中级 SQL

本章我们继续学习 SQL。我们考虑具有更复杂形式的 SQL 查询、视图定义、事务、完整性约束、关于 SQL 数据定义的更详细介绍以及授权。

4.1 连接表达式

在 3.3.3 节我们介绍了**自然连接**运算。SQL 提供了连接运算的其他形式，包括能够指定显式的**连接谓词**（join predicate），能够在结果中包含被**自然连接**排除在外的元组。本节我们将讨论这些连接的形式。

本节的例子涉及 *student* 和 *takes* 两个关系，分别如图 4-1 和图 4-2 所示。注意到对于 ID 为 98988 的学生，他在 2010 夏季选修的 BIO-301 课程的 1 号课程段的 *grade* 属性为空值。该空值表示这门课程的成绩还没有得到。

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

图 4-1 *student* 关系

4.1.1 连接条件

在 3.3.3 节我们介绍了如何表达自然连接，并且介绍了 **join…using** 子句，它是一种自然连接的形式，只需要在指定属性上的取值匹配。SQL 支持另外一种形式的连接，其中可以指定任意的连接条件。

on 条件允许在参与连接的关系上设置通用的谓词。该谓词的写法与 **where** 子句谓词类似，只不过使用的是关键词 **on** 而不是 **where**。与 **using** 条件一样，**on** 条件出现在连接表达式的末尾。

考虑下面的查询，它具有包含 **on** 条件的连接表达式：

```
select *
from student join takes on student.ID = takes.ID;
```

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	*null*

图 4-2 *takes* 关系

上述 **on** 条件表明：如果一个来自 *student* 的元组和一个来自 *takes* 的元组在 *ID* 上的取值相同，那么它们是匹配的。在上例中的连接表达式与连接表达式 *student* **natural join** *takes* 几乎是一样的，因为自然连接运算也需要 *student* 元组和 *takes* 元组是匹配的。这两者之间的一个区别在于：在上述连接查询结果中，*ID* 属性出现两次，一次是 *student* 中的，另一次是 *takes* 中的，即便它们的 *ID* 属性值是相同的。

实际上，上述查询与以下查询是等价的（换言之，它们产生了完全相同的结果）：

```
select *
from student, takes
where student.ID = takes.ID;
```

正如我们此前所见，关系名用来区分属性名 *ID*，这样 *ID* 的两次出现被分别表示为 *student. ID* 和 *takes. ID*。只显示一次 *ID* 值的查询版本如下：

```
select student. ID as ID, name, dept_name, tot_cred,
    course_id, sec_id, semester, year, grade
from student join takes on student. ID = takes. ID;
```

上述查询的结果如图 4-3 所示。

ID	name	dept_name	tot_cred	course_id	sec_id	semester	year	grade
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2010	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2009	A
19991	Brandt	History	80	HIS-351	1	Spring	2010	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2010	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B+
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	*null*

图 4-3 *student* **join** *takes* **on** *student. ID = takes. ID* 的结果，其中省略了 *ID* 的第二次出现

on 条件可以表示任何 SQL 谓词，从而使用 **on** 条件的连接表达式就可以表示比**自然连接**更为丰富的连接条件。然而，正如上例所示，使用带 **on** 条件的连接表达式的查询可以用不带 **on** 条件的等价表达式来替换，只要把 **on** 子句中的谓词移到 **where** 子句中即可。这样看来，**on** 条件似乎是一个冗余的 SQL 特征。

但是，引入 **on** 条件有两个优点。首先，对于我们马上要介绍的，被称作外连接的这类连接来说，**on** 条件的表现与 **where** 条件是不同的。其次，如果在 **on** 子句中指定连接条件，并在 **where** 子句中出现其余的条件，这样的 SQL 查询通常更容易让人读懂。

4.1.2 外连接

假设我们要显示一个所有学生的列表，显示他们的 *ID*、*name*、*dept_name* 和 *tot_cred*，以及他们所选修的课程。下面的查询好像检索出了所需的信息：

```
select *
from student natural join takes;
```

遗憾的是，上述查询与想要的结果是不同的。假设有一些学生，他们没有选修任何课程。那么这些学生在 *student* 关系中所对应的元组与 *takes* 关系中的任何元组配对，都不会满足自然连接的条件，从而这些学生的数据就不会出现在结果中。这样我们就看不到没有选修任何课程的学生的任何信息。例如，在图 4-1 的 *student* 关系和图 4-2 的 *takes* 关系中，*ID* 为 70557 的学生 Snow 没有选修任何课程。Snow 出现在 *student* 关系中，但是 Snow 的 ID 号没有出现在 *takes* 的 *ID* 列中。从而 Snow 不会出现在自然连接的结果中。

更为一般地，在参与连接的任何一个或两个关系中的某些元组可能会以这种方式“丢失”。**外连接**（outer join）运算与我们已经学过的连接运算类似，但通过在结果中创建包含空值元组的方式，保留了那些在连接中丢失的元组。

例如，为了保证在我们前例中的名为 Snow 的学生出现在结果中，可以在连接结果中加入一个元

组，它在来自 *student* 关系的所有属性上的值被设置为学生 Snow 的相应值，在所有余下的来自 *takes* 关系属性上的值被设为 *null*，这些属性是 *course_id*、*sec_id*、*semester* 和 *year*。

实际上有三种形式的外连接：

- **左外连接**(left outer join)只保留出现在**左外连接**运算之前(左边)的关系中的元组。
- **右外连接**(right outer join)只保留出现在**右外连接**运算之后(右边)的关系中的元组。
- **全外连接**(full outer join)保留出现在两个关系中的元组。

115
116

相比而言，为了与外连接运算相区分，我们此前学习的不保留未匹配元组的连接运算被称作**内连接**(inner join)运算。

我们现在详细解释每种形式的外连接是怎样操作的。我们可以按照如下方式计算左外连接运算：首先，像前面那样计算出内连接的结果；然后，对于在内连接的左侧关系中任意一个与右侧关系中任何元组都不匹配的元组 t，向连接结果中加入一个元组 r，r 的构造如下：

- 元组 r 从左侧关系得到的属性被赋为 t 中的值。
- r 的其他属性被赋为空值。

图 4-4 给出了下面查询的结果：

```
select *
from student natural left outer join takes;
```

与内连接的结果不同，此结果中包含了学生 Snow(*ID* 70557)，但是在 Snow 对应的元组中，在那些只出现在 *takes* 关系模式中的属性上取空值。

ID	name	dept_name	tot_cred	course_id	sec_id	semester	year	grade
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2010	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2009	A
19991	Brandt	History	80	HIS-351	1	Spring	2010	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2010	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B+
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A-
70557	Snow	Physics	0	*null*	*null*	*null*	*null*	*null*
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	*null*

图 4-4　*student* **natural left outer join** *takes* 的结果

作为使用外连接运算的另一个例子，我们可以写出查询“找出所有一门课程也没有选修的学生”：

```
select ID
from student natural left outer join takes
where course_id is null;
```

右外连接和**左外连接**是对称的。来自右侧关系中的不匹配左侧关系任何元组的元组被补上空值，并加入到右外连接的结果中。这样，如果我们使用右外连接来重写前面的查询，并交换列出关系的次序，如下所示：

```
select *
from takes natural right outer join student;
```

我们得到的结果是一样的，只不过结果中属性出现的顺序不同(见图 4-5)。

ID	course_id	sec_id	semester	year	grade	name	dept_name	tot_cred
00128	CS-101	1	Fall	2009	A	Zhang	Comp. Sci.	102
00128	CS-347	1	Fall	2009	A-	Zhang	Comp. Sci.	102
12345	CS-101	1	Fall	2009	C	Shankar	Comp. Sci.	32
12345	CS-190	2	Spring	2009	A	Shankar	Comp. Sci.	32
12345	CS-315	1	Spring	2010	A	Shankar	Comp. Sci.	32
12345	CS-347	1	Fall	2009	A	Shankar	Comp. Sci.	32
19991	HIS-351	1	Spring	2010	B	Brandt	History	80
23121	FIN-201	1	Spring	2010	C+	Chavez	Finance	110
44553	PHY-101	1	Fall	2009	B-	Peltier	Physics	56
45678	CS-101	1	Fall	2009	F	Levy	Physics	46
45678	CS-101	1	Spring	2010	B+	Levy	Physics	46
45678	CS-319	1	Spring	2010	B	Levy	Physics	46
54321	CS-101	1	Fall	2009	A-	Williams	Comp. Sci.	54
54321	CS-190	2	Spring	2009	B+	Williams	Comp. Sci.	54
55739	MU-199	1	Spring	2010	A-	Sanchez	Music	38
70557	*null*	*null*	*null*	*null*	*null*	Snow	Physics	0
76543	CS-101	1	Fall	2009	A	Brown	Comp. Sci.	58
76543	CS-319	2	Spring	2010	A	Brown	Comp. Sci.	58
76653	EE-181	1	Spring	2009	C	Aoi	Elec. Eng.	60
98765	CS-101	1	Fall	2009	C-	Bourikas	Elec. Eng.	98
98765	CS-315	1	Spring	2010	B	Bourikas	Elec. Eng.	98
98988	BIO-101	1	Summer	2009	A	Tanaka	Biology	120
98988	BIO-301	1	Summer	2010	*null*	Tanaka	Biology	120

图 4-5 *takes* **natural right outer join** *student* 的结果

全外连接是左外连接与右外连接类型的组合。在内连接结果计算出来之后，左侧关系中不匹配右侧关系任何元组的元组被添上空值并加到结果中。类似地，右侧关系中不匹配左侧关系任何元组的元组也被添上空值并加到结果中。

作为使用全外连接的例子，考虑查询："显示 Comp. Sci. 系所有学生以及他们在 2009 年春季选修的所有课程段的列表。2009 年春季开设的所有课程段都必须显示，即使没有 Comp. Sci. 系的学生选修这些课程段"。此查询可写为：

```
select *
from (select *
      from student
      where dept_name = 'Comp. Sci')
  natural full outer join
  (select *
   from takes
   where semester = 'Spring' and year = 2009);
```

on 子句可以和外连接一起使用。下述查询与我们见过的第一个使用"*student* **natural left outer join** *takes*"的查询是相同的，只不过属性 *ID* 在结果中出现两次。

```
select *
from student left outer join takes on student.ID = takes.ID;
```

