第3章　关系数据库介绍

3.1　引言

在第1章中已经介绍，本书的重点主要是在关系系统。特别是第二部分比较深入地阐述了这些系统（关系模型）的理论基础。为了更好地理解本书随后部分的内容，本章对第二部分的内容（附带随后的部分）做一个直观、非正式的介绍。本章所述的多数论题将在后面章节做更详细、更正式的讨论。

3.2　关系模型概述

如前所述，关系系统基于正规的关系基础或理论，即关系数据模型。直观地说，关系系统是这样的系统：

1) 结构化方面：数据库中的数据对用户来说是表，并且只是表；
2) 完整性方面：数据库中的这些表满足一定的完整性约束（在本节最后讨论）；
3) 操纵性方面：用户可以使用用于表操作的操作符 —— 例如，为了检索数据，需要使用从一个表导出另一个表的操作符。其中，选择、投影和连接这三种尤为重要。

图3-1中显示的是一个简单的关系数据库，即部门和雇员数据库。正像你所看到的，数据库给人的感觉就是一张张的表（这些表的含义是不言而喻的）。

DEPT

DEPT#	DNAME	BUDGET
D1	Marketing	10M
D2	Development	12M
D3	Research	5M

EMP

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K
E4	Saito	D2	35K

图3-1　部门和雇员数据库（样本值）

图3-2中显示了对图3-1中数据库的选择、投影和连接操作。下面给出这些操作的定义：

- 选择操作是从表中提取特定的几行。
- 投影操作是从表中提取特定的几列。
- 连接操作是根据某一列的值将两个表连接起来。

以上三个例子中，最后一个关于连接的例子需要进一步解释。先观察到 DEPT和EMP两个表都有一个共同的列 DEPT#，因此它们可以根据这一列的相同值连接起来。当且仅当两个表中对应行的DEPT#值相同时，DEPT的一行才能连接EMP表中对应的一行（产生结果表的一行）。

Restrict: *Result:*

DEPTs where BUDGET > 8M

DEPT#	DNAME	BUDGET
D1	Marketing	10M
D2	Development	12M

Project: *Result:*

DEPTs over DEPT#, BUDGET

DEPT#	BUDGET
D1	10M
D2	12M
D3	5M

Join:

DEPTs and EMPs over DEPT#

Result:

DEPT#	DNAME	BUDGET	EMP#	ENAME	SALARY
D1	Marketing	10M	E1	Lopez	40K
D1	Marketing	10M	E2	Cheng	42K
D2	Development	12M	E3	Finzi	30K
D2	Development	12M	E4	Saito	35K

图3-2 选择、投影和连接（例子）

例如，DEPT和EMP行

DEPT#	DNAME	BUDGET
D1	Marketing	10M

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	40K

连接起来产生如下的结果行：

DEPT#	DNAME	BUDGET	EMP#	ENAME	SALARY
D1	Marketing	10M	E1	Lopez	40K

因为在公共列它们有相同的值D1。注意在结果行中相同的值只出现一次。连接的结果包含了这样所有可能得到的行。尤其注意尽管在DEPT表中有D3一行，但因为EMP行中没有DEPT#的值D3，结果中就没有出现D3行。

现在，图3-2中清楚地显示出三种操作的每个结果都是一个表（如前所述，实际上是从一个表导出另一个表的操作）。这是关系系统的闭包特性，这一特性非常重要。基本上，因为任何操作的输出和其输入的对象种类相同——它们都是表——所以一个操作的输出能变成另一个的输入。因此，可以采取如连接的投影、两个选择后的连接或一个投影的选择等等操作。即我们可以编写嵌套的表达式来处理数据——操作数本身也是由一个表达式来表示的，而不仅仅是一个表名。这样随之而来又有许多重要结论，在后面会见到（在本章和后面的许多章）。

顺便提一下，当提到一个操作的输出是另一个表时，要知道这是从概念视图的角度来说的，这一点非常重要。这并不是意味着系统实际上必须将每个单独操作的结果实例化。例如，假设要计算一个连接的选择，那么，连接后的行一形成，系统就立即检查该行是否满足指定的条件以判定是否属于最终结果，如果不是就立即抛弃。也就是说，连接操作的中间结果根本不会以完整的实例表形式存在。在实际操作中，通常为了提高效率，系统尽可能不将中间结果生成完整的表。注意：如果中间结果全部实例化，整个表达式的计算策略就是实例化的计算；如果中间结果分块地提供给下一步操作，就称作流水线计算。

图3-2中还清楚地说明了一点：操作是一次一集合，而不是一次一行；也就是说，操作数和

结果是完整的表，而不只是单行，是包含行集的表（当然，只包含一行的表是合法的；空表，即根本不包含任何行的表，也是合法的）。例如，图3-2中，分别对两个表对应的3行和4行的连接操作，就返回一个4行的结果表。相应地，非关系系统中典型的操作是一次一行或一次一记录；因此，集合处理能力是关系系统区别于其他系统的一个重要特征（见下面3.5节进一步的讨论）。

再回到图3-1。结合图中的样本数据库，还可以得出以下两点：

- 首先，关系系统要求只让用户所感觉的数据库是一张张表。在关系系统中，表是逻辑结构而不是物理结构。实际上，系统在物理层可以使用所喜欢的方式来存储数据——使用有序文件，索引，哈希，指针链，压缩，等等——只要能将存储表示映射到逻辑模式的表。换句话说，表是对物理存储数据的一种抽象表示——对许多存储细节的抽象，如存储记录的位置，存储记录顺序，存储数据值的表示，存储记录的前缀，存储的访问结构如索引等等，对用户来说都是不可见的。

  此外，在ANSI/SPRARC中，前述的逻辑结构一词意味着包含概念模式和外模式。关键是——如第2章中所述——概念模式和外模式都是关系的，而物理模式和内模式不是。关系理论与内模式毫无关系，它只是考虑数据库怎样呈现给用户[⊖]。关系系统唯一的要求是无论选择什么物理结构，一定要全部实现其逻辑结构。

