

高级 SQL

在第3章和第4章我们详细地介绍了SQL的基本结构。在本章中我们将介绍SQL的一些高级特性。^⑥本章首先介绍如何使用通用程序设计语言来访问SQL，这对于构建用数据库存取数据的应用有重要意义。我们将介绍两种在数据库中执行程序代码的方法：一种是通过扩展SQL语言来支持程序的操作；另一种是在数据库中执行程序语言中定义的函数。接下来本章将介绍触发器，用于说明当特定事件（例如在某个表上进行元组插入、删除或更新操作）发生时自动执行的操作。然后本章将讨论递归查询和SQL支持的高级聚集特性。最后，我们将对联机分析处理（OLAP）系统加以介绍，它可用于海量数据的交互分析。

5.1 使用程序设计语言访问数据库

SQL提供了一种强大的声明性查询语言。实现相同的查询，用SQL写查询语句比用通用程序设计语言要简单得多。然而，数据库程序员必须能够使用通用程序设计语言，原因至少有以下两点：

- 因为SQL没有提供通用程序设计语言那样的表达能力，所以SQL并不能表达所有查询要求。也就是说，有可能存在这样的查询，可以用C、Java或Cobol编写，而用SQL做不到。要写这样的查询，我们可以将SQL嵌入到一种更强大的语言中。
- 非声明性的动作（例如打印一份报告、和用户交互，或者把一次查询的结果送到一个图形用户界面中）都不能用SQL实现。一个应用程序通常包括很多部分，查询或更新数据只是其中之一，而其他部分则用通用程序设计语言实现。对于集成应用来说，必须用某种方法把SQL与通用编程语言结合起来。

可以通过以下两种方法从通用编程语言中访问SQL：

- 动态SQL：通用程序设计语言可以通过函数（对于过程式语言）或者方法（对于面向对象的语言）来连接数据库服务器并与之交互。利用动态SQL可以在运行时以字符串形式构建SQL查询，提交查询，然后把结果存入程序变量中，每次一个元组。动态SQL的SQL组件允许程序在运行时构建和提交SQL查询。

在这一章中，我们将介绍两种用于连接到SQL数据库并执行查询和更新的标准。一种是Java语言的应用程序接口JDBC（5.1.1节）。另一种是ODBC（5.1.2节），它最初是为C语言开发的，后来扩展到其他语言如C++、C#和Visual Basic。

- 嵌入式SQL：与动态SQL类似，嵌入式SQL提供了另外一种使程序与数据库服务器交互的手段。然而，嵌入式SQL语句必须在编译时全部确定，并交给预处理器。预处理程序提交SQL语句到数据库系统进行预编译和优化，然后它把应用程序中的SQL语句替换成相应的代码和函数，最后调用程序语言的编译器进行编译。5.1.3节涵盖嵌入式SQL的内容。

把SQL与通用程序语言相结合的主要挑战是：这些语言处理数据的方式互不兼容。在SQL中，数据的主要类型是关系。SQL语句在关系上进行操作，并返回关系作为结果。程序设计语言通常一次操

⑥ 注意关于章节的先后顺序：数据库设计（第7章和第8章）可以脱离本章独立学习。完全可以先学习数据库设计，再读本章内容。然而，对于强调编程能力的课程而言，在学习了5.1节之后可以做更多的实验练习，所以我们建议在学习数据库设计之前先掌握这部分的内容。

作一个变量，这些变量大致相当于关系中一个元组的一个属性的值。因此，为了在同一应用中整合这两类语言，必须提供一种转换机制，使得程序语言可以处理查询的返回结果。

5.1.1 JDBC

JDBC 标准定义了 Java 程序连接数据库服务器的应用程序接口 (Application Program Interface, API) (JDBC 原来是 **Java 数据库连接** (Java Database Connectivity) 的缩写，但其全称现在已经不用了)。

图 5-1 给出了一个利用 JDBC 接口的 Java 程序的例子。它向我们演示了如何打开数据库连接，执行语句，处理结果，最后关闭连接。我们将在本节详细讨论这个实例。注意，Java 程序必须引用 java.sql.*，它包含了 JDBC 所提供功能的接口定义。

```
public static void JDBCexample(String userid, String passwd)
{
    try
    {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:1521:univdb",
            userid, passwd);
        Statement stmt = conn.createStatement();
        try {
            stmt.executeUpdate(
                "insert into instructor values('77987', 'Kim', 'Physics', 98000)");
        } catch (SQLException sqle)
        {
            System.out.println("Could not insert tuple. " + sqle);
        }
        ResultSet rset = stmt.executeQuery(
            "select dept_name, avg (salary) "+
            "from instructor "+
            "group by dept_name");
        while (rset.next()) {
            System.out.println(rset.getString("dept_name") + " " +
                rset.getFloat(2));
        }
        stmt.close();
        conn.close();
    }
    catch (Exception sqle)
    {
        System.out.println("Exception : " + sqle);
    }
}
```

图 5-1 JDBC 代码示例

5.1.1.1 连接到数据库

要在 Java 程序中访问数据库，首先要打开一个数据库连接。这一步需要选择要使用哪个数据库，例如，可以是你的机器上的一个 Oracle 实例，也可以是运行在另一台机器上的一个 PostgreSQL 数据库。只有在打开数据库连接以后，Java 程序才能执行 SQL 语句。

可以通过调用 `DriverManager` 类 (在 java.sql 包中) 的 `getConnection` 方法来打开一个数据库连接。该方法有三个参数。^①

- `getConnection` 方法的第一个参数是以字符串类型表示的 URL，指明服务器所在的主机名称 (在我们的例子里是 db.yale.edu) 以及可能包含的其他信息，例如，与数据库通信所用的协议 (在我们的例子里是 jdbc:oracle:thin:，我们马上就会看到为什么需要它)，数据库系统用来通信的端口号 (在我们的例子中是 2000)，还有服务器端使用的特定数据库 (在我们的例子中是 univdb)。注意 JDBC 只是指定 API 而不指定通信协议。一个 JDBC 驱动器可能支持多种协议，

① 有多种版本的 `getConnection` 函数，它们所接受的参数各不相同，我们只介绍其中最常用的一个。

我们必须指定一个数据库和驱动器都支持的协议。协议的详细内容是由提供商设定的。

- `getConnection` 方法的第二个参数用于指定一个数据库用户标识，它为字符串类型。
- 第三个参数是密码，它也是字符串类型。（注意，把密码直接写在 JDBC 代码中会增加安全隐患，因为你的代码有可能会被某些未被授权的用户所访问。）

在图 5-1 中的例子中，我们已经建立了一个 `Connection` 对象，其句柄是 `conn`。

每个支持 JDBC（大多数的数据库提供商都支持）的数据库产品都会提供一个 JDBC 驱动程序（JDBC driver），该驱动程序必须被动态加载才能实现 Java 对数据库的访问。事实上，必须在连接数据库之前完成驱动程序的加载。

在图 5-1 中程序的第一行调用 `Class.forName` 函数完成驱动程序的加载，在调用时需要通过参数来指定一个实现了 `java.sql.Driver` 接口的实体类。这个接口的功能是为了实现不同层面的操作之间的转换，一边是与产品类型无关的 JDBC 操作，另一边是与产品相关的、在所使用的特定数据库管理系统中完成的操作。图中的实例采用了 Oracle 的驱动程序，`oracle.jdbc.driver.OracleDriver`^①。该驱动程序包含在一个 `.jar` 文件里，可以从提供商的网站下载，然后放在 Java 的类路径（classpath）里，用于 Java 编译器访问。

用来与数据库交换信息的具体协议并没有在 JDBC 标准中定义，而是由所使用的驱动程序决定的。有些驱动程序支持多种协议，使用哪一种更合适取决于所连接的数据库支持什么协议。我们的例子里，在打开一个数据库连接时，字符串 `jdbc:oracle:thin:` 指定了 Oracle 支持的一个特定协议。

5.1.1.2 向数据库系统中传递 SQL 语句

一旦打开了一个数据库连接，程序就可以利用该连接来向数据库发送 SQL 语句用于执行。这是通过 `Statement` 类的一个实例来完成的。一个 `Statement` 对象并不代表 SQL 语句本身，而是实现了可以被 Java 程序调用的一些方法，通过参数来传递 SQL 语句并被数据库系统所执行。我们的例子在连接变量 `conn` 上创建了一个 `Statement` 句柄（`stmt`）。

我们既可以使用 `executeQuery` 函数又可以用 `executeUpdate` 函数来执行一条语句，这取决于这条 SQL 语句是查询语句（如果是查询语句，自然会返回一个结果集），还是像更新（`update`）、插入（`insert`）、删除（`delete`）、创建表（`create table`）等这样的非查询性语句。在我们的例子里，`stmt.executeUpdate` 执行了一条更新语句，向 `instructor` 关系中插入数据。它返回一个整数，表示被插入、更新或者删除的元组个数。对于 DDL 语句，返回值是 0。try { ... } catch { ... } 结构让我们可以捕捉 JDBC 调用产生的异常（错误情况），并显示给用户适当的出错信息。

5.1.1.3 获取查询结果

示例程序用 `stmt.executeQuery` 来执行一次查询。它可以把结果中的元组集合提取到 `ResultSet` 对象变量 `rset` 中并每次取出一个进行处理。结果集的 `next` 方法用来查看在集合中是否还存在至少一个尚未取回的元组，如果存在的话就取出。`next` 方法的返回值是一个布尔变量，表示是否从结果集中取回了一个元组。可以通过一系列的名字以 `get` 为前缀的方法来得到所获取元组的各个属性。方法 `getString` 可以返回所有的基本 SQL 数据类型的属性（被转换成 Java 中的 `String` 类型的值），当然也可以使用像 `getFloat` 那样一些约束性更强的方法。这些不同的 `get` 方法的参数既可以是一个字符串类型的属性名称，又可以是一个整数，用来表示所需获取的属性在元组中的位置。图 5-1 给出了两种在元组中提取属性值的办法：利用属性名提取（`dept_name`）或者利用属性位置提取（2，代表第二个属性）。

Java 程序结束的时候语句和连接都将被关闭。注意关闭连接是很重要的，因为数据库连接的个数是有限制的；未关闭的连接可能导致超过这一限制。如果发生这种情况，应用将不能再打开任何数据库连接。

① 其他产品的类似的驱动名称如下：IBM DB2: `com.ibm.db2.jdbc.app.DB2Driver`; Microsoft SQL Server: `com.microsoft.sqlserver.jdbc.SQLServerDriver`; PostgreSQL: `org.postgresql.Driver`; MySQL: `com.mysql.jdbc.Driver`。Sun 公司还提供了一种“桥接驱动器”，可以把 JDBC 调用转换成 ODBC。该驱动仅是用于那些支持 ODBC 但不支持 JDBC 的厂商。

5.1.1.4 预备语句

我们也可以以“?”来代表以后再给出的实际值，而创建一个预备语句。数据库系统在准备查询语句的时候对它进行编译。在每次执行该语句时(用新值替换“?”)，数据库可以重用预先编译的查询的形式，应用新值进行查询。图 5-2 的代码框架给出了如何使用预备语句的示例。

```
PreparedStatement pstmt = conn.prepareStatement(
    "insert into instructor values(?,?,?,?)");
pstmt.setString(1, "88877");
pstmt.setString(2, "Perry");
pstmt.setString(3, "Finance");
pstmt.setInt(4, 125000);
pstmt.executeUpdate();
pstmt.setString(1, "88878");
pstmt.executeUpdate();
```

图 5-2 JDBC 代码中的预备语句

可以使用 Connection 类的 prepareStatement 方法来提交 SQL 语句用于编译。它返回一个 PreparedStatement 类的对象。此时还没有执行 SQL 语句。执行需要 PreparedStatement 类的两个方法 executeQuery 和 executeUpdate。但是在它们被调用之前，我们必须使用 PreparedStatement 类的方法来为“?”参数设定具体的值。setString 方法以及诸如 setInt 等用于其他的 SQL 基本类型的其他类似的方法使我们能够为参数指定值。第一个参数用来确定我们为哪个“?”设定值(参数的第一个值是 1，区别于大多数的 Java 结构的从 0 开始)。第二个参数是我们要设定的值。

在图 5-2 中的例子里，我们预备一个 insert 语句，设定“?”参数，并且调用 executeUpdate。例子中的最后两行显示，参数设定保持不变，直到我们特别地进行重新设定。这样，最后的语句调用 executeUpdate，元组(“88878”，“Perry”，“Finance”，125000)被插入到数据库。

在同一查询编译一次然后设置不同的参数值执行多次的情况下，预备语句使得执行更加高效。然而，预备语句有一个更加重要的优势，它使得只要使用了用户输入值，即使是只运行一次，预备语句都是执行 SQL 查询的首选方法。假设我们读取了一个用户输入的值，然后使用 Java 的字符串操作来构造 SQL 语句。如果用户输入了某些特殊字符，例如一个单引号，除非我们采取额外工作对用户输入进行检查，否则生成的 SQL 语句会出现语法错误。setString 方法为我们自动完成检查，并插入需要的转义字符，以确保语法的正确性。

在我们的例子中，假设用户已经输入了 ID、name、dept_name 和 salary 这些变量的值，相应的元组将被插入到关系 instructor 中。假设我们不用预备语句，而是使用如下的 Java 表达式把字符串连接起来构成查询：

```
"insert into instructor values(' " + ID + "' , ' " + name + "' ,
    " + " " + dept_name + " " , ' " + salary + " " )"
```

并且查询通过 Statement 对象的 executeQuery 方法被直接执行。现在，如果用户在 ID 或者 name 域中敲入一个单引号，查询语句就会出现语法错误。一个教员的名字很有可能带有引号(例如“O’Henry”)。

也许以上的例子会被认为是一个小问题，而某些情况会比这糟糕得多。一种叫做 SQL 注入(SQL injection)的技术可以被恶意黑客用来窃取数据或损坏数据库。

假设一个 Java 程序输入一个字符串 name，并且构建下面的查询：

```
"select* from instructor where name = ' " + name + " "
```

如果用户没有输入一个名字，而是输入：

```
X' or 'Y' = 'Y
```

这样，产生的语句就变成：

```
"select* from instructor where name = ' " + "X' or 'Y' = 'Y' + " "
```

即

```
select* from instructor where name = 'X' or 'Y' = 'Y'
```

在生成的查询中，where 子句总是真，所以查询结果返回整个 instructor 关系。更诡计多端的恶意用户甚至可以编写输入值以输出更多的数据。使用预备语句就可以防止这类问题，因为输入的字符串将被插入转义字符，因此最后的查询变为：

```
"select" from instructor where name = 'X\ ' or \'Y\' = \'Y'
```

这是无害的查询语句，返回结果为空集。

比较老的系统允许多个由分号隔开的语句在一次调用里被执行。此功能正逐渐被淘汰，因为恶意的黑客会利用 SQL 注入技术插入整个 SQL 语句。由于这些语句在 Java 程序所有者的权限上运行，像删除表 (drop table) 这样毁灭性的 SQL 语句会被执行。SQL 应用程序开发者必须警惕这种潜在的安全漏洞。

5.1.1.5 可调用语句

JDBC 还提供了 CallableStatement 接口来允许调用 SQL 的存储过程和函数(稍后在 5.2 节描述)，此接口对函数和过程所扮演的角色跟 prepareStatement 对查询所扮演的角色一样。

```
CallableStatement cStmt1 = conn. prepareCall(" ? = call some_function(?) |");
CallableStatement cStmt2 = conn. prepareCall(" | call some_procedure(?,?) |");
```

