第7章 关系演算

7.1 引言

在第6章中，我们说关系模型的操作部分是基于关系代数的，同样我们也可以说它是基于关系演算的。换句话说，关系代数和关系演算是可以相互替代的。它们之间的基本区别是：关系代数提供了像连接、并和投影等明确的集合操作符，并且这些集合操作符告诉系统如何从给定关系构造所要求的关系；而关系演算仅提供了一种描述（ notation）来说明所要求的关系（这一关系是根据给定关系导出的）的定义。例如：查询提供零件 P2的供应商的号码和所在城市。此查询的一个代数操作形式可以描述如下（在这里有意不使用第 6章的语法形式）：

- 首先，根据供应商号（S#）连接供货商表（supplier）和供货表（shipment）中的元组；
- 其次，在上述连接结果中选择零件号为 P2的元组；
- 最后，将上述选择结果在供应商号（S#）和供应商所在城市（city）列上投影。

相比而言，一个演算形式可以简单地描述为：

- 查取供应商号（S#）和供应商所在城市（city），当且仅当在关系供货中存在这样的一个元组：它具有同样的供应商号（S#），且它的零件号（P#）取值P2。

在后一种形式中，用户仅仅描述了所要求结果的定义，而把具体的连接、选择等操作留给了系统。

于是，我们可以这样说（至少表面地看）：关系演算是描述性(descriptive)形式的，而关系代数是说明性(prescriptive)形式的。关系演算描述了问题是什么，而关系代数说明了解决问题的过程。或者，可以说，关系代数是过程化的（诚然，是高级的，但仍然是过程化的）；而关系演算是非过程化的。

然而，我们强调的上述区别仅仅是表面上的。实际上，关系代数和关系演算在逻辑上是等价的。即每一个代数表达式都有一个等价的演算表达式，每一个演算表达式都有一个等价的代数表达式。两者是一一对应关系。所以两者的区别仅仅是形式上的。可以证明 ,关系演算更接近自然语言,而关系代数更像程序语言。特别地，没有哪一种方法真正地比另外一种更加非过程化。我们将在7.4节详细讨论这一等价问题。

关系演算是基于谓词演算，它是数理逻辑的一个分支。使用谓词演算作为查询语言的基础的思想起源于Kuhns[7.6]的一篇论文。关系演算的概念（即特别适合关系数据库的一个应用型的谓词演算）最早由Codd提出（参考[6.1]）；Codd在另外一篇论文[7.1]中提出了一种基于关系演算的语言，称作数据子语言 ALPHA。ALPHA从没有实际实现过，但有一种叫作QUEL[7.5, 7.10~7.12]的语言主要是参照它研制的。 QUEL语言已经实现，并且一度是 SQL的强劲的竞争对手。

范围变量是关系演算的一个基本特征。简单地说，范围变量就是限制其取值范围在一个指定关系内的变量。即其允许取值是这个关系中的一个元组。所以 ,如果范围变量V限制在关系*r*上,则在任何时候，表达式“ V ”都表示关系*r*的某一元组*t*。例如：查询“ 在London的供应

商的供应商号”，用QUEL语言可以表达如下：

```
RANGE OF SX IS S;
RETRIEVE (SX.S#) WHERE SX.CITY="London";
```

在这里SX是范围变量，且它限制在关系变量 S的当前关系上(RANGE 语句是对范围变量的定义)。RETRIEVE语句可以这样解释：对于变量 SX的每一个取值，要取出其 S#字段的值，当且仅当其CITY字段的值为London。

由于对值为元组的范围变量的依赖性(并且也是为了同域演算区分开来——下文介绍)，所以，起初关系演算就是元组演算。元组演算将在 7.2节详细叙述。注意：为了简单起见，在这本书中，一般用演算和关系演算，而不加“元组”或“域”这样的修饰词，来专指元组演算。

Lacroix和Pirotte提出了另一种演算形式，即域演算（参考[7.7]）。这种演算的范围变量的取值限制在域上而不是在关系上[⊖]。在所有已提出的域演算语言的文献中，可能Query-By-Example语言(QBE，参考[7.14])是最有名的(尽管实际上QBE混合了某些元组演算成分)。现在QBE已经在商业上实现了。在7.6节我们将概略介绍一下域演算内容。参考文献[7.14]简单地讨论了QBE。

注意：在演算讨论中，我们有意省略了一些第 6章中讨论的主题，如传递闭包、分组与分组还原以及关系比较。在演算中我们也忽略了关系更新操作。关于这些问题，参考文献 [3.3]作了简单的讨论。

7.2 元组演算

与第6章中讨论关系代数的方法一样，在这里我们首先介绍语法，然后讨论语义。注意：本章介绍的语法虽与Tutorial D 给出的演算语法不完全一致，但基本风格是一样的。

1. 语法

注意：这一小节的许多语法规则是以比较零散的形式给出的，只有当了解了后面的语义解释后，才能理解其含义。不过这里把它们先罗列在此，便于后面讨论时引用。

这里先给出第6章介绍的关系表达式<*relational expression*>的语法：

```
<relational expression>
     ::=  RELATION{<tuple expression commalist>}
        | <relvar name>
        | <relational operation>
        |(<relational expression>)
```

换句话说，这里的关系表达式和以前的一样，但其中最重要的部分之一关系操作 <*relational operation*>有着不同的解释（以后将会看到）。

```
<range var definition>
     ::=  RANGEVAR <range var name>
          RANGES OVER <relational expression commalist>;
```

在一定的上下文中，范围变量名<*range var name*>可以是元组表达式<*tuple expression*>[⊖]。这种上下文是：

- 范围属性引用<range attribute reference>中量词的点号之前；

⊖ 这一术语是有悖逻辑的：如果域演算之所以叫域演算是因为上面的原因（确实是这样），那么元组演算就可以堂堂正正地叫做关系演算。

- 紧接量化布尔表达式<*quantified boolean expression*>的量词；
- 作为布尔表达式<*boolean expression*>的一个操作数；
- 作为一个<*proto tuple*>或<*proto tuple*>中的表达式或其中的操作数。

```
<range attribute reference>
     ::=  <range var name> . <attribute reference>
[ AS <attribute name> ]
```

在一定的上下文中，<*range attribute reference*>可以用作一个表达式<*expression*>[⊖]。这种上下文是：

- 作为布尔表达式<*boolean expression*>中的一个操作数；
- 作为一个<*proto tuple*>或<*proto tuple*>中的表达式或其中的操作数。

```
< boolean expression >
      ::=  ...一般可能的表达形式，加上：
         | < quantified boolean expression >
```

只有当下面两个条件都成立时，布尔表达式 <*boolean expression*>中范围变量的引用才不受其约束。

- 关系操作<*relational operation*>中直接出现的布尔表达式<*boolean expression*>(即布尔表达式直接跟在关键字WHERE 后面)；
- <*proto tuple*>中直接出现的同一个范围变量的引用（必须是自由的）直接包含在同一个关系操作中（即<*proto tuple*>直在关键字WHERE之前)。

术语解释：在关系演算（包括元组演算和域演算）这一部分里，布尔表达式通常叫做合式公式（well-formed formulas）或简写为WFFs。在大多数如下情况中，我们使用这一术语：

```
< quantified boolean expression >
      ::=  EXISTS <range var name> (<boolean expression>)
         |FORALL <range var name> (<boolean expression>)

<relational operation>
      ::=  <proto tuple> [WHERE <boolean expression>]
```

在第6章的关系代数中我们讲到，关系操作 <*relational operation*>是关系表达式<*relational expression*>的一种形式。但这里我们给出一个不同的定义。对于关系表达式的其他形式，包括关系变量名和关系选择子调用（selector invocation)，都和以前一样，仍然合法。

```
<proto tuple>
    ::=  <tuple expression>
```

所有在原型元组<*proto tuple*>中直接出现的范围变量的引用都必须独立于该原型元组。注意：“ proto tuple ”表示“ prototype tuple ”。在这里这个术语是合适的，但不标准。

2. 范围变量

下面给出一些范围变量的例子（用供应商表和零件表为例）：

```
RANGEVAR  SX RANGES OVER S;
RANGEVAR  SY RANGES OVER S;
RANGEVAR  SPX RANGES OVER SP;
RANGEVAR  SPY RANGES OVER SP;
RANGEVAR  PX RANGES OVER P;
```

⊖ 这里不再给出<tuple expression> 的具体描述，具体请看第6章及其给出的例子。

```
RANGEVAR  SU RANGES OVER
          ( SX WHERE SX.CITY = 'London' )
          ( SX WHERE EXISTS SPX (SPX.S# = SX.S# AND
                                 SPX.P# SPX = P# ('P1') ) );
```

上例中的范围变量SU定义在一组元组集合的并上，这个集合并是：所有既住在伦敦又提供零件P1的供应商的元组。注意：范围变量SU的定义利用了范围变量SX和SPX；因此根据并的定义，所有参加“并”的关系，其类型必须是一样的。

注意：从通常的程序语言的角度讲，范围变量并不是变量。这里讲的变量是从逻辑意义上讲的。实际上，与第3章讨论的谓词*placeholders*（占位符）或*parameters*（参数）有点类似。不同的是：第3章讲的*placeholders*取值为域值，而元组演算中的元组变量取值为元组。

在本章以后的讨论中，我们将采用上述范围变量的定义。我们注意到，在实际的语言中，应该有一些规则去限定这样定义的范围。但在本章中我们忽略了这问题。

3. 自由变量引用和约束变量引用

在一些上下文中，尤其在合式公式中，每一个变量的引用要么是自由的，要么是约束的。首先我们从纯语法的角度解释这个概念，接着继续讨论它的重要性。

设V为范围变量，则：

- 在合式公式（WFF）“NOT *p*”中的V引用是否受此合式公式（WFF）的约束，要看它们是否受*p*的约束。在合式公式“(*p* AND *q*)”和“(*p* OR *q*)”中的V引用是否受此合式公式的约束，要看它们是否受*p*或*q*的约束。
- 如果V引用在合式公式“ *p* ”中是自由的，则它在合式公式“ EXISTS V (*p*)”和“FORALL V(*p*)”中一定是受约束的。其它在“*p*”中的范围变量的引用是否受合式公式“EXISTS V(*p*)”和“FORALL V(*p*)”的约束，要看它们是否受“*p*”的约束。

为了完整性，我们再增加下面几条：

- 在<*range var name* >“V”中的V单独引用(sole reference)是不受此<*range varname*>的约束的。
- 在<*range attribute reference*>“V.*A*”中的V单独引用是不受此 <*range attribute reference*>约束的。
- 如果在某一表达式*exp*中V引用是自由的，那么在其他任何包含 *exp*的表达式*exp*'中，此V引用也是自由的，除非表达式*exp*'中有某一量词限制了此引用。

下面是包含范围变量的合式公式的一些例子：

- 简单比较：

```
SX.S# = S# ( 'S1' )
SX.S# = SPX.S#
SPX.P# ≠ PX.P#
```

在此例中，所有对SX、PX和SPX的引用都是自由的。

- 简单比较的布尔联合：

```
PX.WEIGHT < WEIGHT (15.5 ) OR PX.CITY = ' Rome '
NOT ( SX.CITY = 'London ' )
SX.S# = SPX.S# AND SPX.P# ≠ PX.P#
PX.COLOR = COLOR ( ' Red ' ) OR PX.CITY = ' London '
```

在此例中，所有对SX、PX和SPX的引用也都是自由的。

• 量化合式公式：

```
EXISTS SPX ( SPX.S# = SX.S# AND SPX.P# = P# (' P2 ') )
FOR ALL PX ( PX.COLOR = COLOR (' Red ') )
```

在这两个例子中，SPX和PX 引用是约束的，SX 的引用是自由的。

4. 量词

量词有两个，即EXISTS和FORALL。EXISTS是存在量词，而FORALL是全称量词。基本上讲，如果*p*一是合式公式(WFF)，且此公式中V是自由的，则：

```
EXISTS V ( p )
```

和

```
FORALL V ( p )
```

是合法的合式公式，且在两者中 V都是受约束的。第一句的意思是：至少存在一个 V值，使得*p*为真。第二句的意思是：对所有的 V值，*p*总是为真。例如：假设变量 V限制在“ 1999年美国参议院议员 ”这个集合体中，并假设 *p*是合式公式“ V都为女性 ”（当然，在这里，我们不试图使用形式化语法！），则“ EXISTS V (*p*) ”和“ FORALL V (*p*) ”都是合法的合式公式，并且它们的取值分别为真和假。

再看上一节结束时EXSITS 量词的例子：

```
EXISTS SPX ( SPX.S# = SX.S# AND SPX.P# = P# (' P2 ') )
```

根据以上所述，我们可以这样理解此合式公式：在关系变量 SP的当前值里存在一个 SPX，其S#的字段值等于任一 SX.S#字段值，且P#取值为P2。在这里SPX的引用是约束的，只有 SX的引用是自由的。

我们可以将EXSITS量词理解为多个OR的重复。换句话说，如果（a）*r*是一个关系，且其中有元组*t1*,*t2*,...,*tm*; (b) V是一个定义在*r*上的范围变量；(c) *p* (V)是一个合式公式 ，且在这里 V 是自由变量，则合式公式：

```
EXISTS V ( p ( V ) )
```

就等于：

```
false OR p ( t1 ) OR ... ORp ( tm )
```

特别地，如果*r*是空的（即*m*为零），则此表达式取值为假。

下面我们通过例子来说明。首先假设关系 *r*包含下面的元组：

```
(1, 2, 3)
(1, 2, 4)
(1, 3, 4)
```

为了简单起见，再假设（ a）这三个属性按从左到右的顺序分别记为 *A*、*B*和*C*；（b）每一属性都取整型。这样，下面的合式公式就有所示的取值：

```
EXISTS V ( V.C > 1 )                    true
EXISTS V ( V.B > 3 )                    false
EXISTS V ( V.A > 1 OR VC= 4 )          :true
```

现在，我们开始讨论FORALL量词。还是从上一小节结束时的例子开始：

```
FORALL PX ( PX.COLOR = COLOR (' Red ') )
```

我们可以如下理解此合式公式：在关系变量 P的当前值中，对所有的元组 PX，其COLOR字段的取值为RED。在这里两次PX引用都是受约束的。

正像我们把EXISTS量词理解为OR的重复使用一样，我们把FORALL量词理解为AND的重复。换句话说，如果*r*、V和*p*(V)都是和上述EXISTS量词中讨论的一样，则合式公式：

```
FORALL V ( p ( V ) )
```

