第6章 关系代数

6.1 引言

自从Codd的文章［5.1, 6.1］发表后，关系模型的操作部分已取得了相当大的发展。操作的主要组成部分是关系代数，它是一个操作符的集合，以关系作为操作对象，返回的结果是一个关系。在第3章中，已经比较简单地介绍了三个操作符：选择（restrict），投影（project）和连接（join）；本章将要深入地学习这些操作符以及其他几个操作符。

在参考文献［6.1］中，Codd定义了一个如图6-1所示的含8个操作符的集合，即通常所说的基本（original）关系代数。现在，Codd想要给这8个操作符作出精确的定义，下一章将会看到。但必须明白这8个操作符绝不是全部，实际上满足“关系进，关系出”这一简单要求的任何操作符都可以定义。许多学者也已经定义了不少操作符（看参考文献［6.10］）。本章首先讨论Codd最初的操作符（或至少是我们的版本），然后以此为基础讨论有关代数学的种种思想；并考虑对这8个操作符的基本集合进行有益的扩展。

基本代数概述

图6-1中的基本代数由8个操作符组成，可分成两组：

1) 传统的集合操作符并（union），交（intersection），差（difference）和笛卡尔积（Cartesian product）（所做的修改只是操作对象变为了特定的关系，而不再是任意的集合）；
2) 专门的关系操作符restrict（也称为select——选择）、投影（project）、连接（join）和除（divide）。

下面给出这8个操作符的简单定义（参考图6-1）：

选择：返回一个关系，其中的元组来自指定关系中所有满足指定条件的元组。

投影：返回一个关系，由去掉若干属性列后的指定关系中剩余的所有（子）元组组成。

积：返回一个关系，包含任意两个分别来自两个指定关系的元组组合的所有可能的元组。

并：返回的关系由两个指定关系中所有的元组构成。

交：返回关系由同时出现在两个指定关系中的元组构成。

差：返回的关系由那些属于第一个关系却不属于第二个关系的元组构成。

连接：返回关系中的元组是两个元组的结合，这两个元组分别来自两个指定的关系，需满足的条件是此两个关系存在相同的属性，且在相同属性上有相同的值（在结果元组中，共同的值只出现一次，而不是两次）。

除：此操作是在两个单目关系和一个双目关系上，返回关系的元组满足以下条件：这些元组来自一个单目关系，其在双目关系中的对应元组能与另一个单目关系中的所有元组相匹配。

对基本操作符的概述先到此为止。本章接下来的安排如下：在本节的简介之后，6.2节再次讨论关系封闭的问题，着重举例说明。6.3节和6.4节详细讨论Codd的8个基本操作符，6.5节

给出用这些操作符进行查询的一些例子。接下来，6.6节考虑了代数应用的更多问题。6.7节描述了对Codd基本代数的一些有益扩充，包括EXTEND和SUMMERIZE这两个重要的操作符。6.8节讨论了在含有关系型属性的关系和只含有标量属性的关系之间进行映射的操作符。6.9节讲了关系的比较。最后，6.10节提供了一个简要的小结。

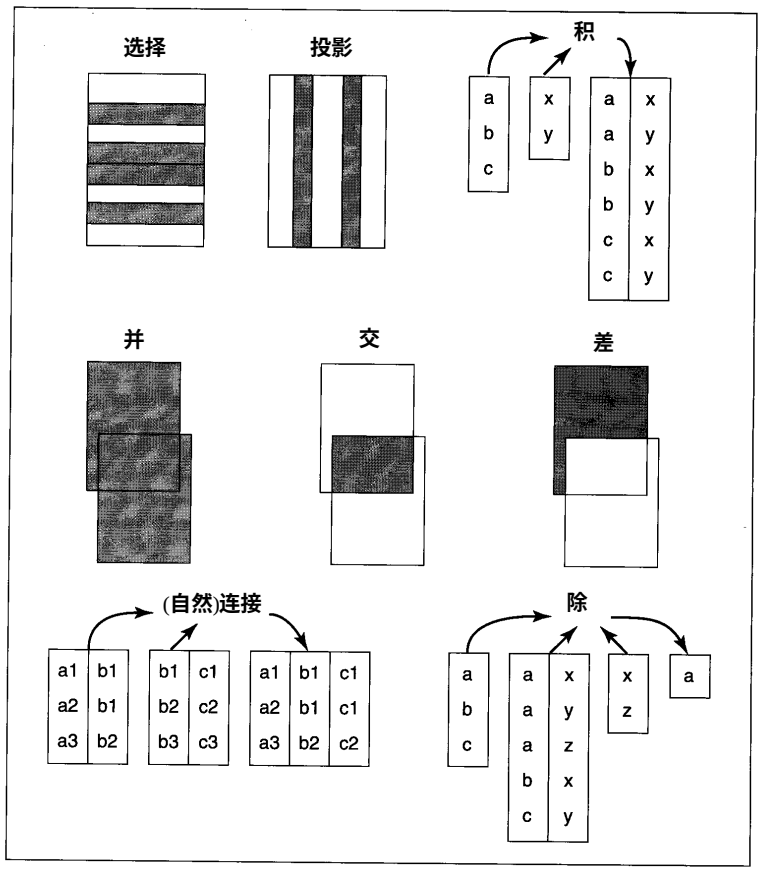

图6-1 基本关系代数的8个操作符（概览）

以下是两个预先的提示：

- 由于众所周知的原因，经常说到“ X是R上的一个选择 ”(其中R是一个具体的关系变量)，而更准确的说法是“ X是关系变量R当前值的一个选择 ”，或者“ X是一个变量，它的值是关系变量R当前值的一个选择 ”。我们认为该简略的说法不会引起混淆。
- SQL中类似于关系代数操作符的问题推迟到第7章进行讨论，原因将在该章讲到。

6.2 关系封闭性

在第3章中已经看到，任何关系操作的结果是另一个关系，这一事实被称为是关系封闭的特性。封闭性意味着可以写出嵌套的关系表达式，也就是说，关系表达式的操作对象可以用任意复杂的关系表达式来表示（关系代数中的关系嵌套和普通算术中的算术表达式嵌套有着类似之处；代数中的关系是封闭的，这一点和“ 普通算术中的数是封闭的 ”这一事实具有同样的重要性)。

当第3章讨论封闭性时，故意忽略了非常重要的一点。回顾一下可知，每个关系分为两部分——表头和主体；不严格地讲，表头是属性，主体是元组。现在，基本关系（即基本关系变量的值）的表头对系统来说是非常容易理解的。但是由此而得出的关系会怎样呢？例如，考虑下面表达式：

```
S JOIN P
```

（本例表示了供应商和零件关系通过匹配城市进行连接。CITY是两个关系之间的唯一的共同属性）。我们知道结果中主体的形式，但表头是何种形式呢？封闭性规定，结果必须有一个表头，并且对系统是可知的（实际上，用户也必须知道，过一会儿将会看到）。换句话说，结果必须是一个定义好的关系类型。如果严格地根据封闭性，定义的关系操作必须保证每个操作的结果具有正确关系类型——即，带有正确的属性名称[⊖]。

每个结果关系需要含有正确的属性名称，原因之一当然是可以使在后面的操作中利用这些属性名——特别是在整个嵌套表达式的其余部分的操作中。例如，如果不知道 S JOIN P的结果中有一个叫CITY的属性名，就不能写一个如下的表达式：

```
(S JOIN P) WHERE CITY='Athens'
```

因此，我们需要的是一个嵌入到代数中的类型参照规则的集合（关系的），以利于如果知道给定关系操作中输入关系的类型，则可以推断输出的类型。假如有这样一个规则，那么任意的关系表达式，不论怎样复杂，产生的结果就会有一个定义好的（关系）类型，且有一个定义好的属性名称的集合。

为此先介绍一个新的操作符——RENAME，它的作用是在给定的关系中更改属性的名称。确切地说，RENAME返回的关系与原给定关系是一样的，除了至少一个属性名改变以外（给定的关系可通过关系表达式来指定，当然其中可以包含一些关系操作）。例如，可以写出如下的式子：

```
S RENAME CITY AS SCITY
```

注意，这不是一个命令或声明，而是一个表达式。因此，它可以嵌套在别的表达式里面——产生一个与关系S有相同表头和主体的关系，只是属性CITY更名为SCITY：

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

注意：请注意SNAME表达式没有改变数据库中的供应商关系变量——它只是一个表达式，就象S JOIN SP一样，产生了一个确定的结果（在这个特殊的例子中，这个结果碰巧看起来象供应商关系变量的当前值）。

下面有另外一个例子（这次是多个改名）：

```
P  RENAME  PNAME  AS  PN , WEIGHT  AS  WT
```

结果如下：

⊖ 代数的这方面内容在很多文献中被相当程度地忽略了(SQL语言和SQL产品中同样如此)，Hall et al.[6.10]和Darwen[6.2]除外。本章有关代数的内容受了这两本书的很大影响。

P#	PN	COLOR	WT	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

有必要明确地指出，RENAME的应用意味着关系代数（不像SQL）没有必要使用像S.S#那样严格的名字。

6.3 语法

本节中，将描述关系代数表达式具体的语法。注意：大多数数据库书籍的关系操作符使用了某种数学的或希腊的符号，典型的有以σ表示选择，π表示投影，∩表示交，⋈表示连接，等等。大家可能注意到，我们喜欢用诸如JOIN和WHERE等关键字。使用关键字虽会使表达式变得稍长一些，但它使表达式变得更易读。

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

*<relational expression>*是一个关系表达式（当然是一个关系值）。第一个格式是关系选择子调用（包括作为特殊情况的关系文字）；这里不列出*<tuple expression>*的详细语法，只是用例子来说明基本的思想。其余的格式意思比较明显。

```
<relational operation>
    ::=   <project> | <nonproject>
```

在语法中，由于操作符优先的原因，把*<project>*与*<nonproject>*区分开来（给projection赋予一个高优先级是很方便的）。

```
<project>
    ::=   <relational expression>
                { [ ALL BUT ] <attribute name commalist
```

*<relational expression>*必须不是一个*<nonproject>*（非投影符）。

```
<nonproject>
    ::=   <rename>
        | <union> | <intersect> | <minus> | <times>
        | <restrict> | <join> | <divide>

<rename>
    ::=   <relational expression>
                RENAME <renaming commalist>
```

*<relational expression>*必须不是一个*<nonproject>*。前一节介绍了*<renaming>*的语法。

```
<union>
    ::=   <relational expression>
                UNION <relational expression>
```

*<relational expression>*必须不是一个*<nonproject>*。除非其中之一或两者都是另一个*<union>*。

```
<intersect>
    ::=   <relational expression>
                INTERSECT <relational expression>
```

*<relational expression>*必须不是一个*<nonproject>*。除非其中之一或两者都是另一个

<intersect>。

```
<minus>
    ::=   <relational expression>
                MINUS <relational expression>
```

<relational expression>必须不是一个<nonproject>。

```
<times>
    ::=   <relational expression>
                TIMES <relational expression>
```

<relational expression>必须不是一个<nonproject>。除非其中之一或两者都是另一个<times>。

```
<restrict>
    ::=   <relational expression> WHERE <boolean expression>
```

<relational expression>必须不是一个<nonproject>。注意：<boolean expression>可以包含<relational expression>表示的关系的一个属性的引用，其中<relational expression>允许一个选择子调用（即S WHERE CITY = 'London' 是一个合法的<restrict>）。

```
<join>
    ::=   <relational expression>
                JOIN <relational expression>
```

<relational expression>必须不是一个<nonproject>。除非其中之一或两者都是另一个<join>。

```
<divide>
    ::=   <relational expression>
            DIVIDEBY <relational expression> PER <per>
```

<relational expression>必须不是一个<nonproject>。

```
<per>
    ::=   <relational expression>
        | ( <relational expression>, <relational expression> )
```

<relational expression>必须不是一个<nonproject>。

6.4 语义

本节将举例说明6.3节中的语法。依次考虑下面的操作符：并、交、差、积、选择、投影、连接和除（重命名在6.2节已经讲过）。

1. 并

数学中两个集合的并是这两个集合的所有元素组成的集合。因为一个关系是（或更确切地说是包含）一个集合，即一个元组的集合，所以构造这样两个集合的并是完全可能的；所得结果包含了出现在任一个或两个原关系中的所有元组。例如，出现在关系变量S中的供应商元组的集合与出现在关系变量P中的零件元组的集合的并当然是一个集合。

