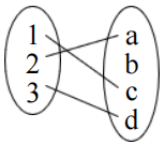
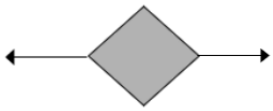
Chapter 3

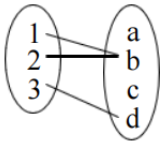
1. ER模型是一种概念设计模型
2. one-one many-one many-many

- one-one:

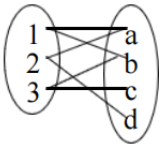


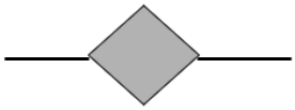


- many-one



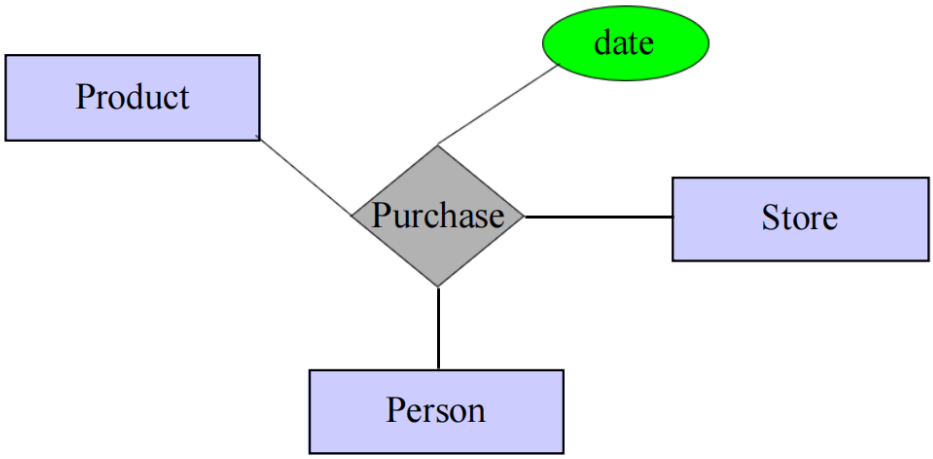
- many-many



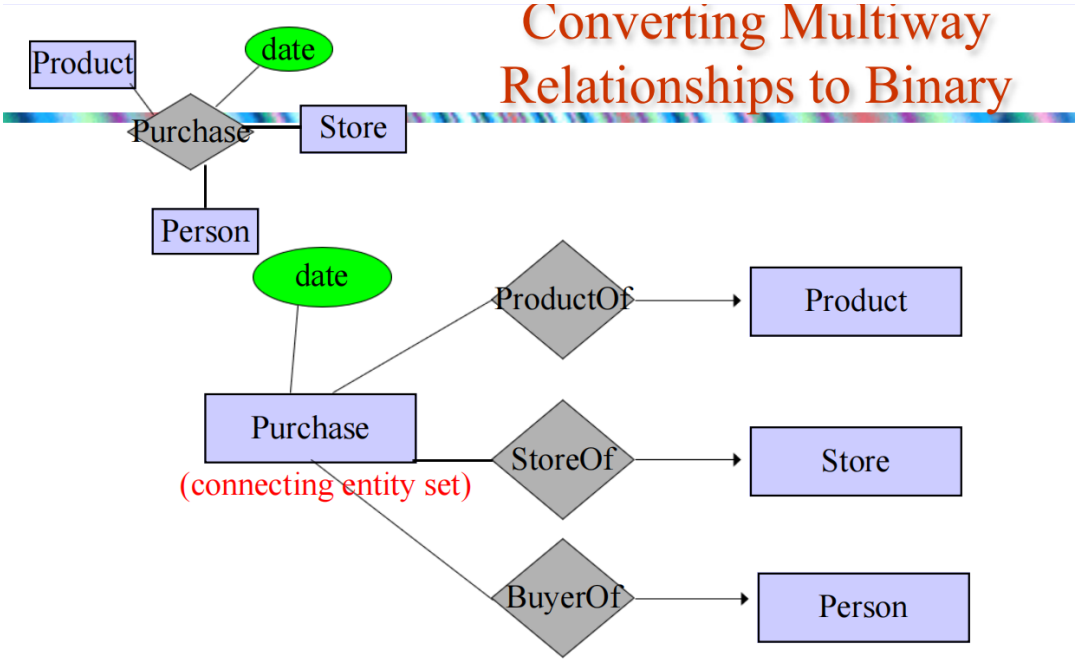


注意多对一的关系中的圆箭头，表示必须有并且只有一个

3. 关系同样可以拥有属性，在必要的时候可以把关系的属性作为一个实体集合

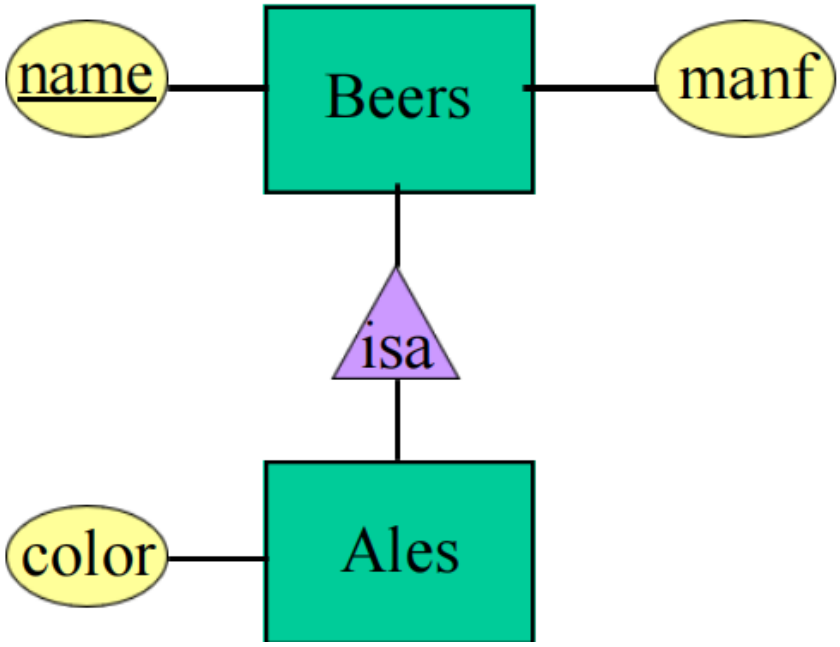


4. 多路关系可以转化成多个多对一的关系



5. 继承关系：

- ER模型中的继承不包含多路继承，只能有一个父类，但是一个实体可以属于多个实体集合
- 只有父类有Key，子类没有Key



6. ER模型中的约束关系

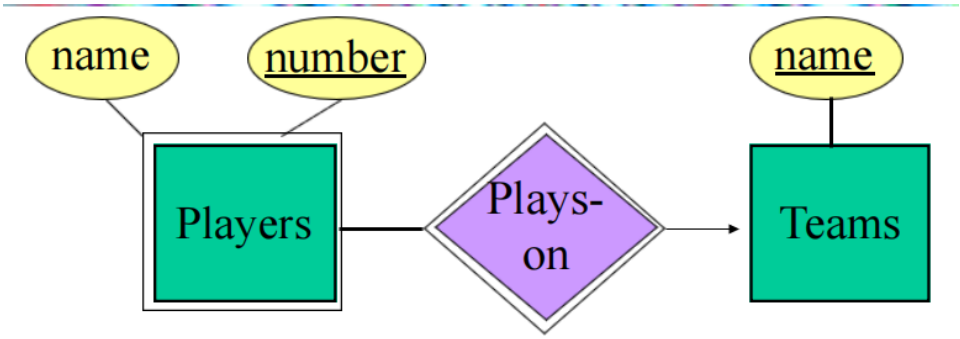
- 键约束：在主键上不能有重复
- 参照完成性约束：如果有外键，则必须存在
- 域约束：域中具体取值存在一定条件
- 其他约束

7. 键值（Key）

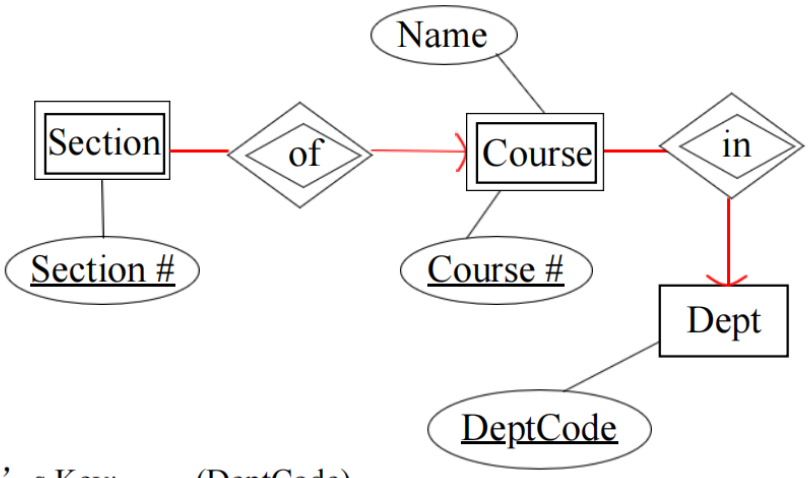
- 一个实体集合可以有多个键值，主键只能有一个
- 键值可以有多个属性

8. 弱实体集合

- 和其依赖的实体集合之间必须是多对一的关系

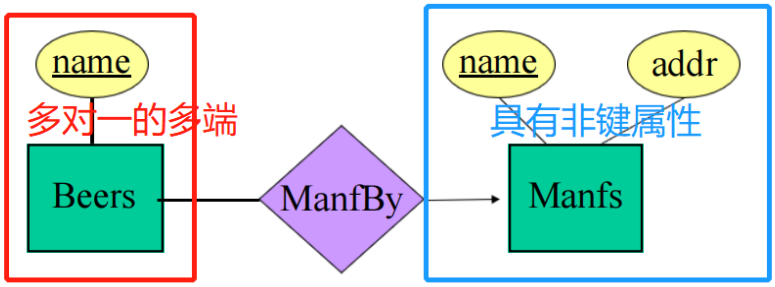


- 键值由自身的键值和依赖的实体集合的键值共同构成



Dept’ s Key: (DeptCode)
Course’ s Key: (Course #, DeptCode)
Section’ s key: (Section #, Course #, DeptCode)

9. ER图的设计原则
    - 反应数据的真实意图
