第4章 SQL 概 述

4.1 引言

正如在第1章中所说的，SQL是处理关系数据库的标准语言，并且市场上的任何数据库产品都支持SQL。SQL是20世纪70年代早期在IBM公司的研究所开发的（参见[4.8~4.9]和[4.28]），其大部分标准首先在IBM的System R中实现（参见[4.1~4.3]和[4.11~4.13]），随后又在IBM公司的一些其他商品和其他公司的一些商品中实现（参见[4.20]）。在本章中，将主要介绍SQL语言；另外的部分，如完整性、安全性等将在后续章节中介绍。现在所有的讨论是建立在SQL/92（也叫*SQL-92*，或者是*SQL2*，参见[4.22~2.23]）的标准之上的，其正式的名称是国际标准数据库语言(International Standard Database Language) SQL (1992)。

注意：值得说明的一点是，SQL3标准的制定已经接近尾声，它是现在的SQL标准的升级，预计在1999年末会得到通过。因此，在本书付印之际，SQL的标准有可能已经是“SQL/99”，而不是“SQL/92”了。但是，作者认为不应该将SQL3作为讲解的基础，因为现在一个非常明显的事实是没有产品支持SQL3。所以本书将SQL3单独放在附录中（见附录B）中讲解。目前还没有一个数据库产品完全支持SQL/92[⊖]；相反，数据库产品支持的是SQL/92的“子集的超集”，即：大部分的产品没有完全支持SQL/92，但却在SQL的一些其他方面进行了扩展。例如：IBM公司的DB2并不完全支持SQL/92的标准，但在视图的更新规则上却支持得比SQL/92还要多。

一些附加的基本注意事项：

- SQL原先是作为特殊的“数据子语言”出现的。然而，随着持久存储模块（PSM）在1996年成为了标准，SQL已经变成了计算上完全（computationally complete）的语言。除了几个关系变量和特殊的处理之外，它现在包括很多语句，如：CALL、RETURN、SET、CASE、IF、LOOP、LEAVE、WHILE、REPEAT等。因此，在开发应用中就没有必要再将SQL与一些有区别的“宿主”语言捆绑在一起了。然而，在此不对PSM进行详细的讨论。
- SQL使用表来表示关系和关系变量（见第3章）。因此为了SQL标准和SQL产品的一致性，在本章中也做同样的处理，对本书其他涉及到SQL的部分，都做此处理。而且，SQL不在表中或关系中使用“头”和“体”的概念。
- SQL是一个庞大的语言。它的标准文档[4.22]就有600多页，而SQL3是它的两倍大。因此，在这本书中不可能对其进行过多的讨论；现在能做的只是以一种相对来说较好理解的方式介绍其主要部分，但是读者必须注意到这些介绍是非常浅显的。另外，本书虽然忽略了与主要内容无关的材料，但是却没有因为叙述简洁的要求而将其简单化。读者可以在参考资料[4.4]、[4.19]和[4.27]中找到更为完整的描述。

⊖ 实际上，没有任何的产品完全支持SQL/92，因为现在发布的SQL/92本身就有许多错误和矛盾之处。这方面的具体讨论可参见[4.19]。

- 最后说明一下，因为 SQL比较冗长，而且是委托的，因此它还不是非常完美的关系语言。然而，它是一种标准，市场上的每一种产品几乎都支持它，而且数据库的专业人员也需要了解它。而这就是本书所涵盖的内容。

4.2 综述

SQL兼有数据定义和数据操纵的功能。首先考虑定义操作。图 4-1为供应商和零件数据库给出了 SQL的定义，读者可以与第 3章的图 3-9对照来看。三类基表（可参见第 3章，CREATE TABLE 语句中的关键字 TABLE指的是一个特殊的基表）在数据定义语言中都有 CREATE TABLE语句。CREATE TABLE语句可定义出每个基表的名字，该表中的列名，各列的数据类型，该表的主码，以及该表的任何外码。如果还有其他的对该表的描述信息，也可以进行定义，详见图 4-1。下面说明惯用的几个写法：

- 举例时，经常在列名中使用“ #”，但是该字符在 SQL/92中是不合法的。
- 使用分号“ ；”作为语句的结束符。但是 SQL/92中是不是使用该符号作为结束符要依情况而定。具体的细节问题超出了本书的范围。

```
CREATE TABLE S
     ( S#      CHAR(5),
       SNAME   CHAR(20),
       STATUS  NUMERIC(5),
       CITY    CHAR(15),
     PRIMARY KEY ( S# ) ) ;

CREATE TABLE P
     ( P#      CHAR(6),
       PNAME   CHAR(20),
       COLOR   CHAR(6),
       WEIGHT  NUMERIC(5,1),
       CITY    CHAR(15),
     PRIMARY KEY ( P# ) ) ;

CREATE TABLE SP
     ( S#      CHAR(5),
       P#      CHAR(6),
       QTY     NUMERIC(9),
     PRIMARY KEY ( S#, P# ),
     FOREIGN KEY ( S# ) REFERENCES S,
     FOREIGN KEY ( P# ) REFERENCES P ;
```

图4-1 供应商和零件商数据库（SQL定义）

与第3章中定义非常类似的图 3-9相比，图 4-1最大的不同之处在于：没有包括图 3-9中的类型定义，例如， TYPE语句。其原因在于 SQL不允许用户定义自己的数据类型；[⊖] 这样，列就只能按照系统给出的类型（即系统定义的类型）进行定义。 SQL只支持下列的几种不需说明的系统给定的数据类型：

```
CHARACTER [ VARYING ] (n)      INTEGER        DATE
BIT [ VARYING ] (n)            SMALLINT       TIME
NUMERIC (p,q)                  FLOAT (p)      TIMESTAMP
DECIMAL (p,q)                                 INTERVAL
```

SQL还支持缺省值、缩写词和一些替换的拼写，如： CHARACTER可以简写为 CHAR，此处省略一些具体的细节。另外，在一些字母和数字上加注方括号“ [”和“]”，表示括号中的内容是可选的。最后， SQL还需要对一定的数据类型（如 CHAR）声明其长度和精度，而在第3章的假设语法中是没有的。实际上， SQL是将长度和精度作为数据类型的一部分

⊖ SQL确实允许定义“域”，但是这个“域”的概念不是真正的关系意义（见第 5章）上的域，如，类型。注意：SQL3中支持用户定义的类型（详见附录 B）。

来看的，这就表示 CHAR（3）和CHAR（4）不是一种数据类型。本书中认为将其看成是完整性约束会更好一些，具体可参见第 8章的练习8.4。

定义了数据库之后，就可以通过 SQL的数据操纵运算对数据进行操纵，数据操纵运算包括SELECT、INSERT、UPDATE和DELETE。使用SELECT语句可以完成关系的选择、投影和连接操作。图 4-2示例了如何用 SQL实现选择、投影和连接运算。注意：连接操作的例子中用到了有限定的列名（如 S.S#，SP.S#），使用有限定的列名的目的是为了避免引用列的歧义。在 SQL中可以随时使用有限定的列名，但是当使用列名不会造成歧义时，也可以使用无限定的列名。

如下例所示， SQL还支持SELECT子句的简记方式。

```
SELECT *    -- or "SELECT S.*" (i.e., the "*" can be qualified)
FROM   S ;
```

结果就是 S表的整个拷贝；星号表示列出 FROM子句中参考的表的所有列，并且按照在表中定义的从左往右的顺序列出。注意这个例子中的注释 (用两个连字符开始，用换行符结束)。注意： SELECT * FROM *T*中的*T*是一个表的名字，在 SQL句子中是 TABLE *T*的简写。

第7章（7.7节）中将对 SELECT语句进行更详尽的介绍。

Restrict:

```
SELECT S#, P#, QTY
FROM   SP
WHERE  QTY < 150 ;
```

Result:

S#	P#	QTY
S1	P5	100
S1	P6	100

Project:

```
SELECT S#, CITY
FROM   S ;
```

Result:

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

Join:

```
SELECT S.S#, SNAME, STATUS, CITY, P#, QTY
FROM   S, SP
WHERE  S.S# = SP.S# ;
```

Result:

S#	SNAME	STATUS	CITY	P#	QTY
S1	Smith	20	London	P1	300
S1	Smith	20	London	P2	200
S1	Smith	20	London	P3	400
..	.....	..	.....	..	...
S4	Clark	20	London	P5	400

图4-2 SQL中的选择、投影和连接运算的例子

现在介绍更新操作：第 1章已经给出了 SQL中的INSERT、UPDATE和DELETE语句的例子，但是该例却仅仅是对单行的操作。然而，一般来说， INSERT、UPDATE及DELETE和SELECT语句一样，都是集合操作运算符，这在第 1章的练习和答案中也可以看出。下面给出几个在供应商和零件商数据库上进行更新操作的例子。

```
INSERT
INTO   TEMP ( P#, WEIGHT )
       SELECT P#, WEIGHT
       FROM   P
       WHERE  COLOR = 'Red' ;
```

这个例子假定已经创建了一个名为 TEMP的表，该表有两列： P#和WEIGHT。INSERT语句在零件表中插入所有的红色零件的标识号以及与标识号相对应的零件的重量。

```
UPDATE S
SET    STATUS = STATUS * 2
WHERE  CITY = 'Paris' ;
```

这个UPDATE语句将所有的在 Paris的供应商的状态值（ STATUS）都加倍。

```
DELETE
FROM   SP
WHERE  P# = 'p2';
```

这个DELETE语句删除了零件号为 P2的所有的记录。

注意：SQL没有直接的关系赋值的操作。然而，可以通过下面的方法实现该操作：首先将目标表中的所有的行删除，然后执行一个 INSERT... SELECT...（如上面的第一个例子）将数据插入表中。

4.3 目录

SQL标准还包括一个被称为信息模式（Information Schema）的标准目录的详细说明。实际上，惯用的两个名词“目录”和“模式”都在 SQL中使用，但是各具有 SQL的特定含义。笼统地说，一个 SQL的目录包括的是对某一单个数据库的描述，⊖ 而一个SQL模式包括的则是某一用户数据库的某一部分的描述。换句话说，目录可以有很多（每个数据库一个），每个目录可以由很多模式组成。然而每一个目录必须要包括一个叫做信息模式的模式，而从用户的观点来看，正是该模式起到了目录的作用。

信息模式由一系列的 SQL表组成，这些表的内容以一种非常精确的方式定义，因此可以非常有效地显示该目录中其他模式的定义。更精确地说，信息模式包含一些假定的“定义模式”。实现中不需要支持所有的“定义模式”，但是必须得（ a）支持某些类型的“定义模式”；（ b）并且支持跟信息模式类似的“定义模式”的视图。下面说明几点注意事项：

1) 之所以提出如上所说的 (a)和(b)两个概念，是基于以下的考虑。首先，现在的很多产品支持跟“定义模式”类似的内容。然而，那些“定义模式”在各种产品中的定义有很大的差别，即使是同类的、但是属于不同厂家的产品，其差别也很大。因此，就需要有对这些“定义模式”的视图的支持。
2) 因为在每一个目录中都有一个信息模式，所以提到信息模式时，不应该理解成是特定的某个。因此，一般来说，对一个用户有用的所有数据不可能由一个信息模式来描述。然而，为了简单起见，仍假定只有一个信息模式。

这里没有必要详细讨论信息模式的内容。下面仅列出一些重要的信息模式中的视图，读者可以很容易地依据这些模式的名字看出该模式中包含的内容。然而，需要说明的是，TABLES视图包含了所有的视图和基本表的信息，而 VIEWS视图则只包含了视图的信息。

```
SCHEMATA                    REFERENTIAL_CONSTRAINTS
DOMAINS                     CHECK_CONSTRAINTS
TABLES                      KEY_COLUMN_USAGE
VIEWS                       ASSERTIONS
COLUMNS                     VIEW_TABLE_USAGE
TABLE_PRIVILEGES            VIEW_COLUMN_USAGE
COLUMN_PRIVILEGES           CONSTRAINT_TABLE_USAGE
USAGE_PRIVILEGES            CONSTRAINT_COLUMN_USAGE
DOMAIN_CONSTRAINTS          CONSTRAINT_DOMAIN_USAGE
TABLE_CONSTRAINTS
```

⊖ 为了精确起见，还要注意在 SQL 标准中没有“数据库”的概念！准确的说，由目录描述的数据的集合是在执行时生成的。然而，将其作为数据库来理解就不是很合理。

参见[4.19]可以得到更多关于信息模式的细节描述。特别地，该参考资料还介绍了如何在信息模式的基础上显示查询。当然，查询过程没有读者想像的那么直接。

4.4 视图

首先给出一个定义视图的例子：

```
CREATE VIEW GOOD_SUPPLIER
    AS SELECT S#, STATUS, CITY
       FROM   S
       WHERE  STATUS > 15 ;
```

以下给出了一个在该视图上定义的查询：

```
SELECT S#, STATUS
FROM   GOOD_SUPPLIER
WHERE  CITY = 'London' ;
```

用视图的定义代替视图的名字，可以写成如下的表示（注意 FROM子句中的嵌套子查询）：

```
SELECT GOOD_SUPPLIER.S#, GOOD_SUPPLIER.STATUS
FROM ( SELECT S#, STATUS, CITY
       FROM   S
       WHERE  STATUS > 15 ) AS GOOD_SUPPLIER
WHERE  GOOD_SUPPLIER.CITY = 'London' ;
```

上面的表示还可以简化成如下的形式：

```
SELECT S#, STATUS
FROM   S
WHERE  STATUS > 15
AND    CITY = 'London' ;
```

后一个查询是真正要执行的查询。

考虑另外一个关于 DELETE操作的例子：

```
DELETE
FROM   GOOD_SUPPLIER
WHERE  CITY = 'London' ;
```

这个DELETE语句的实际执行是这样的：

```
DELETE
FROM   S
WHERE  STATUS > 15
AND    CITY = 'London' ;
```

4.5 事务

SQL中的COMMIT WORK和ROLLBACK WORK跟通常所说的 COMMIT和ROLLBACK语句是类似的，在两种情况下，关键字 WORK都是可选的。然而， SQL标准中却没有明显的BEGIN TRANSACTION语句。相反，当一个程序执行了“事务初始化”语句，并且没用执行中的事务，就隐含着开始了一个新事务。具体什么 SQL语句是事务初始化语句，在此处暂不做介绍；可以把本章中的所有语句都看作具有事务初始化功能的语句（当然COMMIT和ROLLBACK除外）。

4.6 嵌入式SQL

大多数的 SQL语句允许直接（例如直接在联机终端上交互运行）或嵌入应用程序中（如，SQL语句可以嵌入程序语言中，并跟它们一起使用）执行。而且，在嵌入的情况下，

应用程序可以使用多种宿主语言，如 COBOL、Java、PL/I等[⊖] 。本节主要介绍嵌入式 SQL的情况。

嵌入式SQL的基本思想就是任何可以交互使用的 SQL语句都可以在一个应用程序中使用，这叫做双重模式原理。当然，SQL语句的交互形式和嵌入形式之间有很大的区别，尤其是检索操作需要在一个主程序的环境中提供一些必要的扩展（见本节中后面的描述），但是基本原则是不变的。注意反过来就不一定对了，如一些嵌入式的 SQL就不能在交互的环境下运行。

