SQL

商业性使用或实验性使用的数据库查询语言有好几种。在本章以及第4章和第5章，我们学习使用最为广泛的查询语言：SQL。

尽管我们说SQL语言是一种"查询语言"，但是除了数据库查询，它还具有很多别的功能，它可以定义数据结构、修改数据库中的数据以及说明安全性约束条件等。

我们的目的并不是提供一个完整的SQL用户手册，而是介绍SQL的基本结构和概念。SQL的各种实现可能在一些细节上有所不同，或者只支持整个语言的一个子集。

3.1 SQL查询语言概览

SQL最早的版本是由IBM开发的，它最初被叫做Sequel，在20世纪70年代早期作为System R项目的一部分。Sequel语言一直发展至今，其名称已变为SQL(结构化查询语言)。现在有许多产品支持SQL语言，SQL已经很明显地确立了自己作为标准的关系数据库语言的地位。

1986年美国国家标准化组织(ANSI)和国际标准化组织(ISO)发布了SQL标准：SQL-86。1989年ANSI发布了一个SQL的扩充标准：SQL-89。该标准的下一个版本是SQL-92标准，接着是SQL:1999，SQL:2003，SQL:2006，最新的版本是SQL:2008。文献注解中提供了关于这些标准的参考文献。

SQL语言有以下几个部分：

- **数据定义语言**(Data-Definition Language，DDL)：SQL DDL提供定义关系模式、删除关系以及修改关系模式的命令。
- **数据操纵语言**(Data-Manipulation Language，DML)：SQL DML提供从数据库中查询信息，以及在数据库中插入元组、删除元组、修改元组的能力。
- **完整性**(integrity)：SQL DDL包括定义完整性约束的命令，保存在数据库中的数据必须满足所定义的完整性约束。破坏完整性约束的更新是不允许的。
- **视图定义**(view definition)：SQL DDL包括定义视图的命令。
- **事务控制**(transaction control)：SQL包括定义事务的开始和结束的命令。
- **嵌入式SQL和动态SQL**(embedded SQL and dynamic SQL)：嵌入式和动态SQL定义SQL语句如何嵌入到通用编程语言，如C、C++和Java中。
- **授权**(authorization)：SQL DDL包括定义对关系和视图的访问权限的命令。

本章我们给出对SQL的基本DML和DDL特征的概述。在此描述的特征自SQL-92以来就一直是SQL标准的部分。

在第4章我们提供对SQL查询语言更详细的介绍，包括：(a)各种连接的表达；(b)视图；(c)事务；(d)完整性约束；(e)类型系统；(f)授权。

在第5章我们介绍SQL语言更高级的特征，包括：(a)允许从编程语言中访问SQL的机制；(b)SQL函数和过程；(c)触发器；(d)递归查询；(e)高级聚集特征；(f)为数据分析设计的一些特征，它们在SQL:1999中引入，并在SQL的后续版本中使用。在后面第22章，我们将概述SQL:1999引入的对SQL的面向对象扩充。

尽管大多数SQL实现支持我们在此描述的标准特征，读者还是应该意识到不同SQL实现之间的差异。大多数SQL实现还支持一些非标准的特征，但不支持一些更高级的特征。万一你发现在此描述的

一些语言特征在你使用的系统中不起作用，请参考你的数据库系统用户手册，看看它所支持的特征究竟是什么。

3.2 SQL 数据定义

数据库中的关系集合必须由数据定义语言(DDL)指定给系统。SQL 的 DDL 不仅能够定义一组关系，还能够定义每个关系的信息，包括：

- 每个关系的模式。
- 每个属性的取值类型。
- 完整性约束。
- 每个关系维护的索引集合。
- 每个关系的安全性和权限信息。
- 每个关系在磁盘上的物理存储结构。

58

我们在此只讨论基本模式定义和基本类型，对 SQL DLL 其他特征的讨论将放到第 4 章和第 5 章进行。

3.2.1 基本类型

SQL 标准支持多种固有类型，包括：

- **char**(n)：固定长度的字符串，用户指定长度 n。也可以使用全称 **character**。
- **varchar**(n)：可变长度的字符串，用户指定最大长度 n，等价于全称 **character varying**。
- **int**：整数类型(和机器相关的整数的有限子集)，等价于全称 **integer**。
- **smallint**：小整数类型(和机器相关的整数类型的子集)。
- **numeric**(p, d)：定点数，精度由用户指定。这个数有 p 位数字(加上一个符号位)，其中 d 位数字在小数点右边。所以在一个这种类型的字段上，**numeric**(3，1)可以精确储存 44.5，但不能精确存储 444.5 或 0.32 这样的数。
- **real**，**double precision**：浮点数与双精度浮点数，精度与机器相关。
- **float**(n)：精度至少为 n 位的浮点数。

更多类型将在 4.5 节介绍。

每种类型都可能包含一个被称作空值的特殊值。空值表示一个缺失的值，该值可能存在但并不为人所知，或者可能根本不存在。在可能的情况下，我们希望禁止加入空值，正如我们马上将看到的那样。

char 数据类型存放固定长度的字符串。例如，属性 A 的类型是 **char**(10)。如果我们为此属性存入字符串“Avi”，那么该字符串后会追加 7 个空格来使其达到 10 个字符的串长度。反之，如果属性 B 的类型是 **varchar**(10)，我们在属性 B 中存入字符串“Avi”，则不会增加空格。当比较两个 **char** 类型的值时，如果它们的长度不同，在比较之前会自动在短值后面加上额外的空格以使它们的长度一致。

当比较一个 **char** 类型和一个 **varchar** 类型的时候，也许读者会期望在比较之前会自动在 **varchar** 类型后面加上额外的空格以使长度一致；然而，这种情况可能发生也可能不发生，这取决于数据库系统。其结果是，即便上述属性 A 和 B 中存放的是相同的值“Avi”，$A = B$ 的比较也可能返回假。我们建议始终使用 **varchar** 类型而不是 **char** 类型来避免这样的问题。

59

SQL 也提供 **nvarchar** 类型来存放使用 Unicode 表示的多语言数据。然而，很多数据库甚至允许在 **varchar** 类型中存放 Unicode(采用 UTF-8 表示)。

3.2.2 基本模式定义

我们用 **create table** 命令定义 SQL 关系。下面的命令在数据库中创建了一个 *department* 关系。

```
create table department
    (dept_name varchar (20),
    building varchar (15),
    budget numeric (12, 2),
    primary key (dept_name));
```

上面创建的关系具有三个属性，*dept_name* 是最大长度为 20 的字符串，*building* 是最大长度为 15 的字符串，*budget* 是一个 12 位的数，其中 2 位数字在小数点后面。**create table** 命令还指明了 *dept_name* 属性是 *department* 关系的主码。

create table 命令的通用形式是：

create table *r*
(A_1 D_1,
A_2 D_2,
…,
A_n D_n,
<完整性约束$_1$>,
…,
<完整性约束$_k$>);

其中 *r* 是关系名，每个 A_i 是关系 *r* 模式中的一个属性名，D_i 是属性 A_i 的域，也就是说 D_i 指定了属性 A_i 的类型以及可选的约束，用于限制所允许的 A_i 取值的集合。

create table 命令后面用分号结束，本章后面的其他 SQL 语句也是如此，在很多 SQL 实现中，分号是可选的。

SQL 支持许多不同的完整性约束。在本节我们只讨论其中少数几个：

- **primary key**(A_{j1}, A_{j2}, …, A_{jm})：**primary-key** 声明表示属性 A_{j1}, A_{j2}, …, A_{jm} 构成关系的主码。主码属性必须非空且唯一，也就是说没有一个元组在主码属性上取空值，关系中也没有两个元组在所有主码属性上取值相同。虽然主码的声明是可选的，但为每个关系指定一个主码通常会更好。
- **foreign key**(A_{k1}, A_{k2}, …, A_{kn}) **references**：**foreign key** 声明表示关系中任意元组在属性(A_{k1}, A_{k2}, …, A_{kn})上的取值必须对应于关系 *s* 中某元组在主码属性上的取值。

  图 3-1 给出了我们在书中使用的大学数据库的部分 SQL DDL 定义。*course* 表的定义中声明了“**foreign key** (*dept_name*) **references** *department*”。此外码声明表明对于每个课程元组来说，该元组所表示的系名必然存在于 *department* 关系的主码属性(*dept_name*)中。没有这个约束的话，就可能有某门课程指定了一个不存在的系名。图 3-1 还给出了表 *section*、*instructor* 和 *teaches* 上的外码约束。

```
create table department
    (dept_name        varchar(20),
     building         varchar (15),
     budget           numeric (12,2),
     primary key (dept_name));

create table course
    (course_id        varchar (7),
     title            varchar (50),
     dept_name        varchar (20),
     credits          numeric (2,0),
     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),
     primary key (ID),
     foreign key (dept_name) references department);

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),
     foreign key (course_id) references course);

create table teaches
    (ID               varchar (5),
     course_id        varchar (8),
     sec_id           varchar (8),
     semester         varchar (6),
     year             numeric (4,0),
     primary key (ID, course_id, sec_id, semester, year),
     foreign key (course_id, sec_id, semester, year)
references section,
     foreign key (ID) references instructor);
```

图 3-1 大学数据库的部分 SQL 数据定义

- **not null**：一个属性上的 **not null** 约束表明在该属性上不允许空值。换句话说，此约束把空值排除在该属性域之外。例如在图 3-1 中，*instructor* 关系的 *name* 属性上的 **not null** 约束保证了教师的姓名不会为空。

有关外码约束的更多细节以及 **create table** 命令可能包含的其他完整性约束将在后面 4.4 节介绍。

SQL 禁止破坏完整性约束的任何数据库更新。例如，如果关系中一条新插入或新修改的元组在任意一个主码属性上有空值，或者元组在主码属性上的取值与关系中的另一个元组相同，SQL 将标记一个错误，并阻止更新。类似地，如果插入的 *course* 元组在 *dept_name* 上的取值没有出现在 *department* 关系中，就会破坏 *course* 上的外码约束，SQL 会阻止这种插入的发生。

一个新创建的关系最初是空的。我们可以用 **insert** 命令将数据加载到关系中。例如，如果我们希望插入如下事实：在 Biology 系有一个名叫 Smith 的教师，其 *instructor_id* 为 10211，工资为 66 000 美元，可以这样写：

```
insert into instructor
    values (10211, 'Smith', 'Biology', 66000);
```

值被给出的顺序应该遵循对应属性在关系模式中列出的顺序。插入命令有很多有用的特性，后面将在 3.9.2 节进行更详细的介绍。

我们可以使用 **delete** 命令从关系中删除元组。命令

```
delete from student;
```

将从 *student* 关系中删除所有元组。其他格式的删除命令允许指定待删除的元组；我们将在 3.9.1 节对删除命令进行更详细的介绍。

如果要从 SQL 数据库中去掉一个关系，我们使用 **drop table** 命令。**drop table** 命令从数据库中删除关于被去掉关系的所有信息。命令

```
drop table r;
```

是比

```
delete from r;
```

更强的语句。后者保留关系 *r*，但删除 *r* 中的所有元组。前者不仅删除 *r* 的所有元组，还删除 *r* 的模式。一旦 *r* 被去掉，除非用 **create table** 命令重建 *r*，否则没有元组可以插入到 *r* 中。

我们使用 **alter table** 命令为已有关系增加属性。关系中的所有元组在新属性上的取值将被设为 *null*。**alter table** 命令的格式为：

```
alter table r add A D;
```

其中 *r* 是现有关系的名字，*A* 是待添加属性的名字，*D* 是待添加属性的域。我们可以通过命令

```
alter table r drop A;
```

从关系中去掉属性。其中 *r* 是现有关系的名字，*A* 是关系的一个属性的名字。很多数据库系统并不支持去掉属性，尽管它们允许去掉整个表。

3.3 SQL 查询的基本结构

SQL 查询的基本结构由三个子句构成：**select**、**from** 和 **where**。查询的输入是在 **from** 子句中列出的关系，在这些关系上进行 **where** 和 **select** 子句中指定的运算，然后产生一个关系作为结果。我们通过例子介绍 SQL 的语法，后面再描述 SQL 查询的通用结构。