然而，尽管这一结果是一个集合，却不是一个关系；关系不能含有不同类型的元组，其中的元组必须是同类的。当然，我们希望结果是一个关系，因为要保持封闭性。所以，关系代数中的并，不是通常数学中的并；它是一种特殊类型的并，要求两个参与操作的关系是同一类型 —— 即它们或者都包含供应商元组，或者都包含零件元组，而不能是两者的混合。如果两个关系属于同一类型，那就可以进行并操作，得到的结果是一个相同类型的关系；换句话说，封闭的特性被保持了下来[⊖]。

⊖ 以前，大多数数据库书籍（包括本书的早期版本）使用术语“并相容性(unioncompatibility)”来表示两个关系必须是同一类型。然而，由于多种原因，这个术语并不恰当，最明显的一点是此术语不是仅适用于并的。

下面是关系并操作的定义：给定两个相同类型的关系 *A*和*B*，两者的并即*A* UNION *B*是相同类型的一个关系，关系的主体由出现在*A*中或*B*中或同时出现在两者之中的所有元组组成。

例如：假设关系*A*和*B*如图6-2所示（都从供应商关系变量导出；*A*是在伦敦的供应商，*B*是提供零件P1的供应商）。*A* UNION *B*（参看图的a部分）就是或者在伦敦、或者提供零件 P1的供应商（或者两者兼有）。注意，结果有3个而不是4个元组；重复的元组按照定义被删除掉了。其他涉及到删除重复元组的操作符只有投影（在本节的后半部分将会讲到）。

2. 交

由于和并基本相同的原因，关系交操作符的操作对象必须是相同类型的。给定类型相同的关系*A*和*B*，它们的交*A* INTERSECT *B*是一个相同类型的关系，关系的主体包含同时出现在*A*和*B*中的所有元组。

例如：假设关系*A*和*B*如图6-2所示，*A* INTERSECT *B*（参看图的b部分）就是既在伦敦又提供零件P1的供应商。

3. 差

像并和交一样，关系的差操作符也要求操作对象是同一类型。给定两个类型相同的关系 *A*和*B*，它们的差*A* MINUS *B*（两者有先后次序）是一个与它们的类型相同的关系，关系的主体包含属于*A*但不属于*B*的所有元组。

例如：*A*和*B*如图6-2所示。*A* MINUS *B*（参看图的c部分）是在伦敦、但不提供零件 P1的供应商，*B* MINUS *A*（参看图的d部分）是提供零件P1、但不在伦敦的供应商。注意 MINUS有一个顺序性，就像通常数学中的减法（“5-2”和“2-5”不是同一件事情）。

A

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London

B

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris

a. *Union*
(A UNION B)

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London
S2	Jones	10	Paris

b. *Intersection*
(A INTERSECT B)

S#	SNAME	STATUS	CITY
S1	Smith	20	London

c. *Difference*
(A MINUS B)

S#	SNAME	STATUS	CITY
S4	Clark	20	London

d. *Difference*
(B MINUS A)

S#	SNAME	STATUS	CITY
S2	Jones	10	Paris

图6-2 并、交、差举例

4. 积

数学里的两个集合的笛卡尔积是满足如下条件的有序对的集合：每个有序对的第一个元素来自于第一个集合，第二个元素来自第二个集合。因此，两个关系的笛卡尔积可粗略地说是有序元组对的集合。但我们想保持封闭的特性；换句话说，我们想要结果包含元组本身，

而不是有序的元组对。因此，关系的笛卡尔积是对这一操作的一个扩充，其中的每个有序元组对代替以两个相关元组相并得出的一个元组（这里的“并”是一般集合理论上的并，而不是特殊的关系意义上）。因此给定：

{A1:a1, A2:a2, …, Am:am}

和

{B1:b1, B2:b2, …, Bn:bn}

两者的并是一个单个元组：

{A1:a1, A2:a2, …, Am:am, B1:b1, B2:b2, …, Bn:bn}

笛卡尔积连接中的另一个问题是：需要结果关系有一个正确形式的表头（即正确的关系类型）。现在已明确的是结果的表头包含了两个输入关系的所有属性。如果两个关系的表头有共同的属性名，问题就会出现了；如果操作允许，结果的表头会有两个相同名称的属性，这就不再是一个“好的形式”（well-formed）。如果对两个有相同属性名称的关系进行笛卡尔积操作，必须首先用RENAME操作符适当地更改属性的名称。

前面定义了两个关系*A*和*B*的笛卡尔积：*A* TIMES *B*，其中*A*和*B*没有共同的属性名称，两者的笛卡尔积是一个关系，它的表头是*A*和*B*表头的并，主体包括所有*A*中的元组和*B*中的元组进行并操作而得到的元组。注意结果的势是*A*的势和*B*的势的乘积，结果的度是*A*的度和*B*的度的和。

例如：关系*A*和*B*如图6-3所示（*A*是所有的供应商号码，*B*是所有的零件号码）。于是*A* TIMES *B*（参看图的下半部分）就是所有的供应商号码和零件号码对。

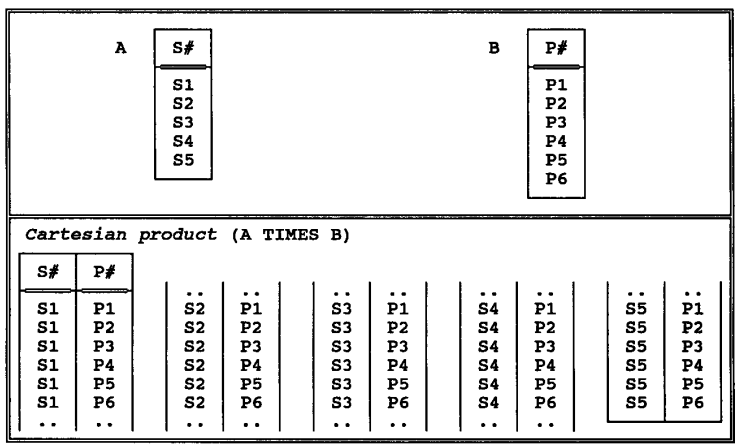

图6-3 笛卡尔积举例

5. 选择

假设关系*A*含有属性*X*和*Y*（也可能有别的），假设θ是一个比较操作符，诸如“=”、“>”，等等，*X*θ*Y*为定义好的条件，若给*X*和*Y*赋予具体的值，则能计算出真值（true或false）。关系*A*在属性*X*和*Y*上的θ选择是一个关系，关系的表头和*A*的一样，主体包括所有满足条件*X*θ*Y*为真的元组。

需说明几点：

1) 条件中的*X*或*Y*或两者都可以被一个选择子调用代替；实际上，这是很普通的情况。然而，应该解释清楚这种“普通情况”只是简写。例如，选择

```
S WHERE CITY = 'London'
```

实际上是下面形式的简写

```
( EXTEND S ADD 'London' AS TEMP ) WHERE CITY = TEMP
```

（其中TEMP是任意一个名字）。可以参看6.7节有关EXTEND的论述。

2) 形式为b（b是一个布尔型的选择子调用）的条件也是合法的。

3) 这里定义的选择在WHERE子句中只允许有一个条件。然而借助于封闭性，可以把它扩展为在WHERE子句中含任意布尔型条件的组合，就像下面的等价形式：

```
A WHERE c1 AND c2   ≡  ( A WHERE c1 ) INTERSECT ( A WHERE c2 )

A WHERE c1 OR c2    ≡  ( A WHERE c1 ) UNION ( A WHERE c2 )

A WHERE NOT c       ≡  A MINUS ( A WHERE c )
```

自此以后，假定在WHERE子句中的<*boolean expression*>可包含任意条件的组合（必要时也可带括号来保证操作的次序），其中每个条件包含相关关系的属性或选择子调用或两者兼有。注意<*boolean expression*>可通过只检查特定的元组来判断是真或假。<*horizontal*>可说是一个选择条件。

选择操作符能有效地产生给定关系的水平（horizontal）子集——即给定关系元组的子集满足特定的选择条件。图6-4给出了几个例子。

S WHERE CITY = 'London'

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London

P WHERE WEIGHT < WEIGHT (14.0)

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P5	Cam	Blue	12.0	Paris

SP WHERE S# = S# ('S6') OR P# = P# ('P7')

S#	P#	QTY

图6-4 选择举例

6. 投影

假定关系*A*有属性***X***, *Y*,…, *Z*（可能还有其他的）。关系*A*在*X*, Y, …, *Z*上的投影

A {*X*, *Y*, …, *Z* }

是一个满足如下条件的关系：

- 表头由*A*的表头除去不包含在集合 {***X***, *Y*, …, ***Z***} 中的属性而得到；
- 主体包含所有形式为 {*X*: *x*，*Y*: *y* , …, ***Z***: *z*} 的元组，这样的元组是由对应关系*A*的属性*X*, *Y*, …, ***Z***的值*x*, *y*, …, *z* 组成。

投影操作能有效地产生给定关系的垂直（vertical）子集——子集是由除去不包含在指定列表中的属性，且消除由此产生的重复（子）元组而得出。

几点解释：

1) 在属性名称的列表中，不能有重复的属性（为什么不能？）。

2) 如果属性名称列表包含*A*的所有属性，则此投影是一个等同（identity）投影。

3) 型如*A* {}（也就是属性列表是空的）的投影是合法的。它表示一个空（ nullary）投影。参看本章后面的练习6.8~6.10。

图6-5给出了几个投影的例子。注意第一个例子（供应商在 CITY上的投影），尽管关系变量S含有5个元组即有5个城市，但结果中只有 3个城市 —— 重复的城市（即重复的元组）被删除了。在其他例子上可以有类似的情况。

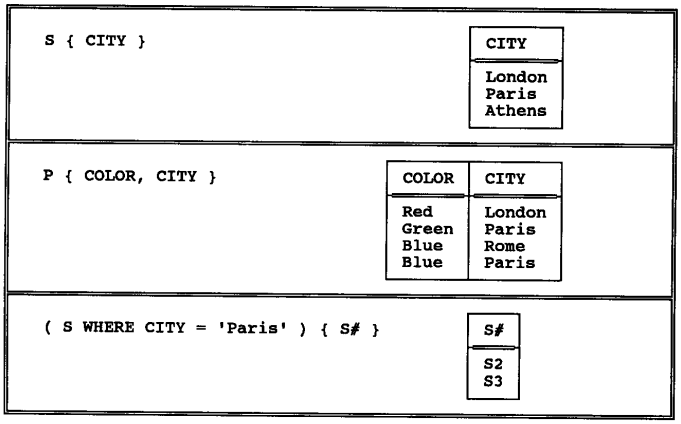

图 6-5

实际中经常出现这种情况：指定被投影掉的属性比指定投影的属性更方便。例如，可以说“从关系P中投影掉属性WEIGHT”，而不是“关系P在属性P#、PNAME、COLOR和CITY上的投影”，就像下面：

```
P { ALL BUT WEIGHT }
```

7. 连接

连接存在几种不同种类的变种。然而，最简单、且最重要的是所谓的自然连接，按实际中非正式的说法，join总是特指自然连接，本书也采用这种用法。下面是有关定义（有点抽象，但读者应从第3章的讨论中对自然连接有些直观的了解）。假设A和*B*分别有表头：

```
{X1, X2,..., Xm, Y1, Y2, ...,} Yn
```

和

```
{Y1, Y2, ... Yn, Z1, Z2, ..., Zp}
```

即*Y*属性*Y1*, *Y2*, ..., *Yn*是两个关系的共同属性，*X*属性*X1*, *X2*, ..., *Xm*是*A*的特有属性，*Z*属性*Z1*, *Z2*, ..., *Zp*是*B*的特有属性。现在把 {*X1*, *X2*, ..., *Xm*}、{*Y1*, *Y2*, ...,*Yn*} 和 {*Z1*, *Z2*, ..., *Zp*} 分别看作是复合属性*X*、*Y*和*Z*。于是*A*和*B*的自然连接

```
A JOIN B
```

是一个关系。它的表头是 {*X*，*Y*，*Z*} ；主体包含所有元组 {*X* : *x*,*Y* : *y*, *Z* : *z*}，其中含有*X*的值*x*和*Y*的值*y*的元组出现在*A*中，含有*Y*的值*y*和*Z*的值*z*的元组出现在*B*中。

