第9章 视 图

9.1 引言

正如在第3章中所述的，视图只是一个命名了的关系代数表达式（或关系代数的等价形式）。比如：

```
VAR GOOD_SUPPLIER VIEW
   ( S WHERE STATUS > 15 ) { S#, STATUS, CITY } ;
```

当这条语句被执行时，此关系代数表达式（即这个视图定义表达式）没有被计算，只是让系统把它记住——准确地说是在系统目录中把它以 GOOD_SUPPLIER这个名字保存起来。对于用户来说，数据库中好像确实有一个叫作 GOOD_SUPPLIER的关系变量，其元组如图 9-1 中没有阴影的部分所示（假设的样本数据）。换句话说，GOOD_SUPPLIER指向一个导出的（虚的）关系变量，其值是视图定义表达式计算后所得到的结果。

GOOD_SUPPLIER

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

图9-1 基本关系变量S上的视图GOOD_SUPPLIER (如阴影部分所示)

在第3章中我们这样解释视图，如 GOOD_SUPPLIER，它是一个用来看数据的窗口：任何对所映射的数据的更新，通过这个窗口都可以实时和自动地看见（只要是在视图可见的范围内）；相应地，任何对视图的更新将自动和实时地在所映射的数据上进行更新，这样，通过个窗口也是可见的。

现在，依据环境不同，用户可能有或没有意识到 GOOD_SUPPLIER是一个视图；也有的用户可能意识到这一点，明白后面有一个真正的关系变量 S，还有一些用户以为 GOOD_SUPPLIER是一个“真正的”关系变量。无论用户处于哪一种情况，都没什么不同。重要的是，用户能像操纵一个真正的关系变量一样来操纵 GOOD_SUPPLIER。例如，下面有一个对GOOD_SUPPLIER 的查询：

```
GOOD_SUPPLIER WHERE CITY ≠ 'London'
```

设数据如图9-1所示，则结果为：

S#	STATUS	CITY
S3	30	Paris
S5	30	Athens

此查询的结果就像是常规的关系变量上的一个常规查询。并且，如在第 3章所说的，系统对这种查询的处理是将它转化为在所映射的关系变量（或基本关系变量）上的等价查询。实现的方法是将查询中出现的视图名字替换为定义视图的表达式。在这个例子中，替代过程为：

```
( ( S WHERE STATUS > 15 ) { S#, STATUS, CITY } )
                                WHERE CITY ≠ 'London'
```

它的等价形式为

```
( S WHERE STATUS > 15 AND CITY ≠ 'London' )
                                  { S#, STATUS, CITY }
```

查询产生前面所示的结果。

顺便提一下，替代过程只是更精确地描述出这个工作（由于关系封闭的特性）——将视图名字用视图定义表达式代替。关系封闭性是指，无论何时，若关系变量 *R*出现在一个表达式中，则一个关系表达式可以替代它出现（只要它与 *R*的类型相当）。换句话说，视图能够正常工作是因为在关系代数中关系是封闭的——这也就是为什么封闭性有如此重要地位的原因。

更新操作也以同样的方式工作。例如以下操作：

```
UPDATE GOOD_SUPPLIER WHERE CITY = 'Paris'
       STATUS := STATUS + 10 ;
```

被转化为：

```
UPDATE S WHERE STATUS > 15 AND CITY = 'Paris'
       STATUS := STATUS + 10 ;
```

对INSERT和DELETE进行相似的操作。

1. 进一步举例

这一小节将举出更多的例子，以便于后文参考。

1)
```
VAR REDPART VIEW
    ( ( P WHERE COLOR = COLOR ( 'Red' ) ) { ALL BUT COLOR } )
                                          RENAME WEIGHT AS WT ;
```

视图REDPART包含属性P#，PNAME，WT，CITY并且只包含红色零件的元组。

2)
```
VAR PQ VIEW
    SUMMARIZE SP PER P { P# } ADD SUM ( QTY ) AS TOTQTY ;
```

与GOOD_SUPPLIER和REDPART不同，视图PQ不只是某个简单的子集——一个限制和/或投影——而是那个选定关系变量的压缩或统计摘要。

3)
```
VAR CITY_PAIR VIEW
    ( ( S RENAME CITY AS SCITY ) JOIN SP JOIN
      ( P RENAME CITY AS PCITY ) ) { SCITY, PCITY } ;
```

不严格地说，当且仅当位于 *x*市的供应商提供了存放在 *y*市的零件时，城市对 (*x*,*y*) 在视图CITY_PAIR中出现。比如，供应商S1提供零件P1，S1位于London，零件也存放在London，则城市对（London ,London）出现在视图中。

4)
```
VAR HEAVY_REDPART VIEW
    REDPART WHERE WT > WEIGHT ( 12.0 ) ;
```

此视图经由另一个视图定义。

2. 定义和删除视图

下面是定义视图的语法：

```
VAR <relvar name> VIEW <relational expression>
                  <candidate key definition list> ;
```

*<candidate key definition list>*允许为空，因为系统可以推断出视图的候选码 [10.6]。在GOOD_SUPPLIER的例子中，系统应该意识到唯一的候选码是 {S#}，是从选定的基本关系变量 S中继承来的。

我们说——用第2章中的ANSI/SPARC术语——视图定义综合了外部模式的功能和外模式 /

概念模式映射的功能，因为它既指出了外部实体的内容，也说明了外部实体到概念模式（就是对应的基本关系变量）映射的方式。注意：一些视图的定义没有指明外模式 /概念模式映射，而是外模式/外模式映射。前面的HEAVY_REDPART就是这样的一个例子。

删除一个视图的语法是：

```
DROP VAR <relvar name> ;
```

<*relvar*>是指一个视图。在第5章中曾提到过，如果存在任一视图定义指向一个基本关系变量，则删除此基本关系变量的请求将会失败。与此相似，如果存在另一个视图定义指向某个视图，则删除此视图的请求也将会失败。替换的方法是，可以考虑扩展视图定义语句，使之包括“ RESTRICT ”或者“ CASCADE ”。RESTRICT（缺省值）的意思是当某个视图被其它视图所引用时，对此视图的删除请求将会失败；CASCADE的意思是删除请求会级联地删除引用此视图的视图。注意： SQL不支持这样的选项，但可以在删除语句而不是视图定义语句中对这一选项进行设置。这里没有缺省值，所需选项必须显式给出（参见 9.6节）。

9.2 视图的用途

使用视图有很多原因，下面给出其中的一些理由：

- 视图对于隐藏的数据自动给与安全保障。

  “ 隐藏的数据 ”是指在某个视图中不可见的数据（如，在视图 GOOD_ SUPPLIER中的供应商名称）。显然，这些数据对于特定的视图来说，在存取中是安全的，因此强制用户通过视图来对数据库进行存取，相当于建立了一个简单而有效的安全机制，对于这一用途，第16章进行了更详细的讨论。

- 视图提供了一个快捷方式或者是“ 宏 ”的功能。

  考虑查询“ 存放在伦敦的供应商所提供的零件的城市 ”。用前面的“ 进一步举例 ”小节中的视图CITY_PAIR，下述形式就可以实现：

```
( CITY_PAIR WHERE SCITY = 'London' ) { PCITY }
```

相反，没有视图，这个查询的实现就要复杂得多：

```
( ( ( S RENAME CITY AS SCITY )
      JOIN SP
      JOIN ( P RENAME CITY AS PCITY ) )
              WHERE SCITY = 'London' ) { PCITY }
```

用户可以直接使用第二种格式 —— 在安全性约束上是相同的 —— 但第一种显然要简单一些（第一种只是第二种的快捷方式；在执行时，系统的视图处理机制将有效地把第一种格式转化为第二种格式）。

它与编程语言中的“ 宏 ”很相似。原则上，编程语言的用户可以在原代码中直接写出宏的扩展后的形式 —— 但当用到宏，让宏处理系统来进行这一转换时，这一过程要简便得多（也可能是出于便于理解的原因）。相似的道理也可用于视图。这样，视图在数据库中的作用与宏在编程语言中的作用是相似的，宏的优势显然也就是视图的优势。尤其注意对于视图操作没有运行时的系统开销，只有视图处理时的一点系统开销（与宏的扩展相似）。

- 视图使相同的数据在同一时间被不同的用户以不同的方式查看。

  视图可以方便地让用户仅注意到自己关心的数据而忽略其它的。当有很多要求不同的用户时，这一点就显得很重要了，视图使这些用户同时与同一个数据库交互。

- 视图能提供逻辑上的数据独立性。

这是视图的最重要的用途之一，详见下一小节。

1. 逻辑上的数据独立性

这里提醒读者，逻辑上的数据独立性可以定义为：当用户或用户程序对数据库逻辑结构（逻辑结构是指概念上的结构——参见第2章）进行改变时的抗扰性。即通过视图，可以取得关系系统中逻辑上的数据独立性。这种逻辑上的数据独立性有两个方面：可成长性(growth)和可重构性(restructuring)。可成长性这个概念很重要，但与视图关系不大，介绍它的原因只是为了完整。

- 可成长性

随着数据库不断纳入新的数据而增长时，对于此数据库的定义也相应地增长了。对于“成长”有两种可能的情况：

1) 已有的基本关系变量为增加新的属性而扩张，这是相应于对对象的某一种现存的类型增加一种新的信息而言——如，对于供应商基本关系变量增加折扣这一属性。

2) 引入一个新的基本关系变量，这是相应于增加对象的一种新的类型对象的——例如，对于供应商-零件数据库增加工程信息。

这两种情况对于已存在的用户或用户程序都没有影响，至少理论上是这样（针对于SQL，请参考第7章7.7节中的例1）。

- 可重构性

有时，需要对数据库进行重新构造，这样，虽然全部信息的内容没有改变，但信息的逻辑位置可能已经改变——也就是说，对于基本关系变量的属性的分配以同样方式发生了改变。这里，我们只考虑一个简单的例子。假设出于某种原因，希望用下面的两个基本关系变量替代基本关系变量S：

```
VAR SNC BASE RELATION { S# S#, SNAME NAME, CITY CHAR }
    PRIMARY KEY { S# } ;

VAR ST BASE RELATION { S# S#, STATUS INTEGER }
    PRIMARY KEY { S# } ;
```

重要的是，旧关系变量S是新关系变量SNC和ST的连接（SNC和ST是S的投影）。因此以这个连接创建一个视图S：

```
VAR S VIEW
    SNC JOIN ST ;
```