3.3.1 单关系查询

我们考虑使用大学数据库例子的一个简单查询："找出所有教师的名字"。教师的名字可以在 *instructor* 关系中找到，因此我们把该关系放到 **from** 子句中。教师的名字出现在 *name* 属性中，因此我们把它放到 **select** 子句中。

```
select name
from instructor;
```

其结果是由属性名为 *name* 的单个属性构成的关系。如果 *instructor* 关系如图 2-1 所示，那么上述查询的结果关系如图 3-2 所示。

name
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim

图 3-2 "**select** *name* **from** *instructor*"的结果

现在考虑另一个查询："找出所有教师所在的系名"，此查询可写为：

```
select dept_name
from instructor;
```

因为一个系有多个教师，所以在 *instructor* 关系中，一个系的名称可以出现不止一次。上述查询的结果是一个包含系名的关系，如图 3-3 所示。

在关系模型的形式化数学定义中，关系是一个集合。因此，重复的元组不会出现在关系中。在实践中，去除重复是相当费时的，所以 SQL 允许在关系以及 SQL 表达式结果中出现重复。因此，在上述 SQL 查询中，每个系名在 *instructor* 关系的元组中每出现一次，都会在查询结果中列出一次。

dept_name
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

图 3-3 "**select** *dept_name* **from** *instructor*"的结果

有时候我们想要强行删除重复，可在 **select** 后加入关键词 **distinct**。如果我们想去除重复，可将上述查询重写为：

```
select distinct dept_name
from instructor;
```

在上述查询的结果中，每个系名最多只出现一次。

SQL 允许我们使用关键词 **all** 来显式指明不去除重复：

```
select all dept_name
from instructor;
```

既然保留重复元组是默认的，在例子中我们将不再使用 **all**。为了保证在我们例子的查询结果中删除重复元组，我们将在所有必要的地方使用 **distinct**。

select 子句还可带含有 +、-、*、/运算符的算术表达式，运算对象可以是常数或元组的属性。例如，查询

```
select ID, name, dept_name, salary * 1.1
from instructor;
```

返回一个与 *instructor* 一样的关系，只是属性 *salary* 的值是原来的 1.1 倍。这显示了如果我们给每位教师增长 10% 的工资的结果。注意这并不导致对 *instructor* 关系的任何改变。

SQL 还提供了一些特殊数据类型，如各种形式的日期类型，并允许一些作用于这些类型上的算术函数。我们在 4.5.1 节进一步讨论这个问题。

where 子句允许我们只选出那些在 **from** 子句的结果关系中满足特定谓词的元组。考虑查询"找出所有在 Computer Science 系并且工资超过 70 000 美元的教师的姓名"，该查询用 SQL 可以写为：

60 ~ 65

```
select name
from instructor
where dept_name = 'Comp. Sci.' and salary > 70000;
```

如果 *instructor* 关系如图 2-1 所示，那么上述查询的结果关系如图 3-4 所示。

name
Katz
Brandt

图 3-4 "找出所有在 Computer Science 系并且工资超过 70 000 美元的教师的姓名"的结果

SQL 允许在 **where** 子句中使用逻辑连词 **and**、**or** 和 **not**。逻辑连词的运算对象可以是包含比较运算符 <、<=、>、>=、=和<>的表达式。SQL 允许我们使用比较运算符来比较字符串、算术表达式以及特殊类型，如日期类型。

在本章的后面，我们将研究 **where** 子句谓词的其他特征。

3.3.2 多关系查询

到此为止我们的查询示例都是基于单个关系的。通常查询需要从多个关系中获取信息。我们现在来学习如何书写这样的查询。

作为一个示例，假设我们想回答这样的查询："找出所有教师的姓名，以及他们所在系的名称和系所在建筑的名称"。

考虑 *instructor* 关系的模式，我们发现可以从 *dept_name* 属性得到系名，但是系所在建筑的名称是在

department 关系的 *building* 属性中给出的。为了回答查询，*instructor* 关系中的每个元组必须与 *department* 关系中的元组匹配，后者在 *dept_name* 上的取值相配于 *instructor* 元组在 *dept_name* 上的取值。

为了在 SQL 中回答上述查询，我们把需要访问的关系都列在 **from** 子句中，并在 **where** 子句中指定匹配条件。上述查询可用 SQL 写为：

```
select name, instructor. dept_name, building
from instructor, department
where instructor. dept_name = department. dept_name;
```

如果 *instructor* 和 *department* 关系分别如图 2-1 和图 2-5 所示，那么此查询的结果关系如图 3-5 所示。

注意 *dept_name* 属性既出现在 *instructor* 关系中，也出现在 *department* 中，关系名被用作前缀（在 *instructor. dept_name* 和 *department. dept_name* 中）来说明我们使用的是哪个属性。相反，属性 *name* 和 *building* 只出现在一个关系中，因而不需要把关系名作为前缀。

这种命名惯例需要出现在 **from** 子句中的关系具有可区分的名字。在某些情况下这样的要求会引发问题，比如当需要把来自同一个关系的两个不同元组的信息进行组合的时候。在 3.4.1 节，我们将看到如何使用更名运算来避免这样的问题。

name	*dept_name*	*building*
Srinivasan	Comp. Sci.	Taylor
Wu	Finance	Painter
Mozart	Music	Packard
Einstein	Physics	Watson
El Said	History	Painter
Gold	Physics	Watson
Katz	Comp. Sci.	Taylor
Califieri	History	Painter
Singh	Finance	Painter
Crick	Biology	Watson
Brandt	Comp. Sci.	Taylor
Kim	Elec. Eng.	Taylor

图 3-5 “找出所有教师的姓名，以及他们所在系的名称和系所在建筑的名称”的结果

现在我们考虑涉及多个关系的 SQL 查询的通用形式。正如我们前面已经看到的，一个 SQL 查询可以包括三种类型的子句：**select** 子句、**from** 子句和 **where** 子句。每种子句的作用如下：

- **select** 子句用于列出查询结果中所需要的属性。
- **from** 子句是一个查询求值中需要访问的关系列表。
- **where** 子句是一个作用在 **from** 子句中关系的属性上的谓词。

一个典型的 SQL 查询具有如下形式：

```
select A1, A2, ···, An
from r1, r2, ···, rm
where P;
```

每个 A_i 代表一个属性，每个 r_i 代表一个关系。P 是一个谓词。如果省略 **where** 子句，则谓词 P 为 **true**。

66
67

尽管各子句必须以 **select**、**from**、**where** 的次序写出，但理解查询所代表运算的最容易的方式是以运算的顺序来考察各子句：首先是 **from**，然后是 **where**，最后是 **select** ⊖。

通过 **from** 子句定义了一个在该子句中所列出关系上的笛卡儿积。它可以用集合理论来形式化地定义，但最好通过下面的迭代过程来理解，此过程可为 **from** 子句的结果关系产生元组。

```
for each 元组 t1 in 关系 r1
    for each 元组 t2 in 关系 r2
        ···
        for each 元组 tm in 关系 rm
        把 t1, t2, ···, tm 连接成单个元组 t
        把 t 加入结果关系中
```

此结果关系具有来自 **from** 子句中所有关系的所有属性。由于在关系 r_i 和 r_j 中可能出现相同的属性名，正如我们此前所看到的，我们在属性名前加上关系名作为前缀，表示该属性来自于哪个关系。

⊖ 实践中，SQL 也许会将表达式转换成更高效执行的等价形式。我们将把效率问题推迟到第 12 章和第 13 章中探讨。

例如，关系 *instructor* 和 *teaches* 的笛卡儿积的关系模式为：

(*instructor. ID*, *instructor. name*, *instructor. dept_name*, *instructor. salary*
teaches. ID, *teaches. course_id*, *teaches. sec_id*, *teaches. semester*, *teaches. year*)

有了这个模式，我们可以区分出 *instructor. ID* 和 *teaches. ID*。对于那些只出现在单个模式中的属性，我们通常去掉关系名前缀。这种简化并不会造成任何混淆。这样我们可以把关系模式写为：

(*instructor. ID*, *name*, *dept_name*, *salary*
teaches. ID, *course_id*, *sec_id*, *semester*, *year*)

为举例说明，考察图2-1中的 *instructor* 关系和图2-7中的 *teaches* 关系。它们的笛卡儿积如图3-6所示，图中只包括了构成笛卡儿积结果的一部分元组。⊖

inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	CompSci.	65000	10101	CS-101	1	Fall	2009
10101	Srinivasan	CompSci.	65000	10101	CS-315	1	Spring	2010
10101	Srinivasan	CompSci.	65000	10101	CS-347	1	Fall	2009
10101	Srinivasan	CompSci.	65000	12121	FIN-201	1	Spring	2010
10101	Srinivasan	CompSci.	65000	15151	MU-199	1	Spring	2010
10101	Srinivasan	CompSci.	65000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2009
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2010
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2009
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2010
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2010
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Music	40000	10101	CS-101	1	Fall	2009
15151	Mozart	Music	40000	10101	CS-315	1	Spring	2010
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2009
15151	Mozart	Music	40000	12121	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2010
15151	Mozart	Music	40000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2009
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2010
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2009
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2010
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

图3-6 *instructor* 关系和 *teaches* 关系的笛卡儿积

通过笛卡儿积把来自 *instructor* 和 *teaches* 中相互没有关联的元组组合起来。*instructor* 中的每个元组和 *teaches* 中的所有元组都要进行组合，即使是那些代表不同教师的元组。其结果可能是一个非常庞大的关系，创建这样的笛卡儿积通常是没有意义的。

反之，**where** 子句中的谓词用来限制笛卡儿积所建立的组合，只留下那些对所需答案有意义的组合。我们希望有个涉及 *instructor* 和 *teaches* 的查询，它把 *instructor* 中的特定元组 *t* 只与 *teaches* 中表示跟 *t* 相同教师的元组进行组合。也就是说，我们希望把 *teaches* 元组只和具有相同 *ID* 值的 *instructor* 元组进行匹配。下面的SQL查询满足这个条件，从这些匹配元组中输出教师名和课程标识。

68
69

```
select name, course_id
from instructor, teaches
where instructor. ID = teaches. ID;
```

⊖ 注意为了减小图3-6中表的宽度，我们把 *instructor. ID* 更名为 *inst. ID*。

注意上述查询只输出讲授了课程的教师，不会输出那些没有讲授任何课程的教师。如果我们希望输出那样的元组，可以使用一种被称作外连接的运算，外连接将在 4.1.2 节讲述。

如果 *instructor* 关系如图 2-1 所示，*teaches* 关系如图 2-7 所示，那么前述查询的结果关系如图 3-7 所示。注意教师 Gold、Califieri 和 Singh，由于他们没有讲授任何课程，就不出现在上述结果中。

如果我们只希望找出 Computer Science 系的教师名和课程标识，我们可以在 **where** 子句中增加另外的谓词，如下所示：

```
select name, course_id
from instructor, teaches
where instructor.ID = teaches.ID and instructor.dept_name = 'Comp. Sci.';
```

注意既然 *dept_name* 属性只出现在 *instructor* 关系中，我们在上述查询中可以只使用 *dept_name* 来替代 *instructor.dept_name*。

name	*course_id*
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101
Crick	BIO-301
Brandt	CS-190
Brandt	CS-190
Brandt	CS-319
Kim	EE-181

图 3-7 “对于大学中所有讲授课程的教师，找出他们的姓名以及所讲述的所有课程标识”的结果

通常说来，一个 SQL 查询的含义可以理解如下：

1. 为 **from** 子句中列出的关系产生笛卡儿积。
2. 在步骤 1 的结果上应用 **where** 子句中指定的谓词。
3. 对于步骤 2 结果中的每个元组，输出 **select** 子句中指定的属性(或表达式的结果)。

