

《交通大数据技术》



2024/6/14

交通大数据技术

马晓磊
交通科学与工程学院
2024年

结构化查询语言 (SQL) II



本节大纲

- **SQL中的分组和聚合**
- **子查询**
- **并集、交集和差异**
- **SQL中的约束**

SQL支持五个聚合运算符

- SUM
- MIN
- MAX
- AVG
- COUNT

这些聚合应用于单个属性或值，除了COUNT:

- COUNT (*) 可用于对所有元组进行计数。

SQL中的聚合

Product

PName	Price	Category	Manufacturer
Gizmo	19.99	Gadgets	GizmoWorks
Powergizmo	29.99	Gadgets	GizmoWorks
SingleTouch	149.99	Photography	Canon
MultiTouch	203.99	Household	Hitachi

```
SELECT AVG(price)
FROM product
WHERE manufacturer = 'GizmoWorks'
```



(No column name)

24.99

```
SELECT COUNT(*) AS ProductCount
FROM product
WHERE manufacturer = 'GizmoWorks'
```



ProductCount

2

SQL中的聚合

Product

PName	Price	Category	Manufacturer
Gizmo	19.99	Gadgets	GizmoWorks
Powergizmo	29.99	Gadgets	GizmoWorks
SingleTouch	149.99	Photography	Canon
MultiTouch	203.99	Household	Hitachi

COUNT适用于副本，除非另有说明：

```
SELECT COUNT(category) AS CategoryCt  
FROM product
```



CategoryCt

4

```
SELECT COUNT(DISTINCT category) AS CategoryCt  
FROM product
```



CategoryCt

3

SQL中的聚合

Sale

Product	Date	Price	Quantity
Banana	2016-10-19	0.52	17
Bagel	2016-10-20	0.85	20
Bagel	2016-10-21	0.85	15
Banana	2016-10-22	0.52	7

- 示例1：查找总销售额

```
SELECT SUM(price * quantity) AS TotalSale  
FROM sale
```



TotalSale

42.23

- 示例2：查找bagel的总销售额

```
SELECT SUM(price * quantity) AS BagelSale  
FROM sale  
WHERE product = 'bagel'
```



BagelSale

29.75

SQL中的分组与聚合

通常，我们需要对关系的某些部分进行聚合。

Sale(product, date, price, quantity)

- 示例3：找到每个产品的总销售额。

```
SELECT Product, SUM(price * quantity) AS TotalSale
FROM sale
GROUP BY Product
```



Product	TotalSale
Bagel	29.75
Banana	12.48

SQL中的分组与聚合

SQL中分组和聚合的过程：

1. 计算FROM和WHERE子句。
2. 为GROUP BY属性的每个组合分表。
3. 应用聚合并为每个子表返回一个元组。

使用聚合时，SELECT只能有两种类型的表达式：

- GROUP BY子句中的属性
- 聚合

SQL中的分组与聚合

GROUP BY如何让事情变得简单?

- 使用GROUP BY

```
SELECT Product, SUM(price * quantity) AS TotalSale
FROM sale
GROUP BY Product
```

- 与不使用GROUP BY (嵌套查询) 相比较

```
SELECT DISTINCT x.Product,
    (SELECT SUM(price * quantity) FROM sale AS y
     WHERE x.product = y.product) AS TotalSale
FROM sale AS x
```

SQL中的分组与聚合

多个聚合

- 每个产品的总销售额和最大销售量是多少？

```
SELECT Product, SUM(price * quantity) AS TotalSale,  
              MAX(quantity) AS MaxQuantity  
FROM sale  
GROUP BY Product
```



Product	TotalSale	MaxQuantity
Bagel	29.75	20
Banana	12.48	17

HAVING子句

HAVING <条件>可以跟在GROUP BY子句后面。
条件适用于每个组，不满足<条件>的组将被排除。

HAVING子句中的条件可以引用属性，只要该属性在组中有意义，
即它是：

- GROUP BY子句中的属性
- 聚合

HAVING子句

查找2016年10月1日后售出且总销售量超过30个的每个产品的产品名称和总销售额。

```
SELECT Product, SUM(price * quantity) AS TotalSale
FROM sale
WHERE date > '2016-10-1'
GROUP BY Product
HAVING SUM(quantity) > 30
```