- 其次，关系数据库遵守一条非常好的原则，即信息原则。数据库全部的信息内容有一种表示方式而且只有一种，也就是表中的行列位置有明确的值。这种表示是关系系统中唯一可行的方式（当然，在逻辑层）。特别地，没有连接一个表到另一个表的指针。在图3-1中，例如，表DEPT中的D1行和表EMP的E1行有联系，因为雇员E1在部门D1工作；但是这种联系不是通过指针来表示的，而是通过表EMP的E1行中DEPT#列位置的值为D1来联系的。相反地，在非关系系统中，这些信息典型地由指针来表示，这种指针对用户来说是可见的。

注意：当指出关系数据库中没有指针时，并不是指在物理层没有指针——正好相反，关系数据库在物理层使用指针，但在关系系统中所有物理存储的细节对用户来说都是不可见的。

对关系模型的结构和操纵方面就提这些；现在来看完整性方面。观察图3-1 中的部门和雇员数据库。事实上，可以要求数据库满足任何条件的完整性约束——如，雇员工资必须在25K~95K的范围，部门预算在1M~15M，等等。某些约束在应用上非常重要，它们喜欢用特定的术语。如：

1) DEPT表中每一行的DEPT#必须是唯一的；同时，EMP表中每一行EMP#的值必须唯一。DEPT表中的DEPT#列和EMP表中的EMP#列都是它们各自表的主码（回忆第1章中，在图中主码是用双下划线标出来的）。

2) EMP表中的每个DEPT#的值必须在DEPT表中有相同的DEPT#值，以反映每个雇员必须安排在现有的部门中。即EMP表中的DEPT#列是外码，参照了DEPT表的主码。

现在定义关系模型，以便于后面参照（尽管该定义十分抽象，且此时非常难理解）。简而言之，关系模型包括下列五部分：

1) 一个可扩展的标量类型的集合（尤其包括布尔型或真值型）；

2) 关系类型生成器和对应这些关系类型的解释器；

⊖ 遗憾的是，目前大多数的SQL产品不能恰当地支持这方面的理论。更准确地说，只支持相当有限的概念模式/内模式之间的映象（典型地，将一个逻辑表直接映射到一个存储文件）。结果，几乎不能提供关系技术理论上所能达到的数据物理独立性。

3) 实用程序，用于定义生成关系类型的关系变量；

4) 向关系变量赋关系值的关系赋值操作；

5) 从其他关系值中产生关系值的、可扩充的关系操作符集合。

关系模型要比“表加上选择、投影和连接”多得多，尽管关系模型常常以此为特征。

注意：你可能会惊讶于没有对完整性约束进行明确的定义。事实上这些约束表示的只是关系操作符的一个应用；即这些约束是由操作符来表达的，见第 8章。

3.3 关系和关系变量

如果关系数据库只是一个用表来表示数据的数据库 —— 当然这是真的 —— 那么问题是：为什么称这样的数据库是关系的？答案很简单（这在第 1章已经提到）：“关系”只是表的数学术语 —— 精确地说，是某特殊种类的表（细节见第 5章）。因此我们可以说，图3-1 中的部门和雇员数据库包含了两个关系。

在非正式的情况下，经常将“关系”和“表”作为同义词，并且“表”一词比“关系”一词更常用。但是花点时间去理解为什么首先介绍“关系”这个词是值得的。简单地说，如下原因：

- 关系系统基于关系模型，反过来关系模型又是基于数学方面的数据抽象理论（主要是集合论和谓词逻辑）。
- 关系模型理论是E.F.Codd在1969年到1970年提出的，当时他是IBM的研究员。1968年末，数学家Codd首先意识到数学可以对数据管理领域注入坚强的理论和严密的逻辑，在那之前，数据管理领域太缺乏这些理论的支持。 Codd的想法首先在一篇经典的论文中提出，即“大型共享数据库数据的关系模型”(参见第5章中文献[5.1])。
- 从那时起，那些思想 —— 至今几乎全世界都接受了 —— 就在数据库技术的各个方面产生了广泛的影响。同样在其他领域，如人工智能领域、自然语言处理和硬件系统的设计领域也产生了广泛的影响。

目前，最初由Codd建立的关系模型特意采用了这些术语，如关系这个词本身，当时在信息技术(IT)界还不是很流行（尽管有时候较常见），问题是许多常用的术语很模糊——它们缺乏精确性，而精确性对Codd所提出的形式理论是很必要的。例如，“记录”这个词，不同情况下，“记录”可以表示记录值或记录型；逻辑记录或物理记录，存储记录或虚记录，还可以有其他的表示。因此，关系模型不使用记录这一词，而用“元组”来进行精确的定义。这里不给出这个定义，目前可以说“元组”这个词近似地对应一行的概念（就像“关系”这个词近似地对应一个表的概念）。当在第二部分进一步讨论关系系统的方面，将利用这些正式术语，但是本章只进行非正式的介绍。将继续使用像“行”或“列”这些相当常见的词，不过以后还会使用“关系”这个词。

再回到图3-1中的部门和雇员数据库来看另外一个重要的观点。实际上，数据库中 DEPT和EMP是关系变量，即它们的值是关系值（不同的时候是不同的值）。例如，假定EMP中的值如图3-1所示，并假定删除Saito这一行（雇员号为E4）：

```
DELETE EMP WHERE EMP# = 'E4'
```

结果如图3-3所示。

从概念上说，以上就是EMP旧的关系值被一个新的关系值所代替。当然旧值（ 4行）和新

EMP

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K

图3-3 删除E4行后的关系变量

值（3行）很相似，但是从概念上说，这两者是不同的值。确实，删除操作可以简化为一个关系赋值操作，如下：

```
EMP :=EMP MINUS (EMP WHERE EMP# = 'E4')
```