    - 避免冗余
    - 属性代替实体集合
        - 只有当某个实体集合具有非键属性或者在多对一关系的多端的时候，才必须作为一个实体集合，否则可以作为属性

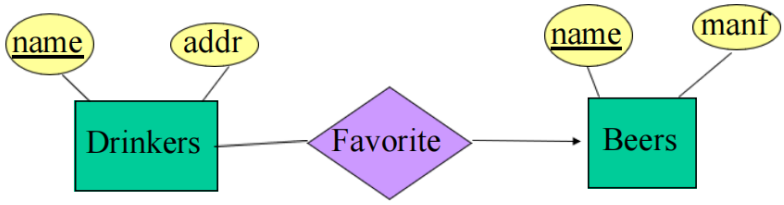


    - 不要过度使用弱实体集合
        - 必要的时候可以采用唯一标号的方式

Chapter 4

1. 关系的模式：表名（属性名[:域类型]），数据库的模式：关系模式的集合
2. 如果实体集合A到B之间是**一对一或多对一**的关系，则可以把这个关系融合到多端，而不需要为这个关系单独开一张表。

Example: Drinkers(name, addr) and Favorite(drinker, beer) combine to make Drinker1(name, addr, favoriteBeer).



但是需要注意，多对多关系就不可以，这样会导致冗余

3. 对于弱实体集合，在建立表的时候需要所有的键属性，但是不需要为表明弱实体集的关系建表
4. 对于继承关系
    - 面向对象的方式：每个实体集合成立一张表，并且吸纳所有的属性
        - 方便查找每一个实体的全部属性和子类属性，最节约空间
    - ER图的方式：子类通过key外键到父类
        - 方便查找父类属性，键值会在父类表和子类表中重复
    - 空值的方法：只有一张表，没有值的地方置为NULL
        - 浪费存储空间，但是结构统一
5. 数据模型：
    - 概念模型：ER模型
    - 逻辑模型：层次模型、网状模型、关系模型、面向对象模型
    - 物理模型：底层存储方式
6. 完整性约束
    - 实体完整性：主键属性不能为空
    - 参照完整性：外键或为参考关系中主键的值或者为空
    - 用户定义的完整性：反映某一具体应用所设计的数据必须满足的语义要求

Chapter 5

1. DDL：对数据库结构的定义和修改 DML：对数据的增删改查
2. SQL语言具有：数据定义、数据操作和数据控制的功能
3. numeric(a,b) 总共a位，小数点后b位
4. unique 取值可以为空并且NULL可重复 primary key的任何一个属性取值不能为空

   需要注意当某些属性组合不能重复的时候需要单独写 UNIQUE(a,b,c) 或者 PRIMARY KEY(a,b,c)