正如我们前面提到的，**on** 和 **where** 在外连接中的表现是不同的。其原因是外连接只为那些对相应内连接结果没有贡献的元组补上空值并加入结果。**on** 条件是外连接声明的一部分，但 **where** 子句却不是。在我们的例子中，ID 为 70557 的学生"Snow"所对应的 *student* 元组的情况就说明了这样的差异。假设我们把前述查询中的 **on** 子句谓词换成 **where** 子句，并使用 **on** 条件 *true*：

```
select *
from student left outer join takes on true
where student.ID = takes.ID;
```

117 ~ 119

早先的查询使用带 **on** 条件的左外连接，包括元组(70557, Snow, Physics, 0, *null*, *null*, *null*, *null*, *null*)，因为在 *takes* 中没有 *ID* = 70557 的元组。然而在后面的查询中，每个元组都满足连接条件 *true*，因此外连接不会产生出补上空值的元组。外连接实际上产生了两个关系的笛卡儿积。因为在 *takes* 中没有 *ID* = 70557 的元组，每次当外连接中出现 *name* = "Snow"的元组时，*student.ID* 与 *takes.ID* 的取

值必然是不同的，这样的元组会被 **where** 子句谓词排除掉。从而学生 Snow 不会出现在后面查询的结果中。

4.1.3 连接类型和条件

为了把常规连接和外连接区分开来，SQL 中把常规连接称作**内连接**。这样连接子句就可以用 **inner join** 来替换 **outer join**，说明使用的是常规连接。然而关键词 **inner** 是可选的，当 **join** 子句中没有使用 **outer** 前缀，默认的连接类型是 **inner join**。从而，

```
select *
from student join takes using (ID);
```

等价于：

```
select *
from student inner join takes using (ID);
```

类似地，**natural join** 等价于 **natural inner join**。

图 4-6 给出了我们所讨论的所有连接类型的列表。从图中可以看出，任意的连接形式（内连接、左外连接、右外连接或全外连接）可以和任意的连接条件（自然连接、**using** 条件连接或 **on** 条件连接）进行组合。

连接类型
inner join
left outer join
right outer join
full outer join

连接条件
natural
on <predicate>
using (A_1, A_2, ..., A_n)

图 4-6 连接类型和连接条件

4.2 视图

在上面的所有例子中，我们一直都在逻辑模型层操作，即我们假定了给定的集合中的关系都是实际存储在数据库中的。 120

让所有用户都看到整个逻辑模型是不合适的。出于安全考虑，可能需要向用户隐藏特定的数据。考虑一个职员需要知道教师的标识、姓名和所在系名，但是没有权限看到教师的工资值。此人应该看到的关系由如下 SQL 语句所描述：

```
select ID, name, dept_name
from instructor;
```

除了安全考虑，我们还可能希望创建一个比逻辑模型更符合特定用户直觉的个人化的关系集合。我们可能希望有一个关于 Physics 系在 2009 年秋季学期所开设的所有课程段的列表，其中包括每个课程段在哪栋建筑的哪个房间授课的信息。为了得到这样的列表，我们需要创建的关系是：

```
select course.course_id, sec_id, building, room_number
from course, section
where course.course_id = section.course_id
  and course.dept_name = 'Physics'
  and section.semester = 'Fall'
  and section.year = '2009';
```

我们可以计算出上述查询的结果并存储下来，然后把存储关系提供给用户。但如果这样做的话，一旦 *instructor*、*course* 或 *section* 关系中的底层数据发生变化，那么所存储的查询结果就不再与在这些关系上重新执行查询的结果匹配。一般说来，对像上例那样的查询结果进行计算并存储不是一种好的方式（尽管也存在某些例外情况，我们会在后面讨论）。

相反，SQL 允许通过查询来定义“**虚关系**”，它在概念上包含查询的结果。虚关系并不预先计算并存储，而是在使用虚关系的时候才通过执行查询被计算出来。

任何像这种不是逻辑模型的一部分，但作为虚关系对用户可见的关系称为**视图**（view）。在任何给

定的实际关系集合上能够支持大量视图。

4.2.1 视图定义

我们在 SQL 中用 **create view** 命令定义视图。为了定义视图，我们必须给视图一个名称，并且必须提供计算视图的查询。**create view** 命令的格式为：

121

```
create view v as < query expression > ;
```

其中 < query expression > 可以是任何合法的查询表达式，v 表示视图名。

重新考虑需要访问 *instructor* 关系中除 *salary* 之外的所有数据的职员。这样的职员不应该授予访问 *instructor* 关系的权限(我们将在后面 4.6 节介绍如何进行授权)。相反，可以把视图关系 *faculty* 提供给职员，此视图的定义如下：

```
create view faculty as
select ID, name, dept_name
from instructor;
```

正如前面已经解释过的，视图关系在概念上包含查询结果中的元组，但并不进行预计算和存储。相反，数据库系统存储与视图关系相关联的查询表达式。当视图关系被访问时，其中的元组是通过计算查询结果而被创建出来的。从而，视图关系是在需要的时候才被创建的。

为了创建一个视图，列出 Physics 系在 2009 年秋季学期所开设的所有课程段，以及每个课程段在哪栋建筑的哪个房间授课的信息，我们可以写出：

```
create view physics_fall_2009 as
  select course.course_id, sec_id, building, room_number
  from course, section
  where course.course_id = section.course_id
      and course.dept_name = 'Physics'
      and section.semester = 'Fall'
      and section.year = '2009';
```

4.2.2 SQL 查询中使用视图

一旦定义了一个视图，我们就可以用视图名指代该视图生成的虚关系。使用视图 *physics_fall_2009*，我们可以用下面的查询找到所有于 2009 年秋季学期在 Watson 大楼开设的 Physics 课程：

```
select course_id
from physics_fall_2009
where building = 'Watson';
```

在查询中，视图名可以出现在关系名可以出现的任何地方。

视图的属性名可以按下述方式显式指定：

```
create view departments_total_salary(dept_name, total_salary) as
  select dept_name, sum (salary)
  from instructor
  group by dept_name;
```

122 上述视图给出了每个系中所有教师的工资总和。因为表达式 **sum**(*salary*)没有名称，其属性名是在视图定义中显式指定的。

直觉上，在任何给定时刻，视图关系中的元组集是该时刻视图定义中的查询表达式的计算结果。因此，如果一个视图关系被计算并存储，一旦用于定义该视图的关系被修改，视图就会过期。为了避免这一点，视图通常这样来实现：当我们定义一个视图时，数据库系统存储视图的定义本身，而不存储定义该视图的查询表达式的执行结果。一旦视图关系出现在查询中，它就被已存储的查询表达式代替。因此，无论我们何时执行这个查询，视图关系都被重新计算。

一个视图可能被用到定义另一个视图的表达式中。例如，我们可以如下定义视图 *physics_fall_2009_watson*，它列出了于 2009 年秋季学期在 Watson 大楼开设的所有 Physics 课程的标识和房间号：

```
create view physics_fall_2009_watson as
    select course_id, room_number
    from physics_fall_2009
    where building = 'Watson';
```

其中 *physics_fall_2009_watson* 本身是一个视图关系，它等价于：

```
create view physics_fall_2009_watson as
  (select course_id, room_number
   from (select course.course_id, building, room_number
         from course, section
         where course.course_id = section.course_id
         and course.dept_name = 'Physics'
         and section.semester = 'Fall'
         and section.year = '2009')
   where building = 'Watson';
```

4.2.3 物化视图

特定数据库系统允许存储视图关系，但是它们保证：如果用于定义视图的实际关系改变，视图也跟着修改。这样的视图被称为**物化视图**(materialized view)。

例如，考察视图 *departments_total_salary*。如果上述视图是物化的，它的结果就会存放在数据库中。然而，如果一个 *instructor* 元组被插入到 *instructor* 关系中，或者从 *instructor* 关系中删除，定义视图的查询结果就会变化，其结果是物化视图的内容也必须更新。类似地，如果一位教师的工资被更新，那么 *departments_total_salary* 中对应于该教师所在系的元组必须更新。 123

保持物化视图一直在最新状态的过程称为**物化视图维护**(materialized view maintenance)，或者通常简称**视图维护**(view maintenance)，这将在 13.5 节进行介绍。当构成视图定义的任何关系被更新时，可以马上进行视图维护。然而某些数据库系统在视图被访问时才执行视图维护。还有一些系统仅采用周期性的物化视图更新方式，在这种情况下，当物化视图被使用时，其中的内容可能是陈旧的，或者说过时的。如果应用需要最新数据的话，这种方式是不适用的。某些数据库系统允许数据库管理员来控制在每个物化视图上需要采取上述的哪种方式。

频繁使用视图的应用将会从视图的物化中获益。那些需要快速响应基于大关系上聚集计算的特定查询也会从创建与查询相对应的物化视图中受益良多。在这种情况下，聚集结果很可能比定义视图的大关系要小得多，其结果是利用物化视图来回答查询就很快，它避免了读取大的底层关系。当然，物化视图查询所带来的好处还需要与存储代价和增加的更新开销相权衡。

SQL 没有定义指定物化视图的标准方式，但是很多数据库系统提供了各自的 SQL 扩展来实现这项任务。一些数据库系统在底层关系变化时，总是把物化视图保持在最新状态；也有另外一些系统允许物化视图过时，但周期性地重新计算物化视图。

4.2.4 视图更新

尽管对查询而言，视图是一个有用的工具，但如果我们用它们来表达更新、插入或删除，它们可能带来严重的问题。困难在于，用视图表达的数据库修改必须被翻译为对数据库逻辑模型中实际关系的修改。

假设我们此前所见的视图 *faculty* 被提供给一个职员。既然我们允许视图名出现在任何关系名可以出现的地方，该职员可以这样写出：

```
insert into faculty
    values ('30765', 'Green', 'Music');
```

这个插入必须表示为对 *instructor* 关系的插入，因为 *instructor* 是数据库系统用于构造视图 *faculty* 的实际关系。然而，为了把一个元组插入到 *instructor* 中，我们必须给出 *salary* 的值。存在两种合理的解决方法来处理该插入：

- 拒绝插入，并向用户返回一个错误信息。
- 向 *instructor* 关系插入元组('30765', 'Green', 'Music', null)。 124

通过视图修改数据库的另一类问题发生在这样的视图上：

```
create view instructor_info as
select ID, name, building
from instructor, department
where instructor.dept_name = department.dept_name;
```

这个视图列出了大学里每个教师的 *ID*、*name* 和建筑名。考虑如下通过该视图的插入：

```
insert into instructor_info
    values ('69987', 'White', 'Taylor');
```