在实际讨论嵌入式 SQL之前，有必要对一些基本的细节知识作一下介绍。读者可在图4-3中看到这些细节的说明。书中的宿主语言使用 PL/I，若使用其他的宿主语言，只需要做小的改动即可。下面做几点说明：

1) 为了将SQL语句与主语言分开，嵌入式 SQL以EXEC SQL开始，并以特殊的结束符号（PL/I中使用一个分号）结束。

2) 一个可执行的 SQL语句（在本节的其他部分，对嵌入式 SQL将省略嵌入式这个限定词）可以出现在任何的主语言可以执行的地方。另外，注意“可执行的”意思。嵌入式SQL包含很多说明的、但不可执行的语句，这跟交互式 SQL是不一样的。例如，DELCARE CURSOR就不是一个可执行的语句（见后面的“包括游标的操作”部分），BEGIN和END DECLARE SECTION（见下面的第5段）和WHENEVER（见下面的第9段）也不是可执行语句。

```
EXEC SQL BEGIN DECLARE SECTION ;

   DCL SQLSTATE CHAR(5) ;
   DCL P#       CHAR(6) ;
   DCL WEIGHT   FIXED DECIMAL(5,1);

EXEC SQL END DECLARE SECTION ;

P# = 'P2' ;                  /* for example                */
EXEC SQL SELECT P.WEIGHT
         INTO   :WEIGHT
         FROM   P
         WHERE  P.P# = :P# ;
IF SQLSTATE = '00000'
THEN ... ;                   /* WEIGHT = retrieved value   */
ELSE ... ;                   /* some exception occurred    */
```

图4-3 PL/I嵌入SQL程序片段

3) SQL语句可以使用主变量；但是使用之前必须在其前面加一个冒号前缀，将其与SQL的列名加以区分。在交互式 SQL中可以出现文字的任何地方，在嵌入式 SQL中主变量都可以出现。主变量还可以出现在 SELECT（见第4章下面的说明）或FETCH语句（见下面的“使用游标的操作”）中的INTO子句中，为一些修改操作指定目标。

4) 注意图4-3的SELECT语句中的INTO子句。该子句的作用是给出新值要插入到的目标变量，INTO子句中的第 *i*个目标变量对应 SELECT子句中检索的第 *i*个值。

5) 所有在SQL语句中使用的主变量必须要在嵌入式SQL的声明部分进行声明（在 PL/I中用DCL），DECLARE SECTION在BEGIN和END两个声明语句之间。

⊖ SQL标准[4.22]现在支持Ada、C、COBOL、Fortran、M(以前叫做MUMPS)、Pascal和PL/I。在本书的写作过程中，还不支持Java，但是很快会支持（参见[4.6]或着附录B），而且很多的产品现在也已经支持 Java。

6) 每一个包含嵌入式SQL语句的程序必须包括一个叫做 SQL STATE的主变量。在每一个SQL语句执行之后，执行的状态值将返回到该变量中；特别地，状态码为 00000表示该语句正确执行， 02000的状态值表示语句执行但是没有满足要求的数据返回。因此，原则上来看，程序中的每一个 SQL语句之后应该有一个对 SQL STATE的值的检测，并且当返回的结果值不是预想的情况时，对其做适当的处理。然而，这样的检测实际上是经常可以隐含的（见下面的第 9段）。

7) 主变量的数据类型必须要跟它所使用的方式一致。特别地，当一个主变量作为目标变量（如在SELECT语句中）时，它的数据类型必须跟要赋值的数据类型相一致；同样，当一个主变量要用作源变量（如在 INSERT语句中）时，它的数据类型必须与要插入到的列的数据类型相一致。在比较中使用一个主变量，或者在其他的使用中都有这个问题，因此需要注意。至于数据类型怎样相一致的细节描述可参见 [4.22]。

8) 主变量和SQL的列可以有相同的名字。

9) 如上面所提到的，每一个 SQL语句之后最好有一个检测 SQLSTATE返回值的语句。WHENEVER语句可用来实现这个功能。 WHRNEVER的语法如下：

```
EXEC SQL WHENEVER <condition> <action> ;
```

在这里 <*condition*>可以是一个 SQLERROR，也可以是一个 NOT FOUND；<*action*>或者是CONTINUE，或者是GO TO语句。WHENEVER不是一个可执行语句，而是给SQL编译器的一个说明语句。“ WHENEVER <*condition*> GO TO <*label*> ”可以使得在遇到每一个可执行的 SQL语句之后插入一个形式为“ IF <*condition*> GO TO <*label*> ”的语句；“ WHENEVER <*condition*> CONTINUE ”就不会插入这样的语句，隐含的是程序员手工插入这样的语句。这两个 <*condition*>语句是这么定义的：

```
NOT FOUND      表示      无数据
                         —SQLSTATE=02000(通常)
SQLERROR       表示      出错
                         —SQLSTATE的值见参考文献[4.22]
```

在对程序的顺序扫描过程中， SQL编译器遇到的每一个 WHENEVER语句将覆盖其前一个WHENEVER语句。

10) 最后需要注意：嵌入式 SQL（使用第2章的术语）在SQL和宿主语言之间组成了一个松散的组合。

基础知识就介绍到此，在本节的剩余部分将专门对数据操作做主要的介绍。正如前面所说的，大多数的这种操作可以以直接的方式处理，仅需要在语法上做很小的改变。然而，检索操作需要一些特殊的处理。这其中的问题是检索操作一般来说会检索到多行，而不是一行，而宿主语言一般不可能一次处理多行。因此，必须要有一种方式来协调 SQL语言的集操作的检索能力和宿主语言的行操作的处理能力，而游标就是协调两者的桥梁。游标是特殊类型的 SQL实体，只在嵌入式 SQL中使用（因为交互式 SQL不需要）。游标实际上是一个（逻辑）指针，可以使用该指针来处理数据行的集合，按顺序将指针指向每一行，这样就可以一次定位一行。有关游标的细节将在后面的小节中讨论，下面首先介绍那些不需要使用游标的语句。

1. 不用游标的操作

不需要游标的数据操作语句有：

- 查询结果为单记录的 SELECT语句
- INSERT语句
- UPDATE（除了CURRENT形式的UPDATE）语句
- DELETE（除了CURRENT形式的DELETE）语句

下面按顺序给出每个语句的示例。

查询结果为单记录的SELECT语句：列出供应商号跟主变量 GIVENS#相同的供应商的状态和所在的城市 .

```
EXEC SQL SELECT STATUS, CITY
         INTO   :RANK, :CITY
         FROM   S
         WHERE  S# = :GIVENS# ;
```

用单记录（singleton） *SELECT*表示一个SELECT语句的返回结果只有一行。在这个例子中，如果在S表中只有一条记录满足 WHERE子句的条件，则 STATUS和CITY的返回值将赋值到主变量 RANK和CITY中，并且SQLSTATE将设为00000；如果S表中没有满足条件的记录，则 SQLSTATE将为02000；如果有多于一条的记录满足条件，则程序将出错，SQLSTATE将返回一个错误代码。

INSERT：在P表中增加一个新的零件（主变量 P#、PNAME和PWT分别给出零件号、零件名和零件的重量；不知道颜色和城市）。

```
EXEC SQL INSERT
         INTO   P ( P#, PNAME, WEIGHT )
         VALUES ( :P#, :PNAME, :PWT ) ;
```

新零件的COLOR和CITY的值将设为程序的缺省值。第 5章的5.5节将对SQL的缺省值做详细的介绍。

UPDATE：将伦敦的供应商的状态增加由主变量 RAISE给定的值。

```
EXEC SQL UPDATE S
         SET    STATUS = STATUS + :RAISE
         WHERE  CITY = 'London' ;
```

