() returns trigger
as $$
begin
    if (old.dept is not null) then
        update Department
        set    totSal = totSal - old.salary
        where  Department.id = old.dept;
    end if;
    return old;
end;
$$ language plpgsql;
```

- 栗子3: Lab 6
- Case 1: Insertion trigger

```
create trigger InsertRating
after insert on  Ratings
for each row execute procedure insertRating();

create or replace function insertRating()
returns trigger
as $$
declare #initialize variable b as type beer
    b beer
begin
    #找到旧表中所有和插入新数据相关的数据, 储存为局部变量b
    select * into b from Beer where id = new.beer;

    #update the records in b
    b.nratings := b.nratings + 1;
    b.totrating := b.totrating + new.score;
    b.rating = b.totrating / b.nratings;

    #then update the table Beer via b
    update Beer
```

```
    set    nratings = b.nratings,
           totrating = b.totrating,
           rating = b.rating
    where  id = new.beer;
    return new;
end;
$$ language plpgsql;
```

- Case 2: update trigger

```
create trigger UpdateRating
after update on Ratings
for each row execute procedure updateRating();

create or replace function
    updateRating() returns trigger
as $$
declare #创建两个局部变量
    nb Beer; ob Beer;
begin
    select * into nb from Beer where id = new.beer;
    if (new.beer = old.beer) then #just change the rating of this beer
        if (new.rating = old.rating) then
            null;  # no change happens
        else
            # replace with the new rating
            nb.totrating := nb.totrating + new.score - old.score;
            nb.rating = nb.totrating / nb.nratings;
        end if;
    else
        # 更改了评分的啤酒名
        # 比如一开始给啤酒a评价4星,后来将啤酒a改成啤酒b

        # find the records about the 'old id'
        select * into ob from Beer where id = old.beer;
        # update the old records: a record is removed
        ob.totrating := ob.totrating - old.score;
        ob.nratings := ob.nratings - 1;
        ob.rating := ob.totrating / ob.nratings;
        # update the new records: a record is added
        nb.totrating := nb.totrating + new.score;
        nb.nratings := nb.nratings + 1;
        nb.rating := nb.totrating / nb.nratings;
        # update the TABLE with OB
```

```
        update Beer
        set    nratings = ob.nratings,
               totrating = ob.totrating,
               rating = ob.rating
        where  id = old.beer;
    end if;
    # update the TABLE with NB
    update Beer
    set    nratings = nb.nratings,
           totrating = nb.totrating,
           rating = nb.rating
    where  id = new.beer;
    return new;
end;
$$ language plpgsql;
```

- Case 3: Deletion trigger

```
create trigger DeleteRating
after delete on Ratings
for each row execute procedure deleteRating();

create or replace function
deleteRating() returns trigger
as $$
declare
    b Beer;
begin
    select * into b from Beer where id = old.beer;
    # update the old records
    b.nratings := b.nratings - 1;
    b.totrating := b.totrating - old.score;
    if (b.nratings = 0) then
        b.rating := null;
    else
        b.rating = b.totrating/b.nratings;
    end if;

    # update Table Beer via b
    update Beer
    set    nratings = b.nratings,
           totrating = b.totrating,
           rating = b.rating
    where  id = old.beer;
```

```
    return old;
end;
$$ language plpgsql;
```

Lecture 7 More Triggers, Programming with Databases

7.1 PHP/DB Interface

- 使用php从数据库中提取数据

```
$db = dbConnect("dbname=myDB");
...
$query = "select a,b,c from R where c >= %d";
$result = dbQuery($db, mkSQL($query, $min));
while ($tuple = dbNext($result)) {
   $tmp = $tuple["a"] - $tuple["b"] - $tuple["c"];
   # or ...
   list($a,$b,$c) = $tuple;
   $tmp = $a - $b - $c;
}
...
```

- 基本函数

```
dbConnect(conn) #establish connection to DB
dbQuery(db,sql) #send SQL statement for execution
dbNext(res) #fetch next tuple from result set
dbUpdate(db,sql) #send SQL insert/delete/update
```

- 栗子:
  - `$t = dbNext(resource $r);`迭代器,返回结果集r中的下一个元素

```
$q = "select name,max(mark) from Enrolments ...";
$r = dbQuery($db,$q);
$t = dbNext($r);
# results in $t with value
>>> array(0=>'John', "name"=>'John', 1=>95, "max"=>95)
```

- 来一个比较复杂的php栗子