销售

Product	Date	Price	Quantity
Banana	2016-10-19	0.52	17
Bagel	2016-10-20	0.85	20
Bagel	2016-10-21	0.85	15
Banana	2016-10-22	0.52	7



Product	TotalSale
Bagel	29.75

SQL中的分组与聚合

分组和聚合的一般形式：

```
SELECT S  
  FROM R1,...,Rn  
 WHERE C1  
 GROUP BY a1,...,ak  
HAVING C2
```

S：可能包含属性 a_1, \dots, a_k 和/或相应的聚合，但不包含其他属性

C1：R1, ..., Rn中属性的任何条件

C2：聚合表达式上的任何条件

聚合示例

Author (AuthorID, Name)

Write (PaperName, AuthorID)

- 查找写过至少10篇论文的所有作者

```
SELECT a.name
  FROM author AS a, write AS w
 WHERE a.authorid = w.authorid
 GROUP BY a.name
HAVING COUNT(w.papername) > 10
```

带括号的 SELECT-FROM-WHERE 语句（子查询）可用于许多位置，包括FROM和WHERE子句

- 在FROM子句中，我们可以放置另一个查询，然后查询其结果
- 您可以使用保证返回单个值的查询来代替值

按商品种类统计每种商品的平均销售价格，要求仅取出平均价格超出1500的商品组，返回商品种类和平均销售单价两列，平均单价列命名为Avg_price。

```
SELECT product_type, AVG(sale_price) AS Avg_price
FROM Product
GROUP BY product_type
HAVING [填空1] > 1500
```

- ☐ A Avg_price
- ☒ B AVG(sale_price)
- ☐ C SUM(sale_price)
- ☐ D Sale_price

Product (商品) 表

product_id (商品编号)	product_name (商品名称)	product_type (商品种类)	sale_price (销售单价)
0001	T恤衫	衣服	500
0002	打孔器	办公用品	320
0003	运动T恤	衣服	2800
0004	菜刀	厨房用具	2800
0005	高压锅	厨房用具	6800

提交

Product

PName	Price	Category	Manufacturer
Gizmo	19.99	Gadgets	GizmoWorks
Powergizmo	29.99	Gadgets	GizmoWorks
SingleTouch	149.99	Photography	Canon
MultiTouch	203.99	Household	Hitachi

Company

CName	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

- 找出所有生产 “gadgets” 产品的公司的名称和股票价格

```
SELECT DISTINCT c.cname, stockprice
  FROM (SELECT cname, stockprice
        FROM company, product
        WHERE cname = manufacturer
          AND product.category = 'gadgets'
       ) AS c
```

对用作关系表的子查询命名

返回关系表的子查询

Product (pname, price, category, manufacturer)

Purchase (buyer, seller, store, product)

Company (cname, stockPrice, country)

- 查找制造Joe购买的某些产品的公司的股票价格

```
SELECT stockprice
  FROM company, product
 WHERE cname = manufacturer
        AND pname IN (SELECT product
                        FROM purchase
                        WHERE buyer = 'Joe')
```

返回关系表的子查询

类似的工作也可以不用子查询来完成：

- 找到制造Joe购买的一些产品的公司的股票价格

```
SELECT stockPrice
  FROM company, product, purchase
 WHERE cname = manufacturer
        AND pname = purchase.product
        AND buyer = 'Joe'
```

返回关系表的子查询

以下两个查询将返回完全相同的结果：

```
SELECT DISTINCT stockprice
  FROM company, product
 WHERE cname = manufacturer
       AND pname IN (SELECT product
                     FROM purchase
                     WHERE buyer = 'Joe')
```

```
SELECT DISTINCT stockPrice
  FROM company, product, purchase
 WHERE cname = manufacturer
       AND  pname = purchase.product
       AND  buyer = 'Joe'
```

