COMP9311 Database Systems

Author : Yunqiu Xu

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

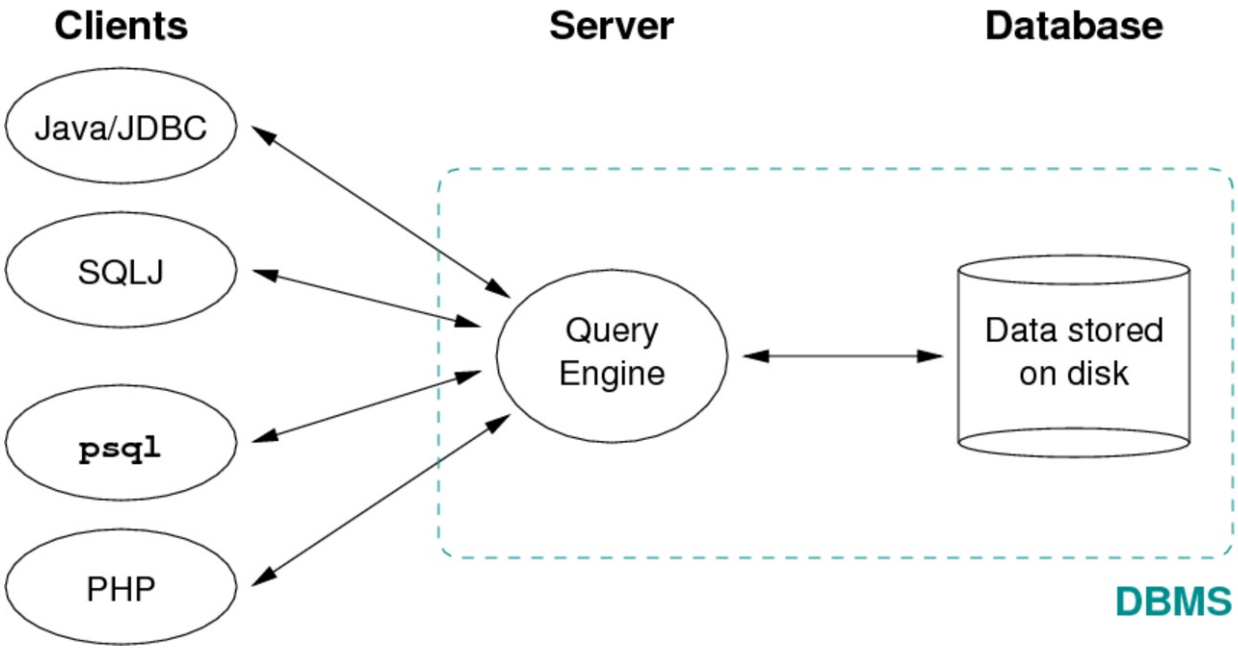
Lecture 1 Introduction, Data Modelling, ER Notation

- Some basic knowledge about database
- Entity-relationship model

1.1 Home Computing

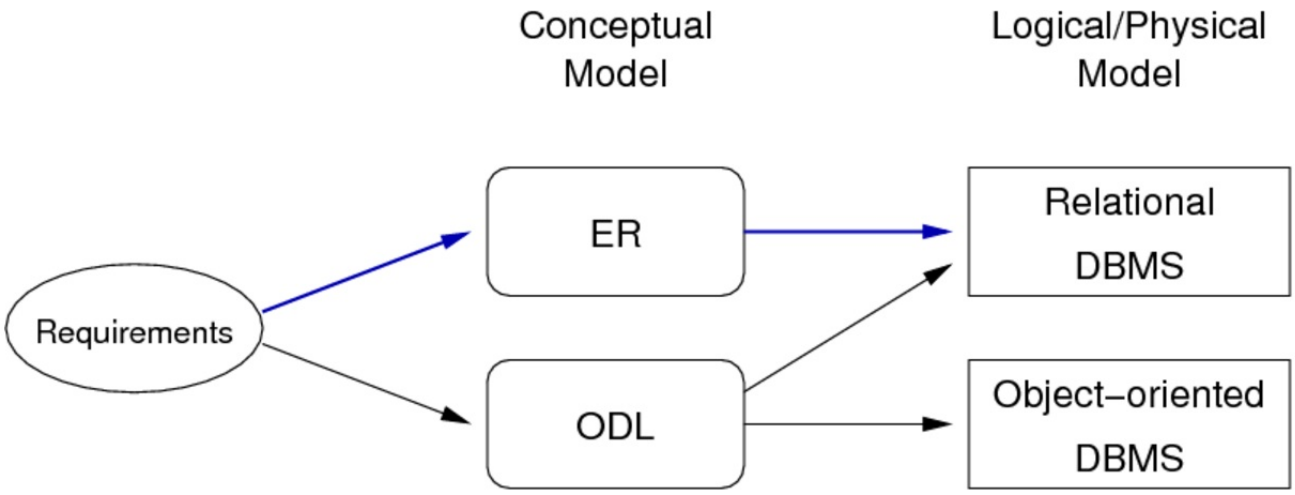
- Software needed: PostgreSQL 9.0 , PHP 5.3 , aPACHE 2
- Alternative: run them on CSE servers(grieg), need VPN

1.2 Typical environment for a modern DBMS



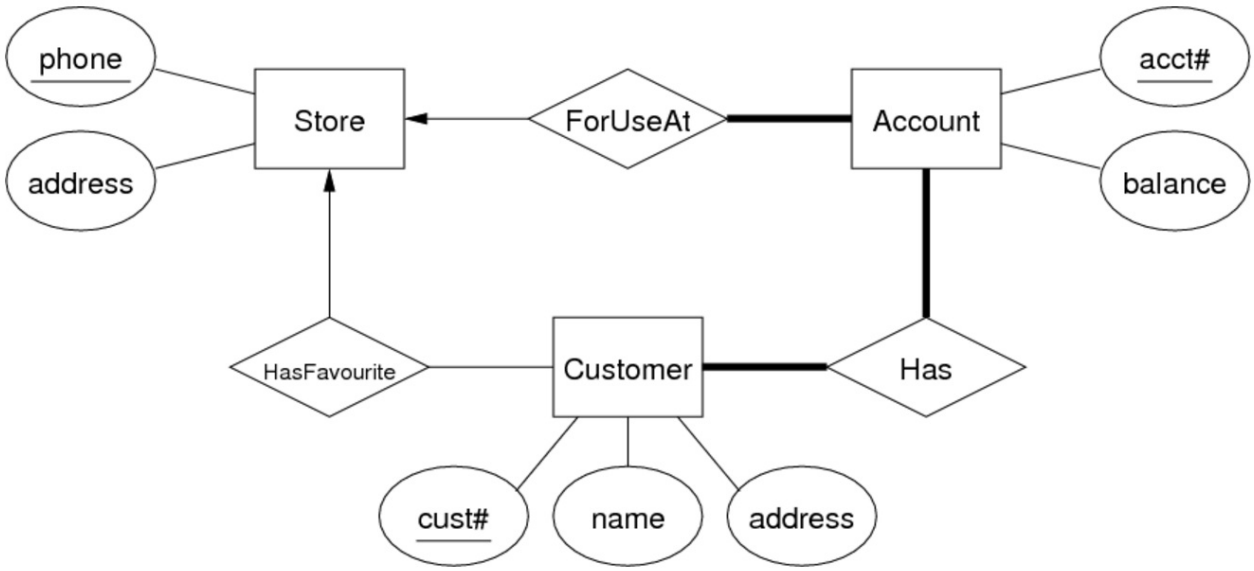
1.3 Data Modelling

- 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

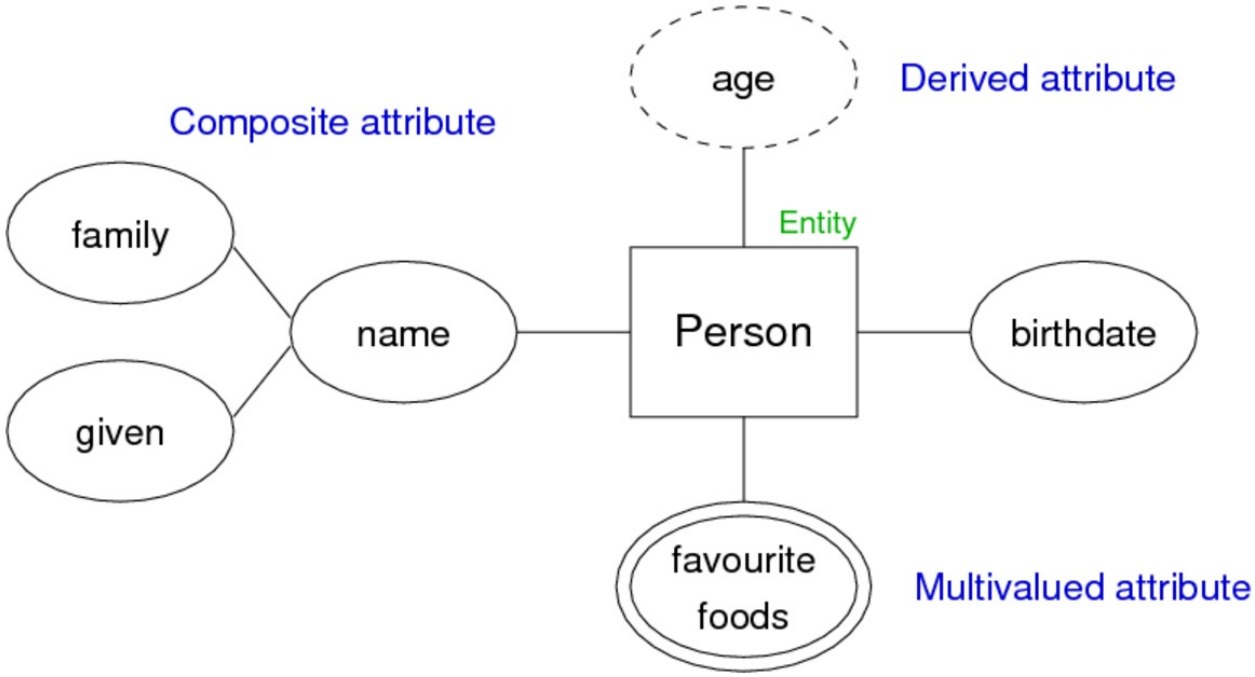


1.3 ER Data Modelling

- 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

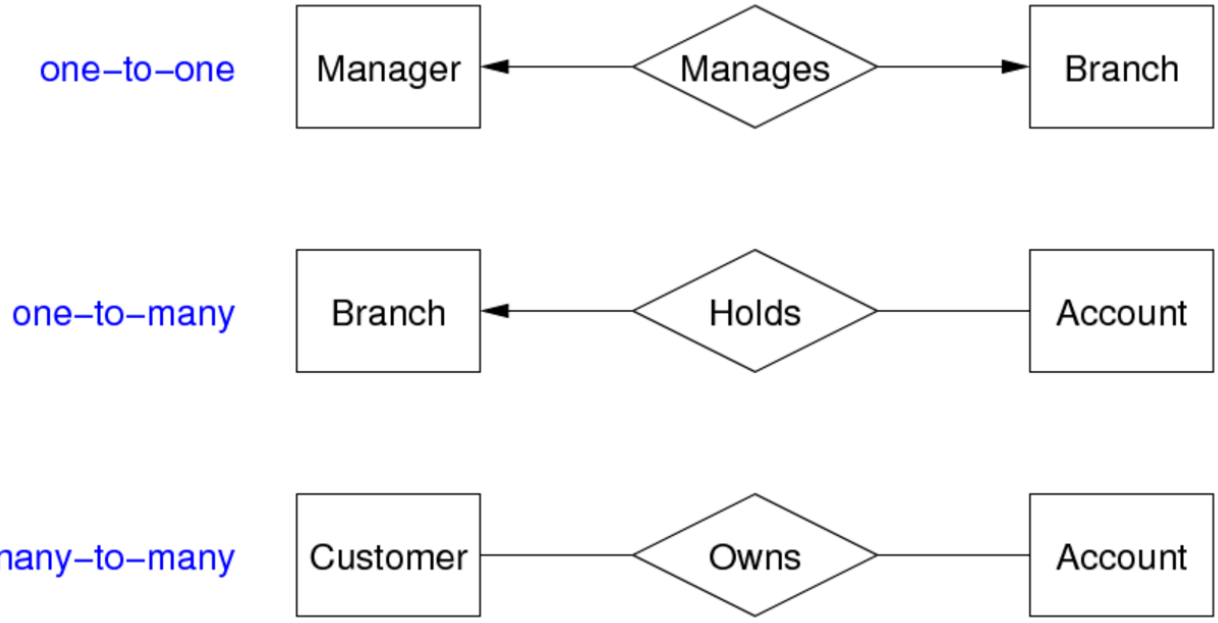
1.3.3 Keys (set of attributes)

