数据库期末复习（自用）国家一级假勤奋大学生整理

数据库概述

数据库定义：长期存储在计算机内、有组织的、可共享的大量数据集合

数据库特定：

- 数据按一定的数据模型组织、描述和存储
- 可为各种用户共享
- 冗余度较小（节约，不浪费）
- 数据独立性较高
- 易扩展

数据：数据库中存储的基本对象

数据的定义：描述事物的符号记录

信息：数据+语义

数据是信息的符号表示或称载体

信息是数据的内涵，是对数据语义的解释

设置

```
设置UNIQUE
CREATE TABLE Beers
(   name  CHAR(20) UNIQUE,
    manf     CHAR(20)
);

CREATE TABLE Beers
(
    name CHAR(20),
    manf CHAR(20),
    PRIMARY KEY(name)
);
联合主键
CREATE TABLE Beers
(
    name CHAR(20),
    manf CHAR(20),
    PRIMARY KEY(name, manf)
);

DROP TABLE Beers;

ALTER TABLE Sells ADD discount float;
ALTER TABLE Sells DROP discount;
ALTER TABLE Sells ADD discout float DEFAULT 0.0;
设置默认值
```

PRIMARY KEY VS UNIQUE

- 一个关系只能有一个主键，但是可以有多个UNIQUE属性
- 设为主键的属性不能为NULL，但是设为UNIQUE的可以为NULL

关系代数

- union： R∪S
- intersection：R∩S
- difference：差集 R-S，在R中不在S中
- selection：选择：选择特定行

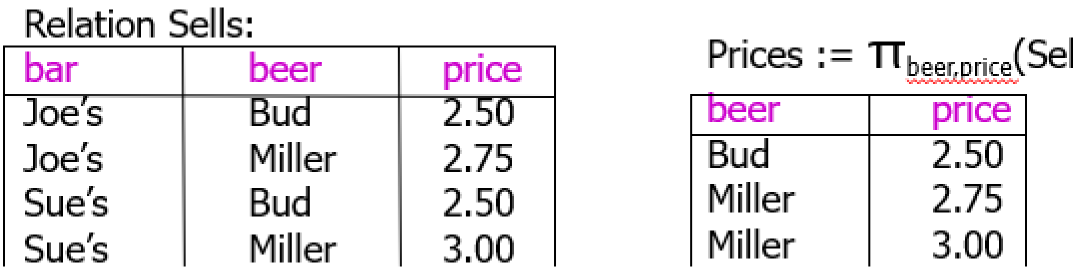
$$R1 := \sigma_C(R2)$$

C代表了一个条件，R1则是满足该条件的R2的所有元组

- projection：投影：选择特定列

$$R1 := \pi_L(R2)$$

L表示R2关系中的某个属性的一列（完全一致重复的会被DROP掉）



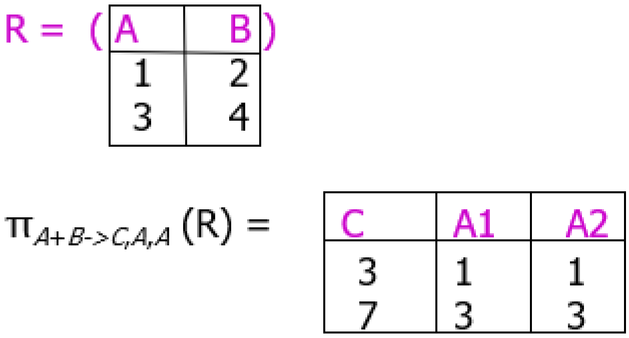
Relation Sells:

bar	beer	price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Miller	3.00

$\text{Prices} := \pi_{beer,price}(\text{Sel}$

beer	price
Bud	2.50
Miller	2.75
Miller	3.00

允许L中包含涉及属性的任一表达式



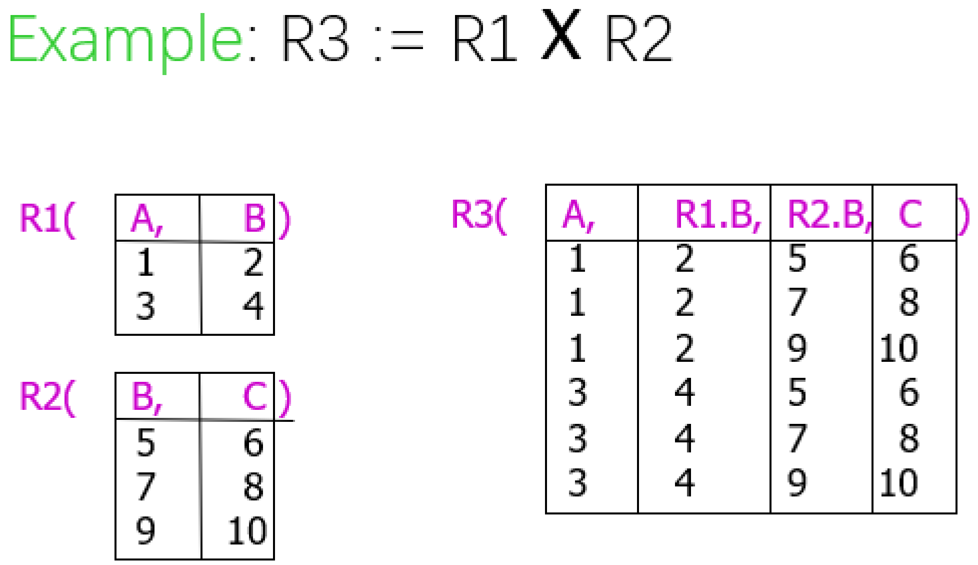
R =

A	B
1	2
3	4

$\pi_{A+B\text{->}C,A,A}(R) =$

C	A1	A2
3	1	1
7	3	3

- products：关系的笛卡尔乘积（叉积）R1XR2



Example: R3 := R1 X R2

R1(

A,	B)
1	2
3	4

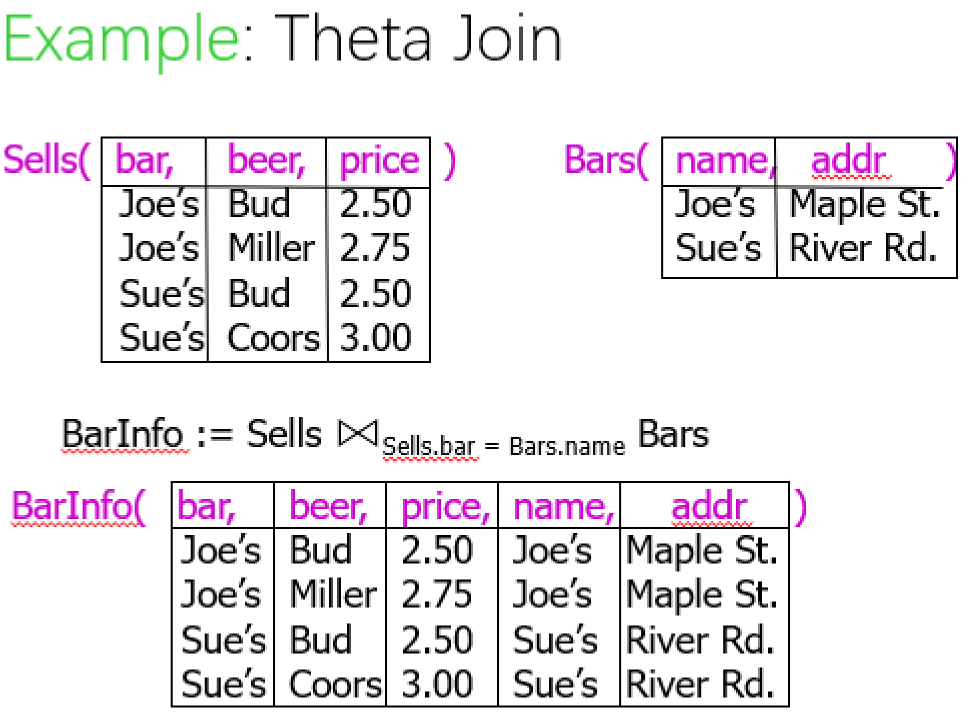
R2(

B,	C)
5	6
7	8
9	10

R3(

A,	R1.B,	R2.B,	C)
1	2	5	6
1	2	7	8
1	2	9	10
3	4	5	6
3	4	7	8
3	4	9	10

- joins：关系的连接
  - θ连接——先做笛卡尔乘积，然后做选择selection



Example: Theta Join

Sells(

bar,	beer,	price)
Joe’s	Bud	2.50
Joe’s	Miller	2.75
Sue’s	Bud	2.50
Sue’s	Coors	3.00

Bars(

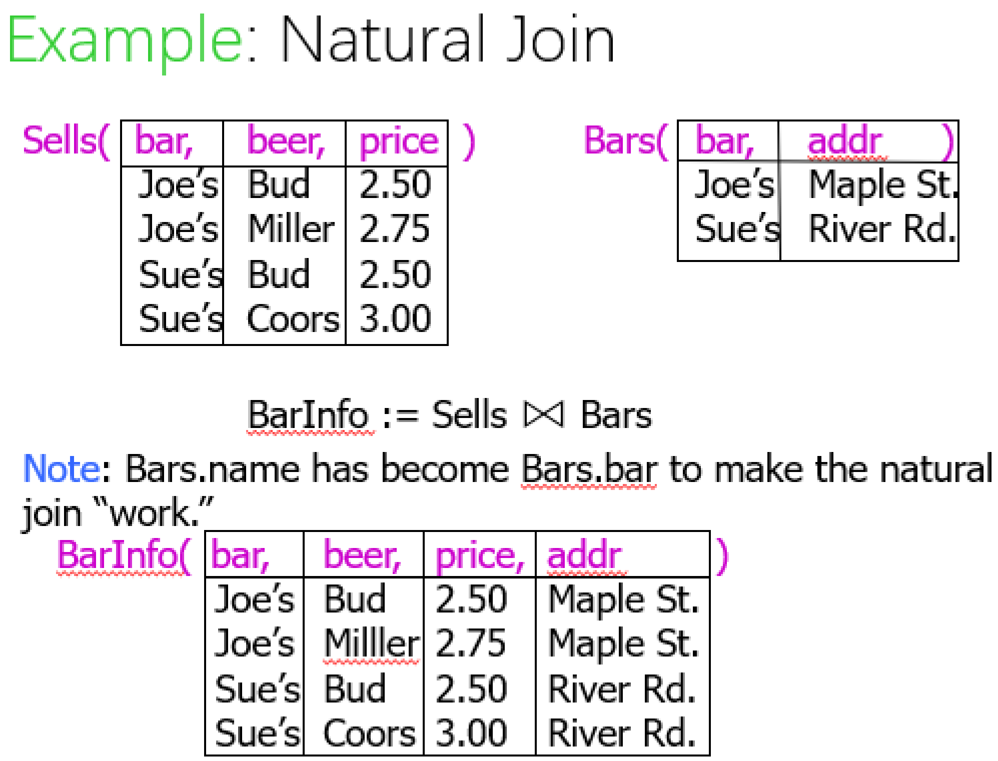
name,	addr)
Joe’s	Maple St.
Sue’s	River Rd.

BarInfo := Sells $\bowtie_{Sells.bar\ =\ Bars.name}$ Bars

BarInfo(

bar,	beer,	price,	name,	addr)
Joe’s	Bud	2.50	Joe’s	Maple St.
Joe’s	Miller	2.75	Joe’s	Maple St.
Sue’s	Bud	2.50	Sue’s	River Rd.
Sue’s	Coors	3.00	Sue’s	River Rd.

  - 自然连接



Example: Natural Join

Sells(

bar,	beer,	price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Coors	3.00

)

Bars(

bar,	addr
Joe's	Maple St.
Sue's	River Rd.

)

BarInfo := Sells ⋈ Bars

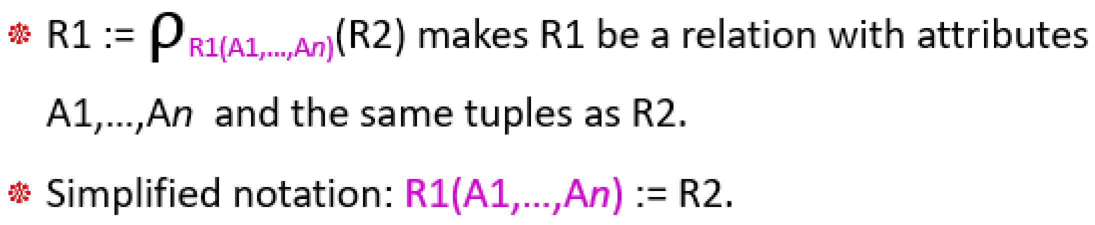
Note: Bars.name has become Bars.bar to make the natural join "work."

BarInfo(

bar,	beer,	price,	addr
Joe's	Bud	2.50	Maple St.
Joe's	Milller	2.75	Maple St.
Sue's	Bud	2.50	River Rd.
Sue's	Coors	3.00	River Rd.

)

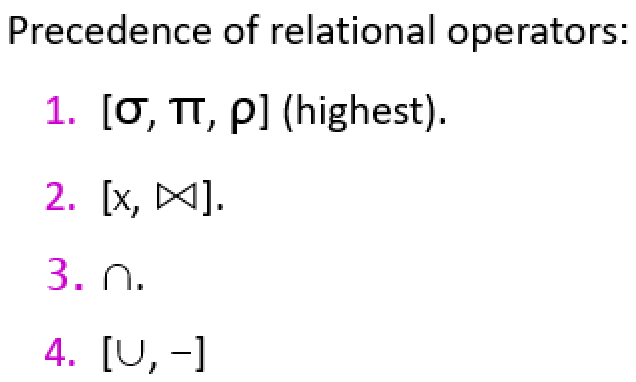
- renaming：关系或属性的重命名



- R1 := $\rho_{R1(A1,\ldots,An)}$(R2) makes R1 be a relation with attributes A1,…,An and the same tuples as R2.
- Simplified notation: R1(A1,…,An) := R2.

两种表示形式

上述关系代数的优先级：



Precedence of relational operators:

1. [σ, π, ρ] (highest).
2. [x, ⋈].
3. ∩.
4. [∪, –]

SQL是bag language而不是set language

一些集合适用的法则，bag不一定适用，比如幂等律

SQL

SQL： structured query language

```
SELECT * FROM table WHERE condition
```

- 从FROM开始

- 应用WHERE子句指示的选择
- 应用SELECT子句指示的扩展投影

AS to rename attribute or schema

任何有意义的表达式都可以作为SELECT子句的元素

```
用于condition的条件：
AND OR NOT
= <> (!=) < > >= <=
```

```
Pttern:
<Attribute> LIKE <pattern> or <Attribute> NOT LIKE <pattern>
% 任何字符串
_ 任何字符
想要匹配%该字符需要使用关键字escape
Match the string “100 %”

like ‘_100 \%_'  escape  '\'
```

NULL的含义：

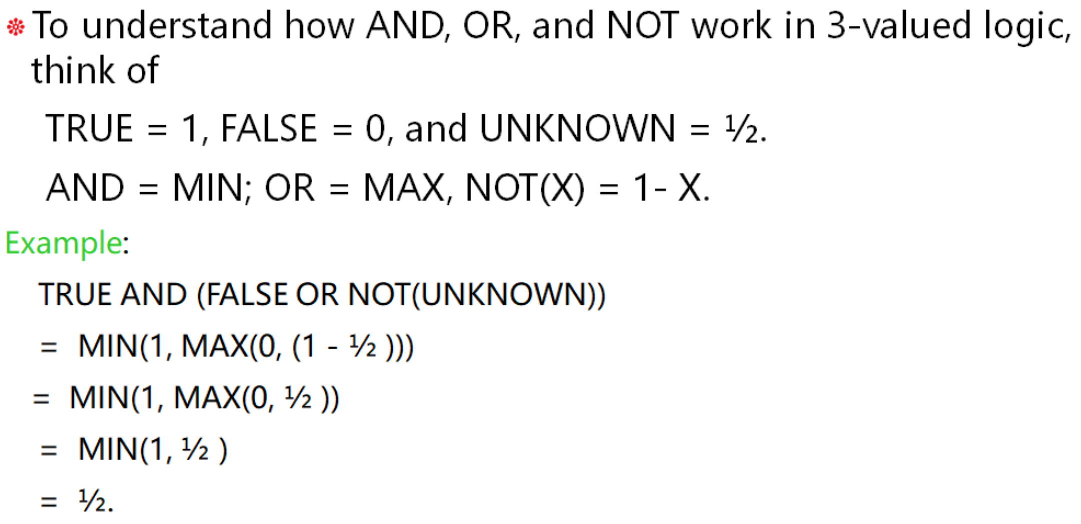
- 缺失值
- 不适用的值：比如配偶属性无法给一个未婚的人
- 不支持的值：未在电话簿中记载的电话属性

NULL的算数运算结果仍然是NULL

NULL和任何值的比较结果为UNKNOWN

WHERE子句为TRUE时，查询有结果

如何判断属性的值是否为NULL： x is NULL



❉ To understand how AND, OR, and NOT work in 3-valued logic, think of

TRUE = 1, FALSE = 0, and UNKNOWN = ½.

AND = MIN; OR = MAX, NOT(X) = 1- X.