函数返回值和过程的对外参数的数据类型必须先用方法 registerOutParameter() 注册，它们可以用与结果集用的方法类似的 get 方法获取。请参看 JDBC 手册以获得更细节的信息。

5.1.1.6 元数据特性

正如我们此前提到的，一个 Java 应用程序不包含数据库中存储的数据的声明。这些声明是 SQL 数据定义语言 (DDL) 的一部分。因此，使用 JDBC 的 Java 程序必须要么将关于数据库模式的假设硬编码到程序中，要么直接在运行时从数据库系统中得到那些信息。后一种方法更可取，因为它使得应用程序可以更健壮地处理数据库模式的变化。

回想一下，当我们提交一个使用 executeQuery 方法的查询时，查询结果被封装在一个 ResultSet 对象中。接口 ResultSet 有一个 getMetaData() 方法，它返回一个包含结果集元数据的 ResultSetMetaData 对象。ResultSetMetaData 进一步又包含查找元数据信息的方法，例如结果的列数、某个特定列的名称，或者某个特定列的数据类型。这样，即使不知道结果的模式，我们也可以方便地执行查询。

下面的 Java 代码片段使用 JDBC 来打印出一个结果集中所有列的名称和类型。代码中的变量 rs 假定是执行查询后所获得的一个 ResultSet 实例。

```
ResultSetMetaData rsmd = rs. getMetaData();
for(int i=1; i <= rsmd. getColumnCount(); i++) {
    System. out. println(rsmd. getColumnName(i));
    System. out. println(rsmd. getColumnTypeName(i));
}
```

getColumnCount 方法返回结果关系的元数(属性个数)。这使得我们能够遍历每个属性(请注意，和

JDBC 的惯例一致，我们从 1 开始)。对于每一个属性，我们采用 getColumnName 和 getColumnTypeName 两个方法分别得到它的名称和数据类型。

DatabaseMetaData 接口提供了查找数据库元数据的机制。接口 Connection 包含一个 getMetaData 方法用于返回一个 DatabaseMetaData 对象。接口 DatabaseMetaData 进一步又含有大量的方法可以用于获取程序所连接的数据库和数据库系统的元数据。

例如，有些方法可以返回数据库系统的产品名称和版本号。另外一些方法可以用来查询数据库系统所支持的特性。

还有其他可以返回数据库本身信息的方法。图 5-3 中的代码显示了如何找出数据库中的关系的列(属性)信息。变量 conn 假定存储了一个已经打开的数据库连接。方法 getColumns 有四个参数：一个目录名称(null 表示目录名称被忽略)、一个模式名称模板、一个表名称模板，以及一个列名称模板。模式名称、表名称和列名称的模板可以用于指定一个名字或一个模板。模板可以使用 SQL 字符串匹配特殊字符如“%”和“_”；例如模板“%”匹配所有的名字。只有满足特定名称或模板的模式中的表的列被检索到。结果集中的每行包含一个列的信息。结果集中的行包括若干个列，如目录名称、模式、表和列、列的类型，等等。

```

DatabaseMetaData dbmd = conn. getMetaData();
ResultSet rs = dbmd. getColumns( null, "univdb", "department", "%");
// getColumns 的参数:类别,模式名,表名,列名
// 返回值:每列返回一行,包含一系列属性,例如: COLUMN_NAME, TYPE_NAME
while( rs. next()) {
    System. out. println( rs. getString( " COLUMN_NAME"),
        rs. getString( " TYPE_NAME");
}

```

图 5-3 在 JDBC 中使用 DatabaseMetaData 查找列信息

DatabaseMetaData 还提供了获取数据库信息的一些其他方法,比如,可以用来获取关系 (getTables()), 外码参照 (getCrossReference()), 授权、数据库限制如最大连接数等的元数据。

元数据接口可以用于许多不同的任务。例如,它们可以用于编写数据库的浏览器,该浏览器允许用户找出一个数据库中的关系表,检查它们的模式,检查表中的行,用选择操作查看想要的行,等等。元数据信息可以用于使这些任务的代码更通用;例如,用来显示一个关系中的行的代码可以用这样的方法编写使得它能够在所有可能的关系上工作,无论这些关系的模式是什么。类似地,可以编写这样的代码,它获得一个查询字符串,执行查询,然后把结果打印成一个格式化的表;无论提交了的实际查询是什么,这段代码都可以工作。

165

5.1.1.7 其他特性

JDBC 提供了一些其他特性,如可更新的结果集 (updatable result sets)。它可以从一个在数据库关系上执行选择和/或投影操作的查询中创建一个可更新的结果集。然后,一个对结果集中的元组的更新将引起对数据库关系中相应元组的更新。

回想一下 4.3 节,事务把多个操作封装成一个可以被提交或者回滚的原子单元。

默认情况下,每个 SQL 语句都被作为一个自动提交的独立的事务来对待。JDBC 的 Connection 接口中的方法 setAutoCommit() 允许打开或关闭这种行为。因此,如果 conn 是一个打开的连接,则 conn.setAutoCommit(false) 将关闭自动提交。然后事务必须用 conn.commit() 显式提交或用 conn.rollback() 回滚。自动提交可以用 conn.setAutoCommit(true) 来打开。

JDBC 提供处理大对象的接口而不要求在内存中创建整个大对象。为了获取大对象,ResultSet 接口提供方法 getBlob() 和 getClob(), 它们与 getString() 方法相似,但是分别返回类型为 Blob 和 Clob 的对象。这些对象并不存储整个大对象,而是存储这些大对象的定位器,即指向数据库中实际大对象的逻辑指针。从这些对象中获取数据与从文件或者输入流中获取数据非常相似,可以采用像 getBytes 和 getSubString 这样的方法来实现。

反向操作时,为了向数据库里存储大对象,可以用 PreparedStatement 类的方法 setBlob(int parameterIndex, InputStream inputStream) 把一个类型为二进制大对象 (blob) 的数据库列与一个输入流关联起来。当预备语句被执行时,数据从输入流被读取,然后被写入数据库的二进制大对象中。与此相似,使用方法 setClob 可以设置字符大对象 (clob) 列,该方法的参数包括该列的序号和一个字符串流。

JDBC 还提供了行集 (row set) 特性,允许把结果集打包起来发送给其他应用程序。行集既可以向后又可以向向前扫描,并且可被修改。行集一旦被下载下来就不再是数据库本身的内容了,所以我们在这里并不对其做详细介绍。

5.1.2 ODBC

开放数据库互连 (Open DataBase Connectivity, ODBC) 标准定义了一个 API, 应用程序用它来打开一个数据库连接、发送查询和更新,以及获取返回结果等。应用程序 (例如图形界面、统计程序包或者电子表格) 可以使用相同的 ODBC API 来访问任何一个支持 ODBC 标准的数据库。

166

每一个支持 ODBC 的数据库系统都提供一个和客户端程序相连接的库,当客户端发出一个 ODBC API 请求,库中的代码就可以和服务器通信来执行被请求的动作并取回结果。

图 5-4 给出了一个使用 ODBC API 的 C 语言代码示例。利用 ODBC 和服务器通信的第一步是,建立

一个和服务器的连接。为了实现这一步，程序先分配一个 SQL 的环境变量，然后是一个数据库连接句柄。ODBC 定义了 HENV、HDBC 和 RETCODE 几种类型。程序随后利用 SQLConnect 打开和数据库的连接，这个调用有几个参数，包括数据库的连接句柄、要连接的服务器、用户的身份和密码等。常数 SQL_NTS 表示前面参数是一个以 null 结尾的字符串。

```
void ODBCexample()
{
    RETCODE error;
    HENV env; /* 环境变量 */
    HDBC conn; /* 数据库连接 */

    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, "db.yale.edu", SQL_NTS, "avi", SQL_NTS,
               "avipasswd", SQL_NTS);

    char deptname[80];
    float salary;
    int lenOut1, lenOut2;
    HSTMT stmt;

    char * sqlquery = "select dept_name, sum (salary)
                      from instructor
                      group by dept_name";
    SQLAllocStmt(conn, &stmt);
    error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
    if (error == SQL_SUCCESS) {
        SQLBindCol(stmt, 1, SQL_C_CHAR, deptname, 80, &lenOut1);
        SQLBindCol(stmt, 2, SQL_C_FLOAT, &salary, 0, &lenOut2);
        while (SQLFetch(stmt) == SQL_SUCCESS) {
            printf(" %s %g\n", deptname, salary);
        }
        SQLFreeStmt(stmt, SQL_DROP);
    }

    SQLDisconnect(conn);
    SQLFreeConnect(conn);
    SQLFreeEnv(env);
}
```

图 5-4 ODBC 代码示例

一旦一个连接建立了，C 语言就可以通过 SQLExecDirect 语句把命令发送到数据库。因为 C 语言的变量可以和查询结果的属性绑定，所以当元组被 SQLFetch 语句取回的时候，结果中相应的属性的值就可以放到对应的 C 变量里了。SQLBindCol 做这项工作；在 SQLBindCol 函数里面第二个参数代表选择属性中哪一个位置的属性，第三个参数代表 SQL 应该把属性转化成什么类型的 C 变量。再下一个参数给出了存放变量的地址。对于诸如字符数组这样的变长类型，最后两个参数还要给出变量的最大长度和一个位置来存放元组取回时的实际长度。如果长度域返回一个负值，那么代表着这个值为空 (null)。对于定长类型的变量如整型或浮点型，最大长度的域被忽略，然而当长度域返回一个负值时表示该值为空值。

SQLFetch 在 while 循环中一直执行，直到 SQLFetch 返回一个非 SQL_SUCCESS 的值，在每一次 fetch 过程中，程序把值存放在调用 SQLBindCol 所说明的 C 变量中并把它们打印出来。

在会话结束的时候，程序释放语句的句柄，断开与数据库的连接，同时释放连接和 SQL 环境句柄。好的编程风格要求检查每一个函数的结果，确保它们没有错误，为了简洁，我们在这里忽略了大部分检查。

可以创建带有参数的 SQL 语句，例如，insert into department values(?,?,?)。问号是为将来提供值的占位符。上面的语句可以先被“准备”，也就是在数据库中先编译，然后可以通过为占位符提供具体

值来反复执行——在该例中，为 *department* 关系提供系名、楼宇名和预算数。

ODBC 为各种不同的任务定义了函数，例如查找数据库中所有的关系，以及查找数据库中某个关系的列的名称和类型，或者一个查询结果的列的名称和类型。

在默认情况下，每一个 SQL 语句都被认为是一个自动提交的独立事务。调用 `SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0)` 可以关闭连接 `conn` 的自动提交，事务必须通过显式地调用 `SQLTransact(conn, SQL_COMMIT)` 来提交或通过显式地调用 `SQLTransact(conn, SQL_ROLLBACK)` 来回滚。

ODBC 标准定义了符合性级别 (*conformance level*)，用于指定标准定义的功能的子集。一个 ODBC 实现可以仅提供核心级特性，也可以提供更多的高级特性 (level 1 或 level 2)。level 1 需要支持取得目录的有关信息，例如什么关系存在，它们的属性是什么类型的等。level 2 需要更多的特性，例如发送和提取参数值数组以及检索有关目录的更详细信息的能力。

SQL 标准定义了调用级接口 (Call Level Interface, CLI)，它与 ODBC 接口类似。

ADO. NET

ADO. NET API 是为 Visual Basic. NET 和 C# 语言设计的，它提供了一系列访问数据的函数，与 JDBC 在上层架构没有什么不同，只是在细节上有差别。像 JDBC 和 ODBC 一样，ADO. NET API 可以访问 SQL 查询的结果，以及元数据，但使用起来比 ODBC 简单得多。可以使用 ADO. NET API 来访问支持 ODBC 的数据库，此时，ADO. NET 调用被转换成 ODBC 调用。ADO. NET API 也可以用在某些非关系数据库上，例如微软的 OLE-DB、XML (在第 23 章介绍)，以及微软最近开发的实体框架。有关 ADO. NET 的更多信息请参考文献注解。

5.1.3 嵌入式 SQL

SQL 标准定义了嵌入 SQL 到许多不同的语言中，例如 C、C++、Cobol、Pascal、Java、PL/I 和 Fortran。SQL 查询所嵌入的语言被称为宿主语言，宿主语言中使用的 SQL 结构被称为嵌入式 SQL。

使用宿主语言写出的程序可以通过嵌入式 SQL 的语法访问和修改数据库中的数据。一个使用嵌入式 SQL 的程序在编译前必须先由一个特殊的预处理器进行处理。嵌入的 SQL 请求被宿主语言的声明以及允许运行时执行数据库访问的过程调用所代替。然后，所产生的程序由宿主语言编译器编译。这是嵌入式 SQL 与 JDBC 或 ODBC 的主要区别。

在 JDBC 中，SQL 语句是在运行时被解释的 (即使是利用预备语句特性对其进行准备也是如此)。当使用嵌入式 SQL 时，一些 SQL 相关的错误 (包括数据类型错误) 可以在编译过程中被发现。

为使预处理器识别嵌入式 SQL 请求，我们使用 EXEC SQL 语句，格式如下：

```
EXEC SQL <嵌入式 SQL 语句>;
```

嵌入式 SQL 的确切语法依赖于宿主语言。例如，当宿主语言是 Cobol 时，语句中的分号用 END-EXEC 来代替。

我们在应用程序中合适的地方插入 SQL INCLUDE SQLCA 语句，表示预处理器应该在此处插入特殊变量以用于程序和数据库系统间的通信。

在执行任何 SQL 语句之前，程序必须首先连接到数据库。这是用下面语句实现的：

```
EXEC SQL connect to server user user-name using password;
```

这里，*server* 标识将要建立连接的服务器。

在嵌入的 SQL 语句中可以使用宿主语言的变量，不过前面要加上冒号 (:) 以区别于 SQL 变量。如此使用的变量必须声明在一个 DECLARE 区段里，见下面的代码。不过声明变量的语法还是因循宿主语言的惯例。

```
EXEC SQL BEGIN DECLARE SECTION;
int credit_amount;
EXEC SQL END DECLARE SECTION;
```

嵌入式 SQL 语句的格式和本章描述的 SQL 语句类似。但这儿要指出几点重要的不同之处。

为了表示关系查询, 我们使用**声明游标**(`declare cursor`)语句。然而这时并不计算查询的结果, 而程序必须用 **open** 和 **fetch** 语句(本章后面将讨论)得到结果元组。接下来我们将看到, 使用游标的方法与 JDBC 中对结果集的迭代处理是很相似的。

考虑我们使用的大学模式。假设我们有一个宿主变量 `credit_amount`, 声明方法如前所见, 我们想找出学分高于 `credit_amount` 的所有学生的名字。我们可写出查询语句如下:

```
EXEC SQL
  declare c cursor for
  select ID, name
  from student
  where tot_cred > :credit_amount;
```

上述表达式中的变量 `c` 被称为该查询的游标(*cursor*)。我们使用这个变量来标识该查询, 然后用 **open** 语句来执行查询。

我们的例子中的 **open** 语句如下:

```
EXEC SQL open c;
```

这条语句使得数据库系统执行这条查询并把执行结果存于一个临时关系中。当 **open** 语句被执行的时候, 宿主变量(`credit_amount`)的值就会被应用到查询中。