- 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

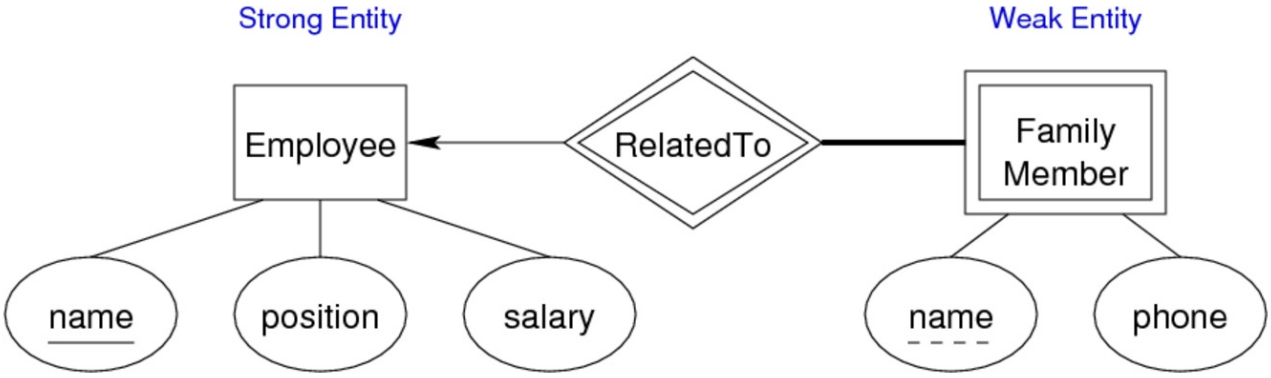
1.3.4 Relationship sets

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

- Cardinality(arrow line)



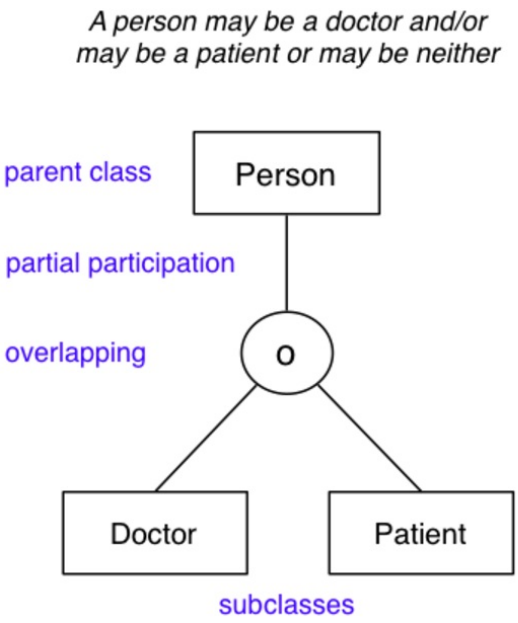
1.3.5 Weak Entity Sets: 需要依赖其他实体才能存在的实体

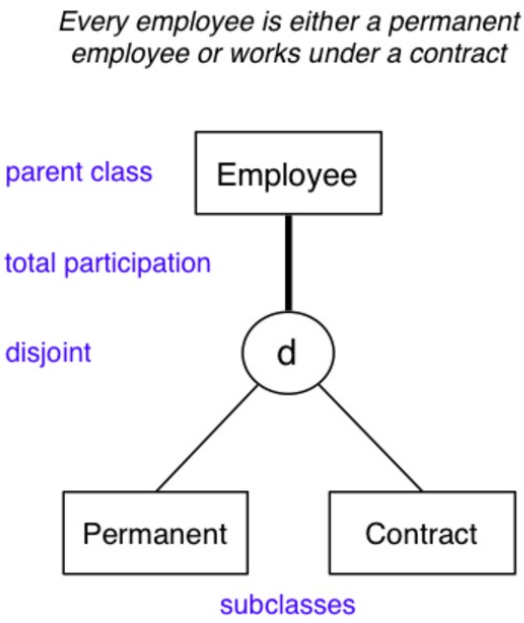


- Analysis:
  - Arrow:一个家庭成员只能有一种工作， 而可能有不止一个家庭成员是同行
  - Thick line: 所有家庭成员都有工作,但不是所有工作都由该家庭成员担任
  - Weak entity: 家庭成员依赖于其他实体集(仅仅靠自有属性无法标识主码)
  - dashed line: 弱实体没有主键, 'name' is discriminator

1.3.6 Subclasses and Inheritance

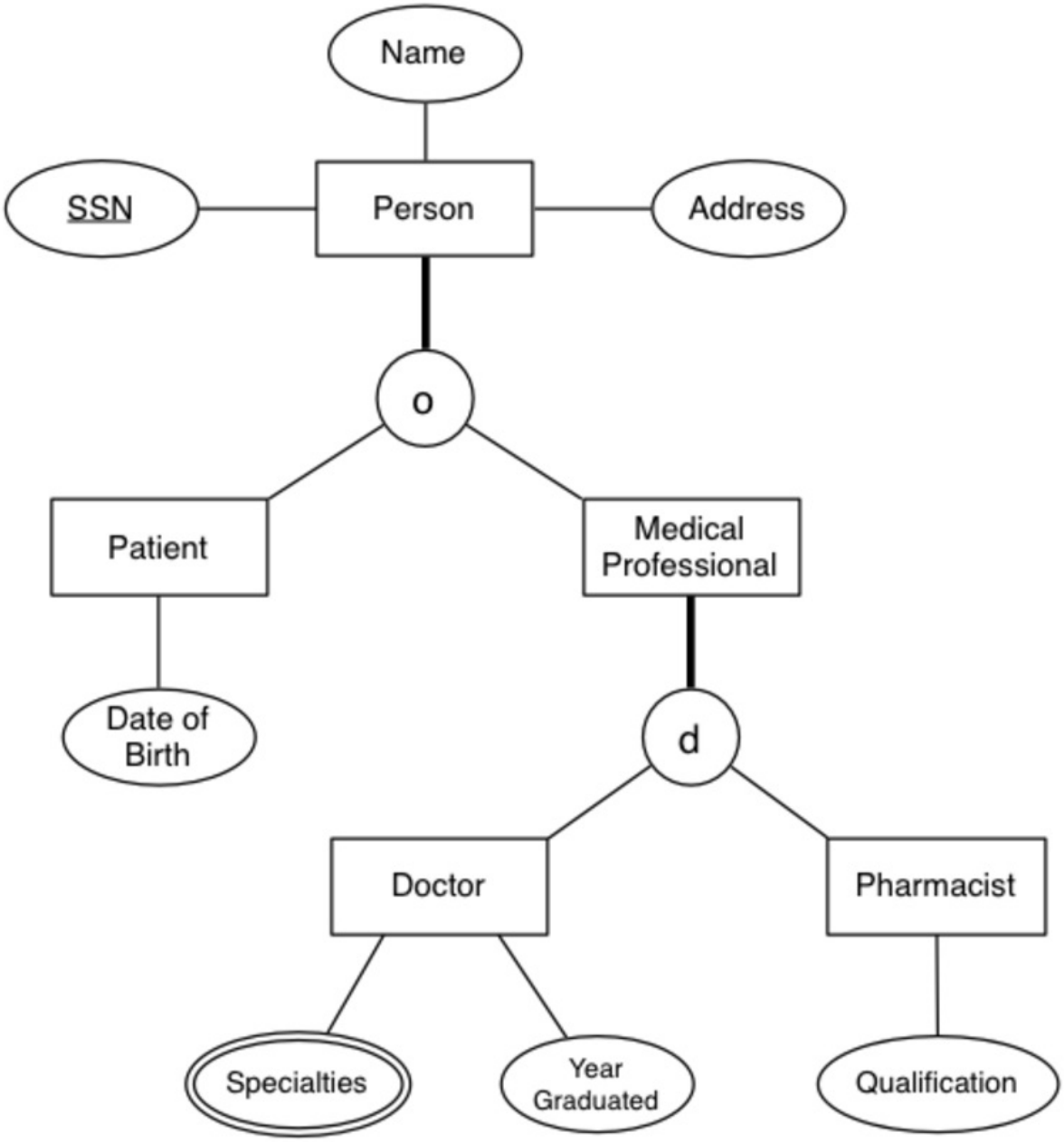
- overlapping/disjoint
- total/partial





1.3.7 最后举一个比较复杂的栗子

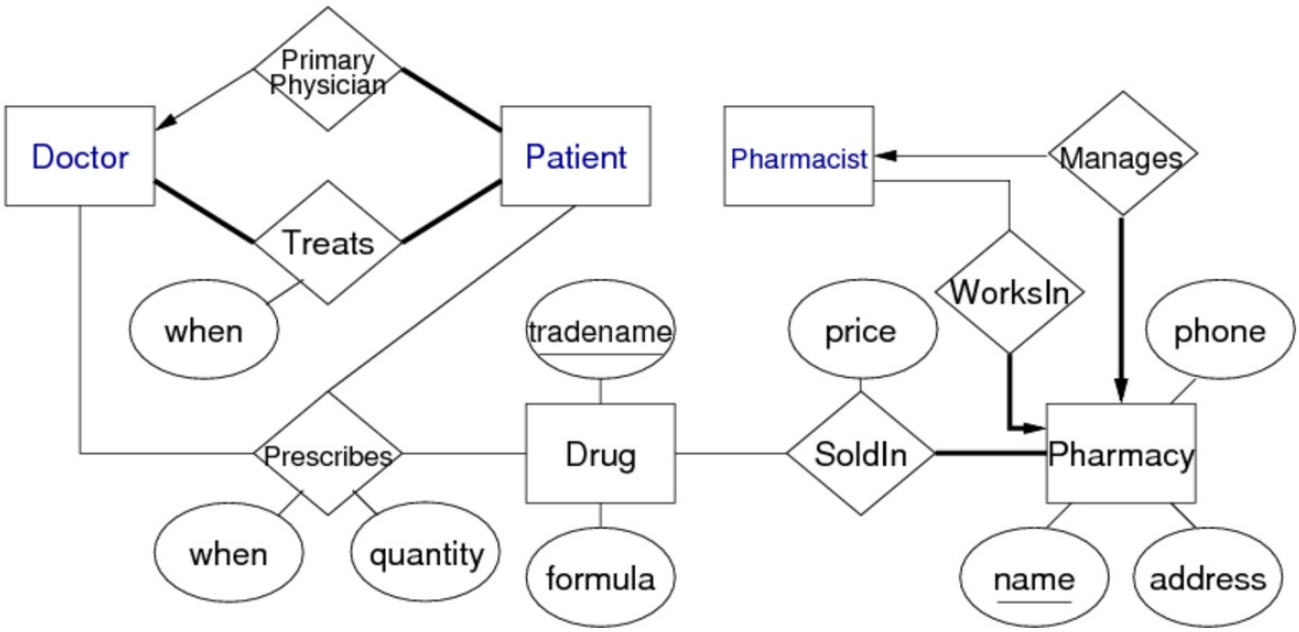
People subclasses



- **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)