上述步骤的顺序有助于明白一个 SQL 查询的结果应该是什么样的，而不是这个结果是怎样被执行的。在 SQL 的实际实现中不会执行这种形式的查询，它会通过(尽可能)只产生满足 **where** 子句谓词的笛卡儿积元素来进行优化执行。我们在后面第 12 章和第 13 章学习那样的实现技术。

当书写查询时，需要小心设置合适的 **where** 子句条件。如果在前述 SQL 查询中省略 **where** 子句条件，就会输出笛卡儿积，那是一个巨大的关系。对于图 2-1 中的 *instructor* 样本关系和图 2-7 中的 *teaches* 样本关系，它们的笛卡儿积具有 $12 \times 13 = 156$ 个元组，比我们在书中能显示的还要多！在更糟的情况下，假设我们有比图中所示样本关系更现实的教师数量，比如 200 个教师。假使每位教师讲授 3 门课程，那么我们在 *teaches* 关系中就有 600 个元组。这样上述迭代过程会产生出 $200 \times 600 = 120\,000$ 个元组作为结果。

3.3.3 自然连接

在我们的查询示例中，需要从 *instructor* 和 *teaches* 表中组合信息，匹配条件是需要 *instructor.ID* 等于 *teaches.ID*。这是在两个关系中具有相同名称的所有属性。实际上这是一种通用的情况，也就是说，**from** 子句中的匹配条件在最通常的情况下需要在所有匹配名称的属性上相等。

为了在这种通用情况下简化 SQL 编程者的工作，SQL 支持一种被称作**自然连接**的运算，下面我们就来讨论这种运算。事实上 SQL 还支持几种另外的方式使得来自两个或多个关系的信息可以被**连接**(join)起来。我们已经见过怎样利用笛卡儿积和 **where** 子句谓词来连接来自多个关系的信息。连接来自多个关系信息的其他方式在 4.1 节介绍。

自然连接(natural join)运算作用于两个关系，并产生一个关系作为结果。不同于两个关系上的笛卡儿积，它将第一个关系的每个元组与第二个关系的所有元组都进行连接；自然连接只考虑那些在两个关系模式中都出现的属性上取值相同的元组对。因此，回到 *instructor* 和 *teaches* 关系的例子上，*instructor* 和 *teaches* 的**自然连接**计算中只考虑这样的元组对：来自 *instructor* 的元组和来自 *teaches* 的元组在共同属性 *ID* 上的取值相同。

70
|
71

结果关系如图 3-8 所示，只有 13 个元组，它们给出了关于每个教师以及该教师实际讲授的课程的信息。注意我们并没有重复列出那些在两个关系模式中都出现的属性，这样的属性只出现一次。还要注意列出属性的顺序：先是两个关系模式中的共同属性，然后是那些只出现在第一个关系模式中的属性，最后是那些只出现在第二个关系模式中的属性。

考虑查询“对于大学中所有讲授课程的教师，找出他们的姓名以及所讲述的所有课程标识”，此前我们曾把该查询写为：

ID	name	dept_name	salary	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

图 3-8 *instructor* 关系和 *teaches* 关系的自然连接

```
select name, course_id
from instructor, teaches
where instructor.ID = teaches.ID;
```

该查询可以用 SQL 的自然连接运算更简洁地写作：

```
select name, course_id
from instructor natural join teaches;
```

以上两个查询产生相同的结果。

正如我们此前所见，自然连接运算的结果是关系。从概念上讲，**from** 子句中的"*instructor* **natural join** *teaches*"表达式可以替换成执行该自然连接后所得到的关系。[⊖]然后在这个关系上执行 **where** 和 **select** 子句，就如我们在前面 3.3.2 节所看到的那样。 72

在一个 SQL 查询的 **from** 子句中，可以用自然连接将多个关系结合在一起，如下所示：

```
select A1, A2, ···, An
from r1 natural join r2 natural join ···natural join rm
where P;
```

更为一般地，**from** 子句可以为如下形式：

```
from E1, E2, ···, En
```

其中每个 E_i 可以是单个关系或一个包含自然连接的表达式。例如，假设我们要回答查询"列出教师的名字以及他们所讲授课程的名称"。此查询可以用 SQL 写为：

```
select name, title
from instructor natural join teaches, course
where teaches.course_id = course.course_id;
```

先计算 *instructor* 和 *teaches* 的自然连接，正如我们此前所见，再计算该结果和 *course* 的笛卡儿积，**where** 子句从这个结果中提取出这样的元组：来自连接结果的课程标识与来自 *course* 关系的课程标识相匹配。注意 **where** 子句中的 *teaches.course_id* 表示自然连接结果中的 *course_id* 域，因为该域最终来自 *teaches* 关系。

相反，下面的 SQL 查询不会计算出相同的结果：

```
select name, title
from instructor natural join teaches natural join course;
```

为了说明原因，注意 *instructor* 和 *teaches* 的自然连接包括属性(*ID*, *name*, *dept_name*, *salary*, *course_id*, *sec_id*)，而 *course* 关系包含的属性是(*course_id*, *title*, *dept_name*, *credits*)。作为这二者自然连接的结果，

⊖ 其结果是不可能用包含了原始关系名的属性名来指代自然连接结果中的属性，例如 *instructor.name* 或 *teaches.course_id*，但是我们可以使用诸如 *name* 和 *course_id* 那样的属性名，而不带关系名。

需要来自这两个输入的元组既要在属性 *dept_name* 上取值相同，还要在 *course_id* 上取值相同。该查询将忽略所有这样的(教师姓名，课程名称)对：其中教师所讲授的课程不是他所在系的课程。而前一个查询会正确输出这样的对。

为了发扬自然连接的优点，同时避免不必要的相等属性带来的危险，SQL 提供了一种自然连接的构造形式，允许用户来指定需要哪些列相等。下面的查询说明了这个特征：

```
select name, title
from (instructor natural join teaches) join course using (course_id);
```

join…using 运算中需要给定一个属性名列表，其两个输入中都必须具有指定名称的属性。考虑运算 r_1 **join** r_2 **using**(A_1, A_2)，它与 r_1 和 r_2 的**自然连接**类似，只不过在 $t_1.A_1 = t_2.A_1$ 并且 $t_1.A_2 = t_2.A_2$ 成立的前提下，来自 r_1 的元组 t_1 和来自 r_2 的元组 t_2 就能匹配，即使 r_1 和 r_2 都具有名为 A_3 的属性，也不需要 $t_1.A_3 = t_2.A_3$ 成立。

这样，在前述 SQL 查询中，**连接**构造允许 *teaches.dept_name* 和 *course.dept_name* 是不同的，该 SQL 查询给出了正确的答案。

3.4 附加的基本运算

SQL 中还支持几种附加的基本运算。

3.4.1 更名运算

重新考察我们此前使用过的查询：

```
select name, course_id
from instructor, teaches
where instructor.ID = teaches.ID;
```

此查询的结果是一个具有下列属性的关系：

name, *course_id*

结果中的属性名来自 **from** 子句中关系的属性名。

但我们不能总是用这个方法派生名字，其原因有几点：首先，**from** 子句的两个关系中可能存在同名属性，在这种情况下，结果中就会出现重复的属性名；其次，如果我们在 **select** 子句中使用算术表达式，那么结果属性就没有名字；再次，尽管如上例所示，属性名可以从基关系导出，但我们也许想要改变结果中的属性名字。因此，SQL 提供了一个重命名结果关系中属性的方法。即使用如下形式的 **as** 子句：

73
|
74

old-name **as** *new-name*

as 子句既可出现在 **select** 子句中，也可出现在 **from** 子句中。⊖

例如，如果我们想用名字 *instructor_name* 来代替属性名 *name*，我们可以重写上述查询如下：

```
select name as instructor_name, course_id
from instructor, teaches
where instructor.ID = teaches.ID;
```

as 子句在重命名关系时特别有用。重命名关系的一个原因是把一个长的关系名替换成短的，这样在查询的其他地方使用起来就更为方便。为了说明这一点，我们重写查询“对于大学中所有讲授课程的教师，找出他们的姓名以及所讲述的所有课程标识”：

```
select T.name, S.course_id
from instructor as T, teaches as S
where T.ID = S.ID;
```

⊖ SQL 的早期版本不包括关键字 **as**。其结果是在一些 SQL 实现中，特别是 Oracle 中，不允许在 from 子句中出现关键字 **as**。在 Oracle 的 from 子句中，“*old-name* **as** *new-name*”被写作“*old-name new-name*”。在 **select** 子句中允许使用关键字 **as** 来重命名属性，但它是可选项，在 Oracle 中可以省略。

重命名关系的另一个原因是为了适用于需要比较同一个关系中的元组的情况。为此我们需要把一个关系跟它自身进行笛卡儿积运算，如果不重命名的话，就不可能把一个元组与其他元组区分开来。假设我们希望写出查询："找出满足下面条件的所有教师的姓名，他们的工资至少比 Biology 系某一个教师的工资要高"，我们可以写出这样的 SQL 表达式：

```
select distinct T. name
from instructor as T, instructor as S
where T. salary > S. salary and S. dept_name = 'Biology';
```

注意我们不能使用 *instructor. salary* 这样的写法，因为这样并不清楚到底是希望引用哪一个 *instructor*。

在上述查询中，*T* 和 *S* 可以被认为是 *instructor* 关系的两个拷贝，但更准确地说是被声明为 *instructor* 关系的别名，也就是另外的名字。像 *T* 和 *S* 那样被用来重命名关系的标识符在 SQL 标准中被称作**相关名称**(correlation name)，但通常也被称作**表别名**(table alias)，或者**相关变量**(correlation variable)，或者**元组变量**(tuple variable)。

注意用文字表达上述查询更好的方式是："找出满足下面条件的所有教师的姓名，他们比 Biology 系教师的最低工资要高"。我们早先的表述更符合我们所写的 SQL，但后面的表述更直观，事实上它可以直接用 SQL 来表达，正如我们将在 3.8.2 节看到的那样。

3.4.2 字符串运算

SQL 使用一对单引号来标示字符串，例如 'Computer'。如果单引号是字符串的组成部分，那就用两个单引号字符来表示，如字符串"it's right"可表示为"it''s right"。

在 SQL 标准中，字符串上的相等运算是大小写敏感的，所以表达式"'comp. sci.' = 'Comp. Sci.'"的结果是假。然而一些数据库系统，如 MySQL 和 SQL Server，在匹配字符串时并不区分大小写，所以在这些数据库中"'comp. sci.' = 'Comp. Sci.'"的结果可能是真。然而这种默认方式是可以在数据库级或特定属性级被修改的。

SQL 还允许在字符串上有多种函数，例如串联(使用"||")、提取子串、计算字符串长度、大小写转换(用 **upper**(*s*)将字符串 *s* 转换为大写或用 **lower**(*s*)将字符串 *s* 转换为小写)、去掉字符串后面的空格(使用 **trim**(*s*))，等等。不同数据库系统所提供的字符串函数集是不同的，请参阅你的数据库系统手册来获得它所支持的实际字符串函数的详细信息。

在字符串上可以使用 **like** 操作符来实现模式匹配。我们使用两个特殊的字符来描述模式：

- 百分号(%)：匹配任意子串。
- 下划线(_)：匹配任意一个字符。

模式是大小写敏感的，也就是说，大写字符与小写字符不匹配，反之亦然。为了说明模式匹配，考虑下列例子：

- 'Intro%' 匹配任何以"Intro"打头的字符串。
- '% Comp%' 匹配任何包含"Comp"子串的字符串，例如 'Intro. to Computer Science' 和 'Computational Biology'。
- '___' 匹配只含三个字符的字符串。
- '___%' 匹配至少含三个字符的字符串。

在 SQL 中用比较运算符 **like** 来表达模式。考虑查询"找出所在建筑名称中包含子串 'Watson' 的所有系名"，该查询的写法如下：

```
select dept_name
from department
where building like '%Watson%';
```

