第23章 基于逻辑的数据库

23.1 引言

在20世纪80年代中期，在数据库研究领域出现了一个重要的研究方向，这就是基于逻辑的数据库系统。这时有关逻辑数据库、推理DBMS、专家DBMS、演绎DBMS、知识库、知识库管理系统（KBMS）、数据模型逻辑和递归查询处理等的论文相继发表。然而，很难把这些术语和思想同熟悉的数据库术语和概念联系起来；而且也很难从传统数据库的角度理解其潜在的研究动机。所以，从传统的数据库思想和原理去解释所有这些问题就变得迫切。这一章试图解决这一问题。

我们的目标是从传统数据库的角度去解释什么是基于逻辑的系统，而并不是就逻辑谈逻辑。所以，当我们介绍有关逻辑的新思想时，会用传统的数据库术语去解释它，这样是可能的，也是合适的（当然，本书中已经讨论了一些有关逻辑的概念，特别是在第7章介绍关系演算时。关系演算是直接基于逻辑的。但基于逻辑的系统中用到的逻辑概念要远不止这些）。

本章的内容如下：23.2节将简单地概述，并介绍一些历史；23.3节和23.4节分别简单地介绍命题演算和谓词演算；23.5节介绍所谓的数据库证明理论(proof-theoretic)；23.6节介绍演绎DBMS；23.7节介绍递归查询过程的一些方法；最后，23.8节对本章作了小结。

23.2 综述

数据库理论和逻辑之间的关系的研究至少追溯到20世纪70年代后期，这一时期的论文可参看[23.5]、[23.7]和[23.13]。然而，近来在这一有兴趣的领域的发展中，最基本的突破应该是1984年Reiter发表的一篇具有里程碑意义的论文（参看[23.15]）。在这篇论文里，Reiter认为传统的数据库是模型理论。不严格地讲，他认为：

a. 在任何时候，数据库都可看作一系列明确的关系，每一个关系都包括一系列明确的元组；
b. 执行一个查询就是对这些确定的元组和关系执行一些指定的公式（即真值表达式(truth-valued expression)）。

注意：我们将在23.5节更加精确地解释“模型理论”这一术语。

Reiter还认为，一个可替换的证明理论的观点在某些方面是可能的，并且是很好的。不严格地讲，它是指：

a. 在任何时候，数据库都可看作一系列公理（相应于基本关系中的域和元组及某些所谓的“演绎”公理来说，就是“基本”公理(ground axiom)）；
b. 执行一个查询就是证明一些确定的公式是这些公理的逻辑结果，或者说，证明它是一些定理。

注意：我们也将在23.5节更加精确地解释“证明理论”这一术语，我们会看到它和第1章1.3节（参看“数据和数据模型”这一小节）介绍的数据库的特征非常相似，是真命题的一个集合

(参看[1.2])。

下面是一个很适合的例子，它是基于供应商-零件数据库的关系演算查询：

```
SPX WHERE SPX.QTY > 250
```

(当然，这里的SPX是定义在供货表上的范围变量）用传统的即模型理论的解释，供货表中的元组一个接一个地依次执行公式“ QTY>250 ”；查询结果仅仅包括一些供货表的一些元组，而且对于每个元组，此公式取真值。相比之下，用证明理论解释，我们就把供货表的一些元组（还有其它的项）看作一定的“逻辑理论”的公理；在这一理论里，我们用定理证明技术去决定范围变量SPX取哪些可能的值时，其逻辑结果为公式“ SPX.QTY>250 ”。此查询结果即由SPX的这些特殊值构成。

当然，这个例子是非常简单的，以至于很难辨别这两种解释之间的差别。但是，证明（即证明理论的解释）的推理机制显然比这个简单的例子所表达的要复杂得多。我们将会看到，它能解决那些经典关系系统所不能解决的问题，而且这一证明理论还有另外的很吸引人的特征（参看[23.15]）。

- 描述统一性：用它定义的数据库语言中，基本关系中的元组和域值、“演绎公理”、查询和完整性约束基本上用统一的方法描述。
- 操作统一性：它为各种各样明显不同的问题的统一操作提供了一个基础，这包括查询优化（尤其是语义优化）、完整性约束的实施、数据库设计（依赖理论）、程序正确性证明和其它的问题。
- 语义建模：它为各种基本模型语义的扩充提供了坚实的基础。
- 扩充应用：最后，它为处理用那些经典的方法很难处理的问题提供了基础，例如，对于析取信息处理（如，供应商 S5或者供应了零件 P1或者供应了零件 P2，但不知道它供应了哪一个）。

演绎公理

下面将简单扼要地解释演绎公理这一概念（或者叫推理规则）。基本说来，一个演绎公理就是一个规则，它可从给定的事实推断别的事实。例如，给定事实“ Anne是Betty的母亲 ”和“ Betty是Celia的母亲 ”，这里存在一个明显的演绎公理，从而使我们推断 Anne是Celia的祖母。因此，用前面所讲的理论，就可以把这个演绎 DBMS中的两个给定的事实描述为关系中的元组，即是：

MOTHER_OF

MOTHER	DAUGHTER
Anne	Betty
Betty	Celia

这两个事实提供了此系统的基本公理。我们再假设系统中以如下的形式给出这一演绎公理：

```
IF   MOTHER_OF       ( x, y )
AND  MOTHER_OF       ( y, z )
THEN GRANDMOTHER_OF  ( x, z ) ;
```

(假设并简化的语法)。现在，用23.4节介绍的方法，系统把演绎公理所表达的规则应用于上述基本公理中的数据，从而推导出 GRANDMOTHER_OF(Anne,Celia)的结果。这样用户就可查询如下问题：谁是Celia的祖母？或谁是Anne的孙女？（或者，更精确地讲， Anne是谁的祖母？)。

现在把前面的思想与传统的数据库的概念联系起来。从传统的术语讲，演绎公理可以认为是一种视图定义，例如：

```
VAR GRANDMOTHER_OF VIEW
  ( MX.MOTHER AS GRANDMOTHER, MY.DAUGHTER AS GRANDDAUGHTER )
    WHERE MX.DAUGHTER = MY.MOTHER ;
```

（这里我们有意使用关系演算的形式； MX和MY分别是定义在MOTHER_OF上的范围变量）。上面所述查询现在可基于视图概念重写如下：

```
GX.GRANDMOTHER WHERE GX.GRANDDAUGHTER = NAME ( 'Celia' )

GX.GRANDDAUGHTER WHERE GX.GRANDMOTHER = NAME ( 'Anne' )
```

（GX是定义在GRANDMOTHER_OF上的范围变量）。

到目前为止，我们所讲的都只不过是对一些熟悉的例子给出不同语法和解释。但是，在下一节将看到，实际上在基于逻辑的系统和更多传统的 DBMS之间的一些重要的区别是不能用这些简单的例子解释清楚的。

23.3 命题演算

在这一节和下一节，我们对一些基本的逻辑思想作一些非常简单的介绍。本节讲命题演算，下一节讲谓词演算。这里应当指出，命题演算本身并不是最重要的；本节真正的目的仅仅是为下一节的理解铺平道路。这两节合起来是为本章后面几节提供基础。

这里，大家需要熟悉布尔代数的概念。为了引用的需要，下面列出了一些将用到的布尔代数的法则：

- 分配律：

```
f AND ( g OR  h )  ≡  ( f AND g ) OR  ( f AND h )
f OR  ( g AND h )  ≡  ( f OR  g ) AND ( f OR  h )
```

- 摩根律：

```
NOT ( f AND g )  ≡  NOT f OR  NOT g
NOT ( f OR  g )  ≡  NOT f AND NOT g
```

这里，*f*、*g*和*h*都是任意的布尔表达式（结果为真值）。

下面，谈谈逻辑本身。逻辑可被定义为规范的推理方法。因为它是规范的，所以，它能用作执行规范的任务。如，只要检查作为每一步结果的变元结构，就能测试这个变元的正确性（即，不必注意这些步骤本身）。特别因为它是规范化的，故它有按部就班的过程 —— 即，它可以设计成程序，由机器所应用。

一般地讲，命题演算和谓词演算是两个特殊的逻辑形式（实际上，前者是后者的子集）。相对于任何符号计算的系统而言，“ 演算 ”仅是一般的术语；目前情况下，所涉及的各种计算是一定公式或表达式的真值（真或假）计算。

1. 项

假设有一些对象的集合，称为常量。对这些常量，我们可作多种说明。在数据库用语里，常量是域中的值，语句是诸如“ 3>2 ”的项表达式。把项定义成包含常量的语句，并且：

a. 它不包含任何逻辑连接词（下面将看到）或者括号；

b. 可以明确地计算其结果为真或假。

例如，“ 供应商S1在伦敦 ”、“ 供应商S2在伦敦 ”和“ 供应商S1供应零件P1 ”是项（用通常的样本值计算，其值分别为真、假和真）。相比之下，“ 供应商S1供应零件p(p是一变量) ”和“ 供应

商S5在未来某一时刻供应P1”就不是项，因为它们不能明确地计算出取真值还是取假值。

2. 公式

下面，我们定义公式的概念。命题演算（更一般地讲，谓词演算）公式是以查询表达式的形式出现在数据库中的。

```
<formula>
    ::=    <term>
         | NOT <term>
         | <term> AND <formula>
         | <term> OR <formula>
         | <term> ==> <formula>

<term>
    ::=    <atomic formula>
         | ( <formula> )
```

公式的计算是基于项的真值以及连接谓词真值表。注意：

1) 一个原子公式是一个真值表达式，其中没有连接词，也不包含括号。

2) 符号“ ==> ”表示逻辑蕴含连接。表达式 *f* ==> *g* 在逻辑上等价于表达式NOT *f* OR *g*。注意在第7章或其它章节里，我们使用“ IF...THEN... ”表示这个连接。

3) 为了表达一个请求的计算顺序，通常对这些连接词采取优先次序（即 NOT、AND、OR、 ==>，由高到低的次序）来减少要使用括号的数量。

4) 一个命题，就是上面定义的一个公式（为了与下一节保持一致性，这里使用了术语“公式”）。

3. 推理规则

下面讨论命题演算的推理规则。这样的规则有很多。其中每一条规则都可表述为下述形式的语句：

```
|-  f ==> g
```

（这里符号 |- 读作“总是有”；为了能产生某些语句 —— 即语句的语句，的确需要这样的符号）。下面是此推理规则的一些例子：

1) `|-  ( f AND g ) ==> f`

2) `|-  f ==> ( f OR g )`

3) `|-  ( ( f ==> g ) AND ( g ==> h ) ) ==> ( f ==> h )`

4) `|-  ( f AND ( f ==> g ) ) ==> g`

注意：这一推理规则特别重要。这叫假言推理规则。非正式地讲，其含义是如果 *f* 为真，且 *f* 蕴含 *g*，则 *g* 一定为真。例如，下面的 *a* 和 *b* 都为真，

a. 我没有钱；

b. 如果我没有钱，那我就不得不刷碟子；

—— 则我们就一定能推断 c 也为真：

c. 我不得不刷碟子。

下面仍是这一推理规则的一些例子：

5) `|-  ( f ==> ( g ==> h ) ) ==> ( ( f AND g ) ==> h )`

6) `|-  ( ( f OR g ) AND ( NOT g OR h ) ) ==> ( f OR h )`

注意：这是另一个特别重要的规则，叫作归结规则。在下面要讲的“证明”一节和第23.4节将详细讨论。

4. 证明方法过程

现在，我们讨论形式证明的方法问题（在命题演算的范畴）。这就是决定是否一给定的公式*g* (结论)是另一些给定的公式*f1*、*f2*、..., *fn*（前提）的逻辑结果，用符号表示就是：

```
f1, f2, ..., fn  ⊢  g
```