可以理解为：

```
true AND p ( t1 ) AND ... ANDp ( tm )
```

特别地，如果*R*是空的（即*m*为零），则此表达式取值为真。

下面我们通过例子来说明。首先假设关系 *r*包含和上面同样的元组，则下面的合式公式（WFF）就有所示的取值：

```
FORALL V ( V.A> 1 )                    : false
FORALL V ( V.B> 1 )                    : true
FORALL V ( V.A=1 AND V.C >2 )          : true
```

注意：FORALL量词的使用仅仅是为了方便，实际上它不是基本的。为了更加明确，请看下面的式子：

```
FORALL V ( p )  ≡  NOT EXISTSV  ( NOT p )
```

(不严格地说，就是“所有使得 *p*为真的V”和“不存在这样的V使得*p*为假”这两种说法是一样的)。此式表明：任何一个包含FORALL的合式公式都可以被一个等价的包含EXISTS的合式公式代替。例如，下面的语句（值为真）：

对所有的整数*x*，存在一个整数*y*，使得$y > x$

（即每一个整数都有一个比它大的整数）等价于下面的语句：

不存在一个整数*x*，使得不存在一个整数*y*，使得$y > x$

（即不存在一个最大整数）。但在通常情况下，用FORALL量词比用EXISTS量词及双重否定更容易理解。在实际引用中，这两个量词都要求支持。

5. 自由变量引用和约束变量引用补充

假设*x*的取值限制在整数集上，看下面的合式公式：

```
EXISTS x ( x > 3 )
```

注意到这里*x*是哑元（dummy），它仅仅起到连接括号里面的布尔表达式和外面的量词的作用。这个合式公式只是说明：存在某个整数 *x*，它比3大。因此，如果所有 *x*的引用被其他的一些变量*y*的引用所代替，则此合式公式的意义不会发生变化。即：合式公式

```
EXISTS y (y > 3 )
```

在语义上和上式是一样的。

再看下面的合式公式：

```
EXISTS x ( x > 3 ) ANDx < 0
```

此式对*x*进行了三次引用，表示两个不同的变量。前两次引用是受约束的，它可以被其他的引用所代替而不改变整句的意思。第三个引用是自由的，它不能随便地被替换。所以对下面的合式公式，第一个和上面的等价，第二个就不是：

```
EXISTSy ( y > 3 ) ANDx < 0
EXISTS y ( y > 3 ) ANDy < 0
```

而且，还要注意：如果不知道自由变量引用 *x*表示的合式公式的值，就不能确定原合式公式的

值。反过来，如果一合式公式的所有变量引用都是受约束的，也不能确定此合式公式的值一定为真或假。这里有两个术语：一个是封闭式合式公式（closed WFF），就是说在此公式内的所有变量引用都是受约束的（实际上这是一个命题）；另一个是开放式合式公式（open WFF），所有非封闭式合式公式都归为此类，即其中至少包含一个自由变量引用。

6. 关系操作

关系操作这个术语在演算里也许不太合适，关系定义可能更加合适。我们这样用是为了和第6章保持一致性。参看下面的语法：

```
< relational operation >
      ::=  < proto tuple > [ WHERE < boolean expression > ]
< proto tuple >
      ::=  < tuple expression >
```

回忆一下前面的语法规则，在此我们稍微作了一点修改：

- 首先，所有在原型元组中引用的范围变量都必须不受此原型元组的约束。
- 其次，WHERE子句中的范围变量的引用可能是自由的，仅当在相应的原型元组中同一范围变量的引用存在，并且是自由的。

例如，下述操作是一个合法的关系操作（“找位于伦敦的供应商的供应商号”）：

```
SX.S# WHERE SX.CITY =  'London'
```

在此原型元组中，SX的引用是自由的；在WHERE子句中，SX的引用也是自由的。这种应用是合法的。因为在这一原型元组中出现的是同一范围变量，且它们是自由的。

再看下面的例子（“找供应零件P2的供应商的供应商号”——参看本小节前面关于EXISTS量词的讨论）：

```
SX.SNAME WHERE EXISTS SPX ( SPX.S# = SX.S# AND
                            SPX.P# = P# ( 'P2' ) )
```

这里SX的引用都是自由的；WHERE子句中的SPX引用正如它必须是的那样，都是受约束的，因为在这个原型元组中没有同一范围变量的引用。

直观上看，一个给定的关系操作等价于包含每一个原型元组可能的值的关系。并且对这些原型元组来说，在WHERE子句中指定的布尔表达式取值为真（若省略了WHERE子句，就默认为WHERE子句取值为真）。为了更加明确，下面作出更加详细的解释：

- 首先，一个原型元组是一个可能的多项的集合。在此集合中，每一项要么是一个范围属性引用（可能包含一个AS子句来引入一个新的属性名），要么是一个简单的范围变量名㊀。但是：

  a) 这里的范围变量名基本上是范围属性列表的简记，而范围属性就是此范围变量被限制的关系的每一个属性；

  b) 没有AS子句的范围属性的引用基本上仅是一个简记，即其中每一个新属性名与原属性名是一样的。

  因此不失一般性，我们把原型元组看作范围属性的列表，如：*Vi.Aj AS Bj*。注意：*Vi*和*Aj*不必都是明确的，但*Bj*一定要明确。
- 设原型元组中的范围变量定为 *V1,V2,...,Vm*；设这些范围变量定义在其上的这些关系分

㊀ 特别注意：从现在开始，无论怎样，我们都要把注意力放在这两个可能性上。

别为：*r1*,*r2*,...,*rm*；在应用AS子句中的属性重命名之后,设相应的关系为*r1*',*r2*',..., *rm*'；设*r*'是*r1*',*r2*',...,*rm*'的笛卡尔积。

- 设*r*是*r*'中使得WHERE子句中合式公式的取值为真的子集。注意：为了这些说明，我们还要假设前面步骤中WHERE子句中的重命名是应用在属性上。然而，实际上，具体语法并不依赖于这一假设，而是依赖于圆点符去消除必要的岐义。下一小节我们还将作说明。
- 所有关系操作的值都是定义在所有的上的关系 *r*的投影。

具体请看下一节的例子。

7.3 举例

下面，我们用明确的查询给出一些有关演算使用的例子。作为练习，为了对比起见，你可以试着给出代数形式。

1. 找出位于巴黎且其状态（status）大于20的供应商的供应商号及状态

```
(SX.S#, SX.STATU$
WHERE SX.CITY = ' Paris ' AND SX.STATUS > 20
```

2. 找出所有成对的住在同一城市的供应商的供应商号

```
(  SX.S# AS SA, SY.S# AS SB )
                WHERE SX.CITY = SY.CITY AND SX.S# < SY.S#
```

注意：原型元组中给出了最终结果的属性名；这些名字在 WHERE 子句中是不能使用的，这就是为什么WHERE 子句的第二个比较用“ SX.S# < SY.S# ”的形式，而不是“ SA < SB ”的形式。

3. 找出供应零件P2的所有的供应商的信息

```
SX
WHERE EXISTS SPX ( SPX.S# = SX.S# AND SPX.P# = P# ( ' P2 ' )
```

注意此原型元组中范围变量名的使用。这个例子可以写成以下形式：

```
( SX.S#, SX.SNAME, SX.STATUS, SX.CITY )
WHERE EXISTS SPX ( SPX.S# = SX.S# AND SPX.P# = P# ( ' P2 ' )
```

4. 找出至少供应一个红色零件的供应商名

```
SX.SNAME
WHERE EXISTS SPX ( SX.S# = SPX.S# AND
                   EXISTS PX ( PX.P# = SPX.P# AND
                               PX.COLOR = COLOR ( ' Red ' ) ) )
```

或者用下述前束范式表示。此式中所有量词都提到合式公式之前。

```
SX.SNAME
WHERE EXISTS SPX ( EXISTS PX ( SX.S# = SPX.S# AND
                               SPX.P# = PX.P# AND
                               PX.COLOR = COLOR ( ' Red ' ) ) )
```

前束范式并不是天然地比其他形式更正确或没有其他形式正确，但是在许多情况下，这种范式比较自然明确。而且，它的使用有可能使得括号的数量减少。例如，看下面的合式公式：

```
quant1 vble1 ( quant2 vble2( wff ))
```

(这里，每一个*quant1*和 *quant2*或者是EXISTS或者是 FORALL)。此式可能很随意，但很明确。此式可以缩写为：

```
quant1 vble1 quant2 vble2 ( wff )
```

这样我们可以简化上面的演算表达式，如下：

```
SX.SNAME
WHERE EXISTS SPX EXISTS PX ( SX.S# = SPX.S# AND
                             SPX.P# = PX.P# AND
                             SPY.COLOR = COLOR ( ' Red ' ))
```

然而，为了简明起见，我们将继续使用所有的括号。

5. 找出至少供应S2供应的零件中的一个的供应商名

```
SX.SNAME
WHERE EXISTS SPX ( EXISTS SPY ( SX.S# = SPX.S# AND
                                SPX.P# = SPY.P# AND
                                SPY.S# = S# ( ' S2 ' ) ) )
```

6. 找出供应所有零件的供应商名

```
SX.SNAME WHERE FORALL PX ( EXISTS SPX ( SPX.S# = SX.S# AND
                                       SPX.P# = PX.P# ))
```

不使用FORALL量词，可以等价地表示为：

```
SX.SNAME WHERE NOT EXISTS PX ( NOT EXISTS SPX
                                  ( SPX.S# = SX.S# AND
                                    SPX.P# = PX.P# ))
```

7. 找出不供应零件P2的供应商名

```
SX.SNAME WHERE NOT EXISTS SPX
                    ( SPX.S# = SX.S# AND SPX.P# = P# ( ' p2 " ))
```

注意：此结果很容易从3的结果导出来。

8. 找出至少由供应商S2所供应的零件的供应商号

```
SX.S# WHERE FORALL SPX ( SPX.S# = S# ( ' S2 ' ) OR
                       EXISTS SPY ( SPY.S# =SX.S# AND
                                    SPY.P# = SPX.P# ))
```

此句可解释为：取供应商SX的号码,对于每一个SPX，下列条件为真：要么此供货不是来自于供应商S2；要么，存在一个供货SPY，使得SPY成为SX供应的SPX中的一员。

对这种复杂的查询，我们引入了另一个简单明了的语法简写形式，即逻辑蕴含。如果 *p*和*q*是合式公式，则逻辑蕴含表达式是：

```
IF p THEN q END IF
```

也是一个合式公式，在语义上和下式相同：

```
( NOT p ) OR q
```

此例还可以用下列形式表达：

```
SX.S# WHERE FORALL SPX ( IF SPX.S# = S# ( ' S2 ' ) THEN
                           EXISTS SPY  ( SPY.S# = SX.S# AND)
                                       SPY.P# = SPX.P# )
                         END IF )
```

此句可以这样理解：取供应商SX的供应商号，对于每一个SPX，下列条件为真：如果供货SPX由供应商S2供应，则存在一个供货SPY,使得SPY成为SX供应的SPX中的一员。

9. 找出重量超过16磅或由供应商S2供应的零件

```
RANGEVAR PU RANGES OVER
        ( PX.P# WHERE PX.WEIGHT > WEIGHT ( 16.0 )),
        ( SPX.P# WHERE SPX.S# = S# ( ' S2 ' ) );
PU.P#
```

这与关系代数相似，包含了一显式并。

下面我们给出此查询的一个可替换形式。但这第二个式子要依赖于关系变量P的零件号包含了关系变量SP的零件号，这一点是并的形式不可缺少的。

```
PX.P# WHERE PX.WEIGHT > WEIGHT ( 16.0 )
       OR  EXISTS SPX ( SPX.P# = PX.P# AND
                        SPX.S# = S# ( ' S2 ' ) )
```

7.4 关系演算与关系代数的比较

在引言里我们就谈到关系演算与关系代数是等价的，现在我们更加具体地讨论这一点。首先，Codd认为，至少关系代数跟关系演算一样，具有强大的表达能力（参看[6.1]）。他给出了一算法证明了这一点。这一算法叫Codd简约算法。通过这个算法，任意一个演算表达式都可以简约为在语法上等价的代数表达式。这里我们不具体提供Codd的算法，但是我们用较大众化的术语举一个适当复杂的例子来解释这个算法如何实现⊖。

现在我们来举一个例子。不使用我们熟悉的供应商与零件数据库，而使用扩展的供应商-零件-工程数据库（参看第3章及其他一些地方）。为方便起见，我们用图7-1给出一些示例（参看第4章图4-5）。

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Rome
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

J

J#	JNAME	CITY
J1	Sorter	Paris
J2	Display	Rome
J3	OCR	Athens
J4	Console	Athens
J5	RAID	London
J6	EDS	Oslo
J7	Tape	London

SPJ

S#	P#	J#	QTY
S1	P1	J1	200
S1	P1	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S2	P3	J4	500
S2	P3	J5	600
S2	P3	J6	400
S2	P3	J7	800
S2	P5	J2	100
S3	P3	J1	200
S3	P4	J2	500
S4	P6	J3	300
S4	P6	J7	300
S5	P2	J2	200
S5	P2	J4	100
S5	P5	J5	500
S5	P5	J7	100
S5	P6	J2	200
S5	P1	J4	100
S5	P3	J4	200
S5	P4	J4	800
S5	P5	J4	400
S5	P6	J4	500

图7-1 供应商-零件-工程数据库(含样本数据)

⊖ 实际上，这个算法（参看[6.1]）有点缺陷（参看[7.2]）。而且，Codd在他的论文里定义的演算并没有对类似并操作作出一些描述，因此严格地讲，他的演算的功能没有他的代数的能力强。不过，可以确认，给关系演算增加了此功能后，它们是等价的。这一点已经有人证实了。如：Klug（参看[6.12]）。

现在来看查询：查找在 Athens城市至少供应一个工程项目且每种零件至少供应 50个的供应商名及其所在城市。这个查询的一个演算表达式是：

```
( SX.SNAME, SX.CITY ) WHERE EXISTS JX FORALL PX EXISTS SPJX
                                ( JX.CITY = ' Athens ' AND
                                  JX.J# = SPJX.J# AND
                                  PX.P# = SPJX.P# AND
                                  SX.S# = SPJX.S# AND
                                  SPJX.QTY≥ QTY ( 50 ) )
```

其中SX、PX、JX和SPJX分别是定义在S、P、J和SPJ上的范围变量。下面我们来看此查询如何进行计算。

步骤1：对每一个范围变量，只要可能，就对其进行选择而确定其范围（即确定所有可能的值）。这样，就意味着在WHERE子句里嵌入了限制性条件，使得 WHERE子句一执行，就消减了一定的元组。本例中，确定后的元组是：

SX ：S中所有元组 5个元组

PX ：P中所有元组 6个元组

JX ：J中CITY='Athens'的元组 2个元组

SPJX ：SPJ中QTY≥QTY(50)的元组 24个元组

步骤2：根据步骤1的结果，构造笛卡尔积，得下表：

S#	SN	STATUS	CITY	P#	PN	COLOR	WEIGHT	CITY	J#	JN	CITY	S#	P#	J#	QTY
S1	Sm	20	Lon	P1	Nt	Red	12.0	Lon	J3	OR	Ath	S1	P1	J1	200
S1	Sm	20	Lon	P1	Nt	Red	12.0	Lon	J3	OR	Ath	S1	P1	J4	700
..	..	..	...	..	..	...	..	...	..	..	...	..	..	..	...
..	..	..	...	..	..	...	..	...	..	..	...	..	..	..	...
..	..	..	...	..	..	...	..	...	..	..	...	..	..	..	...