[170]

如果 SQL 查询出错, 数据库系统将在 SQL 通信区域(SQLCA)的变量中存储一个错误诊断信息。

然后我们利用一系列的 **fetch** 语句把结果元组的值赋给宿主语言的变量。**fetch** 语句要求结果关系的每一个属性有一个宿主变量相对应。在我们的查询例子中, 需要一个变量存储 `ID` 的值, 另一个变量存储 `name` 的值。假设这两个变量分别是 `si` 和 `sn`, 并且都已经在 **DECLARE** 区段中被声明。那么以下语句:

```
EXEC SQL fetch c into :si, :sn;
```

产生结果关系中的一个元组。接下来应用程序就可以利用宿主语言的特性对 `si` 和 `sn` 进行操作了。

一条单一的 **fetch** 请求只能得到一个元组。如果我们想得到所有的结果元组, 程序中必须包含对所有元组执行的一个循环。嵌入式 SQL 为程序员提供了对这种循环进行管理的支持。虽然关系在概念上是一个集合, 查询结果中的元组还是有一定的物理顺序的。执行 SQL 的 **open** 语句后, 游标指向结果的第一个元组。执行一条 **fetch** 语句后, 游标指向结果中的下一个元组。当后面不再有待处理的元组时, SQLCA 中变量 `SQLSTATE` 被置为 '02000' (意指“不再有数据”); 访问该变量的确切的语法依赖于所使用的特定数据库系统。于是, 我们可以用一条 **while** 循环(或其他类似循环语句)来处理结果中的每一个元组。

我们必须使用 **close** 语句来告诉数据库系统删除用于保存查询结果的临时关系。对于我们的例子, 该语句格式如下:

```
EXEC SQL close c;
```

用于数据库修改(**update**、**insert** 和 **delete**)的嵌入式 SQL 表达式不返回结果。因此, 这种语句表达起来在某种程度上相对简单。数据库修改请求格式如下:

```
EXEC SQL <任何有效的 update, insert 和 delete 语句>;
```

前面带冒号的宿主语言的变量可以出现在数据库修改语句的表达式中。如果在语句执行过程中出错, SQLCA 中将设置错误诊断信息。

也可以通过游标来更新数据库关系。例如, 要为音乐系的每个老师的 `salary` 属性都增加 100, 我们

[171]

可以声明这样一个游标:

```
EXEC SQL
  declare c cursor for
  select *
  from instructor
  where dept_name = 'Music'
  for update;
```

然后我们利用在游标上的 **fetch** 操作对元组进行迭代(就像我们先前看到的例子一样), 每取到一个元组我们都执行以下的代码:

```
EXEC SQL
  update instructor
  set salary = salary + 100
  where current of c;
```

可以用 EXEC SQL COMMIT 语句来提交事务, 或者用 EXEC SQL ROLLBACK 进行回滚。

嵌入式 SQL 的查询一般是在编写程序时被定义的。但是在某些比较罕见的情况下查询需要在运行时被定义。例如, 一个应用程序接口可能会让用户来指定某个关系的一个或多个属性上的选择条件, 然后在运行时只利用用户做了选择的属性的条件来构造 SQL 查询的 **where** 子句。此种情况下, 可以使用“EXEC SQL PREPARE <query-name> FROM: <variable>”这样格式的语句, 在运行时构造和准备查询字符串; 并且可以在查询名字上打开一个游标。

SQLJ

与其他的嵌入式 SQL 的实现方法相比, SQLJ(SQL 嵌入 Java 中)提供了相同的特性, 但是它使用了一种不同的语法, 更接近 Java 的固有特性, 比如迭代器。例如, SQLJ 使用句法 `#sql` 代替 EXEC SQL, 并且不使用游标, 而是用 Java 迭代器接口来获取查询结果。因此执行查询的结果被储存在 Java 迭代器里, 然后利用 Java 迭代器接口中的 `next()` 方法来逐步遍历结果元组, 就像前面例子中对游标使用 **fetch** 一样。必须对迭代器的属性进行声明, 其类型应该与 SQL 查询结果中的各属性的类型保持一致。下面的代码段演示了迭代器的使用方法。

```
#sql iterator deptinfoIter ( String dept_name, int avgSal);
deptinfoIter iter = null;

#sql iter = { select dept_name, avg(salary)
              from instructor
              group by dept_name };

while (iter.next()) {
    String deptName = iter.dept_name();
    int avgSal = iter.avgSal();
    System.out.println(deptName + " " + avgSal);
}

iter.close();
```

IBM 的 DB2 和 Oracle 都支持 SQLJ, 并且提供从 SQLJ 代码到 JDBC 代码的转换器。该转换器可以在编译时连接数据库来检查查询的语法是否正确, 并用来确保查询结果的 SQL 类型与所赋值的 Java 变量类型相一致。从 2009 年年初开始, 其他的数据库系统就不支持 SQLJ 了。

我们在这里不详细介绍 SQLJ, 更多信息请看看参考文献。

5.2 函数和过程

我们已经介绍了 SQL 语言的几个内建函数。在本节中, 我们将演示开发者如何来编写他们自己的函数和过程, 把它们存储在数据库里并在 SQL 语句中调用。函数对于特定的数据类型比如图像和几何对象来说特别有用。例如, 用在地图数据库中的一个线段数据类型可能有一个相关函数用于判断两个线段是否重叠, 一个图像数据类型可能有一个相关函数用于比较两幅图的相似性。

函数和过程允许“业务逻辑”作为存储过程记录在数据库中, 并在数据库内执行。例如, 大学里通常有许多规章制度, 规定在一个学期里每个学生能选多少课, 在一年里一个全职的教师至少要上多少节课, 一个学生最多可以在多少个专业中注册, 等等。尽管这样的业务逻辑能够被写成程序设计语言过程并完全存储在数据库以外, 但把它们定义成数据库中的存储过程有几个优点。例如, 它允许多个应用访问这些过程, 允许当业务规则发生变化时进行单个点的改变, 而不必改变应用系统的其他部分。应用代码可以调用存储过程, 而不是直接更新数据库关系。

SQL 允许定义函数、过程和方法。定义可以通过 SQL 的有关过程的组件, 也可以通过外部的程序设计语言, 例如 Java、C 或 C++。我们首先查看 SQL 中的定义, 然后在 5.2.3 节了解如何使用外部语言中的定义。

我们在这里介绍的是 SQL 标准所定义的语法, 然而大多数数据库都实现了它们自己的非标准版本的语法。例如 Oracle(PL/SQL)、Microsoft SQL Sever(TransactSQL) 和 PostgreSQL(PL/pgSQL) 所支持的过程语言都与我们在描述的标准语法有所差别。我们将在后面用 Oracle 来举例说明某些不同之处。更进一步的详细信息可参见各自的系统手册。尽管我们介绍的部分语法在这些系统上并不支持, 但是所阐述的概念在不同的实现上都是适用的, 只是语法上有所区别。

5.2.1 声明和调用 SQL 函数和过程

假定我们想要这样一个函数: 给定一个系的名字, 返回该系的教师数目。我们可以如图 5-5 所示定义函数。^①这个函数可以用在返回教师数大于 12 的所有系的名称和预算的查询中:

```
select dept_name, budget
from department
where dept_count(dept_name) > 12;
```

```
create function dept_count(dept_name varchar(20))
returns integer
begin
declare d_count integer;
select count(*) into d_count
from instructor
where instructor.dept_name = dept_name
return d_count;
end
```

图 5-5 SQL 中定义的函数

SQL 标准支持返回关系作为结果的函数; 这种函数称为表函数(table functions)。^②考虑图 5-6 中定义的函数。该函数返回一个包含某特定系的所有教师的表。注意, 使用函数的参数时需要加上函数名作为前缀(instructor_of.dept_name)。

这种函数可以如下在一个查询中使用:

```
select *
from table (instructor_of ('Finance'));
```

这个查询返回‘金融’系的所有教师。在上面的简单情况下直接写这个查询而不用以表为值的函数也是很直观的。但通常以表为值的函数可以被看作带参数的视图(parameterized view), 它通过允许参数把视图的概念更加一般化。

```
create function instructor_of (dept_name varchar(20))
returns table (
ID varchar (5),
name varchar (20),
dept_name varchar (20),
salary numeric (8,2))
return table
(select ID, name, dept_name, salary
from instructor
where instructor.dept_name = instructor_of.dept_name);
```

图 5-6 SQL 中定义的表函数

SQL 也支持过程。dept_count 函数也可以写成一个过程:

```
create procedure dept_count_proc(in dept_name varchar(20), out d_count integer)
begin
select count(*) into d_count
from instructor
where instructor.dept_name = dept_count_proc.dept_name
end
```

关键字 in 和 out 分别表示待赋值的参数和为返回结果而在过程中设置值的参数。

① 如果要输入自己的函数和过程, 应该写“create or replace”而不是 create, 这样便于在调试时编辑代码(对旧的函数进行替换)。

② 这个特性最早出现在 SQL: 2003 中。

可以从一个 SQL 过程中或者从嵌入式 SQL 中使用 **call** 语句调用过程：

```
declare d_count integer;
call dept_count_proc('Physics', d_count);
```

过程和函数可以通过动态 SQL 触发，如 5.1.1.4 节中 JDBC 语法所示。

SQL 允许多个过程同名，只要同名过程的参数个数不同。名称和参数个数用于标识一个过程。SQL 也允许多个函数同名，只要这些同名的不同函数的参数个数不同，或者对于那些有相同参数个数的函数，至少有一个参数的类型不同。

5.2.2 支持过程和函数的语言构造

SQL 所支持的构造赋予了它与通用程序设计语言相当的几乎所有的功能。SQL 标准中处理这些构造的部分称为持久存储模块(Persistent Storage Module, PSM)。

变量通过 **declare** 语句进行声明，可以是任意的合法 SQL 类型。使用 **set** 语句进行赋值。

一个复合语句有 **begin...end** 的形式，在 **begin** 和 **end** 之间会包含复杂的 SQL 语句。如我们在 5.2.1 节中曾看到的那样，可以在复合语句中声明局部变量。一个形如 **begin atomic...end** 的复合语句可以确保其中包含的所有语句作为单一的事务来执行。

SQL:1999 支持 **while** 语句和 **repeat** 语句，语法如下：

```
while 布尔表达式 do
    语句序列;
end while

repeat
    语句序列;
until 布尔表达式
end repeat
```

还有 **for** 循环，它允许对查询的所有结果重复执行：

```
declare n integer default 0;
for r as
    select budget from department
    where dept_name = 'Music'
do
    set n = n - r, budget
end for
```

程序每次获取查询结果的一行，并存入 **for** 循环变量(在上面例子中指 *r*)中。语句 **leave** 可用来退出循环，而 **iterate** 表示跳过剩余语句从循环的开始进入下一个元组。

SQL 支持的条件语句包括 **if-then-else** 语句，语法如下：

```
if 布尔表达式
    then 语句或复合语句
elseif 布尔表达式
    then 语句或复合语句
else 语句或复合语句
end if
```

SQL 也支持 **case** 语句，类似于 C/C++ 语言中的 **case** 语句(加上我们在第 3 章看到的 **case** 表达式)。

图 5-7 提供了一个有关 SQL 的过程化结构的更大型一点的例子。图中定义的函数 *registerStudent* 首先确认选课的学生数没有超过该课所在教室的容量，然后完成学生对该课的注册。函数返回一个错误代码，这个值大于等于 0 表示成功，返回负值表示出错，同时以 **out** 参数的形式返回消息来说明失败的原因。

SQL 程序语言还支持发信号通知异常条件(exception condition)，以及声明句柄(handler)来处理异常，代码如下：

```
declare out_of_classroom_seats condition
declare exit_handler for out_of_classroom_seats
begin
    sequence of statements
end
```

——在确保教室能容纳下的前提下注册一个学生
 ——如果成功注册，返回 0，如果超过教室容量则返回 -1

```

create function registerStudent(
    in s_id varchar(5),
    in s_courseid varchar(8),
    in s_secid varchar(8),
    in s_semester varchar(6),
    in s_year numeric(4,0),
    out errorMsg varchar(100)

returns integer
begin
    declare currEnrol int;
    select count(*) into currEnrol
    from takes
    where course_id = s_courseid and sec_id = s_secid
      and semester = s_semester and year = s_year;
    declare limit int;
    select capacity into limit
    from classroom natural join section
    where course_id = s_courseid and sec_id = s_secid
      and semester = s_semester and year = s_year;
    if (currEnrol < limit)
    begin
        insert into takes values
            (s_id, s_courseid, s_secid, s_semester, s_year, null);
        return(0);
    end
    ——否则，已经达到课程容量上限
    set errorMsg = 'Enrollment limit reached for course ' || s_courseid
      || ' section ' || s_secid;
    return(-1);
end;
```

图 5-7 学生注册课程的过程

在 **begin** 和 **end** 之间的语句可以执行 **signal out_of_classroom_seats** 来引发一个异常。这个句柄说明，如果条件发生，将会采取动作终止 **begin end** 中的语句。另一个可选的动作将是 **continue**，它继续从引发异常的语句的下一条语句开始执行。除了明确定义的条件，还有一些预定义的条件，比如 **sqlexception**、**sqlwarning** 和 **not found**。

过程和函数的非标准语法

尽管 SQL 标准为过程和函数定义了语法，但是很多数据库并不严格遵照标准，在语法支持方面存在很多变化。这种情况的原因之一是这些数据库通常在语法标准制定之前就已经引入了对过程和函数的支持机制，然后一直沿用最初的语法。在这里把每个数据库所支持的语法罗列出来并不现实，不过我们可以介绍一下 Oracle 的 PL/SQL 与标准语法不同的一些方面，下面是图 5-5 中的函数在 PL/SQL 里定义的一个版本。

```

create or replace function dept_count(dept_name in instructor.dept_name%type)
return integer
as
    d_count integer;
begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_name;
    return d_count;
end;
```

这两个版本在概念上相似，但还是存在很多次要的语法上的区别，可以利用其中一些不同操作作为区分两个函数版本的手段。尽管没有在这里展示，但是 PL/SQL 中关于控制流的语法与在这里所列出的也有所不同。

可以看到，通过加 %type 前缀，PL/SQL 允许把某个类型设置为关系的一个属性的类型。另一方面，PL/SQL 并不直接支持返回一个表的功能，只能通过创建表类型这样的间接方法来实现。其他数据库所支持的过程语言同样含有很多语法和语义上的差别。更多信息请查阅相关语言的参考资料。

5.2.3 外部语言过程

尽管对 SQL 的过程化扩展非常有用，然而可惜的是这些并不被跨数据库的标准的方法所支持。即使是最基本的特性在不同数据库产品中都可能有不同的语法和语义。所以，程序员必须针对每个数据库产品学习一门新语言。还有另一种方案可以解决语言支持的问题，即在一种命令式程序设计语言中定义过程，然后从 SQL 查询和触发器的定义中来调用它。

SQL 允许我们用一种程序设计语言定义函数，比如 Java、C#、C 或 C++。这种方式定义的函数会比 SQL 中定义的函数效率更高，无法在 SQL 中执行的计算可以由这些函数执行。