返回关系表的子查询

ALL and ANY with comparison运算符

- $S > \text{ALL} \langle \text{set} \rangle$: 如果S大于集合中的所有值, 则返回TRUE
- $S > \text{ANY} \langle \text{set} \rangle$: 如果S大于集合中的任何单个值, 则返回TRUE
- 示例: 查找比“GizmoWorks”公司生产的所有产品都贵的产品名称。

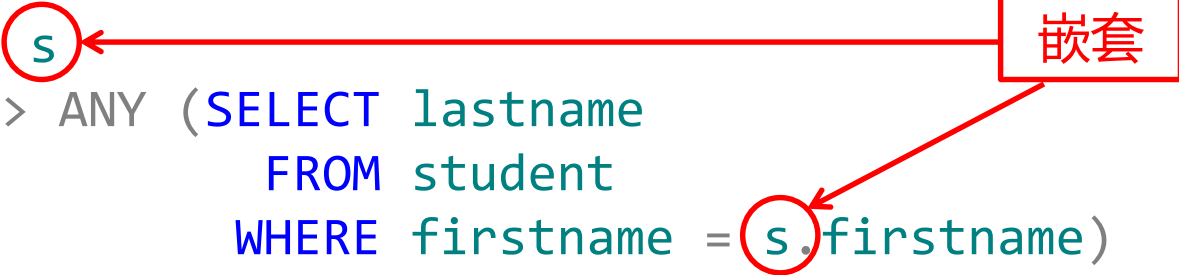
```
SELECT pname
FROM product
WHERE price > ALL (SELECT price
                    FROM product
                    WHERE manufacturer = 'GizmoWorks')
```

关联查询

Student (FirstName, LastName, Gender, Age)

- 查找多个学生使用的名字。

```
SELECT DISTINCT firstname
  FROM student AS s
 WHERE lastname <> ANY (SELECT lastname
                        FROM student
                        WHERE firstname = s.firstname)
```



可以在单个查询语句中完成吗？

```
SELECT DISTINCT s1.firstname
  FROM student AS s1, student AS s2
 WHERE s1.firstname = s2.firstname
       AND s1.lastname <> s2.lastname
```

关联查询

关联子查询（也称为同步子查询）是使用外部查询值的子查询（嵌套在另一个查询中的查询）。对于外部查询处理的每一行，该子查询可能会被计算一次，这可能是低效的。

下面是一个典型的关联子查询的示例。在本例中，目标是查找所在部门工资高于平均水平的所有员工。

```
SELECT employee_number, name
FROM employees emp
WHERE salary > (
    SELECT AVG(salary)
    FROM employees
    WHERE department = emp.department);
```


关联查询

在上面的查询中，外部查询是。

```
SELECT employee_number, name  
FROM employees emp  
WHERE salary > ...
```

内部查询（关联子查询）是

```
SELECT AVG(salary)  
FROM employees  
WHERE department = emp.department
```

在上面的嵌套查询中，必须为每个员工重新执行内部查询。

关联查询

相关子查询可能出现在WHERE 子句之外的其他地方；例如，此查询在SELECT子句中使用一个相关子查询来打印员工的整个列表以及每个员工所在部门的平均工资。

```
SELECT
  employee_number,
  name,
  (SELECT AVG(salary)
   FROM employees
   WHERE department = emp.department) AS department_average
FROM employees emp;
```

因为子查询与外部查询的一列相关，因此必须为结果的每一行重新执行它。

返回一个元组的子查询

如果保证子查询生成一个元组，则可以将该子查询用作值。

- 通常情况下，元组只有一个属性。
- 通常情况下，单个元组通常由属性的键性来保证。
- 如果没有元组或有多个元组时，则会发生运行时错误。

返回一个元组的子查询

Players(Name, Salary, Height, Weight, Team)