（等等）。完全的笛卡尔积的结果是5*6*24=1440个元组。注意：由于空间有限，上表只列出了一部分。我们没有给上述属性重命名，其实为了避免模糊性，应该这样做。但是上表中用不同的位置表明相同属性名的属性来自哪些表。这种非正式的运用仅仅是为了简化说明。

步骤3：根据WHERE子句的连接部分，来限制并选择步骤 2中构造的笛卡尔积。此例中，这一连接部分是：

```
JX.J# = SPJX.J# AND PX.P# = SPJX.P# AND SX.S# = SPJX.S#
```

因此，对于上述笛卡尔积中那些供应商 S#值不等于供货S#值的元组、那些零件P#值不等于供货P#值的元组、那些工程J# 的值不等于供货J# 的值的元组，统统给删掉。这样，我们得到上述笛卡尔积的一个子集，是十个元组，如下所示：

S#	SN	STATUS	CITY	P#	PN	COLOR	WEIGHT	CITY	J#	JN	CITY	S#	P#	J#	QTY
S1	Sm	20	Lon	P1	Nt	Red	12.0	Lon	J4	Cn	Ath	S1	P1	J4	700
S2	Jo	10	Par	P3	Sc	Blue	17.0	Rom	J3	OR	Ath	S2	P3	J3	200
S2	Jo	10	Par	P3	Sc	Blue	17.0	Rom	J4	Cn	Ath	S2	P3	J4	200
S4	Cl	20	Lon	P6	Cg	Red	19.0	Lon	J3	OR	Ath	S4	P6	J3	300
S5	Ad	30	Ath	P2	Bt	Green	17.0	Par	J4	Cn	Ath	S5	P2	J4	100
S5	Ad	30	Ath	P1	Nt	Red	12.0	Lon	J4	Cn	Ath	S5	P1	J4	100
S5	Ad	30	Ath	P3	Sc	Blue	17.0	Rom	J4	Cn	Ath	S5	P3	J4	200
S5	Ad	30	Ath	P4	Sc	Red	14.0	Lon	J4	Cn	Ath	S5	P4	J4	800
S5	Ad	30	Ath	P5	Cm	Blue	12.0	Par	J4	Cn	Ath	S5	P5	J4	400
S5	Ad	30	Ath	P6	Cg	Red	19.0	Lon	J4	Cn	Ath	S5	P6	J4	500

当然，此表为等值连接结果。

步骤4：从右到左，运用量词，如下所示：

• 对于量词 EXISTS *RX* (这里*RX*是限制在关系*r*上的范围变量)，在这中间结果上进行投影

而删除关系*r*上的所有属性。

- 对于量词FORALL *RX*，用步骤1中确定的跟*RX*有关的、限制了范围的关系去除以这中间结果。注意：这里讲的除，就是Codd讲的除操作（参见[6.3]）。

此例中，这些量词是：

```
EXISTS JX FORALL PX EXISTS SPJX
```

因此：

1) （*EXISTS SPJX*）通过投影消去SPJ的属性，即：SPJ.S#、SPJ.P#、SPJ.J#和SPJ.QTY，结果是：

S#	SN	STATUS	CITY	P#	PN	COLOR	WEIGHT	CITY	J#	JN	CITY
S1	Sm	20	Lon	P1	Nt	Red	12.0	Lon	J4	Cn	Ath
S2	Jo	10	Par	P3	Sc	Blue	17.0	Rom	J3	OR	Ath
S2	Jo	10	Par	P3	Sc	Blue	17.0	Rom	J4	Cn	Ath
S4	Cl	20	Lon	P6	Cg	Red	19.0	Lon	J3	OR	Ath
S5	Ad	30	Ath	P2	Bt	Green	17.0	Par	J4	Cn	Ath
S5	Ad	30	Ath	P1	Nt	Red	12.0	Lon	J4	Cn	Ath
S5	Ad	30	Ath	P3	Sc	Blue	17.0	Rom	J4	Cn	Ath
S5	Ad	30	Ath	P4	Sc	Red	14.0	Lon	J4	Cn	Ath
S5	Ad	30	Ath	P5	Cm	Blue	12.0	Par	J4	Cn	Ath
S5	Ad	30	Ath	P6	Cg	Red	19.0	Lon	J4	Cn	Ath

2) （*FORALL PX*）用步骤1中P表的结果去除以上述结果，得：

S#	SNAME	STATUS	CITY	J#	JNAME	CITY
S5	Adams	30	Athens	J4	Console	Athens

3) （*EXISTS JX*）再通过投影消去J的属性，即：J.J#、J.NAME和J.CITY，结果是：

S#	SNAME	STATUS	CITY
S5	Adams	30	Athens

步骤5：根据原型元组中的选择，对4的结果进行投影。在本例中原型元组是：

```
(SX.SNAME, SX.CITY)
```

因此，最终结果是：

SNAME	CITY
Adams	Athens

从前面的叙述可以得出：原始的演算表达式在语义上等价于某一嵌套的代数表达式。精确地讲，就是：四个选择的积的选择的投影的除的投影的投影！

当然，对这种算法还能作出许多改进（具体参看第17章及[17.5]，那里有一些改进的思想），上述例子的解释已经涉及了不少细节；不过，它没有充分地给出简约工作的一般思想。

还有，现在我们能够解释为什么Codd精确地定义了他列出的8个代数操作符的理由之一（其实不仅仅是一个理由）。这8个操作符，作为一种手段，为演算实现的可能提供了一种方便的目标语言。如：QUEL语言就是建立在这个演算的基础上，这种语言实现的一个可能的途径就是用户提交查询——这个查询基本上就是一个演算表达式——再运用简约算法，最后获得一个等价的代数表达式，这些都是在内部实现的。下一步就是继续优化此代数表达式，这在第17章介绍。

另一点要说明的是，Codd的8个代数操作符也为测度任一给定的数据库语言提供了一标准。在6.6节里我们已简单地涉及了这个问题，这里再深入地讨论一下。

首先，如果一门语言具备了关系演算这样的功能，那么它就可以说成是关系完备的(relational complete)。这就是说，如果某一关系能被演算表达式定义，那它就一定能被符合关系代数（参看[6.1]）的语言的某种表达所定义（在第6章，我们说关系完备就是它具备关系代数的功能，而不是关系演算。但是，正如我们将看到的，这两者是一回事。注意：从 Codd的简约算法，直接可得到关系代数具有关系完备性）。

一般情况下，关系完备被认为是数据库语言表达能力的一种测度。特别地，由于关系代数和关系演算都具备关系完备性，故不必运用循环（在面向终端用户的语言中，它是一重要的方法，同时它对程序员也特别有用），就可以为具备这种表达能力的语言的设计提供一个基础。

其次，由于关系代数具备关系完备性，故为了表现某一给定语言 L具备此特征，就必须充分表现：（a）L包括类似8个代数操作符的操作（实际上，要充分表现关系代数的 5个基本的操作）；（b）L语言的任一操作符的操作数可能是 L的任一表达。SQL是用这种方式表示关系完备的语言的一个例子（参看练习 7.9），而QUEL是另一个例子。事实上，在实践中，要表示一门语言具备关系代数操作比表示它具备关系演算表达容易得多。这就是我们为什么解释关系完备性用代数的方式而不用演算的方式。

还有，要注意，关系完备并不意味着其他方面完备。例如一门语言也可能要求提供计算完备，即它应该能计算可能计算的函数。计算完备是我们在第 6章中给关系代数增加EXTEND和SUMMARIZE两个操作符的动机之一。下一节我们讨论这些操作的关系演算。

我们回到关系代数和关系演算等价的问题上：通过例子，我们已经表明，演算能够被等价地转化为代数，因此关系代数至少具有关系演算的功能；相反，代数能够被等价地转化为演算，因此关系演算至少具有关系代数的功能（参看 Ullman [7.13] ,它很好地说明了这两者逻辑上是等价的）。

7.5 计算能力

尽管我们早些时候没有明确地指出，但事实上，正如我们定义的那样，关系演算已包括类似关系代数中的EXTEND和SUMMARIZE那样的操作，因为：

- 一个可能的原型元组的形式是元组选择子调用 *<tuple selector invocation>*，并且其组成部分是任意的表达式。
- 一个布尔表达式中比较操作的比较数可能也是一个任意的表达式。
- 聚集算子调用 *<aggregate operator invocation>* 中的首个或仅有的变元是一个关系操作 *<relational operation>*。

注意：参看[3.3]，那里对Tutorial D 给出了充分的解释。

没有必要对语法和语义作深入地探讨，这里我们给出了一些例子，并且这些例子也作了一些简化。

1. 找出重量超过1000克的零件的零件号及重量

```
( PX.P# , PX.WIGHT * 454 AS GMWT )
                WHERE PX.WEIGHT * 454 > WEIGHT ( 10000.0 )
```

注意到原型元组中的说明“ AS GMWT ”，它给出结果中可用的名字。这个名字不能在WHERE子句中使用。这就是为什么上式中“ PX.WEIGHT * 454 ”出现了两次。

2. 找出所有供应商的信息，并且对供应商值用 tag表示

```
( SX, ' Supplier ' AS TAG )
```

3. 对每一个供货表，取出所有的供货信息，包括供货总重

```
( SPX, PX.WEIGHT * SPX.QTY AS SHIPWT WHERE PX.P# = SPX.P#
```

4. 对每一个零件，取出每一个零件号和其供货总数量

```
( PX.P#, SUM ( SPX WHERE SPX.P# = PX.P#, QTY ) AS TOTQTY )
```

5. 查找总的供货数量

```
SUM ( SPX, QTY ) AS GRANDTOTAL
```

6. 对于每一个供应商，取出其供应商号及其供应的零件的总数量

```
( SX.S# , COUNT ( SPX WHERE SPX.S# = SX.S# ) AS #_OF_PARTS )
```

7. 找出保存至少5个红色零件的城市

```
RANGEVAR PY RANGES OVER P;

 PX.CITY
 WHERE COUNT ( PY WHERE PY.CITY = PX.CITY
                  AND   PY.COLOR = COLOR ( ' Red ' ) ) > 5
```

7.6 域演算

正如7.1节所述的那样，域演算不同于元组演算，它是定义在域上而不是定义在元组上。本书中，我们仅简单地介绍一下域演算。从实际的角度看，语法上最明显的不同是域演算支持布尔表达式的补充形式，我们把这叫作隶属条件（membership condition）。一个隶属条件可能采取的形式是：

```
R ( pair, pair, ... )
```

这里*R*是关系变量名，且每一个*pair*是*A*:*v*的形式，其中*A*是*R*的一个属性，*v*或者是域演算的范围变量名，或者是选择子调用（selector invocation）（通常用文字描述）。当且仅当存在一个元组，对*R*的当前值无论是什么关系，指定的属性都有指定的值时，这一条件取真值。例如表达式：

```
SP ( S#:S# ( ' S1 ') , P#:P# ( ' P1 ' ) )
```

当且仅当当前存在一供货元组，其 S#取S1，P#取P1，这条件才为真。同样，下面的隶属条件：

```
SP ( S#:SX, P#:PX )
```

当且仅当当前存在一个供货元组，其 S#取范围变量SX的当前值（无论此值是什么），且P#取范围变量PX的当前值（无论此值是什么），上述条件取真值。

在本节最后，我们假设存在域演算的范围变量，如下所示：

域:	范围变量
S#	SX, SY,...
P#	PX, PY,...
NAME	NAMEX, NAMEY,...
COLOR	COLORX, COLORY,...
WEIGHT	WEIGHTX, WEIGHTY,...

```
QTY                                QTYX, QTYY,...
CITY                               CITYX, CITYY,...
STATUS                             STATUSX, STATUSY,...
```

下面是域演算表达式的一些例子：

```
SX

SX WHERE S ( S# : SX )

SX WHERE S ( S# : SX , CITY: ' London ' )

( SX, CITY ) WHERE S ( S# : SX, CITY : CITYX )
           AND   SP ( S# :SX , P# : P# ( ' P2 ' )

( SX, PX ) WHERE S ( S# : SX, CITY : CITYX )
             AND  P ( P# : PX , CITY : CITYY )
             AND  CITYX≠ CITYY
```

不严格地说，第一个式子表示所有的供应商号；第二个表示关系变量 S中所有的供应商号；第三个表示位于伦敦的供应商号；第四个是下面查询的域演算的表示：取供应零件 P2的供应商的供应商号和所在城市（注意，此查询的元组演算表示需要一个存在量词 EXISTS）；第五个是下面查询的域演算的表示：取供应商所在城市与零件出厂地不在同一地方的供应商号和零件号。

用7.3节的例子，我们给出其域演算的表示，作了稍微的改动。

1. **找出位于巴黎且其状态大于20的供应商的供应商号**

```
SX  WHERE EXISTS STATUSX
         ( STATUSX > 20 AND
           S ( S#:SX , STATUS:STATUSX, CITY:' Paris ' ) )
```