外部过程和函数可以这样指定（注意，确切的语法决定于所使用的特定数据库系统）：

```
create procedure dept_count( in dept_name varchar(20),
                           out count integer)
language C
external name 'usr/avi/bin/dept_count_proc'

create function dept count ( dept_name varchar(20))
returns integer
language C
external name 'usr/avi/bin/dept_count'
```

通常来说，外部语言过程需要处理参数（包含 in 和 out 参数）和返回值中的空值，还需要传递操作失败/成功状态，以方便对异常进行处理。这些信息可以通过几个额外的参数来表示：一个指明失败/成功状态的 sqlstate 值、一个存储函数返回值的参数，以及一些指明每个参数/函数结果的值是否为空的指示器变量。还可以通过其他机制来解决空值的问题，例如，可以传递指针而不是值。具体采用哪种方法取决于数据库。不过，如果一个函数不关注这些情况，可以在声明语句的上方添加额外的一行 parameter style general 指明外部过程/函数只使用说明的变量并且不处理空值和异常。

用程序设计语言定义并在数据库系统之外编译的函数可以由数据库系统代码来加载和执行。不过这么做存在危险，那就是程序中的错误可能破坏数据库内部的结构，并且绕过数据库系统的访问-控制功能。如果数据库系统关心执行的效率胜过安全性则可以采用这种方式执行过程。关心安全性的数据库系统一般会把这些代码作为一个单独进程的一部分来执行，通过进程间通信，传入参数的值，取回结果。然而，进程间通信的时间代价相当高；在典型的 CPU 体系结构中，一个进程通信所需的时间可以执行数万到数十万条指令。

如果代码用 Java 或 C# 这种“安全”的语言书写，则会有第三种可能：在数据库进程本身的沙盒（sandbox）内执行代码。沙盒允许 Java 或 C# 代码访问它的内存区域，但阻止代码直接在查询执行过程的内存中做任何读操作或者更新操作，或者访问文件系统中的文件。（在如 C 这种语言中创建一个沙盒是不可能的，因为 C 语言允许通过指针不加限制地访问内存。）避免进程间通信能够大大降低函数调用的时间代价。

当今的一些数据库系统支持外部语言例程在查询执行过程中的沙盒里运行。例如，Oracle 和 IBM DB2 允许 Java 函数作为数据库过程中的一部分运行。Microsoft SQL Server 允许过程编译成通用语言运行程序（CLR）来在数据库过程中执行；这样的过程可以用 C# 或 Visual Basic 编写。PostgreSQL 允许在 Perl、Python 和 Tcl 等多种语言中定义函数。

5.3 触发器

触发器(trigger)是一条语句,当对数据库作修改时,它自动被系统执行。要设置触发器机制,必须满足两个要求:

- 指明什么条件下执行触发器。它被分解为一个引起触发器被检测的事件和一个触发器执行必须满足的条件。
- 指明触发器执行时的动作。

一旦我们把一个触发器输入数据库,只要指定的事件发生,相应的条件满足,数据库系统就有责任去执行它。

5.3.1 对触发器的需求

180

触发器可以用来实现未被 SQL 约束机制指定的某些完整性约束。它还是一种非常有用的机制,用来当满足特定条件时对用户发警报或自动开始执行某项任务。例如,我们可以设计一个触发器,只要有元组被插入 *takes* 关系中,就更新 *student* 关系中选课的学生所对应的元组,把该课的学分加入这个学生的总学分中。作为另一个应用触发器的例子,假设一个仓库希望每种物品的库存保持一个最小量;当某种物品的库存少于最小值的时候,自动发出一个订货单。在更新某种物品的库存的时候,触发器会比较这种物品的当前库存和它的最小库存,如果库存数量等于或小于最小值,就会生成一个新的订单。

注意,触发器系统通常不能执行数据库以外的更新,因此,在上面的库存补充的例子中,我们不能用一个触发器去直接在外部的世界下订单,而是在存放订单的关系中添加一个关系记录。我们必须另外创建一个持久运行的系统进程来周期性扫描该关系并订购产品。某些数据库系统提供了内置的支持,可以使用上述方法从 SQL 查询和触发器中发送电子邮件。

5.3.2 SQL 中的触发器

现在我们来如何在 SQL 中实现触发器。我们在这里介绍的是 SQL 标准定义的语法,但是大部分数据库实现的是非标准版本的语法。尽管这里所述的语法可能不被这些系统支持,但是我们阐述的概念对于不同的实现方法都是适用的。我们将在本章末尾讨论非标准的触发器实现。

图 5-8 展示了如何使用触发器来确保关系 *section* 中属性 *time_slot_id* 的参照完整性。图中第一个触发器的定义指明该触发器在任何一次对关系 *section* 的插入操作执行之后被启动,以确保所插入元组的 *time_slot_id* 字段是合法的。一个 SQL 插入语句可以向关系中插入多个元组,在触发器代码中的 **for each row** 语句可以显式地在每一个被插入的行上进行迭代。**referencing new row as nrow** 语句建立了一个变量 *nrow* (称为过渡变量(transition variable)),用来在插入完成后存储所插入行的值。

```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
    select time_slot_id
    from time_slot)) /* time_slot 中不存在该 time_slot_id */
begin
    rollback
end;
create trigger timeslot_check2 after delete on time_slot
referencing old row as orow
for each row
when (orow.time_slot_id not in (
    select time_slot_id
    from time_slot) /* 在 time_slot 中刚刚被删除的 time_slot_id */
and orow.time_slot_id in (
    select time_slot_id
    from section)) /* 在 section 中仍含有该 time_slot_id 的引用 */
begin
    rollback
end;
```

图 5-8 使用触发器来维护参照完整性

when 语句指定一个条件。仅对于满足条件的元组系统才会执行触发器中的其余部分。**begin atomic ...end** 语句用来将多行 SQL 语句集成为一个复合语句。不过在我们的例子中只有一条语句，它对引起触发器被执行的事务进行回滚。这样，所有违背参照完整性约束的事务都将被回滚，从而确保数据库中的数据满足约束条件。

只检查插入时的参照完整性还不够，我们还需要考虑对关系 *section* 的更新，以及对被引用的表 *time_slot* 的删除和更新操作。图 5-8 定义的第二个触发器关注的是 *time_slot* 表的删除。该触发器检查被删除元组的 *time_slot_id* 要么还在 *time_slot* 中，要么 *section* 里不存在包含这个 *time_slot_id* 值的元组；否则将违背参照完整性。

为了保证参照完整性，我们也必须为处理 *section* 和 *time_slot* 的更新来创建触发器；我们接下来将介绍如何在更新时执行触发器，不过，该如何定义这些触发器就留给读者作为练习。

对于更新来说，触发器可以指定哪个属性的更新使其执行；而其他属性的更新不会让它产生动作。例如，为了指定当更新关系 *takes* 的属性 *grade* 时执行触发器，我们可以这样写：

after update of takes on grade

referencing old row as 子句可以建立一个变量用来存储已经更新或删除行的旧值。**referencing new row as** 子句除了插入还可以用于更新操作。

如图 5-9 所示，当关系 *takes* 中元组的属性 *grade* 被更新时，需要用触发器来维护 *student* 里元组的 *tot_cred* 属性，使其保持实时更新。只有当属性 *grade* 从空值或者 'F' 被更新为代表课程已经完成的具

```
create trigger credits_earned after update of takes on (grade)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> 'F' and nrow.grade is not null
and (orow.grade = 'F' or orow.grade is null)
begin atomic
  update student
  set tot_cred = tot_cred +
    (select credits
     from course
     where course.course_id = nrow.course_id)
  where student.id = nrow.id;
end;
```

图 5-9 使用触发器来维护 *credits_earned* 值

本例中触发器的更实际的实现还可以解决分数更正的问题，包括把成功结课的分数的改成不及格分数，以及向关系 *takes* 中插入含有表示成功结课的 *grade* 值的元组。读者可以把这些实现作为练习。

另举一个使用触发器的例子，当删除一个 *student* 元组的事件发生时，需要检查关系 *takes* 中是否存在与该学生相关的项，如果有则删除它。

许多数据库系统支持各种别的触发器事件，比如当一个用户（应用程序）登录到数据库（即打开一个连接）的时候，或者当系统停止的时候，或者当系统设置改变的时候。

触发器可以在事件（insert、delete 或 update）之前激发，而不仅是事件之后。在事件之前被执行的触发器可以作为避免非法更新、插入或删除的额外约束。为了避免执行非法动作而产生错误，触发器可以采取措来纠正问题，使更新、插入或删除操作合法化。例如，假设我们想把一个教师插入某个系，而该系的名称并不在关系 *department* 中，那么触发器就可以提前向关系 *department* 中加入该系名称，以避免在插入时产生外码冲突。另一个例子是，假设所插入分数的值为空白则表明该分数发生缺失。我们可以定义一个触发器，将这个值用 **null** 值代替。**set** 语句可以用来执行这样的修改。这种触发器的例子如图 5-10 所示。

181

182

我们可以对引起插入、删除或更新的 SQL 语句执行单一动作，而不是对每个被影响的行执行一个动作。要做到这一点，我们用 **for each statement** 子句来替代 **for each row** 子句。可以用子句 **referencing old table as** 或 **referencing new table as** 来指向包含所有的被影响的行的临时表（称为过渡表（transition table））。过渡表不能用于 **before** 触发器，但是可以用于 **after** 触发器，无论它们是语句触发器还是行触发器。这样，在过渡表的基础上，一个单独的 SQL 语句就可以用来执行多个动作。

```
create trigger setnull before update of takes
referencing new row as nrow
for each row
when (nrow.grade = '')
begin atomic
set nrow.grade = null;
end;
```

图 5-10 使用 set 来更改插入值的例子

非标准的触发器语法

尽管我们在这里介绍的触发器语法是 SQL 标准的一部分，并被 IBM DB2 所支持，但是大多数其他的数据库系统用非标准的语法来说明触发器，不一定实现了 SQL 标准的所有特性。我们将在下面概述其中的一些不同；更进一步的细节请参考相关的系统手册。

例如，与 SQL 标准语法不同的是，Oracle 的语法中，在 **referencing** 语句中并没有关键词 **row**，而且在 **begin** 之后没有关键词 **atomic**。在 **update** 语句中嵌入的 **select** 语句对 **nrow** 的引用必须以冒号(:)为前缀，用以向系统说明变量 **nrow** 是在 SQL 语句之外所定义的。更不一样的是，在 **when** 和 **子句** 中不允许包含子查询。可以用下面的方法绕过这个限制：把复杂的谓词从 **when** 子句移到单独的查询中，用本地变量保存查询结果，然后在一个 **子句** 里引用该变量，并把触发器的内容移动到相关的 **then** 子句里。更进一步，Oracle 中的触发器不允许直接执行事务回滚；但是，可以使用函数 **raise_application_error** 来代替它，它在回滚事务的同时返回错误信息给执行更新的用户/应用程序。

另一个例子是，在 Microsoft SQL Server 中用关键字 **on** 来替代 **after**。不支持 **referencing** 子句，而是用 **deleted** 和 **inserted** 修饰词组变量来表示被影响的行的旧值和新值。另外，语法中略去了 **for each row** 子句，并用 **when** 来替代 **when**。不支持 **before** 语句样式，但是支持 **instead of** 语法。

在 PostgreSQL 中，触发器的定义不包含执行内容，而是对每一行调用一个过程，使用 **old** 和 **new** 来访问包含该行的旧值和新值的变量。触发器不进行回滚，而是发出一个异常，该异常中包含了相关的错误信息。

触发器可以设为有效或者无效：默认情况下它们在创建时是有效的，但是可以通过使用 **alter trigger trigger_name disable**（某些数据库使用另一种语法，比如 **disable trigger trigger_name**）将其设为无效。设为无效的触发器可以重新设为有效。通过使用命令 **drop trigger trigger_name**，触发器也可以被丢弃，即将其永久移除。

回到仓库库存的例子，假设有如下的关系：

- **inventory(item, level)**，表示物品在仓库中的当前库存量。
- **minlevel(item, level)**，表示物品应该保持的最小库存量。
- **reorder(item, amount)**，表示当物品小于最小库存量的时候要订购的数量。
- **orders(item, amount)**，表示物品被订购的数量。

注意我们已经很小心地仅在当物品库存量从大于最小值降到最小值以下的时候才下达一个订单。如果只检查更新后的新数值是否小于最小值，就可能在物品已经被重订购的时候错误地下达一个订单。我们可以用图 5-11 中的触发器来重新订购物品。

尽管触发器在 SQL:1999 之前并不是 SQL 标准的一部分，但是它仍被广泛地应用在基于 SQL 的数据库系统中。遗憾的是，每个数据库系统都实现了自己的触发器语法，导致彼此不能兼容。我们在这里用的是 SQL:1999 的触发器语法，它与 IBM 的 DB2 以及 Oracle 数据库系统的语法比较相似，但不完全一致。

```

create trigger reorder after update of level on inventory
referencing old row as orow, new row as nrow
for each row
when nrow.level <= (select level
                    from minlevel
                    where minlevel.item = orow.item)
and orow.level > (select level
                 from minlevel
                 where minlevel.item = orow.item)
begin atomic
insert into orders
(select item, amount
 from reorder
 where reorder.item = orow.item);
end;

```

图 5-11 重新订购物品的触发器例子

5.3.3 何时不用触发器

触发器有很多合适的用途，例如我们刚刚在 5.3.2 节中看到的那些，然而有一些场合最好用别的技术来处理。比如，我们可以用触发器替代级联特性来实现外码约束的 **on delete cascade** 特性。然而这样不仅需要完成更大的工作量，而且使数据库中实现的约束集合对于数据库用户来说更加难以理解。

举另外一个例子，可以用触发器来维护物化视图。例如，如果我们希望能够快速得到每节课所注册的学生总数，我们可以创建一个关系来实现这个功能：

```
section_registration(course_id, sec_id, semester, year, total_students)
```

它由以下查询所定义：

```

select course_id, sec_id, semester, year, count(ID) as total_students
from takes
group by course_id, sec_id, semester, year;

```

在对关系 *takes* 进行插入、删除或更新时，每门课的 *total_students* 的值必须由触发器来维护到最新状态。维护时可能要对 *section_registration* 做插入、更新或删除，这些都相应地写在触发器里。

然而，许多数据库现在支持物化视图，由数据库系统自动维护（见 4.2.3 节）。因此没必要编写代码让触发器来维护这样的物化视图。

触发器也被用来复制或者备份数据库；在每一个关系的插入、删除或更新的操作上创建触发器，将改变记录在称为 **change** 或 **delta** 的关系上。一个单独的进程将这些改变复制到数据库的副本。然而，现代的数据库系统提供内置的数据库复制工具，使得复制在大多数情况下不必使用触发器。本书将在第 19 章详细讨论复制数据库。

触发器的另一个问题是当数据从一个备份的拷贝^①中加载，或者一个站点上的数据库更新复制到备份站点的时候，触发器动作的非故意执行。在该情况下，触发器动作已经执行了，通常不应该再次执行。在加载数据的时候，触发器应当显式设为无效。对于要接管主系统的备份复制系统，触发器应该一开始就设为无效，而在备份站点接管了主系统的业务后，再设为有效。作为取代的方法，一些数据库系统允许触发器定义为 **not for replication**，保证触发器不会在数据库备份的时候在备份站点执行。另一些数据库系统提供了一个系统变量用于指明该数据库是一个副本，数据库动作在其上是重放；触发器会检查这个变量，如果为真则退出执行。这两种解决方案都不需要显式地将触发器设为失效或有效。