- 问题：找到薪水最高的球员的名字。

这有点棘手，让我们一步一步来做：

- 首先，在我的表格中找到最高的薪水。

```
SELECT MAX(salary)
FROM player
```



(No column name)

15000000.00

返回一个元组的子查询

Players(Name, Salary, Height, Weight, Team)

- 问题：找到薪水最高的球员的名字。

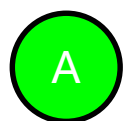
```
SELECT name, salary
FROM player
WHERE salary = (SELECT MAX(salary)
                FROM player)
```



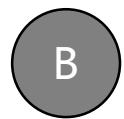
name	salary
Peyton Manning	15000000.00

使用关联子查询选取出各商品种类中高于该商品种类平均销售单价的商品，结果返回商品种类、商品名称和销售单价三列

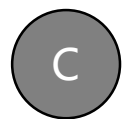
```
SELECT product_type,product_name,sale_price
FROM Product AS P1
WHERE sale_price > (SELECT AVG(sale_price)
                    FROM Product AS P2
                    WHERE P1.product_type= [填空1]
                    GROUP BY product_type)
```



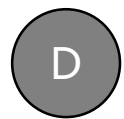
p2.product_type



product_type



p2.product_name



p2.product_id

Product (商品) 表

product_id (商品编号)	product_name (商品名称)	product_type (商品种类)	sale_price (销售单价)
0001	T恤衫	衣服	500
0002	打孔器	办公用品	320
0003	运动T恤	衣服	2800
0004	菜刀	厨房用具	2800
0005	高压锅	厨房用具	6800

提交

视图和临时表

视图是关系表，但它们不是物理存储的。

- 存储查询语句的快照的虚拟表。
- 除非从另一个查询访问视图，否则不会运行或处理查询。
- 每次访问视图时都会重新生成结果。
- 有助于组织代码。

Players(Name, Salary, Height, Weight, Team)

- 示例：创建一个存储 Seahawks 球员信息的视图

```
CREATE VIEW Seahawks AS  
SELECT *  
  FROM player  
 WHERE team = 'Seahawks'
```



```
SELECT MAX(salary)  
FROM Seahawks
```

视图和临时表

临时表将结果存储在tempDB中。

- 很适合存储将来需要多次检索的中间结果

局部临时表（由 #表名 定义）

- 仅适用于当前连接和当前登录。
- 关闭连接时删除。

全局临时表（由 ##表名 定义）

- 创建时可用于任何连接
- 在使用它们的最后一个连接关闭时断开。

视图和临时表

视图是关系表，但它们不是物理存储的。

Players(Name, Salary, Height, Weight, Team)

- 示例：创建一个存储 Seahawks 球员信息的视图

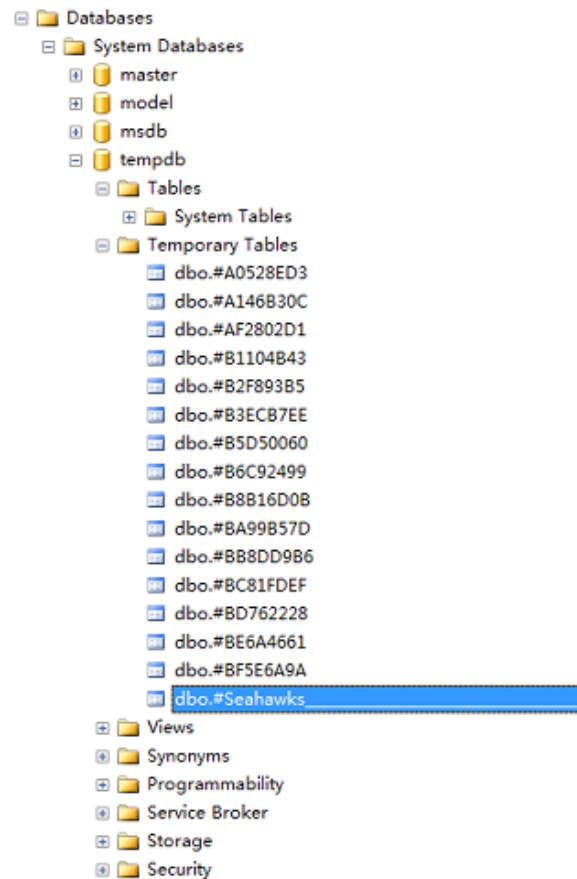
```
SELECT *  
    INTO #Seahawks  
    FROM player  
    WHERE team = 'Seahawks'
```