```
$db_handle = pg_connect("dbname=bpsimple");
$query = "SELECT title, fname, lname FROM customer";
$result = pg_exec($db_handle, $query);
if ($result) {
  echo "The query executed successfully.\n";
  for ($row = 0; $row < pg_numrows($result); $row++) {
    $fullname = pg_result($result, $row, 'title') . " ";
    $fullname .= pg_result($result, $row, 'fname') . " ";
    $fullname .= pg_result($result, $row, 'lname');
    echo "Customer: $fullname\n";
  }
} else {
  echo "The query failed with the following error:\n";
  echo pg_errormessage($db_handle);
}
pg_close($db_handle);
```

7.2 DB/PL Mismatch

- 尽可能避免低效率语句

语句类型	Cost
建立新数据库连接	高消耗, 因此不推荐频繁开关机
建立查询	较高消耗
进入某个行	低消耗

```
# 低效率：需要进入多个不必要的行执行判断
$query = "select * from Student";
$results = dbQuery($db,$query);
while ($tuple = dbNext($results)) {
    if ($tuple["age"] >= 40) {
        --  process mature-age student
    }
}

# 高效率：直接在开头的查询中忽略了不必要的行
$query = "select * from Student where age >= 40";
$results = dbQuery($db,$query);
```

```
while ($tuple = dbNext($results)) {
    --  process mature-age student
}
```

- 再来个栗子:

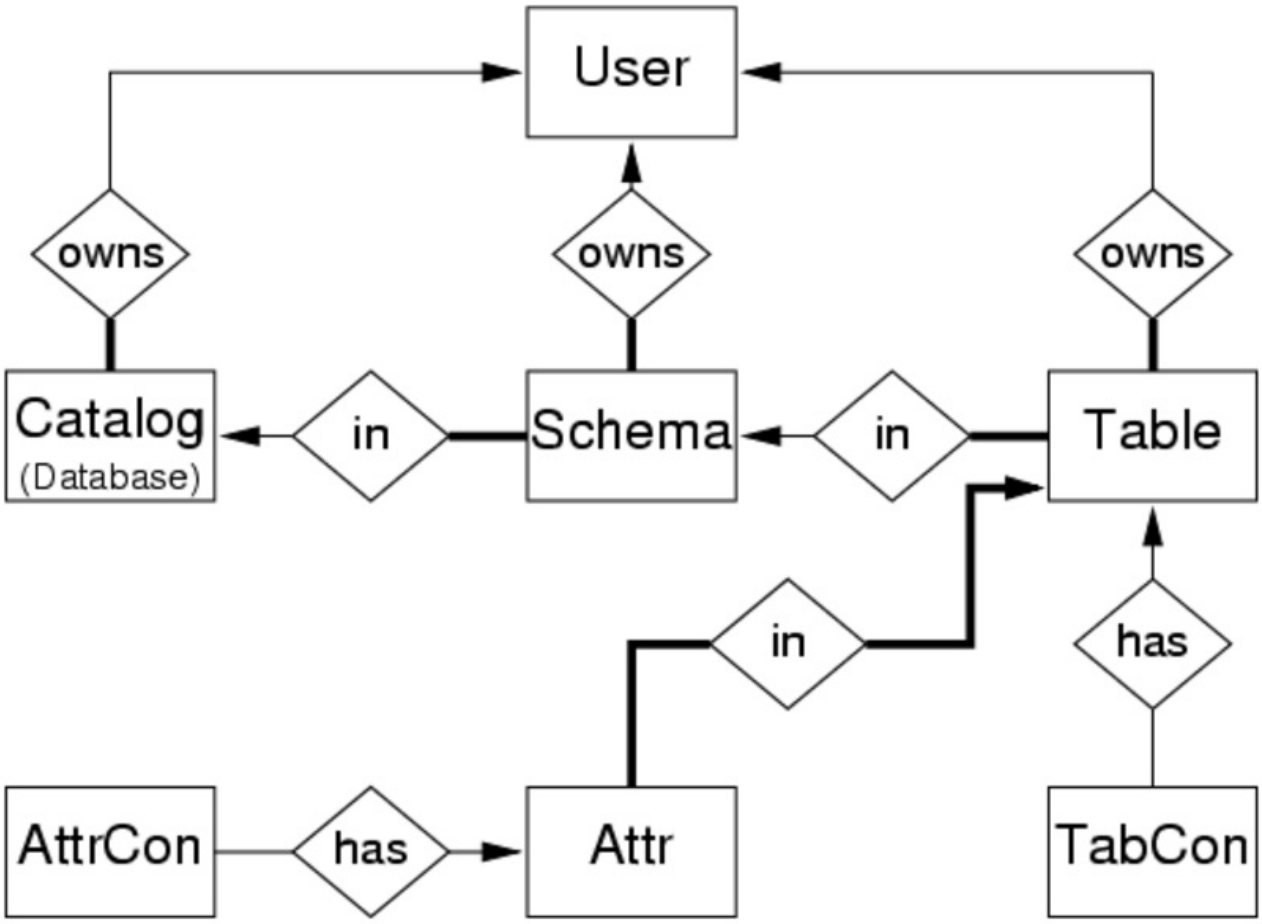
```
# 低效率：在循环中还要进行多次查询
$query1 = "select id,name from Student";
$res1 = dbQuery($db,$query1);
while ($tuple1 = dbNext($res1)) {
    $query2 = "select course,mark from Marks".
              " where student = $tuple1['id']";
    $res2 = dbQuery($db,$query2);
    while ($tuple2 = dbNext($res2)) {
        --  process student/course/mark info
    }
}