Example:

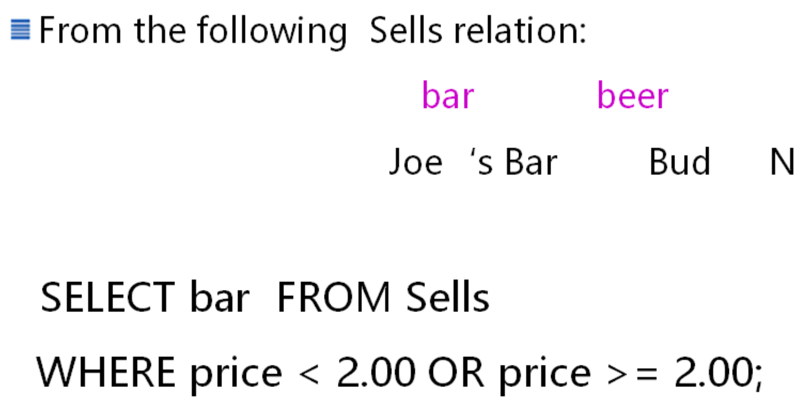
TRUE AND (FALSE OR NOT(UNKNOWN))

= MIN(1, MAX(0, (1 - ½)))

= MIN(1, MAX(0, ½))

= MIN(1, ½)

= ½.



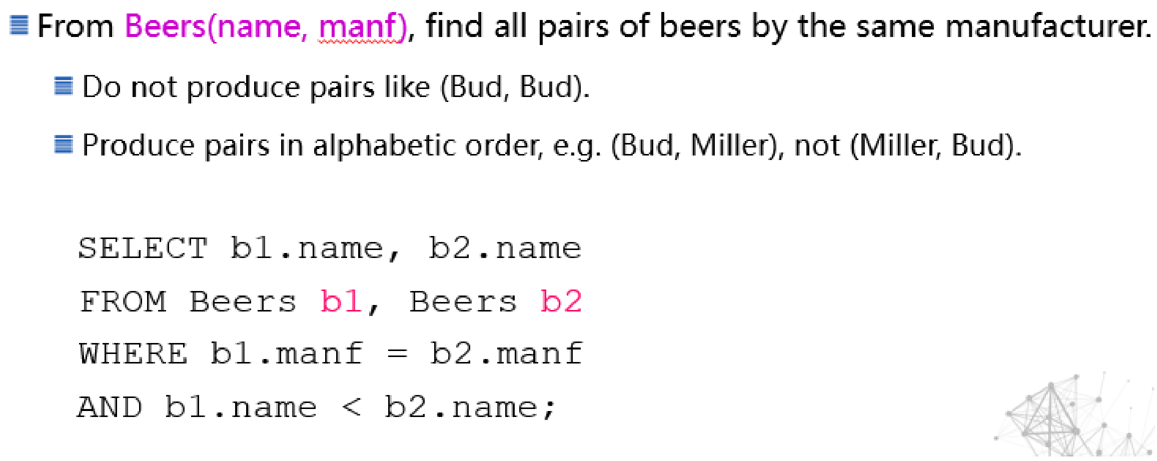
- From the following Sells relation:

bar	beer	price
Joe 's Bar	Bud	NULL

```
SELECT bar FROM Sells
WHERE price < 2.00 OR price >= 2.00;
```

```
排序ORDER BY attribute ASC/DESC 默认ASC
```

self-join——自连接的应用



- From Beers(name, manf), find all pairs of beers by the same manufacturer.
  - Do not produce pairs like (Bud, Bud).
  - Produce pairs in alphabetic order, e.g. (Bud, Miller), not (Miller, Bud).

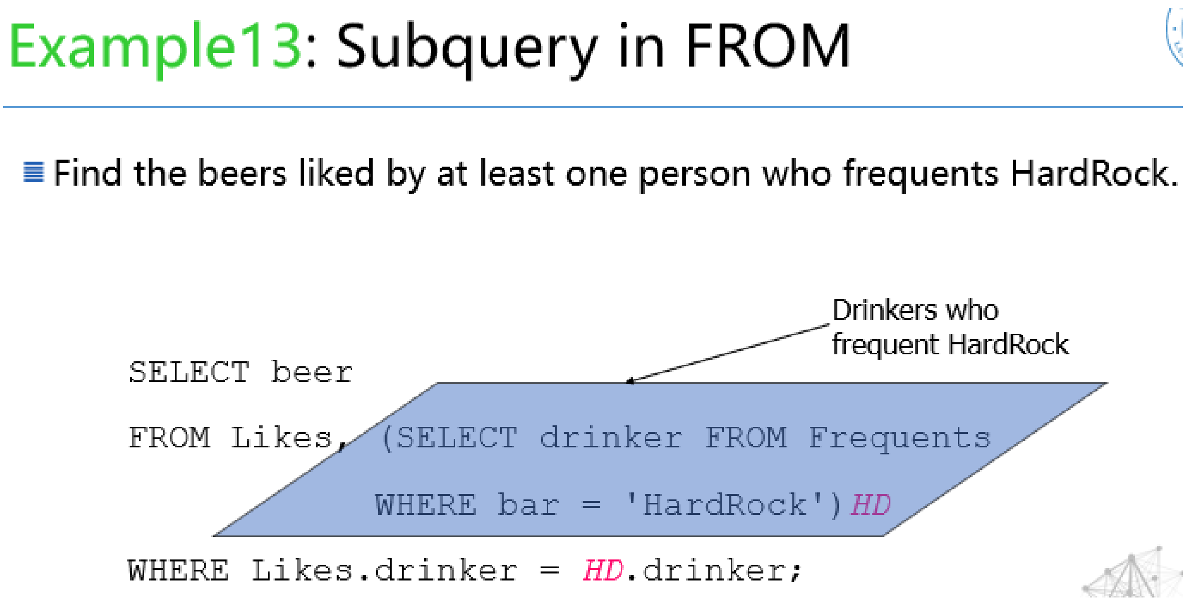
```
SELECT b1.name, b2.name
FROM Beers b1, Beers b2
WHERE b1.manf = b2.manf
AND b1.name < b2.name;
```

```
SQL:交并补 UNION INTERSECT EXCEPT
```

子查询 subqueries

SELECT-FROM-WHERE就是一个子查询

子查询可以当作**值**或**关系**在FROM或WHERE中使用



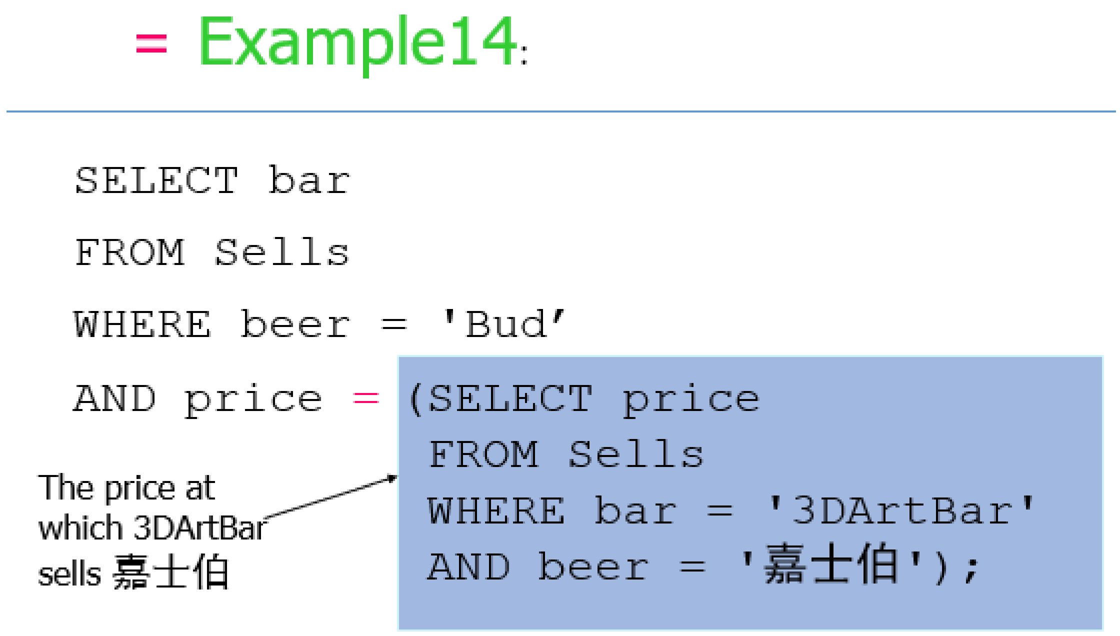
Example13: Subquery in FROM

- Find the beers liked by at least one person who frequents HardRock.

```
SELECT beer
FROM Likes, (SELECT drinker FROM Frequents
             WHERE bar = 'HardRock') HD
WHERE Likes.drinker = HD.drinker;
```

子查询在WHERE的应用

- 子查询可以用作value或set
- 如果子查询保证生成一个只有一个属性的元组，则用作值：



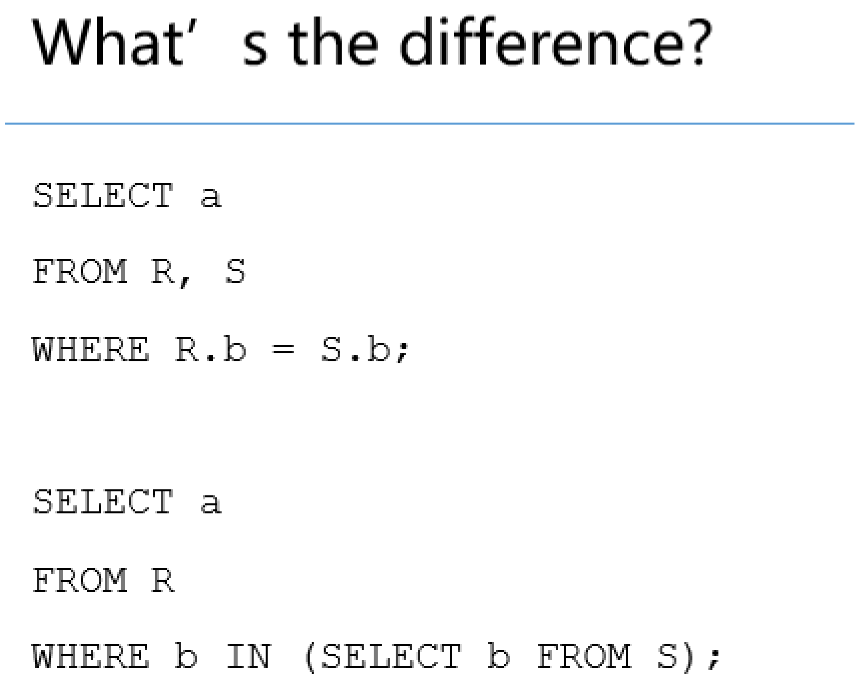
= Example14:

```
SELECT bar
FROM Sells
WHERE beer = 'Bud'
AND price = (SELECT price
             FROM Sells
             WHERE bar = '3DArtBar'
             AND beer = '嘉士伯');
```

The price at which 3DArtBar sells 嘉士伯

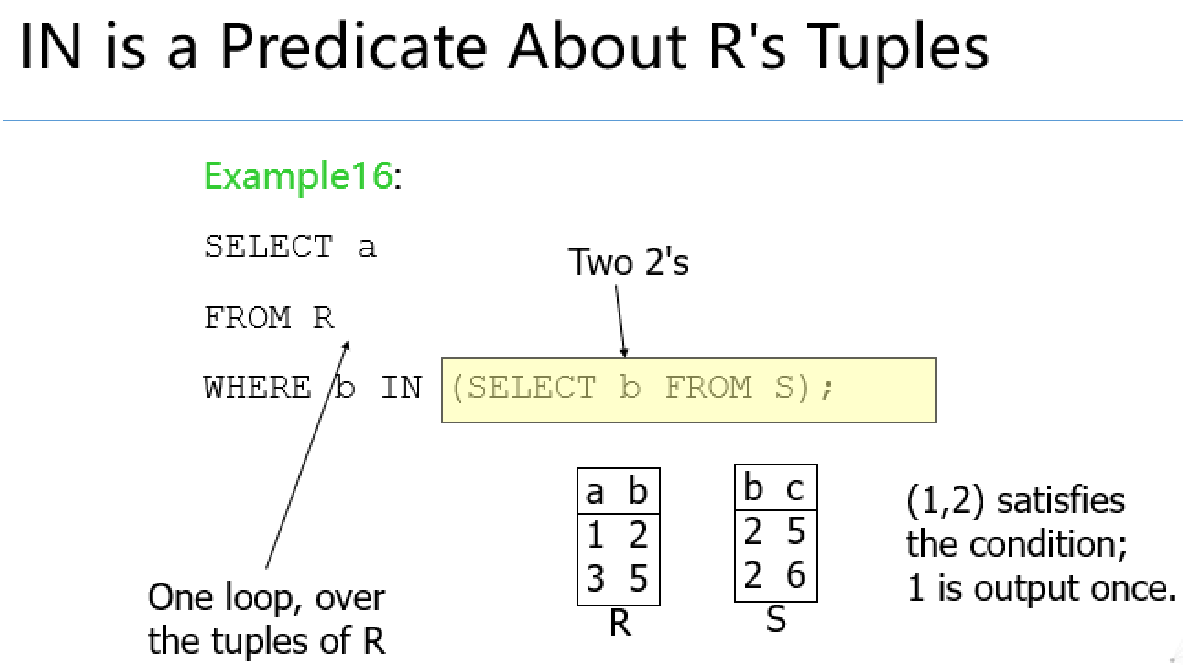
```
IN 关键字
<tuple> IN <subquery> 为true，当tuple中存在subquery中的成员
```

IN可以出现在WHERE子句中



What' s the difference?

```
SELECT a
FROM R, S
WHERE R.b = S.b;


SELECT a
FROM R
WHERE b IN (SELECT b FROM S);
```



IN is a Predicate About R's Tuples

Example16:

```
SELECT a
FROM R
WHERE b IN (SELECT b FROM S);
```

Two 2's

One loop, over the tuples of R

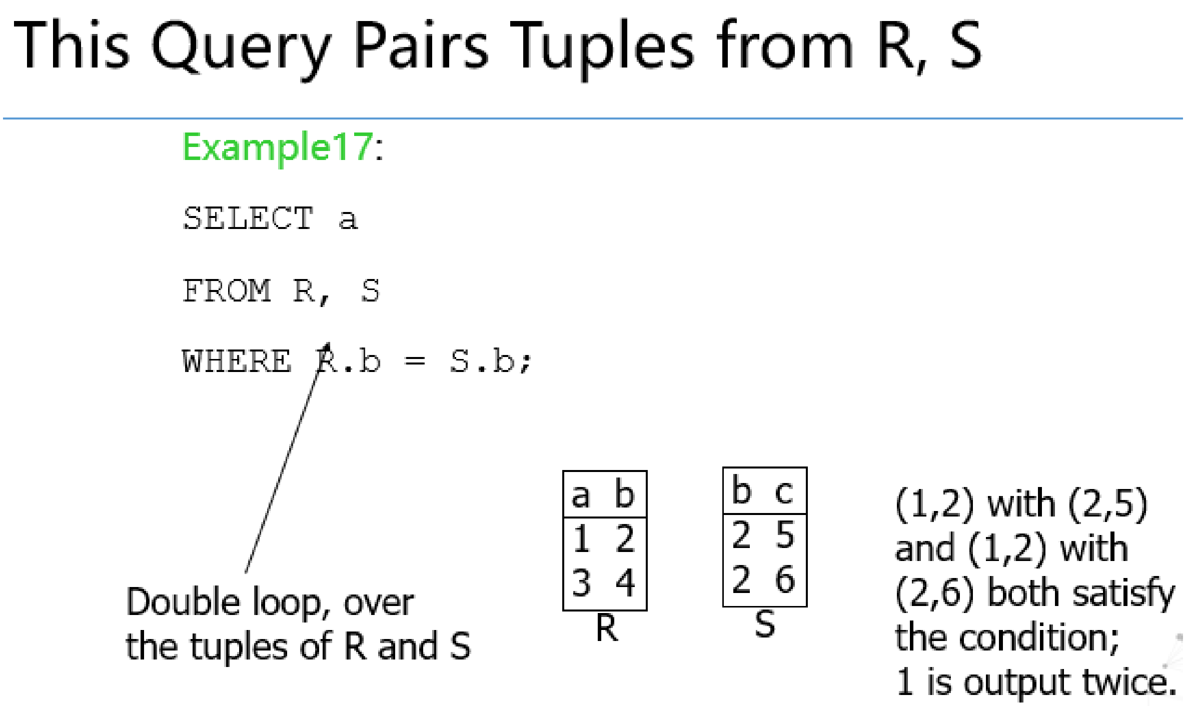
a	b
1	2
3	5

R

b	c
2	5
2	6

S

(1,2) satisfies the condition; 1 is output once.