- 你可以在其他查询中引用临时表

```
SELECT MAX(salary)  
    FROM #Seahawks
```

- 记住在使用完临时表后删除它

```
DROP TABLE #Seahawks
```



二者区别

空间分配：物理空间的分配不一样，视图不分配空间，临时表会分配空间。

虚实：视图是一条预编译的SQL语句，并不保存实际数据，而临时表是保存在tempdb中的实际的表。即视图是一个快照,是一个虚表，而临时表是客观存在的表类型对象。它们的结构一个是表、一个快照。可以把视图想象成联合表的快捷方式。

并、交、差

Purchase(buyer, seller, store, product)

Person(pername, phone number, city)

- 示例：查找居住在西雅图的人或购买 GAP 的人的姓名。

```
(SELECT pername
  FROM person
 WHERE city = 'Seattle')
UNION
(SELECT pername
  FROM person, purchase
 WHERE buyer = pername
        AND store = 'GAP')
```

Outputs from two
tables must have the
same attribute names!

其中一个小技巧是它不会保存重复项。

并、交、差

如果要保留重复项，请使用 ALL 关键字。

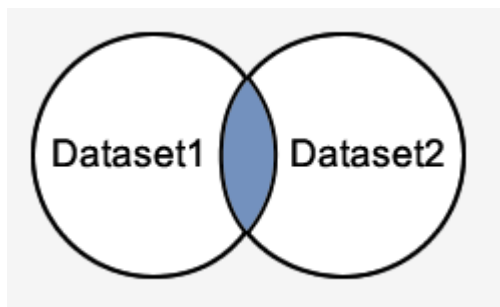
```
(SELECT pername
  FROM person
 WHERE city = 'Seattle')
UNION ALL
(SELECT pername
  FROM person, purchase
 WHERE buyer = pername
       AND store = 'GAP')
```

类似的，您可以使用 INTERSECT 和 EXCEPT.

并、交、差

INTERSECT（交集）返回由左输入查询和右输入查询同时返回的非重复行。

EXCEPT（差集）从左输入查询中返回右输入查询没有找到的所有非重复行。



INTERSECT



EXCEPT

选出Product表中product_id和product_name列不同于Product2表的部分，结果返回

product_id和product_name两列

SELECT product_id,product_name

FROM Product

[填空1]

SELECT product_id,product_name

FROM Product2

- ☐ A UNION
- ☒ B EXCEPT
- ☐ C INTERSECT
- ☐ D UNION ALL

Product表

product_id (商品编号)	product_name (商品名称)	sale_price (销售单价)
0001	T恤衫	500
0002	打孔器	320
0003	运动T恤	2800
0004	菜刀	2800
0005	高压锅	6800

Product2表

product_id (商品编号)	product_name (商品名称)	sale_price (销售单价)
0001	T恤衫	1000
0002	打孔器	500
0003	运动T恤	4000
0006	手套	800
0007	水壶	2000

提交

通常，你会在 INSERT 命令中使用查询来替换值。

Students(Name, StudentID, Gender, Age, Major, Phone)

Freshmen(Name, StudentID, Gender, Age, Major, Phone)

```
INSERT INTO students  
SELECT *  
FROM freshmen
```

选择进入

INSERT INTO: 插入现有表

SELECT INTO: 创建一个包含这些值的新表

- 注意：创建的表将包含选择列表中与源数据相同的数据类型的列。
- 创建一个包含 CEE 学生信息的表

```
SELECT * INTO students  
FROM freshmen
```