（读作“从*f1*、*f2*、..., *fn*推导出*g*”；注意这里使用的另一个元语言符号“⊢”）。这种基本方法叫做前向推理(forward chaining)。前向推理就是对这些前提、由这些前提导出的公式、由这些公式导出的公式，等等重复运用推理规则，直到推导出所需要的结论；或者说这一过程就是从前提到结论的“正向链”。然而，对这一基本的论题，下面还有几个变种：

1) 附加一个前提：如果*g*是*p* ⇒ *q*的形式，采用*p*作为附加的前提，并表明*q*可从给定的前提及*p*推导出来。

2) 反向推理：不直接去证明*p* ⇒ *q*，而是去证明其逆否命题，即NOT *q* ⇒ NOT *P*。

3) 反证法：不是直接去证明*p* ⇒ *q*，而是先假设*p*和NOT *q*为真，再推导出相互矛盾的结果。

4) 归结：这种方法要使用归结推理规则（如上述的第6条推理）。

现在我们来具体地讨论归结规则。它有着广泛的应用（特别地，它可以推广到谓词演算的情况。这在23.4节将会看到）。

首先要注意，归结规则是很有效的。运用它，可以消除一些子公式。如，给出两个公式：

```
f OR g     and    NOT g OR h
```

我们可以删去*g*和NOT *g*，得到如下简化的公式：

```
f OR h
```

特别地，若有*f* OR *g* 和 NOT *g*（即*h*取真值），则可以导出f。

因此，一般情况下，这些规则运用在两个与（AND）公式上，并且每一个公式是两个公式的或（OR）。下面，我们运用这一归结规则（为了使讨论更加具体化，用一特殊的例子来解释这一过程）。假设希望知道下面的假设证明是否有效：

```
A ⇒ ( B ⇒ C ), NOT D OR A, B  ⊢  D ⇒ C
```

（这里*A*、*B*、*C*和*D*都是公式）。现在，我们采用附加的前提，即否定这个结论，并把每个前提写在不同的行，如下所示：

```
A ⇒ ( B ⇒ C )
NOT D OR A
B
NOT ( D ⇒ C )
```

这四行隐含着与（AND）连接。

现在，我们把每一行转化为合取范式，即一个或一个以上的、由AND连接起来的公式，每一个独立的公式可能包含NOT和OR，但不包含AND（参看第17章）。当然，第二行和第三行已符合要求。为了转化其它两行，首先我们根据NOT和OR连接的定义，删除掉所有的符号“⇒”；接着，必要的时候运用分配律和摩根律；同时删除掉冗余的括号和邻近的NOT，这样得到:

```
NOT A OR NOT B OR C
NOT D OR A
B
D AND NOT C
```

接着，对于任何包括ANDs的行，把它转化为一系列独立的行，每一行是一个由AND连接的公式（即在这一过程中删掉AND）。此例中，这一步仅用在第四行。现在，前提看起来如下所

示：

```
NOT A OR NOT B OR C
NOT D OR A
B
D
NOT C
```

下面，我们开始运用归结规则。先选择能被归结的两行，即分别包含某个特殊的公式及其否定形式。我们选择头两行，它们分别包含 NOT *A* 和*A*，归结它们，得到：

```
NOT D OR NOT B OR C
B
D
NOT C
```

（注意：在一般情况下，我们需要保留原始的两行，但在这个特殊的例子里，我们不再需要它们了）。再一次选择头两行，运用这一规则（归结 NOT B和B），得：

```
NOT D OR C
D
NOT C
```

还是选择头两行（NOT D和D），得：

```
C
NOT C
```

再处理头两行（NOT C和C）；最后的结果是命题的空集（通常表示为 []），这是一个矛盾。这样，通过反证法，所要求的结果就被证明了。

23.4 谓词演算

现在，我们来讨论谓词演算。命题演算和谓词演算之间最大的区别在于后者允许公式中有变量 ⊖ 和量词，这使得它的功能更为强大，应用更为广泛。例如，语句“供应商 S1供应零件*p*”和“某位供应商*s*供应零件*p*”在命题演算中不合法，但在谓词演算中它们是合法的公式。因此，谓词演算给我们表达如下的查询提供了基础：“S1供应了哪些零件？”或“查询供应某一零件的供应商”，甚至“查询根本不供应任何零件的供应商”。

1. 谓词

正如第3章解释的那样，一个谓词就是一真值函数，即给出合适的自变量参数，其返回结果为真或假的函数。例如，“*x*,*y*”——更一般地写作“*x*>*y*”——是一个有着两个参数的谓词，即*x*和*y*；如果相应于*x*的变元大于相应于*y*的变元，则它返回真值；否则，返回假值。一个用了*n*个变元的谓词（即，这个谓词由*n*个参数所定义）叫做*n*元谓词。一个命题（23.3节所讲的公式）可以认为是零元谓词，没有参数，并且其结果明确地为真或假。

假设相应于“=”、“>”、“≥”等的谓词是固有的（即它们是定义的规范系统的一部分），并且用一种习惯的方式将使用它们的表达式写出来。当然，用户也应能定义他们自己的谓词。整个问题的关键就是：实际上用数据库的术语讲，一个用户定义的谓词相应于一个用户定义的关系变量（从前面几章我们也可以了解这一点）。例如，供应商的关系变量 S，可以看作有着四个参数的谓词，即S#、SNAME、STATUS和CITY。而且，表达式S（S1,Smith,20,London）和S(S6,White,45,Rome)分别表示该谓词结果为真和假的“实例”，或者“实例化”，或者“调用”。非正式地讲，正像前面几章所讲的那样（特别是第 8章），我们可以把这些谓词，与可能

⊖ 这里的变量是逻辑变量，并不是程序语言中的变量。就讨论的目的而言，可以把它当作第 7章中的所讲的范围变量。

的完整性约束(同时也是谓词）一起作为数据库内容的定义。

2. 合式公式

下一步是扩展“公式”的定义。为了避免与前面几节讲的公式（实际上，那是特殊的情形）混淆，现在我们转到第7章所讲的合式公式这一术语（WFF)。下面是一简单的合式公式的语法：

```
<wff>  ::=   <term>
           | NOT ( <wff> )
           | ( <wff> ) AND ( <wff> )
           | ( <wff> ) OR ( <wff> )
           | ( <wff> ) ==> ( <wff> )
           | EXISTS <var name> ( <wff> )
           | FORALL <var name> ( <wff> )

<term> ::= [ NOT ] <pred name> [ ( <argument commalist> ) ]
```

几点说明：

1) 简单地讲，一个项（<term>）可能不是“谓词实例”(如果把谓词看作真值函数，那么一个谓词实例就是那个函数的调用)。每一个变元（<argument>）必须是一个常量、一个变量名或一个函数调用，而函数调用的每一个变元也必须是这样。在零元谓词里可以删去变元列表（<argument commalist>）(可选的)及相应的括号。注意：为了WFF能使用包括诸如“+ (*x*,*y*)”(更一般地写作“*x*+*y*”）的计算表达式，可以使用建立在作为谓词的函数上的函数。

2) 正如23.3节讲的那样，为了减少因表达计算顺序而引入的括号数目，我们对连接词使用优先规则(即NOT、AND、OR、==>、按优先级排列)。

3) 假设大家熟悉量词EXISTS和FORALL。注意：这里讲的仅是一阶谓词演算，其基本意思是：(a)它没有“谓词变量”(即允许值是谓词的变量)，因此(b)谓词本身并没有受到约束。具体请看第7章练习7.9。

4) 摩根律可以推广应用到合式公式，如下：

```
NOT ( FORALL x ( f ) )  =  EXISTS x ( NOT ( f ) )
NOT ( EXISTS x ( f ) )  =  FORALL x ( NOT ( f ) )
```

这一点也将在第7章讨论。

5) 现在重复一下第7章另一个要点：在一个给定的WFF里，每一个变量的引用要么是自由的，要么是约束的。一个引用是约束的，是指 (a)它紧跟一个量词（即它表示约束的变量）；或者(b)位于量词的范围内并参照可适用的的约束变量。一个变量引用是自由的，当且仅当它不是约束的。

6) 一个封闭的合式公式不包含自由变量（实际上，它是一个命题)。一个开放的合式公式就是一个不封闭的合式公式。

3. 释义和模型

合式公式是什么意思呢？为了形式化地回答这个问题，我们引入了释义（interpretation）这一概念。一系列合式公式的释义定义如下：

- 首先，我们指定要解释这些合式公式所需要的论域（universe of discourse)。或者说，在 (a)规范系统所允许的常量（用数据库的术语讲，就是域值）和 (b)真实世界的对象之间指定一映象。每一个单个的常量明确地对应于论域中的一个对象。
- 第二，根据论域中的对象，为每个谓词指定一个含义。
- 最后，根据论域中的对象，为每个函数指定一个含义。

这样，这一释义就包括论域、论域中的对象与常量个体之间的映象、根据论域对谓词和函数所作的定义。

下面举例说明。设论域是整数集{0,1,2,3,4,5}，常量2以明显的方式对应于此论域中的某对象，设谓词“ *x*>*y* ”定义成通常的含义（若需要的话，也可定义诸如“ + ”、“ - ”等函数）。现在，我们给下面的合式公式赋真值，如下所示：

```
2 > 1                    :  true
2 > 3                    :  false
EXISTS x ( x > 2 )       :  true
FORALL x ( x > 2 )       :  false
```

但是，注意，存在其它的释义是可能的。例如，我们可以给论域取一系列安全分类级别，如下：

```
destroy before reading (level 5)
destroy after reading  (level 4)
top secret             (level 3)
secret                 (level 2)
confidential           (level 1)
unclassified           (level 0)
```

这里谓词“ > ”就表示“ 更安全的（即更高的安全级别）”。

现在，大家可能觉得上面的两个可能的释义在形式上是一样的，即在它们之间建立一一对应关系是可能的，且从更深一层次讲，这两个释义其实就是一个。但是必须知道，释义可以存在真正的不同。例如，我们把论域定义为 0~5的整数，但是定义谓词“ > ”为相等（当然，这样做会引起混淆，但至少这是合法的）。现在上面的第一个合式公式就取假而不是取真。

另一点必须明确的是，从前述的意义讲，这两个释义可能是真正的不同，但是对某些给定合式公式集，它们都取同样的真值。在我们的例子里，如果删除合式公式“ 2 > 1 ”，则对于“ > ”的两个不同的定义，就会出现这种情况。

还需要注意的是，迄今为止，这一小节所讨论的合式公式都是封闭式合式公式。这是因为对每一个给定的封闭的合式公式，总能明确地给它赋一个真值。但是，开放的合式公式的真值依赖于赋给自由变量的值。例如，开放的合式公式：

```
x > 3
```

很明显，只要 *x*比3大，其值为真，否则其值为假（无论“ 大于 ”和“ 3 ”在这里表示 什么）。

现在，我们来解释被释义的一组合式公式（必须是封闭的）的模型，对于这个释义，所有其中的合式公式为真。对于整数0~5，考虑下面四个合式公式：

```
2 > 1
2 > 3
EXISTS x ( x > 2 )
FORALL x ( x > 2 )
```