如果没有满足条件的记录， SQLSTATE将置为02000。

DELETE：删除所在城市为主变量 CITY给定的供应商的记录。

```
EXEC SQL DELETE
         FROM   SP
         WHERE  :CITY =
              ( SELECT CITY
                FROM   S
                WHERE  S.S# = SP.S# ) ;
```

如果SP表中没有满足 WHERE子句中给定的条件的记录，则 SQLSTATE将被赋值为02000。另外，需要注意 WHERE子句中的嵌入式子查询。

2. 使用游标的操作

现在来看集合的检索，例如，包含任意行的集合的检索，而不是上面所说的检索结果只为一行的情况。正如上面所说的，这需要一种机制来逐行存取集合中的记录，而游标就提供了这样的机制。处理过程可见图 4-4，该例是对某些供应商的细节（ S#、SNAME和STATUS）进行查询，这些供应商所在的城市是由主变量 Y给定的。

说明：“ DECLARE X CURSOR ... ”语句定义了一个名为 X的游标，通过在 DECLARE说明中的 SELECT语句可将该游标跟一个表表达式（例如，从表中取值）相关联。因为DECLARE CURSOR是一个纯粹的说明语句，因此表的表达式不会在此时执行。当执行了

“OPEN X”，打开游标时，这个语句才执行。“FETCH X INTO ...”语句则是一次从结果集中取一行，并且根据 INTO子句中的说明将取得的值赋到主变量中（为了简单起见，这里将主变量的名字与数据库中的列名命名为一样的名字。注意在游标声明部分的 SELECT语句没有INTO子句）。因为在结果集中有多行，因此 FETCH语句一般以循环的方式出现，而且只要结果集中还有未处理的行，循环就一直进行。一旦从循环中退出，就执行“CLOSE X”，关闭游标。

```
EXEC SQL DECLARE X CURSOR FOR      /* define the cursor   */
         SELECT S.S#, S.SNAME, S.STATUS
         FROM   S
         WHERE  S.CITY = :Y
         ORDER  BY S# ASC ;

EXEC SQL OPEN X ;                  /* execute the query   */
         DO for all S rows accessible via X ;
            EXEC SQL FETCH X INTO :S#, :SNAME, :STATUS ;
                                   /* fetch next supplier */
            .........
         END ;
EXEC SQL CLOSE X ;                 /* deactivate cursor X */
```

图4-4 多行检索示例

现在详细讨论游标和游标操作。首先，是用“ DECLARE CURSOR”语句定义游标，它的形式一般是：

```
EXEC SQL DECLARE <cursor name> CURSOR
         FOR <table expression> [ <ordering> ] ;
```

为了使该形式看起来更简洁，此处忽略了一些可选的选项。 <*table expression*>的详细说明可参见附录 A。可选的<*ordering*>的语法是这样的：

```
ORDER BY <order item commalist>
```

其中<*order item*>的列表不能为空 —— 可参见后面的介绍 —— 而且每个<*order item*>包含一个列名（注意，这是没有限定的），后面跟着可选的 ASC（升序）和 DESC（降序），默认值是升序。

注意：按如下的方式定义 *commalist*（逗号列表）。使用<*xyz*>指明任意一个符合语法的类，例如，在一些 BNF产生式规则左边出现的符号。用 <*xyz commalist*>指明一个或多个<*xyz*>，在这里面，每一个 <*xyz*>由逗号做间隔，也有可能是一个或多个空格。注意会在以后的语法规则中沿用 commalist（在所有的语法规则中，而不只是 SQL语法中）。

正如前面所说的， DECLARE CURSOR语句是声明性的，而不是可执行的；它声明了一个有明确名字的游标，该游标对应明确的表的表达式，以及相关的排序。在表的表达式中可以使用主变量。一个程序可以包含任意多个 DECLARE CURSOR语句，每一个这样的语句对应一个完全不同的游标。

在游标上可以进行三种操作： OPEN、FETCH和CLOSE。

• 语句

```
EXEC SQL OPEN<cursor name>;
```

执行该语句，打开或激活一个在当前时刻没有打开的游标。跟该游标相关联的表表达式将被执行（使用在表达式中主变量的值）。这样就标识了记录集，并且使该记录集成为游标的当前活动集 (active set)。游标还定位一个指针指向活动集第一行的前面。因为活动

集总是有顺序的，所以位置的概念有一定的意义[⊖]。可以通过ORDER BY子句定义排序，在该子句缺省时，活动集中记录的顺序就是系统默认的顺序。

·语句

```
EXEC SQL FETCH<cursor name>
        INTO <host variable reference commalist>;
```

在当前活动集中将一已经打开的游标向前推进一行，并将结果的第 *i*个值赋值到INTO子句的第 *i*个主变量中。如果 FETCH语句执行时记录集中已经没有记录， SQLSTATE将被赋值为02000，并且不返回任何待修改的数据。

·语句

```
EXEC SQL CLOSE<cursor name>;
```

关闭一个当前已经打开的游标或使一个当前已经打开的游标无效。关闭后的游标没有了当前活动集。然而，它还可以被重新打开，在游标重新打开时，它将跟另外一个活动集对应，该活动集可能不是上一个同样的活动集，在游标声明中引用的主变量的值改变的情况下，更是这样。注意在游标打开的情况下改变主变量的值在当前的活动集中是无效的。

游标还可以跟另外两个语句结合使用。它们是： CURRENT形式的UPDATE语句和DELETE语句。如果将一个名为 X的游标定位在某一特定的行，就可以对“ X游标的当前内容”进行UPDTAE或DELETE操作，如， X游标指向的当前记录。例：

```
EXEC SQL UPDATE S
         SET    STATUS = STATUS + :RAISE
         WHERE  CURRENT OF X ;
```

注意：如果表的表达式中引用的是一个不可更新的视图（见第 9章，第9.6节），那么CURREN形式的UPDATE和DELETE语句就是不允许的。

3. 动态SQL

动态SQL给嵌入式 SQL提供了很多的便利，可以支持开发一般化的、联机的或是交互的应用程序（读者可以回想第 1章中所介绍的联机应用程序，它支持从联机终端存取数据库）。概括地说，一个典型的联机应用程序。要按照如下的步骤执行：

1) 从终端接受命令。

2) 分析命令。

3) 在数据库上执行适当的 SQL语句。

4) 将消息或结果返回到终端。

如果程序在第一步中收到的命令集非常小，如航班预订处理程序的情况，那么要执行的SQL语句就会非常少，因此就可以在程序中做固定处理。在这种情况下，第 2步和第3步就是简单的检查输入的命令，并将其分解到处理预定义的 SQL语句的分支程序进行处理。另一方面，如果输入有很大的灵活性，进行预处理和固定某些处理就是不恰当的。相反，这个时候如果动态地构造 SQL语句就方便得多，然后对构造的 SQL 语句进行动态编译和执行。在这个过程中，动态 SQL的便利性就会有很大的帮助。

两个基本的动态语句是 PREPARE和EXECUTE。可以通过下面的例子（不是很实际，

⊖ 当然，并不是每一个集合都有顺序（见第 5章），因此与其说一个“活动集”是一个集合，还不如说它是一个记录的有序列表或者是数组。

但是非常精确）来说明它们的使用：

```
DCL SQLSOURCE CHAR VARYING (65000) ;

SQLSOURCE = 'DELETE FROM SP WHERE QTY < 300' ;
EXEC SQL PREPARE SQLPREPPED FROM :SQLSOURCE ;
EXEC SQL EXECUTE SQLPREPPED ;
```

说明：

1) SQLSOURCE标识了一个 PL/I的变长字符串变量，该变量需要在程序中构造其 SQL语句的源的形式，如，字符串的表现形式。例子中是一个 DELETE语句。