图6-6给出了一个自然连接（在共同属性 CITY上的自然连接S JOIN P）。

注意：连接并不总是在一个外码和一个相匹配的主码之间，虽然这样的连接很普通且很重要的。关于这一点已经举例说了多遍，实际上图 6-6也作了说明，但仍需要讲清楚。

接下来考虑Θ连接操作。当在比较而不是相等操作符的基础上连接两个关系的时候，需要

Θ连接(这种情况相对来说比较少，但决不是没有)。假设关系*A*和*B*满足笛卡尔积的条件 (即它们没有共同的属性名称)；设*A*有属性*X*，见有属性*Y*并且*X*、*Y*和θ满足选择的需求。于是关系*A*和*B*在属性*X*和*Y*上的Θ连接定义为以下表达式操作的结果。

S#	SNAME	STATUS	CITY	P#	PNAME	COLOR	WEIGHT
S1	Smith	20	London	P1	Nut	Red	12.0
S1	Smith	20	London	P4	Screw	Red	14.0
S1	Smith	20	London	P6	Cog	Red	19.0
S2	Jones	10	Paris	P2	Bolt	Green	17.0
S2	Jones	10	Paris	P5	Cam	Blue	12.0
S3	Blake	30	Paris	P2	Bolt	Green	17.0
S3	Blake	30	Paris	P5	Cam	Blue	12.0
S4	Clark	20	London	P1	Nut	Red	12.0
S4	Clark	20	London	P4	Screw	Red	14.0
S4	Clark	20	London	P6	Cog	Red	19.0

图6-6 自然连接S JOIN P

```
(A TIMES B) WHERE X Θ Y
```

结果是一个关系，关系的表头和*A*与*B*的笛卡尔积的表头相同，主体包含满足以下条件的元组*t*：*t*出现在笛卡尔积中，且使条件“ *X*Θ*Y* ”为真。

假设需要计算关系 S和P在CITY上的大于（greater-than）连接（这里的Θ是“ > ”；假设“ > ”对CITY有意义，简单地解释为“ 在字母的次序上大于 ”)。适当的关系表达式如下所示：

```
( ( S RENAME CITY AS SCITY ) TIMES
  ( P RENAME CITY AS PCITY ) )
                WHERE SCITY > PCITY
```

注意此例中对属性名称的修改（当然，只修改两个 CITY名称中的一个就足够了；两个都修改的原因是为了对称)。全部表达式的结果在图6-7中表式出来。

如果Θ是“ = ”，则Θ连接称为等值连接(equijoin)。由定义可知，等值连接的结果必须包含两个满足以下条件的属性：这两个属性的值在关系的每个元组上是相等的。如果这两个属性的其中一个被投影掉且另一个相应的改名（若需要的话)，则结果就是一个自然连接！例如，表达供应商和零件（在CITY上）的自然连接的式子

```
S JOIN P
```

等价于下面更复杂的式子

```
( ( S TIMES ( P RENAME CITY AS PCITY ) )
                    WHERE CITY = PCITY )
                                { ALL BUT PCITY }
```

注意：Tutorial D不直接支持Θ连接操作符，因为（a）在实际中不经常需要；（ b）无论如何它不是一个基本（primitive）的操作符。

S#	SNAME	STATUS	SCITY	P#	PNAME	COLOR	WEIGHT	PCITY
S2	Jones	10	Paris	P1	Nut	Red	12.0	London
S2	Jones	10	Paris	P4	Screw	Red	14.0	London
S2	Jones	10	Paris	P6	Cog	Red	19.0	London
S3	Blake	30	Paris	P1	Nut	Red	12.0	London
S3	Blake	30	Paris	P4	Screw	Red	14.0	London
S3	Blake	30	Paris	P6	Cog	Red	19.0	London

图6-7 供应商和零件在CITY上的大于连接

8. 除

参考文献［6.3］定义了两个不同的“ 除 ”操作：小除（ small divide）和大除（great

divide)。在Tutorial D中，<per>项只含有一个<relational expression>的<divide>称为小除，含有两个<relational expression>的<divide>称为大除。后面的叙述只适用于小除，且只适用于特定形式的小除。关于大除的讨论和有关小除的更详细的内容请参看［6.3］。

应该说明，这里讲的小除与Codd所说的除不一样——实际上，在处理空关系时原先的操作有一定的困难，而小除是对原先的一个改进。它与本书前几个版本所讲的也不一样。

这里给出定义。设关系*A*和*B*分别有表头

{*X1,X2,....,Xm*}

和

{*Y1,Y2,...,Yn*}

设关系*C*有表头

{*X1,X2,....,Xm, Y1,Y2,...,Yn*}

(即*C*的表头是*A*和*B*的表头的并)。现在把{*X1,X2,...,Xm*}和{*Y1,Y2,...,Yn*}分别看作是复合属性*X*和*Y*。于是*A*根据*C*除以*B*（其中*A*是被除数，*B*是除数，*C*是中间数mediator）——

A DIVIDEBY*B* PER *C*

得到的结果是一个关系，关系的表头是{*X*}，主体包含符合如下条件的元组{*X:x*}：它来自*C*的元组{*X:x,Y:y*}，其中每个元组对应于*B*中所有的元组{*Y:y*}。也可以粗略地说，结果关系包含*A*中满足如下条件的*X*值：在*C*中对应的*Y*值包含*B*中的所有*Y*值。

图6-8显示了几个除的简单例子。在每个例子中的被除数（DEND）是关系S在S#上的投影；中间数（MED）是关系SP在S#和P#上的投影；三个除数（DOR）如图所示。特别要注意最后一个例子，其中的除数是一个包含所有已知零件号码的关系；结果显示了提供所有这些零件的供应商的号码。从例子中可以看出，DIVIDEBY操作符用作查询共同的本质；实际上，在自然语言的查询中包含“所有（all）”一类的词（“查询提供所有零件的供应商”），极有可能用到除法[⊖]。然而，需要指出的是，这样的查询经常以关系比较的形式出现（参看6.9节）。

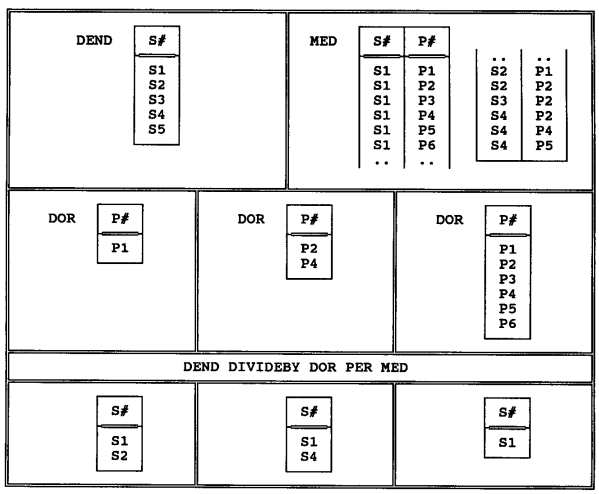

图 6-8

⊖ 实际上，Codd定义“除”，本意是为了对应全称量词(universal quantifier)(参考第7章)，就像投影对应存在量词(existential)。

9. 结合律和交换律

很容易证明并具有结合律——即如果*A*，*B*和*C*是任意的关系表达式，并且产生相同类型的关系，于是表达式

```
( A UNION B ) UNION C
```

和

```
A UNION ( B UNION C )
```

在逻辑上是等价的。方便起见，UNION的表达式中可以不用括号，即以上的式子可以简单地表示为

```
A UNION B UNION C
```

INTERSECT、TIMES和JOIN也有类似的特性（但MINUS除外）。

我们也经常说UNION、INTERSECT、TIMES以及JOIN具有交换律（MINUS除外）——即表达式

```
A UNION B
```

和

```
B UNION A
```

在逻辑上也是等价的，INTERSECT、TIMES和JOIN的情况类似。注意：将在第17章再次讨论结合律和交换律的问题。顺便谈一下TIMES，在集合理论中的笛卡尔积是没有结合律和交换律的，但关系中的笛卡尔积对两个特性都具备。

如果*A*和*B*没有共同的属性，则*A* JOIN *B*等价于*A* TIMES *B*［5.5］——即在这种情况下自然连接退化为笛卡尔积。实际上，正因为这个原因，参考书［3.3］中定义的Tutorial D不直接支持TIMES操作。

6.5 举例

本节提供了用在查询中的有关关系代数表示的一些例子。建议读者对照图3-8中的数据检验这些例子。

1. 求提供零件P2的供应商名称

```
( ( SP JOIN S ) WHERE P# = P# ( 'P2' ) ) { SNAME }
```

解释：首先构造关系SP和S在供应商号码上的自然连接，从概念上来说，它扩展了每个SP元组相应的供应商信息（即SNAME、STATUS和CITY的值）。随后这个连接以零件P2为条件进行了选择。最后选择在SNAME上做了投影。结果中只有SNAME这一个属性。

2. 求提供至少一个红色零件的供应商名称

```
( ( ( P WHERE COLOR = COLOR ( 'RED' ) )
                  JOIN SP ) { S# } JOIN S ) { SNAME }
```

结果唯一的属性还是SNAME。

下面给出对同一个查询的不同的表述：

```
( ( ( P WHERE COLOR = COLOR ( 'Red' ) ) { P# }
                    JOIN SP ) JOIN S ) { SNAME }
```

这个例子说明了一个重要的事实：对同一个查询经常有不同的表述。关于这一点在第17章还有讨论。

3. 求提供所有零件的供应商名称

```
( ( S { S# } DIVIDEBY P { P# } PER SP { S#, P# } )
                                        JOIN S ) { SNAME }
```

结果还是只有一个属性：SNAME。

4. 求至少提供了S2提供的所有零件的供应商号码

```
S { S# } DIVIDEBY ( SP WHERE S# = S# ( 'S2' ) ) { P# }
                                          PER SP { S#, P# }
```

结果只有一个属性：S#。

5. 求住在同一个城市的供应商的号码对(若两个供应商住在同一城市，则它们的号码是一对)

```
( ( ( S RENAME S# AS SA ) { SA, CITY } JOIN
    ( S RENAME S# AS SB ) { SB, CITY } )
        WHERE SA < SB ) { SA, SB }
```

结果有两个属性：SA和SB（当然，只改掉两个S#属性之一的名称就足够了；这里两者都改是为了对称）。注意：假设在类型S#上已经定义了操作符“ < ”。条件SA<SB的作用是双重的：

- 它排除了形式为（*x*,*x*）的供应商号码对；
- 它保证了（*x*, *y*）和（*y*,*x*）不会同时出现。

下面给出这个查询的另一种表示，目的是为了说明 *WITH*的用法：利用WITH可以对表达式进行简写，进而把长的查询简单化（实际上，在第 5章5.2节 —— “ 操作符定义 ”中已经对WITH举过例子）。

```
WITH ( S RENAME S# AS SA ) { SA, CITY } AS T1,
     ( S RENAME S# AS SB ) { SB, CITY } AS T2,
     T1 JOIN T2 AS T3,
     T3 WHERE SA < SB AS T4 :
     T4 { SA, SB }
```

利用WITH就可以考虑大的、复杂的表达式，并且使它不违反关系代数的非过程化特性。在下一个例子之后将对这方面做详细的阐述。

6. 求不提供零件P2的供应商名称

```
( ( S { S# } MINUS ( SP WHERE P# = P# ( 'P2' ) ) { S# } )
                                        JOIN S ) { SNAME }
```

结果只有一个叫做SNAME的属性。

正像前面所说的，我们要对这个例子进行详细的阐述。迅速判断怎样把一个给定的查询表述成一个嵌套表达式并不总是很简单的，也没有必要如此做。下面是例子 6的一个分步表述：

```
WITH S { S# } AS T1,
     SP WHERE P# = P# ( 'P2' ) AS T2,
     T2 { S# } AS T3,
     T1 MINUS T3 AS T4,
     T4 JOIN S AS T5,
     T5 { SNAME } AS T6 :
     T6
```

T6代表想要的结果。解释：WITH子句引进名称 —— 例如型如Ti的名称 —— 对包含子句的声明假定是局部意义上的。如果系统支持“ lazy evaluation ”(例如像PRTV系统所做的)，则把整个查询分解为这种形式的一个有次序的步骤需没有不好形式的隐含。相反，查询可以按下面来进行：