   有的数据库可能为PRIMARY KEY创建索引，但是不会为UNIQUE创建索引
5. 外键

   可以直接在属性后加上 `REFERENCE table(attribute)`

或者在最后加上 `FOREIGN KEY(attribute) REFERENCE table(attribute)`

外键在参照的表中必须声明为 `PRIMARY KEY` 或者 `UNIQUE`

在插入的时候，如果外键在参照的表中不存在则会插入失败

当被参照的表在被参照属性上进行修改的时候，默认情况下同样会被拒绝

级联修改：当被参照表中的被参照属性发生变化的时候，会相应修改参照表中的内容，需要在声明外键的时候加入

`FOREIGN KEY(xxx) REFERENCE xxx(xxx) ON DELETE/UPDATA cascade`

同样也可以使用Set Null的方式，当被参照表中发生修改的时候会将参照表中的外键置为NULL

`FOREIGN KEY(xxx) REFERENCE xxx(xxx) ON DELETE/UPDATA set null`

多属性构成外键

```
create table section
    (course_id      varchar (8),
     sec_id         varchar (8),
     semester       varchar (6),
     year           numeric (4,0),
     building       varchar (15),
     room_number    varchar (7),
     time_slot_id   varchar (4),
     primary key (course_id, sec_id, semester, year),
     foreign key (course_id) references course);

create table teaches
    (ID             varchar (5),
     course_id      varchar (8),
     sec_id         varchar (8),
     semester       varchar (6),
     year           numeric (4,0),
     primary key (ID, course_id, sec_id, semester, year),
     foreign key (course_id, sec_id, semester, year) references section,
     foreign key (ID) references instructor);
```

6. 当设置为主键的属性中有默认值的时候，要注意单独插入某个主键属性的时候可能会出现重复冲突

```
CREATE TABLE Sells (
        bar   CHAR(20),
        beer  VARCHAR(20)  DEFAULT 'HouseBeer',
        price REAL         NOT NULL,
        PRIMARY KEY (bar,beer)
 );
```

```
insert into sells(bar,price)
values ('456',8.9);
select * from sells;
```

	bar	beer	price
▶	456	HouseBeer	8.9

```
insert into sells(bar)
values ('456');
```

7. 使用check进行属性检查

```
CREATE TABLE Sells (
        bar CHAR(20),
        beer CHAR(20)  CHECK (beer IN
        (SELECT name  FROM Beers)),
        price REAL  CHECK (price <= 5.00));
```

CHECK检查类似于外键约束，check检查只有在sells表中插入和更新的时候会起作用，而在beers表中删除或者更新的时候不起作用

8. 在SQL语句中引号内部的单引号要用双引号

9. 语法总结

```sql
create table test(
    id varchar(10),
    name varchar(10),
    age int,
    PRIMARY KEY(id)
);

drop table test;

insert into test(id,name,age) values('111','aaa',18);

delete from test where name='111';

alter table test add phone char(11);
alter table test drop column age;

select xxx as xxx
from xxx
where xxxx;

// 使用特定的字符串作为某一列的值
```

```sql
select drinkr, 'likes Bud' as wholiksBud
from likes
where beer = 'Bud';
```

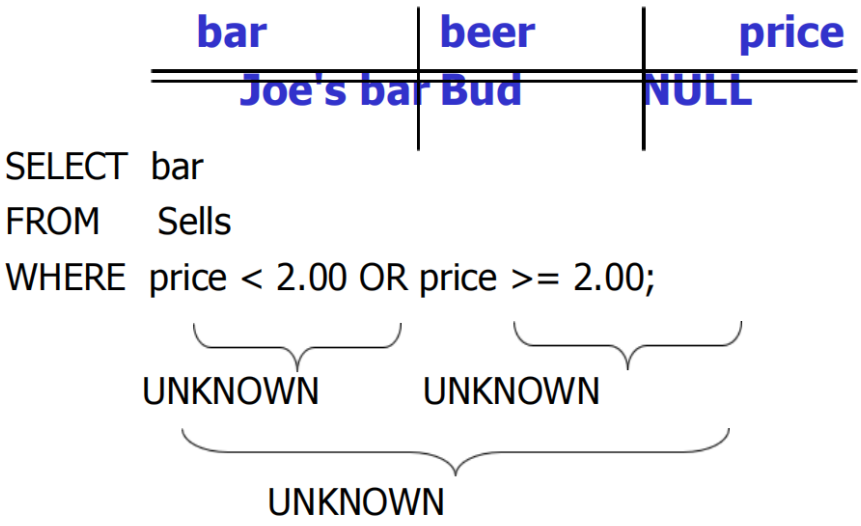
10. like进行字符匹配，%可以匹配多个 _只匹配一个

```sql
select * from employee where empname like '李%';
```

11. SQL中是三值逻辑，所有和NULL的比较结果都为UNKOWN

    逻辑运算 and=min or=max not=1-x

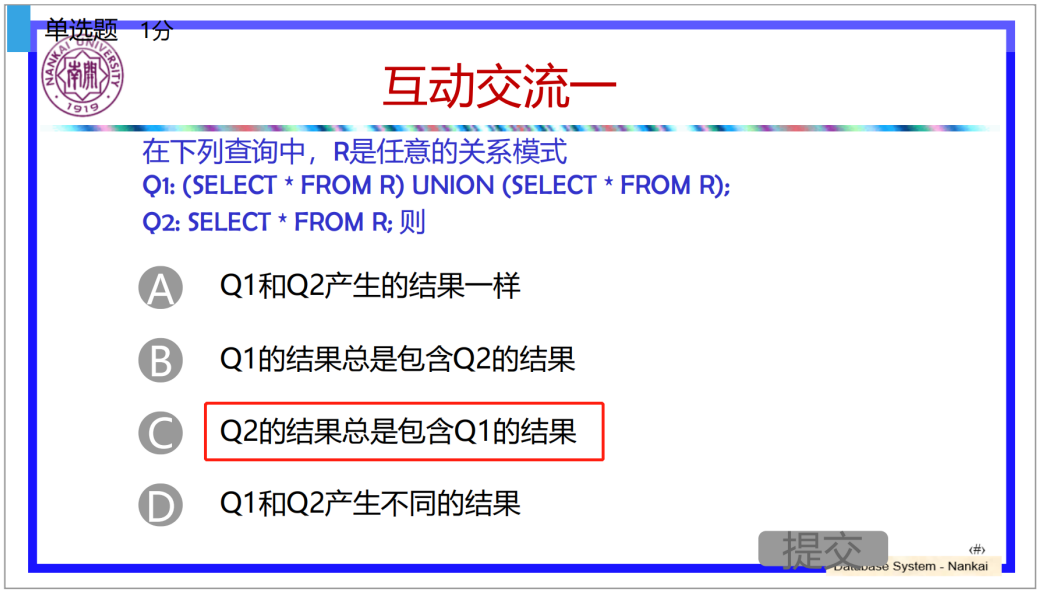
    这里要注意，一个恒真的条件也可能因为NULL导致没有结果



- Joe's Bar is not produced, even though the WHERE condition is a tautology.

    where只有当值为1的时候才成立

12. 查询语句默认为包，可以有重复的元组，而通过union等集合操作之后就转化成了集合，不允许有重复的元组出现，如果想要保留重复元组，需要在union等操作后面加上all保留包语义。如果想要把select的结果消除重复，可以使用distinct关键字，代价比较高，需要进行排序。select distinct price from table;



Q1去重了，Q2没有去重

13. 对于有聚合或者分组的操作，select子句中出现的属性都需要在聚合语句中或者在分组语句中

Sells(bar, beer, price)

- **Possible Query to find the bar that sells Bud the cheapest:**

```
SELECT   bar, MIN(price)
FROM     Sells
WHERE    beer = 'Bud';
```

- Illegal. Why?
- Rule: *Each* element of a SELECT clause must either be aggregated or appear in a group-by clause.