删除

- 一般形式:

```
DELETE FROM R  
WHERE <conditions>
```

- 示例

```
DELETE FROM students  
WHERE name = 'Wenbo Zhu'
```

- 一般形式:

```
UPDATE R SET <new-value assignments>  
WHERE <conditions>
```

- 示例

```
UPDATE students  
  SET phone = '111-222-3333'  
WHERE studentid = 1234567
```

```
UPDATE students  
  SET age = age + 1
```

从其他表的更新

- 一般形式:

```
UPDATE R
  SET a.attribute = b.attribute
FROM a JOIN b
  ON <conditions>
WHERE <conditions>
```

- 示例

```
UPDATE accident
  SET a.roadseg = r.segmentid
FROM accident AS a JOIN road AS r
  ON a.roadnumber = r.roadnumber
   AND a.milepost BETWEEN r.begmp AND r.endmp
```

不同类型的约束

- 键，外键
- 属性级约束
- 元组级约束
- 全局约束

约束越复杂，检查和执行就越困难。

定义主键

以下两个查询相等：

```
CREATE TABLE Person(  
    name        VARCHAR(100),  
    ssn         INT PRIMARY KEY,  
    age         SMALLINT,  
    city        VARCHAR(30),  
    gender      CHAR(1),  
    birthdate   DATE  
)
```

```
CREATE TABLE Person(  
    name        VARCHAR(100),  
    ssn         INT,  
    age         SMALLINT,  
    city        VARCHAR(30),  
    gender      CHAR(1),  
    birthdate   DATE,  
    PRIMARY KEY (ssn)  
)
```

定义主键

定义多属性键：

```
CREATE TABLE Person(  
    firstname    VARCHAR(100),  
    lastname     VARCHAR(100),  
    ssn          INT,  
    age          SMALLINT,  
    city         VARCHAR(30),  
    gender       CHAR(1),  
    birthdate    DATE,  
    PRIMARY KEY (firstname, lastname)  
)
```

唯一性约束

其他候选码的唯一性约束:

```
CREATE TABLE Person(  
    firstname    VARCHAR(100),  
    lastname     VARCHAR(100),  
    ssn          INT,  
    age          SMALLINT,  
    city         VARCHAR(30),  
    gender       CHAR(1),  
    birthdate    DATE,  
    PRIMARY KEY (firstname, lastname),  
    UNIQUE (ssn)  
)
```

外键约束

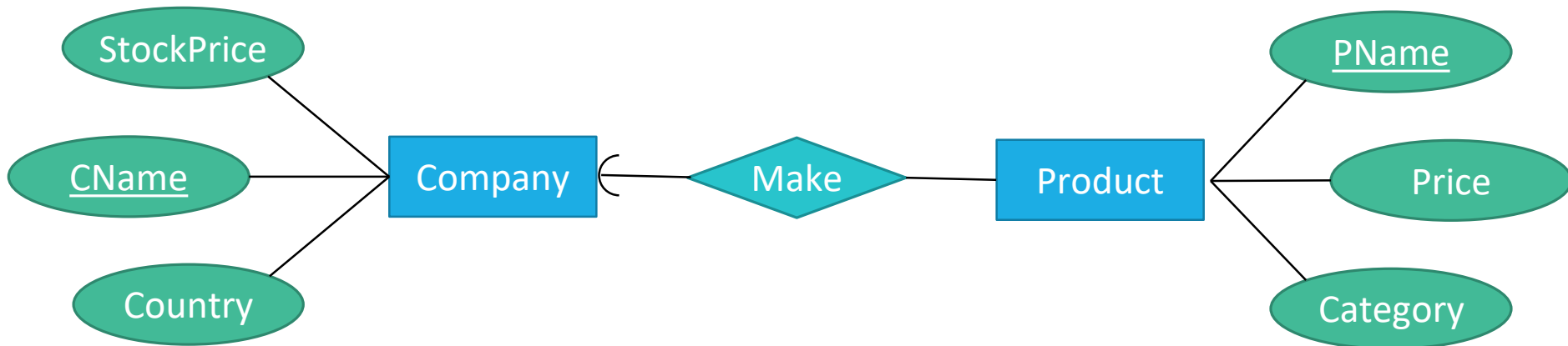
外键（foreign key，FK）是一个列或列的组合，用于在两个表中的数据之间建立和进行链接。