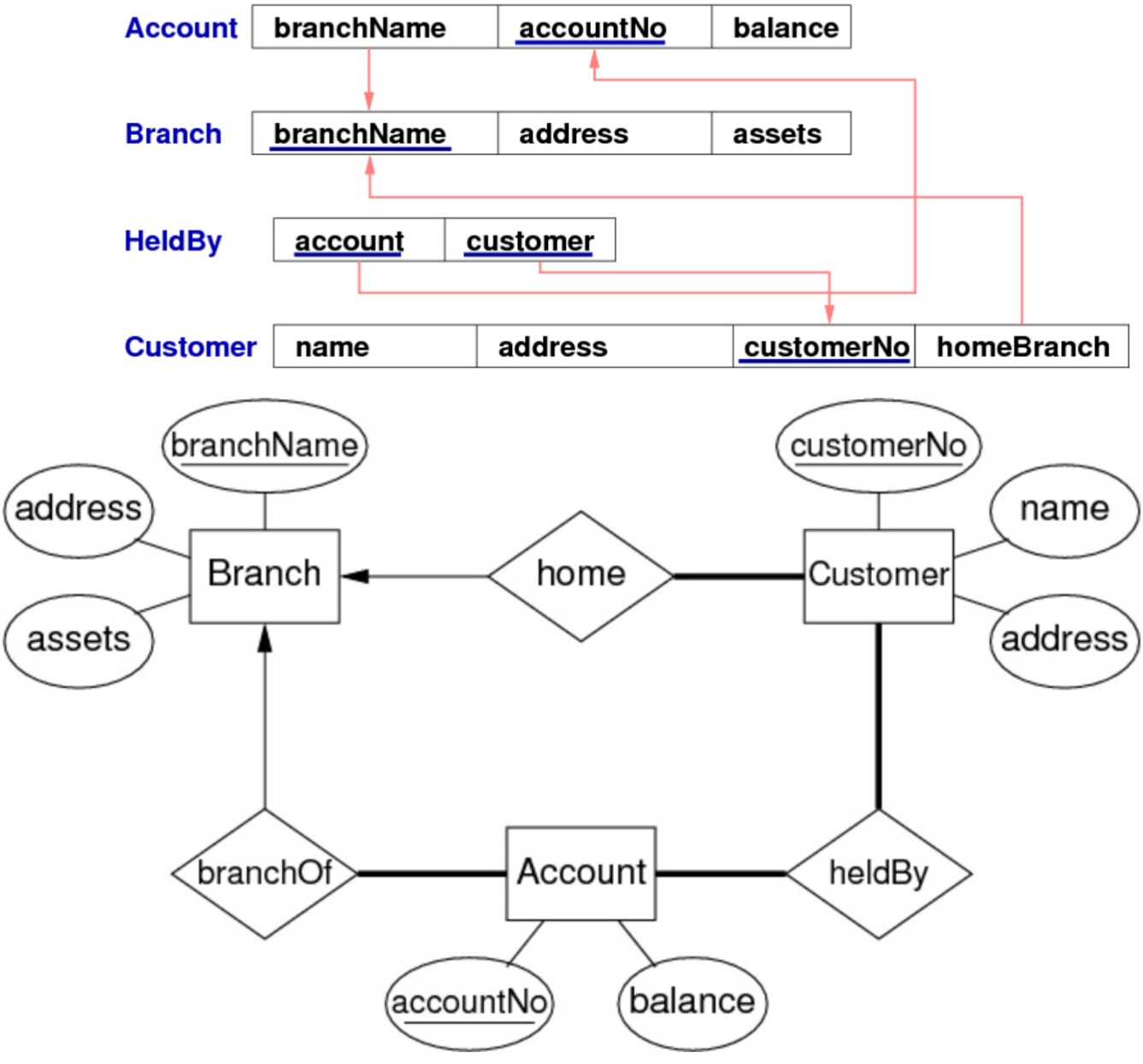
2.2 Relational Data Model

- 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:

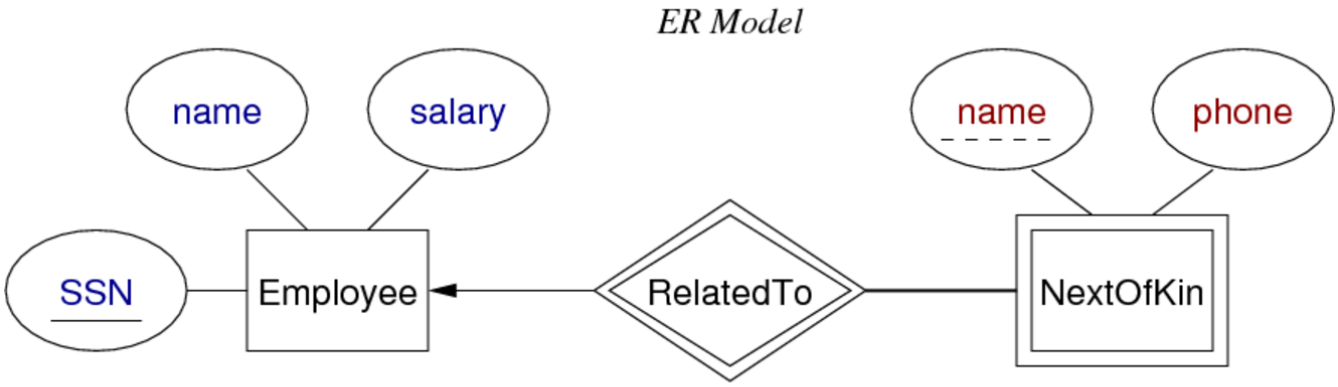
- attribute(ER) ≅ attribute(Rel), entity(ER) ≅ tuple(Rel)
- entity set(ER) ≅ relation(Rel), relationship(ER) ≅ relation(Rel)

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

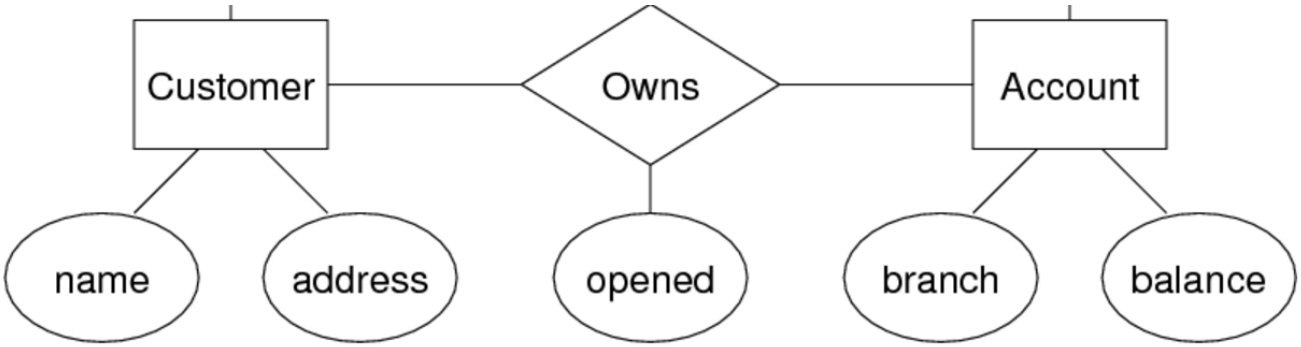
SSN	name	salary

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**

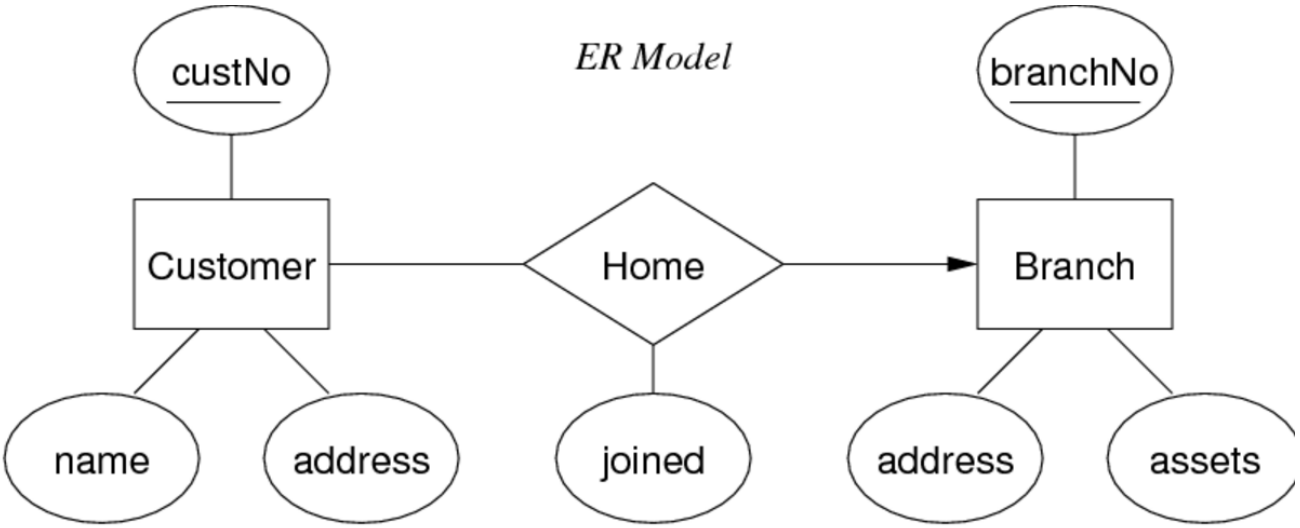
Account

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

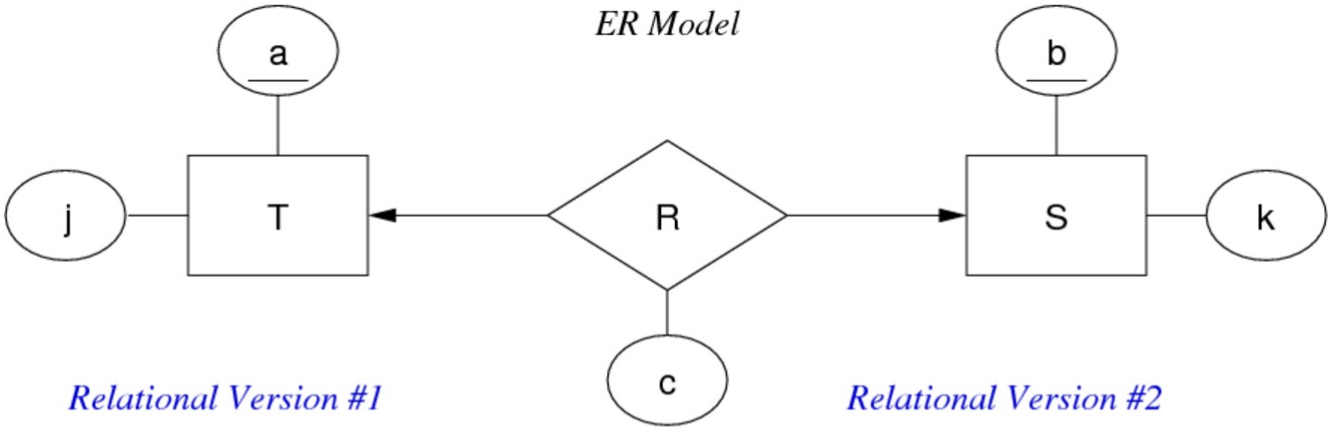
Customer

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,...);
```

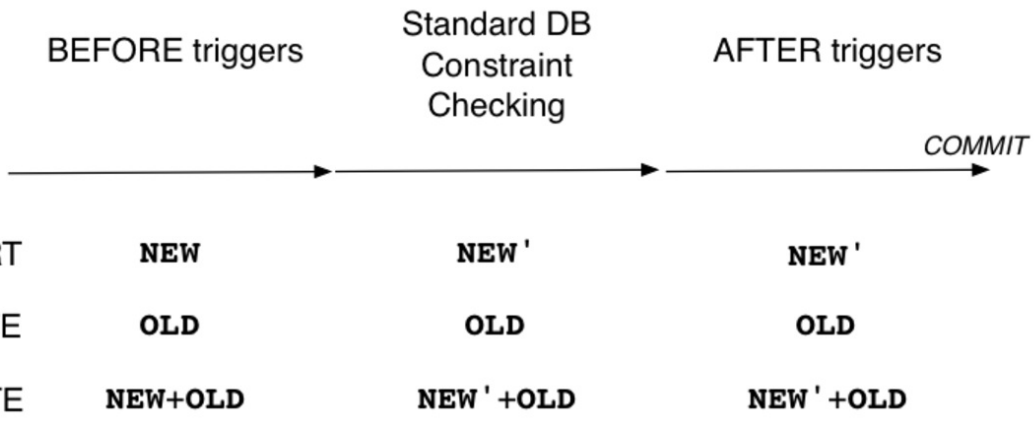
- UPDATE: 假设原表为{1,2,3},更新为{666,2,3}
  - 更新前X被激活, NEW-->666, OLD-->1;
  - X先检查NEW后检查OLD,此时NEW的值还是可以再变的;
  - X检查完成后, DBMS在更新数据时对NEW进行约束检查;
  - 更新完成后Y被激活, 检查NEW(666);