14. having子句是对group的筛选，筛选满足条件的分组
15. 语法总结

```sql
# 排序
select empname
from Employee
where deptno='d1'
order by empno DESC/ASC;

# 多表查询
select *
from A,B
where A.c = B.c;  #A.c 和 B.c 重复出现
select *
from A natural join B;  # A.c 和 B.c 被合并

# 筛除相同元组对
select e1.empname, e2.empname
from employee e1, employee e2
where e1.deptno = e2.deptno and e1.empname < e2.empname;

# 子查询
select empname
from employee
where deptno=(
    select deptno
    from department
    where location in ('北京','天津')
);

```

```sql
# exists   exists(relation) 为真当relation不为空
# 查询生产厂商只生产一种啤酒的啤酒   Beers(name,manf) 主键name
select name
from beers b1
where not exists (
    select *
    from beers b2
    where b1.manf = b2.manf and b1.name <> b2.name
);

# x op any(relation) 为真当且仅当x与relation中至少一个元组满足op运算符
# x op all(relation) 为真当且仅当x与relation中的任何一个元组都满足op运算符
# 需要注意的是，元组中只能有一个属性
# 查询最高价的啤酒和不是最低价的啤酒   Sells(bar,beer,price) bar和beer共同构成主键
select beer
from sells
where price >= all(
    select price
    from sells
);
select beer
from sells
where price > any(
    select price
    from sells
);

select empname
from employee e1
where not exists(
    select *
    from workson w1, workson w2
    where w1.empno = e1.empno and w2.empno = e1.empno and w1.projectno <> w2.projectno
);

# 聚合操作
select count(distinct empno) from workson;

select beer, avg(price)
from sells
group by beer;
```

16. SQL Assertion 要始终保持关系在这个Assertion上为真，每当关系进行修改的时候都要进行检查，如果违背则拒绝修改。

```
# 没有一个酒吧可以将啤酒平均价格超过5块钱
create assertion ass
check (
    not exists(
        select bar
        from sells
        group by bar
        having avg(price) > 5
    )
);
```

17. 触发器：当某一类行为发生的时候，如果满足某个条件，则执行某个操作
    - assertion：始终保持为真
    - triggers：在指定动作触发并且满足条件的时候执行操作

```
# 当一个新的元组插入sells时，如果这个啤酒不在beers中，就将其插入beers中
delimiter //
create trigger beertrig
before insert on sells for each row
begin
if new.beer not in(select beer from beers) then
insert into beers(beer) values(new.beer)
end if;
end; //
delimiter ;
```

18. 语法总结

```
# 插入
insert into table(a,b,c) values(1,2,3);
insert into table (
    select distinct drinker
    from frequents f1
    where f1.drinker <> 'Sally' and
    bar in(
        select bar
        from frequents f2
        where f2.drinkers = 'Sally'
    )
);

# 删除
delete from table
where cond;
# 全部删除
delete from table;   # 保留表的结构  drop table xxx 完全删除表

# 更新
update xxx set xxx=xxx where cond;
```

```
update drinkers
set phone = '555-1212'
where name = 'Fred';
```

19. 视图：从一个或几个基本表中导出的虚拟表

`create view xxx[xxx,xxx,xxx] as {select子句} [with check option]`

视图列名表或者全部省略或者全部指明

不能省略的情况：某列是聚合函数或者表达式、包含同名列、重命名

```
create view v_count(projectno, projectcount)
as
    select projectno, count(*)
    from workson
    group by projectno;
```

视图的更新：并不是所有的视图都可以更新，一般都是单表导出的可以更新

```
CREATE VIEW v_1997_check
  AS SELECT empno,projectno,enterdate
      FROM workson
      WHERE enterdate BETWEEN '1997-01-01' AND '1997-12-31'
      WITH CHECK OPTION;
 INSERT INTO v_1997_check
VALUES(18316,'p1','1998-1-15')
```

因为有`with check option`，因此不满足条件的不会被更新到视图中。但是如果没有，则会通过更新视图插入到依赖的数据表中。

20. 索引：

`create index 索引名 on 表名(属性名);`

DBMS自动选择合适的索引结构进行索引，不需要用户指定。

创建索引的要点：

- 向有索引列的表进行插入、删除、修改都会引起性能下降
- where子句中的条件含有多个and的时候，最好创建多属性索引
- 在连接操作的情况下，最好为每一个连接创建索引

21. 权限：

- select、insert、update都可以应用到一个属性上，而delete只能应用到一个元组上
- 当SQL语句执行的时候，只有拥有相关属性或者元组的权限才能够执行成功
- 授权：GRANT OPTION 如果被授权的用户拥有授权权限，则他能够继续给其他用户授权，只能授予其他用户自己权限的子集

- Suppose we also grant:

```
GRANT UPDATE ON Sells TO sally
WITH GRANT OPTION;
```

- Now, Sally can not only update any attribute of Sells, but can grant to others the privilege UPDATE ON Sells.
  - Also, she can grant more specific privileges like `UPDATE(price) ON Sells`.