（注意：原来的删除和相应的赋值语句都是用一种称为 Tutorial D[3.3]的语言来表示，MINUS是Tutorial D的关键字，表示关系的差操作，见第 6章）。与所有的赋值操作一样，这就是（a）计算表达式的右边；然后（b）计算结果赋给左边的变量（当然，根据定义，左边必须是特定的变量）。如上所述，最后结果就是用一个“新”的 EMP代替“旧”的EMP。

当然，通过类似的方式，关系 INSERT和UPDATE操作也可以简化为特定的关系赋值操作。

目前，遗憾的是，许多文献用“关系”这个词时，表达的意思却是关系变量的含义（在表示关系时，其含义为关系值）。虽然这是历史造成的，但这必然导致了概念上的混淆。因此，贯穿本书，我们将从本质上仔细区分“关系”和“关系变量”这两个词；根据参考文献 [3.3]，我们将使用“ relvar”作为relational variables的简写，而且还要注意，当实际含义是关系变量时，用relvar而不用关系来表示。

注意：提醒大家， relvar一词并不常用 —— 但应该常用！尽管本书的早期版本并未如此，且在这方面目前数据库的文献也未能给予重视，我们深切感到清楚地区分关系本身（即关系值）和关系变量是非常重要的。

3.4 关系的含义

第1章中提到关系中的列与数据类型有关 ㊀。在3.2节末尾，指出关系模型包括“可扩充的[数据]类型集”。这意味着用户可以定义自己的数据类型（当然，这和使用系统定义类型或固有的类型一样）。例如，可以定义如下类型（ Tutorial D语法；省略号“……”代表与目前所讨论的问题并不密切相关的定义部分）：

```
TYPE EMP#...;
TYPE NAME...;
TYPE DEPT#...;
TYPE MONEY...;
```

例如，类型EMP#作为所有可能的雇员号的集合；类型 NAME作为所有可能的名字的集合，等等。

图3-4基本上是图3-1的EMP部分扩展了列的数据类型。如图所示，每个关系——更确切地说，每个关系值 —— 都有两部分，一组列名：类型名对（列标题），以及符合该标题的行集（主体）。注意：实际上，就像前面所举的例子那样，经常省略列标题的类型名部分，但是，要知道在概念上它们是存在的。

㊀ 数据类型更常用的关系术语是域，见第 5章。

EMP

EMP# : EMP#	ENAME : NAME	DEPT# : DEPT#	SALARY : MONEY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K
E4	Saito	D2	35K

图3-4 带有列类型的EMP关系值示例

对“关系”还有一种重要的认识方式。如下所述：

- 首先，对于指定的关系*r*，关系*r*的标题表示了一个特定的谓词或真值函数。
- 其次，如第1章简要说明，*r*主体中的每一行都表示一个真命题，用相应类型的参数的值代替该谓词的占位符或参数型来得到这个谓词（实例化为谓词）。

在图3-4中，例如，谓词如下：

雇员EMP#的名字是ENAME，工作在部门DEPT#，所得工资为SALARY

（参数EMP#、ENAME、DEPT#和SALARY对应于EMP表的四列）。相应的真命题是：

雇员E1的名字是Lopez，工作在部门D1，所得工资为40K

（通过用EMP#值E1，NAME值Lopez，DEPT#值D1和MONEY值40K替换相应的参数来得到）；

雇员E2的名字是Cheng，工作在部门D1，所得工资为42K

（通过用EMP#值E2，NAME值Cheng，DEPT#值D1和MONEY值42K替换相应的参数来得到）；等等，因此：

- 类型是所讨论的事物（的集合）；
- 关系是关于所讨论的事物的事物（的集合）。

（这里有一个类比可以帮助大家来理解和记住这些重要的结论：类型对于关系就像名词对于句子一样。）在上述例子中，我们所讨论的事物是雇员号、名字、部门号和货币值，以及事务，这些是真正的说话形式，即“指定雇员号的雇员有指定的名字，工作在指定的部门，得到指定的工资”。

从前面可以得到：

- 首先，类型和关系都是必要的（没有类型，就没什么可讨论的；没有关系，也没什么可讨论的）。
- 其次，类型和关系不仅必需的，而且是充分的——即从逻辑上说，我们不再需要其他的东西。

注意：类型和关系不是一回事。遗憾的是，某些商业产品——根据定义，不是关系的——常常将这一点混淆。在第25章会提到这个问题（25.2节）。

顺便提及，要知道每个关系都有一个相联系的谓词，包括那些通过如连接关系操作得出的关系，这点也是比较重要的。例如，图3-1中的关系DEPT和图3-2 中的三个结果关系有如下几个谓词：

- DEPT：“部门DEPT#的名字为DNAME预算为BUDGET”
- 选择BUDGET>8M的DEPT：“部门DEPT#的名字为DNAME，预算为BUDGET，预算大于八百万美元”。
- 对DEPT的DEPT#和BUDGET投影：“部门DEPT#有某名字，且预算为BUDGET”。

- DEPT和EMP根据DEPT#的连接："部门DEPT#的名字为DNAME，预算为BUDGET，而且雇员EMP#名为ENAME，在部门DEPT#工作，所得工资为SALARY"。注意，这一谓词有6个参数，不是7个——两个DEPT#是同一个参数。