- 冒号前面的表达式系统不能直接赋值 —— 系统所做的只是记住它们和被 AS语句引进的名称。

- 冒号后面的表达式代表了查询的最终结果（在这个例子中，表达式只是“ T 6 ”）。当运行到这里时，系统就会计算出想要的值（即T 6的值）。
- 为了计算T6（它是T5在SNAME上的投影），系统需要首先计算 T5；为了计算T5（T5是T4和S的连接），系统需先计算 T4；等等。换句话说，系统不得不计算原先的嵌套表达式，就像用户在前面写了嵌套表达式一样。

关于计算嵌套表达式这个问题，在下一节会有一个简要的讨论，在第17章有进一步的阐述。

6.6 关系代数的作用

总结一下本章中目前所学的内容：对关系代数作了定义，它是一个关系操作的集合。涉及的操作有并、交、差、积、选择、投影、连接和除，还有一个给属性改名的操作符：RENAME（除了RENAME，基本上就是Codd在[6.1]中最初定义的集合）。还提供了这些操作的语法，利用这些语法给出了许多例子和说明。

正如所讨论的，Codd的8个操作符不是一个最小的集合，因为它们中的许多不是基本的（primitive） —— 即不可以用其他的操作符来定义。例如，连接、交和除这三个操作符可以用其余的5个来定义（参看练习 6.2）。剩下的5个（选择、投影、积、并和差[⊖]）中的任意一个都不能用其他的 4个来定义，因此称它们为一个基本的或最小的集合（当然不只是这一个）。实际上，其余的 3个操作符（特别是连接）是非常有用的，一个好的系统可以直接支持它们。

现在需要澄清重要的一点。尽管没有明确说明，但本章的内容到目前为止好像显示了代数的主要作用只是数据存取。事实上并非如此。代数的基本目的是描述关系表达式。那些表达式有各种用途，当然也包括存取，但并不只限于此。下面的内容显示了几种用途：

- 定义检索的范围 —— 即定义在检索操作中获得的数据（就像已经详细讨论过的）；
- 定义修改的范围 —— 即定义在修改操作中被插入、修改或删除的数据（参考第 5章）；
- 定义完整性约束 —— 即定义数据库必须满足的一些约束（参考第 8章）；
- 定义导出的关系变量 —— 即定义包含在视图或映射中的数据（参考第 9章）；
- 定义一致性需求 —— 即定义在并发操作中的数据（参考第 15章）；
- 定义安全性约束 —— 即定义被授予某种权限的数据（参考第 16章）。

实际上，这些表达式是对用户意图的高层次的、符号式的表述（例如关于一些特殊的查询）。而正因为它们是高层次和符号形式的，它们可以根据一系列高层次和符号形式的转换规则来操作。例如，表达式

```
((SP JOIN S) WHERE P# = P#('P2')){SNAME}
```

（6.5节中的例1 —— “ 提供零件P2的供应商名称 ”）可以被转换为另一个等价的表达式，并且效率可能更高，即

```
((SP WHERE P# = P#('P2')) JOIN S){ SNAME }
```

（练习：从何种意义上说第二种表述效率可能更高？为何是“ 可能 ”？）

代数因此而可以作为优化的基础（如果想复习优化的概念，可回顾第 3章3.5节）。即使用

⊖ 因为积是连接的一个特殊情况，在基本列表中就可以用连接代替积。我们真正需要把 RENAME也加进去，因为我们的代数 (不像参考书 [6.1] 所定义的) 依赖于属性名称，而不是顺序位置。

户使用了上面的第一种表述方式，在执行前，优化系统会把它转换为第二种形式（理想的情况是，一个给定查询的执行不应依赖于用户提交的方式）。在第17章中将有进一步讨论。

代数由于它的基本特征而经常用来作为衡量一种语言表达能力的尺度。一种语言基本上可以说是关系完备的，当它至少拥有代数的作用——即当它的表达式允许每一个关系通过代数的形式来定义（前几节中描述的基本的代数）。在下一章将详细讨论关系完备的概念。

6.7 附加的操作符

自从Codd定义了他的8个操作符以来，很多人都提出了新的代数操作符。这一节中较为详细地定义几个操作符——SEMIJOIN、SEMIMINUS、EXTEND、SUMMARIZE和TCLOSE。根据Tutorial D的语法，这些操作符包含 <*nonproject*> 的5个新的形式，说明如下：

```
<semijoin>
    ::=   <relational expression>
                  SEMIJOIN <relational expression>

<semiminus>
    ::=   <relational expression>
                  SEMIMINUS <relational expression>

<extend>
    ::=   EXTEND <relational expression>
                 ADD <extend add commalist>

<extend add>
    ::=   <expression> AS <attribute name>

<summarize>
    ::=   SUMMARIZE <relational expression>
                PER <relational expression>
                ADD <summarize add commalist>

<summarize add>
    ::=   <summary type> [ ( <scalar expression> ) ]
                                          AS <attribute name>

<summary type>
    ::=   COUNT  | SUM  | AVG  | MAX | MIN | ALL | ANY
        | COUNTD | SUMD | AVGD

<tclose>
    ::=   TCLOSE <relational expression>
```

在前面的BNF产生式规则中提到的各种 <*relational expression*> 必须不是 <*nonproject*>。

1. 半连接

假定*A*、*B*、*X*、*Y*和Z如6.4节的“连接”小节中所示。于是*A*和*B*（按照先*A*后*B*次序）的半连接(semijoin)——*A* SEMIJOIN *B*——可以等价的定义为

```
(A JOIN B) { X, Y}
```

换句话说，*A*和*B*的半连接就是*A*和*B*连接在*A*的属性上的投影。结果的主体是（近似地讲）在*B*中有对应值的*A*的元组。

例：求提供零件P2的供应商的S#、SNAME、STATUS和CITY：

```
S SEMIJOIN (SP WHERE  P# = P# ( 'P2' ) )
```

2. 半差

假定*A*、*B*、*X*、*Y*和Z如6.4节的“连接”小节中所示。于是*A*和*B*的半差(semidifference)（按照先*A*后*B*次序）——SEMIDIFFERENCE *B*——可以等价地定义为

```
A MINUS (A SEMIJOINB)
```

结果的主体是（近似地讲）在*B*中没有对应值的*A*的元组。

例：求不提供零件P2的供应商的S#、SNAME、STATUS和CITY：

```
S SEMIMINUS (SP WHERE P# = P# ('P2'))
```

3. 扩展

读者可能已经注意到，到目前为止所描述的代数还没有计算的能力（用这样的词语容易理解）。然而，在实际中这样的能力是明显需要的。例如，我们可能想得到形如 WEIGHT*454的算术表达式的值，或者遇到WHERE子句中的这样一个值（给定的零件的重量是以磅为单位的；表达式WEIGHT*454会把它转换为克[⊖]）。扩展(extend)操作的目的是支持这样的适应性。更精确地说，EXTEND接受一个关系，然后返回一个关系，返回关系除了新增一个属性外，其余与给定关系完全相同，新增属性的值通过计算指定操作表达式而获得。例如，可以写：

```
EXTEND P ADD (WEIGHT*454) AS GMWT
```

请注意这是一个表达式而不是一个命令或者命题，因此它可以嵌套在其他的表达式中。它产生一个关系，此关系的表头除新增一个名叫 GMWT的属性外，其余和P相同。除了根据特定算术表达式得到的GMWT之外，关系的每个元组与P的元组相同。参看图6-9。

P#	PNAME	COLOR	WEIGHT	CITY	GMWT
P1	Nut	Red	12.0	London	5448.0
P2	Bolt	Green	17.0	Paris	7718.0
P3	Screw	Blue	17.0	Rome	7718.0
P4	Screw	Red	14.0	London	6356.0
P5	Cam	Blue	12.0	Paris	5448.0
P6	Cog	Red	19.0	London	8626.0

图6-9 一个关于EXTEND 的例子

重点：请注意，表达式 EXTEND并没有改变数据库中的零件这个关系变量；它只是一个表达式，就像S JOIN SP，并且像其他表达式一样只是产生了一个结果——在本例中恰巧看起来像零件关系变量的一个当前值（换句话说，EXTEND不是一个类似于 SQL的ALTER TABLE的关系代数）。

现在可以在投影和选择等操作中使用 GMWT。例如：

```
( ( EXTEND P ADD ( WEIGHT * 454 ) AS GMWT )
          WHERE GMWT > WEIGHT ( 10000.0 ) ) { ALL BUT GMWT }
```

注意：一个对用户比较友好的语言在 WHERE子句中当然会允许使用计算表达式，就像：

```
P WHERE ( WEIGHT * 454 ) > WEIGHT ( 10000.0 )
```

然而，这样的性能实际上只是适合造句法的好处。

总之，表达式

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

的值被定义为一个关系，此关系具有以下特性：（a）表头等于扩充了新属性 *Z*的关系*A*的表头；（b）主体包含所有的元组 *t*，*t*是*A*的一个扩充了新属性 *Z*的值的元组，这个值通过计算 *A*的元组上的表达式而得到。关系 *A*不能有名为*Z*的属性，并且*exp*不能涉及到*Z*，可以看出结果的基数等于*A*的基数，结果的度等于*A*的度加1。结果中*Z*的类型是*exp*的类型。

⊖ 在本例中，假设“*”是用于重量和整数之间的合法的操作符。这样一个操作的结果是什么呢？

下面是几个例子：

1) `EXTEND S ADD 'Supplier' AS TAG`

这个表达式利用字符串‘ Supplier ’标记了关系变量S当前值的每个元组（文字是计算表达式的一个特例，而文字更一般地是一个选择子调用）。

2) `EXTEND (P JOIN SP) ADD (WEIGHT * QTY) AS SHIPWT`

本例是对一个关系表达式的扩展，这比一个简单的关系变量更为复杂。

3) `(EXTEND S ADD CITY AS SCITY){ALL BUT CITY}`

形如CITY的一个属性名称也是一个合法的计算表达式。这个例子等价于

```
S RENAME CITY AS SCITY
```

换句话说，RENAME 不是最基本的——它可以用EXTEND（或投影）来定义。当然，由于使用方便的缘故，我们并不想放弃所熟悉的RENAME操作符，但它只是一个简写。

4) `EXTEND P ADD WEIGHT * 454 AS GMWT, WEIGHT * 16 AS OZWT`

这是多扩展的例子。

5) `EXTEND S`

```
ADD COUNT ((SP RENAME S# AS SX) WHERE X = S#)
AS NP
```

这个表达式的结果在图6-10中显示。解释：

S#	SNAME	STATUS	CITY	NP
S1	Smith	20	London	6
S2	Jones	10	Paris	2
S3	Blake	30	Paris	1
S4	Clark	20	London	3
S5	Adams	30	Athens	0

图 6-10

- 对于一个在关系变量当前值中给定的供应商元组，表达式

  `((SP RENAME S# AS SX) WHERE X = S#)`

  对应于关系变量SP当前值的供应商元组，产生了发货元组的集合。
- 聚集操作符(aggregate operator)COUNT应用于发货元组的集合，返回相应的基数（当然是一个数值）。

  结果中的属性NP代表供应商提供的零件的数量，供应商由相应的 S#的值指定。特别注意对应于供应商S5的NP的值；S5的SP元组的集合是空的，因此COUNT返回零。

  对聚集操作符的问题做一下说明。一般地讲，这样一个操作符的目的是从特定关系的特定属性的值中得出一个数字。典型的例子是 COUNT、SUM、AVG、MAX、MIN、ALL和ANY。在Tutorial D中，一个<*aggregate operator invocation*>（因为它返回一个数字，所以是<*scalar expression*>的一个特例）采用下面的一般形式

  `<op name> ( <relational expression> [, <attribute name> ] )`

  如果是COUNT，则<*attribute name*>是无关的，且必须省略掉；否则，当且仅当<*relationalexpression*>是一个单目的关系，<*attribute name*>才被省略掉，在这种情况下，<*relational expression*>的值的唯一属性假定为缺省值。下面是两个例子：

```
SUM ( SP WHERE S# = S# ( 'S1' ), QTY )
SUM ( ( SP WHERE S# = S# ( 'S1' ) ) { QTY } )
```

注意两者的区别，第一个得出供应商S1的所有发货数量的总计，第二个得出S1的所有不同的发货数量的和。

如果一个聚集操作符的参数恰好是一个空集，则COUNT返回零（就像所看到的），SUM也一样；MAX和MIN分别返回相关域的最小值和最大值； ALL和ANY分别返回ture和false；AVG出现一个异常。