2) SQLPREPPED标识了一个 SQL的变量，而不是一个 PL/I的变量。该变量将用来接收编译后的 SQL语句，这些 SQL语句的源形式是由 SQLSOURCE（当然SQLSOURCE和SQLPREPPED是任选的）给出的。

3) PL/I的赋值语句“ SQLSOURCE=…”将一个 SQL SELETE语句的源形式赋值到SQLSOURCE中。当然，实际构造这样一个源语句的过程非常灵活，可能还包括对终端用户用自然语言或者是用比 SQL更友好的语言输入的请求的分析。

4) PREPARE取得源语句，并且准备将它处理（如编译）为可执行的形式，存放在SQLPREPPED中。

5) 最后，EXECUTE语句执行 SQLPREPPED，并因此生成真正的 DELETE。执行完后将返回值置到 SQLSTATE中，就像是直接运行 DELETE语句一样。

因为 SQLPREPPED声明的是一个 SQL变量，而不是一个 PL/I变量，因此当它在PREPARE和EXECUTE语句中使用时不需要加冒号。而且这样的变量可以不必显式声明。

有时候，上述处理过程与 SQL语句交互式输入的处理情况类似。大多数的系统都包含交互式SQL查询处理器。这些查询处理器其实是特殊的联机应用程序，它可以接受多种类型的输入，不仅是可以接受有效的 SQL语句，有时还可以接受无效的 SQL语句。它根据用户的输入，使用动态 SQL的便利性构造适当的 SQL语句，编译和执行构造好的语句，并将结果信息或结果集返回到终端。

下面简单介绍一下最近（ 1995）有关 SQL 调用级接口 （Call-Level Interface，“ SQL/CLI ”，简写为 CLI）标准的进展情况。 CLI建立在微软的开放数据库互连接口 ODBC的基础之上。 CLI允许一个用一般的宿主语言写的应用程序通过激活开发商提供的 CLI程序来发出数据库操作请求。那些跟应用程有一定关联的 CLI程序代替应用程序使用动态SQL执行数据库的操作请求。换句话说，从 DBMS的角度来看， CLI程序可以简单地理解为是一个一般的应用程序。

读者可以看到， SQL/CLI（和 ODBC）跟动态 SQL一样，都针对了同样的问题：应用程序中的 SQL语句可以在程序运行前不完全确定。然而， SQL/CLI对这个问题提供了一个比动态SQL更好的解决办法。这有两个原因：

首先，动态 SQL是一个源码级的标准。任何使用动态 SQL的应用程序如果要处理按照动态SQL标准所写的操作，如， PREPARE，EXCUTE，等等，都需要 SQL编译器的支持。而CLI则与之不同，它将某些例行程序的细节进行了标准化（例如，子程序的调用等）；因而不需要特殊的编译处理，而只需要一般的宿主语言编译器的支持。因此，应用程序可以以一种封装的目标代码的形式可以由第三方开发商提供。

第二，CLI应用程序可以具有 DBMS独立性。也就是说， CLI具有允许创建一般性应用

程序的特性（可以由第三方软件开发商提供），所创建的应用程序可在几个 DBMS上使用，而不是只能应用到特定的 DBMS上。

现在，CLI、ODBC和JDBC（一个Java的ODBC）等接口标准在实际中越来越重要，具体的原因，读者可以参见第 20章的内容。

4.7 SQL是不完美的

根据本章前面的介绍，可以看出 SQL远不是“完美的”关系语言，因为 SQL比较冗长，而且是委托的 (omssion and commission)。以后的章节中会介绍 SQL的某些不足之处，但是SQL的最大的问题在于：它在很多方面不能很好地支持关系模型。因此，现在的很多 SQL产品不能说是真正的关系型的产品。确实，就笔者的观察来看，现在市场上的产品没有真正支持关系模型的各个细节的。这并不是说模型的某些部分是不重要的；相反，模型的每一个细节都是很重要的，而且从实际的使用来看更重要。但是，不应该过分强调这一点，因为关系理论的本质不是要为了理论而理论，而是要给实际的百分之百的实用系统做基础。但是不太乐观的是各个开发商都没有真正将理论作为整体来实施。因此，从某些方面来说，现在的“关系的”产品都没有能够真正贯彻关系理论。

4.8 小结

本章包括对 SQL标准（“SQL/92”）的一些主要特征的介绍。从商业的角度来强调了SQL的重要性，虽然从纯关系的角度来看，这是不很合适的。

SQL包括数据定义语言（DDL）和数据操作语言（DML）部分。DML可以在外模式（视图）上操作，也可以在概念模式（基表）上操作。同样，SQL的DDL既可以用在外模式（视图）上，也可以用在概念模式（基表）上定义实体。在一些商业系统中，甚至还支持内模式（如索引和其他的物理存储）。另外，SQL还有数据控制的功能，例如，它可以提供很多既不属于 DDL也不属于DML的功能。GRANT便是其中一例，该语句允许用户互相授权存取权限（见第 16章）。

本章说明了如何使用 SQL中的CREATE TABLE语句创建基本表。并且接着给出了SELECT、INSERT、UPDATE和DELETE语句的例子，说明了如何利用 SELECT来实现选择、投影和连接运算。还简单描述了信息模式，信息模式包含了一些假定的“定义模式”的视图。本章还对 SQL在处理视图和事务中的功能做了简单的介绍。

本章的大部分篇幅主要介绍嵌入式SQL。嵌入式 SQL的基本思想是双重模式的原则，即，可以交互 SQL语句，也可以在一个应用程序中使用。这一原则的一个主要例外是多行检索操作，在这种情况下，需要使用游标来协调 SQL的集合操作和宿主语言如 PL/I的单行操作的差异。

在讲解了一些基本知识（虽然大多数都是语法的讲解）——包括对SQLSTATE的简单说明——之后，主要介绍了下列几种操作：查询单记录的 SELECT语句、INSERT、DELETE和UPDATE语句，对它们的操作都不需要游标。之后又介绍了需要游标的操作，讨论了 DECLARE CURSOR、OPEN、FETCH、COLE和CURRENT形式的 UPDATE和DELETE语句（标准中将 CURRENT形式的这些操作称为是定位型 (positioned) UPDATE和DELETE语句；将非 CURRENT形式的 UPDATE和DELETE语句称为是搜索的 UNDATE和

DELETE语句)。最后，简要介绍了动态 SQL的概念，特别描述了 PREPARE和EXECUTE语句，还提到了 SQL调用级接口 CLI。

练习

4.1 图4-5给出了一些示例数据，这些数据来自供应商、零件数据库的扩展数据库：供应商-零件-项目数据库。供应商（ S)、零件（ P）和项目（ J）有唯一的标识，分别为供应商号（ S#)、零件号（ P#）和项目号（ J#)。SPJ的一行表示特定的供应商给某一项目供应特定数量的某一零件（ S#-P#-J#的联合可以唯一地标识这样的一行)。为该数据库写一个适当的 SQL定义。注意：这个数据库将要作为以后章节练习的基础。

S

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

P

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

J

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

SPJ

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

图4-5 供应商-零件-项目数据库（示例数据）

4.2 在4.2节中，按照 SQL标准介绍了 CREATE TABLE语句。然而，许多商用的 SQL产品还在该语句的基础之上支持附加的选项，尤其是在索引的处理，磁盘空间的分配和其他的一些事务的执行方面。这样就破坏了数据的物理独立性和系统间的兼容性。调查任何一个你可得到的 SQL 产品。看一下：自己预先对该产品的某些批评是否还是恰当的？特别地，写出该产品支持哪些 CREATE TABLE附加的选项。