3.5 优化

如3.2节所述，关系操作如选择、投影和连接都是集合操作。因而，关系语言通常称为非过程化语言，因为用户指出是什么而不是怎么做——即他们只说想要什么，而不说出如何得到。为了满足用户需求而对存储数据的导航过程是由系统自动进行而不用用户手工处理。因此，有时称关系系统实现自动导航。相反，在非关系系统中，导航一般由用户负责。对自动导航一系列优点的说明如图3-5所示，该说明对比了某SQL INSERT语句和用户在非关系系统（实际上是一个CODASYL网状系统；例子来自参考文献[1.5]的关于网状数据库的一章）中要实现相应的操作所编写的"手工导航"代码。注意：这里的数据库是供应商和零件数据库，参见3.9节的进一步阐述。

```
INSERT INTO SP ( S#, P#, QTY )
       VALUES ( 'S4', 'P3', 1000 ) ;
```

```
MOVE 'S4' TO S# IN S
FIND CALC S
ACCEPT S-SP-ADDR FROM S-SP CURRENCY
FIND LAST SP WITHIN S-SP
while SP found PERFORM
   ACCEPT S-SP-ADDR FROM S-SP CURRENCY
   FIND OWNER WITHIN P-SP
   GET P
   IF P# IN P < 'P3'
      leave loop
   END-IF
   FIND PRIOR SP WITHIN S-SP
END-PERFORM
MOVE 'P3' TO P# IN P
FIND CALC P
ACCEPT P-SP-ADDR FROM P-SP CURRENCY
FIND LAST SP WITHIN P-SP
while SP found PERFORM
   ACCEPT P-SP-ADDR FROM P-SP CURRENCY
   FIND OWNER WITHIN S-SP
   GET S
   IF S# IN S < 'S4'
      leave loop
   END-IF
   FIND PRIOR SP WITHIN P-SP
END-PERFORM
MOVE 1000 TO QTY IN SP
FIND DB-KEY IS S-SP-ADDR
FIND DB-KEY IS P-SP-ADDR
STORE SP
CONNECT SP TO S-SP
CONNECT SP TO P-SP
```

图3-5 自动与手工导航

尽管前一段已经提到，这里还要强调一下，非过程化并不是一个让人非常满意的术语，因为过程化和非过程化并不是绝对的。最好的提法是某种语言A比另一种语言B多些或少些过程化。或许更好的表述方式为：与像SQL一类的语言相比像C++、COBOL（或在非关系的数据库管理系统中的数据子语言，如图3-5所示）等语言，是在更高层次上的抽象。本质上，它是一种抽象层次的提升，这种抽象提高了关系系统的生产力。

如何实现上面提到的自动导航的功能是DBMS的一个重要组成部分——优化器（在第2章中简要介绍过）的职责。换句话说，对于用户的每个关系请求，优化器的工作是选择一个更高效的方式执行这一请求。通过一个例子，我们假定用户发出如下请求（采用Tutorial D）：

```
RESULT :=(EMP WHERE EMP#='E4'){SALARY};
```

说明：圆括号(“EMP WHERE ...”)中的式子表示选择关系变量EMP中满足EMP#行为E4的当前值。在大括号(“SALARY”)中的列名对选择的结果根据SALARY列进行投影。最后，赋值操作(“:=”)将投影的结果赋值给关系变量RESULT。赋值后，RESULT包括雇员E4的工资的单行、单列的关系（注意，例子中显然利用了关系闭包的性质——在右边写了一个嵌套的表达式，其中选择的输出作为投影的输入）。

目前，即使在很简单的例子中，也可能至少有两种执行数据访问的方式：

1) 通过对关系变量EMP依物理顺序扫描，直到找到所要的数据；

2) 如果在EMP#列有索引——EMP#的值假定是唯一的，因为为了确保唯一性，许多系统实际上需要索引——那么，通过使用索引直接找到所要的数据。

优化器将选择采取其中一种策略。更一般地，对于任何一个给定的请求，优化器根据如下考虑来选择执行请求的策略：

- 请求中参照了哪些关系变量；
- 关系变量的数据规模；
- 存在什么索引；
- 索引的选择性如何；
- 在磁盘上数据是怎样物理聚集的；
- 涉及什么样的关系操作；

等等。因此，用户只需说明他们想要什么，而不用管怎样得到数据；获得这些数据的访问策略是优化器(“自动导航”)来选择的。用户和用户应用程序独立于这些访问策略，如果要得到物理数据的独立性，这一点显然是很重要的。

在第17章中，将详细阐述优化器的功能。

3.6 数据字典

第2章中已经提到，每个DBMS必须提供目录或字典功能。字典存储了各种模式（外模式、概念模式和内模式）和相应的映象（外模式/概念模式的映象，概念模式/内模式的映象）。换句话说，字典包括了对系统自身有用的各种对象的细节信息(有时称作描述信息或元数据)。这些对象例如关系变量、索引、用户、完整性约束、安全性约束，等等。描述信息确保系统正确工作。例如，优化器利用索引和其他物理存储结构的字典信息，以及其他信息来帮助决定怎样实现用户的请求。同样地，安全子系统首先利用用户和安全性约束的字典信息来准许或拒绝这些请求。

关系系统的优点之一是，在系统中字典本身就是由关系变量（更精确地说应为系统关系变量，这样称呼是区别于普通用户的关系变量）组成的。因此，用户能够像访问自己的数据一样访问字典。例如，典型的字典包括TABLE和COLUMN两种系统关系变量，分别描述了数据库中的表（即关系变量）和表中的列（我们说“典型地”是因为每个系统的字典都不同；这些差别是由于一个特定系统的字典必须包含许多系统特定的信息）。对于图3-1中的部门和雇员数据库，TABLE和COLUMN关系变量如图3-6中所示[⊖]。

注意：在第2章中提到，字典通常应该是自描述的，即它包括描述字典关系变量自身的条

⊖ 注意：表中ROWCOUNT列的存在表明对数据库的INSERT和DELETE操作会同时引起数据字典的更新。

目。但在图3-6中没有这样的条目。见本章末尾的练习3.3。

TABLE

TABNAME	COLCOUNT	ROWCOUNT	.....
DEPT	3	3	.....
EMP	4	4	.....
.....	.....	.....	.....

COLUMN

TABNAME	COLNAME	.....
DEPT	DEPT#	.....
DEPT	DNAME	.....
DEPT	BUDGET	.....
EMP	EMP#	.....
EMP	ENAME	.....
EMP	DEPT#	.....
EMP	SALARY	.....
.....	.....	.....