假设没有标识为 69987 的教师，也没有位于 Taylor 大楼的系。那么向 *instructor* 和 *department* 关系中插入元组的唯一可能的方法是：向 *instructor* 中插入元组 ('69987', 'White', *null*, *null*)，并向 *department* 中插入元组 (*null*, 'Taylor', *null*)。于是我们得到如图 4-7 所示的关系。但是这个更新并没有产生出所需的结果，因为视图关系 *instructor_info* 中仍然不包含元组 ('69987', 'White', 'Taylor')。因此，通过利用空值来更新 *instructor* 和 *department* 关系以得到对 *instructor_info* 所需的更新是不可行的。

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
69987	White	*null*	*null*

instructor

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Electrical Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000
null	Taylor	*null*

department

图 4-7 插入元组后的 *instructor* 和 *department* 关系

由于如上所述的种种问题，除了一些有限的情况之外，一般不允许对视图关系进行修改。不同的数据库系统指定了不同的条件以允许更新视图关系；请参考数据库系统手册以获得详细信息。通过视图进行数据库修改的通用问题已经成为重要的研究课题，文献注解中引用了一些这方面的研究。

一般说来，如果定义视图的查询对下列条件都能满足，我们称 SQL 视图是**可更新的**(updatable)(即视图上可以执行插入、更新或删除)：

- **from** 子句中只有一个数据库关系。
- **select** 子句中只包含关系的属性名，不包含任何表达式、聚集或 **distinct** 声明。
- 任何没有出现在 **select** 子句中的属性可以取空值；即这些属性上没有 **not null** 约束，也不构成主码的一部分。
- 查询中不含有 **group by** 或 **having** 子句。

在这些限制下，下面的视图上允许执行 **update**、**insert** 和 **delete** 操作：

```
create view history_instructors as
select *
from instructor
where dept_name = 'History';
```

即便是在可更新的情况下，下面这些问题仍然存在。假设一个用户尝试向视图 *history_instructors* 中插入元组('25566', 'Brown', 'Biology', 100000)，这个元组可以被插入到 *instructor* 关系中，但是由于它不满足视图所要求的选择条件，它不会出现在视图 *history_instructors* 中。

在默认情况下，SQL 允许执行上述更新。但是，可以通过在视图定义的末尾包含 **with check option** 子句的方式来定义视图。这样，如果向视图中插入一条不满足视图的 **where** 子句条件的元组，数据库系统将拒绝该插入操作。类似地，如果新值不满足 **where** 子句的条件，更新也会被拒绝。

125 | 126 SQL:1999 对于何时可以在视图上执行插入、更新和删除有更复杂的规则集，该规则集允许通过一类更大视图进行更新，但是这些规则过于复杂，我们就不在这里讨论了。

4.3 事务

事务(transaction)由查询和(或)更新语句的序列组成。SQL标准规定当一条SQL语句被执行，就隐式地开始了一个事务。下列SQL语句之一会结束一个事务：

- **Commit work**：提交当前事务，也就是将该事务所做的更新在数据库中持久保存。在事务被提交后，一个新的事务自动开始。
- **Rollback work**：回滚当前事务，即撤销该事务中所有SQL语句对数据库的更新。这样，数据库就恢复到执行该事务第一条语句之前的状态。

关键词 **work** 在两条语句中都是可选的。

当在事务执行过程中检测到错误时，事务回滚是有用的。在某种意义上，事务提交就像对编辑文档的变化存盘，而回滚就像不保存变化退出编辑。一旦某事务执行了 **commit work**，它的影响就不能用 **rollback work** 来撤销了。数据库系统保证在发生诸如某条SQL语句错误、断电、系统崩溃这些故障的情况下，如果一个事务还没有完成 **commit work**，其影响将被回滚。在断电和系统崩溃的情况下，回滚会在系统重启后执行。

例如，考虑一个银行应用，我们需要从一个银行账户上把钱转到同一家银行的另一个账户。为了这样做，我们需要更新两个账户的余额，把需要转移的资金额从一个账户划走，并把它加到另一个账户上。如果在从第一个账户上划走资金以后，但在把这笔资金加入第二个账户之前发生了系统崩溃，那么银行账户就会不一致。如果在第一个账户划走资金之前先往第二个账户存款，并且在存款之后马上发生系统崩溃，那么也会出现类似的问题。

作为另一个例子，考虑我们正在使用的大学应用的例子。我们假设 *student* 关系中每个元组在 *tot_cred* 属性上的取值需要保持在最新状态，只要学生成功修完一门课程，该属性值就要更新。为了这样做，只要 *takes* 关系被更新为记录一位学生成功修完一门课程的信息(通过赋予适当的成绩)，相应的 *student* 元组也必须更新。如果执行这两个更新的任务在执行完一个更新后，但在第二个更新前崩溃了，数据库中的数据就是不一致的。

一个事务或者在完成所有步骤后提交其行为，或者在不能成功完成其所有动作的情况下回滚其所有动作，通过这种方式数据库提供了对事务具有**原子性**(atomic)的抽象，原子性也就是不可分割性。要么事务的所有影响被反映到数据库中，要么任何影响都没有(在回滚之后)。 127

如果把事务的概念应用到上述应用中，那些更新语句就会作为单个事务执行。在事务执行其某一条语句时出错会导致事务早先执行的语句的影响被撤销，从而不会让数据库处于部分更新状态。

如果程序没有执行两条命令中的任何一条而终止了，那么更新要么被提交要么被回滚。SQL标准并没有指出究竟执行哪一种，如何选择依赖于具体实现。

在很多SQL实现中，默认方式下每个SQL语句自成一个事务，且一执行完就提交。如果一个事务要执行多条SQL语句，就必须关闭单独SQL语句的自动提交。如何关闭自动提交也依赖于特定的SQL实现，尽管在诸如JDBC或ODBC那样的应用编程接口中存在标准化方式来完成这项工作。我们将在5.1.1和5.1.2节分别学习JDBC和ODBC。

一个较好的选择是，作为SQL:1999标准的一部分(但目前只有一些SQL实现支持)，允许多条SQL语句包含在关键字 **begin atomic…end** 之间。所有在关键字之间的语句构成了一个单一事务。

我们将在第14章学习事务的更多特性；第15章和第16章介绍在单个数据库中实现事务的相关问题，而在第19章介绍跨多个数据库上实现事务的相关问题，这是为了处理不同银行的账户之间转账的问题，不同银行有不同的数据库。

4.4 完整性约束

完整性约束保证授权用户对数据库所做的修改不会破坏数据的一致性。因此，完整性约束防止的是对数据的意外破坏。

完整性约束的例子有：

- 教师姓名不能为 *null*。
- 任意两位教师不能有相同的教师标识。
- *course* 关系中的每个系名必须在 *department* 关系中有一个对应的系名。
- 一个系的预算必须大于 0.00 美元。

一般说来，一个完整性约束可以是属于数据库的任意谓词。但检测任意谓词的代价可能太高。因此，大多数数据库系统允许用户指定那些只需极小开销就可以检测的完整性约束。

在 3.2.2 节我们已经见过了一些完整性约束的形式。本节我们将学习更多的完整性约束形式。在
128 第 8 章我们学习另一种被称作**函数依赖**的完整性约束形式，它主要应用在模式设计的过程中。

完整性约束通常被看成是数据库模式设计过程的一部分，它作为用于创建关系的 **create table** 命令的一部分被声明。然而，完整性约束也可以通过使用 **alter table** *table-name* **add** *constraint* 命令施加到已有关系上，其中 *constraint* 可以是关系上的任意约束。当执行上述命令时，系统首先保证关系满足指定的约束。如果满足，那么约束被施加到关系上；如果不满足，则拒绝执行上述命令。

4.4.1 单个关系上的约束

我们在 3.2 节中描述了如何用 **create table** 命令定义关系表。**create table** 命令还可以包括完整性约束语句。除了主码约束之外，还有许多其他可以包括在 **create table** 命令中的约束。允许的完整性约束包括：

- **not null**。
- **unique**。
- **check**(< 谓词 >)。

我们将在下面的几节中讨论上述每一种约束。

4.4.2 not null 约束

正如我们在第 3 章中讨论过的，空值是所有域的成员，因此在默认情况下是 SQL 中每个属性的合法值。然而对于一些属性来说，空值可能是不合适的。考虑 *student* 关系中的一个元组，其中 *name* 是 *null*。这样的元组给出了一个未知学生的学生信息；因此它不含有有用的信息。类似地，我们不会希望系的预算为 *null*。在这些情况下，我们希望禁止空值，我们可以通过如下声明来通过限定属性 *name* 和 *budget* 的域来排除空值：

name **varchar**(20) **not null**
budget **numeric**(12, 2) **not null**

not null 声明禁止在该属性上插入空值。任何可能导致向一个声明为 **not null** 的属性插入空值的数据库修改都会产生错误诊断信息。

许多情况下我们希望避免空值。尤其是 SQL 禁止在关系模式的主码中出现空值。因此，在我们的
129 大学例子中，在 *department* 关系上如果声明属性 *dept_name* 为 *department* 的主码，那它就不能为空。因此它不必显式地声明为 **not null**。

4.4.3 unique 约束

SQL 还支持下面这种完整性约束：

unique ($A_{j1}, A_{j2}, \cdots, A_{jm}$)

unique 声明指出属性 $A_{j1}, A_{j2}, \cdots, A_{jm}$ 形成了一个候选码；即在关系中没有两个元组能在所有列出的属性上取值相同。然而候选码属性可以为 *null*，除非它们已被显式地声明为 **not null**。回忆一下，空值不等于其他的任何值。(这里对空值的处理与 3.8.4 节中定义的 **unique** 结构一样。)

4.4.4 check 子句

当应用于关系声明时，**check**(P)子句指定一个谓词 P，关系中的每个元组都必须满足谓词 P。

通常用 **check** 子句来保证属性值满足指定的条件，实际上创建了一个强大的类型系统。例如，在创建关系 *department* 的 **create table** 命令中的 **check**(*budget* > 0)子句将保证 **budget** 上的取值是正数。

作为另一个例子，考虑如下语句：

```
create table section
  (course_id varchar (8),
  sec_id varchar (8),
  semester varchar (6),
  year numeric (4, 0),
  building varchar (15),
  room_number varchar (7),
  time_slot_id varchar (4),
  primary key (course_id, sec_id, semester, year),
  check (semester in ('Fall', 'Winter', 'Spring', 'Summer')));
```