```
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:

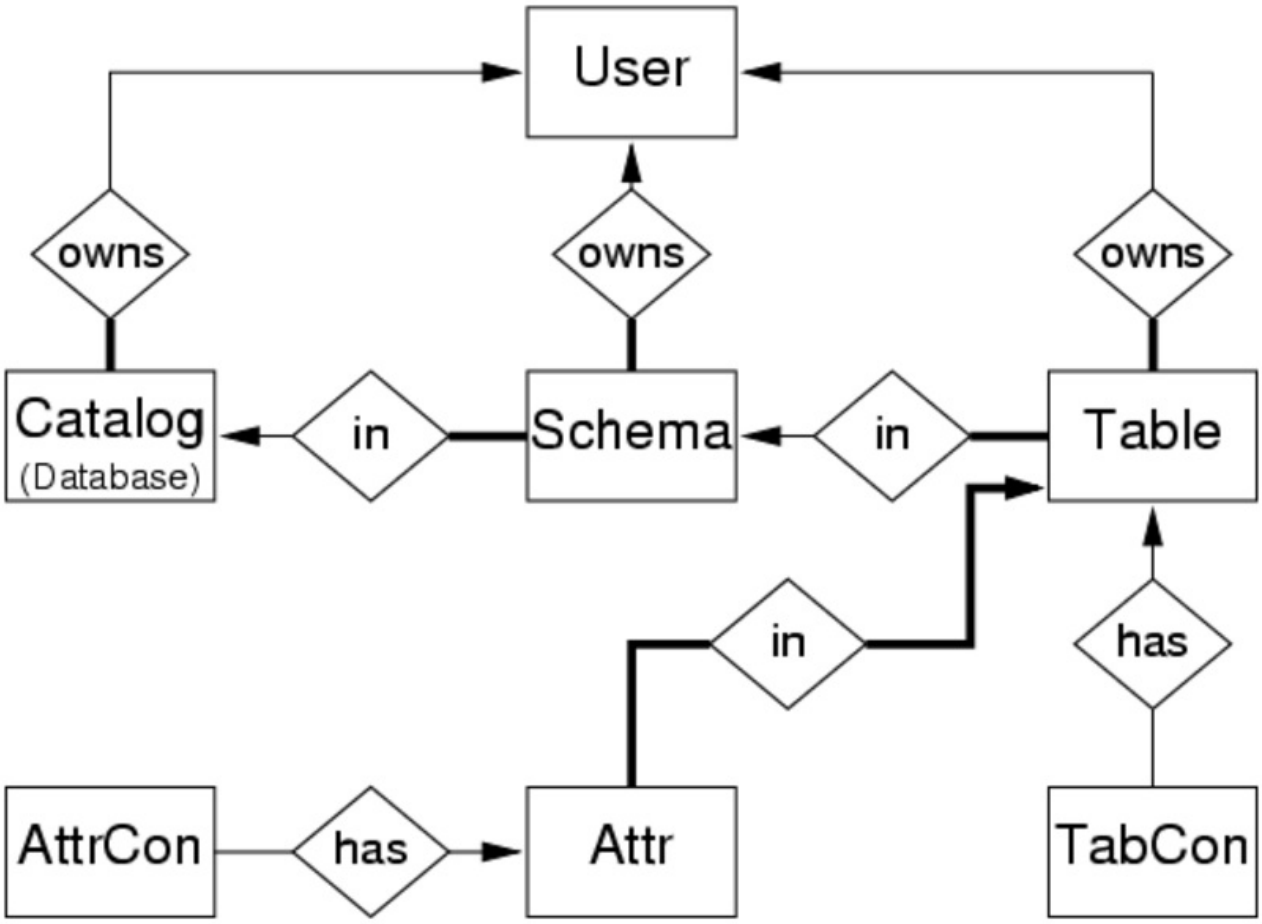
```
# 低效率：在循环中还要进行多次查询
$query1 = "select id,name from Student";
$res1 = dbQuery($db,$query1);
while ($tuple1 = dbNext($res1)) {
    $query2 = "select course,mark from Marks".
              " where student = $tuple1['id']";
    $res2 = dbQuery($db,$query2);
    while ($tuple2 = dbNext($res2)) {
        --  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$X \rightarrow B$，必有 $B \subseteq Y$，且对R的任一满足以上条件的属性集Y_1 ，必有$Y \subseteq Y_1$，此时称Y为属性集X在函数依赖集F下的闭包，记作X^+。
- 栗子:
  - $f = \{a \rightarrow b, b \rightarrow c, a \rightarrow d, e \rightarrow f\}$
  - 则a的闭包就是$\{a, b, c, d\}$
- 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。

  - 但是这里非主属性教材仅仅依赖于课程:对于同一门课, 不同学生需要的教材是一样的
  - 导致的后果: 一旦需要修改教材,所有包含这门课程的数据(所有学生)都要进行修改
  - 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

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

9.3.6 总结下选择候选键并根据候选键判断范式的方法

- 候选键为可推导出所有依赖属性的主键集合, 但这个集合的真子集无法推导出全部依赖属性
  - 若某属性从未被依赖(只出现在右边), 该属性肯定不是候选键一部分
  - 若某属性从未依赖于其他属性(只出现在左边), 该属性肯定是某个候选键一部分(但不是所有)
  - 外部属性: 一定会出现在所有候选键中
  - 其他属性逐个和以上一定会出现在候选键中的属性结合, 求闭包
- 判断BCNF:
  - 若某依赖左侧不是某一个候选键, 则该依赖不符合BCNF
  - 证明: 以上依赖的左侧主属性为候选键的一部分或推导而来, 违背了不允许主属性部分依赖或传递依赖于主属性的原则
- 判断3NF: 两种情况可以认定3NF成立
  - 左边就是候选键
  - 左边不是候选键但右边是候选键一部分(注意这个不满足BCNF)
  - 若左边不是候选键, 右边也不是候选键一部分, 则左边主属性部分依赖或传递依赖于主属性, 右边为非主属性, 违背了不允许非主属性传递依赖于主属性的原则
- 举几个栗子

i. $C \rightarrow D,\ C \rightarrow A,\ B \rightarrow C$

[hide answer]

a. Candidate keys: B
b. Not BCNF ... e.g. in $C \rightarrow A$, C does not contain a key
c. Not 3NF ... e.g. in $C \rightarrow A$, C does not contain a key, A is not part of a key

ii. $B \rightarrow C,\ D \rightarrow A$

[hide answer]

a. Candidate keys: BD
b. Not 3NF ... neither right hand side is part of a key
c. Not BCNF ... neither left hand side contains a key

iii. $ABC \rightarrow D,\ D \rightarrow A$

[hide answer]

a. Candidate keys: ABC BCD
b. 3NF ... $ABC \rightarrow D$ is ok, and even $D \rightarrow A$ is ok, because A is a single attribute from the key
c. Not BCNF ... e.g. in $D \rightarrow A$, D does not contain a key

iv. $A \rightarrow B,\ BC \rightarrow D,\ A \rightarrow C$

[hide answer]

a. Candidate keys: A
b. Not 3NF ... e.g. in $A \rightarrow C$, C is not part of a key
c. Not BCNF ... e.g. in $BC \rightarrow D$, BC does not contain a key

v. $AB \rightarrow C,\ AB \rightarrow D,\ C \rightarrow A,\ D \rightarrow B$

[hide answer]

a. Candidate keys: AB BC CD AD
b. 3NF ... for AB case, first two fd's are ok, and the others are also ok because the RHS is a single attribute from the key
c. Not BCNF ... e.g. in $C \rightarrow A$, C does not contain a key

vi. $A \rightarrow BCD$

[hide answer]

a. Candidate keys: A
b. 3NF ... all left hand sides are superkeys
c. BCNF ... all left hand sides are superkeys

9.4 数据库分解

Properties: $R = S \cup T,\ S \cap T \neq \varnothing$ 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

面向BCNF且具有无损联接性的分解

输入：关系模式R及其函数依赖集F。
输出：R的一个无损联接分解，其中每一个子关系模式都满足
F在其上投影的BCNF。

算法实现：
反复运用逐步分解定理，逐步分解关系模式R，
使得每次分解都具有无损联接性，而且
每次分解出来的子关系模式至少有一个是BCNF的，
即：
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: 最后那个集合里所有的依赖都包含B, 满足BCNF

  $R = ABCDEFGH$
  $F = \{ ABH \rightarrow C,\ A \rightarrow DE,\ BGH \rightarrow F,\ F \rightarrow ADH,\ BH \rightarrow GE \}$

  Produce a BCNF decomposition of *R*.

  [hide answer]

  This is just one of many possible decompositions. Variations occur because of choice of candidate key (although not in this example) and choice of non-BCNF FD to resolve at each step.

  - We start from a schema: *ABCDEFGH*, with key *BH* (work it out from FDs).
  - The FD $A \rightarrow DE$ violates BCNF (FD with non key on LHS).
  - To fix, we need to decompose into tables: *ADE* and *ABCFGH*.
  - FDs for *ADE* are $\{ A \rightarrow DE \}$, therefore key is *A*, therefore BCNF.
  - FDs for *ABCFGH* are $\{ ABH \rightarrow C,\ BGH \rightarrow F,\ F \rightarrow AH,\ BH \rightarrow G \}$
  - Key for *ABCFGH* is *BH*, and FD $F \rightarrow AH$ violates BCNF (FD with non key on LHS).
  - To fix, we need to dcompose into tables: *AFH* and *BCFG*.
  - FDs for *ADE* are $\{ F \rightarrow AH \}$, therefore key is *F*, therefore BCNF.
  - FDs for *BCFG* are $\{\}$, so key is *BCFG* and table is BCNF.
  - Final schema (with keys boldened): ***ADE***, ***FAH***, ***BCFG***

- 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)}

9.4.2 3NF分解算法

- 参考http://blog.sina.com.cn/s/blog_457ac27b0100lppg.html

面向**3NF**且保持函数依赖的分解

输入：关系模式R及其上的最小函数依赖集F。
输出：R的保持函数依赖的分解，其中每一个关系模式是关于F在其上投影的**3NF**。

算法实现：

1)如果R中存在一些不在F中出现的属性，
则将它们单独构成一个关系模式，并从模式R中消去；
2)如果F中有一个函数依赖X→A，且XA=R，则R不用分解，
算法终止；
3)对F中的每一个函数依赖X→A，构造一个关系模式XA。
如果$X\rightarrow A_1, X\rightarrow A_2, \ldots, X\rightarrow A_n$均属于F，则构造一个关系模式$XA_1A_2\ldots A_n$。

- 举个栗子

$R = ABCDEFGH$
$F = \{ ABH \rightarrow C,\ A \rightarrow D,\ C \rightarrow E,\ BGH \rightarrow F,\ F \rightarrow AD,\ E \rightarrow F,\ BH \rightarrow E \}$
$F_c = \{ BH \rightarrow C,\ A \rightarrow D,\ C \rightarrow E,\ F \rightarrow A,\ E \rightarrow F,\ BH \rightarrow E \}$

Produce a 3NF decomposition of R.

[hide answer]

3NF is constructed directly from the minimal cover, after combinining dependencies with a common right hand side where possible.

$F_c = BH \rightarrow C,\ A \rightarrow D,\ C \rightarrow E,\ F \rightarrow A,\ E \rightarrow F$

Gives the following tables (with table keys in **bold**):

BHC ***A***D ***C***E ***F***A ***E***F

A key for R is BHG; G must be included because it appears in no function dependency.

Since no table contains the whole key for R, we must add a table containing just the key, giving:

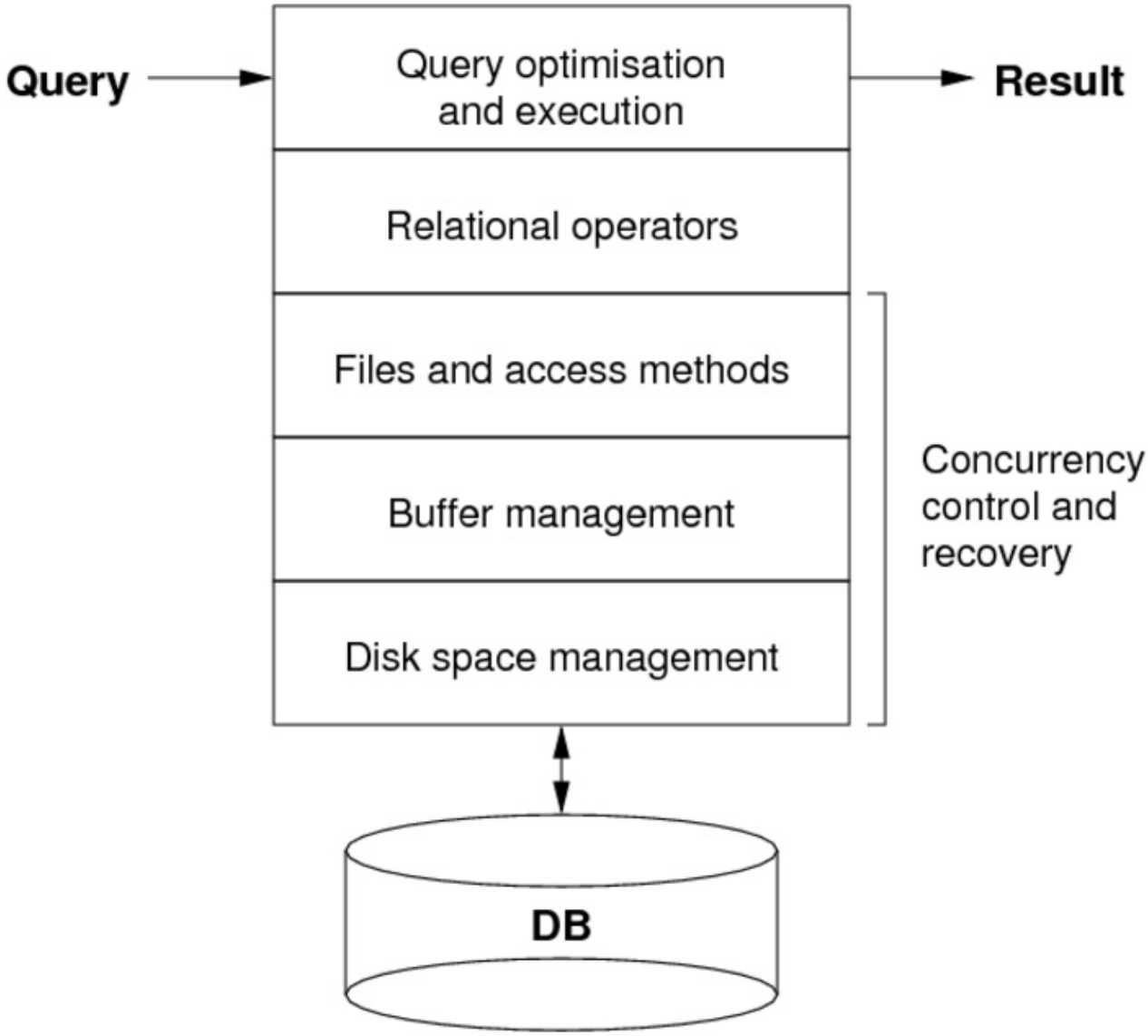
3NF = ***BH***C ***A***D ***C***E ***F***A ***E***F ***BGH***

- 如果存在属性只在左侧出现, 该属性肯定为候选键一部分, 先把含该属性的依赖提出来, 剩下的依赖集为Fc
- 对Fc执行3NF分解, 得到的结果添加之前提出来的那些依赖的左侧(对没错只要增加左侧就行)

Lecture 10 Relational Algebra, Query Processing

10.1 数据库架构与实现

- QP: query processing, 基于关系代数, 令数据库更高效
- TxP: transaction processing, 令数据库更安全
- Layers in a DB Engine



- DBMS Components

组件名称	作用
File manager	manages allocation of disk space and data structures
Buffer manager	manages data transfer between disk and main memory
Query optimiser	translates queries into efficient sequence of relational ops
Recovery manager	ensures consistent database state after system failures
Concurrency manager	controls concurrent access to database
Integrity manager	verifies integrity constraints and user privileges

10.2 数据库性能

- what operations on the data does the application require

  (which queries, updates, inserts and how frequently is each one performed)

- how these operations might be implemented in the DBMS

  (data structures and algorithms for select, project, join, sort, ...)

- how much each implementation will cost

  (in terms of the amount of data transferred between memory and disk)

- 影响查询效率的因素:
  - `join`比子查询效率更高
  - 尽可能避免创建大规模中间表然后进行过滤
  - 尽可能避免在`where/group by`语句中使用函数
- 在表中创建indexes:
  - 有助于提升过滤速度
  - 一般只对equality/gt/lt有效
  - 更新和存储需要额外开支
  - 适用情形: 过滤操作远多于更新
- Query optimiser: 用于评估查询效率
- 举个栗子: 查询工资高于50K的销售人员
  - Method 1: examine all employees, even if not in Sales