75
76

为使模式中能够包含特殊模式的字符(即% 和_)，SQL 允许定义转义字符。转义字符直接放在特殊字符的前面，表示该特殊字符被当成普通字符。我们在 **like** 比较运算中使用 **escape** 关键词来定义转义字符。为了说明这一用法，考虑以下模式，它使用反斜线(\)作为转义字符：

- **like** 'ab\%cd%' **escape** '\' 匹配所有以"ab%cd"开头的字符串。

• **like** 'ab\\cd%' **escape** '\' 匹配所有以"ab\cd"开头的字符串。

SQL 允许使用 **not like** 比较运算符搜寻不匹配项。一些数据库还提供 **like** 运算的变体，不区分大小写。

在 SQL:1999 中还提供 **similar to** 操作，它具备比 **like** 运算更强大的模式匹配能力。它的模式定义语法类似于 UNIX 中的正则表达式。

3.4.3 select 子句中的属性说明

星号"*"可以用在 **select** 子句中表示"所有的属性"，因而，如下查询的 **select** 子句中使用 *instructor.** ：

```
select instructor.*
from instructor, teaches
where instructor.ID = teaches.ID;
```

表示 *instructor* 中的所有属性都被选中。形如 **select** * 的 select 子句表示 **from** 子句结果关系的所有属性都被选中。

3.4.4 排列元组的显示次序

SQL 为用户提供了一些对关系中元组显示次序的控制。**order by** 子句就可以让查询结果中元组按排列顺序显示。为了按字母顺序列出在 Physics 系的所有教师，我们可以这样写：

```
select name
from instructor
where dept_name = 'Physics'
order by name;
```

order by 子句默认使用升序。要说明排序顺序，我们可以用 **desc** 表示降序，或者用 **asc** 表示升序。此
77 外，排序可在多个属性上进行。假设我们希望按 *salary* 的降序列出整个 *instructor* 关系。如果有几位教师的工资相同，就将它们按姓名升序排列。我们用 SQL 将该查询表示如下：

```
select *
from instructor
order by salary desc, name asc;
```

3.4.5 where 子句谓词

为了简化 **where** 子句，SQL 提供 **between** 比较运算符来说明一个值是小于或等于某个值，同时大于或等于另一个值的。如果我们想找出工资在 90 000 美元和 100 000 美元之间的教师的姓名，我们可以使用 **between** 比较运算符，如下所示：

```
select name
from instructor
where salary between 90000 and 100000;
```

它可以取代

```
select name
from instructor
where salary <= 100000 and salary >= 90000;
```

类似地，我们还可以使用 **not between** 比较运算符。

我们可以扩展前面看到过的查找教师名以及课程标识的查询，但考虑更复杂的情况，要求教师是生物系的："查找 Biology 系讲授了课程的所有教师的姓名和他们所讲授的课程"。为了写出这样的查询，我们可以在前面看到过的两个 SQL 查询的任意一个的基础上进行修改、在 **where** 子句中增加一个额外的条件。我们下面给出修改后的不使用自然连接的 SQL 查询形式：

```
select name, course_id
from instructor, teaches
where instructor.ID = teaches.ID and dept_name = 'Biology';
```

SQL 允许我们用记号 $(v_1, v_2, \cdots, v_n)$ 来表示一个分量值分别为 $v_1, v_2, \cdots, v_n$ 的 n 维元组。在元组

上可以运用比较运算符，按字典顺序进行比较运算。例如，$(a_1, a_2) <= (b_1, b_2)$在$a_1 <= b_1$且$a_2 <= b_2$时为真。类似地，当两个元组在所有属性上相等时，它们是相等的。这样，前述查询可被重写为如下形式：[⊖] 78

```
select name, course_id
from instructor, teaches
where (instructor.ID, dept_name) = (teaches.ID, 'Biology');
```

3.5 集合运算

SQL 作用在关系上的 **union**、**intersect** 和 **except** 运算对应于数学集合论中的∪、∩和－运算。我们现在来构造包含在两个集合上使用 **union**、**intersect** 和 **except** 运算的查询。

- 在 2009 年秋季学期开设的所有课程的集合：

```
select course_id
from section
where semester = 'Fall' and year = 2009;
```

- 在 2010 年春季学期开设的所有课程的集合：

```
select course_id
from section
where semester = 'Spring' and year = 2010;
```

在我们后面的讨论中，将用 *c1* 和 *c2* 分别指代包含以上查询结果的两个关系，并在图 3-9 和图 3-10 中给出作用在如图 2-6 所示的 *section* 关系上的查询结果。注意 *c2* 包含两个对应于 *course_id* 为 CS-319 的元组，因为该课程有两个课程段在 2010 年春季开课。

course_id
CS-101
CS-347
PHY-101

图 3-9 *c1* 关系，列出 2009 年秋季开设的课程

course_id
CS-101
CS-315
CS-319
CS-319
FIN-201
HIS-351
MU-199

图 3-10 *c2* 关系，列出 2010 年春季开设的课程

3.5.1 并运算

为了找出在 2009 年秋季开课，或者在 2010 年春季开课或两个学期都开课的所有课程，我们可写查询语句：[⊜]

```
(select course_id
 from section
 where semester = 'Fall' and year = 2009)
 union
(select course_id
 from section
 where semester = 'Spring' and year = 2010);
```

与 **select** 子句不同，**union** 运算自动去除重复。这样，在如图 2-6 所示的 *section* 关系中，2010 年春季开设 CS-319 的两个课程段，CS-101 在 2009 年秋季和 2010 年秋季学期各开设一个课程段，CS-101 和 CS-319 在结果中只出现一次，如图 3-11 所示。

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

图 3-11 *c1* **union** *c2* 的结果

⊖ 尽管这是 SQL-92 标准的一部分，但某些 SQL 实现中可能不支持这种语法。

⊜ 我们在每条 **select-from-where** 语句上使用的括号是可省略的，但易于阅读。

如果我们想保留所有重复，就必须用 **union all** 代替 **union**：

```
(select course_id
 from section
 where semester = 'Fall' and year = 2009)
 union all
(select course_id
 from section
 where semester = 'Spring' and year = 2010);
```

在结果中的重复元组数等于在 $c1$ 和 $c2$ 中出现的重复元组数的和。因此在上述查询中，每个 CS-319 和 CS-101 都将被列出两次。作为一个更深入的例子，如果存在这样一种情况：ECE-101 在 2009 年秋季学期开设 4 个课程段，在 2010 年春季学期开设 2 个课程段，那么在结果中将有 6 个 ECE-101 元组。

3.5.2 交运算

为了找出在 2009 年秋季和 2010 年春季同时开课的所有课程的集合，我们可写出：

```
(select course_id
 from section
 where semester = 'Fall' and year = 2009)
intersect
(select course_id
 from section
 where semester = 'Spring' and year = 2010);
```

结果关系如图 3-12 所示，它只包括一个 CS-101 元组。**intersect** 运算自动去除重复。例如，如果存在这样的情况：ECE-101 在 2009 年秋季学期开设 4 个课程段，在 2010 年春季学期开设 2 个课程段，那么在结果中只有 1 个 ECE-101 元组。

如果我们想保留所有重复，就必须用 **intersect all** 代替 **intersect**：

```
(select course_id
 from section
 where semester = 'Fall' and year = 2009)
intersect all
(select course_id
 from section
 where semester = 'Spring' and year = 2010);
```

course_id
CS-101

图3-12 $c1$ intersect $c2$ 的结果

在结果中出现的重复元组数等于在 $c1$ 和 $c2$ 中出现的重复次数里最少的那个。例如，如果 ECE-101 在 2009 年秋季学期开设 4 个课程段，在 2010 年春季学期开设 2 个课程段，那么在结果中有 2 个 ECE-101 元组。

3.5.3 差运算

为了找出在 2009 年秋季学期开课但不在 2010 年春季学期开课的所有课程，我们可写出：

```
(select course_id
 from section
 where semester = 'Fall' and year = 2009)
except
(select course_id
 from section
 where semester = 'Spring' and year = 2010);
```

该查询结果如图 3-13 所示。注意这正好是图 3-9 的 $c1$ 关系减去不出现的 CS-101 元组。**except** 运算[⊖]从其第一个输入中输出所有不出现在第二个输入中的元组，也即它执行集差操作。此运算在执行集差操作之前自动去除输入中的重复。例如，如果 ECE-101 在 2009 年秋季学期开设 4 个课程段，在 2010 年

⊖ 某些 SQL 实现，特别是 Oracle，使用关键词 **minus** 代替 **except**。

春季学期开设2个课程段，那么在 **except** 运算的结果中将没有 ECE-101 的任何拷贝。

如果我们想保留所有重复，就必须用 **except all** 代替 **except**：

```
(select course_id
 from section
 where semester = 'Fall' and year = 2009)
except all
(select course_id
 from section
 where semester = 'Spring' and year = 2010);
```

course_id
CS-347
PHY-101

图 3-13 *c*1 except *c*2 的结果

结果中的重复元组数等于在 *c*1 中出现的重复元组数减去在 *c*2 中出现的重复元组数(前提是此差为正)。因此，如果 ECE-101 在2009年秋季学期开设4个课程段，在2010年春季学期开设2个课程段，那么在结果中有2个 ECE-101 元组。然而，如果 ECE-101 在2009年秋季学期开设2个或更少的课程段，在2010年春季学期开设2个课程段，那么在结果中将不存在 ECE-101 元组。

3.6 空值

空值给关系运算带来了特殊的问题，包括算术运算、比较运算和集合运算。

如果算术表达式的任一输入为空，则该算术表达式(涉及诸如 +、-、* 或/)结果为空。例如，如果查询中有一个表达式是 *r.A* + 5，并且对于某个特定的元组，*r.A* 为空，那么对此元组来说，该表达式的结果也为空。

涉及空值的比较问题更多。例如，考虑比较运算"1 < **null**"。因为我们不知道空值代表的是什么，所以说上述比较为真可能是错误的。但是说上述比较为假也可能是错误的，如果我们认为比较为假，那么"**not** (1 < **null**)"就应该为真，但这是没有意义的。因而 SQL 将涉及空值的任何比较运算的结果视为 **unknown**(既不是谓词 **is null**，也不是 **is not null**，我们在本节的后面介绍这两个谓词)。这创建了除 *true* 和 *false* 之外的第三个逻辑值。

由于在 **where** 子句的谓词中可以对比较结果使用诸如 **and**、**or** 和 **not** 的布尔运算，所以这些布尔运算的定义也被扩展到可以处理 **unknown** 值。

- **and**：*true* **and** *unknown* 的结果是 *unknown*，*false* **and** *unknown* 结果是 *false*，*unknown* **and** *unknown* 的结果是 *unknown*。
- **or**：*true* **or** *unknown* 的结果是 *true*，*false* **or** *unknown* 结果是 *unknown*，*unknown* **or** *unknown* 结果是 *unknown*。
- **not**：**not** *unknown* 的结果是 *unknown*。

可以验证，如果 *r.A* 为空，那么"1 < *r.A*"和"**not** (1 < *r.A*)"结果都是 unknown。

如果 **where** 子句谓词对一个元组计算出 **false** 或 **unknown**，那么该元组不能被加入到结果集中。

79
83

SQL 在谓词中使用特殊的关键词 **null** 测试空值。因而为找出 *instructor* 关系中 *salary* 为空值的所有教师，我们可以写成：

```
select name
from instructor
where salary is null;
```

如果谓词 **is not null** 所作用的值非空，那么它为真。

某些 SQL 实现还允许我们使用子句 **is unknown** 和 **is not unknown** 来测试一个表达式的结果是否为 unknown，而不是 true 或 false。

当一个查询使用 **select distinct** 子句时，重复元组将被去除。为了达到这个目的，当比较两个元组对应的属性值时，如果这两个值都是非空并且值相等，或者都是空，那么它们是相同的。所以诸如 |('A', null), ('A', null)| 这样的两个元组拷贝被认为是相同的，即使在某些属性上存在空值。使用 **distinct** 子句会保留这样的相同元组的一份拷贝。注意上述对待空值的方式与谓词中对待空值的方式是不同的，在谓词中"null = null"会返回 unknown，而不是 true。