This Query Pairs Tuples from R, S

Example17:

```
SELECT a
FROM R, S
WHERE R.b = S.b;
```

Double loop, over the tuples of R and S

a	b
1	2
3	4

R

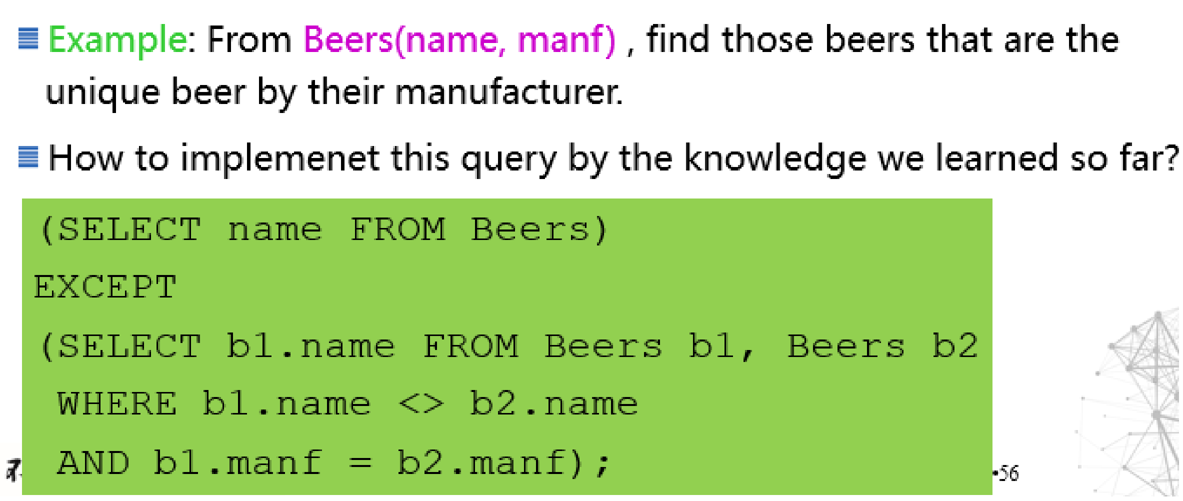
b	c
2	5
2	6

S

(1,2) with (2,5) and (1,2) with (2,6) both satisfy the condition; 1 is output twice.

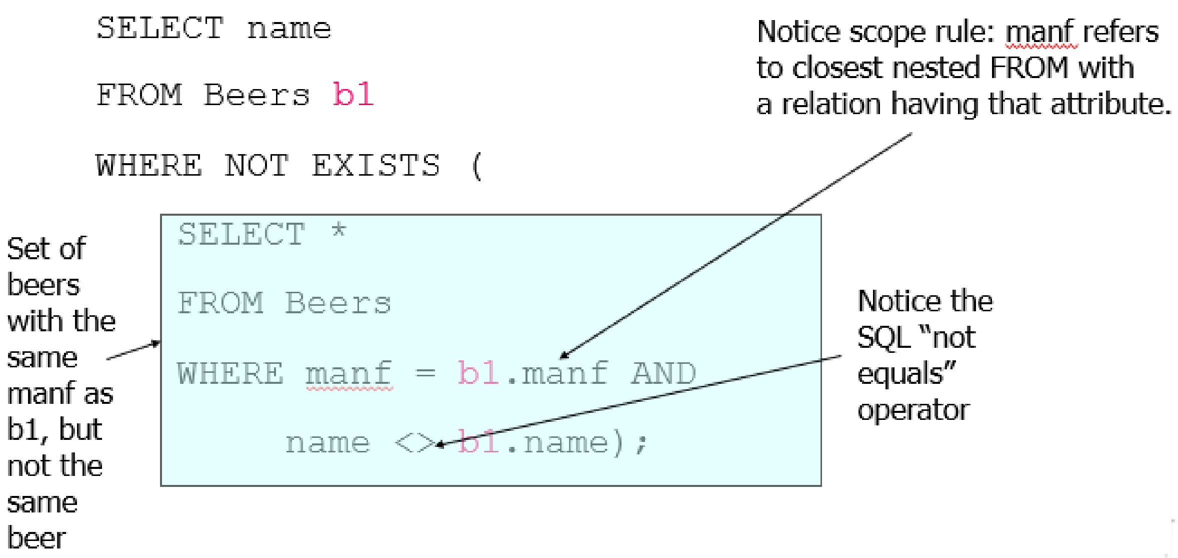
EXISTS

```
EIXTS(<subquery>)为真当且仅当子查询的结构非空
```



- Example: From Beers(name, manf) , find those beers that are the unique beer by their manufacturer.
- How to implemenet this query by the knowledge we learned so far?

```
(SELECT name FROM Beers)
EXCEPT
(SELECT b1.name FROM Beers b1, Beers b2
 WHERE b1.name <> b2.name
 AND b1.manf = b2.manf);
```



```
SELECT name
FROM Beers b1
WHERE NOT EXISTS (
    SELECT *
    FROM Beers
    WHERE manf = b1.manf AND
          name <> b1.name);
```

Notice scope rule: manf refers to closest nested FROM with a relation having that attribute.

Set of beers with the same manf as b1, but not the same beer

Notice the SQL "not equals" operator

EXISTS for Example 15

- Example15:Using Beers(name, manf) and Likes(drinker, beer), find the name and manufacturer of each beer that Tony Hoare likes.

```
SELECT * FROM Beers
WHERE EXISTS (
    SELECT * FROM Likes
    WHERE drinker = 'Tony Hoare'
     AND Likes.beer = Beers.name);
```

- From Sells(bar, beer, price), find the beer(s) sold for the highest price.

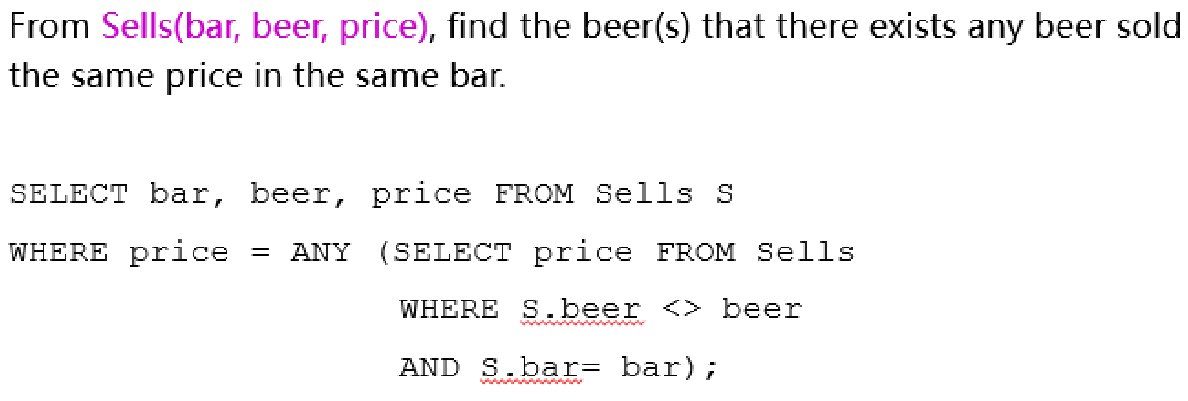
```
SELECT beer FROM Sells S
WHERE NOT EXISTS (SELECT * FROM Sells
                      WHERE price > S.price);
```

- From Sells(bar, beer, price), find the beer(s) that there exists any beer sold the same price in the same bar.

```
SELECT * FROM Sells S
WHERE EXISTS (SELECT * FROM Sells
              WHERE price = S.price
              AND S.beer <> beer AND S.bar= bar);
```

ANY

```
x = ANY(<subquery>)如果x至少等于子查询结果中的一个元组，则为true
x >= ANY(<subquery>)表示x不是子查询生成的唯一最小元组
```



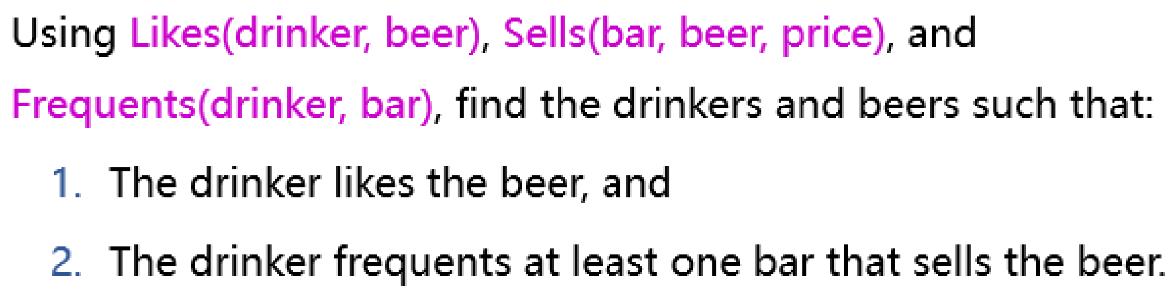
From Sells(bar, beer, price), find the beer(s) that there exists any beer sold the same price in the same bar.

```
SELECT bar, beer, price FROM Sells S
WHERE price = ANY (SELECT price FROM Sells
                        WHERE S.beer <> beer
                        AND S.bar= bar);
```

ALL

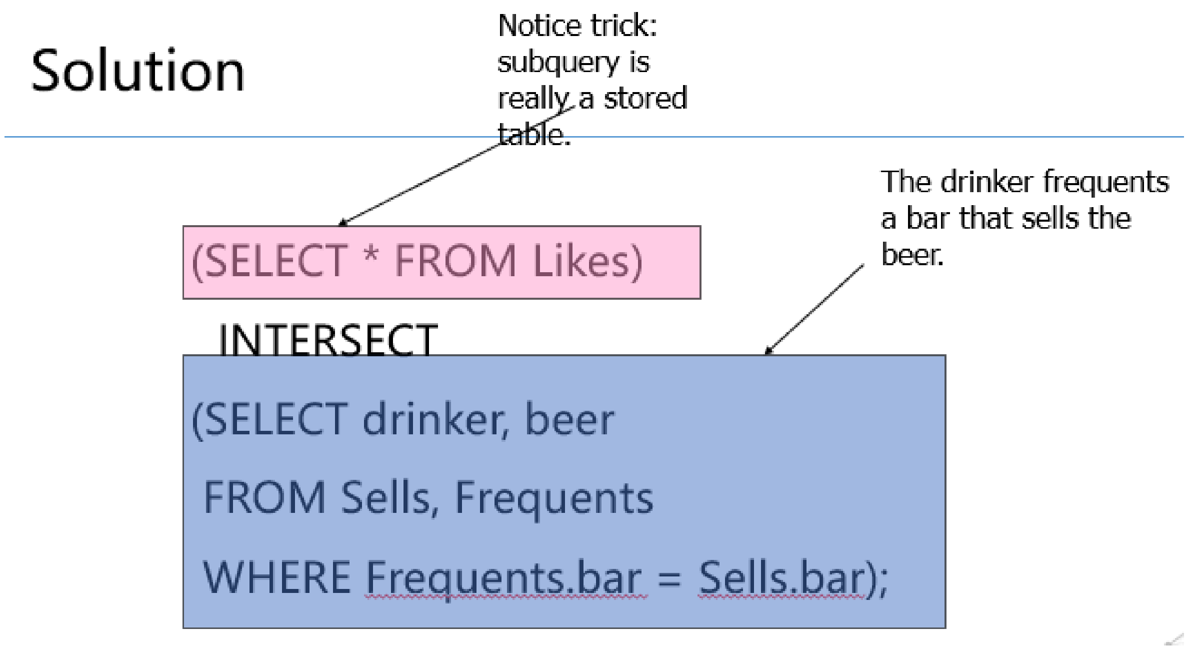
```
X<>ALL(<subquery>)为true，如果对于关系中的每个元组t，t都不等于x
```

```
(<subquery>) UNION (<subquery>)
(<subquery>) INTERSECT (<subquery>)
(<subquery>) EXCEPT (<subquery>)
```



Using Likes(drinker, beer), Sells(bar, beer, price), and Frequents(drinker, bar), find the drinkers and beers such that:

1. The drinker likes the beer, and
2. The drinker frequents at least one bar that sells the beer.



Solution

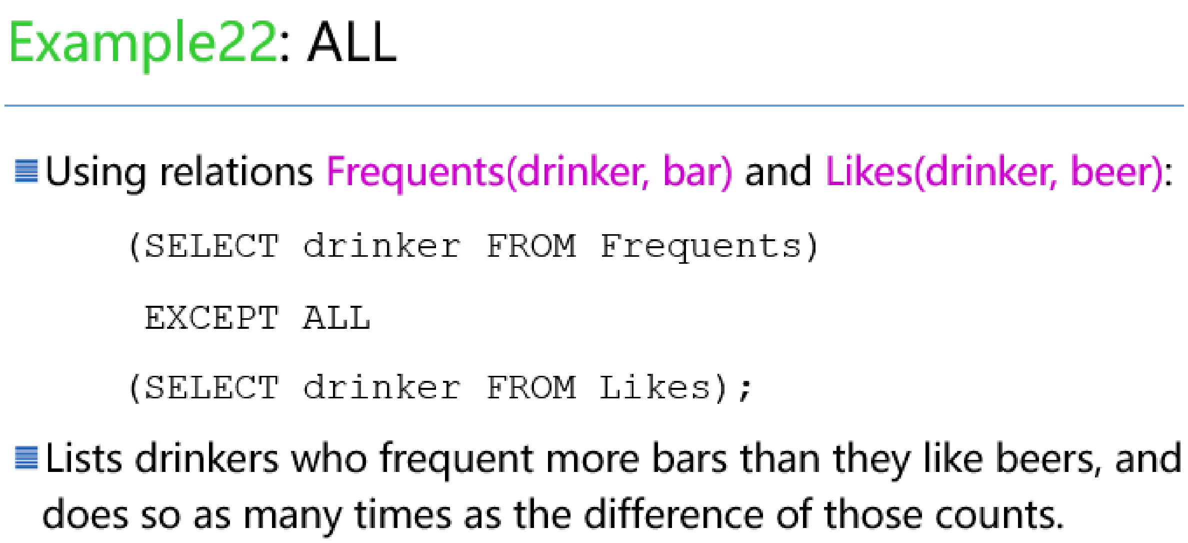
Notice trick: subquery is really a stored table.

The drinker frequents a bar that sells the beer.

```
(SELECT * FROM Likes)
INTERSECT
(SELECT drinker, beer
 FROM Sells, Frequents
 WHERE Frequents.bar = Sells.bar);
```

交并补的默认值是set语义，所以在应用操作时会消除重复率

DISTINCT去重



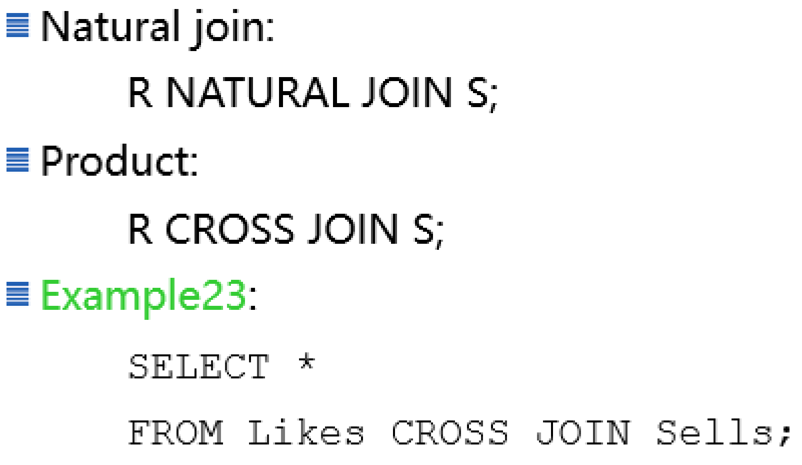
Example22: ALL

- Using relations Frequents(drinker, bar) and Likes(drinker, beer):

```
    (SELECT drinker FROM Frequents)
     EXCEPT ALL
    (SELECT drinker FROM Likes);
```

- Lists drinkers who frequent more bars than they like beers, and does so as many times as the difference of those counts.