# 高效率：减少查询次数，一次搞定
$query = "select id,name,course,mark".
         " from Student s, Marks m".
         " where s.id = m.student";
$results = dbQuery($db,$query);
while ($tuple = dbNext($results)) {
    --  process student/course/mark info
}
```

Lecture 8 Catalogs, Privileges

8.1 Catalogs

- 系统表是关系型数据库存放结构元数据的地方，比如表和字段以及内部登记信息



- 系统表构成

INFORMATION_SCHEMA is available globally and includes:

Schemata(catalog_name, schema_name, schema_owner, ...)

Tables(table_catalog, table_schema, table_name, table_type, ...)

Columns(table_catalog, table_schema, table_name, column_name, ordinal_position, column_default, data_type, ...)

Views(table_catalog, table_schema, table_name, view_definition, ...)

Table_Constraints(..., constraint_name, ..., constraint_type, ...)

- 查询手段

```
1.  SELECT table_name,column_name FROM INFORMATION_SCHEMA.Columns limit 5;
```

8.2 Security, Privilege, Authorisation

8.2.1 UNIX界面

- 关于用户的操作
  - Capabilities: super user, create databases, create users, etc.
  - Config parameters: resource usage, session settings, etc.

```
CREATE USER Name IDENTIFIED BY 'Password'
ALTER USER Name IDENTIFIED BY 'NewPassword'
ALTER USER Name WITH Capabilities
ALTER USER Name SET ConfigParameter = ...
```

- 用户组: 组内用户权限相同

```
CREATE GROUP Name
ALTER GROUP Name ADD USER User1, User2, ...
ALTER GROUP Name DROP USER User1, User2, ...
```

8.2.2 在数据库内进行相关操作

- 创建用户

```
CREATE ROLE UserName Options
#Options include:
PASSWORD 'Password'
CREATEDB | NOCREATEDB
CREATEUSER | NOCREATEUSER
IN GROUP GroupName
VALID UNTIL 'TimeStamp'

#简单的栗子
CREATE ROLE name LOGIN PASSWORD '123456';
```

- 创建用户组

```
CREATE ROLE GroupName WITH USER User1, User2, ...
```

- 修改用户组

```
GRANT GroupName TO User1, User2, ...
```

```
REVOKE GroupName FROM User1, User2, ...
GRANT Privileges ... TO GroupName
REVOKE Privileges ... FROM GroupName
```

8.2.3 权限控制

- 设置权限
  - `WITH GRANT OPTION` 允许用户超越权限(获得其他用户的权限)

```
GRANT Privileges ON Object
TO ( ListOfRoles | PUBLIC )
[ WITH GRANT OPTION ]

#栗子
GRANT UPDATE ON accounts TO joe;
```

- 取消权限
  - `CASCADE`: 将依赖用户的权限一并取消
  - `RESTRICT`: 若存在依赖用户, 则拒绝取消权限
  - `RESTRICT`是默认选项, 我之前创建新数据类型后如果要删除需要特别指明`CASCADE`

```
REVOKE Privileges ON Object
FROM ListOf (Users|Roles) | PUBLIC
CASCADE | RESTRICT