如果元组在所有属性上的取值相等，那么它们就被当作相同元组，即使某些值为空。上述方式还

应用于集合的并、交和差运算。

3.7 聚集函数

聚集函数是以值的一个集合(集或多重集)为输入、返回单个值的函数。SQL 提供了五个固有聚集函数:

- 平均值: **avg**。
- 最小值: **min**。
- 最大值: **max**。
- 总和: **sum**。
- 计数: **count**。

84 **sum** 和 **avg** 的输入必须是数字集,但其他运算符还可作用在非数字数据类型的集合上,如字符串。

3.7.1 基本聚集

考虑查询"找出 Computer Science 系教师的平均工资"。我们书写该查询如下:

```
select avg (salary)
from instructor
where dept_name = 'Comp. Sci.';
```

该查询的结果是一个具有单属性的关系,其中只包含一个元组,这个元组的数值对应 Computer Science 系教师的平均工资。数据库系统可以给结果关系的属性一个任意的名字,该属性是由聚集产生的。然而,我们可以用 **as** 子句给属性赋个有意义的名称,如下所示:

```
select avg (salary) as avg_salary
from instructor
where dept_name = 'Comp. Sci.';
```

在图 2-1 的 *instructor* 关系中,Computer Science 系的工资值是 75 000 美元、65 000 美元和 92 000 美元,平均工资是 232 000/3 = 77 333.33 美元。

在计算平均值时保留重复元组是很重要的。假设 Computer Science 系增加了第四位教师,其工资正好是 75 000 美元。如果去除重复的话,我们会得到错误的答案(232 000/4 = 58 000 美元),而正确的答案是 76 750 美元。

有些情况下在计算聚集函数前需先删掉重复元组。如果我们确实想删除重复元组,可在聚集表达式中使用关键词 **distinct**。比方有这样一个查询示例"找出在 2010 年春季学期讲授一门课程的教师总数",在该例中不论一个教师讲授了几个课程段,他只应被计算一次。所需信息包含在 *teaches* 关系中,我们书写该查询如下:

```
select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2010;
```

我们经常使用聚集函数 **count** 计算一个关系中元组的个数。SQL 中该函数的写法是 **count**(*)。因此,要找出 *course* 关系中的元组数,可写成:

```
select count(*)
from course;
```

85 由于在 *ID* 前面有关键字 **distinct**,所以即使某位教师教了不止一门课程,在结果中他也仅被计数一次。

SQL 不允许在用 **count**(*)时使用 **distinct**。在用 **max** 和 **min** 时使用 **distinct** 是合法的,尽管结果并无差别。我们可以使用关键词 **all** 替代 **distinct** 来说明保留重复元组,但是,既然 **all** 是默认的,就没必要这么做了。

3.7.2 分组聚集

有时候我们不仅希望将聚集函数作用在单个元组集上,而且也希望将其作用到一组元组集上;在 SQL 中可用 **group by** 子句实现这个愿望。**group by** 子句中给出的一个或多个属性是用来构造分组的。

在 **group by** 子句中的所有属性上取值相同的元组将被分在一个组中。

作为示例，考虑查询“找出每个系的平均工资”，该查询书写如下：

```
select dept_name, avg(salary) as avg_salary
from instructor
group by dept_name;
```

图 3-14 给出了 *instructor* 关系中的元组按照 *dept_name* 属性进行分组的情况，分组是计算查询结果的第一步。在每个分组上都要进行指定的聚集计算，查询结果如图 3-15 所示。

相反，考虑查询“找出所有教师的平均工资”。我们把此查询写做如下形式：

```
select avg (salary)
from instructor;
```

在这里省略了 **group by** 子句，因此整个关系被当作是一个分组。

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

图 3-14 *instructor* 关系的元组按照 *dept_name* 属性分组

86

作为在元组分组上进行聚集操作的另一个例子，考虑查询“找出每个系在 2010 年春季学期讲授一门课程的教师人数”。有关每位教师在每个学期讲授每个课程段的信息在 *teaches* 关系中。但是，这些信息需要与来自 *instructor* 关系的信息进行连接，才能够得到每位教师所在的系名。这样，我们把此查询写做如下形式：

```
select dept_name, count (distinct ID) as instr_count
from instructor natural join teaches
where semester = 'Spring' and year = 2010
group by dept_name;
```

其结果如图 3-16 所示。

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

图 3-15 查询“找出每个系的平均工资”的结果关系

dept_name	*instr_count*
Comp. Sci.	3
Finance	1
History	1
Music	1

图 3-16 查询“找出每个系在 2010 年春季学期讲授一门课程的教师人数”的结果关系

当 SQL 查询使用分组时，一个很重要的事情是需要保证出现在 **select** 语句中但没有被聚集的属性只能是出现在 **group by** 子句中的那些属性。换句话说，任何没有出现在 **group by** 子句中的属性如果出现在 **select** 子句中的话，它只能出现在聚集函数内部，否则这样的查询就是错误的。例如，下述查询是错误的，因为 *ID* 没有出现在 **group by** 子句中，但它出现在了 **select** 子句中，而且没有被聚集：

```
/* 错误查询 */
select dept_name, ID, avg (salary)
from instructor
group by dept_name;
```

在一个特定分组（通过 *dept_name* 定义）中的每位教师都有一个不同的 *ID*，既然每个分组只输出一个元组，那就无法确定选哪个 *ID* 值作为输出。其结果是，SQL 不允许这样的情况出现。

3.7.3 having 子句

有时候，对分组限定条件比对元组限定条件更有用。例如，我们也许只对教师平均工资超过 42 000美元的系感兴趣。该条件并不针对单个元组，而是针对 **group by** 子句构成的分组。为表达这样的查询，我们使用 SQL 的 **having** 子句。**having** 子句中的谓词在形成分组后才起作用，因此可以使用聚集函数。我们用 SQL 表达该查询如下：

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name
having avg (salary) > 42000;
```

其结果如图 3-17 所示。

与 **select** 子句的情况类似，任何出现在 **having** 子句中，但没有被聚集的属性必须出现在 **group by** 子句中，否则查询就被当成是错误的。

dept_name	*avg_salary*
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

图 3-17 查询"找出系平均工资超过 42 000 美元的那些系中教师的平均工资"的结果关系

包含聚集、**group by** 或 **having** 子句的查询的含义可通过下述操作序列来定义：

1. 与不带聚集的查询情况类似，最先根据 **from** 子句来计算出一个关系。
2. 如果出现了 **where** 子句，**where** 子句中的谓词将应用到 **from** 子句的结果关系上。
3. 如果出现了 **group by** 子句，满足 **where** 谓词的元组通过 **group by** 子句形成分组。如果没有 **group by** 子句，满足 **where** 谓词的整个元组集被当作一个分组。
4. 如果出现了 **having** 子句，它将应用到每个分组上；不满足 **having** 子句谓词的分组将被抛弃。
5. **select** 子句利用剩下的分组产生出查询结果中的元组，即在每个分组上应用聚集函数来得到单个结果元组。

为了说明在同一个查询中同时使用 **having** 子句和 **where** 子句的情况，我们考虑查询"对于在 2009 年讲授的每个课程段，如果该课程段有至少 2 名学生选课，找出选修该课程段的所有学生的总学分(*tot_cred*)的平均值"。

```
select course_id, semester, year, sec_id, avg (tot_cred)
from takes natural join student
where year = 2009
group by course_id, semester, year, sec_id
having count (ID) >= 2;
```

注意上述查询需要的所有信息来自关系 *takes* 和 *student*，尽管此查询是关于课程段的，却并不需要与 *section* 进行连接。

3.7.4 对空值和布尔值的聚集

空值的存在给聚集运算的处理带来了麻烦。例如，假设 *instructor* 关系中有些元组在 *salary* 上取空值。考虑以下计算所有工资总额的查询：

```
select sum(salary)
from instructor;
```

由于一些元组在 *salary* 上取空值，上述查询待求和的值中就包含了空值。SQL 标准并不认为总和本身为 *null*，而是认为 **sum** 运算符应忽略输入中的 *null* 值。

总而言之，聚集函数根据以下原则处理空值：除了 **count**(*)外所有的聚集函数都忽略输入集合中的空值。由于空值被忽略，有可能造成参加函数运算的输入值集合为空集。规定空集的 **count** 运算值为 0，其他所有聚集运算在输入为空集的情况下返回一个空值。在一些更复杂的 SQL 结构中空值的影响会更难以琢磨。

87
89

在 SQL:1999 中引入了**布尔**(boolean)数据类型，它可以取 **true**、**false**、**unknown** 三个值。有两个聚集函数：**some** 和 **every**，其含义正如直观意义一样，可用来处理**布尔**(boolean)值的集合。

3.8 嵌套子查询

SQL 提供嵌套子查询机制。子查询是嵌套在另一个查询中的 **select-from-where** 表达式。子查询嵌套在 **where** 子句中，通常用于对集合的成员资格、集合的比较以及集合的基数进行检查。从 3.8.1 到 3.8.4 节我们学习在 **where** 子句中嵌套子查询的用法。在 3.8.5 节我们学习在 **from** 子句中嵌套的子查询。在 3.8.7 节我们将看到一类被称作标量子查询的子查询是如何出现在一个表达式所返回的单个值可以出现的任何地方的。

3.8.1 集合成员资格

SQL 允许测试元组在关系中的成员资格。连接词 **in** 测试元组是否是集合中的成员，集合是由 **select** 子句产生的一组值构成的。连接词 **not in** 则测试元组是否不是集合中的成员。

作为示例，考虑查询“找出在 2009 年秋季和 2010 年春季学期同时开课的所有课程”。先前，我们通过对两个集合进行交运算来书写该查询，这两个集合分别是：2009 年秋季开课的课程集合与 2010 年春季开课的课程集合。现在我们采用另一种方式，查找在 2009 年秋季开课的所有课程，看它们是否也是 2010 年春季开课的课程集合中的成员。很明显，这种方式得到的结果与前面相同，但我们可以用 SQL 中的 **in** 连接词书写该查询。我们从找出 2010 年春季开课的所有课程开始，写出子查询：

```
(select course_id
 from section
 where semester = 'Spring' and year = 2010)
```

然后我们需要从子查询形成的课程集合中找出那些在 2009 年秋季开课的课程。为完成此项任务可将子查询嵌入外部查询的 **where** 子句中。最后的查询语句是：

```
select distinct course_id
from section
where semester = 'Fall' and year = 2009 and
    course_id in (select course_id
                  from section
                  where semester = 'Spring' and year = 2010);
```

90

该例说明了在 SQL 中可以用多种方法书写同一查询。这种灵活性是有好处的，因为它允许用户用最接近自然的方法去思考查询。我们将看到在 SQL 中有许多这样的冗余。

我们以与 **in** 结构类似的方式使用 **not in** 结构。例如，为了找出所有在 2009 年秋季学期开课，但不在 2010 年春季学期开课的课程，我们可写出：

```
select distinct course_id
from section
where semester = 'Fall' and year = 2009 and
    course_id not in (select course_id
                      from section
                      where semester = 'Spring' and year = 2010);
```

in 和 **not in** 操作符也能用于枚举集合。下面的查询找出既不叫“Mozart”，也不叫“Einstein”的教师的姓名：

```
select distinct name
from instructor
where name not in ('Mozart', 'Einstein');
```

在前面的例子中，我们是在单属性关系中测试成员资格。在 SQL 中测试任意关系的成员资格也是可以的。例如，我们可以这样来表达查询“找出(不同的)学生总数，他们选修了 *ID* 为 10101 的教师所讲授的课程段”：

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year) in (select course_id, sec_id, semester, year
                                              from teaches
                                              where teaches.ID = 10101);
```

3.8.2 集合的比较

作为一个说明嵌套子查询能够对集合进行比较的例子，考虑查询“找出满足下面条件的所有教师的姓名，他们的工资至少比 Biology 系某一个教师的工资要高”，在 3.4.1 节，我们将此查询写作：

```
select distinct T. name
from instructor as T, instructor as S
where T. salary > S. salary and S. dept_name = 'Biology';
```