图3-6 部门和雇员数据库的字典

假定部门和雇员数据库的一些用户想要确切地知道关系变量DEPT包含哪些列（假定由于某种原因，用户不知道该信息）。那么用表达式

```
(COLUMN WHERE TABNAME ='DEPT'){'COLNAME'}
```

可以实现。注：如果要将查询结果永久保存起来，就像前一节中的例子，可以将表达式的值赋予另一个关系变量。大多数的例子都省略了最后赋值这一步。

下面是另一个例子："哪一个关系变量包含了名为EMP#的列？"

```
(COLUMN WHERE COLNAME ='EMP#'){'TABNAME'}
```

练习：下面语句的执行结果是什么？

```
((TABLE JOIN COLUMN
        WHERE COLCOUNT ) 5){TABNAME, COLNAME}
```

3.7 基本关系变量和视图

从一组关系变量如DEPT和EMP，以及一组关系变量的关系值开始，从指定的关系值可以通过关系表达式来得到其他的关系值——例如，通过将两个指定的关系值连接起来。我们再介绍一些术语，源（指定的）关系变量称为基本关系变量，而它们的关系值称为基本关系；通过某关系表达式从基本关系中得出的、或能够得出的关系称为导出关系或可导出关系。注：在参考文献[3.3]中，基本关系又称为实关系变量。

首先，关系系统必须提供创建基本关系的方法。例如，在SQL中，这一任务是由CREATE TABLE语句来执行的（很明确，这里的"TABLE"是基本关系变量）。显然，基本关系变量必须是已命名的——例如：

```
CREATE TABLE EMP...;.
```

但是，关系系统通常也支持另一种命名的关系变量——视图，在任何指定的时候，它的值都是导出的关系（因此，视图也可以当作导出的关系变量）。在指定时刻，指定视图的值是当时关系表达式的计算结果；视图创建时的关系表达式是确定的。例如，语句

```
CREATE VIEW TOPEMP AS
      (EMP WHERE SALARY >3)3{ EMP#, ENAME, SALARY};
```

可用于定义名为TOPEMP的视图。注：为了方便，例子用SQL和 Tutorial D的混合形式来表示。

当语句执行时，AS之后的关系表达式——视图定义表达式——没有计算，只是系统以某种方式保存下来（实际上，把它保存在字典中，在名字 TOPEMP之下）。但是，对用户来说，好像在数据库中真正存在名为 TOPEMP的关系变量，而且在图3-7中的非阴影部分是它的当前值。用户能够像操纵基本关系一样操纵视图。注：如果（如前所述） DEPT 和EMP被当作实关系变量，则TOPEMP就被当作虚关系变量——即关系变量看起来以自身的形式存在，但实际上并不是。（在任何特定时候，它的值依赖于其他特定的关系变量）。

TOPEMP

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K
E4	Saito	D2	35K

图3-7 EMP的视图TOPEMP（非阴影部分）

注意，尽管说TOPEMP的值是关系，该关系是计算视图定义表达式而得到的结果，但是它并不表示现在有了一份独立的数据拷贝；也就是说不表示真的计算了视图定义表达式。相反，视图只是它所依赖的基本关系变量 EMP的一个窗口。结果，任何基本关系变量的改变都会通过窗口自动地看到（当然，假定在非阴影部分）。同样，TOPEMP的改变也会自动地同时反映到关系变量EMP上（见后面的例子）。

以下是一个对视图TOPEMP的检索操作的例子：

```
(TOPEMP WHERE SALARY< 42K) { EMP#, SALARY}
```

对图3-7中的样本数据，结果如下：

EMP#	SALARY
E1	40K
E4	35K

从概念上讲，对视图的操作（如前面提到的检索操作）是通过替换为视图所对应的视图定义表达式（保存在目录中的表达式）来处理的。因此，例子中的原表达式

```
(TOPEMP WHERE SALARY< 42K) { EMP#, SALARY}
```

被系统替换为

```
((EMP WHERE SALARY > 33K) { EMP#, ENAME, SALARY})
          WHEERE SALARY < 42K{ EMP#, SALARY}
```

(把原表达式中的视图名用斜体表示，改变的代替部分也用斜体表示)，替换后的表达式可以简化为

```
(EMP WHERE SALARY > 33K AND SALARY < 42K) { EMP#, SALARY}
```

(见第17章)，对此计算后就得到前面的结果。换句话说，对视图的操作其实是转变为对基本关系变量的等价操作，等价操作以正常的方式执行（更确切地说，以正常的方式优化并执行）。

再举另一个例子，考虑如下的DELETE操作：

```
DELETE TOPEMP WHERE SALARY < 42K
```

实际执行的DELETE是如下的：

```
DELETE  EMP WHERE SALARY > 33K AND SALARY< 42K
```

当前视图TOPEMP非常简单，只包含了一个基本关系变量的一个行列子集。但是，理论上，视图定义可以是任意复杂的（甚至可以参照其他视图），因为实质上它只是一个命名的关系表达式。例如，下面的视图涉及了两个基本关系变量的连接：

```
CREATE VIEW JOINEX AS
   ((EMP JION DEPT)  WHERE BUDGE >)M{EMP#,DEP#};
```

我们将在第9章详细讨论视图定义和处理的问题。

顺便在这里解释一下第2章2.2节末尾的评论，大意是“视图”在关系范畴中有一个特定的含义，和在ANSI/SPARC体系结构中所述的含义不尽相同。在该体系的外模式，通过外部视图观察数据库，由外模式来定义外部视图（不同的用户有不同的外部视图）。相反，在关系系统中，视图是命名的、导出的虚关系变量。 ANSI/SPARC的外部视图类似于几个关系变量的集合，其中每个都是关系意义下的一个视图，“外模式”包含那些视图的定义（可以看出，在关系意义下的视图是一种提供逻辑数据独立性的关系模型的方式，尽管目前的商业产品在这方面还有欠缺。见第9章）。