任何以前对于基本关系变量S的应用程序和交互操作现在都成为针对视图S的。这样——假设系统能够正确处理视图中的数据操作——相对于数据库的重构，对用户和用户程序是没有影响的[⊖]。

另外，对于将原来的供应商关系变量S用它的两个投影SNC和ST代替，并不是随随便便就可以这样做的，尤其是当发觉发货关系变量SP也要进行一些处理时——因为它有一个参照原关系变量S的外码。见本章末的练习9.13。

回到本章讨论的主线：当然，从SNC-ST这个例子并不能得出结论，对于所有的重构行为，都能同样地得到逻辑上的数据独立性。关键是是否存在一个从重构后的数据库

⊖ 这只是理论！不幸的是，现在的SQL产品（SQL标准）不能正确支持视图中的大部分数据操作，因此也无法实现所期望的独立性，如前面的例子所示的。更明确地说，这些数据库产品能够正确地为视图进行查询数据的操作，但没有任何产品——在作者所知的范围内——能在视图的更新上做到百分之百正确地支持。因此，一些产品在数据的查询操作上能完全实现逻辑上的数据独立性，但目前还没有产品能在更新操作上达到同样的程度。

版本到原来的数据库版本的明确的映射关系（也就是说，重构是否可逆），或者说，数据库的这两个版本是否是信息等价的（information-equivalent）。如果不是，就无法得到逻辑上的数据独立性。

2. 两个重要的准则

通过前面的讨论，我们得出另一个要点：视图被用于两个非常不同的目的：

- 定义视图*V*的用户显然知道相应的视图定义表达式 *X*；只要表达式*X*可用的地方，视图 *V* 都可用。但这种用法（如前文所示）只是快捷方式。
- 当用户只被告知视图 *V*存在并可用时，他可能并不知道视图定义表达式 *X*；对于这样的用户，视图*V*在外观上和行为上都应像是基本关系变量。

继续前面的讨论，对于哪个关系变量是基本的而哪个关系变量是导出的这样一个问题具有很大的随意性，现在讨论这个问题。还以前面“重构”小节中的关系变量 S、SNC和ST为例。很显然，可以 (a)定义S为基本关系变量，而 SNC和ST为该基本关系变量投影得到的视图；(b)也可以定义 SNC和ST为基本关系变量，而 S为这两个基本关系变量连接操作得到的视图 ㊀。在基本关系变量和导出的关系变量中不能有随意的或不必要的区别。这一事实被称为“互换性准则”（The Principle of Interchangeability）（对于基本关系变量和导出的关系变量）。尤其注意这一准则暗示视图一定是可更新的 —— 数据库的可更新性不能取决于“哪些关系变量是基本的，哪些是导出的”这样一个可随意的选择上。详见 9.4节。

暂时，我们将基本关系变量的集合称为真实数据库（real database）。一个典型的用户并非只和真实数据库交互，而是与可表达数据库(expressible database)交互，可表达数据库是基本关系变量和视图的混和体。现在假设可表达数据库中的任何一个关系变量都不能由其余的关系变量派生出来（因为这样的关系变量可以被删除而没有信息丢失）㊁，这样，从用户的角度看，它们都是基本关系变量。当然，它们之间都是相互独立的（用第3章中的术语来说就是“自治的”）。对数据库来说也是同样的 —— 可以随意指定哪个数据库是真实的，只要所有的选择是信息等价的。这一事实被称为“数据库相对性准则”(The Principle of Database Relativity)。

9.3 视图检索

本章已经简单解释了在一个视图上的检索操作如何被翻译成在视图对应的基本关系变量上的等价操作。现在，把这个解释作得更型式化。

首先，每一个给定的关系表达式可以被看作是一个以关系为变量的函数：对表达式中的各关系变量赋值（表示调用此函数时的参数）后，表达式产生另一个关系。现在设 *D*是一个数据库（目前把它看作是基本关系变量的集合），*V*是定义在*D*上的视图，也就是说，视图的定义表达式是定义在*D*上的一个函数：

```
V = X ( D )
```

设*R*是一个在*V*上的检索操作；*R*也是一个以关系为变量的方程，检索的结果是：

```
R ( V ) = R ( X ( D ) )
```

这样，检索的结果被定义为等于将 *X*应用于*D*上的结果 —— 也就是说，物化一个关系的拷贝，这个关系就是视图的当前值 —— 然后将*R*应用到这个物化（materializing）后的拷贝上。

㊀ 参见第11章，11.2节“无损分解”。

㊁ 这里忽略用户定义的视图，这样的视图只是快捷形式。

使用替代过程显然更有效率，正如9.1节中所讲的（现在可以看出该过程等价于构成一个函数*C*，而*C*是函数*X*和*R*的组合*R*（*X*），然后将*C*施加于*D*上）。用物化而不是用替代来定义视图检索的语义很方便，至少在理论上；换句话说，如果替代产生的结果能够保证与使用物化时产生的结果相同（当然，确实如此），那么替代就是有效的。

经过上述讨论，应该已经对这些内容有了基本了解，了解它们是出于下列原因：

- 首先，它为后面讨论与此相似的（更繁琐的）更新操作打下基础。
- 其次，它阐明了物化是很好的又合法的视图实现方法（虽然效率不是很高）——至少是对于检索操作。但物化不适用于更新操作，因为对视图更新实际上要精确地转化为对视图所对应的基本关系变量的更新，而不是某些数据的临时物化的拷贝。参见9.4节。
- 再次，虽然替代过程很直观而且在理论上百分之百有效，但不幸的是，在一些SQL产品中它实际上无法工作！——也就是说，在一些SQL产品中，对视图的检索会莫名其妙地失败。对于SQL/92以前的SQL标准也在实际中不起作用。失败的原因是这些产品，以及SQL标准的早期版本，不能完全支持关系封闭属性。参见练习9.14。

9.4 视图更新

视图的更新问题可以这样表达：对于给定视图上的一个更新操作，要在对应的基本关系变量上进行怎样的操作，才能实现对于视图的更新？更准确地说，设*D*是数据库，*V*是*D*上的视图，也就是说，此视图是定义在*D*上的函数：

```
V = X ( D )
```

（如9.3节所示）。设*U*是一个在*V*上的更新操作；*U*可以被看作是一个能改变参数的操作，于是

```
U ( V ) = U ( X ( D ) )
```

在视图上进行更新这一问题就变成找到一个在*D*上的更新操作*U'*：

```
U ( X ( D ) ) = X ( U' ( D ) )
```

因为*D*是唯一真正存在的东西（视图是虚拟的），因此更新无法被直接施加到视图上。

在进一步讨论之前，有必要强调一下，视图更新一直是近些年来研究的重点，也有很多人提出了多种不同的解决方案（本书作者也提出了其中之一）；参见[9.7]，[9.10~9.13]，[9.15]，特别是Codd针对RM/V2的提议书[5.1]。本章介绍在[9.9]中提到的一种相对较新的方法，它不如上述提议书那么典型，但它的一些思想可以和该提议中某些最好的方面相媲美。它的优点之一是与以前的方法相比，可以支持更多种类的可更新视图：实际上，它把所有视图都看作是潜在可更新的，而不考虑完整性约束。

1. 修订后的黄金法则

回忆前一章学过的黄金法则：

> 如果一个更新操作将使一个关系变量处于违反自身某个谓词的状态，这样的更新是被禁止的。

当刚引入这条法则时，我们强调它适用于所有的关系变量，基本的或导出的。也就是说，导出的关系变量也有谓词——事实上，由于互换定律的原因，它必须有——为了正确实施视图更新，系统也要知道这些谓词是什么。那么视图的谓词是什么样的呢？显然，我们要的是谓词推理规则（predicate inference rule）的集合，这样，如果知道对于某个关系操作的输入

的谓词，我们就能从此操作中推断出输出的谓词。有了这样的一些规则，就能通过视图直接或间接参照的那个基本关系变量的谓词来推断出视图的谓词（当然，这些基本关系变量的谓词是已知的：它们是在此关系变量上定义的约束——即候选码约束——的逻辑与）。

找到这些规则是很容易的——它们紧跟在关系操作符定义的后面。比如，*A*和*B*是同类型的关系变量，它们各自的谓词是 *PA*和*PB*，如果视图*C*被定义为*A*交*B*，则视图*C*的谓词显然就是(*PA*) AND (*PB*)；也就是说，如果一个元组要在 *C*中出现，当且仅当它对于 *PA*和*PB*都为真。下面会继续讨论其它的关系操作。

注意：导出的关系变量会自动从被导出的关系变量中继承某些约束。但也可以让导出的关系变量还受附加约束的制约。因此，能够显式地描述约束是必要的（如视图的候选码就是一个例子），TutorialD支持这一可能性。为了简单起见，下文忽略这一可能性。

2. 关于视图更新机制

对于视图更新，系统操作有很多要遵守的重要准则（黄金法则是其中最重要的一个）。涉及到的准则如下：

1) 视图可更新性是一个语义学的问题，而与语法无关——也就是说，视图能否更新不能依赖于视图定义的语法形式。比如，下面两个定义在语义上是相同的：

```
VAR V VIEW
    S WHERE STATUS > 25 OR CITY  = 'Paris' ;

VAR V VIEW
  ( S WHERE STATUS > 25 ) UNION ( S WHERE CITY = 'Paris' ) ;
```

显然，这两个视图应该都可更新，或者都不可更新（实际上，它们是可更新的）。相反，SQL标准和当前的大部分 SQL产品采用这样的特殊处理办法：即第一个视图是可更新的，但第二个是不可更新的（参见 9.6节）。

2) 下一个问题是，对于视图就是基本关系变量这一特殊情况，视图更新机制必须要正确处理——因为如果视图*V*定义为*B* UNION *B*（或者*B* INTERSECT *B*，或者*B* WHERE *true*，或其它结果等于*B*的表达式），那么*B*和*V*在语义上是无法区分的。因此，适用于*V*=*B* UNION *B*这一视图上的更新规则，必须要与在 *B*上直接进行更新时产生的结果相同。换句话说，这一小节的题目虽然是“视图更新”，但不如“关系变量更新”更合适；我们一直是在描述一种对于所有关系变量都正常工作的理论，而不仅仅是视图。

3) 更新法则必须要在用到它的地方保持对称。比如，对于视图 *V*=*A* INTERSECT *B*的DELETE法则不应该产生这样的结果：在 *A*中这个元组被删除了，而在 *B*中却没有。虽然这种单方面的删除也会产生元组在视图上被删除的效果。相反，元组应该在 *A*和*B*上都被删除（换言之，不能产生歧义——对于视图更新应该有一个明确的工作方式，在任何情况下它都能正常工作。尤其是，当两个视图分别被定义为 *A* INTERSECT *B*和*B* INTERSECT *A*时，它们之间应没有任何逻辑差别）。