- 收权：
  - `CASCADE`：级联收回
  - `RESTRICT`：如果存在传递关系直接拒绝
  - 必须要有上述选项
- 授权图
  - 用AP表示用户A拥有P权限
  - AP*表示用户A对权限P拥有授权权限
  - AP**表示用户A拥有权限P的对象
  - 用户拥有某个权限，当且仅当该用户能够连到Q**上
  - 当修改了边之后，要检查每一个节点是不是能够关联一个**节点，不符合条件的节点要被移除
  - 收回权限的时候只删除该权限对应的那条边

Chapter 6

1. 数据库中的异常
   - 冗余
   - 更新异常
   - 删除异常
   - 插入异常
2. 如何确定关系的key
   - 如果是实体集合，则key是实体集合的键
   - 如果来自**多对多关系**，则key是每一个**实体集合的键值以及关系的属性值**，但是如果有箭头，则不需要关系的属性
3. Armstrong公理系统
   - Reflexivity rule：属性集合可以函数决定其子集
   - Augmentation rule：在原函数依赖关系左右加上相同的属性，函数依赖仍然成立（A1A2->B1 ==> A1A2C1 -> B1C1）
   - Transitivity rule：函数依赖可以传递（A->B,B->C ==> A->C）
   - Union rule：结合律（A->B,A->C ==> A->BC）
   - Decomposition rule：分解律（A->BC ==> A->B,A->C）
   - Pseudo-transitivity rule：伪传递律（A->B,BC->D ==> AC->D）
4. 函数依赖集合的闭包：

- 将给定的函数依赖集合使用自反律和增广律进行扩充
- 然后在使用传递律结合律分解律等进行扩充
- 直到函数依赖集合不再扩充

通常运用函数依赖的闭包来判断两个函数依赖是否等价

5. 属性集合的闭包：$\{A1, A2 \ldots\ldots An\}$的闭包用$\{A1, A2 \ldots\ldots An\}^+$表示

   使用函数依赖不断扩充属性，最终得到属性的闭包

6. 求键的启发式算法

   1. 不在任何函数依赖右边出现的属性都是键的一部分
   2. 不在函数依赖左边出现的属性不是键的一部分
   3. 先用1后用2

7. 属性闭包的用法

   1. 判断属性组合是不是超键
   2. **判断函数依赖(A->B)是否被蕴含，只需要计算B在不在A的闭包里**
   3. 求函数依赖关系的闭包。先求出所有属性子集X的X^+，然后计算X^+的子集Y，输出所有的$X \to Y$

8. 模式求精的原则

   - 最小化冗余
   - 避免信息丢失，无损分解，**拆分后的表进行自然连接不应该多出数据行**
   - 保持函数依赖
   - 确保是好的查询

9. BCNF范式

   - **所有非平凡的函数依赖的左侧都是超键**
   - 判断BCNF范式
     - 首先求出所有的key
     - 判断所有的非平凡的函数依赖是否左侧都是超键
   - **只有两个属性的关系模式一定是BCNF范式，没有函数依赖的关系模式一定是BCNF**
   - 对不满足BCNF的范式如何进行拆分
     - 首先确定违背BCNF的函数依赖$X \to Y$
     - 然后求X的闭包X^+，将R拆成两部分X^+和$R - X^+ + X$。
     - 求出函数依赖的闭包，然后分配到两个集合中
   - BCNF范式能够保证无损连接，但是不能够保证不丢失函数依赖，需要降到3范式

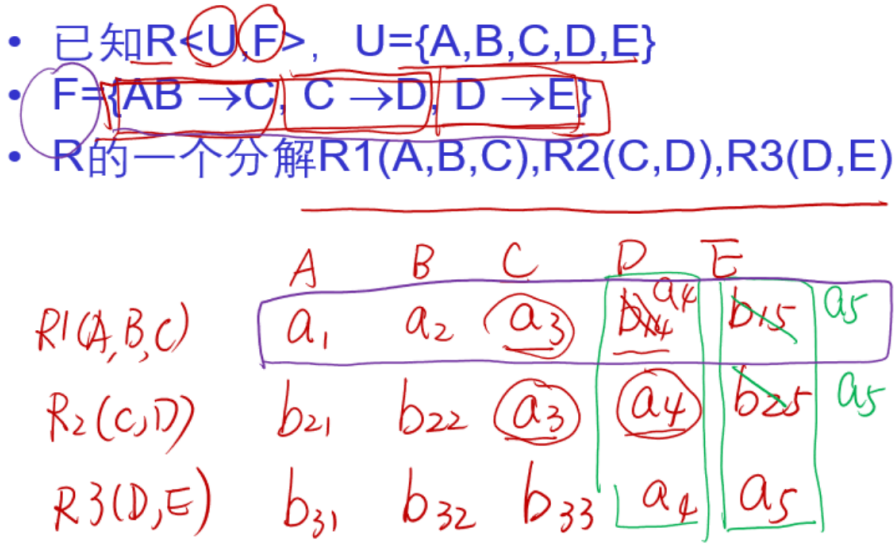
10. 3范式

    - **函数依赖$X \to A$违背3范式，当且仅当X不是超键并且A不是主属性**
    - BCNF分解一定是可恢复的分解，3范式分解既是可恢复分解也是保持函数依赖的分解
    - 极小函数依赖集合
      - 任意函数依赖右部都只有一个属性
      - 不存在$X \to A$使得$F - \{X \to A\}$与F等价
      - 不存在$X \to A$，X有真子集Z，使得$F - \{X \to A\} \cup \{Z \to A\}$与$F$等价

- 3范式分解
    - 先求出极小函数依赖集合和关系模式的所有key
    - 在极小函数依赖集合中按照左部相同的原则进行分组，每个组的全部属性构成分解后的子关系
    - 如果某个子关系模式中的所有属性被另一个包含，则将其删除
    - 判断是否所有的key都在关系模式中被包含，如果没有，则将其作为一个新的子关系模式

11. 判断无损连接分解

- 首先将所有的属性和关系模式列成表
- 给每个关系模式标注覆盖的属性
- 依次遍历函数依赖集合，左部相同的列右部也要相同，有a全用a，没有a用较小的b
- 重复操作直到出现一行全是a或者表不再变化，判断是否有一行全是a



12. 多值依赖

- 如果$X \rightarrow\rightarrow Y$，则任意交换两行的$Y$，新的元组还在表中
- 函数依赖是多值依赖的特例，如果$X \rightarrow Y$则$X \rightarrow\rightarrow Y$
- 如果$X \rightarrow\rightarrow Y$并且$Z$是其他属性，则$X \rightarrow\rightarrow Z$
- 多值依赖属性不可拆分

If AB→C then A→C，B→C 错误 If A→BC then A→B，A→C 正确	If AB→→C then A→→C，B→→C 错误 If A→→BC then A→→B，A→→C 错误

13. 4范式

- **任何一个非平凡多值依赖$X \rightarrow\rightarrow Y$都满足$X$是超键，则是4范式**
    - 非平凡多值依赖：Y不是X的子集，除了X、Y还有其他属性
    - **在计算key的时候，不考虑多值依赖**
- 满足4范式一定满足BCNF