写触发器时，应特别小心，这是因为在运行期间一个触发器错误会导致引发该触发器的动作语句

① 我们将在第 16 章详细讨论数据库备份和从故障中恢复的内容。

失败。而且，一个触发器的动作可以引发另一个触发器。在最坏的情况下，这甚至会导致一个无限的触发链。例如，假设在一个关系上的插入触发器里有一个动作引起在同一关系上的另一个（新的）插入，该新插入动作也会引起另一个新插入，如此无穷循环下去。有些数据库系统会限制这种触发器链的长度（例如 16 或 32），把超过此长度的触发器链看作是一个错误。另一些系统把引用特定关系的触发器标记为错误，对该关系的修改导致了位于链首的触发器被执行。

触发器是很有用的工具，但是如果有其他候选方法就最好别用触发器。很多触发器的应用都可以用适当的存储过程来替换，后者我们在 5.2 节已经介绍过了。

5.4 递归查询**

考虑图 5-12 中的例子，关系 *prereq* 的实例包含大学开设的各门课程的信息以及每门课的先修条件。[⊖]

假设我们想知道某个特定课程（例如 CS-347）的直接或者间接的先修课程。也就是说，我们想找到这样的课，要么是选修 CS-347 的直接先决条件，要么是选修 CS-347 的先修课程的先决条件，以此类推。

因此，如果 CS-301 是 CS-347 的先修课程，并且 CS-201 是 CS-301 的先修课程，还有 CS-101 是 CS-201 的先修课程，那么 CS-301、CS-201、CS-101 都是 CS-347 的先修课程。

关系 *prereq* 的传递闭包 (transitive closure) 是一个包含所有 (*cid*, *pre*) 对的关系，*pre* 是 *cid* 的一个直接或间接先修课程。有许多要求计算与此类似的层次 (hierarchy) 的传递闭包的应用。例如，机构通常由几层组织单元构成。机器由部件构成，而部件又有子部件，如此类推；例如，一辆自行车可能有子部件如车轮和踏板，它们又有子部件如轮胎、轮圈、辐条。可以对这种层次结构使用传递闭包来找出，例如，自行车中的所有部件。

course_id	prereq_id
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

图 5-12 关系 *prereq*

5.4.1 用迭代来计算传递闭包

一个写上述查询的方法是使用迭代：首先找到 CS-347 的那些直接先修课程，然后是第一个集合中的所有课程的先修课程，如此类推。迭代持续进行，直到某次循环中没有新课程加进来才停止。图 5-13 显示了执行这项任务的函数 *findAllPrereqs*(*cid*)；这个函数以课程的 *course_id* 为参数 (*cid*)，计算该课程所有直接或间接的先修课程并返回它们组成的集合。

过程中用到了三个临时表：

- *c_prereq*：存储要返回的元组集合。
- *new_c_prereq*：存储在前一次迭代中找到的课程。
- *temp*：当对课程集合进行操作时用作临时存储。

注意，SQL 允许使用命令 **create temporary table** 来创建临时表；这些表仅在执行查询的事务内部才可用，并随事务的完成而被删除。而且，如果 *findAllPrereqs* 的两个实例同时运行，每个实例都拥有自己的临时表副本；假设它们共享一份副本，结果就会出错。

该过程在 **repeat** 循环之前把课程 *cid* 的所有直接先修课程插入 *new_c_prereq* 中。**repeat** 循环首先把 *new_c_prereq* 中所有课程加入 *c_prereq* 中。然后，它为 *new_c_prereq* 中的所有课程计算先修课程，从结果中筛选掉此前已经计算出来是 *cid* 的先修课的课程，把剩下的存放在临时表 *temp* 中。最后，它把 *new_c_prereq* 中的内容替换成 *temp* 中的内容。当找不到新的（间接）先修课程时，**repeat** 循环终止。

以 CS-347 为参数调用以上过程，图 5-14 显示了每次迭代中找到的先修课程。

我们注意到在函数中使用 **except** 子句保证即使在先修关系中存在环时（非正常情况），函数也能工作。例如，如果 *a* 为 *b* 的先修课，*b* 为 *c* 的先修课，*c* 为 *a* 的先修课，则存在一个环。

[⊖] *prereq* 的实例与我们以前使用的有所不同，其原因当我们用它来解释递归查询时就能看得很清楚了。

```
create function findAllPrereqs(cid varchar(8))
--找出 cid 的所有(直接或间接)先修课程
returns table (course_id varchar(8))
--关系 prereq(course_id, prereq_id) 指明哪一门课程是另一门课的直接先修课
begin
create temporary table c_prereq (course_id varchar(8));
-- 表 c_prereq 存储将要返回的课程集合
create temporary table new_c_prereq (course_id varchar(8));
-- 表 new_c_prereq 包含上一次迭代中发现的课程
create temporary table temp (course_id varchar(8));
-- 表 temp 用来存储中间结果
insert into new_c_prereq
select prereq_id
from prereq
where course_id = cid;
repeat
insert into c_prereq
select course_id
from new_c_prereq;
insert into temp
(select prereq, course_id
from new_c_prereq, prereq
where new_c_prereq.course_id = prereq.prereq_id
)
except (
select course_id
from c_prereq
);
delete from new_c_prereq;
insert into new_c_prereq
select *
from temp;
delete from temp;
until not exists (select * from new_c_prereq)
end repeat;
return table c_prereq;
end
```

图 5-13 找到某门课的所有先修课程

尽管在课程先修关系中不存在环是不现实的，但循环可能在其他应用中存在。例如，假设我们有一个关系 *flights*(to, from) 表示哪个城市可以从其他城市直接飞达。我们可以写类似于 *findAllPrereqs* 函数的代码来找到所有从某个给定城市出发通过一次或一系列的飞行可以到达的城市。我们所要做的只是用 *flight* 代替 *prereq* 并且替换相应的属性名称。在这个情况下可能存在可达关系的环，但函数仍然会正确工作，因为它会将所有已见过的城市去掉。

Iteration Number	Flights in c1
0	
1	(CS-301)
2	(CS-301), (CS-201)
3	(CS-301), (CS-201)
4	(CS-301), (CS-201), (CS-101)
5	(CS-301), (CS-201), (CS-101)

图 5-14 函数 *findAllPrereqs* 迭代过程中产生的 CS-347 的先修课程

5. 4. 2 SQL 中的递归

用迭代表示传递闭包是挺不方便的。还有另一种方法，即使用递归视图定义，这种方法更加容易使用。

我们可以用递归为某个指定课程如 CS-347 定义先修课程集合，方法如下。CS-347 的(直接或间接)先修课程是：

- CS-347 的先修课程。
- 作为 CS-347 的(直接或间接)先修课程的先修课程的课程。

注意，第二条是递归，因为它用 CS-347 的先修课程来定义 CS-347 的先修课程。传递闭包的其他例子

如找出一个给定部件的所有直接或间接子部件同样可以类似地递归定义。

从 SQL:1999 开始, SQL 标准中用 **with recursive** 子句来支持有限形式的递归, 在递归中一个视图(或临时视图)用自身来表达自身。例如, 递归查询可以用于精确地表达传递闭包。回忆 **with** 子句用于定义一个临时视图, 该视图的定义只对定义它的查询可用。附加的关键字 **recursive** 表示该视图是递归的。

例如, 用图 5-15 中显示的递归 SQL 视图, 我们可以找到每一对 (cid, pre) , 其中 pre 是 cid 的直接或间接的先修课程。

任何递归视图都必须被定义为两个子查询的并: 一个非递归的基查询(base query)和一个使用递归视图的递归查询(recursive query)。在图 5-15 所示的例子中, 基查询是关系 $prereq$ 上的选择操作, 而递归查询则计算 $prereq$ 和 rec_prereq 的连接。

对递归视图的含义最好的理解如下。首先计算基查询并把所有结果元组添加到视图关系 rec_prereq (初始时为空) 中。然后用当前视图关系的内容计算递归查询, 并把所有结果元组加回到视图关系中。持续重复上述步骤直至没有新的元组添加到视图关系中为止。得到的视图关系实例就称为递归视图定义的一个不动点(fixed point)。(术语“不动”是指不会再有进一步的改变。这样视图关系被定义为正好包含处于不动点实例中的元组。

```
with recursive rec_prereq(course_id, prereq_id) as (
  select course_id, prereq_id
  from prereq
  union
  select rec_prereq.course_id, prereq.prereq_id
  from prereq, rec_prereq
  where prereq.course_id = rec_prereq.prereq_id
)
select*
from rec_prereq;
```

图 5-15 SQL 中的递归查询

把上述逻辑应用到我们的例子中, 我们将首先通过执行基查询找到每门课的直接先修课程。递归查询会在每次迭代过程中增加一层课程, 直到达到课程 - 先修课程关系的最大层次。在这时将不会再有新的元组添加到视图中, 同时达到了一个不动点。

为了找到指定课程的先修课程, 以 CS-347 为例, 我们可以加入 **where** 子句“**where rec_prereq.course_id = 'CS-347'**”来修改外层查询。对如此条件的查询进行求值, 方法之一是: 先使用迭代技术计算 rec_prereq 的所有内容, 然后从结果中选择 $course_id$ 为 CS-347 的那些元组。但是, 这样需要为所有课程计算(课程, 先修课程)对, 其中, 除了涉及 CS-347 的之外, 其他的都不相关。事实上, 数据库系统没有必要使用上述迭代技术计算递归查询的完整结果然后进行筛选。可以使用其他效率更高的技术来得到相同的结果, 比如 $findAllPrereqs$ 函数中用到的方法看起来就更简单些。更多信息请参阅参考文献。

递归视图中的递归查询是有一些限制的; 具体地说, 该查询必须是单调的(monotonic), 也就是说, 如果视图关系实例 V_1 是实例 V_2 的超集的话, 那么它在 V_1 上的结果必须是它在 V_2 上的结果的超集。直观上, 如果更多的元组被添加到视图关系, 则递归查询至少应该返回与以前相同的元组集, 并且还可能返回另外一些元组。

特别指出, 递归查询不能用于任何下列构造, 因为它们会导致查询非单调:

- 递归视图上的聚集。
- 在使用递归视图的子查询上的 **not exists** 语句。
- 右端使用递归视图的集合差(**except**)运算。

例如, 如果递归查询是 $r-v$ 的形式, 其中 v 是递归视图, 如果我们在 v 中增加一个元组, 查询结果可能会变小; 可见该查询不是单调的。

只要递归查询是单调的, 递归视图的含义就可以用迭代过程来定义; 如果递归查询是非单调的, 则视图的含义很难确定。因此 SQL 要求查询必须是单调的。递归查询将在 B.3.6 节 Datalog 查询语言的环境中更加详细地讨论。

SQL 还允许使用 **create recursive view** 代替 **with recursive** 来创建递归定义永久视图。一些系统实现支持使用不同语法的递归查询; 请参考各自的系统手册以获得更多细节。

5.5 高级聚集特性**

此前我们已经看到, SQL 对聚集的支持是十分强大的, 可以很便捷地完成一般性的任务。然而, 有些任务很难用基本的聚集特性来高效实现。在本节中, 我们将研究为完成这些任务而向 SQL 中引入的一些特性。

5.5.1 排名

从一个大的集合中找出某值的位置是一个常见的操作。例如, 我们可能希望基于学生的平均绩点 (GPA) 赋予他们在班级中的名次: GPA 最高的学生排名第 1, 次高学生排名第 2, 等等。另一种相关的查询类型是找某个值在一个 (允许重复值的) 集合中所处的百分点, 比如排在后 1/3、中间 1/3 或是前 1/3。虽然这样的查询可用构造 SQL 语句来完成, 但表达困难且计算效率低。编程人员通常借助于将部分代码写在 SQL 中, 另一部分代码写在程序设计语言中的方式去实现它。我们在这里讨论 SQL 中如何对这类查询进行直接表达。

在我们的大学例子中, 关系 *takes* 存储了每个学生所选的每一门课程上所获得的成绩。为了演示排名, 我们假设有一个视图 *student_grades(ID, GPA)*, 它给出了每个学生的平均绩点。^①

排名是用 **order by** 说明来实现的。下面的查询给出了每个学生的名次。

[192]

```
select ID, rank() over (order by (GPA) desc) as s_rank
from student_grades;
```

注意这里没有定义输出中的元组顺序, 所以元组可能不按名次排序。需要使用一个附加的 **order by** 子句得到排序的元组, 如下所示:

```
select ID, rank() over (order by (GPA) desc) as s_rank
from student_grades
order by s_rank;
```

有关排名的一个基本问题是如何处理多个元组在排序属性上具有相同值的情况。在我们的例子中, 这意味着如果有两个 GPA 相同的学生应如何处理的问题。**rank** 函数对所有在 **order by** 属性上相等的元组赋予相同的名次。例如, 如果两个学生具有相同的最高 GPA, 则两人都得到第 1 名。下一个给出的名次将是 3 而不是 2, 所以如果三个学生得到了次高的 GPA, 他们都将得到第 3 名, 接下来的一个或者多个学生将得到第 6 名, 等等。另有一个 **dense_rank** 函数, 它不在等级排序中产生隔阂。在上面的例子中, 具有次高成绩的元组都得到第 2 名, 具有第三高的值的元组得到第 3 名, 等等。

可以使用基本的 SQL 聚集函数来表达上述查询, 查询语句如下:

```
select ID, (1 + (select count(*)
                  from student_grades B
                  where B.GPA > A.GPA)) as s_rank
from student_grades A
order by s_rank;
```

显然, 正如上述语句所描述的那样, 一个学生的排名就是那些 GPA 比他高的学生的数目再加 1。然而, 计算每个学生的排名所耗时间与关系的大小呈线性增长, 导致整体耗时与关系大小呈平方量级增长。对于大型关系, 上述查询可能要花很长时间来执行。相比之下, **rank** 子句的系统实现能够对关系进行排序并在相当短的时间内计算排名。

排名可在数据的不同分区里进行。例如, 我们可能会希望按照系而不是在整个学校范围内对学生排名。假设有一个视图, 它像 *student_grades* 那样定义, 但是包含系名: *dept_grades(ID, dept_name, GPA)*。那么下面的查询就给出了学生们在每个分区里的排名:

[193]

① 用 SQL 语句来创建视图 *student_grades* 比较难, 因为我们必须把关系 *takes* 中的字母评分转换成数字, 并且要根据课程的学分来考虑该课成绩所占的权重。该视图的定义是在习题 4.5 中要做的。

```
select ID, dept_name,
       rank () over (partition by dept_name order by GPA desc) as dept_rank
from dept_grades
order by dept_name, dept_rank;
```

外层的 **order by** 子句将结果元组按系名排序，在各系内按照名次排序。

一个单独的 **select** 语句中可以使用多个 **rank** 表达式，因此，通过在同一 **select** 语句中使用两个 **rank** 表达式的方式，我们可以得到总名次以及系内的名次。当排名（可能带有分区）与 **group by** 子句同时出现的时候，**group by** 子句首先执行，分区和排名在 **group by** 的结果上执行。因此，得到聚集值之后可以用来做排名。本来也可以不用 *student_grades* 视图来完成排名查询，而是用一个单独的 **select** 子句来写。其细节留给读者作为习题。