就这个例子来讲，它比元组演算显得有点笨拙，并且量词还不能少。不过，有时情况就恰恰相反，请看下面的例子。

2. **找出所有成对的住在同一城市的供应商的供应商号**

```
(  SX AS SA, SY AS SB ) WHERE EXISTS CITYZ
                          (S (S#:SX, CITY:CITYZ ) AND
                           S (S#:SY, CITY:CITYZ ) AND
                           SX < SY )
```

3. **找出至少供应一个红色零件的供应商名**

```
NAMEX WHERE EXISTS SX EXXISTS  PX
          ( S ( S#:SX,SNAME:NAMEX )
            AND SP ( S#:SX,P#:PX )
            AND P (  P#:PX,COLOR:COLOR(' Red ')))
```

4. **找出至少供应S2供应的零件中的一个的供应商名**

```
NAMEX WHERE EXISTS SX EXISTS PX
         ( S ( S#:SX, SNAME:NAMEX )
           AND SP (S#:SX, P#:PX)
           AND SP (S#:S# ('S2'),P#:PX ))
```

5. **找出供应所有零件的供应商名**

```
NAMEX WHERE EXISTS SX ( S ( S#:SX,SNAME:NAMEX )
            AND FORALL PX ( IF P (P#:PX )
                            THEN SP ( S#:SX, P#:PX )
                            END IF )
```

6. 找出不供应零件P2的供应商名

```
NAMEX WHERE EXISTS SX ( S ( S#:SX,SNAME:NSMEX )
                        AND NOT SP ( S#:SX, P#:P# ( ' P2 ' ) ) )
```

7. 找出至少供应S2所供应的零件的供应商号

```
SX WHERE FORALL PX ( IF SP ( S#:SX('S2'), P#:PX )
                     THEN SP ( S#:SX , P#:PX )
                     END IF )
```

8. 找出重量超过16磅或者由S2提供或者两者都具备的零件的零件

```
PX WHERE EXISTS WEIGHTX
         ( P  ( P#：PX，WEIGHT：WEIGHTX)
           AND WEIGHTX > WEIGHT (16.0) )
           OR SP ( S#:S# ('S2'),P#:PX )
```

域演算和关系演算一样，正常情况下都等价于关系代数，即都是关系完备的（详细情况参看[7.13]）。

7.7 SQL语言

在7.4节我们已经谈到，一种给定的关系语言要么基于关系代数，要么基于关系演算。那么SQL语言是基于哪一个呢？很遗憾，SQL只有部分基于这两者，还有一部分并不基于它们。因为提出SQL语言时，就明确要求要不同于这两者（参看[4.8]）。事实上，这个目标就是引入“IN <*subquery*>”结构的动机（参看本节后面的例10）。随着时间的推移，证明代数和演算的一些特征还是需要的，并且SQL也在逐渐地接受它们㊀。今天，SQL语言在某些方面既像代数式的，在某些方面又像演算式的，而某些方面两者都不像。这一现状说明了为什么在第6章我们讲将SQL数据操作的讨论推迟到这一章讲（具体SQL语言哪些部分是基于代数的，哪些部分是基于演算的，哪些部分两者都不基于，这作为练习留给读者自己）。

SQL查询是以表表达式描述的，但其内部很复杂。这里我们不涉及其所有的复杂的内容，而只简单地举出一些例子，希望这些例子能让我们理解其精髓。这些例子基于第4章图4-1的供应商和零件表。要注意此表中的SQL语言没有用户自定义数据类型；事实上，其所有的列都是依据SQL的某一固定类型定义的。注意：附录A给出了一般情况下一个更完全的和更正式的SQL表达式的描述，尤其是SQL表表达式的描述。

1. 找出非巴黎生产的、且重量超过10磅的零件的颜色和出产地

```
SELECT PX.COLOR,PX.CITY
FROM   P AS PX
WHERE  PX.CITY <> 'Paris'
AND    PX.WEIGHT > 10.0;
```

㊀ 其发展的一个结果是（参考［4.18］的注解）：从这一语言中取掉整个“IN < *subquery* >”结构而不会丢失任何功能！这具有讽刺意味，因为在开始的“结构化查询”这一命名中的“结构”正是依赖这一结构；同时，首先采用SQL这一名称而不采用代数或演算也正是依赖于这一结构。

注意：

1) 这里不相等的比较操作符是“ <> ”。通常，SQL中数值比较的操作符是：=、<>、<、>、<=、>=。

2) FROM子句中的“ P AS PX ”的使用。这种描述在当前基本表 P上定义了范围变量 PX（元组型的）。这种定义的作用域就是使用其表表达式。注意：SQL称PX为相关关系名。

3) SQL也支持隐式范围变量的概念。如：上面的查询也很好地等价于：

```
SELECT P.COLOR, P.CITY
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > 10.0;
```

这里基本的思想是:正在使用的表名就指它是隐含的、定义在其上的范围变量（当然不能造成模棱两可的结果）。

此例中，FROM子句“ FROM P ”可以认为是“ FROM P AS P ”的缩写。或者说，如SELECT句和WHERE子句中“ P.COLOR ”中的“ P ”，不是代表基本表P，而是代表具有同名的、定义在该表上的范围变量——这一点必须明确。

4) 正如第4章所述的那样，在此例中我们可以用不受限制的列名，如下所示：

```
SELECT COLOR,CITY
FROM P
WHERE CITY <> ' Paris '
AND   WEIGHT > 10.0;
```

一般的规则是：只有在不引起岐义时，不受修饰的列名才可以使用。但是一般情况下，我们应该使用所有的修饰词，尽管这很冗余。然而，有些上下文中明确地要求列名不受修饰！ORDER BY 子句就是一个例子。请看下面。

5) 在第4章的游标定义中讲的ORDER BY 子句，也能在SQL交互查询中使用，例如：

```
SELECT P.COLOR,P.CITY
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > 10.0
ORDER  BY CITY DESC;
```

6) 再要注意的是第4章的“ SELECT * ”的缩写，例如：

```
SELECT *
FROM  P
WHERE P.CITY <>  'Paris'
AND   P.WEIGHT > 10.0;
```

“ SELECT* ”中的“ * ”号是FROM 子句中所涉及的所有表的所有列的列名的列表的简写。并且它隐含着按原表中列出现的从左到右的顺序。由于此符号的使用，节省了键盘敲击，故在交互式查询中是很方便的。然而，它在嵌入式语言（即 SQL嵌在应用程序中）中却存在潜在的危险。这是因为“ * ”的意思会发生变化。如：用 ALTER TABLE 给表增加列和删除列。

7) （这一点比前面一点更重要）现在，给出表中示例数据，上面的查询将返回四行，而不是两行，即使这里有三行相同。 SQL不会自动地从查询结果里删除多余的重复行，要删除的话，必须要求用户在查询中显式地使用关键字 DISTINCT。例如：

```
SELECT DISTINCT P.COLOR,P.CITY
FROM  P
WHERE P.CITY <> 'Paris'
AND   P.WEIGHT > 10.0;
```

此查询返回两行，而不是四行。

从这可以看出SQL的基本数据对象不是关系，而是表。并且一般情况下，SQL表不是行的集合，而是行的包(bag)（这里的包叫多集(multi-set)，它像一个集合但不允许有重复行）。这样，SQL就违背了信息原则（information principle）(参看3.2节)。所以，SQL的基本操作不是真正的关系操作，而是类似包的操作；而且在关系模型里是正确的结果和原理，由于考虑到表达的变化（例如［5.6］），在SQL里都不一定正确了。

2. 找出所有零件的零件号及其重量

```
SELECT P.P#, P.WEIGHT * 454 AS GMWT
FROM P;
```

这里，“AS GMWT”的说明为结果中的计算列提供了一个适合的列名，于是结果表中的两列分别叫P#和GMWT。如果没有AS子句，则查询结果中相应的列就没有列名。注意到，SQL实际上不允许在这样的环境中给查询结果命名，但在此例中我们仍然这么做了。

3. 找出供应商所在地和零件出产地在同一地方的这两者的所有的信息

SQL提供了许多不同的方法完成这一查询，这里给出三个例子：

1)
```
SELECT S.*,P.P#,P.NAME,P.COLOR,P.WEIGHT
  FROM    S,P
  WHERE   S.CITY = P.CITY;
```
2)
```
S JOIN P USING CITY;
```
3)
```
S NATURAL JOIN P;
```

每种情况下，这结果都是以表S和P的城市进行自然连接。

上面的第一个式子值得讨论，它在SQL最初版本中定义，在SQL/92标准中增加了对显式JOIN连接的支持。从概念上讲，上面的查询是按如下步骤进行的：

- 首先，执行FROM子句，产生S和P的笛卡尔积SP。严格地讲，在计算这个积之前，我们应该当心重命名的列，但这里我们简单地忽略了这个问题。而且，正如我们在第6章练习6.12中看到的那样，一个单独表的笛卡尔积仍然是其本身。
- 其次，当WHERE子句执行时，就根据每行中两个CITY列的值相等对笛卡尔积进行选择，或者说，现在我们已经用CITY列对供应商表和零件表进行了等值连接。
- 最后，当SELECT子句执行时，就根据此列中所指定的列对上述选择结果进行投影。最终结果是自然连接。

因此，不严格地说，在SQL中，FROM子句对应于笛卡尔积，WHERE子句对应于选择，SELECT子句对应于投影。SQL中的SELECT-FROM-WHERE结构表示了笛卡尔积的选择的投影。详情请看附录A。

4. 若供应商为另一个不是他所在的城市供应零件，找出这两个城市名

```
SELECT DISTINCT S.CITY AS SCITY,P.CITY AS PCITY
FROM S JOIN SP USING S# JOIN P USING P#;
```

注意下面的写法是不正确的，因为它在第二个连接中用CITY作为连接列。

```
SELECT DISTINCT S.CITY AS SCITY,P.CITY AS PCITY
```

```
FROM    S NATURAL JOIN SP NATURAL JOIN P;
```

5. 若两个供应商在同一城市，找出他们的供应商号

```
SELECT A.S# AS SA,B.S# AS SB
FROM   S AS A,S AS B
WHERE A.CITY = B.CITY
AND   A.S# < B.S#;
```

此例中很清晰地运用了显式范围变量。注意：这里引入了 SA、SB作为结果表的列名，因此不能在WHERE子句中使用。

6. 找出供应商的总数量

```
SELECT COUNT(*) AS N
FROM   S ;
```

这个查询结果是一个只有一列一行的表，其列名为 N，其值为5。SQL支持聚集操作：COUNT、SUM、AVG、MAX和MIN，但是用户要知道它特殊的方面：

- 一般地，关键字DISTINCT是可选的，位于变元之前，从而在聚集之前删去多余的重复行，如：SUM(DISTINDT QTY)。然而，对于MAX、MIN，DISTINCT是不影响结果的。
- 特别地，对于COUNT(*)操作，关键字DISTINCT不能在其中使用。它是统计表中非重复的所有的行。
- 在自变量列中，除掉COUNT(*)把NULL值当作一存在值进行统计外，其余的都在聚集之前把NULL值所在行忽略，而不管变元之前是否加了 DISTINCT关键字（参看第18章）。
- 若自变量碰巧是一空集，COUNT操作返回零值，其他的操作返回NULL值。SQL虽作了如此的定义，但它在逻辑上是不正确的（参看[3.3]）。

7. 找出零件P2的最多和最少供货数量

```
SELECT MAX ( SP,QTY ) AS MAXQ,MIN ( SP.QTY ) AS MINQ
FROM SP
WHERE SP.P# = 'P2';
```

注意到实际上这里FROM子句和WHERE子句都运用了聚集操作的变元部分。因此，从逻辑上讲，它们应该用括号括起来，正如本查询这样。这种非正统的语法方法对 SQL语言的结构、使用和正交性 ⊖ 产生非常消极的影响。例如：一个直接的后果就是聚集查询不能被嵌套，从而导致实现像“平均总零件数量”这样的查询非常麻烦。明确地讲，下述查询是不合法的：

```
SEELECT AVG ( SUM ( SP.QTY ) )                    --warning! Illegal!
FROM    SP;
```

上面的查询跟下面的有点相似：

```
SELECT AVG ( X )
FROM  ( SELECT SUM ( SP.QTY ) AS X
        FROM   SP
        GROUP  BY SP.S# ) APPOINTLESS;
```

⊖ 正交性就是独立性。只要一门语言能真正保持其独立性，那这门语言就是正交的。正交性是要求的，因为语言是越缺乏正交性，就会变得越复杂，其语言能力也就越弱。

此例中，给出了 ORDER BY 子句。后面有几个例子讲了嵌套子查询。值得注意的是 FROM中嵌套子查询的能力，它是在 SQL/92标准中才添上的，还没有广泛的实现。还有 AS *POINTLESS*的说明是无意义的，但是SQL语法规则还是要求的（参看附录 A）。

8. 对每一个供应的零件，找出其零件号及其供货总数量

```
SELECT SP.P#,SUM ( SP.QTY ) AS TOTQTY
FROM    SP
GROUP   BY SP.P#;
```

下面是关系代数表达式：

```
SUMMARIZE SP PER SP { P# } ADD SUM ( QTY ) AS TOTQTY
```

或者元组演算表达式：

```
( SPX.P#,SUM ( SPY WHERE SPY.P# = SPX.P#,QTY ) AS TOTQTY )
```

的近似表述。注意到，如果运用了 GROUP BY子句，则SELECT子句后面的表述就得单值分组(single-valued per group)。

下面是此查询的另一种表述（实际上更好）：

```
SELECT P.P#,( SELECT SUM ( SP.QTY )
              FROM   SP
              WHERE  SP.P# = P.P# ) AS TOTQTY
FROM P;
```

在SQL/92标准中增加了嵌套子查询来表示数值（如本例中 SELECT子句），并且对最初的版本作了很大的改进。此例中，其结果包括了根本没有供应的零件的行，这一点前面使用 GROUP BY子句的式子并没有（然而，不幸的是，这样的零件的 TOTQTY值是NULL值，而不是零）。

9. 找出多家供应商供应的零件号

```
SELECT SP.P#
FROM SP
GROUP BY SP.P#
HAVING COUNT ( SP.S# ) > 1;
```

HAVING 子句是将符合WHERE子句的几行分组；或者说，HAVING的使用是删除一部分分组，就像WHERE子句删除行一样。在HAVING子句中的表达式必须单值分组。