关系也可以是带括号的子查询



- Natural join:

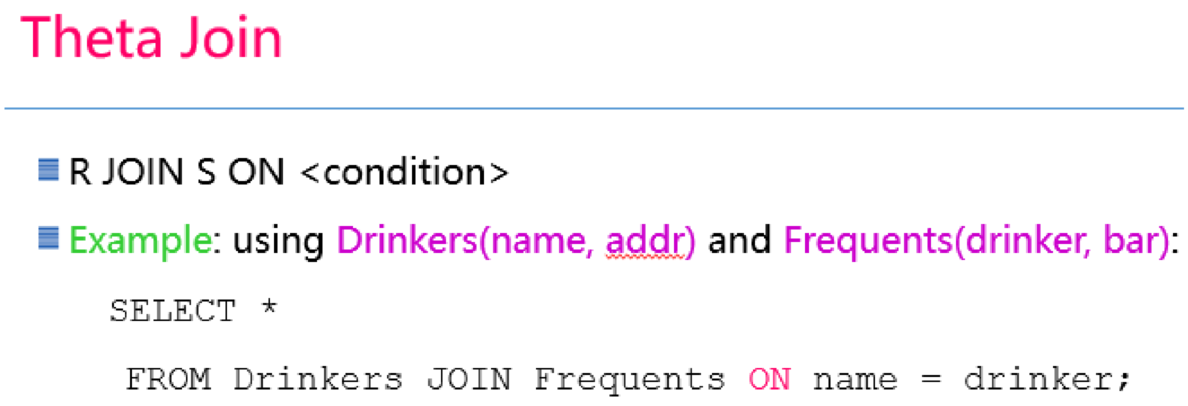
  R NATURAL JOIN S;

- Product:

  R CROSS JOIN S;

- Example23:

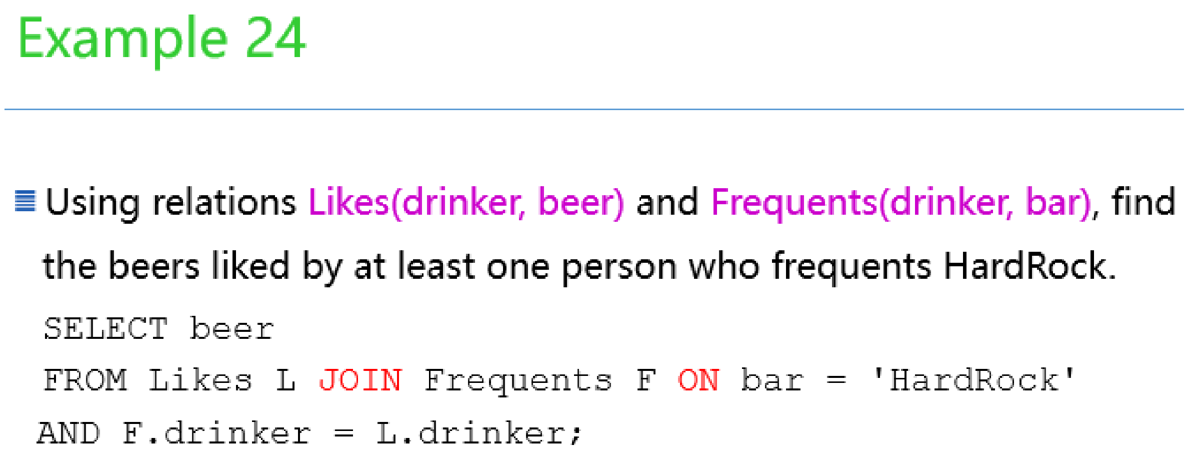
```
SELECT *
FROM Likes CROSS JOIN Sells;
```



Theta Join

- R JOIN S ON <condition>
- Example: using Drinkers(name, addr) and Frequents(drinker, bar):

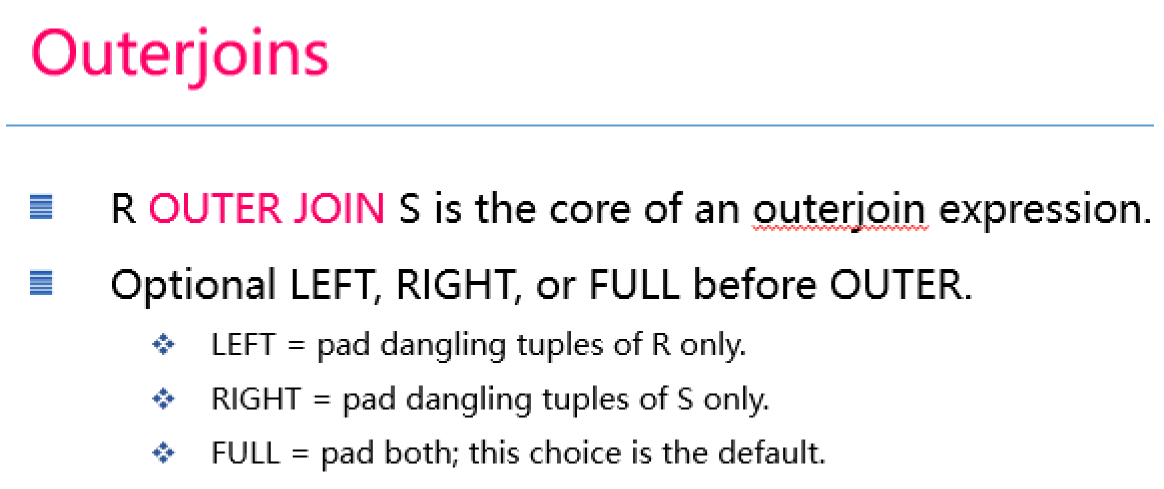
```
SELECT *
 FROM Drinkers JOIN Frequents ON name = drinker;
```



Example 24

- Using relations Likes(drinker, beer) and Frequents(drinker, bar), find the beers liked by at least one person who frequents HardRock.

```
SELECT beer
FROM Likes L JOIN Frequents F ON bar = 'HardRock'
AND F.drinker = L.drinker;
```



Outerjoins

- R OUTER JOIN S is the core of an outerjoin expression.
- Optional LEFT, RIGHT, or FULL before OUTER.
  - LEFT = pad dangling tuples of R only.
  - RIGHT = pad dangling tuples of S only.
  - FULL = pad both; this choice is the default.

R

A	B
1	2
3	4
5	6

S

B	C
1	0
2	4
6	8

```
SELECT A, C
FROM R JOIN S
ON R.B = S.B;
```

A	C
1	4
5	8

```
SELECT A, C
FROM R LEFT JOIN S ON R.B = S.B;
```

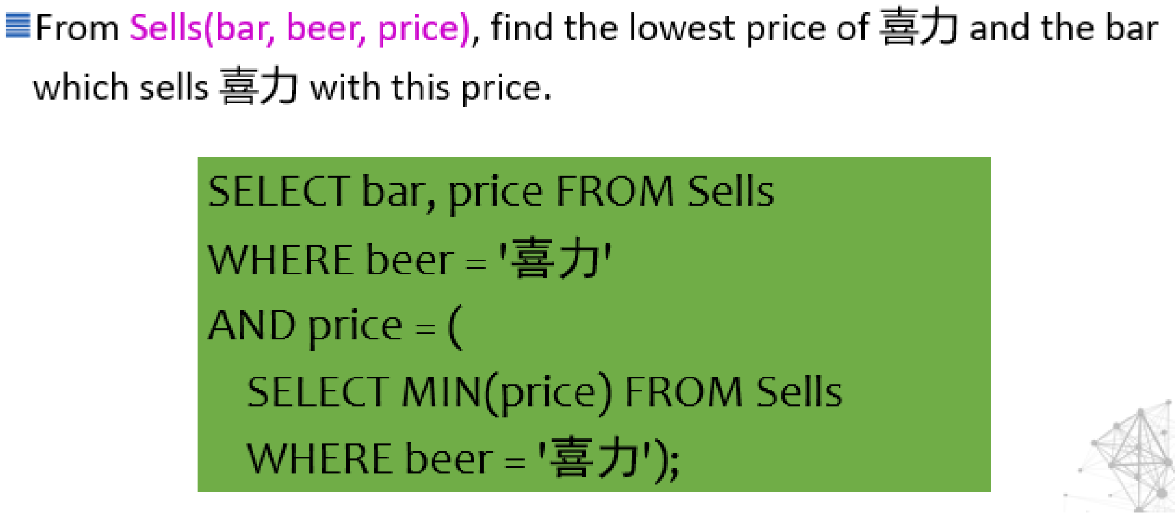
A	C
1	4
3	NULL
5	8

```
SELECT A, C
FROM R FULL JOIN S ON R.B = S.B;
```

A	C
1	4
3	NULL
5	8
NULL	0

聚合函数

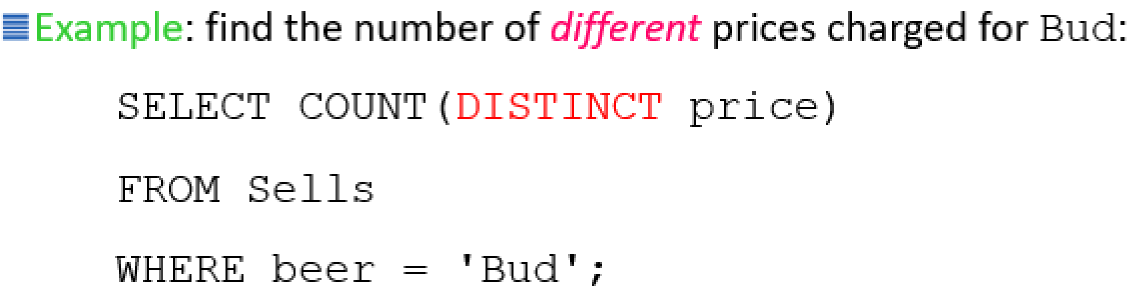
```
SUM AVG COUNT MIN MAX
```



- From Sells(bar, beer, price), find the lowest price of 喜力 and the bar which sells 喜力 with this price.

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

在聚合函数中可以使用DISTINCT进行去重



- Example: find the number of *different* prices charged for Bud:

```
SELECT COUNT(DISTINCT price)
FROM Sells
WHERE beer = 'Bud';
```

NULL属性会被聚合函数所忽略

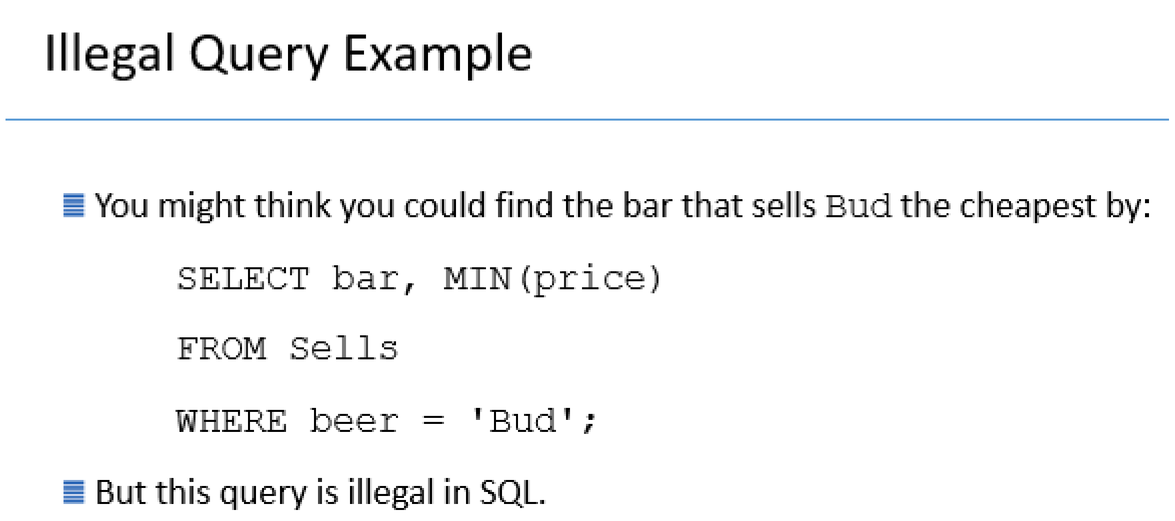
如果一列中没用非NULL属性则聚合的结果为NULL——除了COUNT空集为0

GROUP BY分组

聚合函数可以用于每个分组中

如果聚合函数被使用，则SELECT中仅能出现：

- 聚合函数
- GROUP BY分组的属性



Illegal Query Example

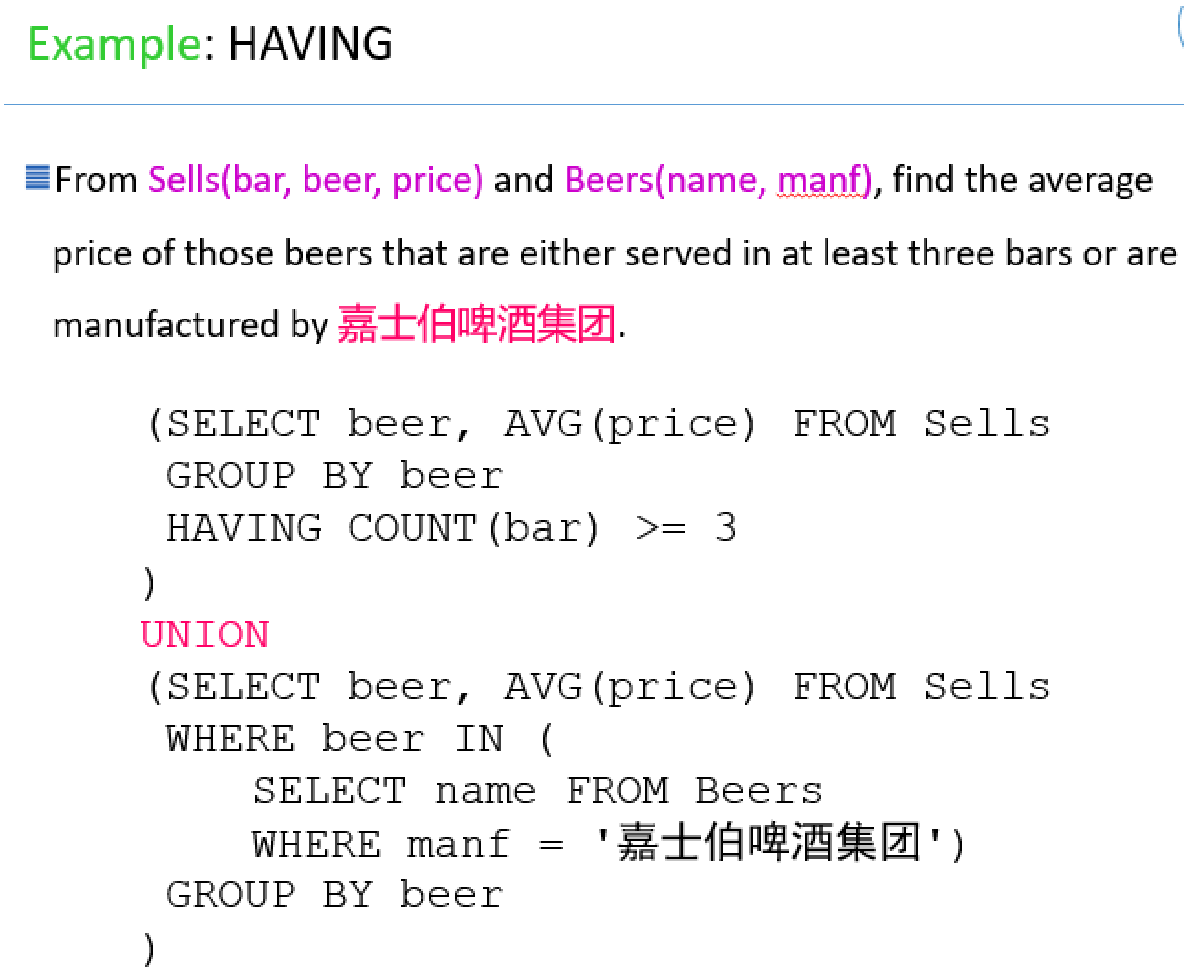
- You might think you could find the bar that sells Bud the cheapest by:

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

- But this query is illegal in SQL.

使用HAVING给GROUP BY添加约束

HAVING中列出的条件要适用于每个组，不满足条件的组将会被剔除



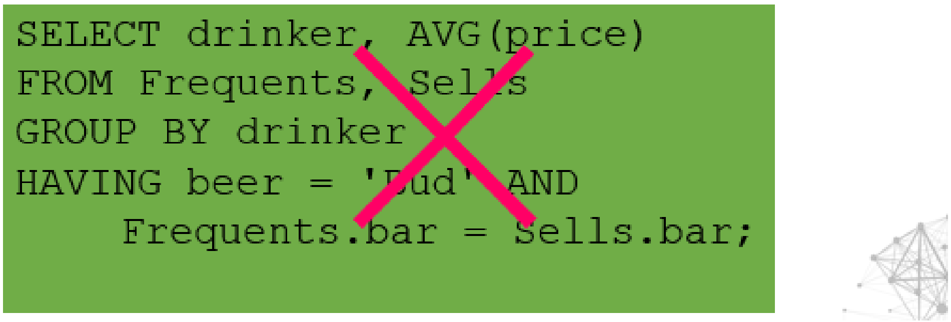
Example: HAVING

- From Sells(bar, beer, price) and Beers(name, manf), find the average price of those beers that are either served in at least three bars or are manufactured by 嘉士伯啤酒集团.

```
(SELECT beer, AVG(price) FROM Sells
 GROUP BY beer
 HAVING COUNT(bar) >= 3
)
UNION
(SELECT beer, AVG(price) FROM Sells
 WHERE beer IN (
     SELECT name FROM Beers
     WHERE manf = '嘉士伯啤酒集团')
 GROUP BY beer
)
```

HAVING中的子查询只能涉及GROUP BY的属性或聚合函数——和SELECT中的情况一致

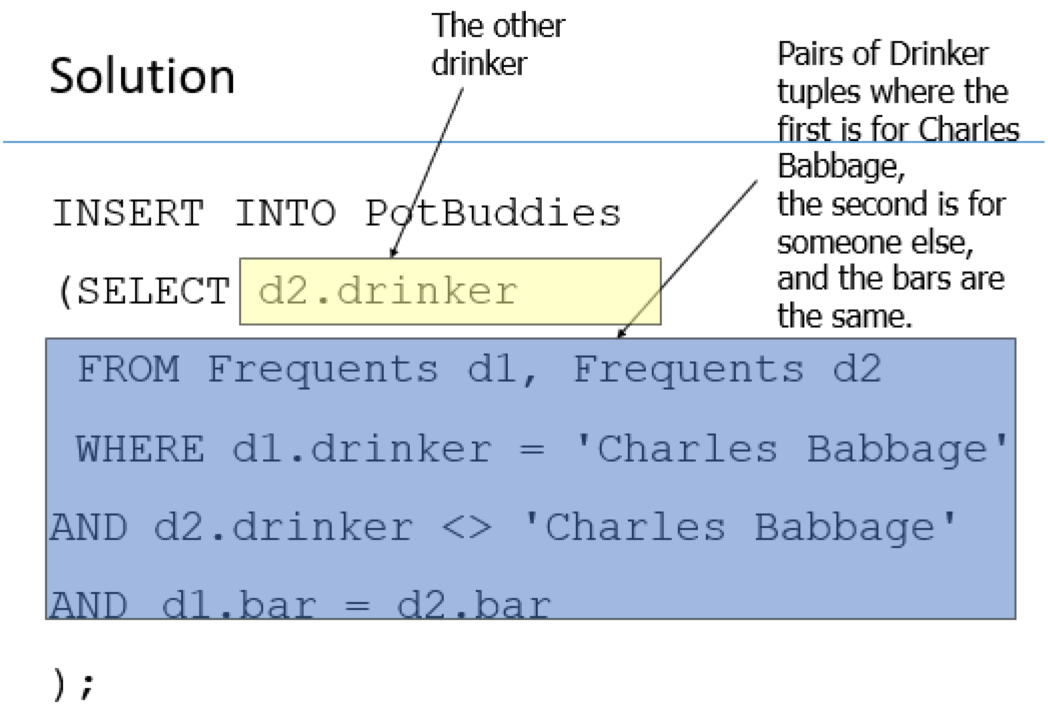
- From Sells(bar, beer, price) and Frequents(drinker, bar), find for each drinker the average price of Bud at the bars they frequent:



```
SELECT drinker, AVG(price)
FROM Frequents, Sells
GROUP BY drinker
HAVING beer = 'Bud' AND
     Frequents.bar = Sells.bar;
```

```
INSERT INTO <relation> (...) ( <subquery> );
可能需要在关系后面添加上字段：
我们可能忘记关系中属性的顺序
可能不必为每个字段赋值，想让系统填充缺失的字段（NULL OR DEFAULT）
VALUES中也可以是子查询
```

- Using Frequents(drinker, bar), enter into the new relation PotBuddies(name) all of Charles Babbage's "potential buddies", i.e., those drinkers who frequent at least one bar that Charles Babbage also frequents.