通过将一个表的主键（或唯一）值添加到另一个表中，在两个表之间创建链接。这将成为第二个表中的外键。

将E/R图转换为关系模式时，什么会生成外键？
参照完整性约束。

外键约束

例子:



公司

CName	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

产品

PName	Price	Category	CName
Gizmo	19.99	Gadgets	GizmoWorks
Powergizmo	29.99	Gadgets	GizmoWorks
SingleTouch	149.99	Photography	Canon
MultiTouch	203.99	Household	Hitachi

外键

外键约束

- 声明一个外键

```
CREATE TABLE Product(  
    Pname      CHAR(30) PRIMARY KEY,  
    Category   CHAR(30),  
    Price      FLOAT,  
    CName      CHAR(30) REFERENCES Company(CName)  
)
```

从上表的定义我们能推断出什么？

Cname 是Product中链接 Company (CName) 的外键

Cname 一定是 Company 中的键，但不一定是主键。

外键约束

- 声明外键的另一种方法:

```
CREATE TABLE Product(  
    Pname      CHAR(30) PRIMARY KEY,  
    Category  CHAR(30),  
    Price     FLOAT,  
    CName     CHAR(30),  
    FOREIGN KEY (Cname) REFERENCES Company(CName)  
)
```

外键约束

更新时会发生什么

可能有两种违反规则的情况：

- 对 Product 表的插入或更新会引起 Company 表中找不到的值。
- 对Company 表的删除或更新会导致Product 表的某些元组 “悬挂”

公司

CName	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

产品

PName	Price	Category	CName
Gizmo	19.99	Gadgets	GizmoWorks
Powergizmo	29.99	Gadgets	GizmoWorks
SingleTouch	149.99	Photography	Canon
MultiTouch	203.99	Household	Hitachi

更新时会发生什么

如果对 Product表 的插入或更新会引起 Company表中的不存在，必须拒绝该插入或更新。

删除或更新Company表时，引起Product表中某些元组Cname的删除，可以通过三种方式进行处理：

- 默认值：拒绝修改
- 级联：在 Product表 中进行相同的更改
 - 已删除的Cname：删除相应的Product
 - 更新的CName：更改 Product表 中的值
- 设置NULL：将 Product表 中的CName更改为NULL

更新时会发生什么

- CASCADE独立进行删除和更新。当我们声明一个外键时，我们可以选择从 SET NULL 到 CASCADE 的策略。

- 遵循外键声明：

ON [UPDATE, DELETE][SET NULL, CASCADE]

- 可以使用两个这样的子句，一个用于更新，一个用于删除。
- 否则，使用默认值（reject）。

更新期间会发生什么

SQL示例:

```
CREATE TABLE Product(  
    PName      CHAR(30) PRIMARY KEY,  
    Category  CHAR(30),  
    Price     FLOAT,  
    CName     CHAR(30),  
    FOREIGN KEY (CName) REFERENCES Company(CName)  
    ON DELETE SET NULL  
    ON UPDATE CASCADE  
)
```