10. 找出供应零件P2的供应商的名字

```
SELECT DISTINCT S.SNAME
FROM    S
WHERE   S.S# IN
       (SELECT SP.S#
        FROM  SP
        WHERE SP.P# = 'P2' );
```

解释：此例运用了 WHERE子句的子查询。不严格地讲，一个子查询就是将 SELECT-FROM-WHERE-GROUP BY-HAVING表述嵌在另一个这样的查询里。正如本例所述的那样，在有些情况下使用子查询是要通过 IN条件表示一系列查询值。这一方式是先进行子查询再完成整个查询 —— 至少概念上是这样的。此例先返回供应零件 P2的供应商号，即{S1，S2，S3，S4}。上面的表达式等价于下面的更简单的表达式：

```
SELECT DISTINCTS S.SNAME
FROM S
WHERE S.S# IN ( 'S1','S2','S3','S4' )
```

值得提出的是本题的查询也可以用连接来完成，即如下：

```
SELECT DISTINCTS S.SNAME
FROM S,SP
WHERE S.S# = SP.S#
AND    SP.P# = 'P2';
```

11. 找出供应至少一个红色零件的供应商名

```
SELECT DISTINCT S.SNAME
FROM S
WHERE S.S# IN
      ( SELECT SP.S#
        FROM    SP
        WHERE SP.P# IN
              ( SELECT P.P#
                FROM P
                WHERE P.COLOR = 'Red' ) );
```

子查询可以嵌套到任何深度。练习：给出此查询的连接查询形式。

12. 找出状态小于当前S表中最大状态值的供应商的供应商号

```
SELECT S.S#
FROM S
WHERE S.STATUS <
     ( SELECT MAX ( S.STATUS )
       FROM S );
```

本例隐含了两个不同的范围变量，但它们都用“ S ”表示，且都定义在S表上。

13. 找出供应零件P2的供应商名

注意：此例与例10是一样的。这里为了引入SQL的另一个特征，我们给出了另一种语句：

```
SELECT DISTINCT S.SNAME
FROM S
WHERE EXISTS
      ( SELECT *
        FROM   SP
        WHERE  SP.S# = S.S#
        AND    SP.P# ='P2');
```

解释：SQL表达式“ EXISTS （SELECT...FROM...）”取真值，当且仅当“ SELECT...FROM... ”取非空值。或者说，SQL中的EXISTS操作符相应于元组演算中的存在量词（参看 [18.6]）。注意：在这个特殊的作为相关子查询的例子中， SQL涉及子查询，因此它包含了一范围变量的引用，即：隐式范围变量 S，它在外查询中定义。在例 8中我们曾讲到这是一个“ 更完美的式子 ”，这也是一个相关子查询。

14. 找出没有供应零件P2的供应商的供应商名

```
SELECT DISTINCTS S.SNAME
FROM S
WHERE NOT EXISTS
```

```
        ( SELECT *
          FROM SP
          WHERE SP.S# = S.S#
          AND   SP.P# ='P2' );
```

或者：

```
SELECT DISTINCT S.SNAME
FROM S
WHERE S.S# NOT IN
      ( SELECT SP S#
        FROM SP
        WHERE SP.P# = 'P2' );
```

15. **找出供应所有零件的供应商的供应商名**

```
SELECT DISTINCT S.SNAME
FROM S
WHERE NOT EXISTS
     ( SELECT *
       FROM P
       WHERE NOT EXISTS
            ( SELECT *
              FROM SP
              WHERE SP.S# = S.S#
              AND   SP.P# = P.P# ) );
```

SQL不直接地支持全称量词FORALL，因此“FORALL”全称量词就不得不像本例这样，利用存在量词和双重否定来表述。

值得注意的是，尽管上述表示乍一看令人觉得烦琐，但它还是能很容易地被熟悉关系演算的用户所构建（参看[7.4]）。如果上述表示还是很烦琐，那可以用几个“工作区”的方法来代替，这种方法可以避免否定量词。例如，本例可写成：

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  ( SELECT COUNT( SP.P# )
         FROM   SP
         WHERE  SP.S# = S.S# =
       ( SELECT COUNT( P.P# )
         FROM   P );
```

（“取供应商名，它们供应的零件的数量等于所有零件的数量”）。然而，要注意：

- 首先，后面的式子依赖于每一个供货的零件号等于某一现存零件的号码，这一点在有NOT EXISTS的量词的式子中不需要。或者说仅仅当实现了一定的完整性约束条件（参看下一章）时，这两个式子才等价，并且第二个式子才正确。
- 其次，第二个式子使用两个COUNT的函数的技术（子查询依赖于两个等号的成立）在SQL最初版中没有，在SQL/92标准中增加了这一点。但它在许多笛卡尔积中不支持。
- 实际上，我们最可能做的是比较两个表（参看第6章关系比较）。因此，上述查询可以如下表示：

```
SELECT DISTINCT S.SNAME
FROM S
```

```
WHERE ( SELECT SP.P#
        FROM   SP
        WHERE  SP.S# = S.S# ) =
      ( SELECT P.P#
        FROM   P );
```

然而，SQL标准并没有直接支持表的比较，所以我们不得不求助于比较表的基本部分（依赖于外部知识去确保只要表的基本部分是一样的，那么这两个表就是一样的）。请看本章后面的练习。

16. 找出那些重量超过16磅或者由S2供应的或者两者都具备的零件号

```
SELECT P.P#
FROM P
WHERE P.WEIGHT > 16.0
UNION
SELECT SP.P#
FROM   SP
WHERE  SP.S# = 'S2';
```

使用不加修饰的UNION、INTERSECT或者EXCEPT（SQL中EXCEPT即是MINUS），结果中多余的重复行会自动地被删去。同时，SQL也提供了它们的另一种形式，即带修饰的UNION ALL、INTERSECT ALL和EXCEPT ALL。它们使用时，对于重复的行，只要有都会保留。这里就不再举例了。

这一节举了不少例子，实际上SQL的例子是相当长的。但是，这只讨论了SQL大量的特征的一小部分。实际上，SQL是非常冗杂的语言（参看[4.8]），它为完成同样的功能，提供了大量的不同的实现方法。由于书的版面有限，我们就不再讨论所有可能的实现方法，甚至对本节讨论的数量有限的例子也是如此。

7.8 小结

我们已经简单地介绍了关系代数的替代——关系演算。从表面上看，这两者是很不相同的——演算是描述性的，而代数是说明性的。但是，从更深一层次讲，它们是一样的。因为任何一种演算表示都可以转换为代数表示，反之亦然。

演算有两种，即元组演算和域演算。它们之间最重要的不同是：元组演算的范围变量定义在关系上，而域演算的范围变量是定义在域上。

元组演算的表达包括一个原型元组、一个可选的包含布尔表达式或合式公式的WHERE子句。此合式公式可允许包含量词EXISTS和FORALL、自由的和约束的范围变量、布尔操作符（AND、OR、NOT等），等等。每一个在合式公式中用到的范围变量必须在原型元组中用到。注意：本章中我们没有明确地讨论这一点，但是演算的表示必须要求明确地满足代数所表示的同一目的（参看第6章6.6节）。

我们通过例子说明了Codd的简约算法如何将任意的演算表示转化为等价的代数表示，这样就为演算策略的实现提供了方便。我们再一次讨论了关系完备，并且就此简单地讨论了语言的完备性。

我们也在元组演算中讨论了计算功能，它类似于代数中EXTEND和SUMMARIZE提供的功能。我们简单地介绍了域演算，并认为它具备关系完备，尽管没有证明。这样，元组演算、

域演算和关系代数这三者是相互等价的。

最后，我们对SQL的相关特征作了概述。SQL混合了代数和（元组）演算。例如：它包括直接对JOIN和UNION代数操作符的支持，但是它也使用演算的范围变量和存在量词。SQL查询由表表达式组成。通常，一个操作由一个选择表达式组成，但也支持多种形式的JOIN表达式，且以各种方式通过使用UNION、INTERSECT和EXCEPT操作符把JOIN和SELECT表达式连接起来。我们也谈到了ORDER BY子句，它是对从一表表达式（任何种类的）导出的表进行强制性排序。

特别地，对于选择表达式，我们再说几点：

- 基本的SELECT子句，包括DISTINCT关键字、标量表达式、结果列名说明或“SELECT *”；
- FROM子句，包括范围变量的使用；
- WHERE子句，包括EXISTS量词的使用；
- GROUP BY和HAVING子句，包括聚集操作COUNT、SUM、AVG、MAX和MIN；
- 在SELECT、FROM和WHERE子句中使用子查询。

我们还给出了概念性的计算算法（conceptual evaluation algorithm），即选择表达式形式化定义的轮廓。

练习

7.1 设$p(x)$和q是任意的合式公式，其中，x为自由变量，分别为出现和不出现。下面哪一个式子是正确的？（符号“ ”表示“蕴含”，符号“≡”表示“恒等于”。注意：“A B”和“B A”在一起使用时，就表示“A≡B”）

a) `EXISTS x ( q ) ≡ q`

b) `FORALL x ( q ) ≡ q`

c) `EXISTS x ( p(x) AND q ) ≡ EXISTS x ( p(x) ) AND q`

d) `FORALL x ( p(x) AND q ) ≡ FORALL x ( p(x) ) AND q`

e) `FORALL x ( p(x) ) ⇒ EXISTS x ( p(x) )`

7.2 设$p(x,y)$是任意的合式公式，x、y为自由变量。下面哪一种说法是正确的？

a) `EXISTS x EXISTS y ( p(x,y) ) ≡ EXISTS y EXISTS x ( p(x,y) )`

b) `FORALL x FORALL y ( p(x,y) ) ≡ FORALL y FORALL x ( p(x,y) )`

c) `FORALL x ( p(x,y) ) ≡ NOT EXISTS x ( NOT p(x,y) )`

d) `EXISTS x ( p(x,y) ) ≡ NOT FORALL x ( NOT p(x,y) )`

e) `EXISTS x FORALL y ( p(x,y) ) ≡ FORALL y EXISTS x ( p(x,y) )`

f) `EXISTS y FORALL x ( p(x,y) ) ⇒ FORALL x EXISTS y ( p(x,y) )`

7.3 设$p(x)$和$q(y)$是任意的合式公式，x、y分别为自由变量。下面哪一种说法是正确的？

a)
```
EXISTS x ( p(x) ) AND EXISTS y ( q(y) ) ≡
EXISTS x EXISTS y ( p(x) AND q(y) )
```

b)
```
EXISTS x ( IF p(x) THEN q(x) END IF ) ≡
IF FORALL x ( p(x) ) THEN EXISTS x ( q(x) ) END IF
```

7.4 看7.3节例8，“找出至少供应S2供应的零件的供应商的供应商号”，下面是一个可能的元组演算表示式：

```
SX.S# WHERE FORALL SPY ( IF SPY.S# = S# ( 'S2' ) THEN
                         EXISTS SPZ ( SPZ.S# = SX.S# AND
                                      SPZ.P# = SPY.P# )
                       END IF )
```

(这里SPZ是另一个定义在供货表上的范围变量)。如果当前S2什么零件都不供应，则本查询返回什么？如果本查询中所有的SX都替换为SPX，它又有什么不同？

7.5 下面是一个对供应商-零件-工程数据库的查询（这是通常考虑范围变量时的习惯应用）：