这里我们用 **check** 子句模拟了一个枚举类型，通过指定 *semester* 必须是'Fall'、'Winter'、'Spring'或'Summer'中的一个来实现。这样，**check** 子句允许以有力的方式对属性域加以限制，这是大多数编程语言的类型系统都不能允许的。

根据 SQL 标准，**check** 子句中的谓词可以是包括子查询在内的任意谓词。然而，当前还没有一个广泛使用的数据库产品允许包含子查询的谓词。 130

4.4.5 参照完整性

我们常常希望保证在一个关系中给定属性集上的取值也在另一关系的特定属性集的取值中出现。这种情况称为**参照完整性**(referential integrity)。

正如我们此前在 3.2.2 节所见，外码可以用作为 SQL 中 **create table** 语句一部分的 **foreign key** 子句来声明。我们用大学数据库 SQL DLL 定义的一部分来说明外码声明，如图 4-8 所示。*course* 表的定义中有一个声明"**foreign key**(*dept_name*) **references** *department*"。这个外码声明表示，在每个课程元组中指定的系名必须在 *department* 关系中存在。没有这个约束，就可能会为一门课程指定一个不存在的系名。

更一般地，令关系 r_1 和 r_2 的属性集分别为 R_1 和 R_2，主码分别为 K_1 和 K_2。如果要求对 r_2 中任意元组 t_2，均存在 r_1 中元组 t_1 使得 $t_1.K_1 = t_2.\alpha$，我们称 R_2 的子集 α 为参照关系 r_1 中 K_1 的**外码**(foreign key)。

这种要求称为**参照完整性约束**(referential-intergrity constraint)或**子集依赖**(subset dependency)。后一种称法是由于上述参照完整性可以表示为这样一种要求：r_2 中 α 上的取值集合必须是 r_1 中 K_1 上的取值集合的子集。请注意，为使参照完整性约束有意义，α 和 K_1 必须是相容的属性集；也就是说，要么 α 等于 K_1，要么它们必须包含相同数目的属性，并且对应属性的类型必须相容(这里我们假设 α 和 K_1 是有序的)。不同于外码约束，参照完整性约束通常不要求 K_1 是 r_1 的主码；其结果是，r_1 中可能有不止一个元组在属性 K_1 上取值相同。

默认情况下，SQL 中外码参照的是被参照表中的主码属性。SQL 还支持一个可以显式指定被参照关系的属性列表的 **references** 子句。然而，这个指定的属性列表必须声明为被参照关系的候选码，要么使用 **primary key** 约束，要么使用 **unique** 约束。在更为普遍的参照完整性约束形式中，被参照的属性不必是候选码，这样的形式还不能在 SQL 中直接声明。SQL 标准提供了另外的结构用于实现这样的约束，4.4.7 节将描述这样的结构。

我们可以使用如下的简写形式作为属性定义的一部分，并声明该属性为外码：

dept_name **varchar**(20) **references** *department*

当违反参照完整性约束时，通常的处理是拒绝执行导致完整性破坏的操作(即进行更新操作的事务被回滚)。但是，在 **foreign key** 子句中可以指明：如果被参照关系上的删除或更新动作违反了约束，那么系统必须采取一些步骤通过修改参照关系中的元组来恢复完整性约束，而不是拒绝这样的动作。 131

考虑在关系 *course* 上的如下完整性约束定义：

```
create table classroom
        (building varchar (15),
        room_number varchar (7),
        capacity numeric (4, 0),
        primary key (building, room_number))
    create table department
        (dept_name varchar (20),
        building varchar (15),
        budget numeric (12, 2) check (budget > 0),
        primary key (dept_name))
    create table course
        (course_id varchar (8),
        title varchar (50),
        dept_name varchar (20),
        credits numeric (2, 0) check (credits > 0),
        primary key (course_id),
        foreign key (dept_name) references department)
    create table instructor
        (ID varchar (5),
        name varchar (20) not null
        dept_name varchar (20),
        salary numeric (8, 2), check (salary > 29000),
        primary key (ID),
        foreign key (dept_name) references department)
    create table section
        (course_id varchar (8),
        sec_id varchar (8),
        semester varchar (6), check (semester in
            ('Fall', 'Winter', 'Spring', 'Summer')),
        year numeric (4, 0), check (year > 1759 and year < 2100)
        building varchar (15),
        room_number varchar (7),
        time_slot_id varchar (4),
        primary key (course_id, sec_id, semester, year),
        foreign key (course_id) references course,
        foreign key (building, room_number) references classroom)
```

图 4-8 大学数据库的部分 SQL 数据定义

```
create table course
    ( ···
    foreign key (dept_name) references department
        on delete cascade
        on update cascade,
    ···);
```

由于有了与外码声明相关联的 **on delete cascade** 子句，如果删除 *department* 中的元组导致了此参照完整性约束被违反，则删除并不被系统拒绝，而是对 *course* 关系作“级联”删除，即删除参照了被删除系的元组。类似地，如果更新被参照字段时违反了约束，则更新操作并不被系统拒绝，而是将 *course* 中参照的元组的 *dept_name* 字段也改为新值。SQL 还允许 **foreign key** 子句指明除 **cascade** 以外的其他动作，如果约束被违反：可将参照域(这里是 *dept_name*)置为 *null*(用 **set null** 代替 **cascade**)，或者置为域的默认值(用 **set default**)。

如果存在涉及多个关系的外码依赖链，则在链一端所做的删除或更新可能传至整个链。实践习题 4.9 中的一个有趣的情况是，在一个关系上定义的 **foreign key** 约束所参照的关系就是它自己。如果一个级联更新或删除导致的对约束的违反不能通过进一步的级联操作解决，则系统中止该事务。于是，该事务所做的所有改变及级联动作将被撤销。

空值使得SQL中参照约束的语义复杂化了。外码中的属性允许为 *null*，只要它们没有被声明为 **not null**。如果给定元组中外码的所有列上均取非空值，则对该元组采用外码约束的通常定义。如果某外码列为 *null*，则该元组自动被认为满足约束。

这样的规定有时不一定是正确的选择，因此SQL也提供一些结构使你可以改变对空值的处理；我们在此不讨论这样的结构。

4.4.6 事务中对完整性约束的违反

事务可能包括几个步骤，在某一步之后完整性约束也许会暂时被违反，但是后面的某一步也许就会消除这个违反。例如，假设我们有一个主码为 *name* 的 *person* 关系，还有一个属性是 *spouse*，并且 *spouse* 是在 *person* 上的一个外码。也就是说，约束要求 *spouse* 属性必须包含在 *person* 表里出现的名字。假设我们希望在上述关系中插入两个元组，一个是关于John的，另一个是关于Mary的，这两个元组的配偶属性分别设置为Mary和John，以此表示John和Mary彼此之间的婚姻关系。无论先插入哪个元组，插入第一个元组的时候都会违反外码约束。在插入第二个元组后，外码约束又会满足了。

132
|
133

为了处理这样的情况，SQL标准允许将 **initially deferred** 子句加入到约束声明中；这样完整性约束不是在事务的中间步骤上检查，而是在事务结束的时候检查。一个约束可以被指定为**可延迟的**(deferrable)，这意味着默认情况下它会被立即检查，但是在需要的时候可以延迟检查。对于声明为可延迟的约束，执行 **set constraints** *constraint-list* **deferred** 语句作为事务的一部分，会导致对指定约束的检查被延迟到该事务结束时执行。

然而，读者应该注意的是默认的方式是立即检查约束，而且许多数据库实现不支持延迟约束检查。

如果 *spouse* 属性可以被赋为 *null*，我们可以用另一种方式来避开在上面例子中的问题：在插入John和Mary元组时，我们设置其 *spouse* 属性为 *null*，然后再更新它们的值。然而，这个技术需要更大的编程量，而且如果属性不能设为 *null* 的话，此方法就不可行。

4.4.7 复杂 check 条件与断言

本节描述SQL标准所支持的另外一些用于声明完整性约束的结构。然而，读者应该注意的是，这些结构目前还没有被大多数数据库系统支持。

正如SQL标准所定义的，**check** 子句中的谓词可以是包含子查询的任意谓词。如果一个数据库实现支持在 **check** 子句中出现子查询，我们就可以在关系 *section* 上声明如下所示的参照完整性约束：

check (*time_slot_id* **in** (**select** *time_slot_id* **from** *time_slot*))

这个 **check** 条件检测在 *section* 关系中每个元组的 *time_slot_id* 的确是在 *time_slot* 关系中某个时间段的标识。因此这个条件不仅在 *section* 中插入或修改元组时需要检测，而且在 *time_slot* 关系改变时也需要检测(如在 *time_slot* 关系中，当一个元组被删除或修改的情况下)。

在我们的大学模式上，另一个自然的约束是：每个课程段都需要有至少一位教师来讲授。为了强制实现此约束，一种方案是声明 *section* 关系的属性集(*course_id*, *sec_id*, *semester*, *year*)为外码，它参照 *teaches* 关系中的相应属性。遗憾的是，这些属性并未构成 *teaches* 关系的主码。如果数据库系统支持在check约束中出现子查询的话，可以使用与 *time_slot* 属性类似的check约束来强制实现上述约束。

复杂 **check** 条件在我们希望确保数据完整性的时候是很有用的，但其检测开销可能会很大。例如，**check** 子句中的谓词不仅需要在 *section* 关系发生更新时计算，而且也可能在 *time_slot* 关系发生更新时检测，因为 *time_slot* 在子查询中被引用了。

134

一个**断言**(assertion)就是一个谓词，它表达了我们希望数据库总能满足的一个条件。域约束和参照完整性约束是断言的特殊形式。我们前面用大量篇幅介绍了这几种形式的断言，是因为它们容易检测并且适用于很多数据库应用。但是，还有许多约束不能仅用这几种特殊形式来表达。如有两个这样的例子：

- 对于 *student* 关系中的每个元组，它在属性 *tot_cred* 上的取值必须等于该生所成功修完课程的学分总和。

- 每位教师不能在同一个学期的同一个时间段在两个不同的教室授课。[⊖]

SQL 中的断言为如下形式：

create assertion < assertion-name > **check** < predicate > ;

图 4-9 给出了我们如何用 SQL 写出第一个约束的示例。由于 SQL 不提供"for all X, $P(X)$"结构(其中 P 是一个谓词)，我们只好通过等价的"not exists X such that not $P(X)$"结构来实现此约束，这一结构可以用 SQL 来表示。

```
create assertion credits_earned_constraint check
    (not exists (select ID
      from student
      where tot_cred < > (select sum(credits)
                          from takes natural join course
                          where student. ID = takes. ID
                          and grade is not null and grade < > 'F' ));
```