4. 合计

首先说明这里讨论的SUMMARIZE不同于本书前期版本中所讲的，实际上，它是一个提高版，它克服了由于关系空值所引起的问题。

我们已经看到， extend操作符提供了一种途径，把“水平（ horizontal）”或“行方式（row-wise）”计算结合进了关系代数。 summarize操作符执行类似的“垂直（ vertical）”或“列方式（column-wise）”计算。例如，表达式

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

P#	TOTQTY
P1	600
P2	1000
P3	400
P4	500
P5	500
P6	100

图6-11 关于SUMMARIZE 的一个例子

产生了一个表头为{ P# ，TOTQTY }的关系，它对应于投影SP {P#}的每个P#值有一个元组，内含P#的值和相应的合计数量（看图6-11）。换言之，关系SP从概念上被分成元组的集合，具体为对应P{P#}的每个P#的值产生一个集合，然后每组产生一个结果元组。

CITY	NSP
London	5
Paris	6
Rome	1

一般地说，表达式

```
SUMMARIZE A  PER  B  ADD  SUMMARY  ASZ
```

的值定义如下：

- 首先，*B*必须和*A*的某些投影是相同的类型；即*B*的所有属性必须都是*A*的属性。假设投影的属性（对*B*来说是相同的）是*A1*、*A2*、…．、*An*。
- 结果的表头包含属性*A1*、*A2*、…．、*An*，并加上新属性*Z*。
- 结果的主体包含所有的元组*t*，其中*t*是经过扩展的*B*的一个元组，该扩展使其增加了新属性*Z*上的一个值。*Z*的新值是通过计算*A*元组上的合计而得到的，这些元组在*A1*、*A2*、…．、*An*上与元组*t*有相同的值（当然，如果A没有元组和*t*在*A1*、*A2*、…、*An*上有相同的值，则合计会在一个空集合上操作）。关系*B*不能有名为*Z*的属性，并且合计不能涉及*Z*。于是，结果和*B*有相同的基数，结果的度等于*B*的度加1。结果中*Z*的类型和合计的类型相同。

下面是另一个例子：

```
SUMMARIZE ( P JOIN SP ) PER P {CITY } ADD COUNT AS NSP
```

结果看起来像：

CITY	NSP
London	5
Paris	6
Rome	1

也就是说，结果对应于每个城市（London、Paris和Rome）有一个元组，显示了每个城市中存放的零件的数量。

要点：

1) 我们的语法允许多重SUMMARIZE。例如：

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

2) <*summarize*>的一般格式如下：

```
SUMMARIZE <relational expression>
          PER <relational expression>
                    ADD <summarize add commalist>
```

其中每个<*summarize add*>形为：

```
<summary type> [ ( <scalar expression> ) ] AS <attribute name>
```

典型的<*summariy type*>有COUNT、SUM、AVG、MAX、MIN、ALL、ANY、COUNTD、SUMD和AVGD。在COUNTD、SUMD和AVGD中“D”(distinct）的意思是“执行summary之前去掉多余的重复值”。<*scalar expression*>可以涉及到紧跟在SUMMARIZE后的<*relational expression*>所表示的关系的属性。注意：<*scalar expression*>（在圆括号中）仅当<*summary type*>是COUNT时才能被省略。

顺便提一下，请注意<*summarize add*>不同于<*aggregate operator invocation*>。<*aggregate operator invocation*>是一个标量表达式，只要标量操作符被调用（特殊情况下为一个标量字串），它就能出现。相反，<*summarize add*>只是一个SUMMARIZE的操作对象；却不是一个标量表达式。脱离开SUMMARIZE的上下文环境，它就失去意义，实际上根本不能出现。

3) 读者可能已意识到，SUMMARIZE不是一个基本操作符——它可通过EXTEND来表达。例如，表达式

```
SUMMARIZE SP PER S { S# } ADD COUNT AS NP
```

是下面表达式的简写：

```
( EXTEND S { S# }
         ADD ( ( SP RENAME S# AS X ) WHERE X = S# ) AS Y,
             COUNT ( Y ) AS NP )
{ S#, NP }
```

或等价于：

```
WITH ( S { S# } ) AS T1,
     ( SP RENAME S# AS X ) AS T2,
     ( EXTEND T1 ADD ( T2 WHERE X = S# ) AS Y ) AS T3,
     ( EXTEND T3 ADD COUNT ( Y ) AS NP ) AS T4 :
T4 { S#, NP }
```

4) 考虑下面的例子：

```
SUMMARIZE SP PER SP {} ADD SUM (QTY) AS GRANDTOTAL
```

在本例中，分组和求和所根据的关系根本就没有属性。假设*sp*是关系变量SP的当前值，并且*sp*至少有一个元组。于是对应于没有属性的情况，*sp*的所有元组有相同的值——即

0元组[5.5]；因此只有一组，并且在结果中只有一个元组。也就是说，聚集计算在整个关系*sp*上运行了一次。SUMMARIZE产生的关系只有一个属性和一个元组；属性叫作总计（GRANDTOTAL），结果元组中唯一的标量值是关系*sp*中所有QTY的和。

如果关系*sp*根本就没有元组，于是就没有分成的组，也就没有结果元组——即结果关系是空的。对比一下，接下来的表达式[⊖]——

```
SUMMARIZE SP PER RELATION { TUPLE { } }
          ADD SUM ( QTY ) AS GRANDTOTAL
```

——即使*sp*是空的也会运行（即它会产生零这个“正确”结果）。更精确地讲，它会返回只有一个名叫总计属性的关系，同时关系只有一个在属性总计上值为零的元组。因此可以从SUMMARIZE中删去PER子句，如下：

```
SUMMARIZE SP ADD SUM ( QTY ) AS GRANDTOTAL
```

省略PER子句定义为等价于指定如下形式的 PER 子句

```
PER RELATION { TUPLE { } }
```

5. Tclose

“Tclose”代表传递闭包。这里提到它主要是为了全面；详细内容已超出了本章的范围。接下来只是定义操作。设*A*是包含属性*X*和*Y*的两目关系，其中*X*、*Y*具有相同的类型*T*。于是*A*的传递闭包——TCLOSE *A*——是一个关系A^+，A^+的表头和*A*的相同，主体是*A*的主体的一个超集，定义如下：当且仅当元组{*X*：*x*，*Y*：*y*}出现在*A*中，或存在值*z1*,*z2*,...,*zn*（类型都是*T*）的一个序列，使元组{*X*：*x*，*Y*：*z1*}、{*X*：*z1*，*Y*：*z2*}、...、{*X*：*zn*，*Y*：*y*}都出现在*A*中，元组{*X*：*x*，*Y*：*y*}才属于A^+（也就是说，只有当关系*A*所表示的图中存在从*x*到*y*的一条路径时，元组（*x*，*y*）才会出现，这只是不严格的说法。注意，A^+的主体必须包含*A*的主体，把它作为一个子集）。

第23章会有关于这个主题的更详细的讨论。

6.8 分组与分组还原

因为关系的属性的值可以是关系值，这就需要增加关系型操作符，分别称之为分组(group)和分组还原(ungroup)。首先给出一个关于group的例子：

```
SP  GROUP (P#, QTY)  AS  PQ
```

给定例子中常用的数据，此表达式产生的结果如图 6-12所示。注意：下面的解释不可避免地有些抽象，参照图去看解释将会有所帮助。

基本表达式

```
SP  GROUP  (P#, QTY )  AS  PQ
```

有可能被看作“group SP by S#”，因为S#是唯一没有在GROUP子句中提到的属性。结果是定义如下的一个关系。首先，表头形式为：

```
{ S# S#, PQ RELATION { P# P#, QTY QTY } }
```

他包含了一个值为关系的属性 PQ（其中PQ含有属性P#和QTY），并且包含除 SP外的所有其他属性（当然，除 SP外的所有属性只有 S#）。其次，主体对于 SP中每个不同的S#的值对应一个元组。主体中的每个元组包含一个 S#的值（如*s*）和一个PQ的值（如*pq*），其中PQ的值如

[⊖] 本例的PER子句中，表达式RELATION{TUPLE{}}代表了一个关系，这个关系没有属性，只有一个元组 (即0元组)。它可以简写为TABLE_DEE(参考[3.3]、[5.5]和[6.2])。

下获得：

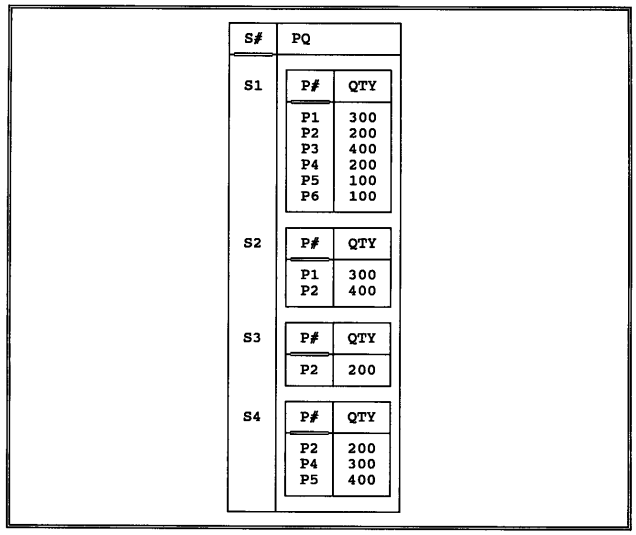

图6-12 按照S#对SP分组

- 每个SP元组在概念上被一个元组（如 *x*）代替，此元组中 P#和QTY被包装（wrapped）成一个元组值（tuple-value）的分量（如*y*）。
- 所有属性S#上值为*s*的元组*x*中，所对应的分量*y*组合成了一个关系（*pq*）。这样，一个结果元组就产生了，其中S#的值是*s*，PQ的值是*pq*。

上面的结果实际上已经在图6-12中显示了。

现在看一下分组还原。假设SPQ是图6-12所示的关系。于是表达式

```
SPQ  UNGROUP  PQ
```

（或许是意料之中）返回了常用的关系SP。更具体的是，它产生了一个定义如下的关系。首先，表头形式为：

```
{ S# S#, P# P#, QTY QTY }
```

表头包含属性P#和QTY（从属性PQ分离出），同时包含SPQ的所有其他属性（在本例中只有S#）。其次，SPQ中一个元组和在此元组中PQ值的一个元组构成一个组合，对于每个组合，主体对应一个元组。主体的每个元组由S#的值（如*s*）和P#、QTY的值（如*p*和*q*）组成，通过以下方式得到：

- 一个元组的集合代替了一个SPQ元组，对应SPQ元组的PQ值中的每个元组产生这样一个元组（如*x*）。元组*x*包含两部分，一部分是等于SPQ 元组中S#的值（如*s*），另一部分等于SPQ元组中PQ分量的某个元组的值（如*y*）。
- 属性S#上值为*s*的元组*x*中，分量*y*被分解成P#和QTY（如*p*和*q*）。于是产生了这样的一个元组，它在属性S#上值为*s*，P#上值为*p*，QTY上值为*q*。整个的结果就是关系SP。

GROUP和UNGROUP一起提供了我们通常所说的关系的嵌套（nest）和非嵌套(unnest)功能。我们更倾向于用术语group/ungroup，因为术语nest/unnest和NF^2关系的概念有着紧密联系，而这个概念比较容易混淆，所以我们不赞成。

为求全面，本节最后谈一下GROUP和UNGROUP操作的可逆性（初次阅读可能会有一定

的困难）。如果通过某种途径对关系*r*分组，总会有一个可逆的撤消分组操作使我们重新得到*r*。然而，如果对某个关系*r*撤消了分组，一个可逆的分组就可能不存在。下面举一个例子（在参考书[5.4]的一个例子的基础上）。假设有关系TWO（如图6-13），对它进行撤消分组操作得到THREE。如果现在对THREE按照A分组（并且把产生的关系型属性也命名为RVX），得到的是ONE而不是TWO。