```
( PX.PNAME, PX.CITY ) WHERE FORALL SX FORALL JX EXISTS SPJX
                            ( SX.CITY = 'London' AND
                              JX.CITY = 'Paris' AND
                              SPJX.S# = SX.S# AND
                              SPJX.P# = PX.P# AND
                              SPJX.J# = JX.J# AND
                              SPJX.QTY < QTY ( 500 ) )
```

a) 把这些查询转化为自然语言。

b) 对这个查询运行DBMS，并“执行”Codd的简化算法，这样你能看出一些改进吗？

7.6 对于查询“取出三个最重的零件”，用元组演算表达式表示。

7.7 看第4章中的材料清单的关系变量PART_STRUCTURE(在练习4.6的解答中给出SQL定义，在图4-6中给出了它的示例关系)。对于知名的零件探寻（part explosion）查询“找出组成零件P1的任何层次上的零件的零件号”——此查询结果，依据PART_BILL，一定是从PART_STRUCTURE导出的一个关系。此查询最初不能用关系代数或关系演算独立地完成。或者说，PART_BILL是一个可导出关系，但它不能由单独的原始关系代数或关系演算导出。请问这是为什么？

7.8 假设供应商关系变量S可被另一些关系变量（如：LS, PS, AS, ...）所代替（每一个代表在不同城市的供应商，如：LS仅仅表示在伦敦的供应商的元组）；假设我们不知道有哪些供应商所在城市存在，并因此不知道有多少这样的关系变量。对于查询“此库中有供应商S1吗？”，能用演算或代数表示吗？请给出你的答案。

7.9 叙述SQL的关系完备性。

7.10 SQL中有与关系操作符EXTEND和SUMMARIZE等价的操作吗？

7.11 SQL中有与比较操作等价的操作吗？

7.12 对于查询“找出供应零件P2的供应商名”，请用尽可能多的SQL表示方法进行表示。

查询练习

下面的练习要用到供应商-零件-工程数据库（参看第4章练习中的图4-5和第5章练习5.4的解答部分)。要求对下面每一个查询给出一个表达式（为了一个有兴趣的变换，你可能试图先看解答并用自然语言来解释这个表达式)。

7.13 写出习题6.13~6.50的元组演算表达式。

7.14 写出习题6.13~6.50的域演算表达式。

7.15 写出习题6.13~6.50的SQL表达式。

参考文献和简介

参看[4.7]及下列书目，它们对SQL的扩充作了描述，来解决传递封闭（transitive closure）及相似的问题。SQL3中有与这非常相似的扩充。附录 B简单地叙说了这一点。参看第 23章，该章把这一特性放在关系演算中讨论。

7.1 E. F. Codd: “A Data Base Sublanguage Founded on the Relational Calculus,” Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control, San Diego, Calif. (November 1971).

7.2 C. J. Date: “A Note on the Relational Calculus,” *ACM SIGMOD Record 18*, No. 4 (December 1989). Republished as “An Anomaly in Codd’s Reduction Algorithm” in C. J. Date and Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

7.3 C. J. Date: “Why Quantifier Order Is Important, ” in C. J. Date and Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass: Addison-Wesley(1992).

7.4 C. J. Date: “Relational Calculus as an Aid to Effective Query Formulation,” in C. J. Date and Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

目前，市场上每一个有关关系的产品都支持 SQL，而不是关系代数或关系演算。而这篇论文提倡并解释了通过关系演算作为中介去构建复杂的 SQL查询。

7.5 G. D. Held, M. R. Stonebraker, and E. Wong: “INGRES —A Relational Data Base System, ” Proc. NCC *44*, Anaheim, Calif, Montvale, N. J. : AFIPS Press (May 1975).

20世纪70年代中后期，发展了两个重要的关系原型。这就是 IBM的System R和加州大学伯克利分校的Ingres（或写作INGRES）。在研究领域，它们非常有影响，并相继导致了两个商业系统，即 System R环境下的DB2和Ingres环境下的商业 Ingres产品。注意：有时，Ingres原型叫做“大学 Ingres”(参看[7.11])，从而与“商业 Ingres”区分开来。[1.5]对这一商业版本作了指导性的概述。

起初，Ingres并不是一个SQL系统，它所支持的语言叫做 QUEL（Query Language）。这一语言在许多方面技术上都超过了 SQL。事实上，现在许多数据库的研究都建立在此语言上，并且QUEL中使用的例子还出现在许多论文里。这篇论文首先介绍了 Ingres原型系统，接着对QUEL作了初步解释。还可参看[7.10~7.12]。

7.6 J. L. Kuhns: “Answering Questions by Computer: A Logical Study, ” Report RM-5428-PR, Rand Corp., Santa Monica, Calif. (1967).

7.7 M. Lacroix and A. Pirotte: “Domain-Oriented Relational Languages ” Proc. 3rd Int. Conf. on Very Large Data Bases, Tokyo, Japan (October 1977).

7.8 T. H. Merrett: “The Extended Relational Algebra, A Basis for Query Languages,” in B. Shneidernian (ed.), *Databases: Improving Usability, and Responsiveness*. New York, N.Y.: Academic Press (1978).

本书建议在关系代数中引入量词 —— 不仅仅是关系演算的存在量词和全称量词，而且包括一般的像“……的数量是……”和“……的部分是……”的量词。如：“至少……的三个”、“不超过……的一半”、“……的奇数个数”，等等。

7.9 M. Negri, G. Pelagatti, and L. Sbattella: " Formal Semantics of SQL Queries," *ACM TODS 16*, No. 3 (September 1991).

引用如下摘要："SQL查询的语法被一系列规则所定义，这些规则决定了把其翻译成一种范式模型。这一模型叫做扩展三值谓词演算（ E3VPC:Extended Three-Valued Predicate Calculus），它在很大程度上基于有名的数学概念。这里也给出了将一个一般的E3VPC表达式转化为一个规范形式的规则。而且这里也解决了诸如 SQL查询的等价分析问题"。然而，注意：这些 SQL所考虑的语言仅是它的最初版，即 SQL/86，而不是SQL/92。

7.10 Michael Stonebraker (ed.): *The INGRES Papers: The Anatomy of a Relational Database Management System*. Reading, Mass.: Addison-Wesley (1986).

这是"大学Ingres"项目的一些重要的论文集，由最初 Ingres的设计者们编辑、注释（下面的[7.11~7.12]都包含在此论文集里）。据作者所知，这是一本目前唯一全面具体地描述了关系DBMS的设计和实现的技术、并适合各种水平的学生阅读的书籍。

7.11 Michael Stonebraker, Eugene Wong, Peter Kreps, and Gerald Held: "The Design and Implementation of INGRES." *ACM TODS 1*, No. 3 (September 1976). Republished in reference [7.10].

本书对"大学Ingres"原型作了具体的描述。

7.12 Michael Stonebraker: "Retrospection on a Data Base System," *ACM TODS 5*, No. 2 (June 1980). Republished in reference [7.10].

本书介绍了Ingres原型项目的历史（到1979年1月）。重点是介绍其失败和教训，而不是成功经验。

7.13 Jeffrey D. Ullman: *Principles of Database and Knowledge-Base Systems: Volume I*. Rockville, Md.: Computer Science Press (1988).

Ullman的这本书比我们这本书更加规范地处理了关系演算及其相关的方面。特别地，它讨论了演算表达式安全性的概念。如果我们对这一演算稍作改动，会变得更有意义。因为这一演算中的范围变量并不是由独立的语句所定义，而是通过 WHERE子句中明确地表述来约束其范围。在这种演算表示中，像查询"找出在伦敦的供应商"，看起来似乎如下表示

```
SX WHERE SX∈ S AND SX.CITY = "London"
```

很明显，这种演算带来的一个问题（不仅仅是只一个问题）是它允许如下的查询：

```
SX WHERE NOT ( SX∈ S )
```

这样的表达式是不安全的，因为它不能返回有限的结果集（非 S元组的所有项是无限的）。于是，必须有一定的规则去强制保证那些合法的表达式是安全的。这些规则在 Ullman的这本书里作了描述（包括元组演算和域演算）。我们注意到，Codd的最初的演算并没有包括这些规则。

7.14 Moshé M. Zloof: "Query-By-Example, " Proc. NCC 44, Anaheim, Calif. (May 1975). Montvale, N. J.: AFIPS Press (1977).

关系语言Query-By-Example(QBE)具备元组演算和域演算的功能（后者强调得更多）。它的语法基于表格查询的思想，简单、直观，且具有吸引力。例如，对于一个相

当复杂的查询“找出至少供应一个S2供应的零件的供应商的名字”，它的可能的QBE查询表达式是：

S	S#	SNAME
	_SX	P._NX

SP	S#	P#
	_SX	_PX

SP	S#	P#
	S2	_PX

注解：用户要求在屏幕上显示三个表格框架：一个用于供应商表，另两个用于供货表，如上表所示作出查询。实体查询以下划线引导示例开始（即域演算范围变量），其他的查询都是文字值。用户要求系统那样显示（“P.”）供应商名值（_NX）。如果供应商是_SX，则_SX供应某个零件_PX，且SX又由S2供应。注意：存在量词隐含在查询过程中（正像它们在QUEL中的那样）——这就是这一语法简单、直观的另一原因。

下面是另外一个例子：查询位于同一城市的两个供应商的供应商号。

S	S#	CITY
	_SX	_CZ
	_SY	_CZ

P.	_SX	_SY

不幸的是，QBE不具备关系完备性。更具体地讲，就是它不完全支持“不存在”量词（NOT EXISTS）。所以，某些查询就不能用QBE表示。如：查询供应所有零件的供应商名（实际上，最初，QBE是“支持”量词“NOT EXISTS”的，至少是隐含支持的。但是这种结构总是有点麻烦）。基本的问题是，没有方法去指明所有隐含的量词的运用顺序，而且不幸的是这一顺序是很重要的（参看[7.3]，或者习题7.2的解答）。所以有些QBE查询表达式是模棱两可的（参看[7.3]）。

Zloof是QBE的最初的发明者及设计者。这篇文章是Zloof在这方面所有论文的第一篇。

部分练习答案

7.1 a) 合法；b) 合法；c) 合法；d) 合法；e) 不合法。注意：e) 不合法是因为运用在空集上的FORALL量词产生真值，而运用在空集上的量词EXISTS产生假值。所以，例如“所有紫色零件超过100磅”就取真值（即这是一个真的命题），但这并不意味着所有紫色零件都存在。

我们谈到用合法的等价和蕴含作为一系列演算表达式变化规则的基础，就像第6章和第17章讨论的代数变化规则一样。习题7.2和7.3中也要注意这一点。

7.2 a) 合法；b) 合法；c) 合法（这一题在本章里已讨论过）；d) 合法（每个量词都能按照另一个来定义）；e) 不合法；f) 合法。注意到，使用LIKE量词（正像a和b所示的那样），其结果可以以任何顺序出现而不改变其意义；而使用量词UNLIKE，这顺序就很重要。现举例说明后一点。设*x*和*y*为定义在整数上的范围变量，设*p*为合式公式“*y*>*x*”。这合式公式就很清楚地表示如下：

```
FORALL x EXISTS y( y>x )
```

（“对所有的整数*x*，存在一个比它大的整数*y*”），结果为真。而对于合式公式：

```
EXISTS y FORALL x ( y>x )
```

（“存在一整数*x*，比任何整数*y*大”），结果为假。从这可以看出，交换UNLIKE量词就改变意义。所以，在基于演算的语言里，交换WHERE子句中的UNLIKE量词，就改变查询的意义（参看[7.3]）。

7.3 a) 合法；b) 合法。

7.4 如果S2当前没有供应零件，则原查询返回当前 S表中所有的供应商号（尤其当 S2出现在S表中而不出现在 SP表中时）。如果我们把所有的 SX都换成SPX，它返回所有当前出现在SP表中的供应商号。这两个查询有如下的不同点：第一个是查询至少供应由 S2所供应的零件（正如所要求的那样）；而第二个是查询至少供应一个 S2供应的零件和至少供应 S2所供应的零件的供应商的供应商号。

7.5 a) 通过伦敦的每一个供应商以不少于 500的数量供应给巴黎的每一个工程项目的零件的出产地和零件名；b) 此查询结果为空值。

7.6 这一题是非常难的 —— 尤其当我们考虑到零件重量不唯一时（如果其重量唯一，我们可以这样表示这个查询：查询所有重于它的零件不超过三个的零件）。事实上，本题难得我们无法仅仅用演算去解决它。关系完备仅仅是表达能力的一个基本的、而不一定是有效的测度很好地说明了这一点（下面的两个习题也说明了这一点）。参看[6.4]，对这一类型的查询作了扩充讨论。

7.7 设PSA、PSB、PSC、...是定义在关系变量PART_STRUCTURE（当前值）上的范围变量。并设给定零件P1，则：

a) 查询“找出所有组成零件 P1的第一层次的零件的零件号”的演算表达式是：

```
PSA.MINOR_P# WHERE PSA.MAJOR_P# = P# ( 'P1' )
```

b) 查询“找出所有组成零件 P2的第二层次的零件的零件号”的演算表达式是：

```
PSB.MINOR_P# WHERE EXISTS PSA
                 ( PSA.MAJOR_P# = P# ( 'P1' ) AND
                   PSB.MAJOR_P# = PSA.MINOR_P# )
```

c) 查询“找出所有组成零件 P2的第二层次的零件的零件号”的演算表达式是：

```
PSC.MINOR_P# WHERE EXISTS PSA EXISTS PSB
                 ( PSA.MAJOR_P# = P# ( 'P1' ) AND
                   PSB.MAJOR_P# = PSA.MINOR_P# AND
                   PSC.MAJOR_P# = PSB.MINOR_P# )
```

查询“找出所有组成零件 P2的所有层次的零件的零件号”的演算表达式是：

```
PSn.MINOR_P# WHERE EXISTS PSA EXISTS PSB ... EXISTS PS(n-1)
                 ( PSA.MAJOR_P# = P# ( 'P1' ) AND
                   PSB.MAJOR_P# = PSA.MINOR_P# AND
                   PSC.MAJOR_P# = PSB.MINOR_P# AND
                      ....................      AND
                   PSn.MAJOR_P# = PS(n-1).MINOR_P# )
```

所有这些结果关系a)、b)、c) ...需要“并”起来构造PART_BILL结果。

然而，问题是：当 *n*不知道时，就无法写出 *n*个表达式。实际上，在仅是关系完备的语言（它的功能没有最初的演算或代数强）里，是不能用一个单独的表达式来解决像零件探寻这样多级的问题。因此，我们需要对最初的演算或代数作扩充。第 6章简单讨论过的TCLOSE操作解决了这个问题（不仅仅是针对零件）。本书没有介绍更多的具体的细节。

注意：尽管这个问题通常称作“材料清单”或“零件探寻”，但实际上它比这可能建议的名字有着更广泛的用途。实际上，这种有着“零件包含零件”结构的关系有着非常广泛的应用。其他类似的例子还有：管理层次、家族树、授权图、通信网、软件模块调用结构、供货网，等等。

7.8 此查询不能用代数或演算表示。如果要能用演算表示它，最基本的是要能解决下列问题：

存在这样的一个关系r及r中的一个元组t，使得t.S# = S#('S1') 吗？

或者说，我们需要对关系进行量化，而不是对元组进行量化。因此我们需要一种新的范围变量，它是定义在关系上而不是在元组上。所以，此查询就不能用我们所讲的关系演算表示。

还要注意的是，我们所讲的查询是“是/非”查询（所期望的结果基本上是真值）。所以你可能认为那些查询不能用代数或演算处理，是因为这样的代数和演算是基于关系值的，而不是真值的。然而，“是/非”查询，只要处理得当，就能用演算和代数解决！问题的关键在于要承认“是/非”（等价于“真/假”）查询是基于关系的表述。这里讲的关系分别为TABLE_DEE和TABLE_DUM（参看[5.5]）。

7.9 为了表明SQL是关系完备的，我们必须首先表明存在5个原始的（代数）操作符——选择、投影、积（product）、并和差——的SQL表达式，并且其操作数依次是任意SQL的表达式。从一开始我们就注意到，SQL实际上支持关系代数操作RENAME。在SQL/92标准中，SELECT子句中的选项“AS < *column name*>”就很好地解决了这个问题。因此我们确信所有基本表必须有合适的列名，尤其积、并和差的操作数必须满足有关列命名的关系代数的需要（我们所讲的）。而且，如果这些操作数的列命名的要求能满足，那么SQL列命名的继承规则实际上就和第6章讲的关系继承规则一致。

5个原始操作符的相应SQL表达式如下：

代数	*SQL*
`A WHERE p`	`SELECT * FROM A WHERE p`
`A {x,y,...,z}`	`SELECT DISTINCT x,y,...,z FROM A`
`A TIMES B`	`A CROSS JOIN B`
`A UNION B`	`SELECT * FROM A UNION SELECT * FROM B`
`A MINUS B`	`SELECT * FROM A EXCEPT SELECT * FROM B`

参看附录A给出的SQL表表达式，我们可以看到，上面SQL表达式中的每一个A和B实际上都是一个表引用。我们也看到，如果我们把上面的5个SQL表达式用括号括起来，那么结果依然还是一个表引用[⊖]。从这可以看出，SQL的确是关系完备的。

注意：实际上，上述所讲的还是不完善——SQL对于不存在的列根本就无法投影，因为它并不支持空的SELECT子句。所以，它不支持TABLE_DEE和TABLE_DUM（参看[5.5]）

7.10 SQL支持EXTEND，但不支持SUMMARIZE（至少并不直接支持），下面的用了EXTEND的关系代数表达式