目前，ANSI/SPARC体系结构已经很普通了，并且允许外模式和概念模式之间的任意可变性。理论上，甚至两层所支持的数据结构的类型都可能是不同的——例如，概念模式可能是关系的，而用户可以有一个层次的外部视图。但是，实际上，许多系统使用同一种数据结构类型作为两层的基础，关系产品也不例外——就像基本关系变量一样，视图仍然是关系变量。既然两层支持同样的对象类型，两层也就支持同样的数据子语言（通常为SQL）。视图是关系变量，这一事实是关系系统的扩展之一；这就像数学中子集是集合一样重要。注意： SQL产品和SQL标准（见第4章）好像对用词有一点误解，比如在SQL中经常说“表和视图”（暗示视图不是表）。建议大家不要掉入这一陷阱，不要把“表”（或关系变量）只理解为基本表（或关系变量）。

最后一点需要指出的是基本关系变量和视图的问题。两者的不同在于：

- 基本关系变量是“真实存在”，意味着它们所表示的数据真正存在于数据库中。
- 相反，视图并不“真实存在”，只是提供了观察“真实数据”的各种方式。

尽管这在非正式情况下可能很有用，但是，这一特征并不能准确反映事情的真正本质。确实，用户可以把基本关系变量理解为物理存储的关系变量；但事实上，在某种程度上，关系系统的本意是允许用户把基本关系变量认为是物理上存在的，而不用考虑在物理上是怎样表示的。但这不能解释为：基本关系变量是以任意一种直接的方式（如，作为一个存储文件）[⊖] 进行存储的。如在3.2节提及的，基本关系变量最好被当作是存储数据集的抽象——即已经掩盖了存储细节的抽象。理论上，基本关系变量和它相应的存储之间可以存在任何程度的差别。

用一个简单的例子来澄清这一点。再看部门和雇员数据库，目前大多数关系系统可能用两个文件实现数据库，每个文件都用于一个基本关系变量。但是，对为什么不用一个存储记录文件，这其中绝对没有逻辑上的原因。每个存储记录包含（ a）每个部门有一个部门号、名

⊖ 下面的引文来自一本书，该书给出了这里提到的几种混淆，以及前面在 3.3 节所讨论的其他问题：“对于存储关系(即表)和表现关系(即视图)的区分非常重要……应该只有在表和视图都适用的时候才使用关系一词。当要强调关系是实际存储的而不是视图时，就应该使用基关系或基本表这些术语”。遗憾的是，这些混淆的情况非常典型。

称和预算；和（b）部门中的每个雇员有雇员号、名称和工资。换句话说，只要数据是正确的，就可以在物理上以任何方式存储，但在逻辑层上看起来总是一样的。

3.8 事务

注意：本节的题目并不只针对关系系统，为了理解第二部分有关的内容必须先理解一些基本概念。但是，这里只是粗略地介绍。

第1章中指出，事务是一个逻辑工作单元，通常包括几个数据库操作，并指出当不同的操作处于同一事务中时，用户要通知系统。BEGIN TRANSACTION、COMMIT 和ROLLBACK操作就是基于这一考虑提出的。基本上，当 BEGIN TRANSACTION执行时，表示一个事务的开始，当COMMIT 或ROLLBACK执行时，表示一个事务的终止。例如（伪码）：

```
BEGIN TRANSACTION ; /* move $$$ from account A to account B */
UPDATE account A ;                          /* withdrawal   */
UPDATE account B ;                          /* deposit      */
IF everything worked fine
   THEN COMMIT ;                            /* normal end   */
   ELSE ROLLBACK ;                          /* abnormal end */
END IF ;
```

注意以下几点问题：

1) 要保证事务的原子性，也就是说，即使系统在处理中发生故障，也要保证（从逻辑的观点）事务中的操作要么都做，要么都不做。
2) 要保证事务的持续性，一旦事务成功地执行了 COMMIT，即使随后系统发生故障，也要确保它的更新写入数据库中。注意：本质上，是事务的持续性保证了数据库中数据的持久性。
3) 要保证事务的隔离性，事务T1对数据库的更新操作对任何不同的事务 T2来说是不可见的，直到或除非T1成功执行COMMIT。COMMIT使事务的更新对其他事务来说是可见的；这些更新已经提交，并且要保证不能取消。若事务执行了 ROLLBACK，所有事务的更新操作都要取消（回滚）。对后一种情况，其结果应该就像事务一开始就没有执行一样。
4) 要保证一组并发事务的交叉执行（通常）是可串行的，即其结果与按某一未指明的次序串行地执行时的结果相同。

对上述几点和许多其他方面的深入讨论参见第 14章和第15章。

3.9 供应商和零件数据库

贯穿本书，我们使用的例子是著名的供应商和零件数据库（和第 1章一样，基于一个虚构的KnowWare公司）。本节主要是解释这个数据库，为后续章节引用提供方便。图 3-8给出了一个样本数据值的集合；当后面做改动时，后面所举的例子实际上基于这些值。图 3-9给出了数据库定义，是用 Tutorial D 来表示的。尤其注意主码和外码的说明。注意：（a）几个列都有与所提到的列同名的数据类型；（ b）STATUS列和两个CITY 列都是用内部类型定义的 —— INTEGER和CHAR （变长字符串） —— 而不是用户定义的。最后要注意的很重要的一点是关于图3-8中所示的列值，在此，还不能说明这一点，留待第 5章5.2节“ 可能的表示 ”来加以阐述。