91

但是 SQL 提供另外一种方式书写上面的查询。短语"至少比某一个要大"在 SQL 中用 **> some** 表示。此结构允许我们用一种更贴近此查询的文字表达的形式重写上面的查询：

```
select name
from instructor
where salary > some (select salary
                     from instructor
                     where dept_name = 'Biology');
```

子查询

```
(select salary
 from instructor
 where dept_name = 'Biology')
```

产生 Biology 系所有教师的所有工资值的集合。当元组的 *salary* 值至少比 Biology 系教师的所有工资值集合中某一成员高时，外层 **select** 的 **where** 子句中 **> some** 的比较为真。

SQL 也允许 **< some**，**<= some**，**>= some**，**= some** 和 **<> some** 的比较。作为练习，请验证 **= some** 等价于 **in**，然而 **<> some** 并不等价于 **not in**。⊖

现在我们稍微修改一下我们的查询。找出满足下面条件的所有教师的姓名，他们的工资值比 Biology 系每个教师的工资都高。结构 **> all** 对应于词组"比所有的都大"。使用该结构，我们写出查询如下：

```
select name
from instructor
where salary > all (select salary
                    from instructor
                    where dept_name = 'Biology');
```

类似于 **some**，SQL 也允许 **< all**，**<= all**，**>= all**，**= all** 和 **<> all** 的比较。作为练习，请验证 **<> all** 等价于 **not in**，但 **= all** 并不等价于 **in**。

作为集合比较的另一个例子，考虑查询"找出平均工资最高的系"。我们首先写一个查询来找出每个系的平均工资，然后把它作为子查询嵌套在一个更大的查询中，以找出那些平均工资大于等于所有系平均工资的系。

```
select dept_name
from instructor
group by dept_name
having avg (salary) >= all (select avg (salary)
                            from instructor
                            group by dept_name);
```

3.8.3 空关系测试

SQL 还有一个特性可测试一个子查询的结果中是否存在元组。**exists** 结构在作为参数的子查询非空时返回 **true** 值。使用 **exists** 结构，我们还能用另外一种方法书写查询"找出在 2009 年秋季学期和 2010 年春季学期同时开课的所有课程"：

```
select course_id
from section as S
where semester = 'Fall' and year = 2009 and
    exists (select *
```

⊖ 在 SQL 中关键词 **any** 同义于 **some**。早期 SQL 版本中仅允许使用 **any**，后来的版本为了避免和英语中 *any* 一词在语言上的混淆，又添加了另一个可选择的关键词 **some**。

```
    from section as T
    where semester = 'Spring' and year = 2010 and
        S.course_id = T.course_id);
```

上述查询还说明了SQL的一个特性，来自外层查询的一个相关名称(上述查询中的*S*)可以用在**where**子句的子查询中。使用了来自外层查询相关名称的子查询被称作**相关子查询**(correlated subquery)。

在包含了子查询的查询中，在相关名称上可以应用作用域规则。根据此规则，在一个子查询中只能使用此子查询本身定义的，或者在包含此子查询的任何查询中定义的相关名称。如果一个相关名称既在子查询中定义，又在包含该子查询的查询中定义，则子查询中的定义有效。这条规则类似于编程语言中通用的变量作用域规则。

我们可以用**not exists**结构测试子查询结果集中是否不存在元组。我们可以使用**not exists**结构模拟集合包含(即超集)操作：我们可将"关系*A*包含关系*B*"写成"**not exists**(*B* **except** *A*)"。(尽管**contains**运算符并不是当前SQL标准的一部分，但这一运算符曾出现在某些早期的关系系统中。)为了说明**not exists**操作符，考虑查询"找出选修了Biology系开设的所有课程的学生"。使用**except**结构，我们可以书写此查询如下：

92
93

```
select S.ID, S.name
from student as S
where not exists ((select course_id
                   from course
                   where dept_name = 'Biology')
                   except
                   (select T.course_id
                   from takes as T
                   where S.ID = T.ID));
```

这里，子查询

```
(select course_id
 from course
 where dept_name = 'Biology')
```

找出Biology系开设的所有课程集合。子查询

```
(select T.course_id
 from takes as T
 where S.ID = T.ID)
```

找出*S.ID*选修的所有课程。这样，外层**select**对每个学生测试其选修的所有课程集合是否包含Biology系开设的所有课程集合。

3.8.4 重复元组存在性测试

SQL提供一个布尔函数，用于测试在一个子查询的结果中是否存在重复元组。如果作为参数的子查询结果中没有重复的元组，**unique**结构[⊖]将返回**true**值。我们可以用**unique**结构书写查询"找出所有在2009年最多开设一次的课程"，如下所示：

```
select T.course_id
from course as T
where unique (select R.course_id
              from section as R
              where T.course_id = R.course_id and
                  R.year = 2009);
```

⊖ 此结构尚未被广泛实现。

注意如果某门课程不在2009年开设，那么子查询会返回一个空的结果，**unique** 谓词在空集上计算出真值。

在不使用 **unique** 结构的情况下，上述查询的一种等价表达方式是：

```
select T. course_id
from course as T
where 1 >= (select count(R. course_id)
            from section as R
            where T. course_id = R. course_id and
                R. year = 2009);
```

我们可以用 **not unique** 结构测试在一个子查询结果中是否存在重复元组。为了说明这一结构，考虑查询"找出所有在2009年最少开设两次的课程"，如下所示：

```
select T. course_id
from course as T
where not unique (select R. course_id
                  from section as R
                  where T. course_id = R. course_id and
                      R. year = 2009);
```

形式化地，对一个关系的 **unique** 测试结果为假的定义是，当且仅当在关系中存在着两个元组 t_1 和 t_2，且 $t_1 = t_2$。由于在 t_1 或 t_2 的某个域为空时，判断 $t_1 = t_2$ 为假，所以尽管一个元组有多个副本，只要该元组有一个属性为空，**unique** 测试就有可能为真。

3.8.5 from 子句中的子查询

SQL 允许在 **from** 子句中使用子查询表达式。在此采用的主要观点是：任何 **select-from-where** 表达式返回的结果都是关系，因而可以被插入到另一个 **select-from-where** 中任何关系可以出现的位置。

考虑查询"找出系平均工资超过42 000美元的那些系中教师的平均工资"。在3.7节我们使用了 **having** 子句来书写此查询。现在我们可以不用 **having** 子句来重写这个查询，而是通过如下这种在 **from** 子句中使用子查询的方式：

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
      from instructor
      group by dept_name)
where avg_salary > 42000;
```

94
/
95

该子查询产生的关系包含所有系的名字和相应的教师平均工资。子查询的结果属性可以在外层查询中使用，正如上例所示。

注意我们不需要使用 **having** 子句，因为 **from** 子句中的子查询计算出了每个系的平均工资，早先在 **having** 子句中使用的谓词现在出现在外层查询的 **where** 子句中。

我们可以用 **as** 子句给此子查询的结果关系起个名字，并对属性进行重命名。如下所示：

```
select dept_name, avg_salary
from (select dept_name, avg (salary)
      from instructor
      group by dept_name)
      as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

子查询的结果关系被命名为 *dept_avg*，其属性名是 *dept_name* 和 *avg_salary*。

很多(但并非全部)SQL实现都支持在 **from** 子句中嵌套子查询。请注意，某些SQL实现要求对每一个子查询结果关系都给一个名字，即使该名字从不被引用；Oracle允许对子查询结果关系命名(省略掉关键字 **as**)，但是不允许对关系中的属性重命名。

作为另一个例子，假设我们想要找出在所有系中工资总额最大的系。在此 **having** 子句是无能为力的，但我们可以用 **from** 子句中的子查询轻易地写出如下查询：

```
select max (tot_salary)
from (select dept_name, sum(salary)
      from instructor
      group by dept_name) as dept_total (dept_name, tot_salary);
```

我们注意到在 **from** 子句嵌套的子查询中不能使用来自 **from** 子句其他关系的相关变量。然而 SQL:2003 允许 **from** 子句中的子查询用关键词 **lateral** 作为前缀，以便访问 **from** 子句中在它前面的表或子查询中的属性。例如，如果我们想打印每位教师的姓名，以及他们的工资和所在系的平均工资，可书写查询如下：

```
select name, salary, avg_salary
from instructor I1, lateral (select avg(salary) as avg_salary
                             from instructor I2
                             where I2.dept_name = I1.dept_name);
```

没有 **lateral** 子句的话，子查询就不能访问来自外层查询的相关变量 *I1*。目前只有少数 SQL 实现支持 **lateral** 子句，比如 IBM DB2。 96

3.8.6 with 子句

with 子句提供定义临时关系的方法，这个定义只对包含 **with** 子句的查询有效。考虑下面的查询，它找出具有最大预算值的系。

```
with max_budget (value) as
    (select max(budget)
     from department)
select budget
from department, max_budget
where department.budget = max_budget.value;
```

with 子句定义了临时关系 *max_budget*，此关系在随后的查询中马上被使用了。**with** 子句是在 SQL:1999 中引入的，目前有许多(但并非所有)数据库系统都提供了支持。

我们也能用 **from** 子句或 **where** 子句中的嵌套子查询书写上述查询。但是，用嵌套子查询会使得查询语句晦涩难懂。**with** 子句使查询在逻辑上更加清晰，它还允许在一个查询内的多个地方使用视图定义。

例如，假设我们要查出所有工资总额大于所有系平均工资总额的系，我们可以利用如下 **with** 子句写出查询：

```
with dept_total (dept_name, value) as
    (select dept_name, sum(salary)
     from instructor
     group by dept_name),
    dept_total_avg(value) as
    (select avg(value)
     from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value >= dept_total_avg.value;
```

我们当然也可以不用 **with** 子句来建立等价的查询，但是那样会复杂很多，而且也不易看懂。作为练习，你可以把它转化为不用 **with** 子句的等价查询。

3.8.7 标量子查询

SQL 允许子查询出现在返回单个值的表达式能够出现的任何地方，只要该子查询只返回包含单个属性的单个元组；这样的子查询称为**标量子查询**(scalar subquery)。例如，一个子查询可以用到下面例子的 **select** 子句中，这个例子列出所有的系以及它们拥有的教师数： 97

```
select dept_name,
    (select count(*)
     from instructor
     where department. dept_name = instructor. dept_name)
     as num_instructors
from department;
```

上面例子中的子查询保证只返回单个值，因为它使用了不带 **group by** 的 **count(*)** 聚集函数。此例也说明了对相关变量的使用，即使用在外层查询的 **from** 子句中关系的属性，例如上例中的 *department. dept_name*。

标量子查询可以出现在 **select**、**where** 和 **having** 子句中。也可以不使用聚集函数来定义标量子查询。在编译时并非总能判断一个子查询返回的结果中是否有多个元组，如果在子查询被执行后其结果中有不止一个元组，则产生一个运行时错误。

注意从技术上讲标量子查询的结果类型仍然是关系，尽管其中只包含单个元组。然而，当在表达式中使用标量子查询时，它出现的位置是单个值出现的地方，SQL 就从该关系中包含单属性的单元组中取出相应的值，并返回该值。

3.9 数据库的修改

目前为止我们的注意力集中在对数据库的信息抽取上。现在我们将展示如何用 SQL 来增加、删除和修改信息。

3.9.1 删除

删除请求的表达与查询非常类似。我们只能删除整个元组，而不能只删除某些属性上的值。SQL 用如下语句表示删除：

```
delete from r
where P;
```

其中 P 代表一个谓词，r 代表一个关系。**delete** 语句首先从 r 中找出所有使 $P(t)$ 为真的元组 t，然后把它们从 r 中删除。如果省略 **where** 子句，则 r 中所有元组将被删除。