上面给出的两个释义并不是这些合式公式的模型。因为在那种释义下，其中某些合式公式结果为假。相比之下，上面的第一个释义（即“ > ”定义为“ 大于 ”）应该是下面合式公式的模型：

```
2 > 1
3 > 2
EXISTS x ( x > 2 )
FORALL x ( x > 2 OR NOT ( x > 2 ) )
```

最后还要注意，因为对于给定的一组合式公式能接受几种释义，且其中的合式公式为真，因此它就有几个模型（一般地讲）。所以，用模型理论的观点来讲，一个数据库仅仅是一组合

式公式。具体见23.5节。

4. 子句式

就像任何一个命题演算公式都可转化为合取范式一样，任何一个谓词演算合式公式都可转化为子句式，它被认为是合取范式的扩展形式。进行这种转化的动机在于，我们可以把归结规则运用于构建和验证证明。

这一转化过程如下（这里是概括性的，详细请看 [23.10]）。通过下述例子来解释这些步骤。

```
FORALL x ( p ( x ) AND EXISTS y ( FORALL z ( q ( y, z ) ) ) )
```

这里*p*和*q*是谓词，*x*、*y*和*z*是变量。

1) 删除符号“ ⇒ ”。此例中，这个转化没有影响。

2) 使用摩根定律，删去两个邻近的 NOT。删去NOT，以便使它仅适合于项，而不适合一般的合式公式。该转化仍对此例没有影响。

3) 把所有的量词都移到前面，从而将这些公式转化为前束范式。如果有必要，可系统地重命名变量：

```
FORALL x ( EXISTS y ( FORALL z ( p ( x ) AND q ( y, z ) ) ) )
```

4) 注意一个有存在量词的量化的合式公式：

```
EXISTS v ( r ( v ) )
```

等价于合式公式：

```
r ( a )
```

其中*a*为未知常量。最初的合式公式中存在 *a*这样的未知常量，我们不知道其值。同样，对于合式公式：

```
FORALL u ( EXISTS v ( s ( u, v ) ) )
```

等价于合式公式：

```
FORALL u ( s ( u, f ( u ) ) )
```

其中，全称量词*u*的函数*f*是未知的。这里常量*a*和函数*f*分别命名为斯科林（Skloem）常量和斯科林（Skloem）函数，这一命名来自于逻辑学家T.A.Skloem（注意：一个Skloem常量仅是一个没有变元的Skloem函数）。因此，下一步就是通过取代相应的约束变量而删除存在量词。其中，这些变量由下式的量词前面的全称量词的Skloem函数（任意的）所限制：

```
FORALL x ( FORALL z ( p ( x ) AND q ( f ( x ), z ) ) )
```

5) 现在，所有的变量都被全称量词约束。因此，我们可以习惯上将所有的变量都隐含为全称量词约束，删除所有的显式量词：

```
p ( x ) AND q ( f ( x ), z )
```

6) 将合式公式转化为合取范式，即由 AND连接的子句，每一子句只可能有 NOT或OR，而没有 AND。此例中，此合式公式已是这种形式。

7) 删去AND，把每个子句写在独立的行上，如下：

```
p ( x )
q ( f ( x ), z )
```

这一子句等价于最初的合式公式。

注意：可以从上述的过程得出，用子句形式的一个合式公式的一般形式就是一组子句，且每一个占一行，形式如下：

```
NOT A1 OR NOT A2 OR ... OR NOT Am OR B1 OR B2 OR ... OR Bn
```

这里A和B都是正项。我们可以转化这个子句，如下所示：

```
A1 AND A2 AND ... AND Am  ==>  B1 OR B2 OR ... OR Bn
```

如果最多有一个B（n=0或1），则这个子句叫做Horn子句，它是以逻辑学家Alfred Horn命名的。

5. 使用归结规则

下面讨论基于逻辑的数据库系统对查询的处理过程。我们沿用23.2节结束时的那个例子。首先，有谓词MOTHER_OF，它包括两个参数，分别表示母亲与女儿，再给出下面两个项（谓词实例）：

```
1) MOTHER_OF ( Anne,  Betty )
2) MOTHER_OF ( Betty, Celia )
```

给出下面的合式公式（演绎公理）：

```
3) MOTHER_OF ( x, y ) AND MOTHER_OF ( y, z ) ==>
                                  GRANDMOTHER_OF ( x, z )
```

（注意这是一个Horn子句）。为了简化归结规则的运用，我们删去上式中的符号“ ”，如下：

```
4) NOT MOTHER_OF ( x, y ) OR NOT MOTHER_OF ( y, z ) OR
                                  GRANDMOTHER_OF ( x, z )
```

现在证明Anne是Celia的祖母——即如何回答查询“Anne是Celia的祖母吗？”，现在我们否定这个已经证明了的结论，而把它作为前提：

```
5) NOT GRANDMOTHER_OF ( Anne, Celia )
```

为了应用归结规则，必须能找到两个子句，分别包含一个合式公式及它的否定公式，并系统地给变量赋值。这样的赋值是合法的，因为这些变量都已隐含为全称量词约束，所以对于每一种变量的值的组合，单个的合式公式（非否定的）必须为真。注意：寻找一系列使这两个子句可归结的值的过程叫做合一。

下面用例子表明上述过程如何进行，注意到4和5分别包含项GRANDMOTHER_OF(x,z)和NOT GRANDMOTHER_OF(Anne,Celia)。因此，我们用Anne取代x，用Celia取代y并归结，得：

```
6) NOT MOTHER_OF ( Anne, y ) OR NOT MOTHER_OF ( y, Celia )
```

第二条包括MOTHER_OF(Betty,Celia)。我们用Betty取代上面6中的y并归结，得：

```
7) NOT MOTHER_OF ( Anne, Betty )
```

归结上面的7和1，我们得空子句集，从而矛盾。所以，起初查询的答案应是“是的，Anne是Ceila的祖母”。

对于查询“Anne的孙女是谁？”又怎么样呢？首先注意到，系统不知道孙女，它只知道祖母。我们可增加另一演绎公理，用于说明z是x的孙女，当且仅当x是z的祖母（此数据库中不允许有男性）。当然，也可以把这个问题叙述为“Anne是谁的祖母？”。我们来讨论后面一个公式。前提是（和以前一样）：

```
1) MOTHER_OF ( Anne,  Betty )
2) MOTHER_OF ( Betty, Celia )
```

3) NOT MOTHER_OF (x, y) OR NOT MOTHER_OF (y, z) OR
GRANDMOTHER_OF (x, z)

我们引入第四个前提，如下：

4) NOT GRANDMOTHER_OF (Anne, r) OR RESULT (r)

这个新前提直观地表明要么Anne不是任何人的祖母，要么有某个人*r*属于这一结果（因为她是*r*的祖母）。我们希望发现*r*的身份。可如下进行：

首先，用Anne取代*x*，*r*取代*z*，并归结上面的3和4，得：

5) NOT MOTHER_OF (Anne, y) OR NOT MOTHER_OF (y, z)
OR RESULT (z)

接着，用Betty取代*y*，并归结上面的5和1，得：

6) NOT MOTHER_OF (Betty, z) OR RESULT (z)

现在，Celia取代*z*，并归结上面的6和2，得：

7) RESULT (Celia)

因此，Anne是Celia的祖母。

注意：如果给出一个附加项，如下：

MOTHER_OF (Betty, Delia)

这样我们就能在最后一步用Delia（不是Celia）取代*z*，得：

RESULT (Delia)

当然，在查询结果中，用户期望看到两个名字。这样，系统就需要运用合一和归结过程去生成所有可能的结果。进一步的细节已超出本讨论的范围。

23.5 数据库的证明理论观点

正像23.4节解释的那样，一个子句就是下述形式的一个表达式：

A1 AND *A2* AND ... AND *Am* ⇒ *B1* OR *B2* OR ... OR *Bn*

这里*A*和*B*都是下述形式的项：

r (*x1*, *x2*, ..., *xt*)

(其中*r*是谓词，*x1*、*x2*、...，*xt*是它的变元)。参看[23.12]，考虑该一般结构中两个重要的特殊的情况：

- 情况1：*m*=0, *n*=1

  这种情况下，这个子句可简单地写为：

  ⇒ *B1*

  或者写为（删去蕴含符号）：

  r (*x1*, *x2*, ..., *xt*)

  其中*r*为谓词，*x1*、*x2*、...，*xt*为变元。如果*x*都是常量，则此子句表示一个基本公理（ground axiom）——即其结果明确地为真。在数据库项里，这样的语句相对应于关系变量*R*的一个元组 ㊀。谓词*r*相对应于关系变量*R*的“含义”，本书的其它一些地方解释过这一点。例如，在供应商和零件数据库里，有一个关系变量叫SP，它的意思是某一个供应商（S#）以某一数量(QTY)供应某一零件(P#)。这个含义相对应于一个开放式合式公式，因为它包含自由变

㊀ 或者对应于域中的一个值。

量（S#，P#，QTY）的引用。相比之下，元组（S1，P1，300）——其变元都是常量——是一个基本公理或封闭式合式公式，它明确地表示供应商S1供应了300个零件P1。

- 情况2：$m>0$, $n=1$

在这种情况下，子句采用下述形式：

```
A1 AND A2 AND ... AND Am  =►  B
```

这叫做演绎公理；它根据蕴含符号左边的谓词给右边的谓词定义（此定义可能不完全）（参看前面例子中的GRANDMOTHER_OF谓词的定义）。

或者，这样的子句可被定义为完整性约束（使用第8章的术语，即一个关系变量约束）。例如，供应商变量S有两个属性S#和CITY，则子句：

```
S ( s, c1 ) AND S ( s, c2 )  =►  c1 = c2
```

表达了这样的约束：CITY函数依赖于S#。注意这里固有谓词“ = ”的使用。

正如前面讨论所解释的那样，关系（“基本公理”）中的元组、导出关系（“演绎公理”）及完整性约束是一般子句结构的特殊情形。现在来看一看对于23.2节所讲的数据库的“证明理论”的观点，是如何由这些思想导出的。

首先，数据库的传统观点被认为是模型理论。这里讲“传统观点”，是指数据库被理解为由显式命名关系变量的集合组成，每一个关系变量又由一系列显式元组及显式完整性约束组成。正是基于这种理解，所以这种观点刻画为模型理论。看下面的解释：

- 基本（underlying）的域包括值或常量，它们被假设代表“现实世界”（更精确地讲，像23.4节那样，是某种释义）中一定的对象。这样它们就对应于论域。
- 关系变量（更精确地讲，是关系变量头部）表示一系列谓词或者开放式合式公式，且这些合式公式在此论域上释义。例如，关系变量头SP表示谓词“供应商S#以数量QTY供应零件P1”。
- 给定的关系变量里的每一个元组表示一个相应谓词的实例；即表示在这个论域里结果明确为真的命题（一个封闭的合式公式，它没有变量）。
- 完整性约束也是封闭式合式公式，并且它们在同一域上释义。所以，这些数据并没有（不可能）违背这些约束，这些约束结果必须为真。
- 元组和完整性约束在一起就被看作一组公理，它定义某种逻辑理论（不严格地讲，一个“理论”在逻辑上就是一组公理）。由于在这一释义里，这些公式为真，所以从23.4节所述的意义讲，该释义就是逻辑理论的模型。注意到正如前一节所指出的，这个模型不是唯一的，即一个给定的数据库可能有几个释义，从逻辑的角度讲，所有这些释义都是同样有效的。