```
select name from Employee
where  salary > 50000 and
       empid in (select empid from Worksin
                 where  dept = 'Sales')
```

A query optimiser might use the strategy

```
SalesEmps = (select empid from WorksIn where dept='Sales'
foreach e in Employee {
    if (e.empid in SalesEmps && e.salary > 50000)
        add e to result set
}
```

- Method 2: Only examines Sales employees, and uses a simpler test

```
select name
from   Employee join WorksIn using (empid)
where  Employee.salary > 5000 and
       WorksIn.dept = 'Sales'
```

Query optimiser might use the strategy

```
SalesEmps = (select * from WorksIn where dept='Sales')
foreach e in (Employee join SalesEmps) {
    if (e.salary > 50000)
        add e to result set
}
```

- Method 3: Needs to run a query for every people

```
select name from Employee e
where  salary > 50000 and
       'Sales' in (select dept from Worksin where empid=e.id)
```

A query optimiser would be forced to use the strategy:

```
foreach e in Employee {
    Depts = (select dept from WorksIn where empid=e.empid)
    if ('Sales' in Depts && e.salary > 50000)
        add e to result set
}
```

10.3 关系代数

10.3.1 基本概念

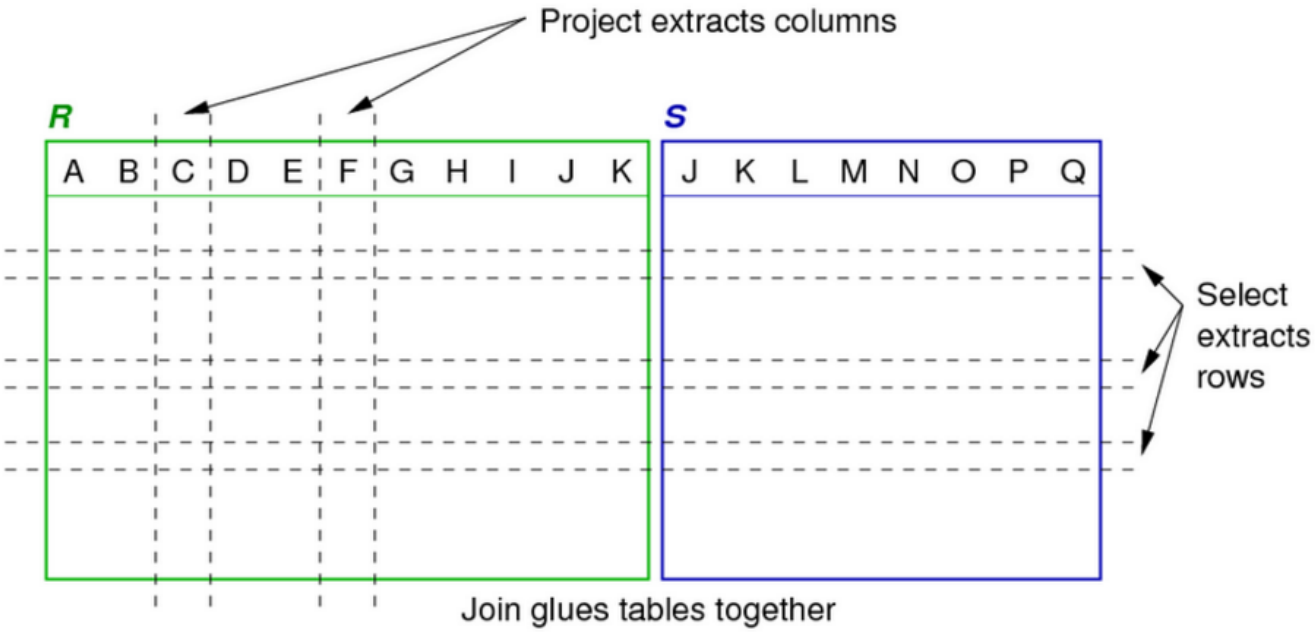
Relational algebra (RA) can be viewed as ...

- mathematical system for manipulating relations, or
- data manipulation language (DML) for the relational model

Relational algebra consists of:

- operands: relations, or variables representing relations
- operators that map relations to relations
- rules for combining operands/operators into expressions
- rules for evaluating such expressions

- 核心操作:
  - selection: choosing a subset of **rows**
  - projection: choosing a subset of **columns**
  - product/join/union/intersection/difference: combining relations
  - rename: change names of relations/attributes



Adding set operations and renaming makes RA complete.

- 表示方式

Operation	Standard Notation	Our Notation
Selection	$\sigma_{expr}(Rel)$	*Sel[expr](Rel)*
Projection	$\pi_{A,B,C}(Rel)$	*Proj[A,B,C](Rel)*
Join	$Rel_1 \bowtie_{expr} Rel_2$	Rel_1 *Join[expr]* Rel_2
Rename	$\rho_{schema}Rel$	*Rename[schema](Rel)*

10.3.2 Describing RA Operations

We define the semantics of RA operations using

- "conditional set" expressions e.g. *{ X | condition on X }*
- tuple notations:
  - *t[AB]* (extracts attributes *A* and *B* from tuple *t*)
  - *(x,y,z)* (enumerated tuples; specify attribute values)
- quantifiers, set operations, boolean operators

For each operation, we also describe it operationally:

- give an algorithm to compute the result, tuple-by-tuple

- 所有的RA操作符都需要返回特定的关系(就是表啦)类型

```
Temp = R op1 S op2 T
Res  = Temp op3 Z
-- which is equivalent to
Res  = (R op1 S op2 T) op3 Z
```

- Rename:

$$Rename[S(B_1, B_2, \ldots B_n)](E)$$

  - 返回关系S, 包含E中的内容

- 区别在于将E中的属性名改为B1,B2,...Bn

- Selection:

$$\sigma_C(r) = Sel[C](r) = \{ t \mid t \in r \wedge C(t) \}$$

  - 返回关系r(R)中满足条件C的子集(行)
  - 返回格式与r相同

```
result = {}
for each tuple t in relation r
    if (C(t)) { result = result ∪ {t} }
```

  - 栗子

Sel [B = 1] (r1)

A	B	C	D
a	1	x	4
e	1	y	4

Sel [A=b ∨ A=c] (r1)

A	B	C	D
b	2	y	5
c	4	z	4

Sel [B ≥ D] (r1)

A	B	C	D
c	4	z	4
d	8	x	5

Sel [A = C] (r1)

A	B	C	D

- Projection:

$$\pi_X(r) = Proj[X](r) = \{ t[X] \mid t \in r \}, \quad \text{where } r(R)$$

  - 返回原表的部分属性(列)
  - X为R中属性集的子集
  - 注意如果主属性不包含在X中, 会产生重复, 但RA会在返回结果中去除重复, 因此行数可能也会减少