注意 **delete** 命令只能作用于一个关系。如果我们想从多个关系中删除元组，必须在每个关系上使用一条 **delete** 命令。**where** 子句中的谓词可以和 **select** 命令的 **where** 子句中的谓词一样复杂。在另一种
98 极端情况下，**where** 子句可以为空，请求

```
delete from instructor;
```

将删除 *instructor* 关系中的所有元组。*instructor* 关系本身仍然存在，但它变成空的了。

下面是 SQL 删除请求的一些例子：

- 从 *instructor* 关系中删除与 Finance 系教师相关的所有元组。

```
delete from instructor
where dept_name = 'Finance';
```

- 删除所有工资在 13 000 美元到 15 000 美元之间的教师。

```
delete from instructor
where salary between 13000 and 15000;
```

- 从 *instructor* 关系中删除所有这样的教师元组，他们在位于 Watson 大楼的系工作。

```
delete from instructor
where dept_name in (select dept_name
                    from department
                    where building = 'Watson');
```

此 **delete** 请求首先找出所有位于 Watson 大楼的系，然后将属于这些系的 *instructor* 元组全部删除。

注意，虽然我们一次只能从一个关系中删除元组，但是通过在 **delete** 的 **where** 子句中嵌套 **select-**

from-where，我们可以引用任意数目的关系。**delete** 请求可以包含嵌套的 **select**，该 **select** 引用待删除元组的关系。例如，假设我们想删除工资低于大学平均工资的教师记录，可以写出如下语句：

```
delete from instructor
where salary < (select avg (salary)
                from instructor);
```

该 **delete** 语句首先测试 *instructor* 关系中的每一个元组，检查其工资是否小于大学教师的平均工资。然后删除所有符合条件的元组，即所有低于平均工资的教师。在执行任何删除之前先进行所有元组的测试是至关重要的，因为若有些元组在其余元组未被测试前先被删除，则平均工资将会改变，这样 **delete** 的最后结果将依赖于元组被处理的顺序！ 99

3.9.2 插入

要往关系中插入数据，我们可以指定待插入的元组，或者写一条查询语句来生成待插入的元组集合。显然，待插入元组的属性值必须在相应属性的域中。同样，待插入元组的分量数也必须是正确的。

最简单的 **insert** 语句是单个元组的插入请求。假设我们想要插入的信息是 Computer Science 系开设的名为"Database Systems"的课程 CS-437，它有 4 个学分。我们可写成：

```
insert into course
    values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

在此例中，元组属性值的排列顺序和关系模式中属性排列的顺序一致。考虑到用户可能不记得关系属性的排列顺序，SQL 允许在 **insert** 语句中指定属性。例如，以下 SQL **insert** 语句与前述语句的功能相同。

```
insert into course (course_id, title, dept_name, credits)
    values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
insert into course (title, course_id, credits, dept_name)
    values ('Database Systems', 'CS-437', 4, 'Comp. Sci.');
```

更通常的情况是，我们可能想在查询结果的基础上插入元组。假设我们想让 Music 系每个修满 144 学分的学生成为 Music 系的教师，其工资为 18 000 美元。我们可写作：

```
insert into instructor
    select ID, name, dept_name, 18000
    from student
    where dept_name = 'Music' and tot_cred > 144;
```

和本节前面的例子不同的是，我们没有指定一个元组，而是用 **select** 选出一个元组集合。SQL 先执行这条 **select** 语句，求出将要插入到 *instructor* 关系中的元组集合。每个元组都有 *ID*、*name*、*dept_name*（Music）和工资（18 000 美元）。

在执行插入之前先执行完 **select** 语句是非常重要的。如果在执行 **select** 语句的同时执行插入动作，如果在 *student* 上没有主码约束的话，像 100

```
insert into student
    select *
    from student;
```

这样的请求就可能会插入无数元组。如果没有主码约束，上述请求会重新插入 *student* 中的第一个元组，产生该元组的第二份拷贝。由于这个副本现在是 *student* 中的一部分，**select** 语句可能找到它，于是第三份拷贝被插入到 *student* 中。第三份拷贝又可能被 **select** 语句发现，于是又插入第四份拷贝，如此等等，无限循环。在执行插入之前先完成 **select** 语句的执行可以避免这样的问题。这样，如果在 *student* 关系上没有主码约束，那么上述 **insert** 语句就只是把 *student* 关系中的每个元组都复制一遍。

在讨论 **insert** 语句时我们只考虑了这样的例子：待插入元组的每个属性都被赋了值。但是有可能待插入元组中只给出了模式中部分属性的值，那么其余属性将被赋空值，用 *null* 表示。考虑请求：

```
insert into student
    values ('3003', 'Green', 'Finance', null);
```

此请求所插入的元组代表了一个在 Finance 系、*ID* 为"3003"的学生，但其 *tot_cred* 值是未知的。考虑查询：

```
select ID
from student
where tot_cred > 45;
```

既然"3003"号学生的 *tot_cred* 值未知，我们不能确定它是否大于 45。

大部分关系数据库产品有特殊的"bulk loader"工具，它可以向关系中插入一个非常大的元组集合。这些工具允许从格式化文本文件中读出数据，且执行速度比同等目的的插入语句序列要快得多。

3.9.3 更新

有些情况下，我们可能希望在不改变整个元组的情况下改变其部分属性的值。为达到这一目的，可以使用 **update** 语句。与使用 **insert** 、 **delete** 类似，待更新的元组可以用查询语句找到。

101 假设要进行年度工资增长，所有教师的工资将增长 5%。我们写出：

```
update instructor
set salary = salary * 1.05;
```

上面的更新语句将在 *instructor* 关系的每个元组上执行一次。

如果只给那些工资低于 70 000 美元的教师涨工资，我们可以这样写：

```
update instructor
set salary = salary * 1.05
where salary < 70 000;
```

总之，**update** 语句的 **where** 子句可以包含 **select** 语句的 **where** 子句中的任何合法结构（包括嵌套的 **select**）。和 **insert**、**delete** 类似，**update** 语句中嵌套的 **select** 可以引用待更新的关系。同样，SQL 首先检查关系中的所有元组，看它们是否应该被更新，然后才执行更新。例如，请求"对工资低于平均数的教师涨 5% 的工资"可以写为如下形式：

```
update instructor
set salary = salary * 1.05
where salary < (select avg (salary)
                from instructor);
```

我们现在假设给工资超过 100 000 美元的教师涨 3% 的工资，其余教师涨 5%。我们可以写两条 **update** 语句：

```
update instructor
set salary = salary * 1.03
where salary > 100000;

update instructor
set salary = salary * 1.05
where salary <= 100000;
```

注意这两条 **update** 语句的顺序十分重要。假如我们改变这两条语句的顺序，工资略少于 100 000 美元的教师将增长 8% 的工资。

SQL 提供 **case** 结构，我们可以利用它在一条 **update** 语句中执行前面的两种更新，避免更新次序引
102 发的问题：

```
update instructor
set salary = case
        when salary <= 100000 then salary * 1.05
        else salary * 1.03
    end
```

case 语句的一般格式如下：

```
case
    when pred1 then result1
    when pred2 then result2
    ...
    when predn then resultn
    else result0
end
```

当 i 是第一个满足的 $pred_1$，$pred_2 \cdots pred_n$ 时，此操作就会返回 $result_i$；如果没有一个谓词可以满足，则返回 $result_0$。case 语句可以用在任何应该出现值的地方。

标量子查询在 SQL 更新语句中也非常有用，它们可以用在 **set** 子句中。考虑这样一种更新：我们把每个 *student* 元组的 *tot_cred* 属性值设为该生成功学完的课程学分的总和。我们假设如果一个学生在某门课程上的成绩既不是'F'，也不是空，那么他成功学完了这门课程。我们需要使用 **set** 子句中的子查询来写出这种更新，如下所示：

```
update student S
set tot_cred = (
    select sum(credits)
    from takes natural join course
    where S.ID = takes.ID and
        takes.grade <> 'F' and
        takes.grade is not null);
```

注意子查询使用了来自 **update** 语句中的相关变量 S。如果一个学生没有成功学完任何课程，上述更新语句将把其 *tot_cred* 属性值设为空。如果想把这样的属性值设为 0 的话，我们可以使用另一条 **update** 语句来把空值替换为 0。更好的方案是把上述子查询中的"**select sum**(*credits*)"子句替换为如下使用 **case** 表达式的 **select** 子句：

```
select case
    when sum(credits) is not null then sum(credits)
    else 0
    end
```

103

3.10 总结

- SQL 是最有影响力的商用市场化的关系查询语言。SQL 语言包括几个部分：
  - □ **数据定义语言**(DDL)，它提供了定义关系模式、删除关系以及修改关系模式的命令。
  - □ **数据操纵语言**(DML)，它包括查询语言，以及往数据库中插入元组、从数据库中删除元组和修改数据库中元组的命令。
- SQL 的数据定义语言用于创建具有特定模式的关系。除了声明关系属性的名称和类型之外，SQL 还允许声明完整性约束，例如主码约束和外码约束。
- SQL 提供多种用于查询数据库的语言结构，其中包括 **select**、**from** 和 **where** 子句。SQL 支持自然连接操作。
- SQL 还提供了对属性和关系重命名，以及对查询结果按特定属性进行排序的机制。
- SQL 支持关系上的基本集合运算，包括并、交和差运算，它们分别对应于数学集合论中的 $\cup$、$\cap$ 和 $-$ 运算。
- SQL 通过在通用真值 true 和 false 外增加真值"unknown"，来处理对包含空值的关系的查询。
- SQL 支持聚集，可以把关系进行分组，在每个分组上单独运用聚集。SQL 还支持在分组上的集合运算。
- SQL 支持在外层查询的 **where** 和 **from** 子句中嵌套子查询。它还在一个表达式返回的单个值所允许出现的任何地方支持标量子查询。
- SQL 提供了用于更新、插入、删除信息的结构。

术语回顾

- 数据定义语言
- 数据操纵语言
- 数据库模式
- 数据库实例
- 关系模式
- 关系实例
- 主码
- 外码
  - 参照关系
  - 被参照关系
- 空值
- 查询语言
- SQL 查询结构
  - **select** 子句
  - **from** 子句
  - **where** 子句
- 自然连接运算
- **as** 子句
- **order by** 子句
- 相关名称(相关变量，元组变量)
- 集合运算
  - **union**
  - **intersect**
  - **except**
- 空值
  - 真值"unknown"
- 聚集函数
  - **avg**，**min**，**max**，**sum**，**count**
  - **group by**
  - **having**
- 嵌套子查询
- 集合比较
  - | <，<=，>，>= | **some**，**all** |
  - **exists**
  - **unique**
- **lateral** 子句
- **with** 子句
- 标量子查询
- 数据库修改
  - 删除
  - 插入
  - 更新

实践习题

3.1 使用大学模式，用 SQL 写出如下查询。(建议在一个数据库上实际运行这些查询，使用我们在本书的 Web 网站 db-book.com 上提供的样本数据，上述网站还提供了如何建立一个数据库和加载样本数据的说明。)

a. 找出 Comp. Sci. 系开设的具有 3 个学分的课程名称。

b. 找出名叫 Einstein 的教师所教的所有学生的标识，保证结果中没有重复。

c. 找出教师的最高工资。

d. 找出工资最高的所有教师(可能有不止一位教师具有相同的工资)。

e. 找出 2009 年秋季开设的每个课程段的选课人数。

f. 从 2009 年秋季开设的所有课程段中，找出最多的选课人数。

g. 找出在 2009 年秋季拥有最多选课人数的课程段。

104
|
105

3.2 假设给你一个关系 *grade_points*(*grad_e*, *points*)，它提供从 *takes* 关系中用字母表示的成绩等级到数字表示的得分之间的转换。例如，"A"等级可指定为对应于 4 分，"A -"对应于 3.7 分，"B +"对应于 3.3 分，"B"对应于 3 分，等等。学生在某门课程(课程段)上所获得的等级分值被定义为该课程段的学分乘以该生得到的成绩等级所对应的数字表示的得分。

给定上述关系和我们的大学模式，用 SQL 写出下面的每个查询。为简单起见，可以假设没有任何 *takes* 元组在 *grade* 上取 *null* 值。

a. 根据 ID 为 12345 的学生所选修的所有课程，找出该生所获得的等级分值的总和。

b. 找出上述学生等级分值的平均值(GPA)，即用等级分值的总和除以相关课程学分的总和。

c. 找出每个学生的 ID 和等级分值的平均值。

3.3 使用大学模式，用 SQL 写出如下插入、删除和更新语句。

a. 给 Comp. Sci. 系的每位教师涨 10% 的工资。

b. 删除所有未开设过(即没有出现在 *section* 关系中)的课程。

c. 把每个在 *tot_cred* 属性上取值超过 100 的学生作为同系的教师插入，工资为 10 000 美元。

3.4 考虑图 3-18 中的保险公司数据库，其中加下划线的是主码。为这个关系数据库构造出如下 SQL 查询：

a. 找出 2009 年其车辆出过交通事故的人员总数。

b. 向数据库中增加一个新的事故，对每个必需的属性可以设定任意值。

c. 删除"John Smith"拥有的马自达车(Mazda)。

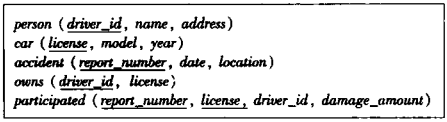

图 3-18 习题 3.4 和习题 3.14 的保险公司数据库

3.5 假设有关系 *marks*(*ID*, *score*)，我们希望基于如下标准为学生评定等级：如果 *score* < 40 得 F；如果 40 ≤ *score* < 60 得 C；如果 60 ≤ *score* < 80 得 B；如果 80 ≤ *score* 得 A。写出 SQL 查询完成下列操作：

a. 基于 *marks* 关系显示每个学生的等级。

b. 找出各等级的学生数。

3.6 SQL 的 **like** 运算符是大小写敏感的，但字符串上的 lower( ) 函数可用来实现大小写不敏感的匹配。为了说明是怎么用的，写出这样一个查询：找出名称中包含了“sci”子串的系，忽略大小写。

3.7 考虑 SQL 查询