因此，在模型理论的观点里，按“模型”的术语讲，数据库的含义就是模型。并且由于有许多可能的模型，故有许多可能的意义，至少从理论上讲是这样的[⊖]。而且，在模型理论观点下的查询过程基本上就是计算某一个开放的合式公式的过程。这一过程是为了发现在这一模型里，合式公式中哪一个自由变量使得这个合式公式取真值。

⊖ 然而，如果我们假设数据库不显式地包含任何否定信息（即一个“ NOT S#（S9）”形式的命题，表示S9不是一个供应商号），则将有一个最小的或规范的意义，这就是所有可能模型的交集（参看 [23.10]）。而且，在这种情况下，这一规范的意义与在证明理论的观点下的数据库的意义一样。这会在以后的适当时候解释。

模型理论的观点内容很丰富。然而，为了能运用 23.3节和23.4节所讲的推理规则，就必须采用一个不同的视角。在这个视角中，数据库被明确地看作一定的逻辑理论，即作为一系列公理。这样，数据库的意义就精确地变成所有真语句的集合，而这些真语句是从一些公理演绎来的，即能由那些公理证明的定理。这就是证明理论的观点。用这一观点，查询工作就变成是一个证明理论的过程（任何情况下，从概念上讲是这样的；但是，为了有更好的效率，系统可能还要使用更多的传统的查询技术。23.7节再讲这一点）。

注意：从前面一段可以得出，直观上模型理论与证明理论的观点间的区别是，在模型理论的观点中，一个数据库有许多“意义”，而在证明理论的观点中，一个数据库仅有一个“意义”。但是（a）正如前面指出的，在模型理论中，数据库的意义是真正的规范的意义，并在任何情况下，（b）一般地讲，如果数据库包含任何一个否定的公理，则在证明理论情形下，数据库只有一个意义就不再成立[参看23.9~23.10]。

证明理论的观点中，给定数据库的公理可总结如下（非正式的）：

1) 基本公理相对于库关系变量中域值或元组值。这些公理有时组成了所谓的外延数据库（相对于内涵数据库，见下一节）。
2) 每一个关系变量的“完整公理(completion axiom)”表明，我们所讲的关系变量中有效元组的失效可被解释为相对于此元组的命题是假的（当然，实际上，这些完整化公理在一起就构成了封闭世界假说，这在第7章已经讨论了）。例如，供应商变量S不包括元组（S6,White,45,Rome）就是命题“存在供应商S6，它的名字是White,它的等级是45，它住在罗马”为假。
3) “唯一命名”公理，表明每一个常量都不同于其它的任何常量，即它有唯一的命名。
4) “域闭包”公理，表明不存在数据库域以外的常量。
5) 一组定义固有等价谓词的基本公理。需要这组公理，是因为上面的 2、3、4利用了等价谓词。

现在，我们对模型理论和证明理论这两个概念之间的基本区别作简短的小结。首先，有人说，仅从实用的角度讲，它们之间根本没有什么区别，至少从今天的 DBMS的角度讲是这样的。但是：

- 对于证明理论的观点，上述 2~5作了一定的明确假设，而在模型理论的观点中，这一假设隐含在释义的概念中（参看 [23.15]）。明确地列出这些假设在一般情况下是一好的思想；而且，为了能应用一般的证明理论的技术（如在 23.3节和23.4节讲的归一方法），明确指定这些公理是必要的。
- 注意到上述公理没有讲到完整性约束规则。这是因为如果加上这些约束（在证明理论的观点里），系统就变为演绎DBMS。23.6节讨论这一问题。
- 证明理论的观点确实有一定的优雅性，而模型理论的观点则没有。因为它为一些结构提供了统一的理解，而这些结构通常被认为有或多或少的不同。这些结构有：基本数据、查询、完整性约束（虽然是以前的观点）、虚拟数据，等等。所以，更统一的接口和更统一的实现的可能性就提高了。
- 证明理论的观点也为那些关系系统在传统上有处理困难的问题提供了良好的基础。如析取信息（例：供应商S6住在伦敦还是巴黎），否定信息导出（例：谁不是供应商？）、递归查询（下一节讲到）等。对递归查询，尽管几个商业系统已能够加以处理（参看附录

B），但原则上还不知道为什么经典关系系统不能被合适地扩展去解决此类查询。我们将在23.6节和23.7节详细讨论这一问题。

- 最后，引用Reiter的话(参看[23.15])，就是证明理论的观点“为关系模型（扩展的）和现实世界语义的结合提供了正确的处理方法”（23.2节还将讲到）。

23.6 演绎数据库系统

一个演绎DBMS就是支持证明理论观点的DBMS。特别地，对给定的外延数据库中的事实运用指定的演绎公理或推理规则，就可以从这些事实演绎或推导别的事实 ⊖。演绎公理和完整性规则（下面将讨论）合在一起有时就叫做内涵数据库。外延数据库和内涵数据库合在一起就构成了通常讲的演绎数据库（这不是一个很恰当的术语，因为它是运用演绎原理的DBMS，而不是数据库）。

正如刚刚所讲的，演绎公理构成了内涵数据库的一部分，而另一部分是表示完整性约束的公理（这些规则的主要目的是约束更新，实际上它们也可以用在从给定事实演绎另外的事实的推理过程中）。

现在我们来看一看图3-8所示的供应商和零件数据库，其在“演绎DBMS”中的形式如何。首先，这里有一系列基本公理定义了一些合法的域值。注意：在这里，为了可读性，我们采用了图3-8中表示值的同样的方法，如表示300在习惯上就可简写为QTY（300）：

```
S# ( S1 )    NAME ( Smith )    STATUS (  5 )    CITY ( London )
S# ( S2 )    NAME ( Jones )    STATUS ( 10 )    CITY ( Paris  )
S# ( S3 )    NAME ( Blake )    STATUS ( 15 )    CITY ( Rome   )
S# ( S4 )    NAME ( Clark )    等等             CITY ( Athens )
S# ( S5 )    NAME ( Adams )                     等等
S# ( S6 )    NAME ( White )
S# ( S7 )    NAME ( Nut   )
等等         NAME ( Bolt  )
             NAME ( Screw )
             等等
```

等等。

对基本关系中的元组有如下的基本公理：

```
S ( S1, Smith, 20, London )
S ( S2, Jones, 10, Paris  )
等等

P ( P1, Nut, Red, 12, London )
等等

SP ( S1, P1, 300 )
等等
```

注意：我们并没有严格地表明上面列出的基本公理将明确地产生外延数据库；不过，我们将使用传统的数据定义和数据登录方法。或者说，演绎DBMS直接作用在已存在的由传统方法构建的数据库上。然而要注意的是，外延数据库不违背那些已定义的完整性约束！——因为一个违背任何如此约束的数据库就表示了这一系列公理的不一致性（从逻辑上讲），并且由此可知，任何命题都可证明为“真”（或者说，可导出矛盾）。正是这一原因，要求完整性约束集是一致的就显得重要。

现在来看外延数据库。下面是一些属性约束：

⊖ 在这种推导里，值得注意的是，在1974年，Codd就认为关系模型中的目标之一是“把事实检索和文件管理域合并起来为后来商业中的推理性服务作好另外的准备”（参看[11.2],[25.8]）。

```
S ( s, sn, st, sc )  =▸  S# ( s ) AND
                         NAME ( sn ) AND
                         STATUS ( st ) AND
                         CITY ( sc )
P ( p, pn, pl, pw, pc )  =▸  P# ( p ) AND
                             NAME ( pn ) AND
                             COLOR ( pl ) AND
                             WEIGHT ( pw ) AND
                             CITY ( pc )
```

等等

候选码约束：

```
S ( s, sn1, st1, sc1 ) AND S ( s, sn2, st2, sc2 )
                       =▸  sn1 = sn2 AND
                           st1 = st2 AND
                           sc1 = sc2
```

等等

外码约束：

```
SP ( s, p, q )  =▸  S ( s, sn, st, sc ) AND
                    P ( p, pn, pl, pw, pc )
```

等等。注意：为了说明的方便，我们假设在蕴含符号右边而不是左边的变量（即此例中的 *sn*、*st*等）被存在量词所约束（所有其它变量就像 23.4节讲的那样，被全称量词约束）。从技术上讲，我们需要一些斯科林函数；例如，实际上， *sn*应该由SN(*s*)所替代，这里SN就是一个斯科林函数。

还要注意，上面的大部分约束不仅仅是 23.5节所讲的意义上的子句，因为上式右边不仅仅是简单项的逻辑和。

下面，我们再增加一些演绎公理；

```
S ( s, sn, st, sc ) AND st > 15
                    =▸  GOOD_SUPPLIER ( s, st, sc )
```

(比较第9章9.1节中GOOD_SUPPLIER视图的定义)

```
S ( sx, sxn, sxt, sc ) AND S ( sy, syn, syt, sc )
                               =▸  SS_COLOCATED ( sx, sy )

S ( s, sn, st, c ) AND P ( p, pn, pl, pw, c )
                               =▸  SP_COLOCATED ( s, p )
```

等等。

为了使这个例子更具说明性，我们把这个数据库扩展到包括“零件结构”这一关系变量，它表明零件*px*由哪些零件*py*构成它的组件（只涉及第一级的）。第一个约束要表明*px*和*py*都必须能标识存在的零件：

```
PART_STRUCTURE ( px, py )  =▸  P ( px, xn, xl, xw, xc ) AND
                               P ( py, yn, yl, yw, yc )
```

数据值包括：

```
PART_STRUCTURE ( P1, P2 )
PART_STRUCTURE ( P1, P3 )
PART_STRUCTURE ( P2, P3 )
PART_STRUCTURE ( P2, P4 )
等等
```

(实际上，PART_STRUCTURE也可能有一个“数量”变元，去表明一个 *px*由多少个*py*组成，但为了简单起见，我们删去了这一细节)。

下面增加两个演绎公理来解释零件*px*由零件*py*（任何一级的）组成是什么意思；

```
PART_STRUCTURE ( px, py )  =▸  COMPONENT_OF ( px, py )

PART_STRUCTURE ( px, pz )  AND COMPONENT_OF ( pz, py )
                           =▸  COMPONENT_OF ( px, py )
```

或者说，如果零件*py*（某一级的）是*px*的中间组件或是*pz*（某一级的）的中间组件，而*pz*又是*px*的中间组件，则*py*就是*px*的组件。注意到第二个公理是递归的，它又根据它本身定义了COMPONENT_OF谓词 ㊀。相比而言，在关系系统的历史上，是不允许如此递归地定义视图（或查询或完整性约束或……）。演绎DBMS相对于经典关系系统来说，支持递归能力是其最直接的最明显的区别之一。尽管，就像23.5节和第6章所说的那样，对于为什么经典的关系系统不能扩展去支持如此的递归，还没有什么根本的解释；但某些系统确实已经开始支持这一点。