#栗子
REVOKE ALL ON accounts FROM PUBLIC;
```

Lecture 9 Relational Design Theory, Normal Forms

- 设计关系型数据库时要注意防止冗余(一个字段在多个表中重复出现)
- 一些概念

Relation schemas	upper-case letters, denoting set of all attributes (e.g. *R*, *S*, *P*, *Q*)
Relation instances	lower-case letter corresponding to schema (e.g. *r(R)*, *s(S)*, *p(P)*, *q(Q)*)
Tuples	lower-case letters (e.g. *t*, *t'*, t_1, *u*, *v*)
Attributes	upper-case letters from start of alphabet (e.g. *A*, *B*, *C*, *D*)
Sets of attributes	simple concatenation of attribute names (e.g. *X=ABCD*, *Y=EFG*)
Attributes in tuples	tuple[attrSet] (e.g. *t[ABCD]*, *t[X]*)

9.1 函数依赖

- 对于R上的任意两个关系r1,r2, 若$r1[x] = r2[x] \Rightarrow r1[y] = r2[y]$, 则称X决定Y或者Y依赖于X,表示为$X \rightarrow Y$
- 说下我的理解: 其实就是若X为定义域,能否得到唯一的Y
- 栗子

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_1	c_2	d_2	e_1
a_3	b_2	c_1	d_1	e_1
a_4	b_2	c_2	d_2	e_1
a_5	b_3	c_3	d_1	e_1

- 可得到以下决定与依赖关系

  Since *A* values are unique, the definition of *fd* gives:

  - $A \rightarrow B, \quad A \rightarrow C, \quad A \rightarrow D, \quad A \rightarrow E$
  - $A \rightarrow BC, \quad A \rightarrow CD, \quad \ldots \quad A \rightarrow BCDE$
  - can be summarised as $A \rightarrow BCDE$

  Since all *E* values are the same, it follows that:

  - $A \rightarrow E, \quad B \rightarrow E, \quad C \rightarrow E, \quad D \rightarrow E$
  - in general, cannot be summarised as $ABCD \rightarrow E$

- 其他一些关系

Other observations:

- combinations of *BC* are unique, therefore $BC \rightarrow ADE$
- combinations of *BD* are unique, therefore $BD \rightarrow ACE$
- if *C* values match, so do *D* values, therefore $C \rightarrow D$
- however, *D* values don't determine *C* values, so $!(D \rightarrow C)$

We could derive many other dependencies, e.g. $AE \rightarrow BC$, ...

- Armstrong's rules

F1. Reflexivity e.g. $X \rightarrow X$

- a formal statement of *trivial dependencies*; useful for derivations

F2. Augmentation e.g. $X \rightarrow Y \Rightarrow XZ \rightarrow YZ$

- if a dependency holds, then we can freely expand its left hand side

F3. Transitivity e.g. $X \rightarrow Y,\ Y \rightarrow Z \Rightarrow X \rightarrow Z$

- the "most powerful" inference rule; useful in multi-step derivations

F4. Additivity e.g. $X \rightarrow Y, X \rightarrow Z \Rightarrow X \rightarrow YZ$

- useful for constructing new right hand sides of *fd*s (also called union)

F5. Projectivity e.g. $X \rightarrow YZ \Rightarrow X \rightarrow Y, X \rightarrow Z$

- useful for reducing right hand sides of *fd*s (also called decomposition)

F6. Pseudotransitivity e.g. $X \rightarrow Y,\ YZ \rightarrow W \Rightarrow XZ \rightarrow W$

- shorthand for a common transitivity derivation

  - 根据以上关系推导的栗子: 证明$AB \rightarrow GH$

$$R = ABCDEFGHIJ$$

$$F = \{ AB \rightarrow E,\quad AG \rightarrow J,\quad BE \rightarrow I,\quad E \rightarrow G,\quad GI \rightarrow H \}$$