```
select distinct p. a1
from p, r1, r2
where p. a1 = r1. a1 or p. a1 = r2. a1
```

在什么条件下这个查询选择的 *p*. *a*1 值要么在 *r*1 中，要么在 *r*2 中？仔细考察 *r*1 或 *r*2 可能为空的情况。

3.8 考虑图 3-19 中的银行数据库，其中加下划线的是主码。为这个关系数据库构造出如下 SQL 查询：

a. 找出银行中所有有账户但无贷款的客户。

b. 找出与“Smith”居住在同一个城市、同一个街道的所有客户的名字。

c. 找出所有支行的名称，在这些支行中都有居住在“Harrison”的客户所开设的账户。

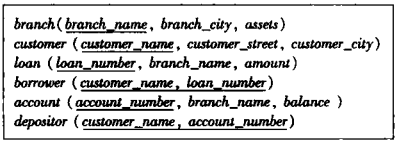

图 3-19 习题 3.8 和习题 3.15 的银行数据库

3.9 考虑图 3-20 的雇员数据库，其中加下划线的是主码。为下面每个查询写出 SQL 表达式：

a. 找出所有为“First Bank Corporation”工作的雇员名字及其居住城市。

b. 找出所有为“First Bank Corporation”工作且薪金超过 10 000 美元的雇员名字、居住街道和城市。

c. 找出数据库中所有不为“First Bank Corporation”工作的雇员。

d. 找出数据库中工资高于“Small Bank Corporation”的每个雇员的所有雇员。

e. 假设一个公司可以在好几个城市有分部。找出位于“Small Bank Corporation”所有所在城市的所有公司。

f. 找出雇员最多的公司。

g. 找出平均工资高于“First Bank Corporation”平均工资的那些公司。

106
107

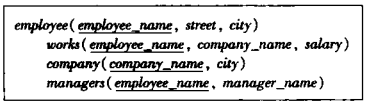

图 3-20 习题 3.9、习题 3.10、习题 3.16、习题 3.17 和习题 3.20 的雇员数据库

3.10 考虑图 3-20 的关系数据库，给出下面每个查询的 SQL 表达式：

a. 修改数据库使"Jones"现在居住在"Newtown"市。

b. 为"First Bank Corporation"所有工资不超过 100 000 美元的经理增长 10% 的工资，对工资超过 100 000 美元的只增长 3%。

习题

3.11 使用大学模式，用 SQL 写出如下查询。

a. 找出所有至少选修了一门 Comp. Sci. 课程的学生姓名，保证结果中没有重复的姓名。

b. 找出所有没有选修在 2009 年春季之前开设的任何课程的学生的 ID 和姓名。

c. 找出每个系教师的最高工资值。可以假设每个系至少有一位教师。

d. 从前述查询所计算出的每个系最高工资中选出最低值。

108 3.12 使用大学模式，用 SQL 写出如下查询。

a. 创建一门课程"CS-001"，其名称为"Weekly Seminar"，学分为 0。

b. 创建该课程在 2009 年秋季的一个课程段，*sec_id* 为 1。

c. 让 Comp. Sci. 系的每个学生都选修上述课程段。

d. 删除名为 Chavez 的学生选修上述课程段的信息。

e. 删除课程 CS-001。如果在运行此删除语句之前，没有先删除这门课程的授课信息(课程段)，会发生什么事情?

f. 删除课程名称中包含"database"的任意课程的任意课程段所对应的所有 *takes* 元组，在课程名的匹配中忽略大小写。

3.13 写出对应于图 3-18 中模式的 SQL DDL。在数据类型上做合理的假设，确保声明主码和外码。

3.14 考虑图 3-18 中的保险公司数据库，其中加下划线的是主码。对这个关系数据库构造如下的 SQL 查询：

a. 找出和"John Smith"的车有关的交通事故数量。

b. 对事故报告编号为"AR2197"中的车牌是"AABB2000"的车辆损坏保险费用更新到 3000 美元。

3.15 考虑图 3-19 中的银行数据库，其中加下划线的是主码。为这个关系数据库构造出如下 SQL 查询：

a. 找出在"Brooklyn"的所有支行都有账户的所有客户。

b. 找出银行的所有贷款额的总和。

c. 找出总资产至少比位于 Brooklyn 的某一家支行要多的所有支行名字。

3.16 考虑图 3-20 中的雇员数据库，其中加下划线的是主码。给出下面每个查询对应的 SQL 表达式：

a. 找出所有为"First Bank Corporation"工作的雇员名字。

b. 找出数据库中所有居住城市和公司所在城市相同的雇员。

c. 找出数据库中所有居住的街道和城市与其经理相同的雇员。

109 d. 找出工资高于其所在公司雇员平均工资的所有雇员。

e. 找出工资总和最小的公司。

3.17 考虑图 3-20 中的关系数据库。给出下面每个查询对应的 SQL 表达式：

a. 为"First Bank Corporation"的所有雇员增长 10% 的工资。

b. 为"First Bank Corporation"的所有经理增长 10% 的工资。

c. 删除"Small Bank Corporation"的雇员在 *works* 关系中的所有元组。

3.18 列出两个原因，说明为什么空值可能被引入到数据库中。

3.19 证明在 SQL 中，**<> all** 等价于 **not in**。

3.20 给出图 3-20 中雇员数据库的 SQL 模式定义。为每个属性选择合适的域，并为每个关系模式选择合适的主码。

3.21 考虑图 3-21 中的图书馆数据库。用 SQL 写出如下查询：

a. 打印借阅了任意由“McGraw-Hill”出版的书的会员名字。
b. 打印借阅了所有由“McGraw-Hill”出版的书的会员名字。
c. 对于每个出版商，打印借阅了多于五本由该出版商出版的书的会员名字。
d. 打印每位会员借阅书籍数量的平均值。考虑这样的情况：如果某会员没有借阅任何书籍，那么该会员根本不会出现在 *borrowed* 关系中。

member(*memb_no*, *name*, *age*)
book(*isbn*, *title*, *authors*, *publisher*)
borrowed(*memb_no*, *isbn*, *date*)

图 3-21 习题 3.21 的图书馆数据库

3.22 不使用 **unique** 结构，重写下面的 **where** 子句：

```
where unique (select title from course)
```

3.23 考虑查询：

```
select course_id, semester, year, sec_id, avg (tot_cred)
from takes natural join student
where year = 2009
group by course_id, semester, year, sec_id
having count (ID) >= 2;
```

解释为什么在 **from** 子句中还加上与 *section* 的连接不会改变查询结果。

3.24 考虑查询：

```
with dept total (dept_name, value) as
    (select dept_name, sum(salary)
    from instructor
    group by dept_name),
dept_total_avg(value) as
    (select avg(value)
    from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total. value >= dept_total_avg. value;
```

不使用 **with** 结构，重写此查询。

工具

很多关系数据库系统可以从市场上购得，包括 IBM DB2、IBM Informix、Oracle、Sybase，以及微软的 SQL Server。另外还有几个数据库系统可以从网上下载并免费使用，包括 PostgreSQL、MySQL(除几种特定的商业化使用外是免费的)和 Oracle Express edition。

大多数数据库系统提供了命令行界面，用于提交 SQL 命令。此外，大多数数据库还提供了图形化的用户界面(GUI)，它们简化了浏览数据库、创建和提交查询，以及管理数据库的任务。还有商品化的 IDE，用于在多个数据库平台上运行 SQL，包括 Embarcadero 的 RAD Studio 与 Aqua Data Studio。

pgAdmin 工具为 PostgreSQL 提供了 GUI 功能；phpMyAdmin 为 MySQL 提供了 GUI 功能。NetBeans IDE 提供了一个 GUI 前端，可以与很多不同的数据库交互，但其功能有限；Eclipse IDE 通过几种不同插件支持类似的功能，这些插件包括 Data Tools Platform(DTP)和 JBuilder。

本书的 Web 网站 db-book. com 提供了 SQL 模式定义和大学模式的样本数据。该 Web 网站还提供了如何建立和访问一些流行的数据库系统的说明。本章讨论的 SQL 结构是 SQL 标准的一部分，但一些特征可能没被某些数据库所支持。Web 网站上列出了这些不相容的特征，在这些数据库上执行查询时需要多加考虑。

110
111

文献注解

SQL 的最早版本 Sequel 2 由 Chamberlin 等[1976]描述。Sequel 2 是从 Square 语言(Boyce 等[1975]以及 Chamberlin 和 Boyce [1974])派生出来的。美国国家标准 SQL-86 在 ANSI [1986]中描述。IBM 系统应用体系结构对 SQL 的定义由 IBM[1987]给出。SQL-89 和 SQL-92 官方标准可分别从 ANSI[1989]和 ANSI[1992]获得。

介绍 SQL-92 语言的教材包括 Date 和 Darwen[1997]、Melton 和 Simon[1993]以及 Cannan 和 Otten [1993]。Date 和 Darwen[1997]以及 Date[1993a]都包含了从编程语言角度对 SQL-92 的评论。

介绍 SQL:1999 的教程包括 Melton 和 Simon[2001]以及 Melton[2002]。Eisenberg 和 Melton[1999]提供了对 SQL:1999 的概览。Donahoo 和 Speegle[2005]从开发者的角度介绍了 SQL。Eisenberg 等[2004]提供了对 SQL:2003 的概览。

SQL:1999、SQL:2003、SQL:2006 和 SQL:2008 标准都是作为 ISO/IEC 标准文档集被发布的，24.4 节将介绍关于它们的更多细节。标准文档中塞满了大量的信息，非常难以阅读，主要用于数据库系统的实现。标准文档可以从 Web 网站 http://webstore.ansi.org 上购买。

很多数据库产品支持标准以外的 SQL 特性，还可能不支持标准中的某些特性。关于这些特性的更多信息可在各产品的 SQL 用户手册中找到。

SQL 查询的处理，包括算法和性能等问题，将在第 12 章和第 13 章讨论。关于这些问题的参考文献也在那里。[112]