我们将在23.7节详细讨论递归。

Datalog

从前面的讨论很明确地看到，演绎DBMS的大部分内容表现为一种语言，在这一语言里，演绎公理（通常叫规则）可以公式化。这样的语言最著名的例子就是Datalog语言（和Prolog类似）（参看[23.9]）。这一节里我们对Datalog语言作简单的介绍。注意：Datalog语言的重点是它的描述能力，而不是它的计算能力（实际上，它与最初关系模型那种情况类似，参看[5.1]）。其目标就是定义一种语言，使它最终能比传统的关系语言有更大的表达能力（参看[23.9]）。所以，Datalog中强调的（的确，一般地讲，这一强调贯穿于整个基于逻辑的系统中）是对查询的处理，而不是更新，尽管这是可能的，并也要求扩展这门语言去支持更新（以后将看到）。

Datalog语言用它最简单的形式，对于那些像简单的Horn子句且没有函数的规则，作了公式化的支持。在23.4节里，我们把Horn子句定义为下述形式的合式公式：

```
A1 AND A2 AND ... AND An

A1 AND A2 AND ... AND An  =►  B
```

（这里的*A*和*B*是仅包含常量和变量的谓词的实例）。然而，根据Prolog的格式，实际上，Datalog将这第二种形式变形，即：

```
B  ◄=  A1 AND A2 AND ... AND An
```

为了和此领域的其它版本保持一致，下面我们作同样处理。

在这样一个子句中，*B*是规则头（或结论），*A*是规则体（或前提或目标；每一个个体是子目标）。为简洁起见，这里的AND通常都用逗号代替。一个Datalog程序是由某些传统的方式分割的一系列子句，例如，可由分号隔开（本书里不使用分号，而是简单地将每一个子句写在单独的一行）。在这样的程序里，子句的顺序是无所谓的。

从上述所讲的意义看，整个“演绎数据库”都可看作Datalog程序。例如，我们可以把上述所有的关于供应商和零件数据库的公理（基本公理、完整性约束、演绎公理）用Datalog的形式写出来，用分号把它们分开，或把它们写在单独的行上，这样，其结果就是Datalog程序。然而，正如早些时候所说的那样，数据库的扩展部分不能用这种方式详细说明，但能用别的更传统的方法。这样，Datalog语言的主要目标就是明确地支持演绎公理的公式化。前面曾指出过，函数可以看成传统的关系DBMS中视图定义机制的扩展。

Datalog语言也可用作查询语言（这里，也很像Prolog）。例如，假设我们对GOOD_SUPPLIER作出Datalog的定义，如下：

㊀ 当然，实际上，我们已经定义了一个传递闭包。在任何时刻，相应于COMPONENT_OF的关系都是相对于PART_STRUCTURE的关系的一个闭包（参看第6章）。

```
GOOD_SUPPLIER ( s, st, sc )  <=  S ( s, sn, st, sc )
                                 AND st > 15
```

下面是有关GOOD_SUPPLIER的典型的查询：

1) 查询所有“好”供应商：

```
?  <=  GOOD_SUPPLIER ( s, st, sc )
```

2) 查询在巴黎的“好”供应商：

```
?  <=  GOOD_SUPPLIER ( s, st, Paris )
```

3) 供应商S1是“好”供应商吗？

```
?  <=  GOOD_SUPPLIER ( S1, st, sc )
```

等等。或者说，Datalog查询由一个以“？”为头的特殊的规则组成，其规则体由一个唯一的项组成，从此项可得查询结果。“？”按其习惯用法表示“显示”(display)。

应当指出，尽管最初定义的Datalog语言支持递归，但还有许多传统的关系数据库语言的特征它不支持，如：标量运算（“+”、“*”等）、聚集运算（COUNT、SUM等）、差运算（因为子句不能被否定）、分组运算和不分组运算(ungrouping)等。它还不支持属性命名（谓词变元依赖于它初始位置），不支持全域（即第5章讲的用户自定义类型）。本节早些时候说过，它不提供任何更新操作，（由此造成的结果）也不支持定义过的外码删除及更新规则的说明(ON DELETE CASEADE等等)。

为了弥补上述这些不足，已经对Datalog语言提出了各种各样的的扩展，这些扩展试图提供下述特征：

- 否定的前提——例如：

```
SS_COLOCATED ( sx, sy )  <=  S ( sx, sxn, sxt, sc ) AND
                             S ( sy, syn, syt, sc ) AND
                             NOT ( sx = sy )
```

- 标量操作（固有的或用户自定义的）——例如：

```
P_WT_IN_GRAMS ( p, pn, pl, pg, pc )  <=
                  P ( p, pn, pl, pw, pc ) AND pg = pw * 454
```

此例中，我们假设符号“*”用传统的记法写在中间。此项也可运用AND的更传统的前缀逻辑表示法为“=(*pg*,*(*pw*,454))”。

- 分组和聚集操作（与关系的SUMMARIZE操作相似，参看第6章）：有时为了叙述总量的问题，这样的操作是必需的；这些问题查询的不仅仅是零件*px*由哪些零件*py*组成，而且要查询组成零件*px*需要多少个零件*py*。（这里我们假设关系变量PART_STRUCTURE包括QTY属性）。
- 更新操作：解决这个要求的一种方法（不仅仅是这种方法）是基于对基本的Datalog语言的观察，(a) 在规则头里面的任何谓词必须是非否定的，并且（b）规则生成的每一个元组可看作是在结果中的“插入”。这样，一个可能的扩充是允许在规则头里面有否定的谓词，且认为这些谓词是请求删除（有关的元组）。
- 规则体里的非Horn子句。或者说在规则的定义中允许有完全通用的合式公式。

Gardarin和Valduriez在[23.10]中举例综述了以上的扩展内容，同时还讨论了各种各样的Datalog语言的实现技术。

23.7 递归查询过程

正如前一节里所述的，演绎数据库里最值得注意的特征是它对递归的支持。这里的递归

包括递归规则的定义及由此决定的递归查询。所以，最近的几年对这样的递归的实现技术有着大量的研究——的确，大约从1986年以来，每一次数据库会议都至少提交这样的一篇论文(请看本章“参考文献”部分)。由于递归查询表达了经典DBMSs中不存在的问题，故本节我们只简单地讨论一下。

下面我们举例说明。仍以23.6节COMPONENT_OF的递归定义来说，它是由PART_STRUCTURE定义的。为简单起见，我们把COMPONENT_OF简写为COMP，把PART_STRUCTURE简写为PS；同时，把这一定义转化为Datalog的形式，如下：

```
COMP ( px, py )  <=  PS ( px, py )

COMP ( px, py )  <=  PS ( px, pz ) AND COMP ( pz, py )
```

下面是基于这一数据库的典型的递归查询(“探寻零件P1”)：

```
?  <=  COMP ( P1, py )
```

现在回过头来看一看刚才的定义：该定义中的第二个规则，即递归规则，是线性递归，因为规则头里面的谓词在规则体里面仅出现一次。相比之下，下面COMP的定义中的第二个规则（递归的）从同一种意义上讲并不是线性递归的：

```
COMP ( px, py )  <=  PS ( px, py )

COMP ( px, py )  <=  COMP ( px, pz ) AND COMP ( pz, py )
```

然而，一般情况下总是认为线性递归表示着“令人感兴趣的情形”。某种意义上讲，在实际中出现的大多数递归都呈线性特征，而且对线性递归已经存在有效处理技术[23.16]。因此，本节的讨论只限于线性递归的范畴。

注意：为了叙述的完整性，有必要对递归规则（包括线性递归）的定义加以概括，以便处理如下列出的更为复杂的情形：

```
P ( x, y )  <=  Q ( x, z ) AND R ( z, y )

Q ( x, y )  <=  P ( x, z ) AND S ( z, y )
```

为了简单起见，这里我们忽略了细节；具体详细的讨论参见[23.16]。

在经典的查询过程（即非递归的）中，实现一个递归查询的全部问题被分解为两个子问题，即（a）把初始查询转化为一些等价的且更有效的形式，接着（b）实际执行这一转化结果。本书描述了对这两个问题的不同解法（参看“参考文献”部分）。本节里，我们粗略地讨论这些简单的技术，显示了这些技术在下述样本数据上对查询“探寻零件P1”的求解过程：

PS

PX	PY
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6

1. 合一与归结

当然，一个可能的方法是使用标准的Prolog技术，这就是在23.4节描述的合一与归结技术。此例中，这种方法的过程如下所示。第一个前提是演绎公理，在合取范式中它看起来如下所示：

```
1) NOT PS ( px, py ) OR COMP ( px, py )
2) NOT PS ( px, pz ) OR NOT COMP ( pz, py ) OR COMP ( px, py )
```

依据所期望的结论，我们再构造另外一个前提：

3) NOT COMP (P1, *py*) OR RESULT (*py*)

基本公理构成了余下的前提。例如，考虑基本公理：

4) PS (P1, P2)

用P1取代上述1中的*px*，P2取代上述1中的*py*，我们可以归结上面的1和4，得：

5) COMP (P1, P2)

现在用P2取代3中的*py*，并归结3和5，得：

6) RESULT (P2)

因此，P2是P1的组件。类似地，P3也是P1的组件。由此得到公理COMP（P1，P2）和COMP（P1，P3）；递归运用上述过程，并确定最终结果。具体细节留做练习。

实际中，合一与归结的处理代价是大的，因此需要寻找一些更有效的处理策略。下一小节讨论对这一问题的一些可能的方法。

2. 朴质计算

朴质计算(naive evaluation)（参见[23.25]）可能是所有方法中最简单的方法，正如它的名字所说的那样，这一算法是非常纯朴的。仍以前述简单查询为例，用下述伪代码可以很容易地解释清楚这一过程：

```
COMP := PS ;
do until COMP reaches a "fixpoint" ;
  COMP := COMP UNION ( COMP ⋇ PS ) ;
end ;
DISPLAY := COMP WHERE PX = P# ('P1') ;
```

对于关系变量COMP和DISPLAY（类似关系变量PS），均有两个属性，即PX和PY。不严格地讲，这种方法就是重复地组合中间结果，直到它达到一个不动点——即它停止生长。这一中间结果是：PS与前面中间结果的连接的并。注意：表达式“ COMP ⋇ PS ”是“ COMP和PS在COMP.PX和COMP.PY上进行连接，然后再在其上进行投影 ”的缩写；为简单起见，我们忽略了属性重命名操作，而这在代数里是不能忽略的（参看第 6章）。

现在用实际数据来看看这一算法的过程。经过第一次循环迭代之后，表达式 COMP ⋇ PS的值如下左表所示， COMP的结果如下右表所示（此迭代过程中增加的元组用星号作标记）：

COMP ⋇ PS

PX	PY
P1	P3
P1	P4
P1	P5
P2	P5
P3	P6
P4	P6

COMP

PX	PY	
P1	P2	
P1	P3	
P2	P3	
P2	P4	
P3	P5	
P4	P5	
P5	P6	
P1	P4	*
P1	P5	*
P2	P5	*
P3	P6	*
P4	P6	*

第二次迭代之后，其结果如下：

看一看第二次迭代计算CP ⋇ PS，它重复利用了第一次迭中COMP ⋇ PS的计算结果，且又增加了两个另外的元组（即上表中的（ P1，P6）和（P2，P6））。这就是为什么朴质算法并