4) 更新规则要考虑所有生效的触发过程，包括像级联删除这样的参照动作。

5) 出于简便，不妨将UPDATE看作是DELETE-INSERT操作序列的简写形式。我们将对它进行这样的处理。当下列条件满足时，这种简写方式是可以接受的：

- 在更新过程中不进行关系变量谓词的检查；也就是说，UPDATE的扩展形式是DELETE-INSERT-检查，而不是DELETE-检查-INSERT-检查。当然，原因是DELETE会暂时违反关系变量谓词，但整个 UPDATE不会违反；比如，设关系变量 *R*含有10个

元组，如果*R*的关系变量谓词要求*R*至少包含10个元组，考虑在*R*上更新元组*t*会产生什么样的结果。

- 触发过程也同样不能在更新过程中被实施（实际上它们在结束时才实施，在关系变量谓词检查之前）。
- 这一简写形式在投影视图中要有一些细小的调整。

6) 所有视图上的更新操作必须是通过在相对应的关系变量上实施同类的更新操作来实现。也就是说，插入映射为插入，删除映射为删除（由于前面的观点，这里可以忽略更新操作）。相反，有一些视图——如UNION视图——插入映射成删除。还有就是在一个基本关系变量上的插入有时也会映射成删除！得到这一结论是因为基本关系变量*B*在语义上等于UNION视图*V*=*B* UNION *B*。在其它类型的视图上（限制、投影、相交，等等）也会产生相似的问题。在一个基本关系变量上的插入可能实际上是一个删除，这一思想显然是荒谬的：因为我们的出发点是INSERT映射为INSERT，DELETE映射为DELETE。

7) 一般来说，当更新规则被应用到视图*V*时，它将指定在*V*所映射的基本关系变量上的操作。即使当这些基本关系变量就是视图自身时，这些规则也要能正确工作。也就是说，这些规则要能递归实施。当然，如果对于被映射的基本关系变量的更新请求失败，则原更新也失败；因此，视图上的更新成功与否和在基本关系变量上的更新一样。

8) 这些规则不能断定数据库是设计良好的（即完全规范化的，参见第11章和第12章）。但如果数据库不是规范设计的，它们有时会产生奇异的结果——在支持规范设计上，这可被看作是一个额外的论据。下一小节中会给出这种“奇异结果”的例子。

9) 不应该有什么很显然的原因使得允许在视图上的某些更新操作，而另一些却不行（如DELETE可以而INSERT不行）。

10) 在一定的范围内，INSERT和DELETE应该是互反的操作。

回忆一下另一个重要的观点。在第5章中曾说过，关系操作——尤其是关系更新——都是集合级的；只包含一个元组的集合是一个特例。此外，多元组更新有时也是必要的（也就是说，某些更新不能用一系列的单元组更新来完成）。并且，这一观点对基本关系变量和视图都是对的。出于简化的目的，把大部分更新规则以单元组操作的形式呈现，但我们要始终意识到单元组操作是简化了的形式，有时甚至是过分简化后的形式。

下面逐个考虑关系代数中的操作符，从并、交、差开始。注意：对这三种操作，假设是在分别处理定义表达式为*A* UNION *B*或*A* INTERSECT *B*或*A* MINUS *B*的视图，其中*A*和*B*是关系表达式（也就是说它们未必是基本关系变量）。*A*和*B*所指代的关系必须是同一关系类型。相应的关系变量谓词分别是*PA*、*PB*。

3. 并

下面是*A* UNION *B*的插入规则：

- 插入：新的元组必须满足*PA*或*PB*，或同时满足*PA*和*PB*。若它仅满足*PA*，则它被插入到*A*中；注意这种插入可能会引起也被插入*B*中去的副作用[⊖]。若它满足*PB*，则被插入到

⊖ 在将要讨论的规则和例子中有几个是用来说明产生副作用的可能性。众所周知，副作用是令人讨厌的：但重要的是，如果*A*和*B*恰巧表示同一个关系变量上有重叠的子集，这在UNION、INTERSECTION和DIFFERENCE视图上经常发生，则副作用是不可避免的。此外，有时副作用是人们所期望的，而并非令人讨厌。

*B*中，除非由于上一种情况的副作用已经被插入 *B*中。

注意：这一规则的程序处理方式（插入 *A*，然后插入 *B*）应该被理解为是因为教学的目的而被简化；这并不暗示 DBMS必须是按这一执行顺序来完成 INSERT操作。实际上，对称法则——前一小节的第三点法则——也暗示了这一层意思，因为 *A*和*B*都没有相对于另一方的优先权。在下述的规则中也是一样的。

解释：新插入的元组至少要满足 *PA*或*PB*中的一个，否则它就不会被包含在 *A* UNION *B*中——也就是，它不满足*A* UNION *B*上的关系变量谓词*PA*或*PB*（再次假设新元组并没有在*A*或*B*中出现过，否则就是企图插入一个已有的元组。有时我们的假设并不足够严格）。假设新元组被插入到它在逻辑上属于的*A*或*B*（或两者都是）。

例子：设视图UV定义如下：

```
VAR UV VIEW
  ( S WHERE STATUS > 25 ) UNION ( S WHERE CITY = 'Paris' ) ;
```

图9-2显示了该视图的一个可能的值，对应于通常的样本数据值。

UV

S#	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris
S5	Adams	30	Athens

图9-2 视图UV（示例取值）

- 设要插入的元组是（ S6,Smith,50,Rome）[⊖]。这一元组满足谓词公式 S WHERE STATUS > 25，而不满足另一个谓词公式 S WHERE CITY = ‘ Paris ’。因此它会被插入到公式S WHERE STATUS > 25中。这是由于有关插入的限制（参见后文）。结果是它被插入到供应商基本关系变量中，这样，此元组会像期望的那样出现在视图中。
- 设要插入的元组是（ S7,Jones,50,Paris）。这一元组满足谓词公式 S WHERE STATUS > 25，也满足另一个谓词公式 S WHERE CITY = ‘ Paris ’。逻辑上它应该被同时插入到两个视图中。但是，插入到其中任何一个都会引起同时被插入到另一个中的副作用，这时，无需再显式地执行第二个 INSERT。

现在，设*SA*和*SB*是两个不同的基本关系变量。 *SA*表示status > 25的供应商，*SB*表示在Paris的供应商（见图9-3）；设视图UV定义为*SA* UNION *SB*，考虑前面讨论过的两个插入操作。插入元组（S6,Smith,50,Rome）到UV中会使它被插入到*SA*中，这正是所要的。但是，插入元组（S7,Jones,50,Paris）到UV中会使它被同时插入到这两个关系变量中！这一结果在逻辑上是正确的，虽然它违反直觉（这就是我们在前一节中所说的“ 奇异结果 ”的一个例子）。如果数据库设计得不好，就会产生这种奇怪的结果。尤其是，让一个相同的元组出现——也就是满足谓词——在两个不同的关系变量中，是一个不好的设计。在第 12章12.6节中将详细讨论这一有争议的情况。

下面讨论*A* UNION *B*的DELETE规则：

- DELETE：如果要删除的元组在*A*中出现，则从*A*中删除（注意这一删除可能会引起同时在*B*中删除的副作用）。如果它（还）在*B*中出现，则在*B*中删除。

[⊖] 为增强可读性，在这部分采用一种简化的元组表示方法。

SA

S#	SNAME	STATUS	CITY
S3	Blake	30	Paris
S5	Adams	30	Athens

SB

S#	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris

图9-3 基本关系变量SA和SB（样值）

这一规则的例子留作练习。注意删除 *A*或*B*中的一个元组可能会引起级联删除或其它触发过程。

最后，是UPDATE 规则。

- UPDATE：更新后的元组必须能满足 *PA*或*PB*或同时满足两者。如果要被更新的元组在 *A*中出现，它会被从 *A*中删除而不引发触发过程（如级联删除，等等），同样也不引起对 *A*的关系变量谓词的检查。注意这一删除会有将这一元组同时从 *B*中删除的副作用。如果元组是（还）在B中出现，它将被从*B*中删除（同样不引发触发过程和谓词检查）。然后，如果更新后的元组满足 *PA*，则被插入到*A*中（可能会有同时插入 *B*中的副作用）。最后，如果更新后的元组满足*PB*，则被插入到*B*中，除非它在插入到*A*时已被插入 *B*中。

这一更新规则实质上是由 DELETE规则和INSERT规则合并而成，除了在 DELETE后不引发触发过程和谓词检查（任何与 UPDATE有关的触发过程将在所有的删除和插入后执行，在谓词检查之前）。

值得指出，这种处理 UPDATE的方式可能会产生这样一种后果：一个元组从一个关系变量迁移到另一个中。以图 9-3中的数据库为例，将元组（ S5,Adams,30,Athens）更新为(S5,Adams,15,Paris)，会使旧元组在*SA*中被删除，新元组被插入到 SB中。

4. 交

现在讨论*A* INTERSECT *B*这一视图的更新规则。暂时只是简单叙述这些规则而不进行深入讨论（它们遵循与 UNION视图大体一样的更新模式），但是要注意，*A* INTERSECT *B*的谓词是（*PA*）AND（*PB*）。它的各种情况的例子将被留作练习。

- INSERT：新的元组必须同时满足 *PA*和*PB*。如果它原先没有在*A*中出现，则它被插入到*A*中（注意，这一操作的副作用可能是它同时被插入到 *B*中）；如果它原先没有（还没有）在*B*中出现，则它就被插入到*B*中。
- DELETE：要删除的元组被从*A*中删除（注意，这一操作的副作用可能是将它同时在 *B*中删除）。如果它还在*B*中，再从*B*中删除它。
- UPDATE：被更新后的元组必须同时满足 *PA*和*PB*。当元组被从*A*中删除时，它不引发任何触发过程和谓词检查（注意，这一操作的副作用可能是将它同时在 *B*中删除）；如果它还在*B*中，则再从*B*中删除也不引发任何触发过程和谓词检查。接下来，更新后的新元组如果没有在*A*中出现，则将之插入到*A*中；如果还没有在*B*中出现，则将之插入到*B*中。

5. 差

下面是*A* MINUS *B*的更新规则（关系变量谓词是（*PA*）AND NOT（*PB*））：

- INSERT：新元组满足*PA*而不满足*PB*。它被插入到*A*中。
- DELETE：元组被从*A*中删除。
- UPDATE：更新后的元组满足 *PA*而不满足*PB*。元组被从*A*中删除而不引发触发过程和谓词检查；更新后的新元组被插入到 *A*中。