1.	$AB \rightarrow E$	(given)
2.	$AB \rightarrow AB$	(using F1)
3.	$AB \rightarrow B$	(using F5 on 2)
4.	$AB \rightarrow BE$	(using F4 on 1,3)
5.	$BE \rightarrow I$	(given)
6.	$AB \rightarrow I$	(using F3 on 4,5)
7.	$E \rightarrow G$	(given)
8.	$AB \rightarrow G$	(using F3 on 1,7)
9.	$AB \rightarrow GI$	(using F4 on 6,8)
10.	$GI \rightarrow H$	(given)
11.	$AB \rightarrow H$	(using F3 on 6,8)
12.	$AB \rightarrow GH$	(using F4 on 8,11)

9.2 闭包

- 闭包就是由一个属性直接或间接推导出的所有属性的集合
- 定义: 设X和Y均为关系R的属性集的子集，F是R上的函数依赖集，若对R的任一属性集B，一旦$\boldsymbol{X \rightarrow B}$，必有 $\boldsymbol{B \subseteq Y}$，且对R的任一满足以上条件的属性集$\boldsymbol{Y_1}$ ，必有$\boldsymbol{Y \subseteq Y_1}$，此时称Y为属性集X在函数依赖集F下的闭包，记作$\boldsymbol{X^+}$。
- 栗子:
  - $\boldsymbol{f = \{a \rightarrow b, b \rightarrow c, a \rightarrow d, e \rightarrow f\}}$
  - 则a的闭包就是$\boldsymbol{\{a, b, c, d\}}$
- 求取关系R中某个属性集X的闭包:

    - 设最终将成为闭包的属性集是Y，把Y初始化为X
    - 检查F中的每一个函数依赖$A \rightarrow B$，如果属性集A中所有属性均在Y中，而B中有的属性不在Y中，则将其加入到Y中
    - 重复第二步，直到没有属性可以添加到属性集Y中为止, 最后得到的Y就是所需要的闭包
- 举个栗子: 求以下关系的主键(可以推导出属性集的所有属性)

1. $FD = \{A \rightarrow B, C \rightarrow D, E \rightarrow FG\}$
2. $FD = \{A \rightarrow B, B \rightarrow C, C \rightarrow D\}$
3. $FD = \{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$
4. $FD = \{ABH \rightarrow C, A \rightarrow D, C \rightarrow E\ F \rightarrow A, E \rightarrow F, BGH \rightarrow E\}$

    - ACE
    - A
    - A or B or C
    - BGH

9.3 Normalization范式

- 一张数据表的表结构所符合的某种设计标准的级别
- 参考以下两篇文章
    - http://blog.csdn.net/dreamrealised/article/details/10474391
    - https://www.zhihu.com/question/24696366

9.3.1 1NF

- 第一范式: 每个属性都不可再分, 即不允许嵌套表
- 不符合第一范式的栗子

编号	品名	进货		销售		备注
		数量	单价	数量	单价	

9.3.2 2NF

- 第二范式: 在符合第一范式的基础上,非主属性完全函数依赖于主属性, 即不允许partial dependencies存在

- 不符合第二范式的栗子

学生	课程	老师	老师职称	教材	教室	上课时间
小明	一年级语文（上）	大宝	副教授	《小学语文1》	101	14：30

一个学生上一门课，一定在特定某个教室。所以有（学生，课程）->教室

一个学生上一门课，一定是特定某个老师教。所以有（学生，课程）->老师

一个学生上一门课，他老师的职称可以确定。所以有（学生，课程）->老师职称

一个学生上一门课，一定是特定某个教材。所以有（学生，课程）->教材

一个学生上一门课，一定在特定时间。所以有（学生，课程）->上课时间

因此（学生，课程）是一个码。

  - 课程与教材为部分依赖关系(该教材不止这一门课用,该课也可能换教材)
  - 拆分为以下形式:

（学生，课程，老师，老师职称，教室，上课时间）和（课程，教材）

9.3.3 3NF

- 第三范式: 第二范式的基础上, 不允许非主属性通过传递依赖主属性
- 还是上面的栗子: 老师职称依赖于老师, 老师依赖于主属性(学生,课程)
- 拆分为以下形式: 老师变为主属性, 老师职称直接依赖于老师

（学生，课程，老师，教室，上课时间）和（课程，教材）和（老师，老师职称）

9.3.4 BCNF

- BCNF: 第三范式的基础上, 不允许主属性部分依赖或传递依赖于主属性
- 举个栗子: 主属性仓库名, 依赖于主属性(管理员,物品名)

仓库名	管理员	物品名	数量
上海仓	张三	iPhone 5s	30
上海仓	张三	iPad Air	40
北京仓	李四	iPhone 5s	50
北京仓	李四	iPad Mini	60

- 拆分为以下形式

仓库（仓库名，管理员）

库存（仓库名，物品名，数量）

9.3.5 检查范式

9.4 数据库分解

Properties: $R = S \cup T, \quad S \cap T \neq \varnothing$ and $r(R) = s(S) \bowtie t(T)$

+ 失败的分解: lossy decomposition
+ 成功的分解: lossless join decomposition, 即分解后仍可以复原原来的关系

if R is decomposed into S and T, then $Join(S,T) = R$

9.4.1 BCNF分解算法

- 参考http://blog.csdn.net/ristal/article/details/6652020
- 只有当一个数据库中所有表都满足BCNF,该数据库才满足BCNF