4.3 对某一SQL产品再进行一次调查。思考下列问题：该产品是否支持信息模式？如果不支持，它的目录支持是什么样子的？

4.4 写出下列对供应商 -零件-项目数据库的更新操作的 SQL语句：

a) 在S表中插入一个新的供应商 S10。供应商的名字是 Smith，其所在的城市是 New York，其状态还是未知的。

b) 将所有颜色为红色的零件改为颜色为橙色。

c) 删除所有没有供货记录的项目。

4.5 使用供应商 -零件-项目数据库，写一个嵌入式 SQL的语句，按照供应商号列出所有的供应商。每一个供应商后面必须紧跟着列出该供应商供应产品的所有项目行，项目

行按照项目号排序。

4.6 假定PART和PART-STRUCTURE表是如下定义的：

```
CREATE TABLE PART
     ( P# ... , DESCRIPTION ... ,
       PRIMARY KEY ( P# ) ) ;

CREATE TABLE PART_STRUCTURE
     ( MAJOR_P# ... , MINOR_P# ... , QTY ... ,
       PRIMARY KEY ( MAJOR_P#, MINOR_P# ),
       FOREIGN KEY ( MAJOR_P# ) REFERENCES PART,
       FOREIGN KEY ( MINOR_P# ) REFERENCES PART ) ;
```

PART-STRUCTURE表说明了组成某一个零件 (MAJOR_P#)第一层的其他零件(MINOR_P#)。写一个SQL程序列出某一个给定零件的各层的组成零件（即零件爆炸问题）。注意：在图 4-6中的示例数据对解决这个问题会有帮助。可以看到，表 PART-STRUCTURE显示了材料单数据（见第 1章，第1.3节的“实体和联系”）在关系系统中如何描述。

PART_STRUCTURE

MAJOR_P#	MINOR_P#	QTY
P1	P2	2
P1	P3	4
P2	P3	1
P2	P4	3
P3	P5	9
P4	P5	8
P5	P6	3

图4-6 表PART-STRUCTURE（样本数据）

参考文献和简介

4.1 M. M. Astrahan and R. A. Lorie: “SEQUEL-XRM: A Relational System” Proc.ACM Pacific Regional Conf., San Francisco, Calif. (April 1975).

描述了第一个 SEQUEL实现的原型和 SQL[4.8]的最早的版本。参见 [4.2~4.3]。SEQUEL的功能跟 System R类似。

4.2 M. M. Astrahan *et al*.: “System R: Relational Approach to Database Management,” *ACM TODS 1*, No. 2 (June 1976).

System R是SEQUEL/2（以后的 SQL）语言 [4.8]的主要实现原型。该文介绍了在初始计划中 System R的体系结构。参见 [4.3]。

4.3 M. W. Blasgen *et al*.: “System R: An Architectural Overview,” *IBM Sys. J. 20*, No. l (February 1981).

描述了 System R已经完全实现之后的体系结构。参照 [4.2的说明和注释]。

4.4 Stephen Cannan and Gerard Otten: *SQL - The Standard Handbook*. Maidenhead, UK: McGraw-Hill International (1993).

“[我们的]目标……是提供一本解释和介绍 SQL/92初始定义的手册，该手册不会像标准那样形式化，但是比标准更加易读。”（摘自该书的概述部分）

4.5 Joe Celko: *SQL for Smarties: Advanced SQL Programming*. San Francisco, Calif.: Morgan Kaufmann (1995).

“这是第一本有用的高级 SQL手册，对于想从 SQL的一般用户升级到专业编程员的人来说，可以提供全面的技术介绍。”（摘自书的封面）

4.6 Andrew Eisenberg and Jim Melton: "SQLJ Part 0, Now Known as SQL/OLB (Object Language Bindings)," *ACM SIGMOD Record 27*, No. 4 (December 1998). See also Gray Clossman *et al.*: "Java and Relational Databases: SQLJ," Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash. (June 1998).

4.7 Don Chamberlin: *Using the New, DB2*. San Francisco, Calif.: Morgan Kaufmann (1996).

这是一本对商业 SQL产品的现状做了全面介绍、可读性非常好的书籍，由 SQL的两个最初的主要设计者所写。

注意：这本书也讨论了在 SQL设计过程中"一些有争议的结论"。主要是： (a)**对空值的支持；和** (b)**允许重复行。"** Chamberlin**认为对这两个问题的认识已经形成定论了，现在不需要来进行说理性的推理，他认为空值和重复行的问题是一个类似于宗教信仰的问题。最主要的是，** [SQL]**的设计者是实用者而不是理论家，这种倾向可以在设计的很多决定上看出来"。这种立场跟现在的许多作者有很大的不同！空值和重复值是一个科学的问题，而不是一个宗教信仰的问题；在这本书的第 18和第5章分别科学地说明了这两个问题。在"实用的而不是 [理论的]"这个问题上，认为理论的就是不实用的观点是不正确的；在第 4.5小节中已经表明了作者的观点：关系理论至少是非常实用的。**

4.8 Donald D. Chamberlin and Raymond F. Boyce: "SEQUEL: A Structured English Query Language," Proc. 1974 ACM SIGMOD Workshop on Data Description, Access, and Control, Ann Arbor, Mich. (May 1974).

这篇论文首先介绍了 SQL语言（或者按照该论文作者原先的叫法是 SEQUEL，因为法律上的原因，名字随后做了修改）。

4.9 Donald D. Chamberlin *et al.*: "SEQUEL/2: A Unified Approach to Data Definition, Manipulation, and Control," *IBM J. R&D. 20*, No. 6 (November 1976). See also the errata in *IBM J. R&D. 21*, No. 1 (January 1977).

基于以下两个条件的基础，即： [4.1]**中对** SEQUEL**原型的实现经验和** [4.28]**中报告的可用测试的结果，提出了修订版 SEQUEL/2。**System R[4.2~4.3]**支持的语言就是** SEQUEL/2**，但是该语言没有"断言"和"触发器"的功能（见第 8章），该版还根据早期用户的使用经验** [4.10]**增加了一定的扩充功能。**

4.10 Donald D. Chamberlin: "A Summary of User Experience with the SQL Data Sublanguage," Proc. Int. Conf. on Databases, Aberdeen, Scotland (July 1980). Also available as IBM Research Report RJ2767 (April 1980).

给出了早期使用 System R**的用户经验，并且根据这些经验，提出了对 SQL语言的一些扩充。有的扩充，如** EXISTS、LIKE、PREPARE**和**EXECUTE**等，实际上一直到** System R**的最后版本中才实现。注意：读者可分别参照第 7章和附录** A**来查看** EXISTS **和**LIKE**。**

4.11 Donald D. Chamberlin *et al.*: "Support for Repetitive Transactions and *Ad Hoc* Queries in System R," *ACM TODS 6*, No. 1 (March 1981).

给出了 System R**在特殊查询和"封装事务" 环境中的性能测试的很多指标。一个"封装事务"是一个可存取数据库的一小部分的应用程序，需要在执行之前编译。**

它与第2章2.8节的计划请求相对应。该测试是在 IBM的System 370 Model 158上，在VM操作系统下，运行 System R进行的。虽然这些认识是“最基本的”，但是，论文仍然说明了：在其他的情况下， (a)编译总是优于解释的，甚至在特殊查询下也不例外；(b)如果系统中有适当的索引，那么一个跟 System R相似的系统有能力在一秒钟之内处理几个封装事务。

当时，很多人宣称“关系系统是没有用的”，而这篇论文则认为这种说法不恰当，该论文也因为第一次对该观点提出质疑而闻名。当然，自从它第一次发表以来，商业的关系产品在事务处理的速度上已经达到了每秒钟上百个或上千个的事务处理能力。

4.12 Donald D. Chamberlin *et al.*: “A History and Evaluation of System R,” CACM 24, No. 10 (October 1981).

介绍了System R工程主要的三个阶段（初步的原型、多用户的原型和对原型的评价），强调了编译和优化技术，而就是这些技术使得 System R比较超前。在这篇论文和参考资料[4.13]之间有一定的重复。

4.13 Donald D. Chamberlin, Arthur M. Gilbert, and Robert A. Yost: “A History of System R and SQL / Data System,” Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France (September 1981).