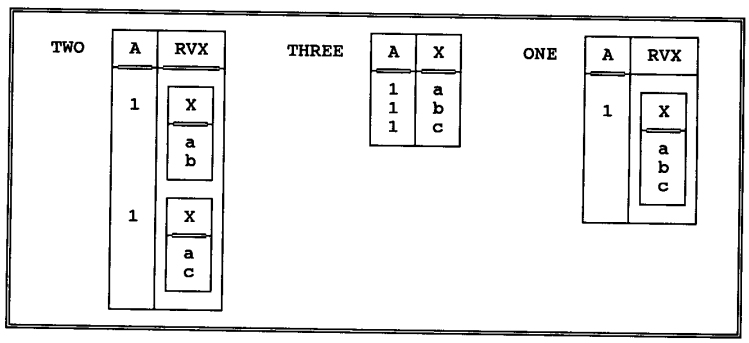

图6-13 撤消分组和（重新）分组不一定是可逆的

注意，在ONE中，RVX函数依赖（functionally dependent）于A，构成了一个侯选码（参看第8、10章）。如果现在对ONE撤消分组，会得到THREE，并且THREE可以分组成ONE；因此，对这一对特殊的关系来说，分组和撤消分组是可逆的。通常，函数依赖是决定一个撤消分组是否可逆的重要因素。事实上，如果关系*r*有一个关系型的属性RVX，则说*r*是撤消分组可逆的，当且仅当满足下面两个条件：

- *r*的任一元组在RVX上不是一个空关系。
- RVX函数依赖于*r*的所有其他属性组成的一组属性。也就是说，*r*必须存在一个不包含RVX的侯选码。

6.9 关系比较

最初定义的关系代数并没有直接给出比较两个关系值的途径——例如，判断它们是否相等，判断出现在一个关系中的每个元组是否也出现在另一个关系中（不严格地说，即判断一个关系是否是另一个关系的子集）。因此而导致很难表述某些查询（本章后面的练习6.48就是一个例子）。然而这个缺陷很容易弥补。首先，定义一种新的比较，即关系比较，语法如下：

```
<relational expression>Θ<relational expression>
```

由两个*<relational expression>*表示的关系必须是同一类型。比较操作符Θ可以是以下符号之一：

= 相等（equals）
≠ 不等（not equals）
≤ 子集（subset of）
< 真子集（proper subset of）
≥ 超集（superset of）
> 真超集（proper superset of）

注意：对操作符要小心选择，因为对“*A*是*B*的一个真子集”的否定并不意味着“*A*是*B*的

一个超集”（即“＜”和“≥”不是互补的）。然而，由于本书中所需，我们还是要利用这些符号。

下面是两个例子：

1) S{ CITY } = P{ CITY }

意思是：供应商关系在CITY上的投影和零件关系在CITY上的投影相同吗？

2) S{ S# } ⊃ SP{ S# }

意思是（意译）：有不提供零件的供应商吗？

下面，在关系表达式中使用这种新的比较。例如：

```
S WHERE ( ( SP RENAME S# AS X ) WHERE X = S# ) { P# } = P { P# }
```

这个表达式得到的关系包含供应商关系中提供所有零件的供应商元组。解释：

- 对于一个给定的供应商，表达式

```
( ( SP RENAME S# AS X ) WHERE X = S# ) { P# }
```

产生这个供应商提供的零件号的集合。

- 然后，这个零件号的集合和包含所有零件号的集合进行比较。如果这两个集合相等，对应的供应商元组就会出现在结果里。

当然，我们已经知道了怎样利用DIVIDEBY来表述这个查询：

```
S JOIN ( S { S# } DIVIDEBY P { P# } PER SP { S#, P# } )
```

然而，你可能已感觉到用关系比较来表述比较容易理解。有一点必须澄清：关系比较并不是6.4节所讲的选择条件，上面所示的包含这样一个比较的例子不是真正的选择！它是下面语句的一个简写：

```
WITH ( EXTEND S
       ADD ( ( SP RENAME S# AS X ) WHERE X = S# ) { P# }
       AS A ) AS T1,
     ( EXTEND T1
       ADD P { P# }
       AS B ) AS T2 :
T2 WHERE A = B
```

这里的A和B是关系型的属性，最后的表达式T2 WHERE A=B是一个真正的选择。注意：若要支持关系比较需要首先支持关系型属性。

实际中经常需要的一个关系比较是判断一个给定的关系是否为空（即不包含元组）。引进一个简便的表述法是有必要的。为此引进一个判断真假的操作符，形式为：

```
IS_EMPTY (<relational expression>)
```

如果*<relational expression>*表示的关系为空，则返回真（true）；否则返回假（false）。

另一个一般的要求是判断元组*t*是否包含在关系*r*中。下面的关系比较能够满足要求：

```
RELATION  {t } ≤ r
```

然而，下面的简写方式对用户更友好，如果熟悉SQL，你就会对它非常熟悉：

```
t  IN  r
```

这里的IN实际上是一个集合操作符，通常代表 ∈。

6.10 小结

我们已经讨论了关系代数。首先重新强调了封闭性（closure）和嵌套关系表达式（nested relational expression）的重要性，并解释了如果要严格保持封闭性，则要有一套关系类型参照

规则（relation type inference rule）（当然我们所讲的代数没有包含这样的规则）。

最初的代数包含8个操作符——传统的集合操作并、交、差和积（对它们只是做了稍微的修改以适应操作对象是关系的操作），和专门的关系操作选择、投影、连接和除。在此基础上又加了RENAME、SEMIJOIN、SEMIMINUS、EXTEND和SUMMARIZE（也提到了TCLOSE，并简单地讨论了GROUP和UNGROUP）。对于某些操作符，参与操作的两个关系需要类型相同（以前称为并相容性）。同时指出了这些操作符并不都是基本的——它们中的几个可以通过其他的来表示。展示了怎样把操作符结合起来执行一系列目的操作：查询、更新等。也简要说明了表达式优化的思想（但详细内容将在第17章讲解）。考虑了用分步(step-at-a-time)方法处理复杂查询的可能性，利用WITH为表达式引进名称。最后讨论了关系比较的思想，它能使某些查询更易于表达。

练习

6.1 本章中，我们声称并、交、积和（自然）连接都具有交互性和结合性。证明之。

6.2 在Codd最初定义的8个操作符中，并、差、积、选择和投影可以被认为是基本的。试用这5种基本操作来表示自然连接、交和除。

6.3 如果*A*和*B* 没有共同的属性，则*A* JOIN *B*等价于*A* TIMES *B*。对其进行证明。如果*A*和*B*有相同的表头，则上述表达式等价于什么？

6.4 证明练习6.2中提到的5个基本操作符是基本的（证明任意一个不能被其余4个来表示）。

6.5 算术里的乘和除是两个互逆的操作。关系代数中的TIMES和DIVIDEBY是互逆操作吗？

6.6 给定前面的供应商和零件数据库，表达式 S JOIN SP JOIN P的值是什么？

6.7 假设*A*是一个*n*目的关系。*A*有多少个不同的投影？

6.8 算术中特殊的数字1使对任意数字*n*都有：`n * 1 = 1 * n = n`。称1对于乘法具有同一性（identity）。在关系代数中存在类似角色的关系吗？如果有，是什么？

6.9 算术中还有另外一个特殊的数字0，它使：对任意的数字*n*有`n * 0 = 0 * n = 0`。在关系代数中存在类似角色的关系吗？如果有，是什么？

6.10 考虑本章所讲的代数操作符作用于满足上面两个练习要求的关系上会有何种结果。

6.11 在6.2节讲到，由于和算术中封闭性的重要性的同样原因，关系封闭性是很重要的。然而算术里有一个封闭性被打破的情况——即被零除。关系代数中有类似的情况吗？

6.12 并、交、积和连接最初都被定义为二元（dyadic）操作符（即每个都有两个操作对象）。在本章中显示了怎样把它们扩展到*n*元操作符（任意*n*>1）；例如，*A* UNION *B* UNION *C*可看作是*A*、*B*和*C*的三目并。但当*n*=1或*n*=0时会怎样呢？

查询练习

下面的练习都以供应商-零件-工程数据库为基础（参看第4章练习中的图4-5和第5章练习5.4的答案）。每个例子要求你为某个查询写出关系代数表达式（由于个人不同的爱好，你可能倾向于先看一些答案，然后用自然语言陈述出给定表达式的意思）。为了方便，下面列出数据库的结构：

```
S     { S#, SNAME, STATUS, CITY }
      PRIMARY KEY { S# }
P     { P#, PNAME, COLOR, WEIGHT, CITY }
      PRIMARY KEY { P# }
```

```
J     { J#, JNAME, CITY }
      PRIMARY KEY { J# }
SPJ   { S#, P#, J#, QTY }
      PRIMARY KEY { S#, P#, J# }
      FOREIGN KEY { S# } REFERENCES S
      FOREIGN KEY { P# } REFERENCES P
      FOREIGN KEY { J# } REFERENCES J
```

6.13 求所有有关工程的信息。

6.14 求在伦敦的所有工程的信息。

6.15 求为工程J1提供零件的供应商的号码。

6.16 求数量在300~750之间的发货。

6.17 求所有的零件颜色 / 城市对。注意：这里及以后所说的“ 所有 ”特指在数据库中。

6.18 求所有的供应商号/零件号/ 工程号三元组。其中所指的供应商、零件和工程在同一个城市。

6.19 求所有的供应商号/零件号/ 工程号三元组。其中所指的供应商、零件和工程不在同一个城市。

6.20 求所有的供应商号/零件号/ 工程号三元组。其中所指的供应商、零件和工程三者中的任意两个都不在同一个城市。

6.21 求由伦敦供应商提供的零件的信息。

6.22 求由伦敦供应商为伦敦工程供应的零件号。

6.23 求满足下面要求的城市对，要求在第一个城市的供应商为第二个城市的工程供应零件。

6.24 求供应商为工程供应的零件号，要求供应商和工程在同一城市。

6.25 求至少被一个不在同一城市的供应商供应零件的工程号。

6.26 求由同一个供应商供应的零件号的对。

6.27 求所有由供应商S1供应的工程号。

6.28 求供应商S1供应的零件P1的总量。

6.29 对每个供应给工程的零件，求零件号、工程号和相应的总量。

6.30 求为单个工程供应的零件数量超过350的零件号。

6.31 求由S1供应的工程名称。

6.32 求由S1供应的零件颜色。

6.33 求供应给伦敦工程的零件号。

6.34 求使用了S1供应的零件的工程号。

6.35 设供应了红色零件的供应商为S2，求供应了至少一个S2供应的零件的供应商号。

6.36 求status比S1低的供应商号。

6.37 求所在城市按字母排序为第一的工程号。

6.38 求被供应零件P1的平均数量大于供应给工程J1的任意零件的最大数量的工程号。

6.39 求满足下面要求的供应商号，该供应商供应给某个工程零件 P1的数量大于供应给这个工程的零件P1的平均数量。

6.40 求没有被伦敦供应商供应过红色零件的工程号。

6.41 求所用零件全被S1供应的工程号。

6.42 求所有伦敦工程都使用的零件号。

6.43 求对所有工程都提供了同一零件的供应商号。

6.44 求使用了S1提供的所有零件的工程号。

6.45 求至少有一个供应商、零件或工程所在的城市。

6.46 求被伦敦供应商供应或被伦敦工程使用的零件号。

6.47 求所有供应商号/零件号对，其中指定的供应商不供应指定的零件。

6.48 求供应商号码对（如S*x*和S*y*），其中S*x*和S*y*供应的零件都相同。

6.49 按供应商号/零件号对SPJ分组，相应的工程号和数量形成二目关系。

6.50 对上题所得的关系撤消分组。

参考文献和简介

6.1 E. F. Codd: “Relational Completeness of Data Base Sublanguages,” in Randall J. Rustin (ed.), *Data Base Systems, Courant Computer Science Symposia Series 6*. Englewood Cliffs, N.J.: PrenticeHall (1972).

这是Codd首次正式定义最初（original）代数操作符的文章（当然，[5.1]中也有定义，但那既不正式也不全面）。

这里面有一个可能的缺陷，他声称为了表述和解释的方便，假设关系的属性有一个从左到右的次序，并且可以通过它们的位置来存取（尽管 Codd也说明了“ 实际中存取信息的时候，应该使用名称而不是位置号 ”）。文章因此没有提到属性改名的操作符（RENAME），并且没有考虑到结果的类型。可能由于省略这些的原因，在关于代数的许多讨论中，在现在的SQL产品乃至SQL标准中，都有同样的批评。

在第7章特别是7.4节里面有关于这篇文章的更多评论。注意：参考书 [3.3]描述了一种名为A的简化的代数集合，它允许用更少的基本操作符定义出功能更强大的操作符。事实上，[3.3]显示了利用两个基本的操作符 remove和nor完成Codd的代数操作符所能完成的所有功能。

6.2 Hugh Darwen (writing as Andrew Warden): “Adventures in Relationland,” in C. J. Date, *Relational Database Writings 1985-1989*. Reading, Mass.: Addison-Wesley (1990).