S

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

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

P

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

图3-8 供应商和零件数据库（样本值）

```
TYPE S# ... ;
TYPE NAME ... ;
TYPE P# ... ;
TYPE COLOR ... ;
TYPE WEIGHT ... ;
TYPE QTY ... ;

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

VAR P BASE RELATION
    { P#      P#,
      PNAME   NAME,
      COLOR   COLOR,
      WEIGHT  WEIGHT,
      CITY    CHAR }
    PRIMARY KEY { P# } ;

VAR SP BASE RELATION
     { S#      S#,
       P#      P#,
       QTY     QTY }
     PRIMARY KEY { S#, P# }
     FOREIGN KEY { S# } REFERENCES S
     FOREIGN KEY { P# } REFERENCES P ;
```

图3-9 供应商和零件数据库（数据定义）

以上数据库的含义如下：

- 关系变量S代表供应商，每个供应商有一个供应商号（S#），对供应商来说这是唯一的；一个供应商名（SNAME），不必唯一（尽管SNAME值在图3-8中是唯一的）；定额或状态值（STATUS）；和一个场所（CITY）。假定一个供应商只住在一个城市。
- 关系变量P代表零件（更精确地说，零件的种类），每种零件有一个零件号（P#），是唯一的；零件名（PNAME）；颜色（COLOR）；重量（WEIGHT）；该种零件储存的地点（CITY）。假定——在不大重要的情况下——零件的重量单位是磅。假定每种零件只有一种颜色，恰好只存在一个城市的仓库中。
- 关系变量SP代表发货。逻辑上，它的含义是将另外两个关系变量连接起来。例如，图3-8中SP的首行连接了表S（供应商S1）的一个特定的供应商和表P（零件P1）的一种特定的零件——换句话说，表示供应商S1供应了P1这种零件（供应数量为300）。这样，每个发货就有一个供应商号（S#）、零件号（P#）和数量（QTY）。假定在任何指定的时刻，对特定的供应商和特定的零件至多只有一次供应。因此，对于每次供应，对于目前在SP

中出现的S#值和 P#值的组合是唯一的。注意：图 3-8中的样本值特意包括了一个供应商S5，没有供应与之对应。

注意到（第1章1.3节已经指出）供应商和零件可以当作实体，发货可以当作一个指定的供应商和指定的零件之间的联系。该节还指出，联系最好当作实体的一个特例。关系数据库的优点是，无论实际上这些实体是否为联系，所有实体都以同样的形式表示 —— 即，例子中显示的关系中的行。

最后注意以下几点：

- 首先，尽管供应商和零件数据库非常简单，可能比任何实际数据库都简单得多；绝大多数实际的数据库都包含比这个例子多很多的实体和联系（多很多种实体和联系），但不管怎样，该例至少可以恰当地说明本书中后面所要得出的结论。在下面几章，大多数还将使用它作为例子。
- 其次，可以使用像SUPPLIERS、PARTS和SHIPMENT这样描述性的名字代替上面使用的P、SP这些简写的名字；实际中推荐使用描述性的名字。但对于供应商和零件这一特定的例子，因为要频繁使用，所以使用了简名。多次重复长名字会令人厌烦。

3.10 小结

简要回顾一下关系技术。显然，对这样一个含义广泛的概念，目前只涉及了一些表面的内容，本章的主旨是为进一步理解做一些简要介绍。尽管如此，我们还是设法尽可能包括更多方面的内容。下面将谈到的主要内容小结如下。

从用户的观点来看，关系数据库是关系变量或表的集合。关系系统是支持关系数据库及其操作的系统，特别是选择、投影和连接操作。这些操作和其他操作都是集合操作。关系系统的封闭性是指每种操作的输出都和其输入是相同类型的（都是关系的），这就表示可以写嵌套的关系表达式。关系变量可以通过关系赋值操作来更新；类似地，INSERT、UPDATE和DELETE更新操作也可以看作某些一般关系赋值操作的缩写。

关系系统所依据的形式理论是关系数据模型。关系模型只涉及逻辑的内容，而不涉及物理的内容，它包括数据的三方面理论 —— 即数据结构、数据完整性和数据操作。结构化方面与关系本身有关；完整性方面与主码和外码有关；操作方面与操作符（选择、投影和连接等等）有关。信息原则表明是以一种而且只以一种方式表示关系数据库的整个内容，即以关系中的行和列交叉位置的明确的值表示。

每个关系都含有一个标题和一个主体；标题是（列名：类型名）对的集合，主体是对应标题的行集。一个指定关系的标题可以当作谓词；主体中的每一行表示一个真命题，用参数的值代替占位符或谓词参数的适当类型来得到这个真命题。换句话说，类型是我们所研究的对象（的集合）；关系是关于我们所研究的对象的内容（的集合）。类型和关系对于我们所要表示的数据（逻辑上的）来说是充分而必要的。

优化器是系统的组成部分，它决定怎样执行用户请求（用户只需考虑要什么，而不必考虑怎么做）。因为关系系统能够负责导航定位所要的数据库中的数据，所以有时被称为自动导航系统。优化和自动导航是保持数据物理独立性的先决条件。

字典是一组系统关系变量，它包括了关于对数据库有用的各种条目的细节信息（基本关系变量、视图、索引和用户，等等）。用户能够像访问自己的数据一样访问字典。

在指定的数据库中，源关系变量称为基本关系变量，而它们的值称为基本关系；通过关系表达式从基本关系中得出的关系称为导出关系（基本关系和导出关系被称为可表现的关系）。视图是一种关系变量，它的值在任何指定的时刻是一个导出的关系（粗略地讲，可以认为视图是导出的关系变量）；在指定的时刻，这一关系变量的值是从相应的视图定义表达式计算得到的。因此，基本关系变量是独立存在的，但视图不是——它们依赖于相应的基本关系变量（另一种说法是基本关系变量是自治的，而视图不是）。用户能够像操纵基本关系一样操纵视图（至少理论上是）。系统是通过替换视图所对应的视图定义表达式来执行视图上的操作的。因此，对视图的操作就转变为对基本关系变量的等价操作。