COMP * PS

PX	PY
P1	P3
P1	P4
P1	P5
P2	P5
P3	P6
P4	P6
P1	P6
P2	P6

COMP

PX	PY	
P1	P2	
P1	P3	
P2	P3	
P2	P4	
P3	P5	
P4	P5	
P5	P6	
P1	P4	
P1	P5	
P2	P5	
P3	P6	
P4	P6	
P1	P6	*
P2	P6	*

不是很智能的原因。

第三次迭代及更多的重复计算之后，MP * PS的值不再发生变化，COMP达到一不动点，这时我们退出循环。最后的计算结果就是COMP的限制子集：

COMP

PX	PY
P1	P2
P1	P3
P1	P4
P1	P5
P1	P6

这里有一显著的副作用：这一算法有效地计算了对每一个零件的探寻，实际上，它已经计算了关系PS的整个传递闭包，但是只保留了所需要的元组，其它元组都抛弃掉了，所以白白地做了许多工作。

质朴计算技术被看作是前向推理的应用：从外延数据库（即实际的数据值）开始，它重复运用定义的前提（即规则体）直到达到所要的结果。事实上，这种算法计算了 Datalog程序中的最小模型（参看[23.5~23.6]）。

3. 半朴质计算

前面讲到，质朴计算中，每一步的计算在下一步里都要重复，这没有必要。半朴质计算就是对此的一大改进（参看[23.28]）。或者说，每一步里，我们仅计算在这一迭代中需要添加的新元组。还用“探寻零件P1”的例子来解释这个思想。伪代码是：

```
NEW  := PS ;
COMP := NEW ;
do until NEW is empty ;
   NEW  := ( NEW * PS ) MINUS COMP ;
   COMP := COMP UNION NEW ;
end ;
DISPLAY := COMP WHERE PX = P# ('P1') ;
```

现在再来看看这一算法。循环开始，对于PS，NEW和COMP是一样的：

NEW

PX	PY
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6

COMP

PX	PY
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6

第一次迭代之后，结果如下所示：

NEW

PX	PY
P1	P4
P1	P5
P2	P5
P3	P6
P4	P6

COMP

PX	PY	
P1	P2	
P1	P3	
P2	P3	
P2	P4	
P3	P5	
P4	P5	
P5	P6	
P1	P4	*
P1	P5	*
P2	P5	*
P3	P6	*
P4	P6	*

COMP与朴质计算中的这一步是一样的，而NEW仅仅是这一迭代中COMP增加的新元组。要特别注意，NEW不包括（P1，P3）（与朴质计算中这一步比较，看有什么不同）。

下一次迭代结束，结果为：

NEW

PX	PY
P1	P6
P2	P6

COMP

PX	PY	
P1	P2	
P1	P3	
P2	P3	
P2	P4	
P3	P5	
P4	P5	
P5	P6	
P1	P4	
P1	P5	
P2	P5	
P3	P6	
P4	P6	
P1	P6	*
P2	P6	*

再下一次迭代，NEW为空，退出循环。

4. 静态筛选

静态筛选（static filtering）是一精细化过程，它来自于经典优化理论中尽可能早地执行选择操作这一基本思想。它是后向推理的一种应用，因为它有效地使用了查询结果信息来修改规则（前提）。同时，它也简化相关事实，因为它利用查询结果信息来删除起初的外延数据库中的无用的元组（参看[23.29]）。用例子的伪码解释这一结果如下：

```
NEW  := PS WHERE PX = P# ('P1') ;
COMP := NEW ;
do until NEW is empty ;
   NEW  := ( NEW * PS ) MINUS COMP ;
   COMP := COMP UNION NEW ;
end ;
DISPLAY := COMP ;
```

再来看看这一算法过程。循环开始，对于PS，NEW和COMP如下所示：

NEW

PX	PY
P1	P2
P1	P3

COMP

PX	PY
P1	P2
P1	P3

第一次迭代之后，结果如下所示：

NEW

PX	PY
P1	P4
P1	P5

COMP

PX	PY	
P1	P2	
P1	P3	
P1	P4	*
P1	P5	*

下一次迭代结束，结果为：

NEW

PX	PY
P1	P6

COMP

PX	PY
P1	P2
P1	P3
P1	P4
P1	P5
P1	P6

*

再下一次迭代，NEW为空，退出循环。

对递归查询策略，我们只作以上简单的介绍。当然，在这一领域内还有许多其它的方法，它们比上面的简单方法要成熟得多；但由于空间有限，我们就不再介绍。

23.8 小结

下面对基于逻辑的数据库作一小结。虽然这一思想在研究领域还受到很大限制，但还是有基于它的商业关系产品上市（尤其对某些优化技术）。总的来讲，基于逻辑的数据库的概念还是令人感兴趣的，前面一节里已经讲了它的潜在的优点。还有一个重要的优点，本章里没有讨论，就是逻辑为通用程序语言和数据库的无缝集成提供了基础。或者说，这一系统提供了唯一的基于逻辑的语言，在这一语言里，“数据就是数据”，而不管它是在共享数据库中还是在本地库中。正是这一语言取代了今天的SQL产品中并不完美的“嵌入数据子语言”的方法（当然，在达到这样的目标之前，还必须克服大量的困难。首先，逻辑是通用程序语言的合适的基础，而目前所取得的进展并不能很好地满足这一需要）。

现在再来简略地回顾一下我们所讲的主要部分。首先，简单地介绍了命题演算和谓词演算，引入了下面的概念：

- 一组合式公式的释义是下面三者的组合：(a)论域；(b)合式公式中单独的常量与此论域中的对应关系；(c)合式公式中定义的谓词和函数的意义。
- 一组合式公式的模型就是一个释义，在此释义中，所有合式公式为真。一般情况下，一组给定的合式公式有许多模型。
- 证明就是这样的一个过程，它表明某一合式公式 *g*（结论）是另一组合式公式 *f1*，*f2*，...，*fn*（前提）的逻辑结果。我们稍微具体地讨论了一个证明方法，即归结与合一。

接着，我们讨论了数据库的证明理论的观点。这一观点认为，数据库是由外延数据库和内涵数据库组成。外延数据库包含基本公理，不严格地讲，就是基本数据；内涵数据库包含完整性约束和演绎公理，不严格地讲就是视图。数据库的“含义”在于由一组从公理演绎来的定理组成；执行一查询就成了定理证明过程（至少概念上是这样的）。演绎DBMS就是这种支持证明理论观点的的DBMS。我们简单地介绍了这种DBMS的一种语言，即Datalog语言。

Datalog语言与传统的关系语言之间最直接最明显的区别在于它支持递归公理，因而支持递归查询。而目前还不知道为什么传统的关系演算和关系代数不能扩展去做同样的功能（参看第6章讨论的TCLOSE操作）[⊖]。我们还讨论了计算这一查询的简单技术。

结论：本章开始时我们讲了许多术语，如：逻辑数据库、推理DBMS、演绎DBMS，等等。这些术语在研究领域经常用到（在一定程度上，甚至在厂家的广告中也会出现）。我们对它们

⊖ 令人感兴趣地注意到关系DBMS需要能解决递归过程，因为在这里有些递归地包含了结构化的信息（根据其它的视图定义的视图就是一个典型的例子）。

作了一些解释。然而，在这些问题上，总是没有一致性。在研究领域，对同一术语可能会出现不同解释。下面的一些解释都是可能会出现的：

- 递归查询过程：这是很简单的概念。它是一查询的计算方法，尤其是优化。它内在地定义了递归（参看23.7节）。
- 知识库：此术语有时就指23.6节讲的内涵数据库，即它由一系列规则（完整性约束和演绎公理）组成。它和库数据相反，库数据是由一系列外延数据库组成。但是有些观点认为它是这两者的组合（下面的演算数据库将讲到），而且还有人认为（参看[23.10]）“知识库经常包括复杂的对象和经典关系”（参看本书第六部分“复杂对象的讨论”）。在自然语言系统里，这一术语还有更多的意义。最好能完全避免这一术语的使用。
- 知识：这也是简单的概念！知识就是知识库里的内容。这一定义简化了前面未能解决的“知识”定义的问题。
- 知识库管理系统（KBMS）：管理知识库的软件。这一术语典型地作为演绎DBMS的同义词使用。
- 演绎DBMS：支持证明理论观点的数据库，尤其是它能根据内涵数据库中的推理（演绎）规则，从外延数据库演绎出其它的信息。演绎DBMS一定能支持递归规则，从而支持递归查询。
- 演绎数据库（有争议的术语）：由演绎DBMS管理的数据库。
- 专家DBMS：演绎数据库的同义词。
- 专家数据库（有争议的术语）：由专家DBMS管理的数据库。
- 推理DBMS：演绎数据库的同义词。
- 基于逻辑的系统：演绎数据库的同义词。
- 逻辑数据库（有争议的术语）：演绎数据库的同义词。
- 逻辑作为数据模型：至少由对象、完整性规则和操作组成的数据模型。在演绎DBMS中，对象、完整性规则和操作都是以统一的方式描述，如在Datalog这一逻辑语言中都描述为公理。正如23.6节所解释的那样，此系统的一个数据库可以看作一种逻辑程序，包含了所有的三种公理。因此，在这样的一个系统中，抽象数据模型就是其逻辑本身。

练习

23.1 试使用归结方法，看下面的中间结果在命题演算中是否构成合法证据。

a. `A => B, C => B, D => ( A OR C ), D |- B`

b.
```
( A => B ) AND ( C => D ), ( B => E AND D => F ),
                              NOT ( E AND F ), A => C |- NOT A
```

c.
```
( A OR B ) => D, D => NOT ( E OR F ), NOT ( B AND C AND E )
                                    |- NOT ( G => NOT ( C AND H ) )
```

23.2 把下面的合式公式转化为子句形式：

a.
```
FORALL x ( FORALL y
           ( p ( x, y ) => EXISTS z ( q ( x, z ) ) ) )
```

b.
```
EXISTS x ( EXISTS y
           ( p ( x, y ) => FORALL z ( q ( x, z ) ) ) )
```

c.
```
EXISTS x ( EXISTS y
           ( p ( x, y ) => EXISTS z ( q ( x, z ) ) ) )
```

23.3 下面是逻辑数据库的相当标准的例子：

```
MAN    ( Adam )
WOMAN  ( Eve )
MAN    ( Cain )
MAN    ( Abel )
MAN    ( Enoch )

PARENT ( Adam, Cain )
PARENT ( Adam, Abel )
PARENT ( Eve, Cain )
PARENT ( Eve, Abel )
PARENT ( Cain, Enoch )

FATHER ( x, y )  <=  PARENT ( x, y ) AND MAN ( x )
MOTHER ( x, y )  <=  PARENT ( x, y ) AND WOMAN ( x )

SIBLING ( x, y )  <=  PARENT ( z, x ) AND PARENT ( z, y )

BROTHER ( x, y )  <=  SIBLING ( x, y ) AND MAN ( x )

SISTER ( x, y )  <=  SIBLING ( x, y ) AND WOMAN ( x )

ANCESTOR ( x, y )  <=  PARENT ( x, y )
ANCESTOR ( x, y )  <=  PARENT ( x, z ) AND ANCESTOR ( z, y )
```

使用归结方法回答下列问题：

a. Cain的母亲是谁？

b. Cain的兄妹是谁？

c. Cain的兄弟是谁？

d. Cain的姐妹是谁？

e. Enoch的祖先是谁？

23.4 解释什么是释义和模型。

23.3 写一组Datalog公理来解释供应商-零件-工程数据库的部分内容。

23.4 试给出习题6.13~6.50的Datalog语句。

23.5 试给出习题8.1的Datalog语句。

23.6 23.7节中用归结与合一的方法实现了“探寻零件 P1”的查询，请给出你自己的解释。

参考文献和简介

基于逻辑的数据库系统在过去几年迅速成长；下面只列出了当前这一研究领域的部分成果，并分组如下：

- 文献[23.1~23.9]是一些书籍，它们讲述了逻辑方面的研究（尤其是在计算和 /或数据库方面）或者收集了基于逻辑的数据库的大量的论文。
- 文献[23.10~23.12]和[23.46]、[23.47]都是一些教程。
- 文献[23.14]、[23.17~23.20]、[23.30]和[23.49~23.50]讲述了传递闭包及其实现。
- 文献[23.21~23.24]描述了一个重要的叫做魔集（及其上变化）的递归查询技术。也可参看[17.24~17.26]。

剩余的文献主要讲述了在这一领域内有多少研究，这里只作了简单的介绍。

23.1 Robert R. Stoll: *Sets, Logic, and Axiomatic Theories.* San Francisco, Calif.: W. H. Freeman and Company (1961).

本书很好地介绍了什么叫逻辑。

23.2 Zohar Manna and Richard Waldinger: *The Logical Basis for Computer Programming- Volume I: Deductive Reasoning* (1985); *Volume II: Deductive Techniques* (1990). Reading, Mass.: Addison-Wesley (1985, 1990).