6. 选择

设视图*V*的定义表达式是*A* WHERE *p*，*A*的谓词是*PA*，则*V*的谓词是（*PA*）AND（*p*）。比如，S WHERE CITY=' London' 的谓词是（PS）AND（CITY=' London '），PS是供应商的谓词。下面是*A* WHERE *p*的更新规则：

- INSERT：新元组要满足*PA*和*p*，它被插入到*A*中。
- DELETE：元组被从*A*中删除。
- UPDATE：更新后的元组必须同时满足*PA*和*p*。旧元组被从*A*中删除而不引发触发过程和谓词检查；新元组被插入到*A*中。

举例：设视图LS定义为：

```
VAR LS VIEW
    S WHERE CITY = 'London' ;
```

图9-4给出了此 视图的样值。

LS

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

图9-4 视图LS（样值）

- 在LS中插入元组（S6,Green,20,London）的请求会成功。新元组会被插入到 S中，也就相当于被插入到LS中了。
- 在LS中插入元组(S1,Green,20,London)的请求会失败，因为它违反了 S的谓词（因此也违反了LS） —— 尤其是，它违反了候选码 {S#}的唯一性约束。
- 在LS中插入元组（S6,Green,20,Athens）的请求将失败，因为它违反了约束CITY='London'。
- 在LS中删除元组（S1,Smith,20,London）的请求会成功。元组被从 S中删除，因此也就是在LS中被删除了。
- 在LS中更新元组（S1,Smith,20,London）为（S6,Green,20,London）的请求会成功。而更新元组（S1,Smith,20,London）为（S2,Smith,20,London）或（S1,Smith,20,Athens）的请求会失败（各自的原因是什么？）。

7. 投影

这里先进行相关谓词的讨论。设关系变量*A*（在其上有谓词*PA*）的属性可分为两个不相交的组*X*和*Y*。将*X*和*Y*分别看成是一个复合属性，考查 *A*在*X*上的投影*A*{*X*}。设{*X*：*x*}是这一投影的一个元组，很显然，这一投影的谓词应该是“ 在 Y的域上存在一个值*y*，使元组{*X*:*x*,*Y*;*y*}满足*PA*”。例如，关系变量*S*是在*S#*、SNAME和CITY上的投影，则在此投影中出现的每一个元组（*s*,*n*,*c*）都应存在一个值*t*使元组(*s*,*n*,*t*,*c*)满足S上的谓词。

下面是在*A*{*X*}上的更新规则：

- INSERT：设要插入的元组是(*x*)，另设*Y*的缺省值是*y*（如果不存在这样的缺省值就会出现错误。也就是说如果*Y*不允许缺省）[⊖]，元组(*x*,*y*)（它必须满足*PA*）被插入到*A*。

候选码属性通常（并非一成不变）没有缺省值（参见 18章）。因此，如果有一个投影不包

⊖ 这一句暗示，有一种可用的方法来指定基本关系变量中某个属性的缺省值。在 Tutorial D中讲到了定义缺省值的语法DEFAULT（<*default spec commanlist*>），每一处<*default spec*>应该是<*attribute name*><*default*>的形式。比如，可以指定供应商关系变量 S的缺省值DEFAULT(STATUS 0,CITY")。

含所有的候选码，它就会不允许被插入。

• DELETE：当从$A\{X\}$中删除一个元组时，所有在A中的有这个X值的元组都被删除。

注意：在实际中，X至少应包含一个A上的候选码，这样在$A\{X\}$上删除一个元组时，在A上也只有一个相应的元组被删除。但是，并没有什么逻辑上的原因使得必须这样做。相似的道理也适用于UPDATE —— 参见下文。

• UPDATE：设要更新的元组是（x），更新后的元组是（x'）。设a是A中在X上有值x的一个元组，a中在Y上的值是y。所有这样的元组都被删除而不引发触发过程和谓词检查。然后，对每一个值y都有一个元组（x',y）（必满足PA）被插入到A中。

注意：在“关于视图更新机制”小节的第 5条原则中曾说过，在投影问题的更新上要有一点调整。应该注意到，在更新规则中的插入步骤里，插入的元组是被重新写入原来的 y值 —— 而不是被写入可用的缺省值，但是单独的插入会是这样。

示例：设视图SC定义为：

```
SC { S#, CITY }
```

图9-5显示了视图的样值。

SC

S#	CITY
S1	London
S2	Paris
S3	Paris
S4	London
S5	Athens

图9-5 视图SC（样值）

• 在SC中插入元组(S6,Athens)的请求会成功，其结果是在S中插入(S6,*n*,*t*,London)，*n*和*t*分别是在SNAME和STATUS上的缺省值。

• 在SC上插入元组（S1,Athens）的请求会失败，因为它违反了S上的谓词（因此也违反了SC上的谓词） —— 尤其是，它违反了候选码{S#}的唯一性约束。

• 在SC上删除元组（S1，London）的请求会成功，此元组被从S中删除。

• 在SC中将元组(S2,London)更新为（S1，Athens）的请求会成功；结果是将S中的元组(S1,Smith,20,London)更新为(S1,Smith,20,Athens) —— 而不是(S1,*n*,*t*,Athens)，其中*n*和*t*为使用的缺省值。

• 将SC中的元组(S1,London)更新为(S2,London)的请求会失败（为什么？）。

对于投影中不包含它所映射的关系变量上的候选码的情况 —— 比如，S在STATUS和CITYY中的投影 —— 留做练习。

8. 扩展

设视图V的定义表达式为：

```
EXTEND A ADD exp AS X
```

（和往常一样，PA是A的谓词）。V的谓词PE为：

```
PA ( a ) AND e.X = exp ( a )
```

其中，e是V中的一个元组，a是当e的X扩展部分被移去后剩下的元组（不太严格地，可以说 a是e在属性集A上的投影）。用自然语言说就是：

每一个扩展后的元组 *e*是：(1)通过投影从*e*中去掉*X*部分而得到的元组 *a*满足*PA*；(2) 在*X*部分上的值等于在*a*上执行*exp*表达式后的结果。

下面是更新规则：

- INSERT：设被插入的元组是 *e*；*e*必须满足*PE*。通过投影去掉 *X*部分的元组*a*被插入到*A*中。
- DELETE：设要删除的元组是*e*；通过投影去掉X部分的元组*a*被从*A*中删除。
- UPDATE：设更新前的元组是*e*，更新后的元组是*e'*；*e'*必须满足*PE*。*e*通过投影去掉*X*部分的元组*a*被从*A*中删除而不引发触发过程和谓词检查， *e'*通过从投影去掉 X部分的元组*a'*被插入到*A*中。

示例：设视图VPX定义为：

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

图9-6显示了这一视图的样值。

VPX

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9-6 视图VPX（样值）

- 插入元组(P7,Cog,Red,12,Paris,5448)的请求会成功，结果是在关系变量 P中插入元组(P7,Cog,Red,12,Paris)。
- 插入元组（P7,Cog,Red,12,Paris,5449）的请求会失败（为什么？）。
- 插入元组（P1,Cog,Red,12,Paris,5448）的请求会失败（为什么？）。
- 删除元组P1的请求会成功，其结果是从关系变量 P中删除元组P1。
- 将P1元组更新为（P1,Nut,Red,10,Paris,4540）的请求会成功，其结果是更新 P中的元组（P1,Nut,Red,12,London）为(P1,Nut,Red,10,Paris)。
- 将P1元组更新为P2（其它值不变）或使GMWT值不是WEIGHT值的454倍的更新请求会失败（各自的原因是什么？）。

9. 连接

以前讲到的大部分更新处理 —— 包括本书的前5版和本书作者的其它著作 —— 认为给定连接的可更新性或不可更新性，依赖于（至少部分依赖于）这个连接的类型是一对一的，一对多的，还是多对多的。和以前的方式不同，现在认为连接总是可以更新的。而且，这三种连接的规则相同，并且都很直接。这一结果虽然初看上去有些令人吃惊，但这确实有道理，是由于通过黄金法则对这一问题进行考查而得出了这一结论。下面对它进行解释。

广义来说，提供视图支持的目的是为了使视图看起来尽可能像基本关系变量，这一目标是值得赞赏的。但是

- 通常（暗含着）假设“对关系变量中一个元组的更新与其它元组是无关的”是可能的。
- 但是，后来又发现对关系变量中一个元组的更新与其它元组无关并不总是可以做到的。

比如，Codd在[11.2]中说明了在某个特定的连接中仅删除一个元组是不可能的，因为这样

结果会使一个关系“不再是任意两个关系的连接”（意思是结果不可能再满足视图的谓词）。而且历史上视图更新的方法一直是简单地拒绝请求，因为它们不可能被完全看作是基本关系变量上的更新。

现在的方法很不相同。确切地说，我们发现即使在一个基本关系变量中，也不可能仅更新一个元组而与其它的元组无关。这样，就接受（那些在历史上）一直被拒绝的视图更新操作，把它们解释为用逻辑上显然正确的方式来对其所映射的关系变量上的更新；接受这些更新，而且承认这些在视图上的更新可能会产生副作用——但是，为了避免可能违反视图上的谓词，副作用可能是必然的。

现在开始讨论细节。下面，先定义几个术语，然后陈述连接视图的更新规则，最后考虑这些规则对三种不同的情况（一对一，一对多，多对多）分别意味着什么。

考查连接*J*=*A* JOIN *B*（第6章6.4节），其中，*A*、*B*和*J*分别包括属性{*X*，*Y*}，{*Y*，*Z*}和{*X*，*Y*，*Z*}。设*A*和*B*的谓词是*PA*和*PB*，则*J*的谓词*PJ*是

```
PA (a) AND PB (b)
```

对于此连接中指定的元组*j*，*a*是*J*中属于*A*的部分（即，通过投影除去*Z*部分后得到的元组），*b*是*J*中属于*B*的部分（即，通过投影除去*X*部分后得到的元组）。换句话说：

> 连接中的每一个元组，*A*部分满足*PA*，*B*部分满足*PB*。

比如，对于关系变量S和SP在S#上的连接有如下谓词：

> 对于连接中的每一个元组（*s*,*n*,*t*,*c*,*p*,*q*），有元组(*s*,*n*,*t*,*c*)满足S上的谓词，元组(*s*,*p*,*q*）满足SP上的谓词。