当数据库变得很大时，加快查询处理时间是**非常**重要的。

假设我们有一个2亿元组的数据

Person(name, age, city)

```
SELECT *
```

```
FROM Person
```

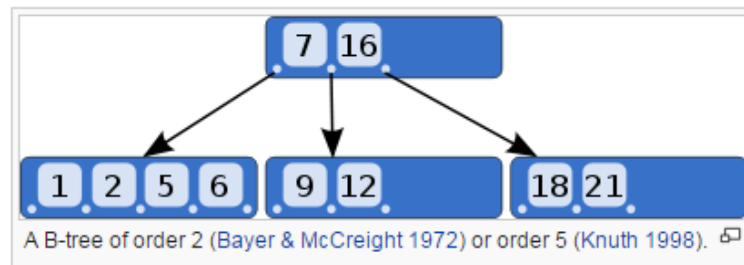
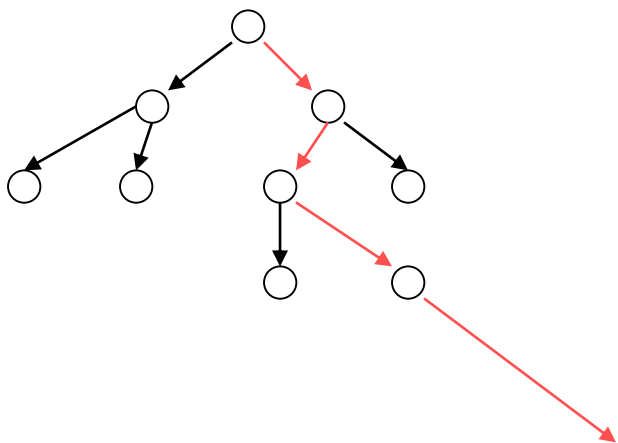
```
WHERE name= 'Smith'
```

文件 Person 的顺序扫描可能需要很长时间！

索引

关系表的属性A上的索引是一种数据结构，它可以有效地查找那些对属性A具有固定值的元组

示例解决方案:按时间创建索引



B-Tree

Adam	Betty	Charles	Smith
------	-------	---------	------	-------	------

两种类型的索引：

- 聚集索引|Clustered indexes (拼音检字法)
- 非聚集索引|Nonclustered Indexes (部首检字法)

聚集索引

- 对数据行重新排序以匹配索引（磁盘上的行按排序顺序排列）
- 每个表**只能有一个**聚集索引
- 适合顺序访问和范围选择
- 根据创建聚集索引的方式插入数据
- 最常见的情况是：主键=>聚集索引
- 每个表都**应该有**聚集索引

非聚集索引

- 页中的数据按随机顺序排列
- 索引中的逻辑数据顺序
- 通常在JOIN、WHERE、ORDER BY中使用的列上创建
- 适用于可能频繁修改值的表
- 语法： Create INDEX->默认情况下非聚集索引
- 在DB表上允许有多个索引

语法？

```
CREATE INDEX ageIndex ON Person (age)
```

B+ trees有助于：

```
SELECT *  
FROM Person  
WHERE age > 25 AND age < 28
```

为什么不所有内容创建索引呢？

创建索引

可以在多个属性上创建索引：

示例：

```
CREATE INDEX doubleindex ON  
Person (age, city)
```

在其中：

```
SELECT *  
FROM Person  
WHERE age = 55 AND city = "Seattle"
```

甚至在：

```
SELECT *  
FROM Person  
WHERE age = 55
```

但是不在：

```
SELECT *  
FROM Person  
WHERE city = "Seattle"
```

索引选择问题

我们得到了一个 workload = 一组 SQL 查询加上它们运行的频率,

我们应该建立哪些索引来加速 workload ?

FROM/WHERE子句 → 支持索引

INSERT/UPDATE子句 → 不支持索引:

关于聚集索引和非聚集索引的描述错误的是()。

- ☐ A 一个表可以有多个非聚集索引，但只能有一个聚集索引
- ☒ B 非聚集索引的值顺序与数据表中记录的物理顺序完全相同
- ☐ C 在建立聚集索引的列上不允许有重复的值
- ☐ D 使用聚集索引查询的速度要比非聚集索引速度快

提交

慕课视频片段

视频名称：Video



温馨提示：此视频框在点击“上传手机课件”时会进行转换，用手机进行观看时则会变为可点击的视频。此视频框可被拖动移位和修改大小