一系列内容新颖、有趣、并具有知识性的短文描述了关系模型和关系DBMS。它们是：

1) The Naming of Columns
2) In Praise of Marriage
3) The Kdysof the Kingdom
4) Chivalry
5) A Constant Friend
6) Table_Dee and Table_Dum
7) Into the Unknown

6.3 Hugh Darwen and C. J. Date: “Into the Great Divide,” in C. J. Date and Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

这篇文章（ a）分析了 Codd在[6.1]中定义的除，（b）分析了 Hall、Hitchcock和Todd[6.10]关于除的概括，它不同于 Codd的除，允许关系可以被任意的关系来除（ Codd的除中，除数关系的表头必须是被除数关系表头的子集）。这篇论文说明了两种除在处理空关系时都遇到了困难，所得的结果都不能处理想要解决的问题（即两者都不是想要的

全称量词）。对它们的改进（即小除small divide和大除great divide）克服了上述问题。注意：按照Tutorial D语法，两者是不同的操作符，即大除不是小除的一个扩充。该论文同时显示了修改的操作符和除的名字有些名不符实！关于这一点，参考练习 6.5。

在这里给出Codd对除的定义。设关系*A*和*B*分别有表头{*X*，*Y*}和{*Y*}（其中*X*和*Y*可以是合成的）。于是*A* DIVIDEBY *B*产生了一个关系，关系的表头是{*X*}，主体包含所有满足下述条件的元组{*X*，*x*}，即对任意出现在*B*中的元组{*Y*，*y*}，*A*都有一个元组{*X*，*x*；*Y*，*y*}存在。换一种不严格的说法，即若*X*在*A*中对应的*Y*的值包含*B*中所有的*Y*值，则X出现在结果中。

6.4 C. J. Date: “Quota Queries” (In three parts), in C. J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994-1997*. Reading, Mass.: Addison-Wesley (1998).

限额查询就是指定结果中基数的查询——例如，查询“求三个最重的零件”。下面是这个例子的Tutorial D语法表示。

```
P QUOTA ( 3, DESC WEIGHT )
```

它是下面表达式的简写：

```
( ( EXTEND P
    ADD COUNT ( ( P RENAME WEIGHT AS WT ) WHERE WT > WEIGHT )
    AS #_HEAVIER )
  WHERE #_HEAVIER < 3 ) { ALL BUT #_HEAVIER }
```

（其中WX、WY和#_HEAVIER是任意的）。给定我们常用的数据，则结果包含零件 P1、P2和P6。

这篇论文深入地分析了限额查询的需求，并提出了用来处理和描述问题的几种简写。

6.5 Michacl J. Carey and Donald Kossmann: “On Saying ‘Enough Already!’ in SQL,” Proc. 1997 Int. Conf. on Management of Data, Tucson, Ariz. (May 1997).

这是又一篇关于限额查询的文章。与 [6.4]不同的是，它侧重于实现而不是模型。查询“求三个最重的零件”按照本文会通过以下方式来实现：

```
SELECT *
FROM   P
ORDER  BY WEIGHT DESC
STOP   AFTER 3 ;
```

这种方法的一个问题是：STOP AFTER应用于ORDER BY的结果，而这个结果（同第5章5.3节“关系的特性”）是一列却不是一个关系；因此，总的结果就不是一个关系，关系封闭性被打破了。该文章没有讨论这个问题。

当然，结果或许可以转换为关系——但由此引出另一个问题：一般情况下， STOP AFTER的结果是不可预测的。例如，给定我们常用的数据，如果在前面的查询中把 STOP AFTER 3改为STOP AFTER 2，则结果不确定。

STOP AFTER子句是由IBM研究中心提出的，有可能应用到IBM产品中，并可能由此而成为SQL标准——尽管在上述问题被解决之前人们不希望如此。

6.6 R. C. Goldstein and A. J. Strnad: “The MacAIMS Data Management System,” Proc. 1970 ACM SICFIDET Workshop on Data Description and Access (November 1970).

参看下面[6.7]中的注解。

6.7 A. J. Strnad: “ The Relational Approach to the Management of Data Bases,” Proc. IFIP Congress, Ljubljana, Yugoslavia (August 1971).

由于历史的缘故我们提到 MacAIMS；它可能是最早支持 *n*目关系和代数语言的系统。有意思的是，它和 Codd在关系模型上的工作是并行发展的，至少部分是无关的。与 Codd 的工作不同的是，MacAIMS的努力没有引来人们在这方面的继续性工作。

6.8 M. G. Notley: "The Peterlee IS/1 System, " IBM UK Scientific Centre Report UKSC-0018(March 1972).

参看[6.9]的注解。

6.9 S. J. P. Todd: " The Peterlee Relational Test Vehicle —— A System Overview," *IBM Sys. J. 15*, No.4 (1976).-

PRTV（The Peterlee Relational Test Vehicle）是IBM位于英国peterlee的UK研究中心开发的实验系统。它建立在较早的原型 IS/1的基础之上 —— 可能正是IS/1首次实现了Codd的思想。它支持*n*目关系和叫做ISBL（Information System Base Language）的一种代数语言，这种语言以[6.10]中列出的提案为基础。本章所讲的关于关系类型的思想可以追溯到ISBL和[6.10]的提案。

PRTV中比较重要的方面如下：

- 支持RENAME、EXTEND和SUMMARIZE。
- 吸收了复杂表达式转换的技术（参看第 17章）。
- 具有一种惰性计算特征 (lazy evaluation feature)，这对优化和视图支持是很重要的（参看本章关于WITH的讨论）。
- 提供可扩展功能 —— 即赋予用户定义自己的操作符的权利。

6.10 P. A. V. Hall, P. Hitchcock, and S. J. P. Todd: "An Algebra of Relations for Machine Computation," Conf. Record of the 2nd ACM Symposium on Principles of Programming Languages, Palo Alto, Calif. (January 1975).

6.11 Patrick A. V. Hall: "Relational Algebras, Logic, and Functional Programming," Proc. 1984 ACM SIGMOD Int. Conf. on Management of Data, Boston, Mass. (June 1984).

提供了对关系代数的功能性解释，目的是：(a) 为4GLs准备理论基础（第 2章）；(b) 结合功能的、逻辑性的关系语言，以使它们也能分享实现技术。作者声称，尽管逻辑设计和数据库已经逐渐地相互靠近，但是在写书的时候，功能或应用的语言很少涉及到数据库的事情。因此该文也算是对两者结合的一点贡献。

6.12 Anthony Klug: "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions, " *JACM 29*, No.3 (July 1982).

对关系代数和关系演算做了扩展（第 7章）以支持聚集操作，并证明了两者是等价的。

部分练习答案

注意：对练习6.13~6.50给出的答案不一定是唯一的。

6.2 JOIN在6.4节已经讨论过了。INTERSECT可以定义如下：

```
A INTERSECT B  =  A MINUS ( A MINUS B )
```

或（等价）

```
A INTERSECT B  =  B MINUS ( B MINUS A )
```

这两个式子尽管是正确的，但并不尽人意。因为 *A* INTERSECT *B*关于*A*和*B*是对称的，但其余两个表达式不对称。这里给出一个对称的等价式以作比较：

```
( A MINUS ( A MINUS B ) ) UNION ( B MINUS ( B MINUS A ) )
```

注意：*A*和*B*必须是同一类型。也可以是：

```
A INTERSECT B  ≡  A JOIN B
```

至于DIVIDEBY，有：

```
A DIVIDEBY B PER C  ≡  A { X }
                       MINUS ( ( A { X } TIMES B { Y } )
                                  MINUS C { X, Y } ) { X }
```

这里*X*是*A*和*C*共有的属性，*Y*是*B*和*C*共有的属性。

注意：刚才定义的 DIVIDEBY是本章中所作的定义的一般形式 —— 尽管它仍是一个小除。在本章中假设了 *A*除*X*外没有别的属性，*B*除*Y*外没有别的属性，*C*除*X*和*Y*之外没有别的属性。上面的一般形式可以使表述变得更简单，例如查询“求供应全部零件的供应商号码”可以写为

```
S DIVIDEBY P PER SP
```

而不是

```
S { S# } DIVIDEBY P { P# } PER SP { S#, P# }
```

6.3 *A* INTERSECT *B*（参看练习6.2的答案）。注意：因为TIMES是JOIN 的一个特例，所以可以用JOIN代替TIMES作为一个基本操作符。

6.4 只给出一个简略的非形式化的证明。

- 积是唯一能使属性数目增加的操作符，它不能用别的操作符来表达。所以积是基本的。
- 投影是唯一能减少属性数目的操作符，它不能用别的操作符来表达。所以积是基本的。
- 并是除积以外唯一能增加元组数目的操作符，积通常也增加属性的数目。假设进行并操作的两个关系是*A*和*B*。注意*A*和*B*必须是同一类型，并且它们的并和它们有相同的属性。为了实现*A*和*B*的积，如果首先改掉 *B*的所有属性的名称，然后利用投影减少结果中的属性，使只剩下*A*的属性，结果是又回到了 *A*：

```
( A TIMES B ) { all attributes of A }  =  A
```

（除非*B*恰巧是空的；为简单起见忽略这种情况）。因此积不能表达并，并是基本的。

- 差不能通过并（因为并永远不能减少元组的数目）或积（同样的原因）或投影（因为投影通常减少属性）来表达。它也不能利用选择来表达，因为第二个关系中的值对差很重要，选择中却不然。因此差是基本的。
- 选择是唯一允许属性上的值相互比较的操作符。因此选择是基本的。

6.5 不是。Codd定义的DIVIDEBY满足下面这个特性：

```
( A TIMES B ) DIVIDEBY B  =  A
```

然而：

- Codd定义的DIVIDEBY有两个操作对象；我们的 DIVIDEBY需要三个操作对象，因此不可能满足一个类似的特性。
- 很多情况下，*A*除以*B*，然后把结果同 *B*作积，会产生和 *A*一样的关系，但更多的情况是产生*A*的真子集：

```
( A DIVIDEBY B ) TIMES B  ≤  A
```

Codd的DIVIDEBY更像是算术中的整数除法（即不重视余数）。

6.6 这里的陷阱在于，连接不仅涉及S#和P#，还涉及CITY属性。其结果如下：

S#	SNAME	STATUS	CITY	P#	QTY	PNAME	COLOR	WEIGHT
S1	Smith	20	London	P1	300	Nut	Red	12.0
S1	Smith	20	London	P4	200	Screw	Red	14.0
S1	Smith	20	London	P6	100	Cog	Red	19.0
S2	Jones	10	Paris	P2	400	Bolt	Green	17.0
S3	Blake	30	Paris	P2	200	Bolt	Green	17.0
S4	Clark	20	London	P4	200	Screw	Red	14.0

6.7 2^n（2的n次方）。这个计算包括了全（identity）投影（即在所有n个属性上的投影）和空（nullary）投影（在0个属性上的投影）。全投影产生的结果等同于关系A；空投影在关系A为空时产生TABLE_DUM，否则产生TABLE_DEE[5.5]。

6.8 存在这样的一个关系，即TABLE_DEE。TABLE_DEE（简写为DEE）类似于算术乘法中的1，因为

```
R TIMES DEE  ≡  DEE TIMES R  ≡  R
```

对所有的关系都成立。换言之，DEE和TIMES相同（更一般地讲是和JOIN）。

6.9 对于TIMES来说，没有关系扮演乘法中零的角色。然而，TABLE_DUM的作用多少类似于零的作用，因为

```
R TIMES DUM  ≡  DUM TIMES R  ≡  an empty relation with
                                 the same heading as R
```

对任意的关系R都成立。

6.10 首先，应该注意到与DEE和DUM类型相同的关系只有它们自己。从而有：

UNION	DEE	DUM
DEE	DEE	DEE
DUM	DEE	DUM

INTERSECT	DEE	DUM
DEE	DEE	DUM
DUM	DUM	DUM

MINUS	DEE	DUM
DEE	DUM	DEE
DUM	DUM	DUM

对于差，第一个操作对象在左边，第二个在顶部（当然，对其他的操作符操作对象是可交换的）。请注意这些表分别与经过OR、AND和AND NOT作用的表有些相像；当然，相似并不是巧合的。