23.3 Peter M.D. Gray: *Logic, Algebra and Databases*. Chichester, England: Ellis Horwood Ltd. (1984).

本书从数据库的观点比较详细地介绍了命题演算和谓词演算及其它许多相关主题。

23.4 Adrian Walker, Michael McCord, John F. Sowa, and Walter G. Wilson: *Knowledge Systems and Prolog* (2nd edition). Reading, Mass.: Addison-Wesley (1990).

本书从总体上介绍了逻辑程序，而不是基于逻辑的数据库，但本书也介绍了大量的与此相关的方面。

23.5 Hervé Gallaire and Jack Minker: *Logic and Data Bases*. New York, N.Y.: Plenum Publishing Corp. (1978).

本书首先、但不是第一次收集了本领域内的大量的论文。

23.6 Larry Kershberg (ed.): *Expert Database Systems* (Proc. 1 st Int. Workshop on Expert Database Systems, Kiawah Island, S.C.). Menlo Park, Calif.: Benjamin/Cummings (1986).

本书收集了大量优秀的、并发人深省的论文。但是它们中没有哪一篇与本章所讲的直接有关。事实上，这里所讲的在一定程度上澄清了对“专家数据库系统”的混淆。本书内容如下：

1) **知识库的理论；**
2) **逻辑程序和数据库；**
3) **专家数据库系统结构、工具和技术；**
4) **专家数据库系统中的推理；**
5) **智能数据库存取和相互作用。**

还有，John Smith**写的一篇关于专家数据库系统的重要论文及来自以下工作组的报告：**（a）**知识库管理系统；**(b)**逻辑程序和数据库；**（c）**对象数据库系统和知识库系统。正如**Kershberg**在序言里所说的那样，专家数据库系统概念“就是多样的定义和截然不同的结构”。**

23.7 Jack Minker (ed.): *Foundations of Deductive Databases and Logic Programming*. San Mateo, Calif.: Morgan Kaufmann (1988).

23.8 John Mylopoulos and Michael L. Brodie (eds.): *Readings in Artificial Intelligence and Databases*. San Mateo, Calif.: Morgan Kaufmann (1988).

23.9 Jeffrey D. Ullman: *Database and Knowledge-Base Systems* (in two volumes). Rockville, Md.: Computer Science Press (1988, 1989).

该书第一卷共有 10**章，其中有一章详细介绍了基于逻辑的方法。该章还讨论了**Datalog**语言的起源，但重要的是介绍了逻辑与关系代数之间的关系。同时作为逻辑方法的一个特殊的例子，介绍了关系演算（包括域演算和元组演算）。第二卷共有**7**章，其中有**5**章讨论了基于逻辑的数据库。**

23.10 Georges Gardarin and Patrick Valduriez: *Relational Databases and Knowledge Bases*. Reading, Mass.: Addison-Wesley (1989).

该书有一章介绍了演算系统，它讨论了基本的理论及优化算法等。虽然在实际中它只是一本指南，但比我们这一章介绍的要详细。

23.11 Michael Stonebraker: *Introduction to* " Integration of Knowledge and Data Management," in Michael Stonebraker (ed.), *Readings in Database Systems*. San Mateo, Calif.: Morgan Kaufmann (1988).

23.12 Hervé Gallaire, Jack Minker, and Jean-Marie Nicolas: " Logic and Databases: A Deductive Approach," *ACM Comp. Surv*. 16, No. 2 (June 1984).

23.13 Veronica Dahl: " On Database Systems Development through Logic," *ACM TODS 7*, No. 1 (March 1982).

对基于逻辑的数据库的基本思想作了详细的描述，还列举了基于 Prolog原型的一些例子。这些例子是由Dahl在1977年实现的。

23.14 Rakesh Agrawal: " Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries," *IEEE Transactions on Software Engineering 14*, No. 7 (July 1988).

提出了一个新的叫做 alpha的操作，它支持 " 递归查询巨类 " 的问题（实际上，这是一个递归查询的超类）。而它是建立在传统的关系代数的基础上的。在处理大部分包括递归的实际问题时它具有强大的功能，同时它比任何其它的递归机制更容易实现。这篇论文列举了此操作符的几个例子。特别地，它还讲了如何简单地处理传递闭包和 " 总量查询的问题 "（分别参看[23.17]和23.6节）。

文献[23.19]和[23.18]也讨论了一些相关的实现问题。

23.15 Raymond Reiter: " Towards a Logical Reconstruction of Relational Database Theory," in Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt (eds.), *On Conceptual Modelling*: *Perspectives from Artificial Intelligence, Databases, and Programming Languages*. New York, N.Y.: Springer Verlag (1984).

23.2节已经讲过，Reiter著作并不是这一领域内的第一本文献，以前已有许多研究者研究了逻辑和数据库之间的关系（参看文献 [23.5]、[23.7]和[23.13]）。但是，好像是 " Reiter的关系理论的逻辑重构 " 激起了这一领域的发展和当前极大的兴趣。

23.16 François Bancilhon and Raghu Ramakrishnan: " An Amateur's Introduction to Recursive Query Processing Strategies," Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data, Washington, DC (May 1986). Republished in revised form in Michael Stonebraker (ed.), *Readings in Database Systems*. San Mateo, Calif.: Morgan Kaufmann (1988). Also republished in reference [23.8].

这是一篇优秀的综述论文。它开始时介绍了在递归查询的实现中存在的消极和积极的问题。积极的是已经实现了的大量技术，至少它解决了这一查询的实现。消极的是，还根本不知道如何选择在给定情形下最适合的技术（特别地，这里的许多论文很少或根本就没有讨论实现上的特征）。在用一小节讨论了逻辑数据库的基本思想后，继续讨论了大量的命题算法，如朴质计算、半朴质计算、迭代查询 /子查询、递归查询 /子查询、APEX、Prolog语言、Henschen/Naqvi、Aho-Ullman、Kifer-Lozinskii、计数(counting)、魔集等、广义魔集等。它还在应用领域（即这一类算法通常应用的问题）、性能和实现的简易性方面上对这些不同的方法进行了比较。论文还包括了来自一个简单的基准测试的

性能参数(并有相应的比较分析)。

23.17 Yannis E. Ioannidis: “ On the Computation of the Transitive Closure of Relational Operators,” Proc. 12th Int. Conf. on Very Large Data Bases, Kyoto, Japan (August 1986).

在递归查询过程中，传递闭包是非常重要的（参看 [23.18]）。这篇论文提出了一新的方法来实现这一操作，它基于分而治之的方法。参看文献 [23.14]、[23.18~23.20]和[23.49~23.50]。

23.18 H. V. Jagadish, Rakesh Agrawal, and Linda Ness: “ A Study of Transitive Closure as a Recursion Mechanism,” Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. (May 1987).

这里引用摘要“ 此文表明，每一个递归查询都可表示为一个传递闭包，其后可能紧跟在关系代数里也适用的操作 ”。作者认为，在一般情况下，有效地实现传递闭包是有效地实现线性递归的充分基础，所以也是在递归巨类上有效地实现演绎 DBMS的基础。

23.19 Rakesh Agrawal and H. Jagadish: “ Direct Algorithms for Computing the Transitive Closure of Database Relations,” Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, UK (September 1987).

此文指出了一组传递闭包的算法，它“ 没有把这一问题看作是计算递归的问题，而是看作从第一个原理获得闭包的问题 ”（所以这一术语是直接的）。它也对早期的其它的直接算法进行了小结。

23.20 Hongjun Lu: “ New Strategies for Computing the Transitive Closure of a Database Relation,” Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, UK (September 1987).

此文讲述更多的传递闭包的算法。与 [23.19]一样，该文也对早期的解决这一方面问题的方法进行了总结。

23.21 François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman: “ Magic Sets and Other Strange Ways to Implement Logic Programs.” Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (1986).

“ 魔集 ”的基本思想是引入新的规则（“ 魔规则 ”）来保证产生与初始查询同样的结果，但它更有效。从这个意义上讲，它们简化了一系列“ 相关事实 ”（参看23.7节）。具体很复杂，且超出了此注解的范围。至于详细解释，请参考此论文或 Bancilhon或Ramakrishnan的观点，或由Ullman写的[23.9]，或Gardarin和Valdurize 的 [23.10]。基于此思想的许多不同变体相继出台，如下面的文献[23.22~23.24]，同时还可参看[17.24~17.26]。

23.22 Catriel Beeri and Raghu Ramakrishnan: “ On the Power of Magic,” Proc. 6th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (1987).

23.23 Domenico Saccã and Carlo Zaniolo: “ Magic Counting Methods,” Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. (May 1987).

23.24 Georges Gardarin: “ Magic Functions: A Technique to Optimize Extended Datalog Recursive Programs,” Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, UK (September 1987).

23.25 A. Aho and J.D. Ullman: “ Universality of Data Retrieval Languages,” Proc. 6th ACM Symposium on Principles of Programming Languages, San Antonio, Tx. (January 1979).

假如有关系 R、$f(R)$、$f(f(R))$, ...（这里 f 为固定的函数），则根据下面的质朴算法，其

最小不动点定义为关系 R^*（参看23.7节）：

```
R* := R ;
do until R* stops growing ;
   R* := R* UNION f(R*) ;
end ;
```

此文指出了另一个关系代数的最小不动点操作。

23.26 Jeffrey D. Ullman: “ Implementation of Logical Query Languages for Databases,” *ACM TODS 10*, No. 3 (September 1985).

此文讲述了一个重要的类，它包括可能的递归查询实现技术。这些技术依据“规则/目标树”的“捕获”规则而定义。它们也是依据子句和谓词而表达查询策略图。这篇论文定义了几个这样的规则。一个是讲关系代数操作的应用，两个分别讲前向链和后向链，还有一个“小路”规则，它允许结果从一个子目标传向另一个子目标。传向后者的“小路”信息是所谓的魔集技术的基础（参看 [23.21~23.24]）。

23.27 Shalom Tsur and Carlo Zaniolo: “ LDL: A Logic-Based Data- Language,” Proc. 12th Int. Conf. on Very Large Data Bases, Kyoto, Japan (August 1986).