利用将排名查询嵌入外层查询中的方法，排名函数可用于找出排名最高的 n 个元组，详细实现作为习题。注意，查找排名最低的 n 个元组与查找具有相反排序顺序的排名最高的 n 个元组是一样的。有些数据库系统提供非标准 SQL 扩展直接指定只需得到前 n 个结果，这样的扩展不需要 **rank** 函数且简化了优化器的工作。例如，一些数据库允许在 SQL 查询后面添加 **limit n** 子句，用来指明只输出前 n 个元组；这个子句可以与 **order by** 子句连用以获取排名最高的 n 个元组，如下面的查询所示，该查询以 GPA 大小顺序来获取前十个学生的 ID 和 GPA：

```
select ID, GPA
from student_grades
order by GPA
limit 10;
```

然而，**limit** 子句不支持分区，所以我们如果不进行排名的话就不能得到每个分区内的前 n 个元组；更甚的是，如果多个学生有相同的 GPA，很有可能其中某个学生被包括到前十，而另一个却不在其中。

还有其他几个可以用来替代 **rank** 的函数。例如，某个元组的 **percent_rank** 以分数的方式给出了该元组的名次。如果某个分区^①中有 n 个元组且某个元组的名次为 r ，则该元组的百分比名次定义为 $(r-1)/(n-1)$ （如果该分区中只有一个元组则定义为 **null**）。函数 **cume_dist**（累积分布的简写），对一个元组定义为 p/n ，其中 p 是分区中排序值小于或等于该元组排序值的元组数， n 是分区中的元组数。函数 **row_number** 对行进行排序，并且按行在排序顺序中所处位置给每行一个唯一行号，具有相同排序值的不同的行将按照非确定的方式得到不同的行号。

194

最后，对于一个给定的常数 n ，排名函数 **ntile(n)** 按照给定的顺序取得每个分区中的元组，并把它们分成 n 个具有相同元组数目的桶。^②对每个元组，**ntile(n)** 给出它所在的桶号（从 1 开始计数）。该函数对构造基于百分比的直方图特别有用。我们可以用下面的查询来演示根据 GPA 把学生分为四个等级：

```
select ID, ntile(4) over (order by (GPA desc)) as quartile
from student_grades;
```

空值的存在可能使排名的定义复杂化，这是因为我们不清楚空值应该出现在排序顺序的什么位置。SQL 允许用户通过使用 **nulls first** 或 **nulls last** 以指定它们出现的位置，例如：

```
select ID, rank() over (order by GPA desc nulls last) as s_rank
from student_grades;
```

5.5.2 分窗

窗口查询用来对于一定范围内的元组计算聚集函数。这个特性很有用，比如计算一个固定时间区间的聚集值；这个时间区间被称为一个窗口。窗口可以重叠，这种情况下一个元组可能对多个窗口都有贡献。这与此前我们看到的分区是不一样的，因为分区查询中一个元组只对一个分区有贡献。

① 如果没有使用显式的分区，则全部集合看成一个单独的分区。

② 如果一个分区中的所有元组的数量不能被 n 整除，则每个桶中的元组数最多相差 1。为了使每个桶中的元组数量相同，具有同样排序属性值的元组可能不确定地赋予不同的桶。

趋势分析是分窗的应用案例之一，这里考虑我们前面讲的销售货物的例子。由于天气等原因，销售量可能会一天天大幅度地变化(例如暴风雪、洪水、飓风、地震在一段时间内一般会降低销售量)。然而，经历足够长的一段时间，影响就会较小(继续前面的例子，由于天气原因造成的销售量下降可能会“赶上去”)。另一个使用分窗查询的例子是股票市场的趋势分析。我们在交易和投资的网站上可以看到各种各样的“移动平均线”。

要写查询来计算一个窗口的聚集值，用我们已经学到的那些特性是相对简单的，例如计算一个固定的三天时间区间的销售量。但是，如果我们想对每个三天时间区间都如此计算，那么查询就变得棘手了。

SQL 提供了分窗特性用于支持这样的查询。假设我们有一个视图 *tot_credits(year, num_credits)*，它记录了每年学生选课的总学分。^①注意，在这个关系中对于每个年份最多有一个元组。考虑如下查询： [195]

```
select year, avg(num_credits)
      over (order by year rows 3 preceding)
      as avg_total_credits
from tot_credits;
```

这个查询计算指定顺序下的前三个元组的均值。因此，对于2009年，如果关系 *tot_credits* 中含有2008年和2007年分别唯一对应的元组，该窗口定义所求的结果就是2007年、2008年和2009年的值的平均数。每年的均值也可以通过类似的方法来计算。对于关系 *tot_credits* 中最早的一年，均值的计算就只包含该年本身，然而对于第二年来说，需要对两年的值来求平均。注意，如果关系 *tot_credits* 在某个特定年份有不止一个元组，那么元组按年份来排序就有多种可能。在这种情况下，前序元组是基于排列顺序而定义的，该顺序取决于具体实现方法，就不是唯一定义了。

假设我们并不想回溯固定数量的元组，而是把前面所有年份都包含在窗口内。这意味着要关注的前面的年数并不恒定。我们编写下面的代码来得到前面所有年的平均总学分：

```
select year, avg(num_credits)
      over (order by year rows unbounded preceding)
      as avg_total_credits
from tot_credits;
```

也可以用关键字 **following** 来替换 **preceding**。如果我们在例子中这样做，*year* 值就表示窗口的起始年份，而不是结束年份。类似地，我们可以指定一个窗口，它在当前元组之前开始、在其之后结束：

```
select year, avg(num_credits)
      over (order by year rows between 3 preceding and 2 following)
      as avg_total_credits
from tot_credits;
```

我们还可以用 **order by** 属性上值的范围而不是用行的数目来指定窗口。为了得到一个包含当前年以及前面四年的范围，我们可以这样写： [196]

```
select year, avg(num_credits)
      over (order by year range between year - 4 and year)
      as avg_total_credits
from tot_credits;
```

一定要注意上例中的关键字 **range** 的使用。对于2010年，从2006年到2010年(包括边界)的数据都被返回，不管该范围内实际有多少元组。

在我们的例子里，所有元组都是针对整个学校而言的。假如不是这样，而是把每个系的学分数据用视图 *tot_credits_dept(dept_name, year, num_credits)* 来表示，它记录了学生在指定年份中选某个系开设的课程的所有学分数。(我们仍旧把对该视图的定义留给读者作为练习。)我们编写如下的分窗查询，按照 *dept_name* 分区，对每个系分别计算：

① 按照本书中大学的例子来定义这个视图，我们把这个作为练习留给读者。


```

select dept_name, year, avg(num_credits)
      over (partition by dept_name
            order by year rows between 3 preceding and current row)
      as avg_total_credits
from tot_credits_dept;

```

5.6 OLAP**

联机分析处理(OLAP)系统是一个交互式系统,它允许分析人员查看多维数据的不同种类的汇总数据。联机一词表示分析人员必须能够提出新的汇总数据的请求,几秒钟之内在线得到响应,无需为了看到查询结果而被迫等待很长的时间。

有很多可用的 OLAP 产品,其中有些随 Microsoft SQL Server 和 Oracle 等数据库产品绑定,另一些是独立的工具。许多 OLAP 工具的最初版本以数据驻留在内存中为前提。小规模数据可以用电子表格进行分析,例如 Excel。然而,对于超大规模数据的 OLAP,需要把数据存储在数据库中,并通过数据库的支持来高效地完成数据预处理和在线查询处理。在本节中,我们将研究 SQL 支持上述任务的扩展特性。

5.6.1 联机分析处理

考虑这样一个应用,某商店想要找出流行的服装款式。我们假设衣服的特性包括商品名、颜色和尺寸,并且我们有一个关系 *sales*, 它的模式是

sales(*item_name*, *color*, *clothes_size*, *quantity*)

假设 *item_name* 可以取的值为 skirt、dress、shirt、pants; *color* 可以取的值为 dark、pastel、white; *clothes_size* 可以取的值为 small、medium、large; *quantity* 是一个整数值,表示一个给定条件 |*item_name*, *color*, *clothes_size*| 的商品数量。关系 *sales* 的一个实例如图 5-16 所示。

统计分析通常需要对多个属性进行分组。给出一个用于


数据分析的关系,我们可以把它的某些属性看作度量属性(measure attribute),因为这些属性度量了某个值,而且可以在其上进行聚集操作。例如,关系 *sales* 的属性 *quantity* 就是一个度量属性,因为它度量了卖出商品的数量。关系的其他属性中的某些(或所有)属性可看作是维属性(dimension attribute),因为它们定义了度量属性以及度量属性的汇总可以在其上观察的各个维度。在关系 *sales* 中, *item_name*、*color* 和 *clothes_size* 是维属性。(关系 *sales* 的更真实的版本还包括另一些维属性,例如时间和销售地点,以及其他的度量属性,例如该次销售的货币值。)

能够模式化为维属性和度量属性的数据统称为多维数据(multidimensional data)。

为了分析多维数据,管理者可能需要查看如图 5-17 所示那样显示的数据。该表显示了 *item_name* 和 *color* 值之间不同组合情况下的商品数目。*clothes_size* 的值指定为 all,意味着表中显示的数值是在所有的 *size* 值上的数据汇总(也就是说,我们把“small”、“medium”和“large”的商品都汇总到一个组里)。

item_name	color	clothes_size	quantity
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2
skirt	white	medium	5
skirt	white	large	3
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
shirt	dark	small	2
shirt	dark	medium	6
shirt	dark	large	6
shirt	pastel	small	4
shirt	pastel	medium	1
shirt	pastel	large	2
shirt	white	small	17
shirt	white	medium	1
shirt	white	large	10
pants	dark	small	14
pants	dark	medium	6
pants	dark	large	0
pants	pastel	small	1
pants	pastel	medium	0
pants	pastel	large	1
pants	white	small	3
pants	white	medium	0
pants	white	large	2

图 5-16 关系 *sales* 的例子

clothes_size 

	<i>color</i>			
<i>item_name</i>	dark	pastel	white	all
	8	35	10	53
	20	10	5	35
	14	7	28	49
	20	2	5	27
	all	all	all	all
	62	54	48	164

图 5-17 关系 *sales* 的关于 *item_name* 和 *color* 的交叉表

图 5-17 所示的表是一个交叉表 (cross-tabulation 或简称为 cross-tab) 的例子, 也可称之为转轴表 (pivot-table)。一般说来, 一个交叉表是从一个关系 (如 *R*) 导出来的, 由关系 *R* 的一个属性 (如 *A*) 的值构成其行表头, 关系 *R* 的另一个属性 (如 *B*) 的值构成其列表头。例如, 在图 5-17 中, 属性 *item_name* 对应 *A* (属性值为 “skirt”、“dress”、“shirt”和 “pants”), 而属性 *color* 对应 *B* (取值 “dark”、“pastel”和 “white”)。

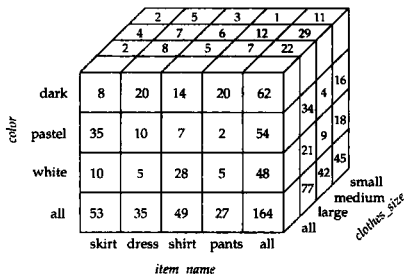
每个单元可记为 (a_i, b_j) , 其中 a_i 代表 *A* 的一个取值, b_j 代表 *B* 的一个取值。转轴表中不同单元的值按照如下方法从关系 *R* 推导出来: 如果对于任何 (a_i, b_j) 值, 最多只存在一个元组具有该值, 则该单元中的值由那个元组得到 (如果存在那个元组的话)。例如, 它可以是该元组中一个或多个其他属性的值。如果对于一个 (a_i, b_j) 值, 可能存在多个元组具有该值, 则该单元中的值必须由这些元组上的聚集得到。在我们的例子中, 所用的聚集是所有不同 *clothes_size* 上的 *quantity* 属性值之和, 如图 5-17 交叉表上方的 “*clothes_size*: all” 所表明的那样。这样, 单元 (skirt, pastel) 的值就是 35, 因为关系 *sales* 有三个元组满足该条件, 它们的值分别为 11、9 和 15。

在我们的例子中, 交叉表还另有一列和一行, 用来存储一行/列中所有单元的总和。大多数的交叉表中都有这样的汇总行和汇总列。

将二维的交叉表推广到 *n* 维, 可视为一个 *n* 维立方体, 称为数据立方体 (data cube)。图 5-18 表示一个在 *sales* 关系上的数据立方体。该数据立方体有三个维, 即 *item_name*、*color* 和 *clothes_size*, 其度量属性是 *quantity*, 每个单元由这三个维的值确定。正如交叉表一样, 数据立方体中的每个单元包含了一个值。在图 5-18 中, 一个单元包含的值显示在该单元的一个面上; 该单元其他可见的面显示为空白。所有的单元都包含值, 即使它们并不可见。某一维的取值可以是 all, 此时就如交叉表显示的情形一样, 该单元包含了该维上所有值的汇总数据。

将元组分组做聚集的不同方式有很多。在图 5-18 的例子中, 有三种颜色、四类商品以及三种尺码, 从而立方体的大小为 $3 \times 4 \times 3 = 36$ 。要是把汇总值也包括进来, 我们就得到一个 $4 \times 5 \times 4$ 的立方体, 其大小为 80。事实上, 对于一个 *n* 维的表, 可在其 *n* 个维^①的 2^n 个子集上进行分组并执行聚集操作。

在 OLAP 系统中, 数据分析人员可以交互地选择交叉表的属性, 从而能够在相同的数据上查看不同内容的交叉表。每个交叉表是一个多维数据立方体上的二维视图。例如, 数据分析人员可以选择一个在 *item_name* 和 *clothes_size* 上的交叉表, 也可以选择一个在 *color* 和 *clothes_size* 上的交叉表。改变交叉表中的维的操作叫做转轴 (pivoting)。



			2	5	3	1	11	
		4	7	6	12	29		
	2	8	5	7	22			
dark	8	20	14	20	62	34	4	16
pastel	35	10	7	2	54	21	9	18
white	10	5	28	5	48	77	42	45
all	53	35	49	27	164			
		skirt	dress	shirt	pants	all		
							small	
							medium	
							large	

图 5-18 三维数据立方体

① 在所有的 *n* 个维的集合上分组只有在表含有重复数据的情况下才是有意义的。

OLAP 系统允许分析人员对于一个固定的 *clothes_size* 值(比如 large 而不是所有 size 的总和)来查看一个在 *item_name* 和 *color* 上的交叉表。这样的操作称为切片(slicing),因为它可以看作是查看一个数据立方体的某一片。该操作有时也叫做切块(dicing),特别是当有多个维的值是固定的时候。

当一个交叉表用于观察一个数据立方体时,不属于交叉表部分的维属性的值显示在交叉表的上方。如图 5-17 所示,这样的属性值可以是 all,表示交叉表中的数据是该属性所有值的汇总。切片/切块仅由这些属性所选的特定值构成,这些值显示在交叉表的上方。

OLAP 系统允许用户按照期望的粒度级别观察数据。从较细粒度数据转到较粗粒度(通过聚集)的操作叫做上卷(rollup)。在我们的例子中,从表 *sales* 上的数据立方体出发,在属性 *clothes_size* 上执行上卷操作得到我们例子中的交叉表。相反的操作,也就是将较粗粒度数据转化为较细粒度数据,称作下钻(drill down)。显然,较细粒度数据不可能由较粗粒度数据产生;它们必须由原始数据产生,或者由更细粒度的汇总数据产生。