```
Inputs: schema R, set F of fds
Output: set Res of BCNF schemas

Res = {R};
while (any schema S ∈ Res is not in BCNF) {
    choose any fd X → Y on S that violates BCNF
    Res = (Res-S) ∪ (S-Y) ∪ XY
}
```

1）置初值ρ={R}；
2）检查ρ中的关系模式，如果均属BCNF，则转4）；
3）在ρ中找出不属于BCNF的关系模式S，那么必有$X \rightarrow A \in F^{+}$，（A不包含于X），且X不是S的关键字。因此XA必不包含S的全部属性。把S分解为$\{S_1, S_2\}$，其中S_1=XA，S2=(S-A)X，并以$\{S_1, S_2\}$代替ρ中的S，返回2）
4）终止分解，输出ρ。

- 举个栗子

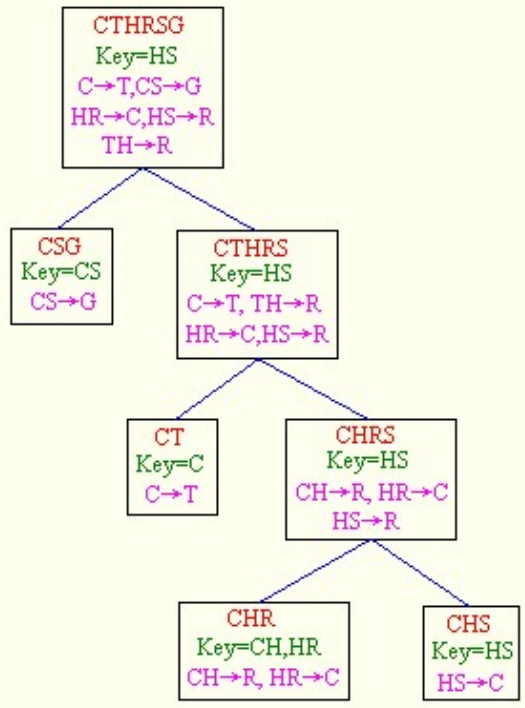


图4.5.1 算法分解树

算法1：存在两个问题：
第一、分解结果不唯一
例：最后一次分解时如果选择HR→C，则分解的最终结果为
CSG、CT、HRC和HRS。
所以分解要结合语义和实际应用来考虑。
第二、分解不保证是保持函数依赖的
例：TH→R未能保持，在分解后各模式的函数依赖的并集中没有逻辑蕴涵TH→R。

- 再举个栗子

例4：关系模式R<U,F>，其中：U={A,B,C,D,E}，F={A→C,C→D,B→C,DE→C,CE→A}，将其分解成BCNF并保持无损连接。

解：

① 令ρ={R(U,F)}。

② ρ中不是所有的模式都是BCNF，转入下一步。

③ 分解R：R上的候选关键字为BE（因为所有函数依赖的右边没有BE）。考虑A→C函数依赖不满足BCNF条件（因A不包含候选键BE），将其分解成R1(AC)、R2(ABDE)。计算R1和R2的最小函数依赖集分别为：F1={A→C}，F2={B→D,DE→D,BE→A}。其中B→D是由于R2中没有属性C且B→C,C→D；DE→D是由于R2中没有属性C且DE→C,C→D；BE→A是由于R2中没有属性C且B→C,CE→A。又由于DE→D是蕴含关系，可以去掉，故F2={B→D,BE→A}。

分解R2：R2上的候选关键字为BE。考虑B→D函数依赖不满足BCNF条件，将其分解成R21(BD)、R22(ABE)。计算R21和R22的最小函数依赖集分别为：F21={B→D}，F22={BE→A}。

由于R22上的候选关键字为BE，而F22中的所有函数依赖满足BCNF条件。故R可以分解为无损连接性的BCNF如：ρ={R1(AC),R21(BD),R22(ABE)}