Chapter 7

- 运算符优先级

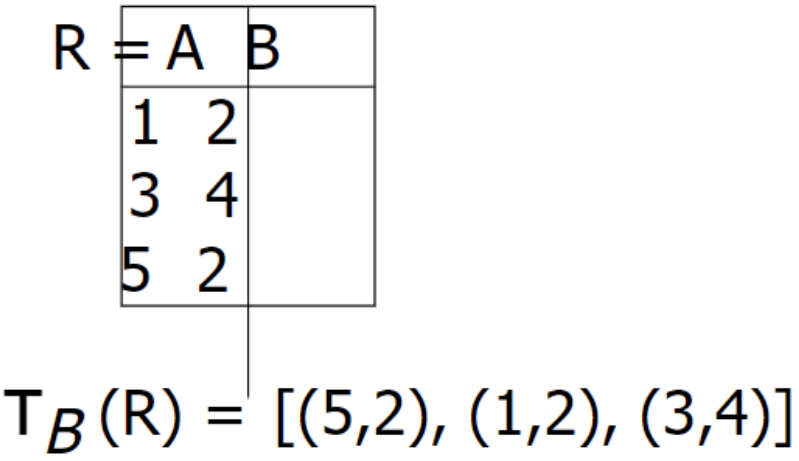
  σ, π, ρ

  $\times, \bowtie_c, \bowtie$

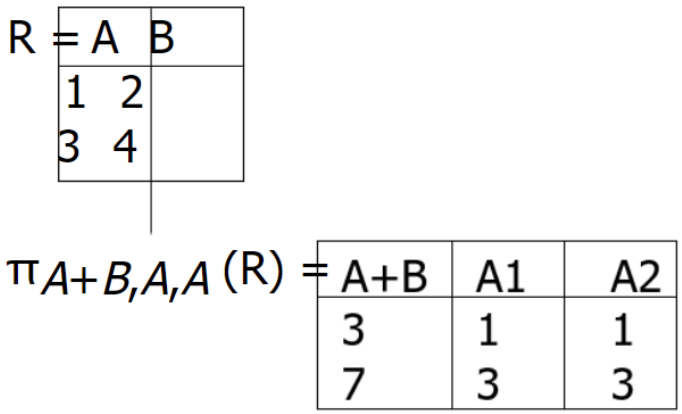
  $\cap$

  $\cup, -$

- 扩展运算符
  - δ 去除重复元组
  - τ 对元组排序



  - 广义投影，对算数运算进行投影



  - γ 用于分组和聚合

R = A B C

1 2 3
4 5 6
1 2 5

$\gamma_{A,B,AVG(C)}$ (R) = ??

First, group R :

A	B	C
1	2	3
1	2	5
4	5	6

Then, average C within groups:

A	B	AVG(C)
1	2	4
4	5	6

- 外连接避免信息丢失，对于无法连接的用NULL占位

R = A B S = B C

1 2 2 3
4 5 6 7

(1,2) joins with (2,3), but the other two tuples are dangling.

$\bowtie$ R S =

A	B	C
1	2	3
4	5	NULL
NULL	6	7

- 除法，$S \div R$，即从S中找到包含R中全部元素的元组，然后去掉R中的属性

S(S#,C#)

1 C1
1 C2
1 C3
1 C4
2 C1
2 C2
3 C2
4 C2
4 C3
4 C4
5 C4

CA(C#)
C2

S÷CA(S#)
1
2
3
4

CB(C#)
C1
C2

S÷CB(S#)
1
2

CC(C#)
C1
C2
C3
C4

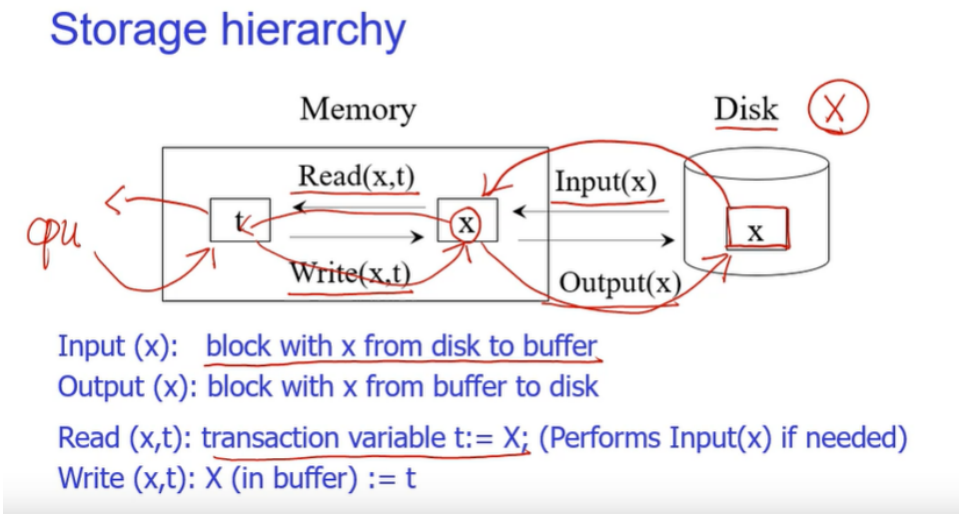
S÷CC(S#)
1

- 关系运算符的最小完备集

  $\cup, -, \times, \sigma, \pi$

Chapter 9

- 事务
  - 事务是数据库操作的最小单元
  - 引入事务为了保证数据库存储的语义时刻是正确的
  - 事务执行的特性ACID
    - A　原子性：事务是不可分割的单位，事务中的操作要么全部完成要么全不完成
    - C　一致性：事务必须保证数据库的一致性
    - I　隔离性：一个事务内部的数据对并发的其他事务是隔离的，并发的事务之间互不干扰
    - D　持久性：数据一旦提交了，其他的故障就不会对其产生影响
  - 事务不会破坏数据库的一致性
  - 事务操作的流程



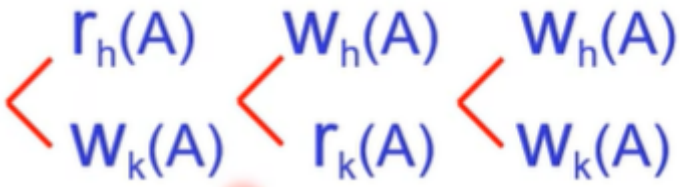
    - 首先从磁盘Input数据到内存
    - 然后从内存Read到事务中
    - 事务运算完成后Write到内存中
    - 然后通过Output写回磁盘
  - 数据完整性：数据的精确性和可靠性，数据语义的正确性
- undo日志
  - undo日志中要记录
    - <Ti , start>
    - <Ti , commit>
    - <Ti , abort>
    - <Ti , X , v> 对X进行更新，原值为v
  - 事务执行之前，要先记录下<Ti , start>
  - 在内存中的读写操作可以正常执行，在进行读写操作的时候，需要先在内存中记录下相应的日志记录
  - **当有Ouput操作的时候，要先写入相关的<Ti , X , v>，才能更改磁盘数据**
  - **当所有的磁盘修改工作完成之后，才能够写入<Ti , commit>**