分析人员可能希望从不同的细节层次观察某一个维。例如,类型为 **datetime** 的属性包括一天的日期和时间。对于一个只对粗略时间感兴趣的分析人员来说,使用精确到秒(或者更小)的时间可能是无意义的,因为他可能只查看小时的值。一个对一周中某天的销售情况感兴趣的分析人员可能会将日期映射到一周的某天并只查看映射后的信息。还有的分析人员可能会对一个月、一个季度或一年的聚集数据感兴趣。

一个属性的不同细节层次可以以组织成一个层次结构(hierarchy)。图 5-19a 显示了一个在 **datetime** 属性上的层次结构。作为另一个例子,图 5-19b 显示了地理位置的层次结构:city 在层次结构的最底层, state 在它上层, country 在更上一层, region 在最顶层。在前面的例子中,衣服可以按类别分组(例如,男装或女装),这样,在关于衣服的层次结构中, *category* 将在 *item_name* 之上。在实际值层面上,裙子和女服在女装类别下面,短裤和衬衫在男装类别下面。

分析人员可能对查看区分为男装和女装的衣服销售情况感兴趣,而对单个值不感兴趣。在查看了女装和男装层次上的聚集值之后,分析人员可能想要沿层次结构下钻以便查看单个值。一个正在查看某个细节层次的分析人员也可能沿层次结构上卷以便查看较粗层次上的聚集值。两种层次可以显示在同一个交叉表中,如图 5-20 所示。

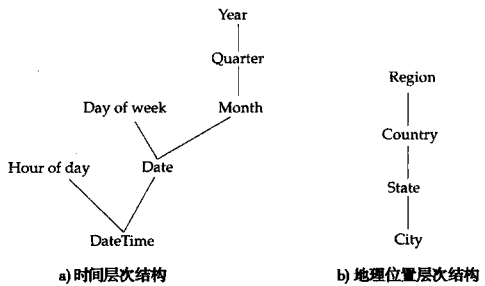


图 5-19 不同的维上的层次结构

分析人员可能对查看区分为男装和女装的衣服销售情况感兴趣,而对单个值不感兴趣。在查看了女装和男装层次上的聚集值之后,分析人员可能想要沿层次结构下钻以便查看单个值。一个正在查看某个细节层次的分析人员也可能沿层次结构上卷以便查看较粗层次上的聚集值。两种层次可以显示在同一个交叉表中,如图 5-20 所示。

5.6.2 交叉表与关系表

交叉表与通常存储在数据库中的关系表是不同的,这是因为交叉表中的列的数目依赖于实际的数据。数据值的变化可能导致增加更多的列,这对于数据存储来说是不期望见到的。然而,作为向用户做的展现,交叉表是比较令人满意的。对于没有汇总值的交叉表,可以把它直接表示为具有固定列数的关系形式。对于有汇总行/列的交叉表,可以通过引入一个特殊值 all 以表示子汇总值,如图 5-21 示。实际上,SQL 标准使用 null 值代替 all,但是,为了避免与通常的 null 值混淆,我们将继续沿用 all。

考虑元组(skirt, all, all, 53)和(dress, all, all, 35)。这两个元组是这样得到的:我们消除在 *color* 和 *clothes_size* 上具有不同值的各个元组,然后将 *quantity* 的值替换成一个聚集值(即数量的和),

clothes_size: all

category	item_name	color			total
womenswear	skirt	8	8	10	53
	skirt	20	20	5	35
	subtotal	28	28	15	88
menswear	pants	14	14	28	49
	shirt	20	20	5	27
	subtotal	34	34	33	76
total		62	62	48	164

图 5-20 对 *sales* 在属性 *item_name* 上分层后的交叉表

这样就得到了这两个元组。值 **all** 可以被看作是代表一个属性的所有值的集合。在 *color* 和 *clothes_size* 这两个维上具有 **all** 值的元组可以通过在关系 *sales* 的列 *item_name* 上进行 **group by** 操作之后进行聚集得到。类似地，在 *color*、*clothes_size* 上进行 **group by** 可用于得到在 *item_name* 上取 **all** 值的元组，不带任何属性的 **group by** (在 SQL 中直接将其省略) 可用于得到在 *item_name*、*color* 和 *clothes_size* 上都取 **all** 值的元组。

也可以用关系来表示层次结构。例如，裙子和女服属于女装类，短裤和衬衫属于男装类，这样的层次可以用关系 *itemcategory*(*item_name*, *category*) 来表示。可以把该关系与关系 *sales* 连接，得到一个包含各个商品所属类别的关系。在该连接关系上进行聚集操作，我们就可以得到带有层次结构的交叉表。另一个例子是，城市的层次结构可以用单个关系 *city_hierarchy*(*ID*, *city*, *state*, *country*, *region*) 来表示，也可以用多个关系表示，每个关系把结构中的某层的值映射到其下一层。我们在这里假设城市有唯一的标识符，它存储在属性 *ID* 中，用来避免当两个城市名称相同时发生混淆，比如，在美国的伊利诺斯州和密苏里州都有叫做 Springfield 的城市。

item_name	color	clothes_size	quantity
skirt	dark	all	8
skirt	pastel	all	35
skirt	white	all	10
skirt	all	all	53
dress	dark	all	20
dress	pastel	all	10
dress	white	all	5
dress	all	all	35
shirt	dark	all	14
shirt	pastel	all	7
shirt	white	all	28
shirt	all	all	49
pants	dark	all	20
pants	pastel	all	2
pants	white	all	5
pants	all	all	27
all	dark	all	62
all	pastel	all	54
all	white	all	48
all	all	all	164

图 5-21 图 5-17 中数据的关系表达

OLAP 的实现

最早的 OLAP 系统使用内存中的多维数组存储数据立方体，称作多维 OLAP (multidimensional OLAP, MOLAP) 系统。后来，OLAP 工具集成到关系系统中，数据存储到关系数据库里，这样的系统称为关系 OLAP (relational OLAP, ROLAP) 系统。复合系统将一些汇总数据存储在内存中，基表数据和另一些汇总数据存储在关系数据库中，称之为混合 OLAP (hybrid OLAP, HOLAP) 系统。

许多 OLAP 系统实现为客户-服务器系统。服务器端包含关系数据库和任何 MOLAP 数据立方体，客户端系统通过与服务器通信获得数据的视图。

在一个关系上计算整个数据立方体(所有的分组)的一种朴素方法是使用任何一种标准算法来计算聚集操作，一次计算一个分组。朴素算法需要对关系进行大量的扫描操作。有一种简单的优化是从聚集(*item_name*, *color*, *clothes_size*)中，而不是从原始关系中计算聚集(*item_name*, *color*)。

对于标准的 SQL 聚集函数，我们可以用一个按属性集 *B* 分组的聚集来计算按属性集 *A* 进行分组的聚集(若 $A \subseteq B$)。读者可以以此作为练习(见习题 5.24)，但是要注意计算 **avg** 时，我们还需要 **count** 值。(对于一些非标准的聚集函数，比如 **median**，不能像上面那样计算聚集，这里所说的优化不能应用于这样的非可分解的聚集函数。)从另一个聚集而不是从原始关系中计算聚集可以大大减少所读的数据量。更进一步的改进也是有可能的，例如，通过一遍数据扫描计算多个分组。

早期的 OLAP 实现预先计算并存储整个数据立方体，即对多维性的所有子集进行分组。预先计算可以使 OLAP 查询在几秒钟内得出结果，即使数据集可能包含数百万的元组，总计达上 G 的数据。然而，对于 *n* 维属性，有 2^n 个分组，属性上的层次更增多了这个数量。这导致整个数据立方体通常大于形成数据立方体的原始关系，而且在许多情况下存储整个数据立方体是不可行的。

作为对预先计算并存储所有可能的分组的替代方案，预先计算并存储某些分组，按需计算其他分组是有意义的。从原始关系中计算查询可能需要相当长的时间，取而代之的做法是从其他预先计算的查询中计算这些查询。例如，假设某个查询要求按(*item_name*, *color*)进行分组，它没有预先计算。该查询的结果可以基于(*item_name*, *color*, *clothes_size*)上的汇总计算得出，假设这是我们预先计算了的。对于在考虑用于存储预先计算结果的可用存储限制的条件下如何选择一个好的用于预先计算的分组集，请查看相关文献注解。

199

203

204

5.6.3 SQL 中的 OLAP

有些 SQL 实现，例如 Microsoft SQL Server 和 Oracle，支持在 SQL 中使用 **pivot** 子句，用于创建交叉表。对于图 5-16 中的关系 *sales*，使用如下查询：

```
select
from sales
pivot (
    sum(quantity)
    for color in ('dark', 'pastel', 'white')
)
order by item_name;
```

可以返回图 5-22 所示的交叉表。注意，**pivot** 子句中的 **for** 子句用来指定属性 *color* 的哪些值应该在转轴的结果中作为属性名出现。属性 *color* 本身从结果中被去除，然而其他所有属性都被保留，另外，新产生的属性的值被设定为来自于属性 *quantity*。在多个元组对给定单元有贡献的情况下，**pivot** 子句中的聚集操作指定了把这些值进行汇总的方法。在上面的例子中，对 *quantity* 值求和。

注意，**pivot** 子句本身并不计算我们在图 5-17 所示的转轴表中所看到的部分和。但是，可以首先像我们马上就要讲的那样生成图 5-21 所示的关系表达，然后对这个表达应用 **pivot** 子句，以得到同样的结果。这种情况下，**all** 值必须也被列在 **for** 子句中，并且 **order by** 子句需要做调整使得 **all** 被排在最后。

单个 SQL 查询使用基本的 **group by** 结构生成不了数据立方体中的数据，因为聚集是对维属性的若干个不同分组来计算的。由于这个原因，SQL 包含了一些函数，用来生成 OLAP 所需的分组。下面我们讨论这些特性。

SQL 支持 **group by** 结构的泛化形式用于进行 **cube** 和 **rollup** 操作。**group by** 子句中的 **cube** 和 **rollup** 结构允许在单个查询中运行多个 **group by** 查询，结果以单个关系的形式返回，其样式类似于图 5-21 中的关系。

再次以零售商店为例，关系是

```
sales (item_name, color, clothes_size, quantity)
```

我们可以编写简单的 **group by** 查询来得到每个商品名所对应商品的销售量：

```
select item_name, sum(quantity)
from sales
group by item_name;
```

这个查询的结果如图 5-23 所示。注意，这和图 5-17 的最后一列（或者同样可以说，图 5-18 所示立方体的第一行）表达的是同一组数据。

类似地，我们可以得到每种颜色的商品的销售量，等等。在 **group by** 子句中用多个属性，我们就知道在某个参数集合条件下每类商品卖出了多少。例如，我们可以编写下面的代码通过商品名和颜色来拆分关系 *sales*：

```
select item_name, color, sum(quantity)
from sales
group by item_name, color;
```

item_name	color	quantity
skirt	small	2
skirt	medium	5
skirt	large	1
dress	small	2
dress	medium	6
dress	large	12
shirt	small	2
shirt	medium	6
shirt	large	6
pants	small	14
pants	medium	6
pants	large	0

图 5-22 对图 5-16 中的关系 *sales* 进行 SQL **pivot** 操作的结果

图 5-24 给出了上述查询的结果。注意，这和图 5-17 的前四行和前四列（或者同样可以说，图 5-18 所示的立方体的前四行和前四列）表达的是同一组数据。

但是，如果想用这种方式生成整个数据立方体，我们就不得不为以下每个属性集写一个独立的

查询：

```
| (item_name, color, clothes_size), (item_name, color), (item_name, clothes_size),
  (color, clothes_size), (item_name), (color), (clothes_size), () |
```

() 表示空的 **group by** 列表。

cube 结构使我们能在一个查询中完成这些任务：

```
select item_name, color, clothes_size, sum(quantity)
from sales
group by cube(item_name, color, clothes_size);
```

上述查询产生了一个模式如下的关系：

```
(item_name, color, clothes_size, sum(quantity))
```

这样，该查询结果实际上是一个关系，对于在特定分组中不出现的属性，在结果元组中包含 **null** 作为其值。例如，在 *clothes_size* 上进行分组所产生的元组模式是 (*clothes_size*, **sum(quantity)**)，通过在属性 *item_name* 和 *color* 上加入 **null**，它们被转换成 (*item_name*, *color*, *clothes_size*, **sum(quantity)**) 上的元组。

item_name	color	quantity
skirt	dark	8
skirt	pastel	35
skirt	white	10
dress	dark	20
dress	pastel	10
dress	white	5
shirt	dark	14
shirt	pastel	7
shirt	white	28
pants	dark	20
pants	pastel	2
pants	white	5

图 5-24 查询结果

数据立方体关系经常相当庞大。上面的 **cube** 查询，有 3 种可能的颜色、4 种可能的商品名和 3 种尺寸，共有 80 个元组。图 5-21 的关系是由 *item_name* 和 *color* 上的分组操作所生成的。它也用 **all** 来替代空值，这样更容易被一般用户读懂。为了在 SQL 中生成该关系，我们设法把 **null** 替换成 **all**。查询：

```
select item_name, color, sum(quantity)
from sales
group by cube(item_name, color);
```

207

生成图 5-21 中的关系，含有 **null**。使用 SQL 的 **decode** 和 **grouping** 函数可以实现 **all** 的替换。**decode** 函数概念简单但是语法有些难以读懂。细节请查看下面的文本框中的内容。

DECODE 函数

decode 函数用来对元组中的属性进行值的替换。**decode** 的大体形式是：

```
decode(value, match-1, replacement-1, match-2, replacement-2, ...,
       match-N, replacement-N, default-replacement);
```

它把值 (*value*) 与匹配 (*match*) 值进行比较。如果发现匹配，就把属性值用相应的替代值来替换。如果未匹配成功，那么属性值被替换成默认的替代值 (*default-replacement*)。

decode 函数并不像我们期望的那样对 **null** 值进行处理，这是因为，正如我们在 3.6 节所看到的那样，**null** 上的谓词等价于 **unknown**，最终被转换成 **false**。为了解决这个问题，我们应用 **grouping** 函数，当参数是 **cube** 或 **rollup** 产生的 **null** 值时它返回 1，否则返回 0。于是，图 5-21 中的关系可以用下面的查询来计算，**null** 替换成 **all** 了：

```
select decode(grouping(item_name), 1, 'all', item_name) as item_name,
       decode(grouping(color), 1, 'all', color) as color,
       sum(quantity) as quantity
from sales
group by cube(item_name, color);
```

rollup 结构与 **cube** 结构基本一样，只有一点不同，就是 **rollup** 产生的 **group by** 查询较少一些。我们看到，**group by cube** (*item_name*, *color*, *clothes_size*) 生成使用部分属性 (或者全部属性，或者不用任何属性) 可以产生的全部 8 种实现 **group by** 查询的方法。而在下面的查询中：

```
select item_name, color, clothes_size, sum(quantity)
from sales
group by rollup(item_name, color, clothes_size);
```

group by rollup (*item_name*, *color*, *clothes_size*) 只产生四个分组：

```
{ (item_name, color, clothes_size), (item_name, color), (item_name), () }
```

注意, **rollup** 中的属性按照顺序不同有所区分; 最后一个属性 (在我们例子中是 *clothes_size*) 只在一个分组中出现, 倒数第二个属性出现在两个分组中, 以此类推, 第一个属性出现在 (除了空分组之外的) 所有组中。