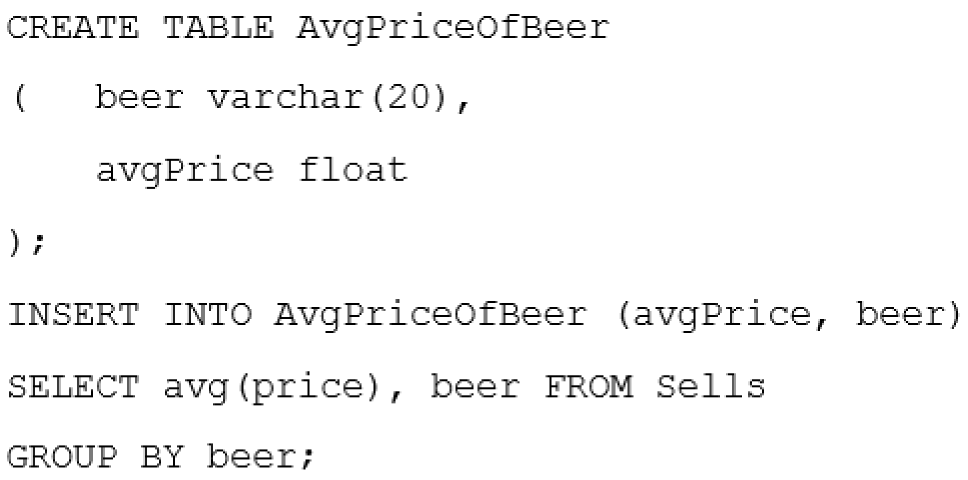
Solution

```
INSERT INTO PotBuddies
(SELECT d2.drinker
 FROM Frequents d1, Frequents d2
 WHERE d1.drinker = 'Charles Babbage'
AND d2.drinker <> 'Charles Babbage'
AND d1.bar = d2.bar
);
```

The other drinker

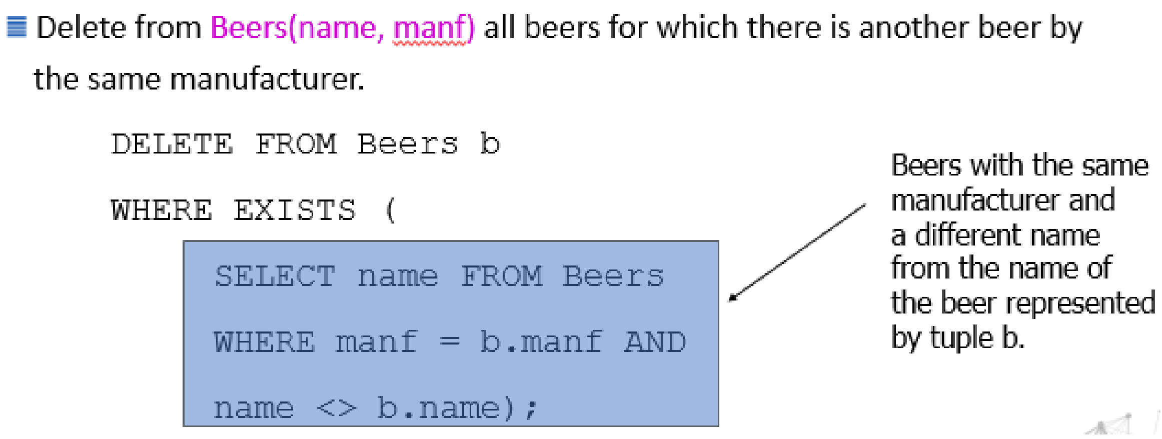
Pairs of Drinker tuples where the first is for Charles Babbage, the second is for someone else, and the bars are the same.

- Relation AvgPriceOfBeer(beer, avgPrice) is used to store the average price of each beer.
- Fill the ralation by means of insert a subquery operation.



```
CREATE TABLE AvgPriceOfBeer
(   beer varchar(20),
    avgPrice float
);
INSERT INTO AvgPriceOfBeer (avgPrice, beer)
SELECT avg(price), beer FROM Sells
GROUP BY beer;
```

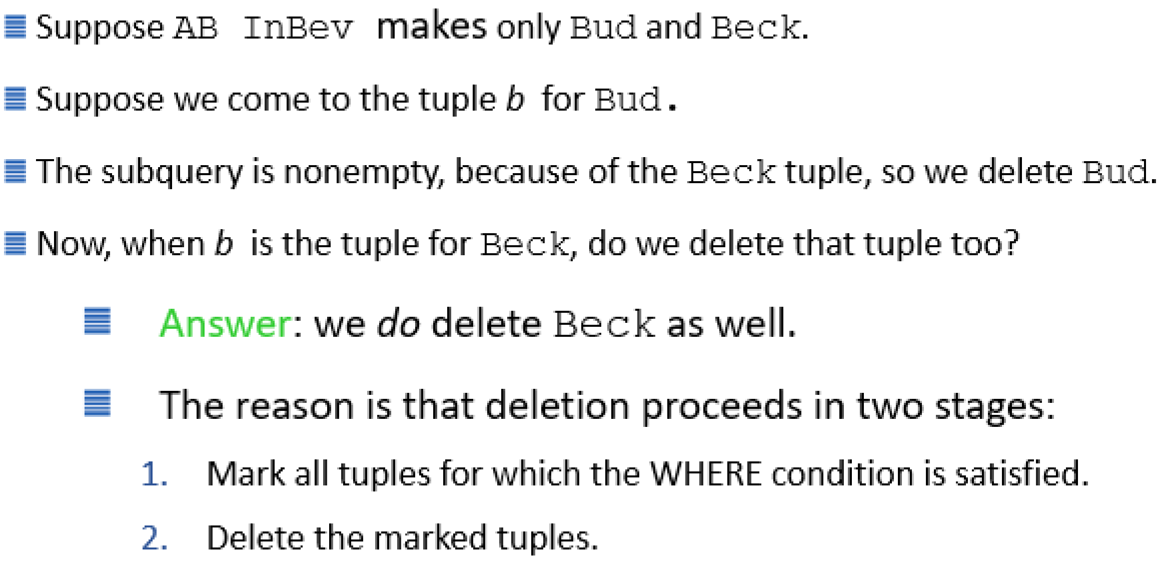
```
DELETE FROM <relation> WHERE <condition>
删除满足条件的一行

DELETE FROM LIKES——将关系LIKES清空（不需要WHERE）
```



- Delete from Beers(name, manf) all beers for which there is another beer by the same manufacturer.

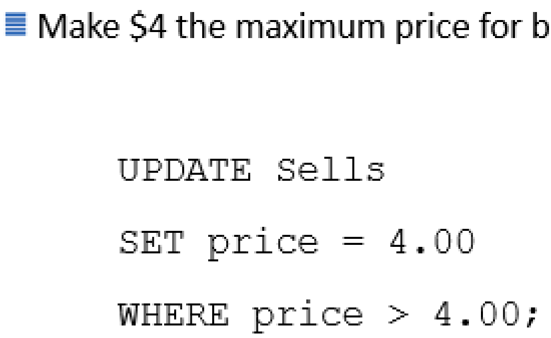
```
DELETE FROM Beers b
WHERE EXISTS (
    SELECT name FROM Beers
    WHERE manf = b.manf AND
    name <> b.name);
```

Beers with the same manufacturer and a different name from the name of the beer represented by tuple b.



- Suppose AB InBev makes only Bud and Beck.
- Suppose we come to the tuple *b* for Bud.
- The subquery is nonempty, because of the Beck tuple, so we delete Bud.
- Now, when *b* is the tuple for Beck, do we delete that tuple too?
  - Answer: we *do* delete Beck as well.
  - The reason is that deletion proceeds in two stages:
    1. Mark all tuples for which the WHERE condition is satisfied.
    2. Delete the marked tuples.

```
UPDATE <relation>
        SET <list of attribute assignments>
        WHERE <condition on tuples>
```



Make $4 the maximum price for beer:

```
UPDATE Sells
SET price = 4.00
WHERE price > 4.00;
```

视图和索引

视图时根据存储表和其他视图定义的关系

- 虚拟视图：不存储在数据库中，一个用于构建关系的查询
- 物理视图：真实构造并存储的视图

```
CREATE VIEW <name> [Attributes] AS <query>;
默认为虚拟视图
```

对视图进行删除、插入、更新操作会影响到原表的结构吗？

对原表进行删除、插入、更新操作会影响到视图的结构吗？

物理视图——

```
CREATE MATERIALIZED VIEW CanDrink2
  AS
        SELECT drinker, beer
        FROM Frequents, Sells
        WHERE Frequents.bar = Sells.bar;

REFRESH MATERIALIZED VIEW CanDrink2;
DROP MATERIALIZED VIEW CanDrink2;
# 增量物化视图
CREATE INCREMENTAL MATERIALIZED VIEW BarBeerView2
AS
SELECT bar, beer FROM Sells
WHERE price > 35;
```

索引

用于加速访问关系元组的数据结构——内部以B树实现

```
CREATE INDEX SellInd ON Sells(bar, beer);
```

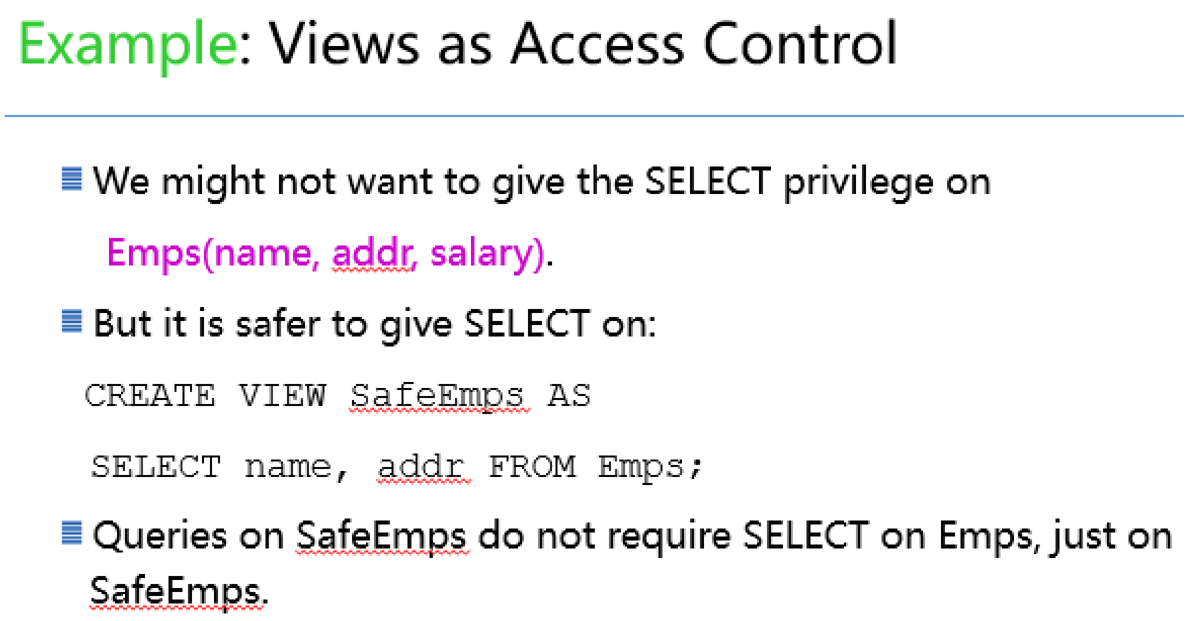
对经常访问的字段建立索引

安全与用户认证

增删改查四种重要权限

存在权限的对象包括存储的表和视图

视图是权限控制的重要工具



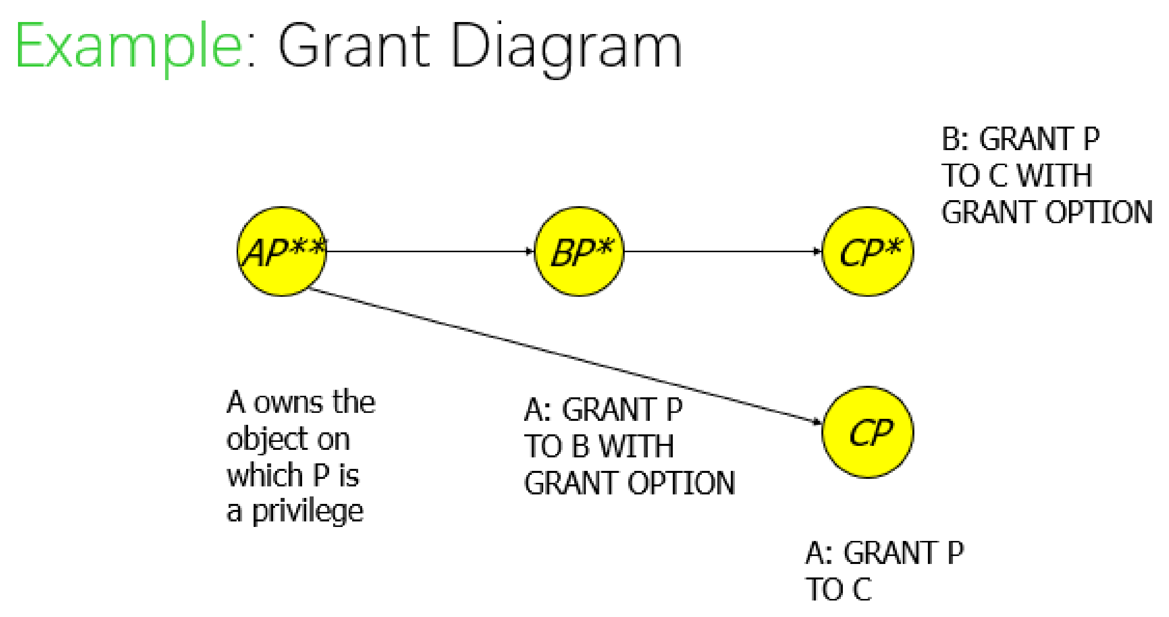
Example: Views as Access Control

- We might not want to give the SELECT privilege on Emps(name, addr, salary).
- But it is safer to give SELECT on:

```
CREATE VIEW SafeEmps AS
SELECT name, addr FROM Emps;
```

- Queries on SafeEmps do not require SELECT on Emps, just on SafeEmps.

```
WITH GRANT OPTION：使得被授予权限的对象也拥有授予该权限的权力

GRANT <list of privilege>
ON <relation>
TO <list of authorization ID'S> [WITH GRANT OPTION]

例子：
GRANT SELECT, UPDATE(price)
ON Sells TO Sally
则Sally拥有查询和更新price属性的权力

REVOKE <list of privileges>
ON <relation or other object>
FROM <list of authorization ID'S>
权限回收（但是如果author从其他地方获得了相同的权限则其依然可以使用）

两种收回模式：
CASCADE：将权力回收对象发放的权力一并全部收回
RESTRICT：仅回收该对象的权力，如果该对象放发了权力，则回收失败
```