图 4-9 一个断言的例子

我们把第二个约束的声明留作练习。

当创建断言时，系统要检测其有效性。如果断言有效，则今后只有不破坏断言的数据库修改才被允许。如果断言较复杂，则检测会带来相当大的开销。因此，使用断言应该特别小心。由于检测和维护断言的开销较大，一些系统开发者省去了对一般性断言的支持，或者只提供易于检测的特殊形式的断言。

目前，还没有一个广泛使用的数据库系统支持在 **check** 子句的谓词中使用子查询或 **create assertion** 结构。然而，如果数据库系统支持触发器的话，可以通过使用触发器来实现等价的功能，触发器将在 5.3 节介绍，5.3 节还将介绍如何用触发器来实现 *time_slot_id* 上的参照完整性约束。

4.5 SQL 的数据类型与模式

在第 3 章中，我们介绍了一些 SQL 支持的固有数据类型，如整数类型、实数类型和字符类型。SQL 还支持一些其他的固有数据类型，我们将在下面描述。我们还将描述如何在 SQL 中创建基本的用户定义类型。

4.5.1 SQL 中的日期和时间类型

除了我们在 3.2 节介绍过的基本数据类型以外，SQL 标准还支持与日期和时间相关的几种数据类型：

- **date**：日历日期，包括年(四位)、月和日。
- **time**：一天中的时间，包括小时、分和秒。可以用变量 **time**(p)来表示秒的小数点后的数字位数(这里默认值为 0)。通过指定 **time with timezone**，还可以把时区信息连同时间一起存储。
- **timestamp**：**date** 和 **time** 的组合。可以用变量 **timestamp**(p)来表示秒的小数点后的数字位数(这里默认值为 6)。如果指定 **with timezone**，则时区信息也会被存储。

日期和时间类型的值可按如下方式说明：

date '2001 - 04 - 25'
time '09: 30: 00'
timestamp '2001 - 04 - 25 10: 29: 01. 45'

日期类型必须按照如上年月日的格式顺序指定。**time** 和 **timestamp** 的秒部分可能会有小数部分，像上述时间戳中的情况一样。

我们可以利用 **cast** e **as** t 形式的表达式来将一个字符串(或字符串表达式)e 转换成类型 t，其中 t 是

⊖ 我们假设不在第二课程采用远程授课的形式！另一个约束是指定"一位教师在一个给定学期的相同时间段不能讲授两门课程"，由于有时候课程是交叉排课的，即对一门课程给出了两个标识和名称，所以可能不满足此约束。

date、**time**、**timestamp** 中的一种。字符串必须符合正确的格式，像本段开头说的那样。当需要时，时区信息可以从系统设置中得到。

我们可以利用 **extract**(*field* **from** *d*)，从 **date** 或 **time** 值 *d* 中提取出单独的域，这里的域可以是 **year**、**month**、**day**、**hour**、**minute** 或者 **second** 中的任意一种。时区信息可以用 **timezone_hour** 和 **timezone_minute** 来提取。

135
/
136

SQL 定义了一些函数以获取当前日期和时间。例如，**current_date** 返回当前日期，**current_time** 返回当前时间(带有时区)，还有 **localtime** 返回当前的本地时间(不带时区)。时间戳(日期加上时间)由 **current_timestamp**(带有时区)以及 **localtimestamp**(本地日期和时间，不带时区)返回。

SQL 允许在上面列出的所有类型上进行比较运算，也允许在各种数字类型上进行算术运算和比较运算。SQL 还支持 **interval** 数据类型，它允许在日期、时间和时间间隔上进行计算。例如，假设 x 和 y 都是 **date** 类型，那么 $x - y$ 就是时间间隔类型，其值为从日期 x 到日期 y 间隔的天数。类似地，在日期或时间上加减一个时间间隔将分别得到新的日期或时间。

4.5.2　默认值

SQL 允许为属性指定默认值，如下面的 **create table** 语句所示：

```
create table student
  (ID varchar (5),
  name varchar (20) not null,
  dept_name varchar (20),
  tot_cred numeric (3, 0) default 0,
  primary key (ID));
```

tot_cred 属性的默认值被声明为 0。这样，当一个元组被插入到 *student* 关系中，如果没有给出 *tot_cred* 属性的值，那么该元组在此属性上的取值就被置为 0。下面的插入语句说明了在插入操作中如何省略 *tot_cred* 属性的值：

```
insert into student(ID, name, dept_name)
  values ('12789', 'Newman', 'Comp. Sci.');
```

4.5.3　创建索引

许多查询只涉及文件中的少量记录。例如，像这样的查询“找出 Physics 系的所有教师”，或“找出 *ID* 为 22201 的学生的 *tot_cred* 值”，只涉及学生记录中的一小部分。如果系统读取每条记录并一一检查其 *ID* 域是否为“22201”，或者 *dept_name* 域是否取值为“Physics”，这样的方式是很低效的。

在关系的属性上所创建的**索引**(index)是一种数据结构，它允许数据库系统高效地找到关系中那些在索引属性上取给定值的元组，而不用扫描关系中的所有元组。例如，如果我们在 *student* 关系的属性 *ID* 上创建了索引，数据库系统不用读取 *student* 关系中的所有元组，就可以直接找到任何像 22201 或 44553 那样具有指定 *ID* 值的记录。索引也可以建立在一个属性列表上，例如在 *student* 的属性 *name* 和 *dept_name* 上。

137

我们将在后面第 11 章学习索引是如何实现的，包括一种被称作 B^+ 树的索引，它是一种特别的、广泛使用的索引类型。

尽管 SQL 语言没有给出创建索引的正式语法定义，但很多数据库都支持使用如下所示的语法形式来创建索引：

```
create index studentID_index on student(ID);
```

上述语句在 *student* 关系的属性 *ID* 上创建了一个名为 *studentID_index* 的索引。

如果用户提交的 SQL 查询可以从索引的使用中获益，那么 SQL 查询处理器就会自动使用索引。例如，给定的 SQL 查询是选出 *ID* 为 22201 的 *student* 元组，SQL 查询处理器就会使用上面定义的 *studentID_index* 索引来找到所需元组，而不用读取整个关系。

4.5.4　大对象类型

许多当前的数据库应用需要存储可能很大(KB 级)的属性，例如一张照片；或者非常大的属性

(MB 级甚至 GB 级)，例如高清晰度的医学图像或视频片断。因此 SQL 提供字符数据的大对象数据类型(**clob**)和二进制数据的大对象数据类型(**blob**)。在这些数据类型中字符"lob"代表"Large OBject"。例如，我们可以声明属性

book_review **clob** (10KB)
image **blob** (10MB)
movie **blob** (2GB)

对于包含大对象(好几个 MB 甚至 GB)的结果元组而言，把整个大对象放入内存中是非常低效和不现实的。相反，一个应用通常用一个 SQL 查询来检索出一个大对象的"定位器"，然后在宿主语言中用这个定位器来操纵对象，应用本身也是用宿主语言书写的。例如，JDBC 应用编程接口(5.1.1 节描述)允许获取一个定位器而不是整个大对象；然后用这个定位器来一点一点地取出这个大对象，而不是一次取出全部，这很像用一个 **read** 函数调用从操作系统文件中读取数据。

4.5.5 用户定义的类型

SQL 支持两种形式的用户定义数据类型。第一种称为**独特类型**(distinct type)，我们将在这里介绍。
138 另一种称为**结构化数据类型**(structured data type)，允许创建具有嵌套记录结构、数组和多重集的复杂数据类型。在本章我们不介绍结构化数据类型，而是在后面第 22 章描述。

一些属性可能会有相同的数据类型。例如，用于学生名和教师名的 *name* 属性就可能有相同的域：所有人名的集合。然而，*budget* 和 *dept_name* 的域肯定应该是不同的。*name* 和 *dept_name* 是否应该有相同的域，这一点就不那么明显了。在实现层，教师姓名和系的名字都是字符串。然而，我们通常不认为"找出所有与某个系同名的教师"是一个有意义的查询。因此，如果我们在概念层而不是物理层来看待数据库的话，*name* 和 *dept_name* 应该有不同的域。

更重要的是，在现实中，把一个教师的姓名赋给一个系名可能是一个程序上的错误；类似地，把一个以美元表示的货币值直接与一个以英镑表示的货币值进行比较几乎可以肯定是程序上的错误。一个好的类型系统应该能够检测出这类赋值或比较。为了支持这种检测，SQL 提供了**独特类型**(distinct type)的概念。

可以用 **create type** 子句来定义新类型。例如，下面的语句：

create type *Dollars* **as numeric** (12, 2) **final**;
create type *Pounds* **as numeric** (12, 2) **final**;

把两个用户定义类型 *Dollars* 和 *Pounds* 定义为总共 12 位数字的十进制数，其中两位放在十进制小数点后。(在此关键字 **final** 并不是真的有意义，它是 SQL:1999 标准要求的，其原因我们不在这里讨论了；一些系统实现允许忽略 **final** 关键字。)然后新创建的类型就可以用作关系属性的类型。例如，我们可以把 *department* 表定义为：

create table *department*
(*dept_name* **varchar** (20),
building **varchar** (15),
budget Dollars);

尝试为 *Pounds* 类型的变量赋予一个 *Dollars* 类型的值会导致一个编译时错误，尽管这两者都是相同的数值类型。这样的赋值很可能是由程序错误引起的，或许是程序员忘记了货币之间的区别。为不同的货币声明不同的类型能帮助发现这些错误。

由于有强类型检查，表达式(*department*. *budget* + 20)将不会被接受，因为属性和整型常数 20 具有不同的类型。一种类型的数值可以被**转换**(也即 *cast*)到另一个域，如下所示：

139 **cast** (*department*. *budget* **to** *numeric*(12, 2))

我们可以在数值类型上做加法，但是为了把结果存回到一个 *Dollars* 类型的属性中，我们需要用另一个类型转换表达式来把数值类型转换回 *Dollars* 类型。

SQL 提供了 **drop type** 和 **alter type** 子句来删除或修改以前创建过的类型。

在把用户定义类型加入到 SQL(在 SQL:1999 中)之前，SQL 有一个相似但稍有不同的概念：域

(domain)(在SQL-92中引入)，它可以在基本类型上施加完整性约束。例如，我们可以定义一个域*DDollars*，如下所示：

```
create domain DDollars as numeric(12, 2) not null;
```