```
EXTEND A ADD exp AS Z
```

用SQL表示就是：

```
SELECT A.*, exp' AS Z
FROM   ( A ) AS A
```

SELECT子句的*exp*'对应于SQL中EXTEND子句中的*exp*。FROM子句括号中的A（对应于EXTEND子句中的操作数A）可以是任意复杂的表引用；FROM子句中另一个A是范围变量名。

对于用了SUMMARIZE的关系代数表达式：

⊖ 实际上，SQL要求这样的一个表引用包括一个无意义的范围变量的定义（参看附录A）。

```
SUMMARIZE A PER B ...
```

其基本问题是它产生的结果的基数等于*B*的基数，而等价的SQL表达式：

```
SELECT ...
FROM   A
GROUP  BY C ;
```

产生的结果的基数等于*A*在*C*上的投影的基数。

7.11 SQL不能直接支持关系比较，但这种比较可被模拟出来，尽管这一方法比较笨拙。例如，比较：

```
A = B
```

（这里*A*和*B*都是关系变量）能用下述SQL表达式模拟出来：

```
NOT EXISTS ( SELECT * FROM A
             WHERE NOT EXISTS ( SELECT * FROM B
                                WHERE A-row = B-row ) )
```

（这里*A_row*和*B_row*都是行构造器<*row constructor*>（参看附录*A*)分别表示整个的*A*行和*B*行）

7.12 下面是一些查询表达式。注意这些表达式不是很详尽的，但也不是很简单的！

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  S.S# IN
     ( SELECT SP.S#
       FROM   SP
       WHERE  SP.P# = 'P2' ) ;

SELECT DISTINCT T.SNAME
FROM ( S NATURAL JOIN SP ) AS T
WHERE  T.P# = 'P2' ;

SELECT DISTINCT T.SNAME
FROM ( S JOIN SP ON S.S# = SP.P# AND SP.P# = 'P2' ) AS T ;

SELECT DISTINCT T.SNAME
FROM ( S JOIN SP USING S# ) AS T
WHERE  T.P# = 'P2' ;

SELECT DISTINCT S.SNAME
FROM   S
WHERE  EXISTS
     ( SELECT *
       FROM   SP
       WHERE  SP.S# = S.S#
       AND    SP.P# = 'P2' ) ;

SELECT DISTINCT S.SNAME
FROM   S, SP
WHERE  S.S# = SP.S#
AND    SP.P# = 'P2' ;

SELECT DISTINCT S.SNAME
FROM   S
WHERE  0 <
     ( SELECT COUNT(*)
       FROM   SP
       WHERE  SP.S# = S.S#
       AND    SP.P# = 'P2' ) ;

SELECT DISTINCT S.SNAME
FROM   S
WHERE  'P2' IN
     ( SELECT SP.P#
       FROM   SP
       WHERE  SP.S# = S.S# ) ;

SELECT S.SNAME
FROM   S, SP
WHERE  S.S# = SP.S#
AND    SP.P# = 'P2'
GROUP  BY S.SNAME ;
```

还有：这些表达式里隐含着什么问题呢？

7.13 **我们用7.13.*n*的形式列出了下述习题，其中的6.*n*表示第6章中的题目。我们假设SX、SY、PX、PY、JX、JY、SPJX、SPJY（等等）分别表示定义在供应商表、零件表、工程表和供货表上的范围变量。这里没有列出这些范围变量的定义。**

7.13.13
```
JX
```

7.13.14
```
JX WHERE JX.CITY = 'London'
```

7.13.15
```
SPJX.S# WHERE SPJX.J# = J# ( 'J1' )
```

7.13.16
```
SPJX WHERE SPJX.QTY ≥ QTY ( 300 ) AND
           SPJX.QTY ≤ QTY ( 750 )
```

7.13.17
```
( PX.COLOR, PX.CITY )
```

7.13.18
```
( SX.S#, PX.P#, JX.J# ) WHERE SX.CITY = PX.CITY
                            AND   PX.CITY = JX.CITY
                            AND   JX.CITY = SX.CITY
```

7.13.19
```
( SX.S#, PX.P#, JX.J# ) WHERE SX.CITY ≠ PX.CITY
                            OR    PX.CITY ≠ JX.CITY
                            OR    JX.CITY ≠ SX.CITY
```

7.13.20
```
( SX.S#, PX.P#, JX.J# ) WHERE SX.CITY ≠ PX.CITY
                            AND   PX.CITY ≠ JX.CITY
                            AND   JX.CITY ≠ SX.CITY
```

7.13.21
```
SPJX.P# WHERE EXISTS SX ( SX.S# = SPJX.S# AND
                          SX.CITY = 'London' )
```

7.13.22
```
SPJX.P# WHERE EXISTS SX EXISTS JX
      ( SX.S# = SPJX.S# AND SX.CITY = 'London' AND
        JX.J# = SPJX.J# AND JX.CITY = 'London' )
```

7.13.23
```
( SX.CITY AS SCITY, JX.CITY AS JCITY )
WHERE EXISTS SPJX ( SPJX.S# = SX.S# AND SPJX.J# = JX.J# )
```

7.13.24
```
SPJX.P# WHERE EXISTS SX EXISTS JX
              ( SX.CITY = JX.CITY AND
                SPJX.S# = SX.S# AND
                SPJX.J# = JX.J# )
```

7.13.25
```
SPJX.J# WHERE EXISTS SX EXISTS JX
              ( SX.CITY ≠ JX.CITY AND
                SPJX.S# = SX.S# AND
                SPJX.J# = JX.J# )
```

7.13.26
```
( SPJX.P# AS XP#, SPJY.P# AS YP# )
WHERE SPJX.S# = SPJY.S# AND SPJX.P# < SPJY.P#
```

7.13.27
```
COUNT ( SPJX.J# WHERE SPJX.S# = S# ( 'S1' ) ) AS N
```

7.13.28
```
SUM ( SPJX WHERE SPJX.S# = S# ( 'S1' )
               AND SPJX.P# = P# ( 'P1' ), QTY ) AS Q
```

注意：下面的解答是不正确的，为什么不正确？

```
SUM ( SPJX.QTY WHERE SPJX.S# = S# ( 'S1' )
                 AND SPJX.P# = P# ( 'P1' ) ) AS Q
```

答案：因为在计算总和时，重复的QTY并没有被删除。

7.13.29
```
( SPJX.P#, SPJX.J#,
  SUM ( SPJY WHERE SPJY.P# = SPJX.P#
                 AND   SPJY.J# = SPJX.J#, QTY ) AS Q )
```

7.13.30
```
SPJX.P# WHERE
```

```
        AVG ( SPJY WHERE SPJY.P# = SPJX.P#
                    AND   SPJY.J# = SPJX.J#, QTY ) > QTY ( 350 )
```

7.13.31
```
JX.JNAME WHERE EXISTS SPJX ( SPJX.J# = JX.J# AND
                             SPJX.S# = S# ( 'S1' ) )
```

7.13.32
```
PX.COLOR WHERE EXISTS SPJX ( SPJX.P# = PX.P# AND
                             SPJX.S# = S# ( 'S1' ) )
```

7.13.33
```
SPJX.P# WHERE EXISTS JX ( JX.CITY = 'London' AND
                          JX.J# = SPJX.J# )
```

7.13.34
```
SPJX.J# WHERE EXISTS SPJY ( SPJX.P# = SPJY.P# AND
                            SPJY.S# = S# ( 'S1' ) )
```

7.13.35
```
SPJX.S# WHERE EXISTS SPJY EXISTS SPJZ EXISTS PX
              ( SPJX.P# = SPJY.P# AND
                SPJY.S# = SPJZ.S# AND
                SPJZ.P# = PX.P# AND
                PX.COLOR = COLOR ( 'Red' ) )
```

7.13.36
```
SX.S# WHERE EXISTS SY ( SY.S# = S# ( 'S1' ) AND
                        SX.STATUS < SY.STATUS )
```

7.13.37
```
JX.J# WHERE FORALL JY ( JY.CITY ≥ JX.CITY )
```

或
```
JX.J# WHERE JX.CITY = MIN ( JY.CITY )
```

7.13.38
```
SPJX.J# WHERE SPJX.P# = P# ( 'P1' ) AND
              AVG ( SPJY WHERE SPJY.P# = P# ( 'P1' )
                         AND   SPJY.J# = SPJX.J#, QTY ) >
              MAX ( SPJZ.QTY WHERE SPJZ.J# = J# ( 'J1' ) )
```

7.13.39
```
SPJX.S# WHERE SPJX.P# = P# ( 'P1' )
              AND   SPJX.QTY >
                    AVG ( SPJY
                          WHERE SPJY.P# = P# ( 'P1' )
                          AND   SPJY.J# = SPJX.J#, QTY )
```

7.13.40
```
JX.J# WHERE NOT EXISTS SPJX EXISTS SX EXISTS PX
              ( SX.CITY = 'London' AND
                PX.COLOR = COLOR ( 'Red' ) AND
                SPJX.S# = SX.S# AND
                SPJX.P# = PX.P# AND
                SPJX.J# = JX.J# )
```

7.13.41
```
JX.J# WHERE FORALL SPJY ( IF SPJY.J# = JX.J#
                          THEN SPJY.S# = S# ( 'S1' )
                          END IF )
```

7.13.42
```
PX.P# WHERE FORALL JX
          ( IF JX.CITY = 'London' THEN
            EXISTS SPJY ( SPJY.P# = PX.P# AND
                          SPJY.J# = JX.J# )
            END IF )
```

7.13.43
```
SX.S# WHERE EXISTS PX FORALL JX EXISTS SPJY
              ( SPJY.S# = SX.S# AND
                SPJY.P# = PX.P# AND
                SPJY.J# = JX.J# )
```

7.13.44
```
JX.J# WHERE FORALL SPJY ( IF SPJY.S# = S# ( 'S1' ) THEN
                          EXISTS SPJZ
                          ( SPJZ.J# = JX.J# AND
                            SPJZ.P# = SPJY.P# )
                            END IF )
```

7.13.45
```
RANGEVAR VX RANGES OVER
         ( SX.CITY ), ( PX.CITY ), ( JX.CITY ) ;

        VX.CITY
```

7.13.46
```
SPJX.P# WHERE EXISTS SX ( SX.S# = SPJX.S# AND
                          SX.CITY = 'London' )
          OR    EXISTS JX ( JX.J# = SPJX.J# AND
                          JX.CITY = 'London' )
```

7.13.47
```
( SX.S#, PX.P# )
WHERE NOT EXISTS SPJX ( SPJX.S# = SX.S# AND
                        SPJX.P# = PX.P# )
```

7.13.48
```
( SX.S# AS XS#, SY.S# AS YS# )
WHERE FORALL PZ
  ( ( IF   EXISTS SPJX ( SPJX.S# = SX.S# AND
                         SPJX.P# = PZ.P# )
      THEN EXISTS SPJY ( SPJY.S# = SY.S# AND
                         SPJY.P# = PZ.P# )
      END IF )
    AND
    ( IF   EXISTS SPJY ( SPJY.S# = SY.S# AND
                         SPJY.P# = PZ.P# )
      THEN EXISTS SPJX ( SPJX.S# = SX.S# AND
                         SPJX.P# = PZ.P# )
      END IF ) )
```

7.13.49
```
( SPJX.S#, SPJX.P#, ( SPJY.J#, SPJY.QTY WHERE
                      SPJY.S# = SPJX.S# AND
                      SPJY.P# = SPJX.P# ) AS JQ )
```

7.13.50 设R是前面解答中表达式的计算结果，则：

```
RANGEVAR RX RANGES OVER R ,
RANGEVAR RY RANGES OVER RX.JQ ;

( RX.S#, RX.P#, RY.J#, RY.QTY )
```

我们稍微扩充了范围变量定义的语法和语义。这种思想是RY的定义依赖于RX的定义（它们是两个操作中独立的定义）（详细的情况参看[3.3]）。

7.14 我们用7.14.*n*的形式列出了下述习题，其中的6.*n*表示第6章中的题目。对于范围变量的定义和命名，我们沿用7.6节的习惯。

7.14.13
```
( JX, NAMEX, CITYX )
WHERE J ( J#:JX, JNAME:NAMEX, CITY:CITYX )
```

7.14.14
```
( JX, NAMEX, 'London' AS CITY )
WHERE J ( J#:JX, JNAME:NAMEX, CITY:'London' )
```

7.14.15
```
SX WHERE SPJ ( S#:SX, J#:J#('J1') )
```

7.14.16
```
( SX, PX, JX, QTYX )
WHERE SPJ ( S#:SX, P#:PX, J#:JX, QTY:QTYX )
AND   QTYX ≥ QTY ( 300 ) AND QTYX ≤ QTY ( 750 )
```

7.14.17
```
( COLORX, CITYX WHERE P ( COLOR:COLORX, CITY:CITYX
```

7.14.18
```
( SX, PX, JX ) WHERE EXISTS CITYX
                    ( S ( S#:SX, CITY:CITYX ) AND
                      P ( P#:PX, CITY:CITYX ) AND
                      J ( J#:JX, CITY:CITYX ) )
```

7.14.19
```
( SX, PX, JX )
WHERE EXISTS CITYX EXISTS CITYY EXISTS CITYZ
                ( S ( S#:SX, CITY:CITYX ) AND
                  P ( P#:PX, CITY:CITYY ) AND
                  J ( J#:JX, CITY:CITYZ )
                  AND ( CITYX ≠ CITYY OR
                        CITYY ≠ CITYZ OR
                        CITYZ ≠ CITYX ) )
```

7.14.20
```
( SX, PX, JX )
WHERE EXISTS CITYX EXISTS CITYY EXISTS CITYZ
                ( S ( S#:SX, CITY:CITYX ) AND
                  P ( P#:PX, CITY:CITYY ) AND
```

```
                    J ( J#:JX, CITY:CITYZ )
                    AND ( CITYX ≠ CITYY AND
                          CITYY ≠ CITYZ AND
                          CITYZ ≠ CITYX ) )
```