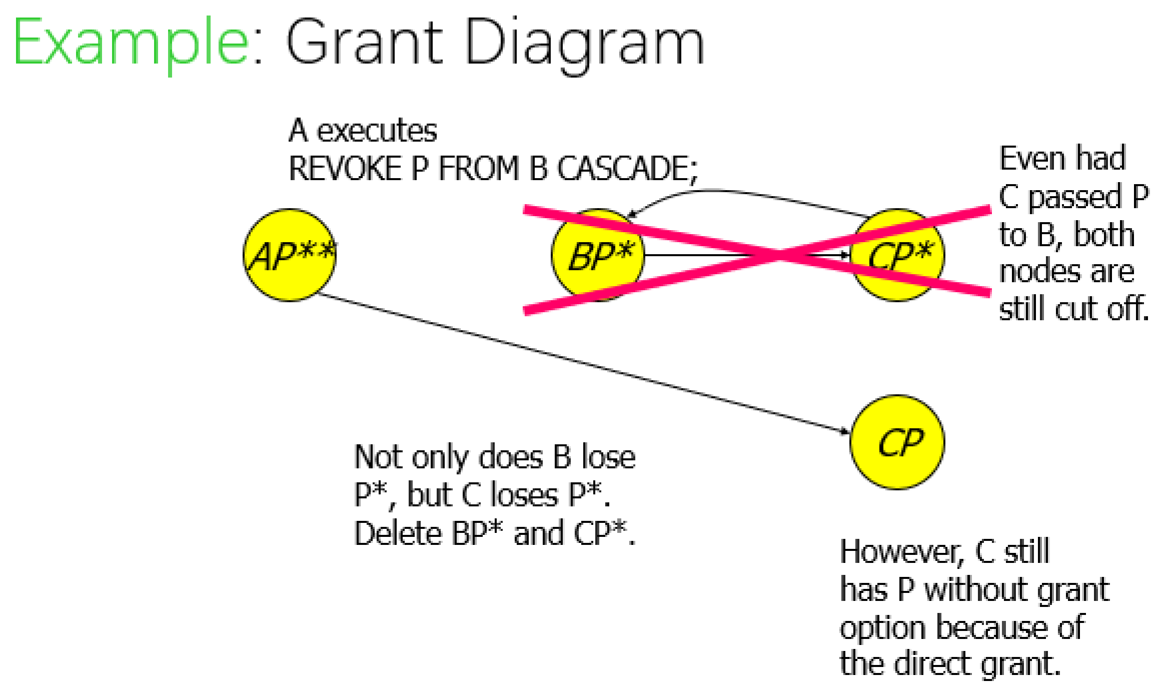
Example: Grant Diagram

B: GRANT P TO C WITH GRANT OPTION

A owns the object on which P is a privilege

A: GRANT P TO B WITH GRANT OPTION

A: GRANT P TO C



Example: Grant Diagram

A executes
REVOKE P FROM B CASCADE;

Even had
C passed P
to B, both
nodes are
still cut off.

Not only does B lose
P*, but C loses P*.
Delete BP* and CP*.

However, C still
has P without grant
option because of
the direct grant.

ER图

可以将ER图转换为DB设计

多对多关系：任一集合的一个实体可以连接到另外一个集合的多个实体

多对一关系：比如每个人最爱的啤酒最多只有一个

一对一关系：任一实体集的每个实体最多与另一个集的一个实体相关

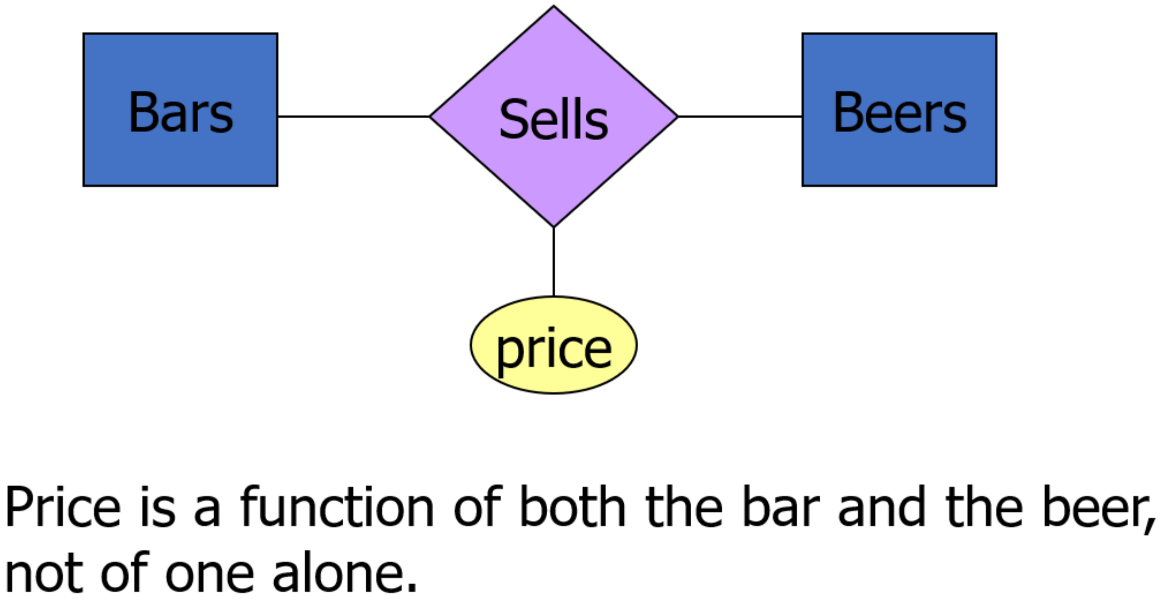
比如：每家厂商只能有一个销量最高的啤酒，一种啤酒只能由一个制造商生产

尖角箭头：指向1，表示最多仅有1个（0， 1）

圆角箭头：指向1，表示有且仅有1个

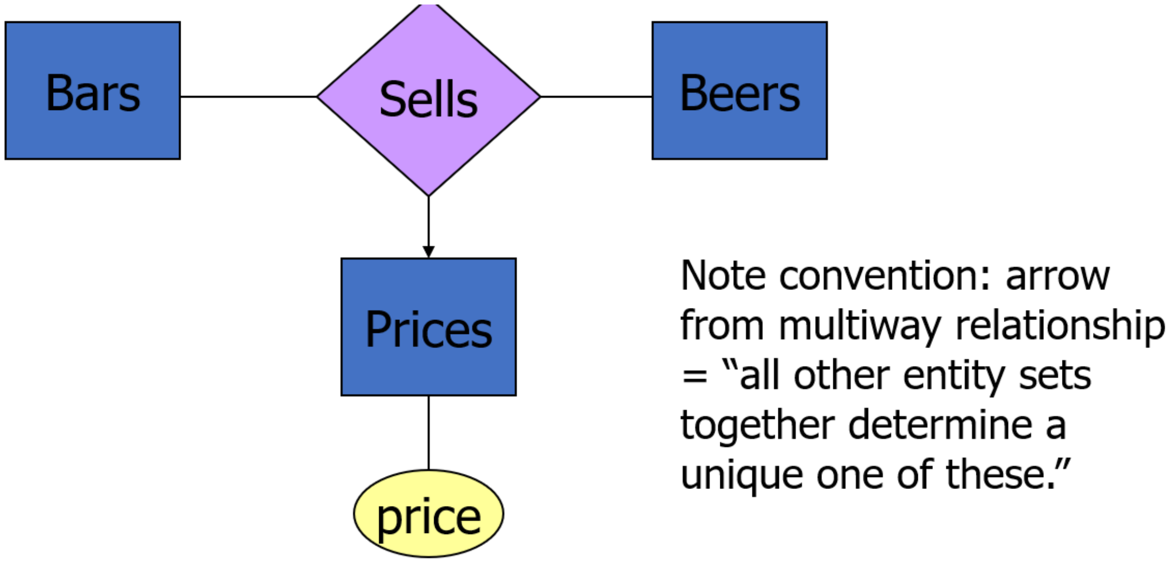
例子：一些啤酒可能不是任何厂商的最佳销售啤酒，但是每个厂商都有一个最佳销售啤酒，则使用尖角箭头更为合适

有时候也可以为关系添加属性



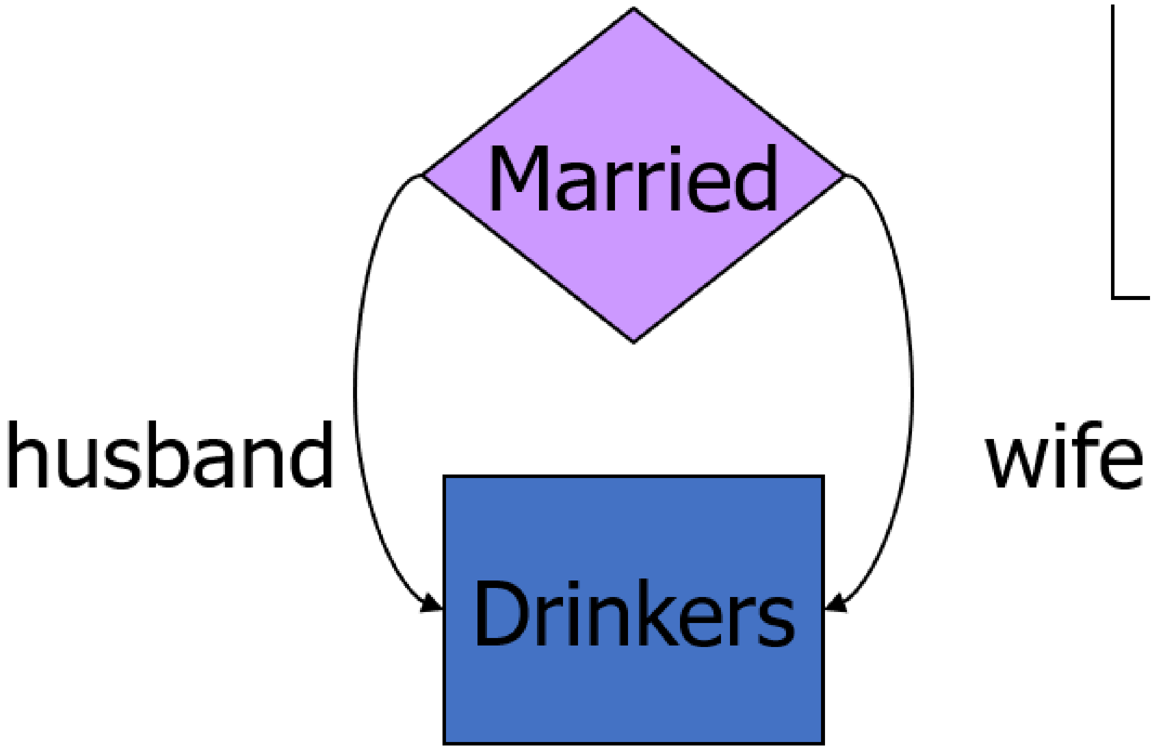
Price is a function of both the bar and the beer, not of one alone.

也可以单独设置一个实体表示价格



Note convention: arrow from multiway relationship = “all other entity sets together determine a unique one of these.”

当一个实体集在关系中出现多次则使用ROLES标记关系和实体集的边



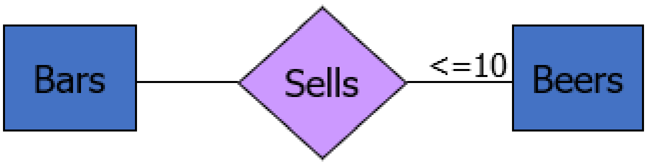
三角形的关系表示子集

ER图中的子集和OO中的不同

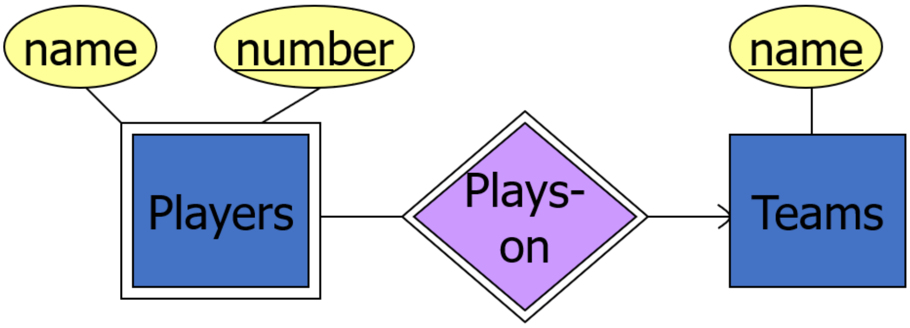
（ER图中若实体E表示在子类中，则E也表示在超类中）

在ER中为属性添加下划线表示主键

在ER图中也可以添加数量关系的限制



弱实体集：自身的属性无法唯一标识



- Double diamond for *supporting* many-one relationship.
- Double rectangle for the weak entity set.

弱实体集与其他支持实体集具有一个或多个“1”的关系

但支持关系必须有一个圆形箭头

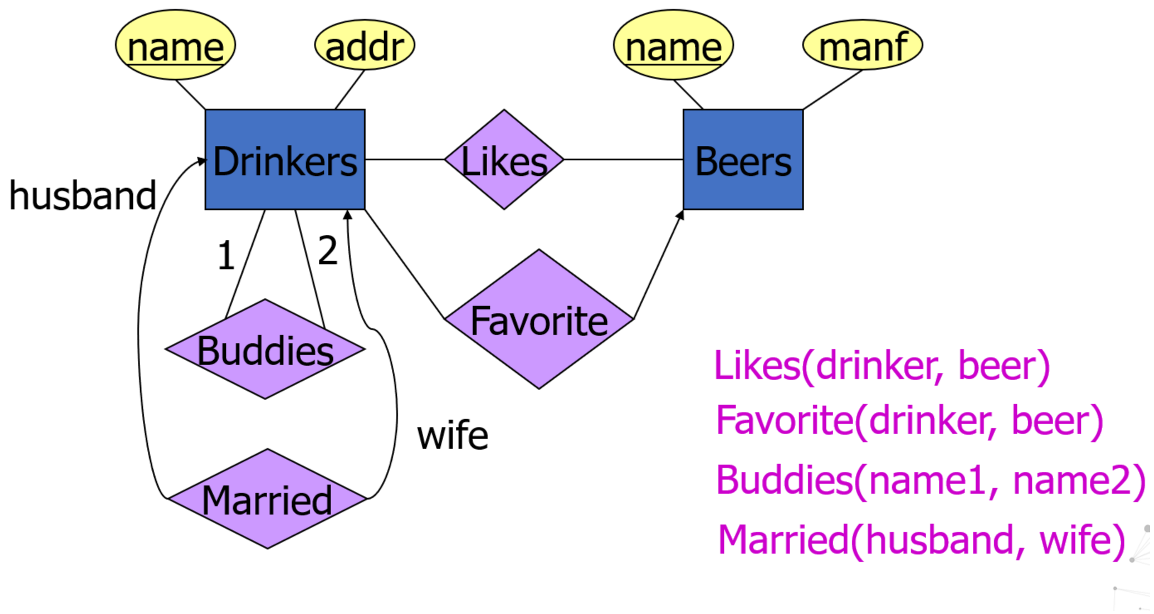
弱实体集的主键是自身的键和支撑实体集的主键一同构成

设计准则

- 避免冗余——讲述相同的事物用不同的方式，造成空间浪费以及不一致性的问题
- 限制使用弱实体集
- 不要使用仅含一个属性的实体集

When Do We Need Weak Entity Sets?

- The usual reason is that there is no global authority capable of creating unique ID’ s.
- Example: it is unlikely that there could be an agreement to assign unique player numbers across all football teams in the world.