讨论了从System R原型中吸取的经验和教训，介绍了将该原型引入到 IBM的DB2产品家族SQL/DS中的过程（随后，就将 DB2改名为VM和VSE）。

4.14 C. J. Date: “A Critique of the SQL Database Language,” *ACM SIGMOD Record 14*, No. 3 (November 1984). Republished in *Relational Database*: *Selected Writings*. Reading, Mass.: Addison-Wesley (1986).

正如在本章中所说的， SQL并不完美。这篇文章对该语言的很多缺点给出了批评的分析（主要从形式化的计算机语言而不是专门从数据库语言来看）。注意：这篇文章的批评有的不适用于 SQL/92。

4.15 C. J. Date: “What's Wrong with SQL? ”, in *Relational Database Writings 1985-1989*. Reading, Mass.: Addison-Wesley (1990).

除去[4.14]中所说的SQL的一些缺点，这篇文章还以下面的标题讨论了 SQL的一些其他的不足之处：“SQL本身 的不足之处”、“SQL标准的不足之处”和“应用的可移植性”。注：本文的一些批评同样不适用 SQL/92。

4.16 C. J. Date: “SQL Dos and Don'ts,” in *Relational Database Writings 1985-1989*. Reading, Mass.: Addison-Wesley (1990).

在[4.14]和[4.15]的参考资料中说到了 SQL的很多潜在的不足，这篇文章给出了很多使用SQL的建议，以尽量避免这些不足，而且文章中还给出建议以充分利用开发效率、可移植性和可连接性等，实现效益的最大化。

4.17 C. J. Date: “How We Missed the Relational Boat,” in *Relational Database Writings 1991-1994*. Reading, Mass.: Addison-Wesley (1995).

通过介绍SQL对关系模型的结构、完整性和操作的支持，简洁地说明了 SQL的不足之处。

4.18 C. J. Date: “Grievous Bodily Harm” (in two parts), *DBP&D 11*, No. 5 (May 1998) and *11*, No. 6

(June 1998); "Fifty Ways to Quote Your Query," on the *DBP&D* website *www.dbpd.com* (July 1998).

SQL是一个冗余非常大的语言，就是差别很小的查询也有多种不同的表达方式。这些文章说明了这一点，并且介绍了它们隐含的不同之处。特别地，文章还说明了GROUP BY子句、HAVING子句和范围变量可以在不失去功能的情况下从语言中有效地去掉（在子查询结构中也一样）。注意：所有的这些SQL结构在第7章（第7.7节）或附录A中有介绍。

4.19 C. J. Date and Hugh Darwen: *A Guide to the SQL Standard* (4th edition). Reading, Mass.: Addison-Wesley (1997).

这是一个更全面的SQL/92的使用指南，包括CLI和PSM。特别地，该书中的附录D说明了"标准中没有恰当定义的，甚至是没有正确定义的很多方面"。注意：参考书[4.4]和[4.27]也是SQL/92的使用指南。

4.20 C. J. Date and Colin J. White: *A Guide to DB2* (4th edition). Reading, Mass.: Addison-Wesley (1993).

对IBM最初的产品DB2和一些相关产品做了广泛全面的综述。DB2跟SQL/DS[4.13]一样，是在System R的基础上开发的。

4.21 Neal Fishman: "SQL *du Jour*," *DBP&D 10*, No. 10 (October 1997).

描述了在一些宣称"支持SQL标准"的SQL产品中发现的很多不能令人满意的不相容性。

4.22 International Organization for Standardization (ISO): *Database Language SQL*, Document ISO/IEC 9075:1992. Also available as American National Standards Institute (ANSI) Document ANSI X3.135-1992.

这是最初的ISO/ANSI SQL/92的定义（有时候被称为是*ISO/IE 9075*，有时候仅仅是*ISO 9075*）。在一般的标题*信息技术 —— 数据库语言 —— SQL*的基础上，原先的单个文档已经扩展到一系列开放的单独部分。在写该书的时候，下面的部分已经定义（虽然还没有都完成）：

第一部分：框架（SQL/框架）
第二部分：基础（SQL/基础）
第三部分：调用接口（SQL/CLI）
第四部分：永久存储模块（SQL/PSM）
第五部分：宿主语言捆绑（SQL/绑定）
第六部分：XA专门化（SQL/事务）
第七部分：时态（SQL/时态）
第八部分：没有第八部分
第九部分：外部数据的管理（SQL/MED）
第十部分：对象语言绑定（SQL/OLB）

预计会在1999年得到批准的SQL3建议属于第一，二，四和五部分。读者可以在*ftp://jerry.ece.umassed.edu/isowg3/dbl/BASEdocs/public*站点上查找到描述这些建议的草案。

注意：值得一提的是，虽然 SQL一直为国际“关系”数据库标准所承认，但是这个标准的文档并没有这样描述本身的一些内容；实际上，文档中从来没有真正使用过“关系”这个名词（正如在前面的一个脚注中所说的，文档中也从来没有提到过“数据库”的概念）。

4.23 International Organization for Standardization (ISO): *Information Technology-Database Languages-SQL-Technical Corrigendum* 2, Document ISO/IEC 9075:1992/Cor.2:1996(E).

包括了对[4.22]中所说的原始版本的大量修订和更正。但是，不幸的是，这些修订和更正并没有解决[4.19]中提到的问题。

4.24 Raymond A. Lorie and Jean-Jacques Daudenarde: *SQL and Its Applications*. Englewood Cliffs, N.J.: Prentice-Hall (1991).

关于如何使用“SQL”的书（该书中有一半的篇幅给出了具体的实际程序的学习示例）。

4.25 Raymond A. Lorie and J. F. Nilsson: “An Access Specification Language for a Relational Data Base System,” *IBM J. R&D. 23*, No. 3 (May 1979).

对System R[4.11，4.25~4.26]的编译机制给出了详细的解释。对于任何一个给定的SQL语句，System R优化器生成一个名为 ASL（Access Specification Language）的内部语言程序。ASL语言是优化器和代码生成器的接口（代码生成器将一个 ASL程序转为机器代码）。ASL包括在索引、存储文件等实体上的“扫描”和“插入”的操作。ASL通过将程序分解为一些定义好的子过程，使所有的翻译过程更好管理。

4.26 Raymond A. Lorie and Bradford W. Wade: “The Compilation of a High-Level Data Language,” IBM Research Report RJ2598 (August 1979).

在System R中，查询的编译过程放在执行之前，而且在物理数据库结构有了明显的改变之后会自动地重编译，这种做法比较超前。这篇文章非常详细地论述了System R的编译和重编译机制，然而没有涉及优化的问题，该问题可参见后面的[17.34]。

4.27 Jim Melton and Alan R. Simon: *Understanding the New, SQL: A Complete Guide*. San Mateo, Calif.: Morgan Kaufmann (1993).

一本SQL/92（按照原先的定义）的使用指南。Melton是最初的SQL/92[4.22]规范的制定者之一。

4.28 Phyllis Reisner, Raymond F. Boyce, and Donald D. Chamberlin: “Human Factors Evaluation of Two Data Base Query Languages: SQUARE and SEQUEL,” Proc. NCC 44, Anaheim, Calif. Montvale, N.J.: AFIPS Press (May 1975).

SQL之前的SEQUEL[4.8]建立在早期的 SQUARE语言的基础之上。这两种语言的功能实际上是一样的，但是 SQUARE使用更多的数学语法，而 SEQUEL使用较多的英语关键字，如：SELECT、FROM、WHERE，等等。这篇文章介绍了利用大学生作为研究对象，这两种语言的可使用性方面的一些实验，在该研究的基础上，对SEQUEL作了许多修订。

4.29 David Rozenshtein, Anatoly Abramovich, and Eugene Birger: *Optimizing Transact-SQL: Advanced Programming Techniques*. Fremont, Calif.: SQL Forum Press (1995).