下面是具体的更新规则：

- INSERT：新元组*j*必须满足*PJ*。如果*j*中*A*部分没有在*A*中出现，则将之插入到*A*中[⊖]。如果*j*中*B*部分没有在*B*中出现，则将之插入到*B*中。
- DELETE：被删除元组的*A*部分从*A*中删除，*B*部分从*B*中删除。
- UPDATE：被更新后的元组必须满足*PJ*。原元组*A*部分被从*A*中删除，不引发触发过程和谓词检查；原元组*B*部分被从*B*中删除，也不引发触发过程和谓词检查。然后，如果更新后新元组的*A*部分没有在*A*中出现，则将之插入到*A*中，如果新元组的*B*部分没有在*B*中出现，则将之插入到*B*中。

下面考查这些规则对三种不同情况下的含意。

第一种情况（一对一）：首先，注意术语“一对一”，更准确地说应该是“（一或零）对（一或零）”。也就是说，有完整性约束可以保证对于*A*中的每一个元组在*B*中至多有一个元组与之匹配，反之亦然——这暗示着在连接上的属性集*Y*是*A*和*B*的超码（如果想回忆一下超码的概念，请参见第8章8.8节）。

示例：

- 第一个例子，可以考虑前述规则在这样一个连接上的结果：将供应商关系变量S（仅有）在供应商编号上与自身进行连接。

⊖ 注意，这一INSERT可能会有将*B*部分插入到*B*中的副作用，这和前面讨论的UNION、DIFFERENCE和INTERSECTION视图一样。相似的结论也适用于DELETE和UPDATE规则；为了简化，不必详细说明每一情况下的各种可能性。

• 第二个例子，设存在含有属性S#和REST的基本关系变量 SR，S#用来标识供应商，REST用来标识此供应商最喜欢的餐馆。假设在S中的供应商并不都在SR中出现。考虑连接更新规则在S JION SR上的结果。如果某些供应商在SR中出现而不在S中出现，则会有什么不同？

第二种情况（一对多）：术语“一对多”，准确地说应该是“（零或一）对（零或更多）”。也就是说，存在完整性约束，它保证对于 *B*中的每一个元组在 *A*中至多有一个元组与之匹配。它表示，连接上的属性集 Y上存在一个属性集 *K*，这个*K*是*A*的一个候选码，也是匹配到 *B*上的一个外码。注意：如果真是这种情况，就可以将“零或一”换为“正好一个”。

示例：设视图SSP被定义为：

```
S JOIN SP
```

（这显然是一个“外码-候选码匹配”连接）。图9-7给出了样值。

SSP

S#	SNAME	STATUS	CITY	P#	QTY
S1	Smith	20	London	P1	300
S1	Smith	20	London	P2	200
S1	Smith	20	London	P3	400
S1	Smith	20	London	P4	200
S1	Smith	20	London	P5	100
S1	Smith	20	London	P6	100
S2	Jones	10	Paris	P1	300
S2	Jones	10	Paris	P2	400
S3	Blake	30	Paris	P2	200
S4	Clark	20	London	P2	200
S4	Clark	20	London	P4	300
S4	Clark	20	London	P5	400

图9-7 视图SSP(样值)

• 在SSP中插入元组(S4,Clark,20,London,P6,100)的请求会成功，其结果是在 SP中插入元组(S4,P6,100)(也就相当于在视图中插入了元组)。

• 在SSP中插入元组(S5,Adams,30,Athens,P6,100)的请求会成功，其结果是在 SP中插入元组(S5,P6,100)(也就相当于在视图中插入了元组)。

• 在SSP中插入元组(S6,Green,20,London,P6,100)的请求会成功，其结果是在关系变量 S中插入元组(S6,Green,20,London)，在关系变量 SP中插入元组(S6,P6,100) (也就相当于在视图中插入了元组)。

注意：假设可能会出现这样一种情况，在SP中存在一个元组，而在S中没有相对应的元组。再进一步假设在SP中已经包含了几个供应商号为 S6的元组，但是它们的零件号都不是 P1。则刚讨论过的INSERT会在视图中插入一些额外的元组——也就是说，元组(S6,Green,20,London)和在SP中已有的供应商号为S6的元组的连接。

• 在SSP中插入元组(S4,Clark,20,Athens,P6,100)的请求会失败（为什么？）。

• 在SSP中插入元组(S1,Smith,20,London,P1,400) 的请求会失败（为什么？）。

• 在SSP中删除元组(S3,Blake,30,Paris,P2,200)的请求会成功，其结果是从 S中删除元组(S3,Blake,20,Paris)，从SP中删除元组(S3,P2,200)。

• 从SSP中删除元组(S1,Smith,20,London,P1,300)的请求会成功——见下面的注释——其结果是从S中删除元组(S1,Smith,20,London)，从SP中删除元组(S1,P1,300)。

注意：这一DELETE请求的结果依赖于从发货量到供应商的外码删除规则。如果这一规则指定为RESTRICT，则所有的操作失败。如果指定为 CASCADE，它就会产生对供应商S1删除所有其他的SP元组的副作用（因此删除了所有相关的 SSP元组）。

- 将SSP元组(S1,Smith,20,London,P1,300)更新为(S1,Smith,20,London,P1,400)的请求会成功，其结果是更新SP中元组（S1,P1,300）为(S1,P1,400)。
- 将SSP元组(S1,Smith,20,London,P1,300)更新为(S1,Smith,20,Athens,P1,400)的请求会成功，其结果是更新S中的元组(S1,Smith,20,London)为(S1,Smith,20,Athens)，更新SP中的元组（S1,P1,300）为(S1,P1,400)。
- 更新SSP中元组（S1,Smith,20,London,P1,300）为(S6,Smith,20,London,,P1,300)的请求会成功 —— 参见下面注释 —— 结果是更新S中元组（S1,Smith,20,London）为(S6,Smith,20,London)，更新SP中元组(S1,P1,300)为(S6,P1,300)。

  注意：这一UPDATE请求的结果依赖于从发货量到供应商的外码更新规则。细节留作练习。

第三种情况（多对多）：术语“多对多”，更准确地说应该是“（零或更多）”对“（零或更多）”。换句话说，不存在有效的完整性约束来保证处于与“第一种情况”和“第二种情况”相同的状态。

示例：设存在定义如下的视图：

```
S JOIN P
```

（S和P在CITY上连接 —— 多对多的连接）。在图9-8中给出样值。

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9-8 S和P在CITY上的连接

- 插入元组(S7,Bruce,15,Oslo,P8,Wheel,White,25)的请求会成功，其结果是在S中插入元组(S7,Bruce,15,Oslo)，在P中插入元组(P8,Wheel,White,25,Oslo)(相当于在视图中插入了指定的元组)。
- 插入元组（S1,Smith,20,London,P7,Washer,Red,5）的请求会成功，其结果是在P中插入元组(P7,Washer,Red,5,London) —— 这相当于在视图中插入了两个元组，指定的元组（S1,Smith,20,London,P7,Washer,Red,5）和另一个元组（S4,Clark,20, ,P7,Washer,Red,5)。

更多的例子被留作练习。

10. 其它操作符

最后，简要介绍一下关系代数中的其它操作符。我们关注一下连接：半连接、半差和除，它们不是原语操作，因此它们的规则可以由定义这些操作符的操作规则导出。其它的也一样：

- 重命名：不用详细讨论。
- 笛卡尔积：如在第6章6.4节提到的，笛卡尔积是自然连接（如果 *A*和*B*没有相同的属性，*A* JOIN *B*就退化为*A* TIMES *B*）的特例，结果，*A* TIMES *B*的规则就是*A* JOIN *B*(也是*A*

INTERSECT *B*)的一个特例。

- 合计：合计也不是原语——它是由扩展（extend）定义而来，因此它的更新规则也可以由扩展的更新规则导出来。注意：在实际中，大部分SUMMARIZE视图上的UPDATE操作都会失败。不过，这些失败不是因为这些视图因继承而来的不可更新性，而是因为这些更新与完整性约束的冲突。比如，设视图定义为：

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

对于删除供应商S1的元组的请求会成功；但是插入元组(S5,500)的请求会失败，因为它违反了约束“TOTQTY值必须等于单个QTY值的和”；插入元组(S5,0)的请求会失败，但原因与上一个不同（为什么？）。

- 分组与分组还原：与“合计”相似。
- Tclose：也和上面的有些相似。

9.5 快照

这一节将讨论一下快照[9.2]。快照与视图有点相似 ⊖，但又有所不同。与视图一样，快照是导出的关系变量；但与视图不同的是，它们是真正的关系变量，而不是虚拟的——也就是说，它们不是通过在其它关系变量上的定义来表示自身，而是（至少在理论上是）通过它们各自的物化的数据备份。比如：

```
VAR P2SC SNAPSHOT
    ( ( S JOIN SP ) WHERE P# = P# ( 'P2' ) ) { S#, CITY }
      REFRESH EVERY DAY ;
```

定义一个快照就像是执行一个查询，但不同的是：

a) 查询的结果是以一个特定的名字保存在数据库中（本例中是 P2SC），它是只读的关系变量（只读，但是会被定期刷新，见b）。

b) 快照被定期刷新（比如每天）——也就是，它当前的值被丢掉，重新执行查询，新的结果成为快照的新值。

这样，快照就会和24小时前一样表示相关的数据（它的谓词是什么？）。

快照的意义在于，很多应用——甚至可能绝大部分——可能容忍，或是需要与某个确切时间相近的数据就可以了。报表和会计就是这一类的应用；这类应用的典型要求是数据被冻结在某一适当的时刻（比如是进行会计统计的一段时期），快照可以使这样的数据冻结而又不影响其它事务在这些数据上的更新操作（在真实数据上）。相似地，它可以为一个查询的大量数据或一个只读的应用服务，而不封锁数据库，这是非常有用的。注意：在分布式数据库或是决策支持环境中——分别参见20章和21章——这一想法非常有吸引力。可以说快照是“受控冗余”(controlled redundancy）的一种特殊情况（见第1章），“快照刷新”就是相应的更新过程（也见第1章）。

一般来说，快照定义语法如下：

```
VAR <relvar name> SNAPSHOT <relational expression>
    <candidate key definition list>
    REFRESH EVERY <now and then> ;
```

*<now and then>*应该是“月，周，天，小时，或是 *n*分钟，还可以是周一，周末，等等”

⊖ 实际上，它们有时被称为物化的视图（参见[9.1],[9.3],[9.6],[9.14]和[9.16]），但很不幸，这一术语是受指责的，因为无论是物化的视图还是非物化的视图都是一个执行问题，而不是模型问题；对于模型来说，视图是非物化的，“物化的视图”这本身就是一个矛盾的说法。