    - undo日志如果看到了结束标记，就不需要进行恢复，只有没有结束的事务才需要进行恢复
    - undo日志数据库恢复
        - 查找磁盘上没有完成的事务
        - 从后向前将这个事务进行回滚，对日志中所有的<Ti , X , v>恢复，并写回磁盘
        - 所有的回滚完成之后，将<Ti , abort>写入磁盘
- redo日志
    - <Ti , X , v> 对X进行更新，新值为v
    - **当Output的时候，必须先进行<Ti , commit>，并且将日志写入磁盘，然后再执行Output**
    - redo日志数据库恢复
        - 查找磁盘日志上完成的事务
        - 顺序重新完成事务
    - redo日志的优化——检查点——静态检查点
        - 插入检查点前的所有日志记录的事务全部完成了，无需恢复
        - 如何设计CKPT
            - 首先停止接收事务
            - 完成正在执行的事务
            - 将日志写入磁盘
            - 将内存中数据的更改写入磁盘
            - 将CKPT写入磁盘
            - 重新接收事务，恢复运行
        - 带有检查点的redo日志数据库恢复
            - 查找最近的CKPT之后磁盘日志上完成的事务
            - 顺序重新完成事务
- undo/redo日志
    - <Ti , X , v-old，v-new> 对X进行更新，新值为v-new，旧值为v-old
    - **当有Ouput操作的时候，要先写入相关的<Ti , X , v-old , v-new>，才能更改磁盘数据**
    - **事务按照内存中的顺序写入磁盘完成之后 <Ti , commit> 写入磁盘，Output操作在 <Ti , commit> 之前之后都可以**
    - undo/redo日志数据库恢复
        - 对于所有未完成的事务，按照从后往前的顺序执行回滚
        - 对于所有完成的事务，按照从前往后的顺序执行重做
    - 非静态检查点
        - 在开始进行检查点的时候向磁盘写入 <START CKPT T1, …… , Tn>，表示这些事务还在执行，没有完成
        - 将<START CKPT T1, …… , Tn>之前对元素的修改操作写入磁盘
        - 完成之后将 < END CKPT > 写入磁盘
        - **发现 < END CKPT > 就知道与之对应的 <START CKPT T1, …… , Tn> 之前的对数据的修改操作全部完成了，并写入磁盘**

    - undo/redo日志数据库恢复
        - 从后向前找到第一个有对应的< END CKPT >的<START CKPT T1, ... , Tn>，在这个区间内找到所有未完成的事务进行回滚，到检查点开始的位置，对于那些尚处在活跃状态的事务要继续向前回滚
        - 从<START CKPT T1, ... , Tn>向后，重做已经完成的事务
- 数据库备份之后，之前的日志就可以丢弃了

Chapter 10

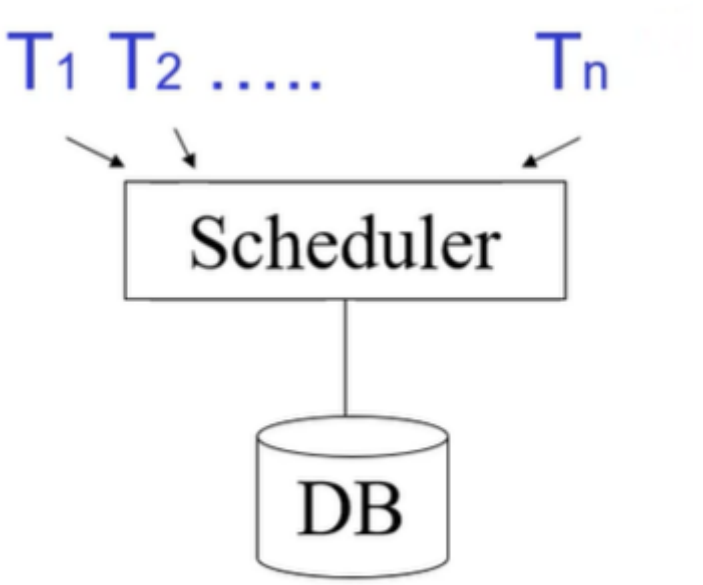
1. 调度
    - 一个好的调度应该满足
        - 不依赖数据库的初始状态
        - 不依赖事务的语义
    - 调度的正确性应该只关心READ/WRITE的顺序，不关系INPUT/OUTPUT的顺序，也就是说只能够表现内存中的值，而不是磁盘状态
    - 可串行化调度：等价于串行调度的调度
        - 冲突：在调度中如果调换了两个动作会改变调度的语义
            - 对同一个事务任意两个动作都是冲突的
            - 两个不同的事务对同一个元素的访问中有写操作
2. 冲突可串行化调度
    - 冲突动作



      如果事务1和事务2之间对同一个元素有共同的操作，其中有写操作，则在前面的事务必须先执行
    - 冲突等价调度：对于事务1如果交换不冲突的动作之后能够和事务2等价

      如果一个调度冲突等价于一个串行调度，则这个调度是冲突可串行化调度
    - 优先图 P(S) 称为调度S的优先图
        - 如果事务Ti有一个操作要在事务Tj之前执行，则Ti到Tj有一条弧
        - 对于每一个元素，列出其相关的事务
        - 通过事务之间的冲突关系画出依赖图
    - **<u>如果两个调度冲突等价，则优先图一定一样。但是如果优先图一样，不一定冲突等价</u>**。如果两个调度的优先图一样并且没有环，则一定是冲突等价的
3. 两段锁
    - 并发控制策略
        - 积极策略：对于事务全部运行，时刻检查优先图，如果出现环，则对部分事务进行回滚
        - 消极策略：事务不能随意进行操作
    - 悲观策略：

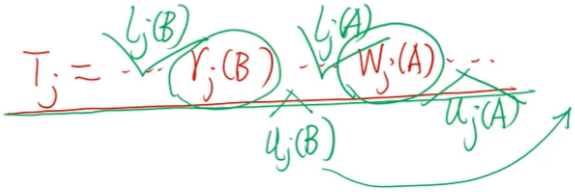
- 增加一个调度器，对于将要执行的事务进行检查



- 调度器通过锁机制实现
- 加锁解锁协议
  - Well formed transactions——对事务的限制

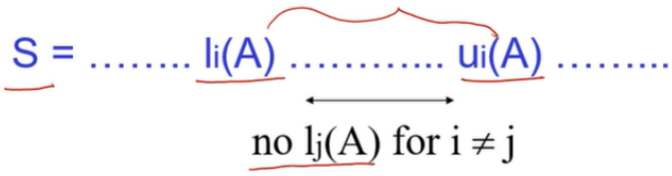
    事务在读写某个元素之前，一定要对元素进行加锁，在完成操作之后要解锁

- Lock elements (A) before accessing them
  (pi is a read or a write)
- Eventually, release the locks (ui(A))