LDL包括：（1）一个“集”类型构造器；（2）非（基于差）；（3）数据定义；（4）更新操作。它是一种纯的逻辑语言，语句间没有序的依赖关系。这种语言是编译型的，而不是解释型的。Naqvi和Tsur也写了一本书讨论了这一问题（参看 [23.45]）。

23.28 Fran¢ois Bancilhon: “ Naive Evaluation of Recursively Defined Relations,” in Michael Brodie and John Mylopoulos (eds.), *On Knowledge Base Management Systems: Integrating Database and AI Systems.* New York, N.Y.: Springer Verlag (1986).

23.29 Eliezer L. Lozinskii: “ A Problem-Oriented Inferential Database System,” *ACM TODS 11*, No. 3 (September 1986).

此文讲述了“相关事实”的来源。由于推理技术导致这一研究领域其它方面的快速扩充，这篇文章就讲述了一原型系统去抑制这种扩充。

23.30 Arnon Rosenthal *et al.*: “ Traversal Recursion: A Practical Approach to Supporting Recursive Applications, ” Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data, Washington, DC (June 1986).

23.31 Georges Gardarin and Christophe de Maindreville: “ Evaluation of Database Recursive Logic Programs as Recurrent Function Series, ” Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data, Washington, DC (June 1986).

23.32 Louiqa Raschid and Stanley Y. W. Su: “ A Parallel Processing Strategy for Evaluating Recursive Queries, ” Proc. 12th Int. Conf. on Very Large Data Bases, Kyoto, Japan (August 1986).

23.33 Nicolas Spyratos: “ The Partition Model: A Deductive Database Model,” *ACM TODS 12*, No. 1 (March 1987).

23.34 Jiawei Han and Lawrence J. Henschen: “ Handling Redundancy in the Processing of Recursive Queries,” Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. (May 1987).

23.35 Weining Zhang, and C. T. Yu: “ A Necessary Condition for a Doubly Recursive Rule to be Equivalent to a Linear Recursive Rule,” Proc. 1987 ACM SIGMOD Int. Conf. on Management

of Data, San Francisco, Calif. (May 1987).

23.36 Wolfgang Neidl: " Recursive Strategies for Answering Recursive Queries-The RQA/FQI Strategy, "Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, UK (September 1987).

23.37 Kyu-Young Whang and Shamkant B. Navathe: "An Extended Disjunctive Normal Form Approach for Optimizing Recursive Logic Queries in Loosely Coupled Environments," Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, UK (September 1987).

23.38 Jeffrey F. Naughton: "Compiling Separable Recursions," Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, I11. (June 1988).

23.39 Cheong Youn, Lawrence J. Henschen, and Jiawei Han: " Classification of Recursive Formulas in Deductive Databases," Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, I11. (June 1988).

23.40 S. Ceri, G. Gottlob, and L. Lavazza: " Translation and Optimization of Logic Queries: The Algebraic Approach," Proc. 12th Int. Conf. on Very Large Data Bases, Kyoto, Japan (August 1986).

23.41 S. Ceri and L. Tanca: "Optimization of Systems of Algebraic Equations for Evaluating Datalog Queries," Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, UK (September 1987).

23.42 Allen Van Gelder: "A Message Passing Framework for Logical Query Evaluation," Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data, Washington, DC (June 1986).

23.43 Ouri Wolfson and Avi Silberschatz: " Distributed Processing of Logic Programs," Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, I11. (June 1988).

23.44 Jeffrey F. Naughton *et al*.: " Efficient Evaluation of Right-, Left-, and Multi-Linear Rules," Proc. 1989 ACM SIGMOD Int. Conf. on Management of Data, Portland, Ore. (June 1989).

23.45 Shamim Naqvi and Shalom Tsur: *A Logical Language for Data and Knowledge Bases*. New York, N.Y.: Computer Science Press (1989).

本文深入地论述了语言LDL**（参看**[23.27]**）**.

23.46 S. Ceri, G. Gottlob, and L. Tanca: *Logic Programming and Databases.* New York, N.Y.: Springer Verlag (1990).

23.47 Subrata Kumar Das: *Deductive Databases and Logic Programming*. Reading, Mass.: AddisonWesley (1992).

23.48 Michael Kifer and Eliezer Lozinskii: " On Compile Time Query Optimization in Deductive Databases by Means of Static Filtering," *ACM TODS 15*, No. 3 (September 1990).

23.49 Rakesh Agrawal, Shaul Dar, and H. V. Jagadish: " Direct Transitive Closure Algorithms: Design and Performance Evaluation," *ACM TODS 15*, No. 3 (September 1990).

23.50 H. V. Jagadish: " A Compression Method to Materialize Transitive Closure," *ACM TODS 15*, No. 4 (December 1990).

此文提出了一索引技术，从而可使给定关系的传递闭包压缩存储。这样，要测试一个元组是否在闭包里出现，就可以通过一单表查询辅以索引比较。

23.51 Serge Abiteboul and Stéphane Grumbach: "A Rule-Based Language with Functions and Sets,"*A CM TODS 16*, No. 1 (March 199 1).

此文描述了COL语言（“Complex Object Language”），它是对Dtatlog语言的扩展，集成了演绎和对象数据库的思想。

部分练习答案

23.1 a. 合法，b. 合法，c. 不合法。

23.2 在下面中，*a*, *b*, *c*是斯科林常量，*f*是有两个变元的斯科林函数。

a. `p ( x, y ) ⇒ q ( x, f ( x, y ) )`

b. `p ( a, b ) ⇒ q ( a, z )`

c. `p ( a, b ) ⇒ q ( a, c )`

23.6 下面我们用23.6.*n*的形式给习题解答编号，其中6.*n*是第6章中习题解答的编号。

23.6.13
```
? ⇐ J ( j, jn, jc )
```

23.6.14
```
? ⇐ J ( j, jn, London )
```

23.6.15
```
RES ( s ) ⇐ SPJ ( s, p, J1 )
? ⇐ RES ( s )
```

23.6.16
```
? ⇐  SPJ ( s, p, j, q ) AND 300 ≤ q AND q ≤ 750
```

23.6.17
```
RES ( p1, pc ) ⇐ P ( p, pn, p1, w, pc )
? ⇐ RES ( p1, pc )
```

23.6.18
```
RES ( s, p, j ) ⇐ S ( s, sn, st, c ) AND
                  P ( p, pn, p1, w, c ) AND
                  J ( j, jn, c )
? ⇐ RES ( s, p, j )
```

23.6.19~23.6.20 只有利用否定才可做。

23.6.21
```
RES ( p ) ⇐ SPJ ( s, p, j, q ) AND
            S ( s, sn, st, London )
? ⇐ RES ( p )
```

23.6.22
```
RES ( p ) ⇐ SPJ ( s, p, j, q ) AND
            S ( s, sn, st, London ) AND
            J ( j, jn, London )
? ⇐ RES ( p )
```

23.6.23
```
RES ( c1, c2 ) ⇐ SPJ ( s, p, j, q ) AND
                 S ( s, sn, st, c1 ) AND
                 J ( j, jn, c2 )
? ⇐ RES ( c1, c2 )
```

23.6.24
```
RES ( p ) ⇐ SPJ ( s, p, j, q ) AND
            S ( s, sn, st, c ) AND
            J ( j, jn, c )
? ⇐ RES ( p )
```

23.6.25 只有利用否定才可做。

23.6.26
```
RES ( p1, p2 ) ⇐ SPJ ( s, p1, j1, q1 ) AND
                 SPJ ( s, p2, j2, q2 )
? ⇐ RES ( p1, p2 )
```

23.6.27~23.6.30 只有利用分组和聚集才可做。

23.6.31
```
RES ( jn ) ⇐ J ( j, jn, jc ) AND
             SPJ ( S1, p, j, q )
? ⇐ RES ( jn )
```

23.6.32
```
RES ( p1 ) ⇐ P ( p, pn, p1, w, pc ) AND
             SPJ ( S1, p, j, q )
? ⇐ RES ( p1 )
```

23.6.33
```
RES ( p ) ⇐ P ( p, pn, pl, w, pc ) AND
            SPJ ( s, p, j, q ) AND
            J ( j, jn, London )
? ⇐ RES ( p )
```

23.6.34
```
RES ( j ) ⇐ SPJ ( s, p, j, q ) AND
            SPJ ( S1, p, j2, q2 )
? ⇐ RES ( j )
```

23.6.35
```
RES ( s ) ⇐ SPJ ( s, p, j, q ) AND
            SPJ ( s2, p, j2, q2 ) AND
            SPJ ( s2, p2, j3, q3 ) AND
            P ( p2, pn, Red, w, c )
? ⇐ RES ( s )
```

23.6.36
```
RES ( s ) ⇐ S ( s, sn, st, c ) AND
            S ( S1, sn1, st1, c1 ) AND st < st1
? ⇐ RES ( s )
```

23.3.37~23.6.39 只有利用分组和聚集才可做。

23.6.40~23.6.44 只有利用否定才可做。

23.6.45
```
RES ( c ) ⇐ S ( s, sn, st, c )
RES ( c ) ⇐ P ( p, pn, pl, w, c )
RES ( c ) ⇐ J ( j, jn, c )
? ⇐ RES ( c )
```

23.6.46
```
RES ( p ) ⇐ SPJ ( s, p, j, q ) AND
            S ( s, sn, st, London )
RES ( p ) ⇐ SPJ ( s, p, j, q ) AND
            J ( j, jn, London )
? ⇐ RES ( p )
```

23.6.47~23.6.48 只有利用否定才可做。

23.6.49~23.6.50 只有利用分组才可做。

23.7 我们把约束看作传统的蕴含，而不是“回溯”Datalog类型。

a.
```
CITY ( London )
CITY ( Paris )
CITY ( Rome )
CITY ( Athens )
CITY ( Oslo )
CITY ( Stockholm )
CITY ( Madrid )
CITY ( Amsterdam )

S ( s, sn, st, c )      ⇒ CITY ( c )
P ( p, pn, pc, pw, c ) ⇒ CITY ( c )
J ( j, jn, c )          ⇒ CITY ( c )
```

b. 只有利用适合的标量操作才可做。

c.
```
P ( p, pn, Red, pw, pc ) ⇒ pw < 50
```

d. 只有利用非和聚集操作才可做。

e.
```
S ( s1, sn1, st1, Athens ) AND
S ( s2, sn2, st2, Athens ) ⇒ s1 = s2
```

f. 只有利用分组和聚集才可做。

g. 只有利用分组和聚集才可做。

h.
```
J ( j, jn, c ) ⇒ S ( s, sn, st, c )
```

i.
```
J ( j, jn, c ) ⇒ SPJ ( s, p, j, q ) AND S ( s, sn, st, c )
```

j.
```
P ( p1, pn1, pl1, pw1, pc1 ) ⇒ P ( p2, pn2, Red, pw2, pc2 )
```

k. 只有利用聚集操作才可做。

l.
```
S ( s, sn, st, London ) ⇒ SP ( s, P2, q )
```

m.
```
P ( p1, pn1, pl1, pw1, pc1 ) ==>
  P ( p2, pn2, Red, pw2, pc2 ) AND pw2 < 50
```

n. **只有利用聚集操作才可做。**

o. **只有利用聚集操作才可做。**

p. **不能做，这是一种传递约束。**

q. **不能做，这是一种传递约束。**