再考虑选择和投影，有：

- 如果选择条件为真，则对DEE的任何选择都会产生DEE；否则产生DUM。
- 对DUM的任何选择都产生DUM 。
- 如果关系为空，则在零个属性上的投影产生 DUM，否则产生DEE。特别地，对DEE或DUM的投影（0个属性上）返回它的输入值。

对于扩展（extend）和汇总（summarize），有：

- 对DEE和DUM进行扩展，增加一个新的属性，就产生一个度为1的关系，基数与输入的相同。
- 对DEE和DUM进行汇总，就产生一个度为1的关系，基数与输入的相同。

注意：我们没有考虑DIVIDEBY、SEMIJOIN和SEMIMINUS，因为它们不是基本的操作符。TCLOSE与此无关（因为它只作用于二元关系上）。由于众所周知的原因，这里也不考虑GROUP和UNGROUP。

6.11 没有！

6.12 可以合理地定义单个关系R上的并、交、连接和积的结果就是R本身。至于零的情况，

设*RT*是某种关系类型。于是：

- 零个类型为*RT*的关系的并产生类型为*RT*的空关系。注意必须指定*RT*为并的操作对象。
- 零个类型为*RT*的关系的交产生类型为*RT*的全称（universal）关系，即结果关系的主体包含了所有符合表头的元组。注意术语全称关系经常用作不同的意思！——参看[12.19]。
- 零个关系的积和连接都是TABLE_DEE。

6.13
```
J
```

6.14
```
J WHERE CITY = 'London'
```

6.15
```
( SPJ WHERE J# = J# ( 'J1' ) ) { S# }
```

6.16
```
SPJ WHERE QTY ≥ QTY ( 300 ) AND QTY ≤ QTY ( 750 )
```

6.17
```
P { COLOR, CITY }
```

6.18
```
( S JOIN P JOIN J ) { S#, P#, J# }
```

6.19
```
( ( ( S RENAME CITY AS SCITY ) TIMES
    ( P RENAME CITY AS PCITY ) TIMES
    ( J RENAME CITY AS JCITY ) )
WHERE SCITY ≠ PCITY
OR    PCITY ≠ JCITY
OR    JCITY ≠ SCITY ) { S#, P#, J# }
```

6.20
```
( ( ( S RENAME CITY AS SCITY ) TIMES
    ( P RENAME CITY AS PCITY ) TIMES
    ( J RENAME CITY AS JCITY ) )
WHERE SCITY ≠ PCITY
AND   PCITY ≠ JCITY
AND   JCITY ≠ SCITY ) { S#, P#, J# }
```

6.21
```
P SEMIJOIN ( SPJ SEMIJOIN ( S WHERE CITY = 'London' ) )
```

6.22 使用分步的方法：

```
WITH ( S WHERE CITY = 'London' ) AS T1,
     ( J WHERE CITY = 'London' ) AS T2,
     ( SPJ JOIN T1 ) AS T3,
     T3 { P#, J# } AS T4,
     ( T4 JOIN T2 ) AS T5 :
     T5 { P# }
```

下面是不使用WITH的查询：

```
( ( SPJ JOIN ( S WHERE CITY = 'London' ) ) { P#, J# }
             JOIN ( J WHERE CITY = 'London' ) ) { P# }
```

下面的练习，有些使用WITH，有些不使用。

6.23
```
( ( S RENAME CITY AS SCITY ) JOIN SPJ JOIN
  ( J RENAME CITY AS JCITY ) ) { SCITY, JCITY }
```

6.24
```
( J JOIN SPJ JOIN S ) { P# }
```

6.25
```
( ( ( J RENAME CITY AS JCITY ) JOIN SPJ JOIN
    ( S RENAME CITY AS SCITY ) )
                      WHERE JCITY ≠ SCITY ) { J# }
```

6.26
```
WITH ( SPJ { S#, P# } RENAME S# AS XS#, P# AS XP# ) AS T1,
     ( SPJ { S#, P# } RENAME S# AS YS#, P# AS YP# ) AS T2,
     ( T1 TIMES T2 ) AS T3,
     ( T3 WHERE XS# = YS# AND XP# < YP# ) AS T4 :
     T4 { XP#, YP# }
```

6.27
```
( SUMMARIZE SPJ { S#, J# }
```

```
PER RELATION { TUPLE { S# S# ( 'S1' ) } }
ADD COUNT AS N ) { N }
```

在PER子句中的表达式是一个选择子调用。

6.28
```
( SUMMARIZE SPJ { S#, P#, QTY }
  PER RELATION { TUPLE { S# S# ( 'S1' ), P# P# ( 'P1' ) } }
  ADD SUM ( QTY ) AS Q ) { Q }
```

6.29
```
SUMMARIZE SPJ PER SPJ { P#, J# } ADD SUM ( QTY ) AS Q
```

6.30
```
WITH ( SUMMARIZE SPJ PER SPJ { P#, J# }
                     ADD AVG ( QTY ) AS Q ) AS T1,
     ( T1 WHERE Q > QTY ( 350 ) ) AS T2 :
     T2 { P# }
```

6.31
```
( J JOIN ( SPJ WHERE S# = S# ( 'S1' ) ) ) { JNAME }
```

6.32
```
( P JOIN ( SPJ WHERE S# = S# ( 'S1' ) ) ) { COLOR }
```

6.33
```
( SPJ JOIN ( J WHERE CITY = 'London' ) ) { P# }
```

6.34
```
( SPJ JOIN ( SPJ WHERE S# = S# ( 'S1' ) ) { P# } ) { J# }
```

6.35
```
( ( ( SPJ JOIN
    ( P WHERE COLOR = COLOR ( 'Red' ) ) { P# } ) { S# }
                     JOIN SPJ ) { P# } JOIN SPJ ) { S# }
```

6.36
```
WITH ( S { S#, STATUS } RENAME S# AS XS#,
                               STATUS AS XSTATUS ) AS T1,
     ( S { S#, STATUS } RENAME S# AS YS#,
                               STATUS AS YSTATUS ) AS T2,
     ( T1 TIMES T2 ) AS T3,
     ( T3 WHERE XS# = S# ( 'S1' ) AND
                 XSTATUS > YSTATUS ) AS T4 :
     T4 { YS# }
```

6.37
```
( ( EXTEND J ADD MIN ( J, CITY ) AS FIRST )
                                  WHERE CITY = FIRST ) { J# }
```

如果关系变量J为空，该查询会有何种结果？

6.38
```
WITH ( SPJ RENAME J# AS ZJ# ) AS T1,
     ( T1 WHERE ZJ# = J# AND P# = P# ( 'P1' ) ) AS T2,
     ( SPJ WHERE P# = P# ( 'P1' ) ) AS T3,
     ( EXTEND T3 ADD AVG ( T2, QTY ) AS QX ) AS T4,
     T4 { J#, QX } AS T5,
     ( SPJ WHERE J# = J# ( 'J1' ) ) AS T6,
     ( EXTEND T6 ADD MAX ( T6, QTY ) AS QY ) AS T7,
     ( T5 TIMES T7 { QY } ) AS T8,
     ( T8 WHERE QX > QY ) AS T9 :
     T9 { J# }
```

6.39
```
WITH ( SPJ WHERE P# = P# ( 'P1' ) ) AS T1,
     T1 { S#, J#, QTY } AS T2,
     ( T2 RENAME J# AS XJ#, QTY AS XQ ) AS T3,
     ( SUMMARIZE T1 PER SPJ { J# }
                    ADD AVG ( QTY ) AS Q ) AS T4,
     ( T3 TIMES T4 ) AS T5,
     ( T5 WHERE XJ# = J# AND XQ > Q ) AS T6 :
     T6 { S# }
```

6.40
```
WITH ( S WHERE CITY = 'London' ) { S# } AS T1,
     ( P WHERE COLOR = COLOR ( 'Red' ) ) AS T2,
     ( T1 JOIN SPJ JOIN T2 ) AS T3 :
     J { J# } MINUS T3 { J# }
```

6.41
```
J { J# } MINUS ( SPJ WHERE S# ≠ S# ( 'S1' ) ) { J# }
```

6.42
```
WITH ( ( SPJ RENAME P# AS X ) WHERE X = P# ) { J# } AS T1,
     ( J WHERE CITY = 'London' ) { J# } AS T2,
     ( P WHERE T1 ≥ T2 ) AS T3 :
     T3 { P# }
```

6.43
```
S { S#, P# } DIVIDEBY J { J# } PER SPJ { S#, P#, J# }
```

6.44
```
( J WHERE
    ( ( SPJ RENAME J# AS Y ) WHERE Y = J# ) { P# } ≥
    ( SPJ WHERE S# = S# ( 'S1' ) ) { P# } ) { J# }
```

6.45
```
S { CITY } UNION P { CITY } UNION J { CITY }
```

6.46
```
( SPJ JOIN ( S WHERE CITY = 'London' ) ) { P# }
  UNION
( SPJ JOIN ( J WHERE CITY = 'London' ) ) { P# }
```

6.47
```
( S TIMES P ) { S#, P# } MINUS SP { S#, P# }
```

6.48 下面列出两种方法。第一种只使用了6.2~6.3节中的操作符：

```
WITH ( SP RENAME S# AS SA ) { SA, P# } AS T1,
     /* T1 {SA,P#} : SA supplies part P# */

     ( SP RENAME S# AS SB ) { SB, P# } AS T2,
     /* T2 {SB,P#} : SB supplies part P# */

     T1 { SA } AS T3,
     /* T3 {SA} : SA supplies some part */

     T2 { SB } AS T4,
     /* T4 {SB} : SB supplies some part */

     ( T1 TIMES T4 ) AS T5,
     /* T5 {SA,SB,P#} : SA supplies some part and
                        SB supplies part P# */

     ( T2 TIMES T3 ) AS T6,
     /* T6 {SA,SB,P#} : SB supplies some part and
                        SA supplies part P# */

     ( T1 JOIN T2 ) AS T7,
     /* T7 {SA,SB,P#} : SA and SB both supply part P# */

     ( T3 TIMES T4 ) AS T8,
     /* T8 {SA,SB} : SA supplies some part and
                     SB supplies some part */

     SP { P# } AS T9,
     /* T9 {P#} : part P# is supplied by some supplier */

     ( T8 TIMES T9 ) AS T10,
     /* T10 {SA,SB,P#} :
        SA supplies some part,
        SB supplies some part, and
        part P# is supplied by some supplier */

     ( T10 MINUS T7 ) AS T11,
     /* T11 {SA,SB,P#} : part P# is supplied,
                         but not by both SA and SB */

     ( T6 INTERSECT T11 ) AS T12,
     /* T12 {SA,SB,P#} : part P# is supplied by SA
                         but not by SB */

     ( T5 INTERSECT T11 ) AS T13,
     /* T13 {SA,SB,P#} : part P# is supplied by SB
                         but not by SA */

     T12 { SA, SB } AS T14,
     /* T14 {SA,SB} :
        SA supplies some part not supplied by SB */

     T13 { SA, SB } AS T15,
     /* T15 {SA,SB} :
        SB supplies some part not supplied by SA */

     ( T14 UNION T15 ) AS T16,
     /* T16 {SA,SB} : some part is supplied by SA or SB
                      but not both */
```

```
    T7 { SA, SB } AS T17,
    /* T17 {SA,SB} :
       some part is supplied by both SA and SB */

    ( T17 MINUS T16 ) AS T18,
    /* T18 {SA,SB} :
       some part is supplied by both SA and SB,
       and no part supplied by SA is not supplied by SB,
       and no part supplied by SB is not supplied by SA
       -- so SA and SB each supply exactly the same parts */

    ( T18 WHERE SA < SB ) AS T19 :
    /* tidy-up step */

T19
```

第二种方法（简单得多）利用了6.9节介绍的关系比较。

```
WITH ( S RENAME S# AS SA ) { SA } AS RA ,
     ( S RENAME S# AS SB ) { SB } AS RB :
     ( RA TIMES RB )
          WHERE ( SP WHERE S# = SA ) { P# } =
                ( SP WHERE S# = SB ) { P# }
          AND   SA < SB
```

6.49 `SPJ GROUP ( J#, QTY ) AS JQ`

6.50 **设SPQ代表练习6.49的结果。于是：**

```
SPQ UNGROUP JQ
```