```
result = {}
for each tuple t in relation r
    result = result ∪ {t[X]}
```

- 栗子:

Proj [A,B,D] (r1)

A	B	D
a	1	4
b	2	5
c	4	4
d	8	5
e	1	4
f	2	5

Proj [B,D] (r1)

B	D
1	4
2	5
4	4
8	5

Proj [D] (r1)

D
4
5

- Union:

$$r_1 \cup r_2 = \{ t \mid t \in r_1 \vee t \in r_2 \}, \quad \text{where } r_1(R), r_2(R)$$

  - 将两个兼容表合并为一个
  - 兼容性: same schema
  - 由于存在重复值, 行数可能会减少

```
result = r1
for each tuple t in relation r2
    result = result ∪ {t}
```

- Intersection:

$$r_1 \cap r_2 = \{ t \mid t \in r_1 \wedge t \in r_2 \}, \quad \text{where } r_1(R), r_2(R)$$

  - 将两个兼容表取交集

```
result = {}
for each tuple t in relation r1
    if (t ∈ r2) { result = result ∪ {t} }
```

- Difference:

$$r_1 - r_2 = \{ t \mid t \in r_1 \wedge \neg t \in r_2 \}, \quad \text{where } r_1(R), r_2(R)$$

  - 返回只存在于表一中的行

```
result = {}
for each tuple t in relation r1
    if (!(t ∈ r2)) { result = result ∪ {t} }
```

  - 栗子

s1 = Sel [B = 1] (r1)

A	B	C	D
a	1	x	4
e	1	y	4

s2 = Sel [C = x] (r1)

A	B	C	D
a	1	x	4
d	8	x	5

s1 - s2

A	B	C	D
e	1	y	4

s2 - s1

A	B	C	D
d	8	x	5

- Product:

$$r \times s = \{ (t_1 : t_2) \mid t_1 \in r \wedge t_2 \in s \}, \quad \text{where } r(R), s(S)$$

  - 返回两个表的笛卡儿积
  - 和之前并集交集等不同,两个表不需要兼容
  - 返回结果大小为两个表大小的乘积, 格式为两个表的并集

```
result = {}
for each tuple t1 in relation r
    for each tuple t2 in relation s
        result = result ∪ {(t1:t2)} }
```

- Natural Join:
  - 排除相同的属性(列)

    Consider relation schemas $R(ABC..JKLM)$, $S(KLMN..XYZ)$.

    The natural join of relations $r(R)$ and $s(S)$ is defined as:

    $$r \bowtie s \;=\; r \text{ Join } s \;=$$
    $$\{\, (t_1[ABC..J] : t_2[K..XYZ]) \mid t_1 \in r \wedge t_2 \in s \wedge match \,\}$$

    where $match \;=\; t_1[K] = t_2[K] \wedge t_1[L] = t_2[L] \wedge t_1[M] = t_2[M]$

```
result = {}
for each tuple t1 in relation r
   for each tuple t2 in relation s
      if (matches(t1,t2))
         result = result ∪ {combine(t1,t2)}
```

  - 另一种定义方式: 假定在并操作中已经排除了重复属性

    $$r \text{ Join } s \;=\; Proj[R \cup S]\,(\,Sel[match]\,(\,r \times s)\,)$$

  - 栗子

Example (assuming that *A* and *F* are the common attributes):

r1 Join r2

A	B	C	D	E	G
a	1	x	4	1	x
a	1	x	4	4	y
b	2	y	5	2	y
b	2	y	5	5	x
c	4	z	4	3	x

Strictly the above was: *r1 Join Rename[r2(E,A,G)](r2)*

- Theta Join:

$$r \bowtie_C s = \{ (t_1 : t_2) \mid t_1 \in r \wedge t_2 \in s \wedge C(t_1 : t_2) \},$$
where *r(R),s(S)*

Examples: *(r1 Join[B>E] r2) ... (r1 Join[E<D∧C=G] r2)*

  - 给定条件的Join
  - 所有属性名都要不同
  - 另一种定义方式:

$$r \bowtie_C s = r \; Join[C] \; s = Sel[C] \; (r \times s)$$

  - 栗子:

$$r1\ Join[D < E]\ r2$$

A	B	C	D	E	F	G
a	1	x	4	5	b	x
c	4	z	4	5	b	x
e	1	y	4	5	b	x

$$r1\ Join[B > 1 \wedge D < E]\ r2$$

A	B	C	D	E	F	G
c	4	z	4	5	b	x

- Division:

$$r / s \quad = \quad r\ Div\ s \quad = \quad \{\, t \mid t \in r[R\text{-}S] \wedge satisfy \,\}$$

where $satisfy \quad = \quad \forall\, t_s \in S\ (\ \exists\, t_r \in R\ (\ t_r[S] = t_s \wedge t_r[R\text{-}S] = t\ )$

  - 这个比较复杂, 用栗子说明下

R

A	B
4	x
4	y
4	z
5	x
5	y
5	z

R'

A	B
4	x
4	y
4	z
5	x
5	z

S

B
x
y

R / S

A
4
5

R' / S

A
4

- S中x,y均对应R中4,5: 返回4,5
- S中x,y均对应R'中4: 返回4

◦ 再来个比较实际的栗子:

E.g. Which beers are sold in all bars?

We can answer this as follows:

- generate a relation of beers and bars where they are sold
- generate a relation of all bars
- find which beers appear in tuples with every bar

RA operations for answering the query:

- beers and bars where they are sold (ignore prices)
  - *r1 = Proj[beer,bar](Sold)*
- bar names
  - *r2 = Rename[r2(bar)](Proj[name](Bars))*
- beers that appear in tuples with every bar
  - *res = r1 Div r2*

- 综合栗子1 : $(R \times S) - (Sel[R.C = S.C](RJoin[B = B]S)$

Tmp1 = *R* Join[R.B=S.B] *S* =

R.A	R.B	R.C	S.B	S.C
a1	b1	c1	b1	c1
a1	b2	c2	b2	c2
a2	b1	c1	b1	c1

Tmp2 = Sel[R.C=S.C](*Tmp1*) =

R.A	R.B	R.C	S.B	S.C
a1	b1	c1	b1	c1
a1	b2	c2	b2	c2
a2	b1	c1	b1	c1

Tmp3 = *R* × *S* =

R.A	R.B	R.C	S.B	S.C
a1	b1	c1	b1	c1
a1	b1	c1	b2	c2
a1	b2	c2	b1	c1
a1	b2	c2	b2	c2
a2	b1	c1	b1	c1
a2	b1	c1	b2	c2

Tmp3 - Tmp2 =

R.A	R.B	R.C	S.B	S.C
a1	b1	c1	b2	c2
a1	b2	c2	b1	c1
a2	b1	c1	b2	c2

- 综合栗子2 : 求最大最小容量, N1>N2>0

Expression	Assumptions	Min	Max
R1 Union *R2*	*R1* and *R2* are union-compatible	*N1* (when $R2 \subset R1$)	*N1+N2* (when $R1 \cap R2 = \varnothing$)
R1 Intersect *R2*	*R1* and *R2* are union-compatible	*0* (when $R1 \cap R2 = \varnothing$)	*N2* *(when $R2 \subset R1$)*
R1 - *R2*	*R1* and *R2* are union-compatible	*N1-N2* (when $R2 \cap R1 = R2$)	*N1* (when $R1 \cap R2 = \varnothing$)
R1 × *R2*	None	*N1*N2*	*N1*N2*
$Sel_{[a=5]}(R1)$	*R1* has an attribute *a*	*0* (no matches)	*N1* (all match)
$Proj_{[a]}(R1)$	*R1* has an attribute *a*, *N1 > 0*	*1*	*N1*
R1 Div *R2*	The set of *R2*'s attributes is a subset of *R1*'s attributes	*0*	*floor(N1/N2)*

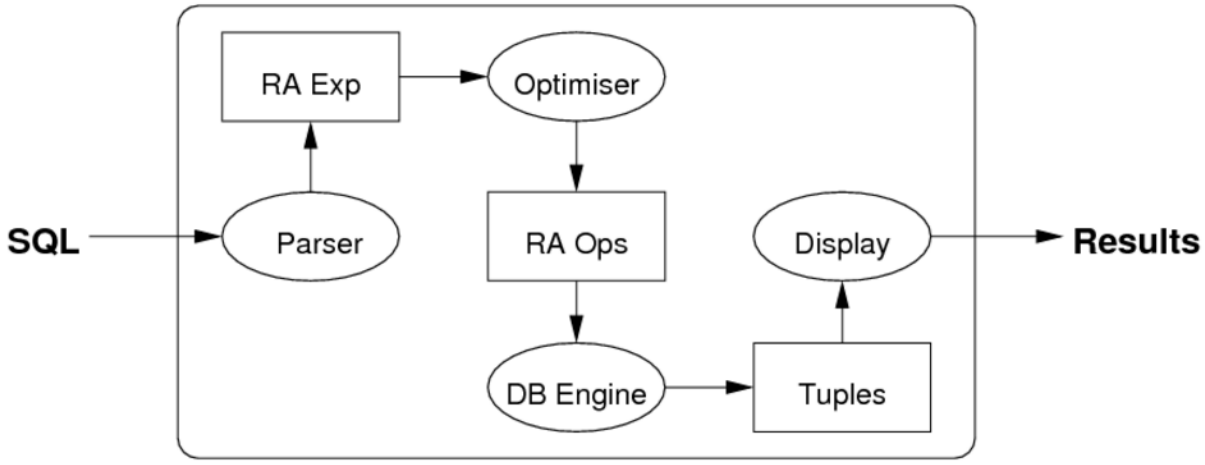
10.4 Query Processing

- 定义: The study of techniques for query evaluation
- 目标: 找到/建立满足要求的tuple集合

10.4.1 几种基本操作的差别

操作	影响
插入	只会影响一个tuple, 或者indexes
删除	寻找+删除, query-like
更新	寻找+删除, query-like, 和删除的区别在于更新只发生在一个单表上

- 因此查询黑盒可被细化为:



10.4.2 关系代数虚拟机:

- 对每次操作DBMS只会给出一个最优选择(通过query optimiser来确定)

selection (σ)	projection (π)	join ($\bowtie$, $\times$)
union ($\cup$)	intersection ($\cap$)	difference (-)
sort	insert	delete

- 评估cost:
  - data is stored in fixed-size blocks (e.g. 4KB)
  - data transferred disk : memory in whole blocks
  - cost of disk : memory transfer is highest cost in system
  - processing in memory is very fast compared to I/O

10.4.3 将SQL语句转化为RA

- 举个栗子:
  - SELECT变为projection
  - WHERE变为selection / join
  - FROM ... JOIN ... ON变为join(c)

```
-- schema: R(a,b) S(c,d)
select a as x
from   R join S on (b=c)
where  d = 100
-- mapped to
Tmp = Sel[d=100] (R join[b=c] S)
Res = Rename[a->x](Proj[a] Ttmp)
```

- 再来个栗子: 选择而所有上课人数超过100的学生
  - SQL:

```
select   distinct s.code
from     Course c, Subject s, Enrolment e
where    c.id = e.course and c.subject = s.id
group by s.id
having   count(*) > 100;
```

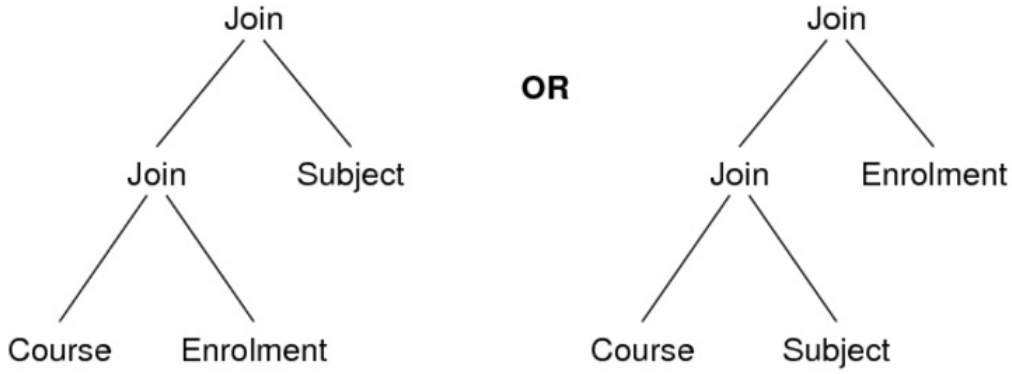
  - RA:

Result = Project'[s.code](GroupSelect[size>100](GroupBy[id] (JoinRes)))

where

JoinRes = Subject Join[s.id=c.subject] (Course Join[c.id=e.course] Enrolment)

  - 两种Join操作: 通过query optimiser进行优化



10.4.4 查询优化 : Query Optimiser

- 优化器作用过程:
  - 始于RA表达式
  - 优化器产生一系列等价表达式
  - 根据这些表达式生成所有可能的方法
  - 评估择优
- Cost的组成:
  - Size of relations (database relations and temporary relations)
  - Access mechanisms (indexing, hashing, sorting, join)
  - Size/number of main memory buffers
- 评估cost:
  - size of intermediate results
  - cost of secondary storage accesses
- 举个栗子:

Consider execution plans for: $\sigma_c (R \bowtie_d S \bowtie_e T)$

```
tmp1   :=  hash_join[d](R,S)
tmp2   :=  sort_merge_join[e](tmp1,T)
result :=  binary_search[c](tmp2)
```

or

```
tmp1   :=  sort_merge_join[e](S,T)
tmp2   :=  hash_join[d](R,tmp1)
result :=  linear_search[c](tmp2)
```

or

```
tmp1   :=  btree_search[c](R)
tmp2   :=  hash_join[d](tmp1,S)
result :=  sort_merge_join[e](tmp2)
```

All produce same result, but have different costs.

10.4.5 RA操作的实现

- 排序:
  - 快速排序无法实现
  - 归并排序: cost $O(N \log_B N)$ with B memory buffers
- 选择:
  - 顺序查找: $O(N)$
  - 基于index的查找: e.g. B-trees, tree nodes are pages, cose $O(\log N)$
  - 哈希查找: 常数级, 但是只适用于euqality tests
- 联结:
  - 嵌套循环联结: $O(NM)$, 使用buffering可优化为$O(N+M)$
  - 排序归并联结: $O(N \log N + M \log M)$

- 哈希联结: $O(N + \frac{MN}{B})$, with B memory buffers

10.4.6 一个综合的栗子: SQL to RA

```
Suppliers(sid, sname, address)
Parts(pid, pname, colour)
Catalog(supplier, part, cost)
```

- a. Find the names of suppliers who supply some red part
  - SELECT -> Proj
  - JOIN -> Join(C)
  - WHERE -> Sel

```
RedPartIds = Proj[pid](Sel[colour='red'](Parts))
RedPartSupplierIds = Proj[sid](RedPartIds Join Catalog)
Answer = Proj[sname](RedPartSupplierIds Join Suppliers)
```

- b. Find the sids of suppliers who supply some red or green part.

```
RedGreenPartIds = Proj[pid](Sel[colour='red' OR colour='green'](Parts))
Answer = Proj[sid](RedGreenPartIds Join Catalog)
```

- c. Find the sids of suppliers who supply some red part or whose address is 221 Packer Street.

```
RedPartIds = Proj[pid](Sel[colour='red'](Parts))
RedPartSupplierIds = Proj[sid](RedPartIds Join Catalog)
GreenPartIds = Proj[pid](Sel[colour='green'](Parts))
GreenPartSupplierIds = Proj[sid](GreenPartIds Join Catalog)
Answer = RedPartSupplierIds   Intersect   GreenPartSupplierIds
```

- d. Find the sids of suppliers who supply some red part and some green part.
  - 需要使用INTERSECT

```
RedPartIds = Proj[pid](Sel[colour='red'](Parts))
RedPartSupplierIds = Proj[sid](RedPartIds Join Catalog)
GreenPartIds = Proj[pid](Sel[colour='green'](Parts))
GreenPartSupplierIds = Proj[sid](GreenPartIds Join Catalog)
Answer = RedPartSupplierIds   Intersect   GreenPartSupplierIds
```

- e. Find the sids of suppliers who supply every part.
  - 需要用到DIV

```
AllPartIds = Proj[pid](Parts)
PartSuppliers = Proj[sid,pid](Catalog)
Answer = PartSuppliers Div AllPartIds
```

- f. Find the sids of suppliers who supply every red part.

```
RedPartIds = Proj[pid](Sel[colour='red'](Parts))
PartSuppliers = Proj[sid,pid](Catalog)
Answer = PartSuppliers Div RedPartIds
```

- g. Find the sids of suppliers who supply every red or green part.

```
RedGreenPartIds = Proj[pid](Sel[colour='red' OR colour='green'](Parts))
Answer = PartSuppliers Div RedGreenPartIds
```

- h. Find the sids of suppliers who supply every red part or supply every green part.
  - 需要用到UNION

```
RedPartIds = Proj[pid](Sel[colour='red'](Parts))
GreenPartIds = Proj[pid](Sel[colour='green'](Parts))
PartSuppliers = Proj[sid,pid](Catalog)
AllRedPartSuppliers = PartSuppliers Div RedPartIds
AllGreenPartSuppliers = PartSuppliers Div GreenPartIds
Answer = AllRedPartSuppliers Union AllGreenPartSuppliers
```

- i. Find the pids of parts that are supplied by at least two different suppliers.

```
C1 = Catalog
C2 = Catalog
SupplierPartPairs = Sel[C1.sid!=C2.sid](C1 Join[pid] C2)
Answer = Proj[C1.pid](SupplierPartPairs)
```

- j. Find pairs of sids such that the supplier with the first sid charges more for some part than the supplier with the second sid.
  - 个人感觉`C1.sid!=C2.sid`是为了提升效率, 其实不加也没什么关系

```
C1 = Catalog
C2 = Catalog
SupplierPartPairs = Sel[C1.sid!=C2.sid AND C1.cost>C2.cost](C1 Join[pid] C2)
Answer = Proj[C1.sid,C2.sid](SupplierPartPairs)
```

- k. Find the pids of the most expensive part(s) supplied by suppliers named "Yosemite Sham".
  - 需要用到笛卡尔积
  - 需要用到Minus

- R3给出了R1中

```
R1 = Proj[sid,pid,cost](Sel[name='Yosemite Sham'](Suppliers Join Catalog))
R2 = R1
R3 = Rename[1->sid,2->pid,3->cost](Sel[R2.cost<R1.cost](R1 × R2))
Answer = Proj[pid](R2 Minus Proj[sid,pid,cost](R3))
```

- l. Find the pids of parts supplied by every supplier at a price less than 200 dollars (if any supplier either does not supply the part or charges more than 200 dollars for it, the part should not be selected).

```
CheapParts = Sel[cost<200](Catalog)
AllSuppliers = Proj[sid](Suppliers)
Answer = Proj[part,supplier](CheapParts Div AllSuppliers)
```

Lecture 11 Transaction Processing, Concurrency Control

- 参考资料
  - http://www.jianshu.com/p/eb150b4f7ce0
  - http://www.crazyant.net/1741.html
  - http://www.hollischuang.com/archives/923

11.1 Transaction, Concurrency, Recovery

- 事务处理: 某些工作的逻辑单元需要多种数据库操作, 用于管理这些单元的技术称作事务处理
- 并发控制: 用于确保多个并发事务不会互相干扰的技术
- 还原机制: 在事故发生后将数据还原到某个特定状态的技术

11.2 数据库事务

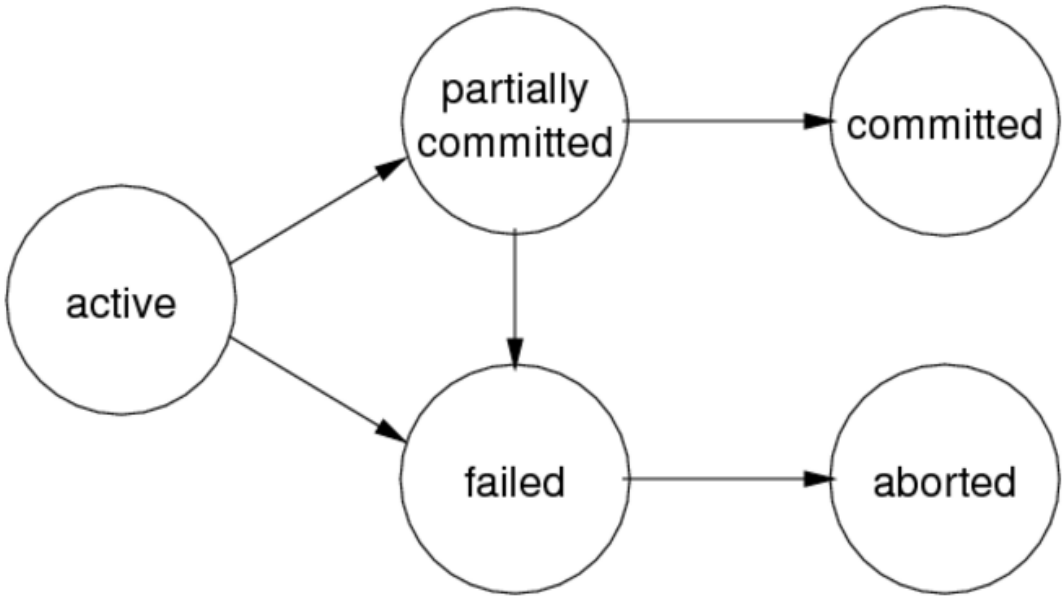
11.2.1 基本概念

- 定义: 访问并可能更新数据库中各种数据项的一个程序执行单元(unit)
- 事务为数据库操作提供了一个从失败中恢复到正常状态的方法，同时提供了数据库即使在异常状态下仍能保持一致性的方法

- 当多个应用程序在并发访问数据库时，可以在这些应用程序之间提供一个隔离方法，以防止彼此的操作互相干扰
- 某个事务被提交给DBMS时, DMBS需要确保该事务中的**所有操作都成功完成**且其结果被**永久保存**在数据库中，如果事务中有的操作没有成功完成，则事务中的所有操作都需要被回滚，回到事务执行前的状态（**要么全执行，要么全都不执行**）;同时，该事务对数据库或者其他事务的执行无影响，所有的事务都**独立运行**

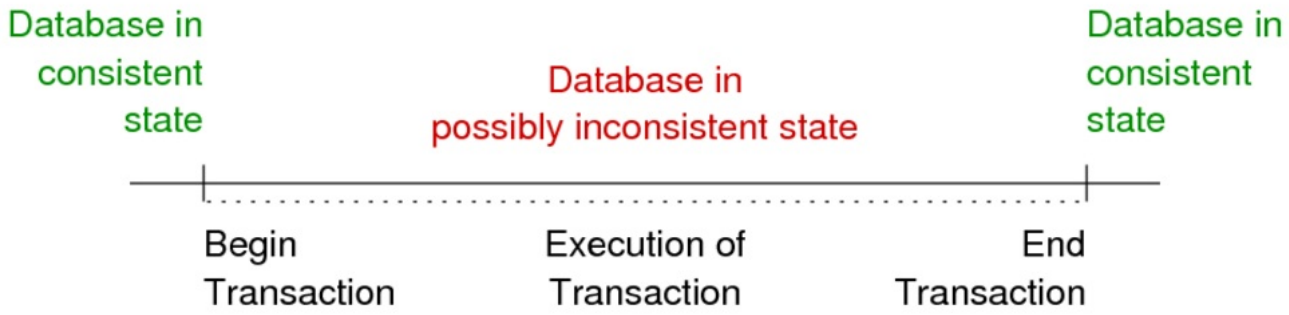
A transaction must always terminate, either:

- successfully (**COMMIT**), with all changes preserved
- unsuccessfully (**ABORT**), with database unchanged



11.2.2 ACID特性

- Atomic原子性:
  - 某个事务中的操作要么全部成功, 要么全部回退
  - 另一方面在读取数据时, 某个事务对同一数据项多次读取的结果一定是相同的
- Consistency一致性:



  - 只有合法数据可以被写入数据库
  - 事务需要保证数据库的正确性/完整性/一致性
  - 一致性可通过数据库内部规则或者应用规则进行保证
  - 中间态是inconsistant的, 但是事务发生前后的状态需要一致
- Isolation隔离性
  - 并发事务彼此独立, 不互相干扰
  - 另一方面不允许其他事务看到该事务的中间态(通过另一事务, 只能该事务进行前或该事务已经完成提交后数据库产生变化的状态)
  - 一般使用锁进行控制
- Durable持久性
  - 事务提交完成后, 事务处理结果被固化, 即使系统发生故障也可进行还原

11.2.3 事务造成的影响

- READ: 将硬盘中的数据读取到内存
- WRITE: 将内存中的数据写入到硬盘
- ABORT: 发生错误, 终止事务
- COMMIT: 事务成功提交, 终止事务
- 简写为R(X), W(X), A, C

SELECT produces **READ** operations on the database.
INSERT produces **WRITE** operations.
UPDATE, **DELETE** produce both **READ** + **WRITE** operations.

11.2.4 Schedules

- Serial schedules:

Serial execution: **T1** then **T2** or **T2** then **T1**

```
T1: R(X) W(X) R(Y) W(Y)
T2:                    R(X) W(X)
```

or

```
T1:              R(X) W(X) R(Y) W(Y)
T2: R(X) W(X)
```

Serial execution guarantees a consistent final state if

- the initial state of the database is consistent
- **T1** and **T2** are consistency-preserving

- Concurrent schedules: 后一个造成了异常, X在T1中进行写入操作后T2不再具有一致性

Concurrent schedules interleave **T1**,**T2**,... operations

Some concurrent schedules are ok, e.g.