(特别地，用REFRESH [ON] EVERY UPDATE可以保持快照始终与它所导出的关系变量同步)。下面是删除快照的语法：

```
DROP VAR <relvar name> ;
```

显然，<*relvar name*>是用来指明快照。注意：假设当一个快照定义被其它关系变量定义所引用时，对这个快照的删除请求会失败。相应地，可以扩展快照定义使之包括“RESTRICT”或“CASCADE”选项。这里不再深入讨论后一个可能性了。

9.6 SQL对视图的支持

这一小节，总结一下SQL对视图的支持（在写作本书时，SQL还不支持快照)。首先，创建视图的语法是：

```
CREATE VIEW <view name> AS <table expression>
     [ WITH [ <qualifier> ] CHECK OPTION ] ;
```

<*qualifier*>可以是CASCADED或LOCAL，并且CASCADED是缺省值（也是唯一合理的选项，这一点在参考书[4.19]中进行了详细论述；这里也不再深入讨论LOCAL)。解释：

1) <*table expression*>是视图定义表达式。关于SQL表表达式的详细解释请见附录A。

2) 如果指明了WITH CHECK OPTION，则它表示，在此视图上的INSERT或DELETE若违反了任何视图定义表达式所蕴含的完整性约束，就会被拒绝。注意，只有当 WITH CHECK OPTION被指明时，这样的操作才会失败——也就是说，在缺省情况下，它们是不会失败的。在9.4节中，我们把这一行为看作是逻辑上不正确的；因此强烈建议在实际中WITH CHECK OPTION总是被指明 ⊖ (参见[9.8])。

示例：

1)
```
CREATE VIEW GOOD_SUPPLIER
     AS SELECT S.S#, S.STATUS, S.CITY
        FROM   S
        WHERE  S.STATUS > 15
     WITH CHECK OPTION ;
```

2)
```
CREATE VIEW REDPART
     AS SELECT P.P#, P.PNAME, P.WEIGHT AS WT, P.CITY
        FROM   P
        WHERE  P.COLOR = 'Red'
     WITH CHECK OPTION ;
```

3)
```
CREATE VIEW PQ
     AS SELECT SP.P#, SUM ( SP.QTY ) AS TOTQTY
        FROM   SP
        GROUP  BY SP.P# ;
```

与9.1节中“进一步举例”小节不同，这个视图不包含不被任何供应商提供的零件。参见第7章7.7节中对例8的讨论。

4)
```
CREATE VIEW CITY_PAIR
     AS SELECT DISTINCT S.CITY AS SCITY, P.CITY AS PCITY
        FROM   S, SP, P
        WHERE  S.S# = SP.S#
        AND    SP.P# = P.P# ;
```

5)
```
CREATE VIEW HEAVY_REDPART
     AS SELECT RP.P#, RP.PNAME, RP.WT, RP.CITY
        FROM   REDPART AS RP
        WHERE  RP.WT > 12.0
     WITH CHECK OPTION ;
```

⊖ 如果视图是可更新的，那么它就是可更新的。在下文中可以发现 SQL中的视图总是不能被更新，而且在SQL中如果视图不可更新，则WITH CHECK OPTION是不合法的。

存在的视图可以用DROP VIE语法来删除：

```
DROP VIEW <view name> <option> ;
```

<*option*>（和DROP TABLE、DROP DOMAIN相同）可以是RESTRICT或CASCADE。如果指明为RESTRICT，而且此视图在其它视图定义中被引用或在某个完整性约束中被引用，则DROP操作会失败；如果指明为CASCADE，则DROP操作成功，参照它的视图定义和完整性约束也会被删除。

1. 视图检索

正如9.3节所述，当前的SQL标准（SQL/92）保证所有的视图检索正确工作。不幸的是现在的数据库产品做不到这些，也不能达到SQL的早期版本的要求。参见本章末的练习9.14中a部分。

2. 视图更新

SQL/92标准对视图更新的支持有限。基本上，唯一被认为可更新的视图是从一个基本表中通过选择和投影操作而导出来的视图。而且，对这种简单的情况也会出错，这是由于SQL很少能理解关系变量谓词，尤其是对SQL表允许重复数据行这一情况。

下面是SQL/92中对于视图可更新性的更准确的描述（这一列表是从参考资料[4.19]中摘出，但在一定程度上作了简化）。在SQL中，只有当视图满足下面8个条件时才是可更新的：

1) 定义视图范围的表表达式是一个选择表达式；也就是说，它不直接包含以下关键词：JOIN、UNION、INTERSECT或EXCEPT。
2) 选择表达式的选择语句不直接包含关键词DISTINCT。
3) 选择语句（可能包括星号作为选项）中的每一个选择项包含一个合适的列名（可以伴着AS子句），表示所对应的表的一个列（参见下面第五段）。
4) 选择表达式的FROM子句只包含一个表的参照。
5) 这个表的参照用来标识一个基本表或是一个可更新的视图。注意：这个用表的参照来标识的表就是这个可更新视图所指向的表（参见上面的第三段）。
6) 这个选择表达式不包含这样一个WHERE子句：这个WHERE子句包含一个子查询，其中FROM所指向的表是与第四段中提到的FROM子句指向的表是同一个表。
7) 此选择表达式不包含GROUP BY子句。
8) 此选择表达式不包含HAVING子句。

几个要点：

1) 在SQL中的可更新性或者是完全可以，或者是完全不可以，这就意味着INSERT、UPDATE和DELETE或者都行，或者都不行 —— 不可能出现这种情况：DELETE适用但INSERT不适用（虽然有的商业化产品可以做到这一点）。
2) 在SQL中，对于一个给定的视图，UPDATE或者可以实施或者不可以 —— 不可能在一个视图中有的列可以更新而有的列不可以（虽然，有些商业化产品在这点上比标准做得更好）。

9.7 小结

视图实际上是一个命名了的关系表达式；它可以看作是一个导出的、虚拟的关系变量。在视图上的操作事实上是被一组替换过程完成的；也就是说，对这一视图名的引用是被定义

这个视图的表达式所替换了——并且由于封闭性，这一替换过程能够准确地工作。对于检索操作，这组替换过程百分之百有效（虽然实际产品中做不到，但理论上是这样的）。对于更新操作，也是百分之百有效[⊖]（也是在理论上，虽然目前的产品肯定做不到）；但对某些视图（如定义的合计视图），更新会由于违反完整性约束而失败。我们列出了更新模式所要遵循的原则的一个超集，也列出了更新模式在用 UNION、INTERSECT、JOIN、DIFFERENCE、RESTRICT和PROJECT操作符定义的视图上如何工作。对于它们中的每一个，也详细讨论了对它们所映射的关系变量的谓词引用规则。

本章考察了视图和逻辑上的数据独立性的问题。对于这种独立性有两方面问题：可成长性和可重构性。视图的其它方面还包括它能隐藏数据的能力，这样它就能够提供一种安全性手段；它有用作快捷方式的能力，这可使用户更轻松。接下来又解释了两个重要的准则，互变准则（它表示无论情况如何，视图总是可更新的）和是数据相对性准则。

本章还对快照进行了简单的讨论。最后，对 SQL在相关方面进行了概略的描述。

练习

9.1 对于9.1节“进一步举例”小节中的基于代数的视图定义，给出其对应的基于演算的定义。

9.2 定义一个视图，这一视图包括了“供应商和他所提供的零件不在同一地点”的供应商号和零件号

9.3 为在London的供应商定义一个视图。

9.4 从供应商-零件-工程数据库定义中的关系变量 SPJ，定义在供应商-零件数据库中的视图 SP。

9.5 在供应商-零件-工程数据库上定义视图，使它包含所有由 S1提供零件而且使用零件 P1的工程（只要求工程编号和城市属性）。

9.6 对于下面的视图定义

```
VAR HEAVYWEIGHT VIEW
  ( ( P RENAME WEIGHT AS WT, COLOR AS COL )
      WHERE WT > WEIGHT ( 14.0 ) ) { P#, WT, COL } ;
```

请给出对下列语句实施替换过程后的转化形式：

a.
```
RA := HEAVYWEIGHT WHERE COL = COLOR ( 'Green' ) ;
```

b.
```
RB := ( EXTEND HEAVYWEIGHT ADD WT + WEIGHT ( 5.3 ) AS WTP )
                                                { P#, WTP } ;
```

c.
```
UPDATE HEAVYWEIGHT WHERE WT = WEIGHT ( 18.0 ) COL := 'White' ;
```

d.
```
DELETE HEAVYWEIGHT WHERE WT < WEIGHT ( 10.0 ) ;
```

e.
```
INSERT INTO HEAVYWEIGHT
RELATION { TUPLE { P# P# ( 'P99' ),
                   WT WEIGHT ( 12.0 ),
                   COL COLOR ( 'Purple' ) } } ;
```

9.7 设练习9.6中的视图HEAVYWEIGHT的定义作如下修改：

```
VAR HEAVYWEIGHT VIEW
( ( ( EXTEND P ADD WEIGHT * 454 AS WT ) RENAME COLOR AS COL )
      WHERE WT > WEIGHT ( 14.0 ) ) { P#, WT, COL } ;
```

(也就是说，属性WT现在是以克为单位的重量，而不再是以磅为单位)。现在，重复练

⊖ 这就是为什么将视图看作是关系变量，或者说是变量；由定义，变量总是可更新的。

习9.6。

9.8 在第8章中说过，有时希望能够在视图上定义候选码，或者是主码。为什么需要这一手段呢？

9.9 在第3章和第5章中提到过，为了支持视图，系统目录需要扩展，系统需要哪些扩展呢？对于快照又如何？

9.10 设一个指定的关系变量*R*可以被两个限制*A*和*B*代替，*A*和*B*满足*A* UNION *B*等于*R*，并且*A* INTERSECT *B*等于空。这种情况下是否有逻辑上的数据独立性？

9.11

a) 如果存在*A* INTERSECT *B*等于*A* JOIN *B*（JOIN是一对一的，但并非严格要求，因为在*A*中存在的元组不一定在 *B*中有与之相对应的元组），那么在9.4节中给出的对于INTERSECTION视图和JOIN视图的可更新性规则是否还适用于这种等价的形式？

b) 如果*A* INTERSECT *B*还等于*A* MINUS（*A* MINUS *B*），也等于*B* MINUS（*B* MINUS *A*），那么9.4节中给出的对于INTERSECTION视图和DIFFERENCE视图的可更新性规则是否还适用于这种等价的形式？