事务是一个逻辑工作单元，通常包括几个数据库操作。当BEGIN TRANSACTION执行时，一个事务开始；当COMMIT（正常终止）或ROLLBACK（非正常终止）执行时，事务终止。事务是原子的、持续的且与其他事务相隔离。一组并发事务的交叉执行（通常）要保证是可串行的。

最后，本书中基本的例子是供应商和零件数据库。如果大家还没有熟悉这个例子，现在得多花点时间来熟悉它；至少应该知道哪个关系变量有哪些列，以及主码和外码是什么（不必确切地知道样本数据值）。

练习

3.1 定义下列术语

自动导航	主码
基本关系变量	投影
字典	命题
闭包	关系数据库
提交	关系数据库管理系统
导出的关系变量	关系模型
外码	选择
连接	回滚
优化	集合操作
谓词	视图

3.2 描述供应商和零件数据库的字典关系变量TABLE和COLUMN的内容。

3.3 在3.6节中提到，字典是自描述的——也就是说，包括字典关系变量自身的条目。扩展图3-6，以包括关系变量TABLE和COLUMN的条目。

3.4 这是一个对供应商和零件数据库的查询。它是做什么的？

```
RESULT:=((S JOIN SP) WHERE P#='P2',{STATUS};
```

注意：实际上这个查询有点小问题，与P#列的数据类型有关。在第5章的5.2节（“类型转换”）将回到这一问题上来。类似的注意也适用于下一个练习。

3.5 假定习题3.4中赋值右边的表达式使用视图定义：

```
CREATE VIEW V AS
(S JOIN SP) WHERE P#='P2'),{STATUS};
```

考虑查询

```
RESULT:=( V WHERE CITY ='London'){ S#};
```

该查询做了什么？给出DBMS处理这一查询的过程。

3.6 你如何理解（事务）的原子性、持续性、隔离性和可串行性？

参考文献和简介

3.1 E.F.Codd: “Relational Database : A Practical Foundation for Production，” *CACM 25*，No.2 (February 1982).Republished in Robert L.Ashenhurst (ed.)，*ACM Turing Award Lectures*: *The First Twenty Years 1966-1985*.Reading，Mass.: Addison-Wesley (*ACM Press Anthology Series*，1987) .

这篇论文使Codd获得1981年ACM 图灵奖，该奖项是对他在关系模型方面的工作给予的奖励。该文指出了著名的应用程序黑箱问题。释义：“计算机应用程序的需求发展迅速——如此迅速以致于信息系统部门（负责提供这些应用程序）的能力越来越落后于需求。”对这一问题有两种弥补办法：

1) 给IT专业人员提供新工具以提高其生产力；

2) 允许最终用户直接与数据库交互，这样完全越过了 IT专业人员。

以上两种方法都需要，本文中Codd给出证明，关系理论能向两者提供必要的基础。

3.2 C.J.Date :“ Why Relational?” ，in *Relational Database Writings 1985-1989*. Reading，Mass.:Addison-Wesley(1990).

试图对关系系统的主要优点做简洁而又易懂的概括。本文的下列观点值得重复：在“关系化（going relational）”的各种优点中，有一点是再强调也不过份的，即一个合理理论基础的存在。引文：“

……关系的（relational）确实与众不同。不同之处是因为它并不特殊（ ad hoc)。相反，旧的系统是特殊的；它们可以解决当时特定的重要问题，但是它们没有坚实的理论基础。相反，关系的确基于这样一个基础……这意味着它是坚实的。”

“感谢这一坚实的基础，关系系统以良好定义的方式工作；并且（也许没有意识到这一事实）用户在头脑中有一个那种状态的简单模型，这使他们有信心预见在特定情况下系统的处理结果。（应该）没有什么惊讶的。这种预见能力意味着用户界面易于理解、存档、教学、学习、使用和记忆。”

3.3 C.J.Date and Hugh Darwen: *Foundation for Object/Relational Databases*: *The Third Manifesto*. Reading，Mass.: Addison-Wesley(1998).See also the introductory overview paper “ *The Third Manifesto* : Foundation for Object/Relational Databases，” in C.J.Date，Hugh Darwen，and David McGoveran，*Relational Database Writings 1994-1997*. Reading，Mass .: Addison-Wesley(1998).

The Third Manifesto（第三次宣言）是对数据库和 DBMS未来方向的一个详尽的、形式化的和严密的论述。它可以被看作 DBMS和其语言界面设计的一个抽象蓝图。它基于传统的核心概念——型、值、变量和操作符。例如，有一个 INTEGER类型；整数“ 3 ”是该类型的一个值；N是该类型的一个变量，其值在特定的时刻是某整数值；“ + ”是应用于整数值（即该类型的值）的操作符。

部分练习答案

3.3 图3-10显示了关系变量TABLE 和COLUMN的条目（用户自己的关系变量的条目省略了）。显然不可能给出COLCOUNT和ROWCOUNT的精确值。

TABLE

TABNAME	COLCOUNT	ROWCOUNT	.....
TABLE	(>3)	(>2)	.....
COLUMN	(>2)	(>5)	.....
.....	.....	.....	.....

COLUMN

TABNAME	COLNAME	.....
TABLE	TABNAME	.....
TABLE	COLCOUNT	.....
TABLE	ROWCOUNT	.....
COLUMN	TABNAME	.....
COLUMN	COLNAME	.....
.....	.....	.....

图3-10 TABLE 和COLUMN自身的字典条目（大致轮廓）

3.4 该查询检索供应P2零件的供应商的供应商号和所在城市。

3.5 查询的意思是：“找出供应零件P2的London供应商的供应商号”。查询处理的第一步是用定义V的表达式来代替名字V：

```
(((  S  JOIN  SP  WHERE  P#='P2'){S#, CITY})
                    WHERE  CITY='London'){S#}
```

简化为：

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

进一步的解释见第9章和第17章。