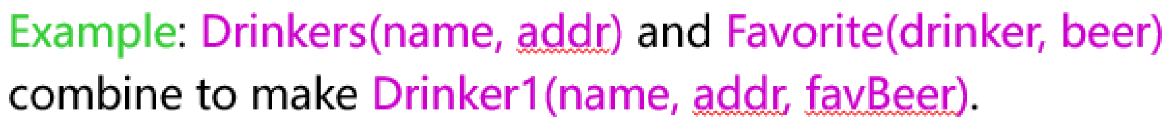
Likes(drinker, beer)

Favorite(drinker, beer)

Buddies(name1, name2)

Married(husband, wife)

关系合并：一个实体集内的关系， 或多对一关系中处于多的实体



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

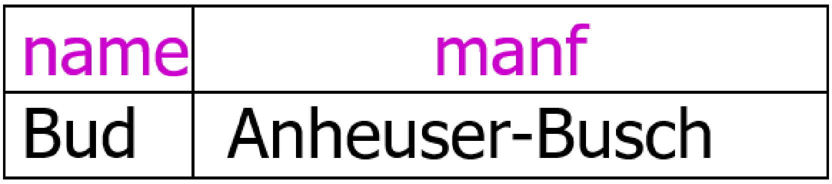
合并多对多的关系可能会导致冗余

弱实体集必须包含其所有的支撑属性——包括其他实体集中的

弱实体集的支持关系是冗余的，不会产生任何影响

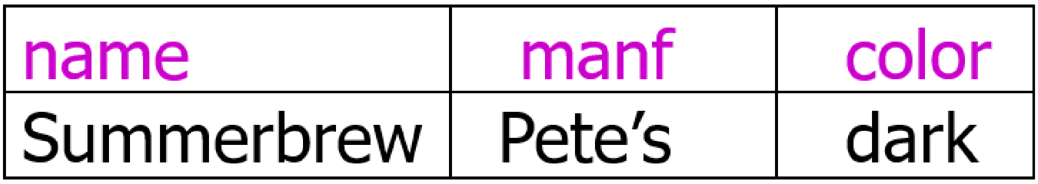
子类实现的三种方式

- 面对对象：子类继承父类的所有属性



name	manf
Bud	Anheuser-Busch

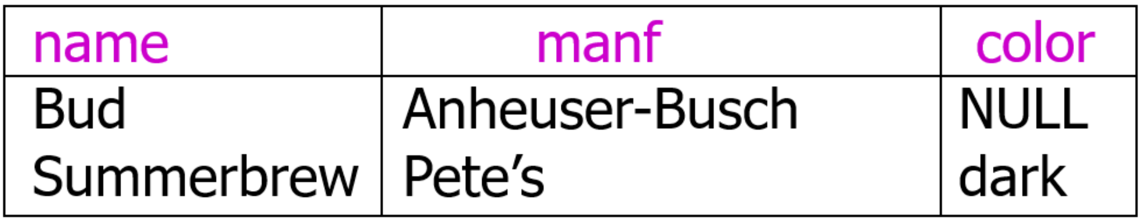
Beers



name	manf	color
Summerbrew	Pete’s	dark

Ales

- 使用NULL：不新创建关系，多出的属性不存在则赋值为NULL

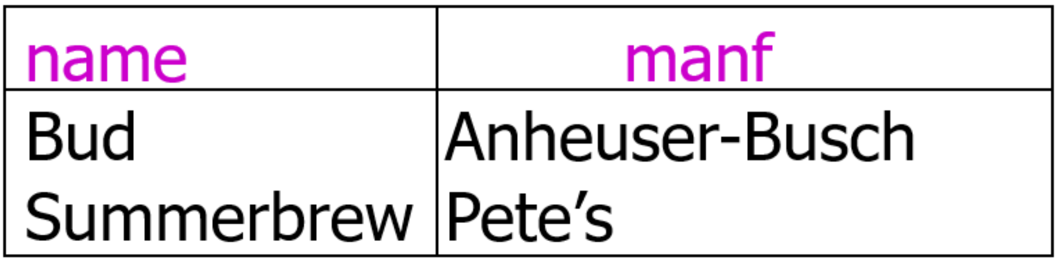


name	manf	color
Bud	Anheuser-Busch	NULL
Summerbrew	Pete’s	dark

Beers

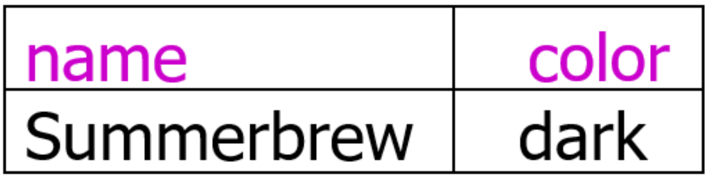
Saves space unless there are *lots* of attributes that are usually NULL.

- ER风格：仅包含子类属性



name	manf
Bud	Anheuser-Busch
Summerbrew	Pete’s

Beers



name	color
Summerbrew	dark

Ales

Good for queries like “find all beers (including ales) made by Pete’s.”

约束

```
列间约束
    CREATE TABLE Sells (
            bar     CHAR(20) UNIQUE,
            beer        VARCHAR(20) REFERENCES manf(beer),
            price   REAL,
        );
```

```
表级约束
    CREATE TABLE Sells (
            bar     CHAR(20),
            beer        VARCHAR(20),
            price   REAL,
            PRIMARY KEY (bar, beer),
            FOREIGN KEY(beer) REFERENCES manf(beer)
    );
```

外键必须是声明的主键或UNQIUE

外键约束：从R到S的关系存在约束

两类冲突：

- 在R中插入或更新在S中查不到的值
- 删除或更新S会导致R中的某些元组“悬空”

外键的附加选项：

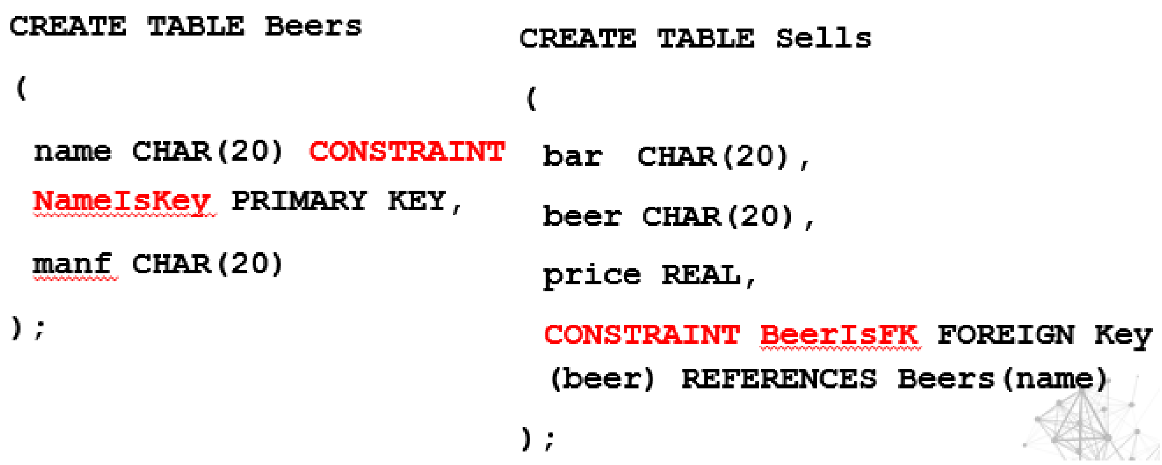
```
default：拒绝修改
cascade：在原表中做同样的修改
set NULL：将修改的置为NULL
```

```
对于操作设置策略
CREATE TABLE Sells (
    bar     VARCHAR(20),
    beer    VARCHAR(20),
    price   REAL,
    FOREIGN KEY(beer) REFERENCES Beers(name)
        ON DELETE SET NULL
        ON UPDATE CASCADE
);

```

CHECK约束

```
CREATE TABLE Sells (
    bar     VARCHAR(20),
    beer    VARCHAR(20),
    price   REAL CHECK ( price <= 35.00 )
);
```

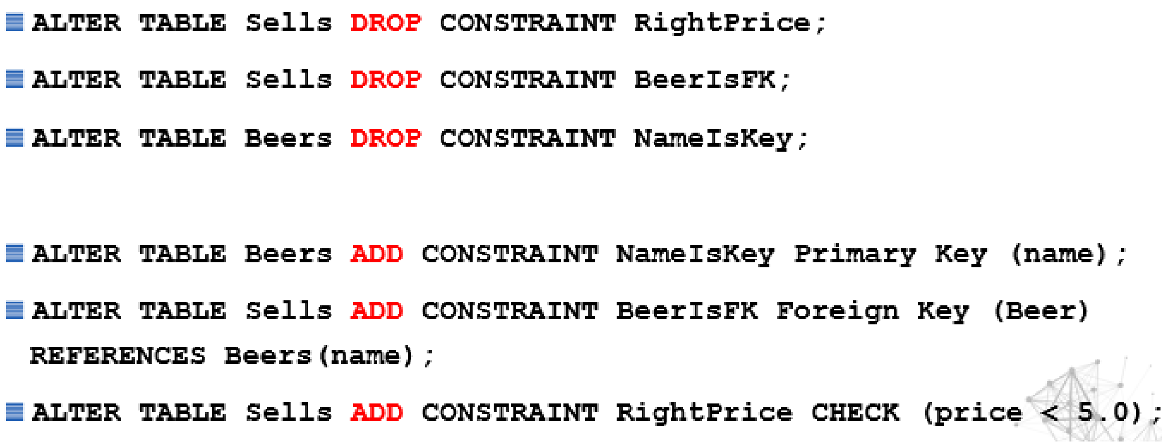
给约束起名



```
CREATE TABLE Beers
(
  name CHAR(20) CONSTRAINT
  NameIsKey PRIMARY KEY,
  manf CHAR(20)
);
```

```
CREATE TABLE Sells
(
  bar  CHAR(20),
  beer CHAR(20),
  price REAL,
  CONSTRAINT BeerIsFK FOREIGN Key
  (beer) REFERENCES Beers(name)
);
```

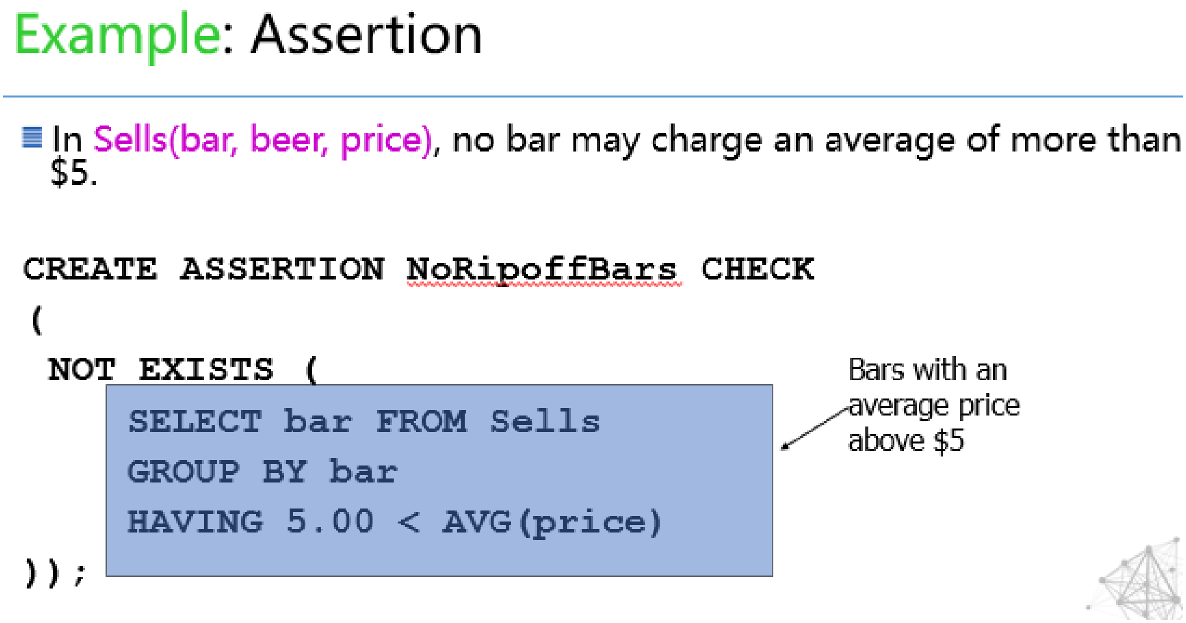
起名后就可以通过ALTER TALBE调整约束



```
ALTER TABLE Sells DROP CONSTRAINT RightPrice;
ALTER TABLE Sells DROP CONSTRAINT BeerIsFK;
ALTER TABLE Beers DROP CONSTRAINT NameIsKey;

ALTER TABLE Beers ADD CONSTRAINT NameIsKey Primary Key (name);
ALTER TABLE Sells ADD CONSTRAINT BeerIsFK Foreign Key (Beer)
REFERENCES Beers(name);
ALTER TABLE Sells ADD CONSTRAINT RightPrice CHECK (price < 5.0);
```

Assertion断言

- 在数据库中任何关系上的修改都需要通过已设置的所有断言验证



Example: Assertion

- In Sells(bar, beer, price), no bar may charge an average of more than $5.

```
CREATE ASSERTION NoRipoffBars CHECK
(
 NOT EXISTS (
    SELECT bar FROM Sells
    GROUP BY bar
    HAVING 5.00 < AVG(price)
));
```

triggers触发器

触发器让用户决定何时进行检查

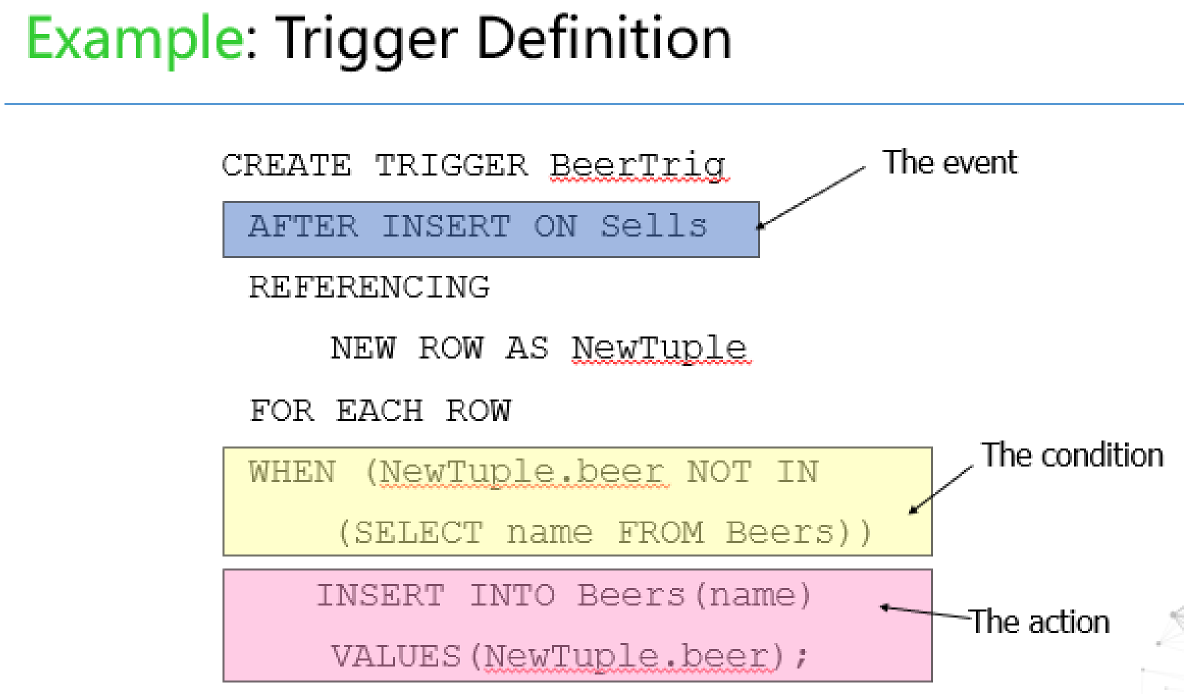
EVENT-CONDICTION-ACTION

- event：通常时数据库的增删改查
- condition：任何bool表达式SQL

- action：任何SQL语句

Instead of using a foreign-key constraint and rejecting insertions into Sells(bar, beer, price) with unknown beers, a trigger can add that beer to Beers, with a NULL manufacturer.

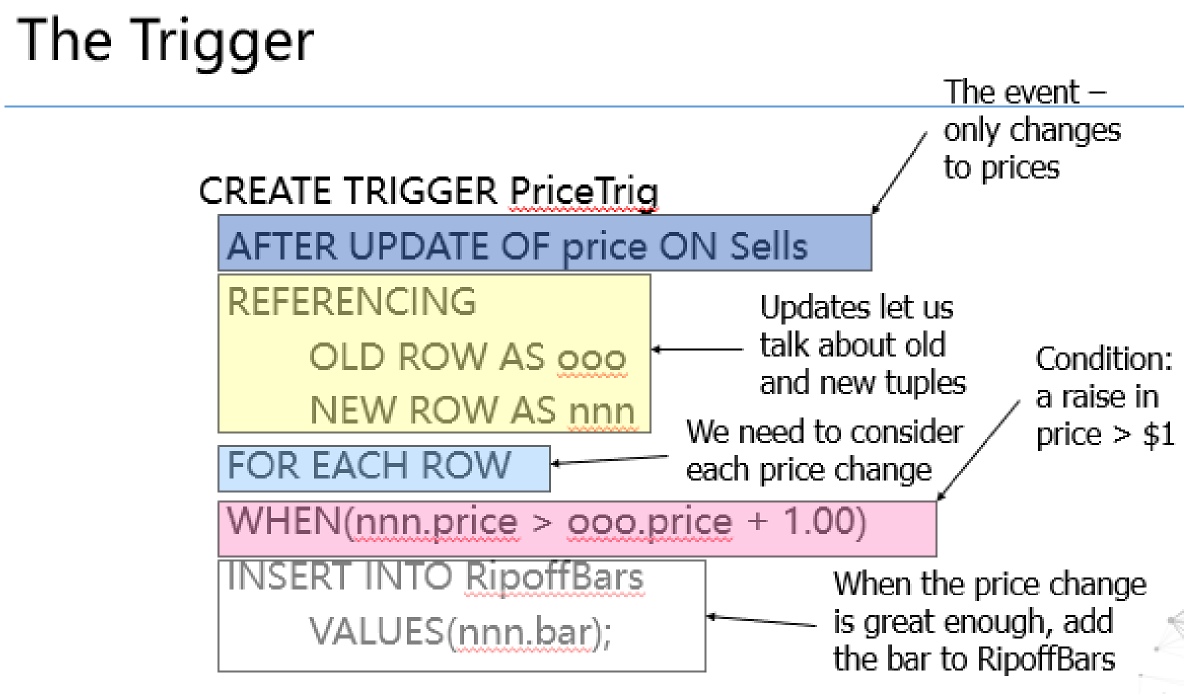
Example: Trigger Definition



```
CREATE TRIGGER BeerTrig
  AFTER INSERT ON Sells
  REFERENCING
        NEW ROW AS NewTuple
  FOR EACH ROW
  WHEN (NewTuple.beer NOT IN
        (SELECT name FROM Beers))
       INSERT INTO Beers(name)
        VALUES(NewTuple.beer);
```

```
OPTION
CREATE TRIGGER <name>
CREATE OR REPLACE TRIGGER <name>--若该触发器存在则可以修改该触发器

AFTER/BEFORE
INSERT/DELATE/UPDATE
```

Using Sells(bar, beer, price) and a unary relation RipoffBars(bar), maintain a list of bars that raise the price of any beer by more than $1.