9.12 在9.4 节中给出了一条准则：INSERT和DELETE是互反的操作。那么还是在那一节中讲到的对于UNION、INTERSECTION和DIFFERENCE视图的更新规则是否遵守这一准则呢？

9.13 在9.2节（逻辑上的数据独立性）中讲到了重构供应商-零件数据库的可能性，这是用关系变量S的两个投影SNC和ST代替基本关系变量来实现的。而且这一重构不是无关紧要的，这是什么意思呢？

9.14 对于你可利用的任一种SQL产品：

a) 能否找到一个使视图检索失败的例子？

b) 视图更新的规则是什么？（它们可能不如在9.6节中所讲到的那么严格。）

9.15 考查供应商-零件数据库，为了简化，忽略零件关系变量。下面是对于供应商和发货的两个可能的设计概要：

a)
```
S   { S#, SNAME, STATUS, CITY }
SP  { S#, P#, QTY }
```

b)
```
SSP { S#, SNAME, STATUS, CITY, P#, QTY }
XSS { S#, SNAME, STATUS, CITY }
```

a方案是常规设计方法。在方案b中，关系变量SSP包含了每一个发货元组，并给出了零件号、数量和供应商的全部细节；关系变量XSS包含了所有不提供任何零件的供应商的详细内容（注意：这两个方案是信息等价的，而且体现了互换性准则）。给出方案a和b的视图定义表达式，说出每一方案的数据库约束（关于数据库约束的内容请参见第8章）。是否每一方案都有相对于另一方案的优势？如果有的话，这一优势是什么？

9.16 给出练习9.2~9.5的解答

9.17 对第3章3.2节给出的关系模型的定义重新加以考虑，并对此有一个充分的理解。

参考文献和简介

9.1 Brad Adelberg, Hector Garcia-Molina, and Jennifer Widom: “ The STRIP Rule System for

Efficiently Maintaining Derived Data," Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz. (May 1997).

STRIP是Stanford Real-time Information Processor(斯坦福实时信息处理器)的首字母缩写。每当快照（这里叫导出数据）所映射的基本数据发生变动时，它就用规则 —— 实际上是触发过程 —— 来更新快照。这样一个系统的问题是，如果基本数据变动频繁，执行规则的计算就多了。这篇文章讲述了用于减少这种计算的 STRIP技术。

9.2 Michel Adiba: " Derived Relations: A Unified Mechanism for Views, Snapshots, and Distributed Data," Proc. 1981 Int. Conf. on Very Large Data Bases, Cannes, France (September 1981). See also the earlier version " Database Snapshots," by Michel E. Adiba and Bruce G. Lindsay, IBM Research Report RJ2772 (March 7th, 1980).

这篇论文首次提出快照的概念，并对其语义和实现进行了讨论。关于实现，着重论述了各种差别刷新（differential refresh）和增量维护（incremental maintenance）的可能 —— 在每一次刷新时，系统并不总是要重新执行全部的原始查询。

9.3 D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek: " Efficient View Maintenance at Data Warehouses," Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz. (May 1997).

在第1章中曾说数据仓库是包含决策支持数据的数据库 —— 用本章的术语来说，就是快照（该文题目中的" view "不是指视图而是快照）。正如参考资料[9.2]的注释中所说，快照可以用递增的方式进行维护，而且出于执行效率的原因，要求进行这种递增方式的维护。但是，如果这个快照是从几个不同的数据库导出来，而且这几个数据库同时被更新了，那么增量维护方式就会出错。这篇论文提出了一个解决方案。

9.4 H. W. Buff: " Why Codd's Rule No. 6 Must Be Reformulated," *ACM SIGMOD Record 17*, No.4 (December 1988).

1985年，Codd发表了一个十二条的准则集，用来检测一个声称是完全关系的数据库是不是真的如此。其中第六条准则要求，理论上可更新的视图实际上也可以被更新。Buff在一篇简短的附注中称一般的视图可更新性问题都是不可预测的 —— 也就是不存在一般的算法来检测任意一个视图的可更新性（Codd的说法）。但是，本章中所用的视图可更新性定义与 Codd所说的有点不同，因为它直接关注所用的关系变量谓词。

9.5 E. F. Codd: " Is Your DBMS Really Relational?" and " Does Your DBMS Run by the Rules?", *Computer world* (October 14th and 21 st, 1985).

9.6 Latha S. Colby *et al*.: " Supporting Multiple View Maintenance Policies," Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz. (May 1997).

此论文题目中的" view "不是指视图，而是快照。其中讲到了三个广义上的方法来维护快照：

1) 立即（immediate）：每一次快照所映射的关系变量的更新立即触发一个相应的快照上的更新。
2) 延迟(deferred)：快照只有被查询时才会刷新。
3) 定期(periodic)：快照定期刷新（比如每天）。

通常，快照的目的是提高查询的性能，但代价是降低了更新的性能，而这三种维护

策略体现了在这两者之间的权衡。这篇论文讨论了在同一时刻同一系统中，在不同的快照上使用不同的策略所引起的问题。

9.7 Donald D. Chamberlin, James N. Gray, and Irving L. Traiger: " Views, Authorization, and Locking in a Relational Data Base System," Proc. NCC *44*, Anaheim, Calif. Montvale, N.J.: AF1PS Press (May 1975).

包括System R原型中用于视图更新的方法的基本原理（因此，也在 SQL/DS、DB2和SQL标准中）。参见参考文献[9.15]，它讨论的是University Ingres原型。

9.8 Hugh Darwen: " Without Check Option," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

9.9 C. J. Date and David McGoveran: " Updating, Union, Intersection, and Difference Views" and "Updating Joins and Other Views," in C. J. Date, *Relational Database Writings 1991 -1994*. Reading, Mass.: Addison-Wesley (1995). 注意：在本书创作时，此论文的正式版本也在准备中。

9.10 Umeshwar Dayal and Philip A. Bernstein: " On the Correct Translation of Update Operations on Relational Views," *ACM TODS 7*, No. 3 (September 1982).

早期用来处理视图更新问题的方法（只用于 RESTRICTION、PROJECTION和JOIN视图）。但不考虑关系变量谓词。

9.11 Antonio L. Furtado and Marco A. Casanova: " Updating Relational Views," in reference [17.1]

在处理视图更新问题上有两个广义的方法。一个是试图提供与具体涉及的数据库无关的通用机制（本书只对这种机制进行了详细论述）；它纯粹由视图定义驱动。另一个方法不这么随意，它要求 DBA为每个视图指定哪些更新是允许的并，给出语义说明，根据所参照的基本关系变量来写出实施更新操作的执行代码。这篇论文考查了这两种方法的工作情况（1985年）。文中大量引用了早期的书籍。

9.12 Nathan Goodman: " View Update Is Practical, " *InfoDB 5*, No.2 (Summer 1990).

对于视图更新问题的非正式的讨论。下面是其导言中的一段稍加解释过的摘要："Dayal和Bernstein[9.10]已经证明，凡是我们感兴趣的视图都不能被更新； Buff[9.4]证明了不存在用来推断任意视图是否可更新的算法。好像没有希望了，但实际上，视图更新既是可能的也是可行的"。文中给出了大量视图更新的特殊技巧。

9.13 Arthur M. Keller: " Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins," Proc. 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Portland, Ore. (March 1985).

提出了视图更新算法应该遵守的五条原则 —— 没有副作用，一步仅做一个变动，没有不必要的变动，没有更简单的其它方法，不用 DELETE-INSERT对来替代UPDATE —— 并给出了满足这五条原则的算法。这一算法可以将一种操作转化为另一种操作；比如，在某个视图上的DELETE操作可以翻译成在视图所参照的基本关系变量上的 UPDATE操作（例如，要在 " 供应商 " 视图上删除一个元组，方法可以是将 London改为Paris）；再举一个例子，在视图*V*上的DELETE（*V*定义为*A* MINUS *B*）可以这样实现：在*B*上插入一个元组，而不是在*A*上删除元组。注意，由于第六条原则，我们显然不会使用这样的算法。

9.14 Dallan Quass and Jennifer Widom: " On-Line Warehouse View Maintenance," Proc. 1997 ACM

SIGMOD Int. Conf. on Management of Data, Tucson, Ariz. (May 1997).

此论文题目中的“ view ”不是指视图，而是快照。本文提出了一种快照的维护机制，这种机制可以使维护事务与对快照的查询在时间上同步。

9.15 M. R. Stonebraker: “ Implementation of Views and Integrity Constraints by Query Modification,” Proc. ACM SIGMOD Int. Conf. on Management of Data, San Jose, Calif. (May 1975).

参见[9.7]的注释。

9.16 Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom: “ View Maintenance in a Warehousing Environment,” Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data, San Jose, Calif. (May 1995).

此论文题目中的“ view ”不是指视图，而是快照。当要对被参照的数据进行更新操作时，数据仓库站点可能会在进行必要的快照维护之前对基本数据站点发出一个查询，在查询与原始的基本数据更新之间的时间滞后会引起异常。本文提出了处理这种异常的算法。

部分练习答案

9.1 **列出了下列解决方案，并分别命名为** 9.1.*n*，*n*表示9.1中各例子的编号。对于范围变量，采用常规的假设。

9.1.1
```
VAR REDPART VIEW
    ( PX.P#, PX.PNAME, PX.WEIGHT AS WT, PX.CITY )
      WHERE PX.COLOR = COLOR ( 'Red' ) ;
```

9.1.2
```
VAR PQ VIEW
    ( PX.P#,
      SUM ( SPX WHERE SPX.P# = PX.P#, QTY ) AS TOTQTY ) ;
```

9.1.3
```
VAR CITY_PAIR VIEW
    ( SX.CITY AS SCITY, PX.CITY AS PCITY )
      WHERE EXISTS SPX ( SPX.S# = SX.S# AND
                         SPX.P# = PX.P# ) ;
```

9.1.4
```
VAR HEAVY_REDPART VIEW
      RPX WHERE RPX.WT > WEIGHT ( 12.0 ) ;
```

RPX**是一个在**REDPART**上的范围变量。**

9.2
```
VAR NON_COLOCATED VIEW
  ( S TIMES P ) { S#, P# } MINUS
  ( S JOIN  P ) { S#, P# } ;
```

9.3
```
VAR LONDON_SUPPLIER VIEW
  ( S WHERE CITY = 'London' ) { ALL BUT CITY } ;
```

注意：这里忽略CITY**属性，因为对此视图中的每一个供应商，其值都是** London**。但要注意，这一省略使这一视图上的所有更新必然会失败（除非视图所映射的供应商关系变量在**CITY**属性上的缺省值恰巧是** London**）。也就是说，像这样的视图可能根本不支持更新操作（或许可以考虑，对在此视图中插入的元组，定义其 CITY属性的缺省值是** London**，这一特定视图的缺省值的方法还有待深入研究）。**