*DDollars*域可以用作属性类型，正如我们用*Dollars*类型一样。然而，类型和域之间有两个重大的差别：

1. 在域上可以声明约束，例如**not null**，也可以为域类型变量定义默认值，然而在用户定义类型上不能声明约束或默认值。设计用户定义类型不仅是用它来指定属性类型，而且还将它用在不能施加约束的地方对SQL进行过程扩展。

2. 域并不是强类型的。因此一个域类型的值可以被赋给另一个域类型，只要它们的基本类型是相容的。

当把**check**子句应用到域上时，允许模式设计者指定一个谓词，被声明为来自该域的任何变量都必须满足这个谓词。例如，**check**子句可以保证教师工资域中只允许出现大于给定值的值：

```
create domain YearlySalary numeric(8, 2)
  constraint salary_value_test check(value >= 29000.00);
```

*YearlySalary*域有一个约束来保证年薪大于或等于29 000.00美元。**constraint** *salary_value_test*子句是可选的，它用来将该约束命名为*salary_value_test*。系统用这个名字来指出一个更新违反了哪个约束。

作为另一个例子，使用**in**子句可以限定一个域只包含指定的一组值：

```
create domain degree_level varchar(10)
  constraint degree_level_test
    check (value in ('Bachelors', 'Masters', or 'Doctorate'));
```

140

在数据库实现中对类型和域的支持

尽管本节描述的**create type**和**create domain**结构是SQL标准的部分，但这里描述的这些结构形式还没有被大多数数据库实现完全支持。PostgreSQL支持**create domain**结构，但是其**create type**结构具有不同的语法和解释。

IBM DB2支持**create type**的一个版本，它使用**create distinct type**语法，但不支持**create domain**。微软的SQL Server实现了**create type**结构的一个版本，支持域约束，与SQL的**create domain**结构类似。

Oracle不支持在此描述的任何一种结构。然而，SQL还定义了一个更复杂的面向对象类型系统，我们将在后面第22章学习。通过使用不同形式的**create type**结构，Oracle、IBM DB2、PostgreSQL和SQL Server都支持面向对象类型系统。

4.5.6 create table的扩展

应用常常要求创建与现有的某个表的模式相同的表。SQL提供了一个**create table like**的扩展来支持这项任务：

```
create table temp_instructor like instructor;
```

上述语句创建了一个与*instructor*具有相同模式的新表*temp_instructor*。

当书写一个复杂查询时，把查询的结果存储成一个新表通常是很有用的；这个表通常是临时的。这里需要两条语句，一条用于创建表(具有合适的列)，另一条用于把查询结果插入到表中。SQL:2003提供了一种更简单的技术来创建包含查询结果的表。例如，下面的语句创建了表*t1*，该表包含一个查询的结果。

```
create table t1 as
  (select *
   from instructor
   where dept_name = 'Music')
with data;
```

在默认情况下，列的名称和数据类型是从查询结果中推导出来的。通过在关系名后面列出列名，可以给列显式指派名字。

正如 SQL:2003 标准所定义的，如果省略 **with data** 子句，表会被创建，但不会载入数据。但即使
141 在省略 **with data** 子句的情况下，很多数据库实现还是通过默认方式往表中加载了数据。注意几种数据库实现都用不同语法支持 **create table**… **like** 和 **create table**… **as** 的功能；请参考相应的系统手册以获得进一步细节。

上述 **create table**… **as** 语句与 **create view** 语句非常相似，并且都用查询来定义。两者主要的区别在于当表被创建时表的内容被加载，但视图的内容总是反映当前查询的结果。

4.5.7 模式、目录与环境

要理解模式和目录的形成，需要考虑文件系统中文件是如何命名的。早期的文件系统是平面的，也就是说，所有的文件都存储在同一个目录下。当然，当代的文件系统有一个目录(或者文件夹)结构，文件都存储在子目录下。要单独命名一个文件，我们必须指定文件的完整路径名，例如，/users/avi/db-book/chapter3.tex。

跟早期文件系统一样，早期数据库系统也只为所有关系提供一个命名空间。用户不得不相互协调以保证他们没有对不同的关系使用同样的名字。当代数据库系统提供了三层结构的关系命名机制。最顶层由**目录**(catalog)构成，每个目录都可以包含**模式**(schema)。诸如关系和视图那样的 SQL 对象都包含在**模式**中。(一些数据库实现用术语"数据库"代替术语"目录")。

要在数据库上做任何操作，用户(或程序)都必须先连接到数据库。为了验证用户身份，用户必须提供用户名以及密码(通常情况下)。每个用户都有一个默认的目录和模式，这个组合对用户来说是唯一的。当一个用户连接到数据库系统时，将为该连接设置好默认的目录和模式。这对应于当用户登录进一个操作系统时，把当前目录设置为用户的主(home)目录。

为了唯一标识出一个关系，必须使用一个名字，它包含三部分，例如：

catalog5. univ_shema. course

当名字的目录部分被认为是连接的默认目录时，可以省略目录部分。这样如果 *catalog5* 是默认目录，我们可以用 *univ_schema. course* 来唯一标识上述关系。

如果用户想访问存在于另外的模式中的关系，而不是该用户的默认模式，那就必须指定模式的名字。然而，如果一个关系存在于特定用户的默认模式中，那么连模式的名字也可以省略。这样，如果 *catalog5* 是默认目录并且 *univ_schema* 是默认模式，我们可以只用 *course*。

当有多个目录和模式可用时，不同应用和不同用户可以独立工作而不必担心命名冲突。不仅如此，
142 一个应用的多个版本(一个产品版本，其他是测试版本)可以在同一个数据库系统上运行。

默认目录和模式是为每个连接建立的 SQL **环境**(SQL environment)的一部分。环境还包括用户标识(也称为*授权标识符*)。所有通常的 SQL 语句，包括 DDL 和 DML 语句，都在一个模式的环境中运行。

我们可以用 **create schema** 和 **drop schema** 语句来创建和删除模式。在大多数数据库系统中，模式还随着用户账户的创建而自动创建，此时模式名被置为用户账号名。模式要么建立在默认目录中，要么建立在创建用户账户时所指定的目录中。新创建的模式成为用户账户的默认模式。

创建和删除目录依据实现的不同而不同，这不是 SQL 标准中的一部分。

4.6 授权

我们可能会给一个用户在数据库的某些部分授予几种形式的权限。对数据的授权包括：

- 授权读取数据。
- 授权插入新数据。
- 授权更新数据。
- 授权删除数据。

每种类型的授权都称为一个**权限**(privilege)。我们可以在数据库的某些特定部分(如一个关系或视图)上授权给用户所有这些类型的权限，或者完全不授权，或者这些权限的一个组合。

当用户提交查询或更新时，SQL执行先基于该用户曾获得过的权限检查此查询或更新是否是授权过的。如果查询或更新没有经过授权，那么将被拒绝执行。

除了在数据上的授权之外，用户还可以被授予在数据库模式上的权限，例如，可以允许用户创建、修改或删除关系。拥有某些形式的权限的用户还可以把这样的权限转授(授予)给其他用户，或者撤销(收回)一种此前授出的权限。本节我们将学习每个这样的权限是如何用SQL来指定的。

最大的授权形式是被授予数据库管理员的。数据库管理员可以授权新用户、重构数据库，等等。这种权限方式和操作系统中的**超级用户**、管理员或操作员的权限是类似的。

4.6.1 权限的授予与收回

SQL标准包括 **select**、**insert**、**update** 和 **delete** 权限。权限**所有权限**(all privileges)可以用作所有允许权限的简写形式。一个创建了新关系的用户将自动被授予该关系上的所有权限。

SQL数据定义语言包括授予和收回权限的命令。**grant** 语句用来授予权限。此语句的基本形式为： 143

```
grant <权限列表>
on <关系名或视图名>
to <用户/角色列表>;
```

权限列表使得一个命令可以授予多个权限。角色的概念将在后面4.6.2节讨论。

关系上的 **select** 权限用于读取关系中的元组。下面的 **grant** 语句授予数据库用户 Amit 和 Satoshi 在 *department* 关系上的 **select** 权限：

```
grant select on department to Amit, Satoshi;
```

该授权使得这些用户可以在 *department* 关系上执行查询。

关系上的 **update** 权限允许用户修改关系中的任意元组。**update** 权限既可以在关系的所有属性上授予，又可以只在某些属性上授予。如果 **grant** 语句中包括 **update** 权限，将被授予 **update** 权限的属性列表可以出现在紧跟关键字 **update** 的括号中。属性列表是可选项，如果省略属性列表，则授予的是关系中所有属性上的 update 权限。

下面的 **grant** 语句授予用户 Amit 和 Satoshi 在 *department* 关系的 *budget* 属性上的更新权限：

```
grant update (budget) on department to Amit, Satoshi;
```

关系上的 **insert** 权限允许用户往关系中插入元组。**insert** 权限也可以指定属性列表；对关系所作的任何插入必须只针对这些属性，系统将其余属性要么赋默认值(如果这些属性上定义了默认值)，要么赋 *null*。

关系上的 **delete** 权限允许用户从关系中删除元组。

用户名 **public** 指系统的所有当前用户和将来的用户。因此，对 **public** 的授权隐含着对所有当前用户和将来用户的授权。

在默认情况下，被授予权限的用户/角色无权把此权限授予其他用户/角色。SQL允许用授予权限来指定权限的接受者可以进一步把权限授予其他用户。我们将在4.6.5节详细讨论这个特性。 144

值得注意的是，SQL授权机制可以对整个关系或一个关系的指定属性授予权限。但是，它不允许对一个关系的指定元组授权。

我们使用 **revoke** 语句来收回权限。此语句的形式与 **grant** 几乎是一样的：

```
revoke <权限列表>
on <关系名或视图名>
from <用户/角色列表>;
```

因此，要收回前面我们所授予的那些权限，我们书写下列语句：

```
revoke select on department from Amit, Satoshi;
revoke update (budget) on department from Amit, Satoshi;
```

如果被收回权限的用户已经把权限授予了其他用户，权限的收回会更加复杂。我们将在4.6.5节回到这个问题。

4.6.2 角色

考虑在一个大学里不同人所具有的真实世界角色。每个教师必须在同一组关系上具有同种类型的权限。无论何时指定一位新的教师，她都必须被单独授予所有这些权限。

一种更好的方式是指明每个教师应该被授予的权限，并单独标示出哪些数据库用户是教师。系统可以利用这两条信息来确定每位教师的权限。当雇佣了一位新的教师时，必须给他分配一个用户标识符，并且必须将他标示为一位教师，而不需要重新单独授予教师权限。