```
T1: R(X) W(X)       R(Y)       W(Y)
T2:              R(X)       W(X)
```

Other concurrent schedules cause anomalies, e.g.

```
T1: R(X)       W(X)       R(Y) W(Y)
T2:       R(X)       W(X)
```

Want the system to ensure that only valid schedules occur.

11.2.5 串行化

- Serializable schedule:
  - concurrent schedule for T1 ..Tn with final state S
  - S is also a final state of one of the possible serial schedules for T1 ..Tn
- 实现串行化: schedule equivalence
  - conflict serializibility: 读写操作需要保证正确的顺序
  - view serializibility : 读取操作需要保证看到的数据是正确的

11.2.6 冲突串行化:

- 对于相同的数据来源, 只有读取的顺序可以更换
- 对于不同的数据来源, 随便怎样交换顺序都是等价的

T_1 first	T_2 first	Equiv?
$R_1(X)\ R_2(X)$	$R_2(X)\ R_1(X)$	yes
$R_1(X)\ W_2(X)$	$W_2(X)\ R_1(X)$	no
$W_1(X)\ R_2(X)$	$R_2(X)\ W_1(X)$	no
$W_1(X)\ W_2(X)$	$W_2(X)\ W_1(X)$	no

If T_1 and T_2 act on different data items, result is always equivalent.

- 通过交换不冲突操作的顺序得到的序列化日程是conflict serializible的

```
T1: R(A) W(A)       R(B)       W(B)
T2:           R(A)       W(A)       R(B) W(B)
swap
T1: R(A) W(A) R(B)             W(B)
T2:                 R(A) W(A)       R(B) W(B)
swap
T1: R(A) W(A) R(B)       W(B)
T2:                 R(A)       W(A) R(B) W(B)
swap
T1: R(A) W(A) R(B) W(B)
T2:                       R(A) W(A) R(B) W(B)
```

- Checking for conflict-serializability:
  - show that ordering in concurrent schedule
  - cannot be achieved in any serial schedule
- 检查冲突串行化的方法:
  - build a precedence-graph
  - nodes represent transactions
  - arcs represent order of action on shared data
  - arc from T1→T2 means T1 acts on X before T2
  - cycles indicate not conflict-serializable.

11.2.7 最后再举个栗子

- A账户向B账户汇款: 总共需要6步操作

  1、从A账号中把余额读出来（500）。
  2、对A账号做减法操作（500-100）。
  3、把结果写回A账号中（400）。
  4、从B账号中把余额读出来（500）。
  5、对B账号做加法操作（500+100）。
  6、把结果写回B账号中（600）。

- 原子性: 这六步操作要么全部执行要么全部部执行, 例如第五步时B账户注销了, 则之前的所有操作都被取消
- 一致性: 转账前两个账户共1000元钱, 转账后总钱数不变

- 隔离性: 在事务提交前, 查询A账户或B账户都不会看到钱数变化
- 持久性: 事务提交后, AB变化后的数据会被写入数据库持久保存

11.3 并发控制

- 串行性测试的不足:
  - 只能在时间发生后进行
  - 实际使用成本高$O(n!)$

11.3.1 四种并发控制方法

- lock-based

  Synchronise transaction execution via locks on some portion of the database.

- version-based

  Allow multiple consistent versions of the data to exist, and allow each transaction exclusive access to one version.

- timestamp-based

  Organise transaction execution in advance by assigning timestamps to operations.

- validation-based (optimistic concurrency control)

  Exploit typical execution-sequence properties of transactions to determine safety dynamically.

11.3.2 基于锁的并发控制

- 在读取数据X前, 获取X的共享锁
- 在写入数据X前, 获取X的排它锁
- 事务T对数据A加上共享锁后，则其他事务只能对A再加共享锁，不能加排他锁。获准共享锁的事务只能读数据，不能修改数据

- 事务T对数据A加上排他锁后，则其他事务不能再对A加任任何类型的封锁。获准排他锁的事务既能读数据，又能修改数据。
- 注意仅仅有这些规则是无法保证串行性的
- 锁与性能的关系: 锁减少了并发性, 吞吐量下降
- 锁的粒度(Granularity)会影响性能
  - 给一个小项目加锁: more of database accessible, 快速更新
  - 给多个小项目加锁: 更多的锁管理
  - 粒度级别: field, row (tuple), table, whole database

11.3.3 基于多版本的并发控制

- 提供数据库的多种版本(consistent)
- 赋予每个事务合适的版本(可以保证事务串行性的版本)
- MVCC与基于锁的模型的区别:
  - 写入操作建立了新版本tuple, 不会导致读取操作被锁
  - 读取操作读取旧版本的tuple, 不会导致写入操作被锁

11.3.4 使用SQL进行并发控制

Transactions in SQL are specified by

- **BEGIN** ... start a transaction
- **COMMIT** ... successfully complete a transaction
- **ROLLBACK** ... undo changes made by transaction + abort

In PostgreSQL, other actions that cause rollback:

- **raise exception** during execution of a function
- returning null from a **before** trigger

Concurrent access can be controlled via SQL:

- table-level locking: apply lock to entire table
- row-level locking: apply lock to just some rows

LOCK TABLE explicitly acquires lock on an entire table.

Other SQL commands implictly acquire appropriate locks, e.g.

- **ALTER TABLE** acquires an exclusive lock on table
- **UPDATE**, **DELETE** acquire locks on affected rows

All locks are released at end of transaction (no explicit unlock)

11.3.5 数据库的隔离级别

- 从低到高可分为: RU - RC - RR - S
- Read uncommitted(RU)：读取未提交的数据(未授权读取)
  - 其他事务已经修改单未commit的数据，这是最低的隔离级别
  - 允许脏读取，但不允许更新丢失
  - 如果一个事务已经开始写数据，则另外一个数据则不允许同时进行写操作，但允许其他事务读此行数据
  - RU可以通过排它锁实现
- Read committed(RC)：允许不可重复读取
  - 在一个事务中，对同一个项，前面的读取跟后面的读取结果可能不一样
  - 例如：第一次读取时另一个事务的修改还没有提交，第二次读取时已经提交了
  - RC可以通过瞬间共享锁和排它锁实现
  - 读取数据的事务允许其他事务继续访问该行数据，但是未提交的写事务将会禁止其他事务访问该行
- Repeatable read(RR)：可重复读取
  - 在一个事务中，对同一个项，要求前面的读取跟后面的读取结果一样
  - RR可以通过共享锁和排它锁实现

    - 读取数据的事务将会禁止写事务（但允许读事务），写事务则禁止任何其他事务
- Serializable(S)：可序列化
    - 最高执行级别
    - 要求事务序列化执行, 不允许并发执行
    - 如前面所说, 仅仅通过锁无法保证串行化，必须通过其他机制保证新插入的数据不会被刚执行查询操作的事务访问到
- 来个栗子:

    - 如果事务A设置为read uncommitted，那么事务B做了update还未提交，事务A能够读取到事务B更新的数据，事务B如果回滚，事务A则看到事务B回滚后的数据
    - 如果事务A设置为read committed，那么事务B做了update并且没有提交，事务A是读取不到事务B更新的数据的
    - 如果事务A设置为repeatable read，那么事务B做了update并提交，事务A仍然读取不到，事务B即使多次commit，事务A全都读取不到
    - 如果事务A设置为serializable，如果事务B已经开始运行并做了更新，那么事务A的任何操作得一直等待；如果B没做更新，则A还是能读取的
    - 以上四种级别，事务A如果正在更新一条数据，事务B如果要更新同一条数据，则会等待直到超时，因为事务A更新这条数据时加上了排它锁

11.3.6 事务执行异常

异常名称	成因

异常名称	成因
Lost update（LU）: 丢失更新	两个事务同时修改一个数据项，但后一个事务中途失败退出，则对数据项的两个修改可能都丢失
Dirty Reads（DR）：脏读	一个事务读取某数据项，但另一个事务更新了此数据项却没有提交，这样所有的操作可能都得回滚
Non-repeatable Reads(BRR): 不可重复读取	一个事务对同一数据项的多次读取得到不同的结果
Second lost updates problem(SLU): 二次更新丢失	两个并发事务同时读取和修改同一数据项，则后面的修改可能使得前面的修改失效
Phantom Reads（PR）：幻读	前面的查询和后面的查询的期间有另外一个事务插入数据，后面的查询结果出现之前未出现的数据

- 隔离级别和异常的关系:

	LU	DR	NRR	SLU	PR
RU	Y	Y	Y	Y	Y
RC	N	N	Y	Y	Y
RR	N	N	N	N	Y
S	N	N	N	N	N

11.3.7 举个栗子来说明事务隔离性

- 假定账户A有1000元,B有1000元,C有1000元
- 操作1: 操作员u1执行一次转账事务m1, 从A转移500元到B，再从A的余额中转移50%元平均分配到三个账户的余额中
  - 操作包括:

```
读A,读B,
写A,写B,
提交AB,读A,
读C,写C,写C,
提交AC
```

- 操作2: 操作员u2执行一次转账事务m2, 从B转移1000元到A

    - 操作包括:

```
读B,读A,
写B,写A,
提交AB
```

- 操作3: 操作员u3执行一次转账事务m3, 从A转移200元到B
    - 操作包括:

```
读A,读B,
写A,写B,
提交AB
```

- 操作4: 操作员u4开户D, 账户表为T_C，其包含字段为"账户名称cname,余额money",记录为"{A,1000},{B,1000}"
    - 操作包括:

```
读A,读B,
写A,写B,
提交AB
```

- RU:
    - m1: 读AB, 写A不写B
    - m2: 不可以写B, 可以读取AB, 但B是脏读
    - 排它锁
- RC:
    - m1: 读AB, 写A不写B
    - m2: 不可写B,读取A不可读取B(脏读)
    - 排它锁
    - m1: 读写AB, 提交AB
    - m3: 提交AB
    - 允许下一次m1
    - 瞬间共享锁
- RR:
    - m1: 读取AB, 仅仅写A

    - m2: 读取A不能读取或写入B(脏读)
    - 排它锁
    - m1: 读写AB, 提交AB
    - m3: 能读不能写A
    - m4: 提交D
    - 共享锁
    - 和RC的差别: 可以读取已经提交的事务但不能写, 同一张表可以插入新的记录
- S:
    - 任何事务都只能等前一事务执行完毕才可执行, 失去并发性

[137B9%95%E6%88%AA%E5%9B%BE.png