Sybase 和SQL Server的产品支持 SQL，并且产生了 Transact-SQL这一特色 SQL。这本书介绍了许多 Transact-SQL的基于一些特色函数（按照作者的定义：“ 是一种可以允许程序员用 SELECT、WHERE和GROUP BY子句编写有条件的程序逻辑 ”）的一系列编程方法。虽然这是专门用 Transact-SQL编写的，但是这个思想在实际中有很大的适用性。注意：应该说明，在本书标题中提到的“ 优化 ”不是指 DBMS的优化器，而是用户可以自己手工执行的优化。

部分练习答案

4.1
```
CREATE TABLE S
     ( S#       CHAR(5),
       SNAME    CHAR(20),
       STATUS   NUMERIC(5),
       CITY     CHAR(15),
     PRIMARY KEY ( S# ) ) ;

CREATE TABLE P
     ( P#       CHAR(6),
       PNAME    CHAR(20),
       COLOR    CHAR(6),
       WEIGHT   NUMERIC(5,1),
       CITY     CHAR(15),
     PRIMARY KEY ( P# ) ) ;

CREATE TABLE J
     ( J#       CHAR(4),
       JNAME    CHAR(20),
       CITY     CHAR(15),
     PRIMARY KEY ( J# ) ) ;

CREATE TABLE SPJ
     ( S#       CHAR(5),
       P#       CHAR(6),
       J#       CHAR(4),
       QTY      NUMERIC(9),
     PRIMARY KEY ( S#, P#, J# ),
     FOREIGN KEY ( S# ) REFERENCES S,
     FOREIGN KEY ( P# ) REFERENCES P,
     FOREIGN KEY ( J# ) REFERENCES J ) ;
```

4.4 a.
```
INSERT INTO S ( S#, SNAME, CITY )
       VALUES ( 'S10', 'Smith', 'New York' ) ;
```

此处的STATUS将设为适当的缺省值。

b.
```
UPDATE P
SET    COLOR = 'Orange'
WHERE  COLOR = 'Red' ;
```

c.
```
DELETE
FROM   J
WHERE  J# NOT IN
     ( SELECT J#
       FROM   SPJ ) ;
```

注意c方法中的嵌套子查询和 IN操作符（实际上，是非 IN操作符）。详细的解释可参见第7章。

4.5 注意，有些供应商并没有给任何一个项目供应零件；下面的答案处理了这样的情况。首先，如下定义 CS和CJ两个游标：

```
EXEC SQL DECLARE CS CURSOR FOR
         SELECT S.S#, S.SNAME, S.STATUS, S.CITY
         FROM   S
         ORDER  BY S# ;

EXEC SQL DECLARE CJ CURSOR FOR
```

```
SELECT J.J#, J.JNAME, J.CITY
FROM   J
WHERE  J.J# IN
       ( SELECT SPJ.J#
         FROM   SPJ
         WHERE  SPJ.S# = :CS_S# )
ORDER  BY J# ;
```

（注意嵌套子查询和 IN操作符。）

当打开游标 CJ时，供应商号的值将放入主变量 CS_S#中，并且通过 CS游标来向前推进。下面给出主要的程序：

```
EXEC SQL OPEN CS ;
DO for all S rows accessible via CS ;
   EXEC SQL FETCH CS INTO :CS_S#, :CS_SN, :CS_ST, :CS_SC ;
   print CS_S#, CS_SN, CS_ST, CS_SC ;
   EXEC SQL OPEN CJ ;
   DO for all J rows accessible via CJ ;
      EXEC SQL FETCH CJ INTO :CJ_J#, :CJ_JN, :CJ_JC ;
      print CJ_J#, CJ_JN, CJ_JC ;
   END DO ;
   EXEC SQL CLOSE CJ ;
END DO ;
EXEC SQL CLOSE CS ;
```

4.6 这是一个 SQL/92无法处理好的一个问题的例子。主要的困难是：需要将给定的零件扩展到 *n*层，但是在写程序的时候， *n*的值却是未知的。一个比较直接的、执行这样一个 *n*层的扩展的解决办法是 —— 如果可能的话 —— 递归该程序，如下所示，在每一次递归中产生一个新的游标：

```
CALL RECURSION ( GIVENP# ) ;

RECURSION: PROC ( UPPER_P# ) RECURSIVE ;
    DCL UPPER_P# ... ;
    DCL LOWER_P# ... ;
    EXEC SQL DECLARE C "reopenable" CURSOR FOR
             SELECT MINOR_P#
             FROM   PART_STRUCTURE
             WHERE  MAJOR_P# = :UPPER_P# ;

    print UPPER_P# ;
    EXEC SQL OPEN C ;
    DO for all PART_STRUCTURE rows accessible via C ;
       EXEC SQL FETCH C INTO :LOWER_P# ;
       CALL RECURSION ( LOWER_P# ) ;
    END DO ;
    EXEC SQL CLOSE C ;
END PROC ;
```

此处假定在 DECLARE CURSOR上的“可重新打开的”的说明的意思是：即使游标已经打开， OPEN也是合法的，这时候使用 OPEN会为某个特定的表达式（使用表达式中任何宿主语言的当前值）创建一个该游标的新的实例。本书还进一步假定在 FETCH语句中使用这样一个游标也是使用当前的实例。换句话说，假定可以重新打开的游标组成了一个栈，而 OPEN和CLOSE就是栈的“ push ”和“ pop ”操作符。

不幸的是，这样的设想现在是纯粹的假设。因为现在在 SQL中还没有一个可以重新打开的游标，实际上，打开一个已经打开的游标会失败，因此前面的代码是不正确的。但是它也说明了一个“可以重新打开的”游标是一个很好的扩展。

既然前面的方法不能有效地工作，现在给出一个可行但是效率非常低的方法。

```
CALL RECURSION ( GIVENP# ) ;
RECURSION: PROC ( UPPER_P# ) RECURSIVE ;
    DCL UPPER_P# ... ;
    DCL LOWER_P# ... INITIAL ( '      ' ) ;
    EXEC SQL DECLARE C CURSOR FOR
                     SELECT MINOR_P#
```

```
                    FROM    PART_STRUCTURE
                    WHERE   MAJOR_P# = :UPPER_P#
                    AND     MINOR_P# > :LOWER_P#
                    ORDER   BY MINOR_P# ;

    print UPPER_P# ;
    DO "forever" ;
       EXEC SQL OPEN C ;
       EXEC SQL FETCH C INTO :LOWER_P# ;
       EXEC SQL CLOSE C ;
       IF no "lower P#" retrieved THEN RETURN ; END IF ;
       IF "lower P#" retrieved THEN
          CALL RECURSION ( LOWER_P# ) ; END IF ;
    END DO ;
END PROC ;
```

这个解决办法中，需要介绍一下每一个 RECURSION中使用的相同的游标：每一次RECURSION重新激活的时候，就动态地创建一个 UPPER_P#和一个LOWER_P#的实例，并且这些实例在本次运行结束时就全部被破坏。因为这个事实，我们要采取一个小的技巧：

```
... AND MINOR_P# > :LOWER_P# ORDER BY MINOR_P#
```

因此，在每一次使用完 RECURSION之后，忽略当前已经处理过的 UPPER_P#的所有临时的内容（LOWRE_P#）。

参见：(a) 参考文献[4.5]和[4.7]有该问题的SQL方式的解决办法，(b) 第6章的6.7节末尾的与名为传递闭包相关的关系操作的描述； (c)附录B概述的SQL3的有关功能。