角色(role)的概念适用于此观念。在数据库中建立一个角色集，可以给角色授予权限，就和给每个用户授权的方式完全一样。每个数据库用户被授予一组他有权扮演的角色(也可能是空的)。

在我们的大学数据库里，角色的例子可以包括 *instructor*、*teaching_assistant*、*student*、*dean* 和 *department_chair*。

另一个不是很合适的方法是建立一个 *instructor* 用户标识，允许每位教师用 *instructor* 用户标识来连接数据库。该方式的问题是它不可能鉴别出到底是哪位教师执行了数据库更新，从而导致安全隐患。使用角色的好处是需要用户用他们自己的用户标识来连接数据库。

145 任何可以授予给用户的权限都可以授予给角色。给用户授予角色就跟给用户授权一样。

在 SQL 中创建角色如下所示：

```
create role instructor;
```

然后角色就可以像用户那样被授予权限，如在这样的语句中：

```
grant select on takes
to instructor;
```

角色可以授予给用户，也可以授予给其他角色，如这样的语句：

```
grant dean to Amit;
create role dean;
grant instructor to dean;
grant dean to Satoshi;
```

因此，一个用户或一个角色的权限包括：

- 所有直接授予用户/角色的权限。
- 所有授予给用户/角色所拥有角色的权限。

注意可能存在着一个角色链；例如，角色 *teaching_assistant* 可能被授予所有的 *instructor*。接着，角色 *instructor* 被授予所有的 *dean*。这样，角色 *dean* 就继承了所有被授予给角色 *instructor* 和 *teaching_assistant* 的权限，还包括直接赋给 *dean* 的权限。

当一个用户登录到数据库系统时，在此会话中用户执行的动作拥有所有直接授予该用户的权限，以及所有(直接地或通过其他角色间接地)授予该用户所拥有角色的权限。这样，如果一个用户 Amit 被授予了角色 *dean*，用户 Amit 就拥有所有直接授予给 Amit 的权限，以及授予给 *dean* 的权限，再加上授予给 *instructor* 和 *teaching_assistant* 的权限，如果像上面那样，这些角色被(直接地或间接地)授予给角色 *dean* 的话。

值得注意的是，基于角色的授权概念并没有在 SQL 中指定，但在很多的共享应用中，基于角色的授权被广泛应用于存取控制。

4.6.3 视图的授权

在我们的大学例子中，考虑有一位工作人员，他需要知道一个给定系(比如 Geology 系)里所有员工的工资。该工作人员无权看到其他系中员工的相关信息。因此，该工作人员对 *instructor* 关系的直接访问必须被禁止。但是，如果他要访问 Geology 系的信息，就必须得到在一个视图上的访问权限，我们
146 称该视图为 *geo_instructor*，它仅由属于 Geology 系的那些 *instructor* 元组构成。该视图可以用 SQL 定义如下：

```
create view geo_instructor as
  (select *
   from instructor
   where dept_name = 'Geology');
```

假设该工作人员提出如下 SQL 查询：

```
select *
from geo_instructor;
```

显然，该工作人员有权看到此查询的结果。但是，当查询处理器将此查询转换为数据库中实际关系上的查询时，它产生了一个在 *instructor* 上的查询。这样，系统必须在开始查询处理以前，就检查该工作人员查询的权限。

创建视图的用户不需要获得该视图上的所有权限。他得到的那些权限不会为他提供超越他已有权限的额外授权。例如，如果一个创建视图的用户在用来定义视图的关系上没有 **update** 权限的话，那么他不能得到视图上的 **update** 权限。如果用户创建一个视图，而此用户在该视图上不能获得任何权限，系统会拒绝这样的视图创建请求。在我们的 *geo_instructor* 视图例子中，视图的创建者必须在 *instructor* 关系上具有 **select** 权限。

正如我们将在 5.2 节看到的那样，SQL 支持创建函数和过程，在函数和过程中可以包括查询与更新。在函数或过程上可以授予 **execute** 权限，以允许用户执行该函数或过程。在默认情况下，和视图类似，函数和过程具有其创建者所拥有的所有权限。在效果上，该函数或过程的运行就像其被创建者调用了那样。

尽管此行为在很多情况下是恰当的，但是它并不总是恰当的。从 SQL:2003 开始，如果函数定义有一个额外的 **sql security invoker** 子句，那么它就在调用该函数的用户的权限下执行，而不是在函数定义者的权限下执行。这就允许创建的函数库能够在与调用者相同的权限下运行。

4.6.4 模式的授权

SQL 标准为数据库模式指定了一种基本的授权机制：只有模式的拥有者才能够执行对模式的任何修改，诸如创建或删除关系，增加或删除关系的属性，以及增加或删除索引。 147

然而，SQL 提供了一种 **references** 权限，允许用户在创建关系时声明外码。SQL 的 **references** 权限可以与 **update** 权限类似的方式授予到特定属性上。下面的 **grant** 语句允许用户 Mariano 创建这样的关系，它能够参照 *department* 关系的码 *dept_name*：

```
grant references (dept_name) on department to Mariano;
```

初看起来，似乎没有理由不允许用户创建参照了其他关系的外码。但是，回想一下外码约束限制了被参照关系上的删除和更新操作。假定 Mariano 在关系 *r* 中创建了一个外码，它参照 *department* 关系的 *dept_name* 属性，然后在 *r* 中插入一条属于 Geology 系的元组。那么就再也不可能从 *department* 关系中将 Geology 系删除，除非同时也修改关系 *r*。这样，Mariano 定义的外码限制了其他用户将来的行为；因此，需要有 **references** 权限。

继续使用 *department* 关系的例子，如果要创建关系 *r* 上的 **check** 约束，并且该约束有参照 *department* 的子查询，那么还需要有 *department* 上的 references 权限。其原因与外码约束的情况类似，因为参照了一个关系的 check 约束限制了对该关系可能的更新。

4.6.5 权限的转移

获得了某些形式授权的用户可能被允许将此授权传递给其他用户。在默认方式下，被授予权限的用户/角色无权把得到的权限再授予给另外的用户/角色。如果我们在授权时允许接受者把得到的权限再传递给其他用户，我们可以在相应的 **grant** 命令后面附加 **with grant option** 子句。例如，如果我们希望授予 Amit 在 *department* 上的 **select** 权限，并且允许 Amit 将该权限授予给其他用户，我们可以写：

```
grant select on department to Amit with grant option;
```

一个对象(关系/视图/角色)的创建者拥有该对象上的所有权限，包括给其他用户授权的权限。

作为一个例子，考虑大学数据库中 *teaches* 关系上更新权限的授予。假设最初数据库管理员将

teaches 上的更新权限授给用户 U_1、U_2 和 U_3，他们接下来又可以将这一授权传递给其他用户。指定权限从一个用户到另一个用户的传递可以表示为**授权图**(authorization graph)。该图中的顶点是用户。

考虑 *teaches* 上更新权限所对应的授权图。如果用户 U_i 将 *teaches* 上的更新权限授给 U_j，则图中包含边 $U_i \rightarrow U_j$。图的根是数据库管理员。在图 4-10 所示的示例图中，注意 U_1 和 U_2 都给用户 U_5 授权了；而 U_4 只从 U_1 处获得了授权。

用户具有权限的充分必要条件是：当且仅当存在从授权图的根(即代表数据库管理员的顶点)到代表该用户顶点的路径。

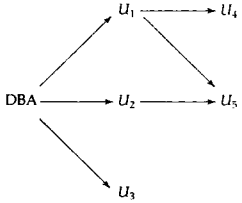

图 4-10 权限授予图(U_1，U_1，…，U_5 是用户，DBA 代表数据库管理员)

4.6.6 权限的收回

假设数据库管理员决定收回用户 U_1 的授权。由于 U_4 从 U_1 处获得过授权，因此其权限也应该被收回。可是，U_5 既从 U_1 处又从 U_2 处获得过授权。由于数据库管理员没有从 U_2 处收回 *teaches* 上的更新权限，U_5 继续拥有 *teaches* 上的更新权限。如果 U_2 最终从 U_5 处收回授权，则 U_5 失去权限。

一对狡猾的用户可能企图通过相互授权来破坏权限收回规则。例如，如果 U_2 最初由数据库管理员授予了一种权限，U_2 进而把此权限授予给 U_3。假设 U_3 现在把此权限授回给 U_2。如果数据库管理员从 U_2 收回权限，看起来好像 U_2 保留了通过 U_3 获得的授权。然而，注意一旦管理员从 U_2 收回权限，在授权图中就不存在从根到 U_2 或 U_3 的路径了。这样，SQL 保证从这两个用户那里都收回了权限。

正如我们刚才看到的那样，从一个用户/角色那里收回权限可能导致其他用户/角色也失去该权限。这一行为称作**级联收回**。在大多数的数据库系统中，级联是默认行为。然而，**revoke** 语句可以申明 **restrict** 来防止级联收回：

revoke select on *department* **from** Amit, Satoshi **restrict**;

148 ~ 149

这种情况下，如果存在任何级联收回，系统就返回一个错误，并且不执行收权动作。

可以用关键字 **cascade** 来替换 **restrict**，以表示需要级联收回；然而，**cascade** 可以省略，就像我们前述例子中的那样，因为它是默认行为。

下面的 **revoke** 语句仅仅收回 grant option，而并不是真正收回 **select** 权限：

revoke grant option for select on *department* **from** Amit;

注意一些数据库实现不支持上述语法；它们采用另一种方式：收回权限本身，然后不带 grant option 重新授权。

级联收回在许多情况下是不合适的。假定 Satoshi 具有 *dean* 角色，他将 *instructor* 授给 Amit，后来 *dean* 角色从 Satoshi 收回(也许由于 Satoshi 离开了大学)；Amit 继续被雇佣为教职工，并且还应该保持 *instructor* 角色。

为了处理以上情况，SQL 允许权限由一个角色授予，而不是由用户来授予。SQL 有一个与会话所关联的当前角色的概念。默认情况下，一个会话所关联的当前角色是空的(某些特殊情况除外)。一个会话所关联的当前角色可以通过执行 **set role** *role_name* 来设置。指定的角色必须已经授予给用户，否则 **set role** 语句执行失败。