9.4 **这里的问题是：在**SP**视图中如何定义属性**QTY**？合理的回答可能是这样：对于每一个** S#-P#**对，它的值应该是所有**SPJ.QTY**值的和，而不管**J#**的值如何。**

```
VAR SP VIEW
    SUMMARIZE SPJ PER SPJ { S#, P# }
                  ADD SUM ( QTY ) AS QTY ;
```

9.5
```
VAR JC VIEW
  ( ( SPJ WHERE S# = S# ( 'S1' ) ) { J# } JOIN
    ( SPJ WHERE P# = P# ( 'P1' ) ) { J# } ) JOIN
                                          J { J#, CITY } ;
```

9.6 这里不给出转化后的形式。但是，我们说e会失败，因为插入的元组不满足视图谓词。

9.7 这一次e也失败了，但原因有点不同。首先，由于用户没有提供真正的WEIGHT值（也不能这样做），DBMS会提供一个缺省的WEIGHT值，假设为W。其次，用户提供的WT值根本不可能正巧是w＊454——即使（不是显示的INSERT的情况）这个WT值正巧大于14.0。这样，插入的元组还是不满足视图的谓词。

注意：WEIGHT值是否应该在被454除后写入WT中这个问题还有争议。这一可能性还要进一步研究。

9.8 下列原因是从参考文献[5.7]中摘录的：

- 如果用户要与视图交互而不是与基本关系变量交互，那么这些视图就要看起来尽可能像基本关系变量。理想的情况是，用户根本不知道它们用的是视图，它们用起来还是和基本关系变量一样，这要归功于数据库相对性准则。正如基本关系变量的用户要知道关系变量的候选码是什么，视图的用户也要知道视图的候选码是什么。显式地声明码值是提供这类信息的正常的方法。
- DBMS也许不能为自己推断出候选码（当前市场上的DBMS产品都是这样），显式声明也许是告诉DBMS——同时也是告诉用户——候选码存在的唯一可用的方法（对于DBA）。
- 即使DBMS能自己推断出候选码，显式声明也至少会使DBMS检查自己的推断是否与DBA的显示声明一致。
- DBA可能知道一些DBMS不了解的知识，因此可以据此改善DBMS的推断。参考文献[5.7]给出了这样的例子。

[11.3]中提出了另一个理由，它认为这一工具提供了一种简便的方式来定义一些重要的完整性约束，否则就要以罗嗦的方式来定义。

9.9 显然，不可能对这个问题作出一个明确的答复。我们给出下列观察结果：

- 每一个视图和快照都在系统目录RELVAR中占有一项，并且在RVKIND属性上的值是“View”和“Snapshot”。
- 每一个视图还也在新的系统目录VIEW中占有一项。该项应该包含相关的视图定义表达式。
- 相似地，每一个快照也在新的系统目录SNAPSHOT中占有一项。其中也包含有相关的定义表达式，还包含相关的快照刷新间隔时间。
- 还有一个系统目录用来记录每个视图快照分别是用哪些关系变量来定义的。注意这个关系变量的结构与PART_STRUCTURE关系变量的结构有些相似（见第4章的图4-6）；正如部分可包含其它部分，视图和快照也可通过其它视图或快照来定义。第7章中对练习7.7的回答在这里会有用。

9.10 是的！但是还有一些要注意的。假设用两个限制SA和SB来取代供应商关系变量S，其中SA是在London的供应商，SB是不在London的供应商。可以将SA UNION SB定义为视图S。如果我们请求（通过这个视图）更新一个在London的供应商城市为London之外的

其它城市，或者非London的供应商地址改为London，这样一个UPDATE的实施就要映射成其中一个限制上的DELETE和另一个上的INSERT。9.4节中给出的规则就能在这种情况下正常工作——实际上，我们有意将UPDATE定义为一个DELETE-INSERT对，但是出于效率上的考虑，有一个默认的假设就是更新由UPDATE来完成。这个例子说明了在什么时候不能直接将UPDATE简单映射为UPDATE；在实际中，检查它是否正常工作是一种优化机制。

9.11 a. 是。b. 是。

9.12 只要数据库是：(a)按照正交设计原则（The Principle of Orthogonal Design）（参见12章12.6节）进行设计；且(b)DBMS能正确支持关系变量谓词，则INSERT和DELETE就总是互反的操作。但是如果这些条件不能满足，它们就有可能不可逆。例如，如果 *A*和*B*是不同的基本关系变量，在 *V* = *A* INTERSECT *B*中插入一个元组*t*，就可能导致它只被插入到*A*中（因为它早就在*B*中出现）；随后从*V*中删除元组*t*，这时就会从*A*和*B*中都删除它（相反，如果如果删除*t*，然后再次插入*t*，就总会维持现状）。注意，这一非对称的操作产生的可能是：仅当*t*满足*A*中的谓词，但最初没有在*A*中出现。

9.13 给出下列注释。首先，这一替换过程包括几个步骤，它被小结如下（在后面，这一操作顺序会被调整一下）。

```
/* define the new base relvars */

VAR SNC BASE RELATION
      { S# S#, SNAME NAME, CITY CHAR }
        PRIMARY KEY { S# } ;

VAR ST BASE RELATION
     { S# S#, STATUS INTEGER }
       PRIMARY KEY { S# } ;

/* copy the data to the new base relvars */

INSERT INTO SNC S { S#, SNAME, CITY } ;

INSERT INTO ST  S { S#, STATUS } ;

/* drop the old relvar */

DROP VAR S ;
```

现在，可以创建想要的视图了：

```
VAR S VIEW
      SNC JOIN ST ;
```

我们发现SNC和ST中的两个S#属性相互之间构成了外码。实际上，SNC和ST之间有严格要求的一对一关系，我们在这种一对一情况下会碰到一些问题，这些问题在[13.7]中已经详细讨论过了。

注意：对于SP中参照原来的基本关系变量S的外码，我们还要进行一定的处理。如果能让这一外码现在参照视图S，那就太好了（现在的产品还做不到），最好是在数据库中再增加基本关系变量S的第三个投影，其定义如下：

```
VAR SS BASE RELATION
     { S# S# } PRIMARY KEY { S# } ;

INSERT INTO SS S { S# } ;
```

（在参考文献[8.10]中，出于其它的原因而推荐了这一设计方式）。现在我们把视图S

的定义改为：

```
VAR S VIEW S
      SS JOIN SNC JOIN ST ;
```

我们还是在关系变量 SNC和ST的定义中加入外码说明：

```
FOREIGN KEY { S# } REFERENCES SS
                  ON DELETE CASCADE
                  ON UPDATE CASCADE
```

最后，必须将关系变量SP中指向S的外码[S#]改为指向SS。

注意：对于允许参照视图的外码还有待进一步研究。

9.14 对于a中的问题，现在确实有一些产品会在视图检索上失败，下面就是这样一个视图的例子。考虑下面这样的一个视图定义（9.6节中的例3）：

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

考虑下面的查询请求：

```
SELECT AVG ( PQ.TOTQTY ) AS PT
FROM   PQ ;
```

如果采用本章所述的简单的替代过程（也就是将对视图的引用改写为视图的定义表达式），就会得到如下的结果：

```
SELECT AVG ( SUM ( SP.QTY ) ) AS PT
FROM   SP
GROUP  BY SP.P# ;
```

这不是一个有效的查询语句，因为（在第 7章7.7节例7中已讲过）SQL不允许操作符的嵌套。

下面是在同一视图PQ上的查询的例子，在有些产品中它也会由于相同的原因而失败：

```
SELECT PQ.P#
FROM   PQ
WHERE  PQ.TOTQTY > 500 ;
```

正是由于这些例子所示的原因，一些实际产品——如IBM的DB2——有时会将视图物化（而不是使用替代过程），然后在物化了的版本中执行查询。这一方法当然会一直正常工作，但在运行效率上会有一些损失。而且，还是以 DB2为例，仍然会在一些视图检索上出现问题；也就是说，就算是在替代过程不能正常工作时， DB2也不总是用物化的方法，再说这两种方法中哪一个更有效还难下定论。比如，上面两个例子中的第二个在DB2中还是不能正常运行。详细讨论请参见 [4.20]。

9.15 下面是通过a给出的b的定义：

```
VAR SSP VIEW
    S JOIN SP ;

VAR XSS VIEW
    S MINUS ( S JOIN SP ) { S#, SNAME, STATUS, CITY } ;
```

下面是通过b给出的a的定义：

```
VAR S VIEW
    XSS UNION SSP { S#, SNAME, STATUS, CITY } ;

VAR SP VIEW
    SSP { S#, P#, QTY } ;
```

这两个方案的数据库约束如下：

```
CONSTRAINT DESIGN_A
    IS_EMPTY ( SP { S# } MINUS S { S# } ) ;

CONSTRAINT DESIGN_B
    IS_EMPTY ( SSP { S# } INTERSECT XSS { S# } ) ;
```

（DESIGN_A给出了用另一种方法来定义参照约束的例子。）

a方案明显要好一些，原因已在第11章讨论过。

9.16 这里列出下列方案，9.16.*n*中的*n*表示原来练习的题号。

9.16.2
```
CREATE VIEW NON_COLOCATED
    AS SELECT S.S#, P.P#
       FROM   S, P
       WHERE  S.CITY <> P.CITY ;
```

9.16.3
```
CREATE VIEW LONDON_SUPPLIER
    AS SELECT S.S#, S.SNAME, S.STATUS
       FROM   S
       WHERE  S.CITY = 'London' ;
```

9.16.4
```
CREATE VIEW SP
    AS SELECT SPJ.S#, SPJ.P#, SUM ( SPJ.QTY ) AS QTY
       FROM   SPJ
       GROUP  BY SPJ.S#, SPJ.P# ;
```

9.16.5
```
CREATE VIEW JC
    AS SELECT J.J#, J.CITY
       FROM   J
       WHERE  J.J# IN ( SELECT SPJ.J#
                        FROM   SPJ
                        WHERE  SPJ.S# = 'S1' )
       AND    J.J# IN ( SELECT SPJ.J#
                        FROM   SPJ
                        WHERE  SPJ.P# = 'P1' ) ;
```