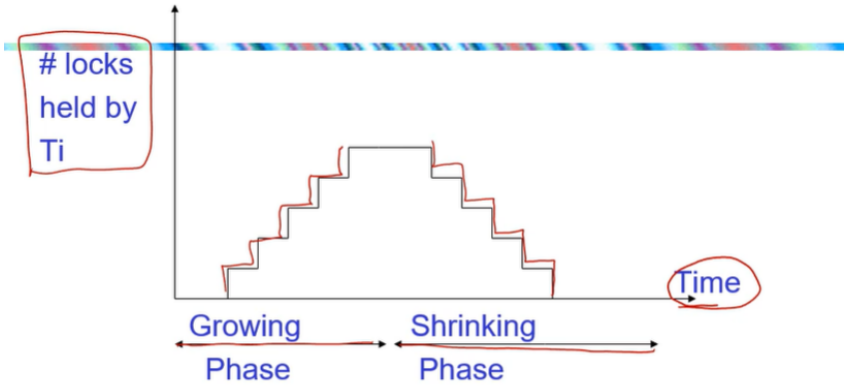
  - Legal scheduler——对调度的限制

    任何一个元素上只能持有一把锁



  - **在满足上述两个条件的调度不一定是可串行化顺序**

- 两段锁协议
  - 对于同一个事务，在加锁之前不能出现解锁，在解锁之后不能出现加锁，也就是加锁的时候一直加锁，解锁的时候一直解锁



- 满足两段锁协议一定是冲突可串行化调度

  对于一个调度，如果满足两段锁协议，但优先图中存在环，则必定存在事务依赖关系 $T_1 \to T_2 \to \cdots \to T_n \to T_1$，则可以推出事务的结束时间
  $SH(T_1) < SH(T_2) < \cdots < SH(T_n) < SH(T_1)$，推出矛盾

- 两段锁协议不能够检查死锁
  - 一次封锁法
    - 要求每个事务必须一次性把所有需要使用的锁全部获取
    - 降低了系统的并发度
  - 顺序封锁法
    - 对数据对象事先规定一个加锁的顺序，所有的事务都按照这个顺序上锁
    - 维护成本高

4. 共享锁

- 对于同一个元素，可以持有多个共享锁，但是互斥锁只能有一个
- 加锁解锁协议
  - Well formed transaction
    - 对于任意的读操作要先上 I-S 锁
    - 对于任意的写操作要先上 I-X 锁
    - 完成操作之后释放锁
    - 对于同时进行读写的操作
      - 可以直接上互斥锁——降低了并发度
      - 可以先上共享锁再在需要的时候升级为互斥锁——当其他事务在升级之间同样加锁，会导致等待
  - Legal schedule
    - 读锁之间不能有写锁但可以有读锁
    - 写锁之间什么锁都不能有

$S = ...l\text{-}S_i(A) \ldots \quad \ldots u_i(A) \ldots$

$\longleftrightarrow$

no $l\text{-}X_j(A)$ for $j \neq i$

$S = \ldots l\text{-}X_i(A) \ldots \quad \ldots u_i(A) \ldots$

$\longleftrightarrow$

no $l\text{-}X_j(A)$ for $j \neq i$
no $l\text{-}S_j(A)$ for $j \neq i$

Compatibility matrix:

Locks already held by some Ti \ Lock requested by some Tj	S	X
S	true	false
X	false	false

- 两段锁协议
  - 加锁和升级锁的过程中不能有解锁操作，一旦解锁就只能进行解锁操作

5. 更新锁

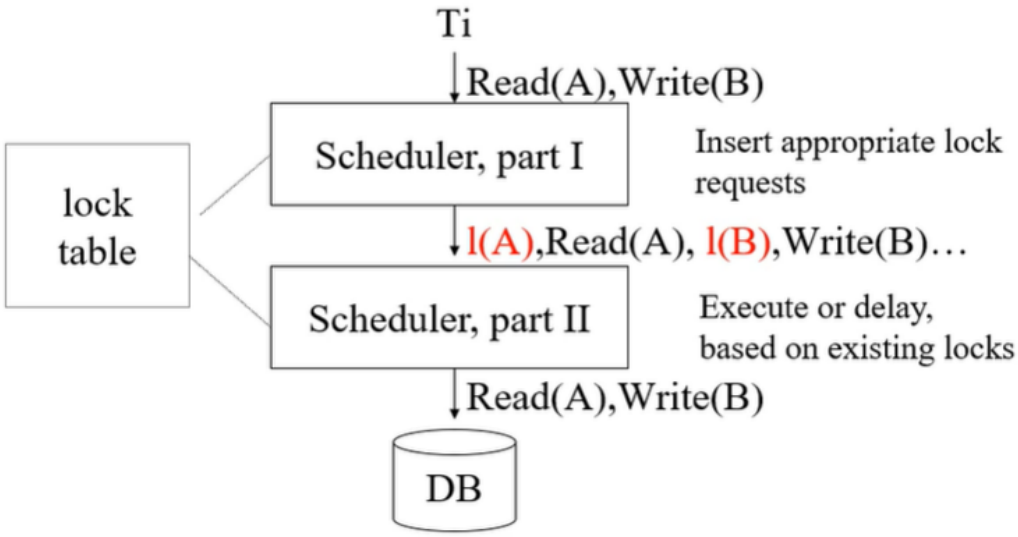
- 如果一个元素有多个读操作，那么一个事务第一个申请互斥锁会失败，但是申请更新锁可以成功

Comp: Lock already held in \ New request	S	X	U
S	T	F	T
X	F	F	F
U	F	F	F

因此一个数据要是想要读A并且知道后续会写A，那么应该申请更新锁

6. 对于一个简单锁机制的实现

- 遇到读写事务的时候，在满足条件的情况下一直加锁不释放锁
- 当事务提交之后一次性释放所有的锁



第一个阶段对事务进行规则1、3检查

第二个阶段对调度进行规则2检查，结合锁表确定哪些动作可以执行

7. Warning Lock
    - 如果一个数据库中的元素要被写操作，则这个元素的所有父节点都要上意向写锁 IX，最后给这个元素上写锁
    - 如果一个数据库中的元素要被读操作，则这个元素的所有父节点都要上意向读锁 IS，最后给这个元素上读锁
    - 兼容矩阵

Holder	IS	IX	S	X
IS	T	T	T	F
IX	T	T	F	F
S	T	F	T	F
X	F	F	F	F

    - 删除操作
        - 对所有的祖先节点上意向写锁
        - 对该元素上写锁
    - 插入操作
        - 由于插入的时候该元素没有在数据库里，因此没有办法对其上锁，成为幻影元组（phantom tuples）
        - 需要对其父节点上写锁

- Use multiple granularity tree
- Before insert of node Q,
  lock parent(Q) in X mode