如果要在授予权限时将授权人设置为一个会话所关联的当前角色，并且当前角色不为空的话，我们可以在授权语句后面加上

granted by current_role

子句。

假设将角色 *instructor*(或其他权限)授给 Amit 是用 **granted by current_role** 子句实现的，当前角色被设置为 *dean* 而不是授权人(用户 Satoshi)，那么，从 Satoshi 处收回角色/权限(包括角色 *dean*)就不会导致收回以角色 *dean* 作为授权人所授予的权限，即使 Satoshi 是执行该授权的用户；这样，即使在 Satoshi 的权限被收回后，Amit 仍然能够保持 *instructor* 角色。

4.7 总结

- SQL 支持包括内连接、外连接在内的几种连接类型，以及几种形式的连接条件。
- 视图关系可以定义为包含查询结果的关系。视图是有用的，它可以隐藏不需要的信息，可以把信息从多个关系收集到一个单一的视图中。 150
- 事务是一个查询和更新的序列，它们共同执行某项任务。事务可以被提交或回滚。当一个事务被回滚，该事务执行的所有更新所带来的影响将被撤销。
- 完整性约束保证授权用户对数据库所做的改变不会导致数据一致性的破坏。
- 参照完整性约束保证出现在一个关系的给定属性集上的值同样出现在另一个关系的特定属性集上。
- 域约束指定了在一个属性上可能取值的集合。这种约束也可以禁止在特定属性上使用空值。
- 断言是描述性表达式，它指定了我们要求总是为真的谓词。
- SQL 数据定义语言提供对定义诸如 **date** 和 **time** 那样的固有域类型以及用户定义域类型的支持。
- 通过 SQL 授权机制，可以按照在数据库中不同数据值上数据库用户所允许的访问类型对他们进行区分。
- 获得了某种形式授权的用户可能允许将此授权传递给其他用户。但是，对于权限怎样在用户间传递我们必须很小心，以保证这样的权限在将来的某个时候可以被收回。
- 角色有助于根据用户在组织机构中所扮演的角色，把一组权限分配给用户。

术语回顾

- 连接类型
  - □ 内连接和外连接
  - □ 左外连接、右外连接和全外连接
  - □ natural 连接条件、using 连接条件和 on 连接条件
- 视图定义
- 物化视图
- 视图更新
- 事务
  - □ 提交
  - □ 回滚
  - □ 原子事务
- 完整性约束
- 域约束
- 唯一性约束
- check 子句
- 参照完整性
  - □ 级联删除
  - □ 级联更新
- 断言
- 日期和时间类型
- 默认值
- 索引
- 大对象
- 用户定义类型
- 域
- 目录
- 模式
- 授权
- 权限
  - □ 选择
  - □ 插入
  - □ 更新
  - □ 所有权限
  - □ 授予权限
  - □ 收回权限
  - □ 授予权限的权限
  - □ grant option
- 角色
- 视图授权
- 执行授权
- 调用者权限
- 行级授权

实践习题

4.1 用 SQL 写出下面的查询：

a. 显示所有教师的列表，列出他们的 ID、姓名以及所讲授课程段的编号。对于没有讲授任何课程段的教师，确保将课程段编号显示为 0。在你的查询中应该使用外连接，不能使用标量子查询。

b. 使用标量子查询，不使用外连接写出上述查询。

c. 显示 2010 年春季开设的所有课程的列表，包括讲授课程段的教师的姓名。如果一个课程段有不止一位教师讲授，那么有多少位教师，此课程段在结果中就出现多少次。如果一个课程段没有任何教师，它也要出现在结果中，相应的教师名置为“—”。

d. 显示所有系的列表，包括每个系中教师的总数，不能使用标量子查询。确保正确处理没有教师的系。

4.2 不使用 SQL 外连接运算也可以在 SQL 中计算外连接表达式。为了阐明这个事实，不使用外连接表达式重写下面每个 SQL 查询：

a. **select** * **from** *student* **natural left outer join** *takes*
b. **select** * **from** *student* **natural full outer join** *takes*

151 ~ 152 4.3 假设有三个关系 $r(A, B)$、$s(B, C)$和$t(B, D)$，其中所有属性声明为非空。考虑表达式：

- r **natural left outer join** (s **natural left outer join** t)
- (r **natural left outer join** s) **natural left outer join** t

a. 给出关系 r、s 和 t 的实例，使得在第二个表达式的结果中，属性 C 有一个空值但属性 D 有非空值。
b. 在第一个表达式的结果中，上述模式中的 C 为空且 D 非空有可能吗？解释原因。

4.4 **测试 SQL 查询**(testing SQL query)：为了测试一个用文字表达的查询能否正确地用 SQL 写出，通常 SQL 查询会在多个测试数据库上执行，并用人工来检测在每个测试数据库上 SQL 查询的结果是否与用文字表达的意图相匹配。

a. 在 3.3.3 节我们见过一个错误的 SQL 查询，它希望找出每位教师讲授了哪些课程；该查询计算 *instructor*、*teaches* 和 *course* 的自然连接，结果是无意地造成了让 *instructor* 和 *course* 的 *dept_name* 属性取值相等。给出一个数据集样例，能有助于发现这种特别的错误。
b. 在创建测试数据库时，对每个外码来说，在被参照关系中创建与参照关系中任何元组都不匹配的元组是很重要的。请用大学数据库上的一个查询样例来解释原因。
c. 在创建测试数据库时，倘若外码属性可空的话，在外码属性上创建具有空值的元组是很重要的(SQL 允许在外码属性上取空值，只要它们不是主码的一部分，并且没有被声明为 **not null**)。请用大学数据库上的一个查询样例来解释原因。

提示：使用来自习题 4.1 中的查询。

4.5 基于习题 3.2 中的查询，指出如何定义视图 *student_grades* (*ID*, *GPA*)，它给出了每个学生的等级分值的平均值；回忆一下，我们用关系 *grade_points*(*grade*, *points*)来把用字母表示的成绩等级转换为用数字表示的得分。你的视图定义要确保能够正确处理在 *takes* 关系的 *grade* 属性上取 *null* 值的情况。

153 4.6 完成图 4-8 中的大学数据库的 SQL DDL 定义以包括 *student*、*takes*、*advisor* 和 *prereq* 关系。

4.7 考虑图 4-11 所示的关系数据库。给出这个数据库的 SQL DDL 定义。指出应有的参照完整性约束，并把它们包括在 DDL 定义中。

employee(*employee_name*, *street*, *city*)
works(*employee_name*, *company_name*, *salary*)
company(*company_name*, *city*)
manages(*employee_name*, *manager_name*)

图 4-11 习题 4.7 和习题 4.12 的雇员数据库

4.8 正如在 4.4.7 节所讨论的，我们希望能够满足约束“每位教师不能在同一个学期的同一个时间段在两个不同的教室授课”。

a. 写出一个 SQL 查询，它返回违反此约束的所有(*instructor*, *section*)组合。
b. 写出一个 SQL 断言来强制实现此约束(正如在 4.4.7 节所讨论的那样，当代的数据库系统不支持这样的断言，尽管它们是 SQL 标准的一部分)。

4.9 SQL 允许外码依赖指向同一个关系，如下面的例子所示：

```
create table manager
  (employee_name  var char(20) not null
  manager_name  var char(20) not null,
  primary key employee_name,
  foreign key (manager_name) references manager
  on delete cascade)
```

这里 *employee_name* 是 *manager* 表的码，意味着每个雇员最多只有一个经理。外码子句要求每个经理都是一个雇员。请准确解释当 *manager* 关系中一个元组被删除时会发生什么情况。

4.10 SQL 提供一个称为 **coalesce** 的 n 维数组运算，其定义如下：**coalesce**(A_1, A_2, …, A_n)返回序列 A_1, A_2, …, A_n 中第一个非空值 A_i，如果所有的 A_1, A_2, …, A_n 全为 *null*，则返回 *null*。

假设 a 和 b 是模式分别为 $A(name, address, title)$ 和 $B(name, address, salary)$ 的关系。如何用带 **on** 条件和 **coalesce** 运算的**全外连接**(full outer-join)运算来表达 a **natural full outer join** b。要保证结果关系中不包含属性 *name* 和 *address* 的两个副本，并且即使 a 和 b 中的某些元组在属性 *name* 和 *address* 上取空值的情况下，所给出的解决方案仍然是正确的。

4.11 一些研究者提出带标记的(*marked*)空值的概念。带标记空值 $\perp_i$ 与它自身相等，但是如果 $i \neq j$，那么 $\perp_i \neq \perp_j$。带标记空值的一个应用是允许通过视图进行特定的更新。考虑视图 *instructor_info*(4.2节)，如何利用带标记空值来允许通过 *instructor_info* 插入元组(99999，“Johnson”，“Taylor”)。

习题

4.12 对于图4-11中的数据库，写出一个查询来找到那些没有经理的雇员。注意一个雇员可能只是没有列出其经理，或者可能有 *null* 经理。使用外连接书写查询，然后不用外连接再重写查询。

4.13 在什么情况下，查询

```
select *
from student natural full outer join takes natural full outer join course
```

将包含在属性 *title* 上取空值的元组？

4.14 给定学生每年修到的学分总数，如何定义视图 *tot_credits*(*year*, *num_credits*)。

4.15 给出如何用 **case** 运算表达习题4.10中的 **coalesce** 运算。

4.16 如本章定义的参照完整性约束正好涉及两个关系。考虑包括如图4-12所示关系的数据库。假设我们希望要求每个出现在 *address* 中的名字必须出现在 *salaried_worker* 或者 *hourly_worker* 中，但不一定要求在两者中同时出现。

a. 给出表达这种约束的语法。

b. 讨论为了使这种形式的约束生效，系统必须采取什么行动。

salaried_worker (*name*, *office*, *phone*, *salary*)
hourly_worker (*name*, *hourly_wage*)
address (*name*, *street*, *city*)

图4-12 习题4.16的雇员数据库

4.17 当一个经理如Satoshi授予权限的时候，授权应当由经理角色完成，而不是由用户Satoshi完成。解释其原因。

4.18 假定用户 A 拥有关系 r 上的所有权限，该用户把关系 r 上的查询权限以及授予该权限的权限授予给 **public**。假定用户 B 将 r 上的查询权限授予A。这是否会导致授权图中的环？解释原因。 154 ~ 155

4.19 将每个关系存放在一个单独的操作系统文件中的数据库系统可能使用操作系统的授权机制，而不是自己定义一种专门的机制。请论述这种方法的一个优点和一个缺点。

文献注解

对于SQL参考资料请参考第3章的文献注解。

SQL用于决定视图更新能力的规则，以及更新操作如何影响底层的数据库关系，都定义在SQL:1999标准中。Melton和Simon[2001]中对此进行了总结。 156