[208]

我们用 **rollup** 产生的特定分组有什么用? 这些组对于层次结构 (例如图 5-19 所示) 具有频繁的实用价值。对于位置的层次结构 (*Region*, *Country*, *State*, *City*), 我们可能希望用 *Region* 来分组, 以得到不同区域的销售情况。之后我们可能希望“下钻”到区域内部的国家层面, 也就是说我们将来以 *Region* 和 *Country* 来分组。继续下钻, 我们希望以 *Region*、*Country* 和 *State* 分组, 然后以 *Region*、*Country*、*State* 和 *City* 来分组。**rollup** 结构允许我们为求更深层的细节而设定这样下钻的序列。

多个 **rollup** 和 **cube** 可以在一个单独的 **group by** 子句中使用。例如, 下面的查询

```
select item_name, color, clothes_size, sum(quantity)
from sales
group by rollup(item_name), rollup(color, clothes_size);
```

产生了以下分组:

```
{ (item_name, color, clothes_size), (item_name, color), (item_name),
  (color, clothes_size), (color), () }
```

为了理解其原因, 我们注意到 **rollup**(*item_name*) 产生了两个分组: { (*item_name*), () }, **rollup**(*color*, *clothes_size*) 产生了三个分组: { (*color*, *clothes_size*), (*color*), () }。这两部分的笛卡儿积得到上面显示的六个分组。

rollup 和 **cube** 子句都不能完全控制分组的产生。例如, 我们不能指定让它们只生成分组 { (*color*, *clothes_size*), (*clothes_size*, *item_name*) }。我们可以用 **having** 子句中使用 **grouping** 结构的方法去产生这类限制条件的分组, 实现细节作为习题留给读者。

5.7 总结

- SQL 查询可以从宿主语言通过嵌入和动态 SQL 激发。ODBC 和 JDBC 标准给 C、Java 等语言的应用程序定义接入 SQL 数据库的应用程序接口。程序员越来越多地通过这些 API 来访问数据库。
- 函数和过程可以用 SQL 提供的过程扩展来定义, 它允许迭代和条件 (if-then-else) 语句。
- 触发器定义了当某个事件发生而且满足相应条件时自动执行的动作。触发器有很多用处, 例如实现业务规则、审计日志, 甚至执行数据库系统外的操作。虽然触发器只是在不久前作为 SQL:1999 的一部分加入 SQL 标准的, 但是大多数数据库系统已经支持触发器很久了。
- 一些查询, 如传递闭包, 或者可以用迭代表示, 或者可以用递归 SQL 查询表示。递归可以用递归视图, 或者用递归的 **with** 子句定义。
- SQL 支持一些高级的聚集特性, 包括排名和分窗查询, 这些特性简化了一些聚集操作的表达方式, 并提供了更高效的求值方法。
- 联机分析处理 (OLAP) 工具帮助分析人员用不同的方式查看汇总数据, 使他们能够洞察一个组织的运行。
 1. OLAP 工具工作在以维属性和度量属性为特性的多维数据之上。
 2. 数据立方体由以不同方式汇总的多维数据构成。预先计算数据立方体有助于提高汇总数据的查询速度。
 3. 交叉表的显示允许用户一次查看多维数据的两个维及其汇总数据。
 4. 下钻、上卷、切片和切块是用户使用 OLAP 工具时执行的一些操作。
- 从 SQL:1999 标准开始, SQL 提供了一系列的用于数据分析的操作符, 其中包括 **cube** 和 **rollup** 操作。有些系统还支持 **pivot** 子句, 可以很方便地生成交叉表。

[209]

术语回顾

- | | | |
|--------|---------|-----------|
| • JDBC | • 预备语句 | • SQL 注入 |
| • ODBC | • 访问元数据 | • 嵌入式 SQL |

- 游标
- 可更新的游标
- 动态 SQL
- SQL 函数
- 存储过程
- 过程化结构
- 外部语言例程
- 触发器
- before 和 after 触发器
- 过渡变量和过渡表
- 递归查询
- 单调查询
- 排名函数
 - Rank
 - Dense rank
 - Partition by
- 分窗
- 联机分析处理(OLAP)
- 多维数据
 - 度量属性
 - 维属性
 - 转轴
 - 数据立方体
 - 切片和切块
 - 上卷和下钻
- 交叉表

实践习题

- 5.1 描述何种情况下你会选择使用嵌入式 SQL，而不是仅仅使用 SQL 或某种通用程序设计语言。
- 5.2 写一个使用 JDBC 元数据特性的 Java 函数，该函数用 ResultSet 作为输入参数，并把结果输出为用合适的名字作为列名的表格形式。
- 5.3 写一个使用 JDBC 元数据特性的 Java 函数，该函数输出数据库中的所有关系列表，为每个关系显示它的属性的名称和类型。
- 5.4 说明如何用触发器来保证约束“一位教师不可能在一个学期的同一时间段在不同的教室里教课”。(要知道，对关系 *teaches* 或 *section* 的改变都可能使该约束被破坏。)
- 5.5 写一个触发器，用于当 *section* 和 *time_slot* 更新时维护从 *section* 到 *time_slot* 的参照完整性约束。注意，我们在图 5-8 中写的触发器不包含更新操作。
- 5.6 为了维护关系 *student* 的 *tot_cred* 属性，完成下列任务：
 - a. 修改定义在 *takes* 更新时的触发器，使其对于能影响 *tot_cred* 值的所有更新都有效。
 - b. 写一个能与关系 *takes* 的插入操作相关的触发器。
 - c. 在什么前提下，关系 *course* 上不创建触发器是有道理的？
- 5.7 考虑图 5-25 中的银行数据库。视图 *branch_cust* 定义如下：

```
create view branch_cust as
select branch_name, customer_name
from depositor, account
where depositor.account_number = account.account_number
```

假设视图被物化，也就是，这个视图被计算且存储。写触发器来维护这个视图，也就是，该视图在对关系 *depositor* 或 *account* 的插入和删除时保持最新。不必管更新操作。

210
211

```
branch(branch_name, branch_city, assets)
customer(customer_name, customer_street, customer_city)
loan(loan_number, branch_name, amount)
borrower(customer_name, loan_number)
account(account_number, branch_name, balance)
depositor(customer_name, account_number)
```

图 5-25 习题 5.7、习题 5.8 和习题 5.28 中用到的银行数据库

- 5.8 考虑图 5-25 中的银行数据库。写一个 SQL 触发器来执行下列动作：在对账户执行 delete 操作时，对账户的每一个拥有者，检查他是否有其他账户，如果没有，把他从 *depositor* 关系中删除。
- 5.9 说明如何使用 rollup 表达 group by cube(a, b, c, d)；答案只允许含有一个 group by 子句。
- 5.10 对于给定的一个关系 *S(student, subject, marks)*，写一个查询，利用排名操作找出总分数排在前 n 位的学生。
- 5.11 考虑 5.6 节中的关系 *sales*，写一个 SQL 查询，计算该关系上的立方体(cube)操作，给出图 5-21 中的关系。不要使用 cube 结构。

习题

- 5.12 考虑有如下关系的雇员数据库：

- *emp*(*ename*, *dname*, *salary*)
- *mgr*(*ename*, *mname*)

和图 5-26 中的 Java 代码，该代码使用 JDBC 应用程序接口。假设用户 id、密码、主机名等都正确。请用简洁的语言描述一下 Java 程序都做了什么。（也就是说，用类似于“查找玩具部门的经理”这样的话来表达，而不是一行一行地对 Java 语句进行解释。）

```
import java.sql.*;
public class Mystery {
    public static void main(String[] args) {
        try {
            Connection con=null;
            Class.forName("oracle.jdbc.driver.OracleDriver");
            con=DriverManager.getConnection(
                "jdbc:oracle:thin:star/X@//edgar.cse.lehigh.edu:1521/XE");
            Statement s=con.createStatement();
            String q;
            String empName = "dog";
            boolean more;
            ResultSet result;
            do {
                q = "select mname from mgr where ename = " + empName + " ";
                result = s.executeQuery(q);
                more = result.next();
                if (more) {
                    empName = result.getString("mname");
                    System.out.println(empName);
                }
            } while (more);
            s.close();
            con.close();
        } catch (Exception e) {e.printStackTrace();} }
}
```

图 5-26 习题 5.12 的 Java 代码

- 5.13 假设你要在 Java 中定义一个 *MetaDisplay* 类，它包含方法 *static void printTable*(*String r*)；该方法以关系 *r* 为输入参数，执行查询“*select from r*”，然后以合适的表格来显示输出结果，表头代表每一列的属性名。
- a. 要想以指定的表格形式输入结果，你需要知道关系 *r* 的哪些信息？
 - b. JDBC 的哪个（些）函数能帮你获取所需的信息？
 - c. 用 JDBC 应用程序接口来编写方法 *printTable*(*String r*)。
- 5.14 用 ODBC 重新做习题 5.13，定义函数 *void printTable*(*char* r*) 来代替原题的方法。
- 5.15 考虑有两个关系的雇员数据库

```
employee(employee_name, street, city)
works(employee_name, company_name, salary)
```

这里主码用下划线标出。写出一个查询找出这样的公司，它的雇员的平均工资比“First Bank Corporation”的平均工资要高。

- a. 使用合适的 SQL 函数。
 - b. 不使用 SQL 函数。
- 5.16 使用 *with* 子句而不是函数调用来重写 5.2.1 节的查询，返回教师数大于 12 的系的名称和预算。
- 5.17 把使用嵌入式 SQL 与在 SQL 中使用定义在一个通用程序设计语言中的函数这两种情况进行比较。在什么情况下你会考虑用哪个特性？
- 5.18 修改图 5-15 中的递归查询来定义一个关系

```
prereq_depth (course_id, prereq_id, depth)
```

这里属性 *depth* 表示在课程和先修课程之间存在多少层中间的先修关系。直接的先修课程的深度为 0。

5.19 考虑关系模式

```
part(part_id, name, cost)
subpart(part_id, subpart_id, count)
```

关系 *subpart* 中的一个元组 ($p_1, p_2, 3$) 表示部件编号为 p_2 的部件是部件编号为 p_1 的部件的直接子部件, 并且 p_1 中包含 3 个 p_2 。注意 p_2 本身可能有自己的子部件。写一个递归 SQL 查询输出编号为“P-100”的部件的所有子部件。

5.20 再一次考虑习题 5.19 中的关系模式。用非递归 SQL 写一个 JDBC 函数来找出“P-100”的总成本, 包括它的所有子部件的成本。注意考虑一个部件可能有重复出现多次的同一个子部件的情况。如果需要, 可以在 Java 中使用递归。

5.21 假设有两个关系 r 和 s , r 的外码 B 参照 s 的主码 A 。描述如何用触发器实现从 s 中删除元组时的 **on delete cascade** 选项。

5.22 一个触发器的执行可能会引发另一个被触发的动作。大多数数据库系统都设置了嵌套的深度限制。解释为什么它们要设置这样的限制。

5.23 考虑图 5-27 中的关系 r 。请给出下面查询的结果。

```
select building, room_number, time_slot_id, count(*)
from r
group by rollup (building, room_number, time_slot_id)
```

building	room_number	time_slot_id	course_id	sec_id
Garfield	359	A	BIO-101	1
Garfield	359	B	BIO-101	2
Saucon	651	A	CS-101	2
Saucon	550	C	CS-319	1
Painter	705	D	MU-199	1
Painter	403	D	FIN-201	1

图 5-27 习题 5.23 中的关系 r

5.24 对 SQL 聚集函数 **sum**、**count**、**min** 和 **max** 中的每一个, 说明在给定多重集合 S_1 和 S_2 上的聚集值的条件下, 如何计算多重集合 $S_1 \cup S_2$ 上的聚集值。

在上述的基础之上, 在给定属性 $T \supseteq S$ 上的分组聚集值的条件下, 对下面的聚集函数, 给出计算关系 $r(A, B, C, D, E)$ 的属性的子集 S 上的分组聚集值的表达式。

a. **sum**、**count**、**min** 和 **max**。

b. **avg**。

c. 标准差。

5.25 在 5.5.1 节中, 我们使用习题 4.5 中的视图 *student_grades* 来写基于绩点平均值对学生排名次的查询。修改这个查询, 只显示前十名的学生(也就是说, 从第一名到第十名的那些学生)。

5.26 给出一个具有两个分组的例子, 它不能使用带有 **cube** 和 **rollup** 的单个 **group by** 子句表达。

5.27 对于给定的一个关系 $r(a, b, c)$, 说明如何使用扩展 SQL 特性产生一个 c 对 a 的直方图。将 a 分成 20 个等分的区(即每个区含有 r 中 5% 的元组, 并按 a 排序)。

5.28 考虑图 5-25 中的银行数据库, 以及关系 *account* 的 *balance* 属性。写一个 SQL 查询, 计算 *balance* 值的直方图, 从 0 到目前最大账户余额的范围分成三个相等的区域。

工具

大部分数据库销售商把 OLAP 工具当作他们数据库系统的一部分或作为附加应用程序来提供。这些工具包括微软公司的 OLAP 工具、Oracle Express 和 Informix Metacube。有些工具被整合到大型“商业智能”产品中, 例如 IBM Cognos。很多公司还提供面向特定应用(比如客户关系管理)的分析工具, 例如 Oracle Siebel CRM。

文献注解

对于 SQL 标准和有关 SQL 的书籍请参考第 3 章的文献注解。

java.sun.com/docs/books/tutorial 是一个极好的了解更多最新 Java 和 JDBC 技术资源的站点, 有关 Java (包括 JDBC) 的书籍也可以在该 URL 中找到。ODBC 的 API 在 Microsoft [1997] 和 Sanders [1998] 中描述过。Melton 和 Eisenberg [2000] 提供了 SQLJ、JDBC 和相关技术的入门向导。更多关于 ODBC、ADO 和 ADO.NET 的信息能在 msdn.microsoft.com/data 上找到。

对于 SQL 中函数和过程的章节, 许多数据库产品支持标准说明以外的一些特性, 而有些则不支持某些标准内的特性。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。

最初有关断言和触发器的 SQL 提议在 Astrahan 等 [1976]、Chamberlin 等 [1976] 及 Chamberlin 等 [1981] 中作了讨论。Melton 和 Simon [2001]、Melton [2002], 以及 Eisenberg 和 Melton [1999] 提供了涵盖 SQL:1999 的教科书, 该版本的 SQL 标准中首次包含了触发器。

递归查询处理最早是在一种叫做 Datalog 的查询语言中详细研究的, 该语言基于数学逻辑, 沿袭了逻辑程序设计语言 Prolog 的语法。Ramakrishnan 和 Ullman [1995] 对该领域的研究结果做了综述, 其中涵盖了从递归定义的视图中查询元组的子集优化技术。

Gray 等 [1995] 和 Gray 等 [1997] 描述了数据立方体操作符。用于计算数据立方体的高效算法在 Agarwal 等 [1996]、Harinarayan 等 [1996]、Ross 和 Srivastava [1997] 中有描述。SQL:1999 中对扩展聚集支持的描述可在数据库系统 (比如 Oracle 和 IBM DB2) 的产品手册中找到。

目前有大量关于如何高效执行“top- k ”查询 (即只返回排名为前 k 的结果) 的研究。Ilyas 等 [2008] 对这方面工作做了综述。