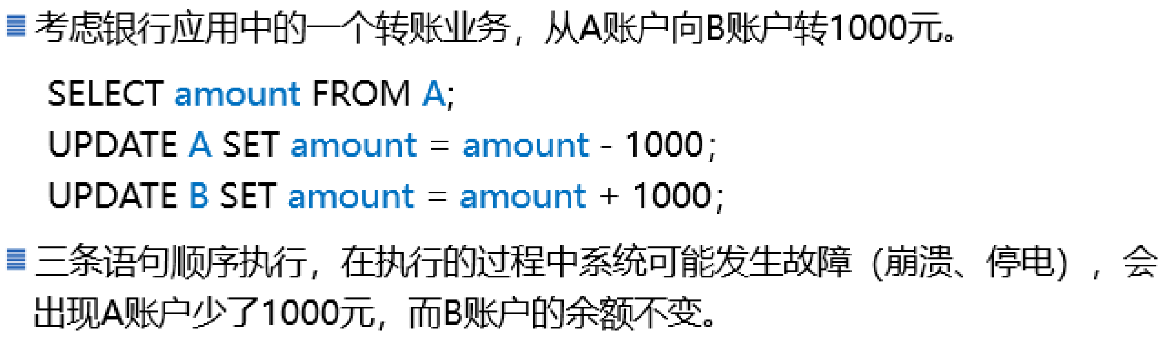


The Trigger

CREATE TRIGGER PriceTrig
AFTER UPDATE OF price ON Sells
REFERENCING
OLD ROW AS ooo
NEW ROW AS nnn
FOR EACH ROW
WHEN(nnn.price > ooo.price + 1.00)
INSERT INTO RipoffBars
VALUES(nnn.bar);

The event – only changes to prices
Updates let us talk about old and new tuples
Condition: a raise in price > $1
We need to consider each price change
When the price change is great enough, add the bar to RipoffBars

事务transaction



- 考虑银行应用中的一个转账业务，从A账户向B账户转1000元。

```
SELECT amount FROM A;
UPDATE A SET amount = amount - 1000;
UPDATE B SET amount = amount + 1000;
```

- 三条语句顺序执行，在执行的过程中系统可能发生故障（崩溃、停电），会出现A账户少了1000元，而B账户的余额不变。

数据库一个重要特征就是可共享

通过**事务**可以保证不会出现一张票卖个两个人，或者上述金额丢失的情况发生

事物是用户定义的一个数据库操作序列，是一个不可分割的工作单位

```
BEGIN TRANSACTION
……
COMMIT;
```

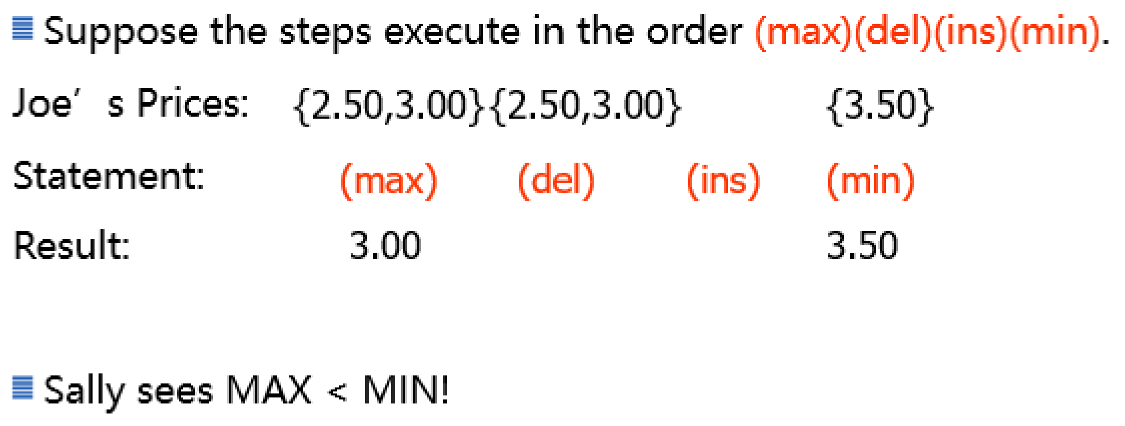
事物的四大性质：ACID

- 原子性：事务不可拆分性，要么都被执行，要么都不执行
- 一致性：数据符合定义和要求，，没用包含非法或无效的错误数据（由用户定义）
- 隔离性：事物并行，互不干扰，并发状态下执行的事务和串行执行的事务产生的结果一样
- 持久性：执行事务的结果会保存，即使服务器在执行完毕后停机，数据没来得及存储到硬盘上，后续数据库也可以通过日志的方法方法进行恢复

数据库的数据恢复技术保证事务的原子性、一致性和持久性（通过日志等）

数据库的并发控制技术可以保证事务的隔离性（通过加锁）

```
例子：
Sally查询Sells中Joes‘s bar中价格最高和最低的啤酒
与此同时Joe删除了当前的bar中销售的啤酒，重新上架了新的啤酒
```



- Suppose the steps execute in the order (max)(del)(ins)(min).

Joe’ s Prices:	{2.50,3.00}	{2.50,3.00}		{3.50}
Statement:	(max)	(del)	(ins)	(min)
Result:	3.00			3.50

- Sally sees MAX < MIN!

以上的执行顺序会导致Sally查出来的最大价格小于最低价格！

需要事务进行管理

```
COMMIT表示提交
ROLLBACK表示回滚——撤销上述的操作
```

不适用ROLLBACK进行事务的撤销，则可能导致Sally看到3.5这个本不该存在的数据

数据库中不存在的数据——脏数据（并发导致）

事务的隔离级别：

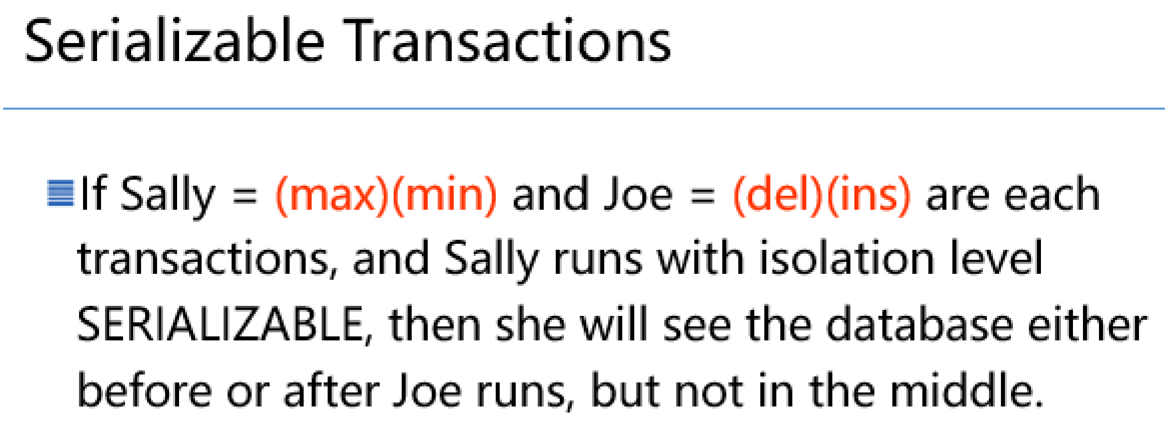
一方面希望保证事务之间的隔离，另一方面又希望提高并发访问

DBMS定义了事务的隔离级别

- SERIALIZABLE（可串行化）：允许事务并发执行，并保证并发调度可串行化
- REPEATABLE READ（可重复读）：只允许事务读已提交数据，且两次读之间不许其他事务修改此数据（如果第二次读相同的数据，即使该数据被其他并行事务提交修改过，也可以得到与第一次修改前相同的查询结果）
- READ COMMITTED（读提交数据）：允许事务读已提交数据，但不要求可重复读
- READ UNCOMMITTED（可以读未提交数据）：允许事务读已提交或未提交的数据

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

只有SERIALIZABLE符合ACID原则



Serializable Transactions

- If Sally = (max)(min) and Joe = (del)(ins) are each transactions, and Sally runs with isolation level SERIALIZABLE, then she will see the database either before or after Joe runs, but not in the middle.

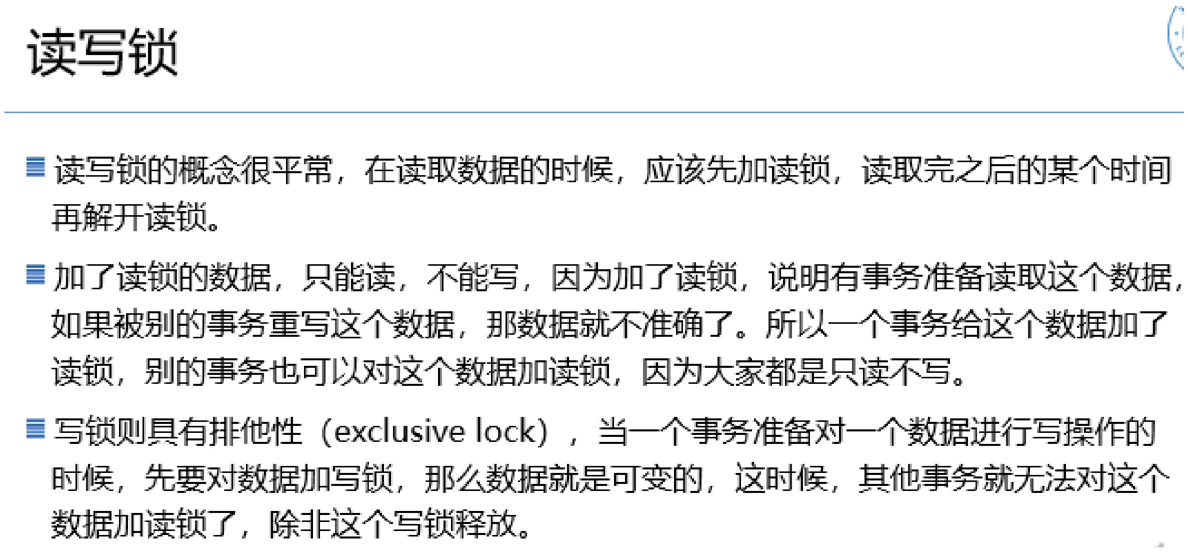
隔离等级仅影响自身的操作，不影响他人对于数据库的观察

```
例：如果Joe以SERIALIZABLE操作，而Sally不，则Sally可能无法看到Joe's bar的价格
解释：Joe事务删除Bar后停止一段时间，期间Sally进行查询，则无法看到price

例：Sally以READ COMMITTED隔离级别，则她只能看到提交后的数据，可能两次查看的数据不会相同
解释：(max)(del)(ins)(min) Joes commit则，Sally会看到MAX<MIN
Sally先查询最大，然后Joe执行操作，由于Sally可以读COMMIT的数据，所以Sally读最小时时修改后的数据，则导致以上结果

例：Sally以 REPEATABLE READ隔离级别，(max)(del)(ins)(min)的运行顺序
max可以看到2.5和3.0
min也看到2.5和3.0（即使中间Joe修改了price并COMMIT了，但是要和第一次读时保持一致）

例：Sally以UNCOMMITTED,可以看到未提交的数据，则Sally可以看到Joe修改的3.5，即使Joe还没有提交
解释：Sally运行max看到2.5 3.0，sleep；此时Joe运行完删除和修改，sleep；然后Sally运行min则可以看到3.5，然后Joe COMMIT
```

OpenGauss隔离级别

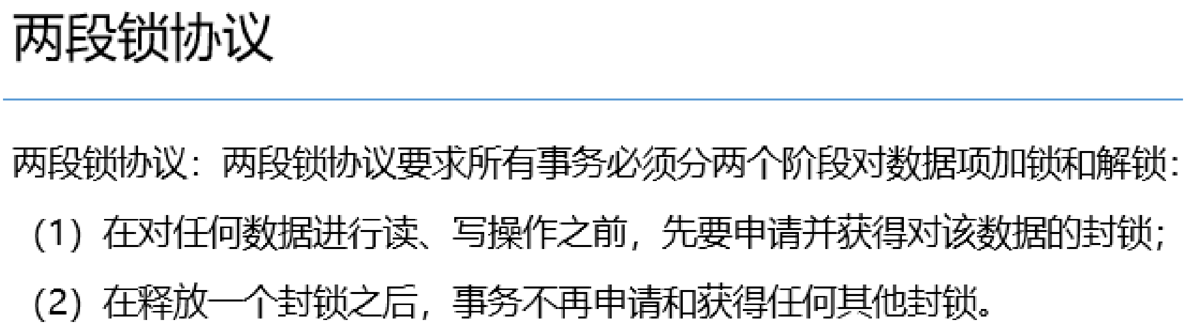
- READ COMMITED：读已提交（默认）
- REPEATABLE READ：可重复读
- SERIALIZABLE：暂不支持

通常来说，事务隔离级别越低，所需持有锁的时间越短，并发性能越好



读写锁

- 读写锁的概念很平常，在读取数据的时候，应该先加读锁，读取完之后的某个时间再解开读锁。
- 加了读锁的数据，只能读，不能写，因为加了读锁，说明有事务准备读取这个数据，如果被别的事务重写这个数据，那数据就不准确了。所以一个事务给这个数据加了读锁，别的事务也可以对这个数据加读锁，因为大家都是只读不写。
- 写锁则具有排他性（exclusive lock），当一个事务准备对一个数据进行写操作的时候，先要对数据加写锁，那么数据就是可变的，这时候，其他事务就无法对这个数据加读锁了，除非这个写锁释放。



两段锁协议

两段锁协议：两段锁协议要求所有事务必须分两个阶段对数据项加锁和解锁：

（1）在对任何数据进行读、写操作之前，先要申请并获得对该数据的封锁；

（2）在释放一个封锁之后，事务不再申请和获得任何其他封锁。

关系数据库设计理论

键：必须最小，属性函数决定关系的所有其他属性，闭包可以包含关系的所有属性

超键：包含键的属性集，不需要最小

平凡的函数依赖：右边是左边的子集

非平凡的函数依赖：右边不是左边的子集

属性的闭包：左边属性在集合中，就把FD右边的属性也进行加入

基本集：任何和S等价的FD集合都是S的基本集

最小化基本集B：

- B所有FD的右边均为单一属性
- 从B中删除任何一个FD后，该集合不再是基本集
- 对于B中任何一个FD，左边如果删除一个或多个属性，则不再是基本集

函数依赖集的投影算法：

输入：关系R通过投影计算得到关系R1，以及在R中成立的FD集合S

输出：在R1中成立的FD集合

- 对于R1中的每个子集X进行闭包运算
- 对于所有在闭包中且属于R1的属性A，讲所有非平凡的FD X->A添加到R1的FD集合中
- 简化：空集和全集不需要计算；若X闭包已经包含了全部属性，则不需要通过其超集来寻找新的FD

关系数据库中可能存在冗余和异常的情况

BCNF：每个非平凡FD的左边都必须是超键

BCNF分解算法——得到的关系都满足BCNF

输入：关系R和其上的函数依赖S

输出：分解得到的关系集合，每个关系均属于BCNF

- 检查R是否属于BCNF
- 存在违例，假设为X->Y，计算左侧X闭包，选择该闭包作为一个新的关系，并讲另一个关系模式包含属性X以及那些不在前一个关系中的属性
- **使用投影算法**，得到新的关系集上的FD
- 重复上述步骤，得到所有的分解关系
- 注意：所有的二元关系都满足BCNF

BCNF的分解不唯一

无损连接的chase检验

- 根据分解的关系绘制tableau 图

- 根据FD修改tableau图
- 若出现一行所有的分量均不带下标，则说明该分解为无损分解

第三范式3NF：左侧为超键或右侧由主属性构成

主属性：某个键的成员

3NF模式综合算法

将关系R分解为满足如下条件的关系：

- 分解得到的关系都属于3NF
- 分解包含无损连接
- 分解具有依赖保持性质

算法：

- 找出关系R和其函数依赖集F的一个最小基本集G
- 将最小基本集中的每个FD都作为某个关系的模式
- 如果分解得到的所有模式不包含R的超键，则增加一个关系，其模式为R的任何一个键

如何验证分解的关系是否违反BCNF：

- 给定的FD投影到分解关系上进行验证
- 左侧必须包含超键