7.14.21
```
PX WHERE EXISTS SX ( SPJ ( P#:PX, S#:SX ) AND
                         S ( S#:SX, CITY:'London' ) )
```

7.14.22
```
PX WHERE EXISTS SX EXISTS JX
          ( SPJ ( S#:SX, P#:PX, J#:JX )
            AND S ( S#:SX, CITY:'London' )
            AND J ( J#:JX, CITY:'London' ) )
```

7.14.23
```
( CITYX AS SCITY, CITYY AS JCITY )
          WHERE EXISTS SX EXISTS JY
              ( S ( S#:SX, CITY:CITYX )
                AND J ( J#:JY, CITY:CITYY )
                AND SPJ ( S#:SX, J#:JY ) )
```

7.14.24
```
PX WHERE EXISTS SX EXISTS JX EXISTS CITYX
          ( S ( S#:SX, CITY:CITYX )
            AND J ( J#:JX, CITY:CITYX )
            AND SPJ ( S#:SX, P#:PX, J#:JX ) )
```

7.14.25
```
JY WHERE EXISTS SX EXISTS CITYX EXISTS CITYY
          ( SPJ ( S#:SX, J#:JY )
            AND S ( S#:SX, CITY:CITYX )
            AND J ( J#:JY, CITY:CITYY )
            AND CITYX ≠ CITYY )
```

7.14.26
```
( PX AS XP#, PY AS YP# ) WHERE EXISTS SX
                               ( SPJ ( S#:SX, P#:PX )
                                 AND SPJ ( S#:SX, P#:PY )
                                 AND PX < PY )
```

7.14.27-7.14.30 略

7.14.31
```
NAMEX WHERE EXISTS JX
            ( J ( J#:JX, JNAME:NAMEX )
              AND SPJ ( S#:S#('S1'), J#:JX ) )
```

7.14.32
```
COLORX WHERE EXISTS PX
             ( P ( P#:PX, COLOR:COLORX ) AND
               SPJ ( S#:S#('S1'), P#:PX ) )
```

7.14.33
```
PX WHERE EXISTS JX
          ( SPJ ( P#:PX, J#:JX ) AND
            J ( J#:JX, CITY:'London' ) )
```

7.14.34
```
JX WHERE EXISTS PX
          ( SPJ ( P#:PX, J#:JX ) AND
            SPJ ( P#:PX, S#:S#('S1') ) )
```

7.14.35
```
SX WHERE EXISTS PX EXISTS SY EXISTS PY
          ( SPJ ( S#:SX, P#:PX ) AND
            SPJ ( P#:PX, S#:SY ) AND
            SPJ ( S#:SY, P#:PY ) AND
            P ( P#:PY, COLOR:COLOR('Red') ) )
```

7.14.36
```
SX WHERE EXISTS STATUSX EXISTS STATUSY
          ( S ( S#:SX, STATUS:STATUSX ) AND
            S ( S#:S#('S1'), STATUS:STATUSY ) AND
            STATUSX < STATUSY )
```

7.14.37
```
JX WHERE EXISTS CITYX
          ( J ( J#:JX, CITY:CITYX ) AND
            FORALL CITYY ( IF J ( CITY:CITYY )
                           THEN CITYY ≥ CITYX
                           END IF )
```

7.14.38~7.14.39 略。

7.14.40
```
JX WHERE J ( J#:JX ) AND
           NOT EXISTS SX EXISTS PX
             ( SPJ ( S#:SX, P#:PX, J#:JX ) AND
```

```
                S ( S#:SX, CITY:'London' ) AND
                P ( P#:PX, COLOR:COLOR('Red') ) )
```

7.14.41
```
JX WHERE J ( J#:JX )
   AND   FORALL SX ( IF SPJ ( S#:SX, J#:JX )
                     THEN SX = S#('S1')
                     END IF )
```

7.14.42
```
PX WHERE P ( P#:PX )
   AND   FORALL JX ( IF J ( J#:JX, CITY:'London' )
                     THEN SPJ ( P#:PX, J#:JX )
                     END IF )
```

7.14.43
```
SX WHERE S ( S#:SX )
   AND   EXISTS PX FORALL JX
       ( SPJ ( S#:SX, P#:PX, J#:JX ) )
```

7.14.44
```
JX WHERE J ( J#:JX )
   AND   FORALL PX ( IF SPJ ( S#:S#('S1'), P#:PX )
                     THEN SPJ ( P#:PX, J#:JX )
                     END IF )
```

7.14.45
```
CITYX WHERE S ( CITY:CITYX )
      OR    P ( CITY:CITYX )
      OR    J ( CITY:CITYX )
```

7.14.46
```
PX WHERE EXISTS SX ( SPJ ( S#:SX, P#:PX ) AND
                     S ( S#:SX, CITY:'London' ) )
   OR    EXISTS JX ( SPJ ( J#:JX, P#:PX ) AND
                     J ( J#:JX, CITY:'London' ) )
```

7.14.47
```
( SX, PX ) WHERE S ( S#:SX ) AND P ( P#:PX )
                 AND NOT SPJ ( S#:SX, P#:PX )
```

7.14.48
```
( SX AS XS#, SY AS YS# )
WHERE S ( S#:SX ) AND S ( S#:SY ) AND FORALL PZ
  ( ( IF SPJ ( S#:SX, P#:PZ ) THEN SPJ ( S#:SY, P#:PZ )
      END IF )
    AND
    ( IF SPJ ( S#:SY, P#:PZ ) THEN SPJ ( S#:SX, P#:PZ )
      END IF ) )
```

7.14.49~7.14.50 略。

7.15 我们用7.15.*n*的形式列出了下述习题，其中的6.*n*表示第6章中的题目。

7.15.13
```
SELECT *
FROM   J ;
```

或者简单地写为：

```
TABLE J ;
```

7.15.14
```
SELECT J.*
FROM   J
WHERE  J.CITY = 'London' ;
```

7.15.15
```
SELECT DISTINCT SPJ.S#
FROM   SPJ
WHERE  SPJ.J# = 'J1' ;
```

7.15.16
```
SELECT SPJ.*
FROM   SPJ
WHERE  SPJ.QTY >= 300
AND    SPJ.QTY <= 750 ;
```

7.15.17
```
SELECT DISTINCT P.COLOR, P.CITY
FROM   P ;
```

7.15.18
```
SELECT S.S#, P.P#, J.J#
FROM   S, P, J
```

```
        WHERE  S.CITY = P.CITY
        AND    P.CITY = J.CITY ;
```

7.15.19
```
        SELECT S.S#, P.P#, J.J#
        FROM   S, P, J
        WHERE  NOT ( S.CITY = P.CITY AND
                     P.CITY = J.CITY ) ;
```

7.15.20
```
        SELECT S.S#, P.P#, J.J#
        FROM   S, P, J
        WHERE  S.CITY <> P.CITY
        AND    P.CITY <> J.CITY
        AND    J.CITY <> P.CITY ;
```

7.15.21
```
        SELECT DISTINCT SPJ.P#
        FROM   SPJ
        WHERE  ( SELECT S.CITY
                 FROM   S
                 WHERE  S.S# = SPJ.S# ) = 'London' ;
```

7.15.22
```
        SELECT DISTINCT SPJ.P#
        FROM   SPJ
        WHERE  ( SELECT S.CITY
                 FROM   S
                 WHERE  S.S# = SPJ.S# ) = 'London'
        AND    ( SELECT J.CITY
                 FROM   J
                 WHERE  J.J# = SPJ.J# ) = 'London' ;
```

7.15.23
```
        SELECT DISTINCT S.CITY AS SCITY, J.CITY AS JCITY
        FROM   S, J
        WHERE  EXISTS
             ( SELECT *
               FROM   SPJ
               WHERE  SPJ.S# = S.S#
               AND    SPJ.J# = J.J# ) ;
```

7.15.24
```
        SELECT DISTINCT SPJ.P#
        FROM   SPJ
        WHERE  ( SELECT S.CITY
                 FROM   S
                 WHERE  S.S# = SPJ.S# ) =
               ( SELECT J.CITY
                 FROM   J
                 WHERE  J.J# = SPJ.J# ) ;
```

7.15.25
```
        SELECT DISTINCT SPJ.J#
        FROM   SPJ
        WHERE  ( SELECT S.CITY
                 FROM   S
                 WHERE  S.S# = SPJ.S# ) <>
               ( SELECT J.CITY
                 FROM   J
                 WHERE  J.J# = SPJ.J# ) ;
```

7.15.26
```
        SELECT DISTINCT SPJX.P# AS PA, SPJY.P# AS PB
        FROM   SPJ AS SPJX, SPJ AS SPJY
        WHERE  SPJX.S# = SPJY.S#
        AND    SPJX.P# < SPJY.P# ;
```

7.15.27
```
        SELECT COUNT ( DISTINCT SPJ.J# ) AS N
        FROM   SPJ
        WHERE  SPJ.S# = 'S1' ;
```

7.15.28
```
        SELECT SUM ( SPJ.QTY ) AS X
        FROM   SPJ
        WHERE  SPJ.S# = 'S1'
        AND    SPJ.P# = 'P1' ;
```

7.15.29
```
        SELECT SPJ.P#, SPJ.J#, SUM ( SPJ.QTY ) AS Y
        FROM   SPJ
        GROUP  BY SPJ.P#, SPJ.J# ;
```

7.15.30
```
SELECT DISTINCT SPJ.P#
FROM   SPJ
GROUP  BY SPJ.P#, SPJ.J#
HAVING AVG ( SPJ.QTY ) > 350 ;
```

7.15.31
```
SELECT DISTINCT J.JNAME
FROM   J, SPJ
WHERE  J.J# = SPJ.J#
AND    SPJ.S# = 'S1' ;
```

7.15.32
```
SELECT DISTINCT P.COLOR
FROM   P, SPJ
WHERE  P.P# = SPJ.P#
AND    SPJ.S# = 'S1' ;
```

7.15.33
```
SELECT DISTINCT SPJ.P#
FROM   SPJ, J
WHERE  SPJ.J# = J.J#
AND    J.CITY = 'London' ;
```

7.15.34
```
SELECT DISTINCT SPJX.J#
FROM   SPJ AS SPJX, SPJ AS SPJY
WHERE  SPJX.P# = SPJY.P#
AND    SPJY.S# = 'S1' ;
```

7.15.35
```
SELECT DISTINCT SPJX.S#
FROM   SPJ AS SPJX, SPJ AS SPJY, SPJ AS SPJZ
WHERE  SPJX.P# = SPJY.P#
AND    SPJY.S# = SPJZ.S#
AND  ( SELECT P.COLOR
       FROM   P
       WHERE  P.P# = SPJZ.P# ) = 'Red' ;
```

7.15.36
```
SELECT S.S#
FROM   S
WHERE  S.STATUS < ( SELECT S.STATUS
                    FROM   S
                    WHERE  S.S# = 'S1' ) ;
```

7.15.37
```
SELECT J.J#
FROM   J
WHERE  J.CITY = ( SELECT MIN ( J.CITY )
                  FROM   J ) ;
```

7.15.38
```
SELECT DISTINCT SPJX.J#
FROM   SPJ AS SPJX
WHERE  SPJX.P# = 'P1'
AND  ( SELECT AVG ( SPJY.QTY )
       FROM   SPJ AS SPJY
       WHERE  SPJY.J# = SPJX.J#
       AND    SPJY.P# = 'P1' ) >
     ( SELECT MAX ( SPJZ.QTY )
       FROM   SPJ AS SPJZ
       WHERE  SPJZ.J# = 'J1' ) ;
```

7.15.39
```
SELECT DISTINCT SPJX.S#
FROM   SPJ AS SPJX
WHERE  SPJX.P# = 'P1'
AND    SPJX.QTY > ( SELECT AVG ( SPJY.QTY )
                    FROM   SPJ AS SPJY
                    WHERE  SPJY.P# = 'P1'
                    AND    SPJY.J# = SPJX.J# ) ;
```

7.15.40
```
SELECT J.J#
FROM   J
WHERE  NOT EXISTS
     ( SELECT *
       FROM   SPJ, P, S
       WHERE  SPJ.J# = J.J#
       AND    SPJ.P# = P.P#
       AND    SPJ.S# = S.S#
       AND    P.COLOR = 'Red'
       AND    S.CITY = 'London' ) ;
```

7.15.41
```
SELECT J.J#
FROM   J
WHERE  NOT EXISTS
     ( SELECT *
       FROM   SPJ
       WHERE  SPJ.J# = J.J#
       AND    NOT ( SPJ.S# = 'S1' ) ) ;
```

7.15.42
```
SELECT P.P#
FROM   P
WHERE  NOT EXISTS
     ( SELECT *
       FROM   J
       WHERE  J.CITY = 'London'
       AND    NOT EXISTS
            ( SELECT *
              FROM   SPJ
              WHERE  SPJ.P# = P.P#
              AND    SPJ.J# = J.J# ) ) ;
```

7.15.43
```
SELECT S.S#
FROM   S
WHERE  EXISTS
     ( SELECT *
       FROM   P
       WHERE  NOT EXISTS
            ( SELECT *
              FROM   J
              WHERE  NOT EXISTS
                   ( SELECT *
                     FROM   SPJ
                     WHERE  SPJ.S# = S.S#
                     AND    SPJ.P# = P.P#
                     AND    SPJ.J# = J.J# ) ) ) ;
```

7.15.44
```
SELECT J.J#
FROM   J
WHERE  NOT EXISTS
     ( SELECT *
       FROM   SPJ AS SPJX
       WHERE  SPJX.S# = 'S1'
       AND    NOT EXISTS
            ( SELECT *
              FROM   SPJ AS SPJY
              WHERE  SPJY.P# = SPJX.P#
              AND    SPJY.J# = J.J# ) ) ;
```

7.15.45
```
SELECT S.CITY FROM S
UNION
SELECT P.CITY FROM P
UNION
SELECT J.CITY FROM J ;
```

7.15.46
```
SELECT DISTINCT SPJ.P#
FROM   SPJ
WHERE  ( SELECT S.CITY
         FROM   S
         WHERE  S.S# = SPJ.S# ) = 'London'
OR     ( SELECT J.CITY
         FROM   J
         WHERE  J.J# = SPJ.J# ) = 'London' ;
```

7.15.47
```
SELECT S.S#, P.P#
FROM   S, P
EXCEPT
SELECT SPJ.S#, SPJ.P#
FROM   SPJ ;
```

7.15.48 略。

7.15.49~7.15.50无法求解。