



第五讲高级SQL

(第五章)

关继宏教授

电子邮件: jhguan@tongji.edu.cn

计算机科学与技术系

同济大学

课程大纲

- **第0部分:概述**

- Ch1:介绍

- **第1部分:关系数据库**

- Ch2:关系模型(数据模型, 关系代数)
- Ch3&4: SQL(结构化查询语言)
- **Ch5:高级SQL**

- **第二部分数据库设计**

- Ch6:基于E-R模型的数据库设计
- Ch7:关系型数据库设计

- **第三部分:应用程序设计与开发**

- Ch8:复杂数据类型
- Ch9:应用开发

- **Part 4大数据分析**

- Ch10:大数据
- Ch11:数据分析

- **第5部分:数据存储和索引**

- Ch12:物理存储系统
- Ch13:数据存储结构
- Ch14:索引

- **第6部分:查询处理与优化**

- Ch15:查询处理
- Ch16:查询优化

- **第7部分事务管理**

- Ch17:交易
- Ch18:并发控制
- Ch19:恢复系统

- **第8部分:并行和分布式数据库**

- Ch20:数据库系统架构
- Ch21-23:并行和分布式存储, 查询处理和事务处理

- **第9部分**

- **DB平台:OceanBase、MongoDB、Neo4J**

从编程语言访问DB

- 函数和过程
- 触发器
- SQL *中的递归
- 高级SQL功能*

从编程语言访问DB

- 用于程序与数据库服务器交互的API(应用程序接口)
- 应用程序调用
 - 连接数据库服务器
 - 向数据库服务器发送SQL命令
 - 逐个获取结果元组到程序变量中
- 各种工具:
 - 动态SQL
 - ODBC(开放数据库连接)与C、c++、c#和Visual Basic一起工作。其他API, 如ADO。 . NET位于ODBC之上
 - JDBC (Java数据库连接)与Java一起工作
 - 嵌入式SQL

JDBC (Java数据库连接)

- **JDBC**

- 用于与支持SQL的数据库系统通信的Java API
- 支持查询和更新数据以及检索查询结果的各种功能
- 支持元数据检索，例如查询数据库中存在的关系以及关系属性的名称和类型
- Java程序必须导入`java.sql.*`，它包含JDBC提供的功能的接口定义

- **与数据库通信的模型：**

- 打开连接
- 创建一个“Statement”对象
- 使用Statement对象执行查询，发送查询并获取结果
- 异常机制来处理错误

JDBC代码

```
public static void JDBCexample(字符串dbid, 字符串userid, 字符串passwd){
    尝试{
        类.forName (" oracle.jdbc.driver.OracleDriver ");
        Connection conn = DriverManager.getConnection (" jdbc:oracle:thin:@aura.bell-labs.com:2000:bankdb",
userid, passwd);
        语句stmt = conn.createStatement ();
        ...Do Actual Work ....
        支撑.close ();
        conn.close ();
    }
    catch (SQLException) {
        System.out.println (" SQLException: " + sqle);
    }
}
```

在连接conn上创建语句句柄(stmt)


JDBC代码(续)

- 更新到数据库

```
尝试{stmt.executeUpdate ("insert into account values ('A-9732', 'Perryridge ',  
1200)");  
  
}  
catch (SQLException) {  
    System.out.println("无法插入元组。" + sqle);  
}
```

- 执行查询，获取和打印结果

```
ResultSet rset = stmt.executeQuery (" select branch_name, avg(balance) from account group by  
branch_name ");  
  
While (rset.next ()) {  
    System.out.println (  
        资源集.getString (" branch_name ") + " " + rset.getFloat (2));  
}
```



JDBC代码详细信息

- **获取结果字段:**

资源集。如果branchname是select结果的第一个参数, `getString (" branchname ")`和
`rs.getString(1)`是等价的。

- **处理空值**

`If (rset.)wasNull ()`

`Systems.out.println("Got null value");`

- **开放数据库连接(ODBC)标准**
 - 应用程序与数据库服务器通信的标准
 - 的应用程序接口(API)
 - **打开与数据库的连接**
 - **发送查询和更新**
 - **获取返回结果**
- GUI、统计分析和电子表格等应用程序可以使用ODBC

ODBC(续)

- 支持ODBC的每个数据库系统都提供了一个必须与客户机程序链接的“驱动程序”库
- 当客户端程序进行ODBC API调用时，库中的代码与服务器通信以执行请求的操作并获取结果
- ODBC程序首先分配一个SQL环境，然后分配一个数据库连接句柄

ODBC(续)

- 使用SQLConnect()打开数据库连接。SQLConnect参数:
 - 连接句柄
 - 要连接的服务器
 - 用户标识符
 - 密码
- 还必须指定参数的类型:
 - 常量(SQL_NTS)表示前一个参数是一个以空结束的字符串

ODBC代码

```
int ODBCexample (){
    RETCODE错误;
    HENV env;/*环境*/
    HDBC康涅狄格州;/*数据库连接*/
    SQLAllocEnv (env);
    SQLAllocConnect (env, &conn);
    SQLConnect (conn, " db.yale.edu", SQL_NTS, " avi ", SQL_NTS, " avipasswd ", SQL_NTS);
    {...做实际工作...}
    柄(康涅狄格州);
    SQLFreeConnect(康涅狄格州);
    SQLFreeEnv (env);
}
```

ODBC代码(续)

- 程序主体

```
Char branchname[80];
```

```
浮动的平衡;
```

```
int lenOut1, lenOut2;
```

```
HSTMT;
```

```
SQLAllocStmt(conn, &stmt);
```

```
Char * sqlquery = " select branch_name, sum (balance) from account group by  
branch_name";
```

```
error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
```

```
if (error == SQL_SUCCESS) {SQLBindCol(stmt, 1, SQL_C_CHAR, branchname, 80,  
&lenOut1);
```

```
SQLBindCol(stmt, 2, SQL_C_FLOAT, &balance, &lenOut2);
```

```
while (SQLFetch(stmt) >= SQL_SUCCESS) {printf (" %s %g\n", 分支名, 余额);}}
```

```
SQLFreeStmt(stmt, SQL_DROP);
```

ODBC代码(续)

- 程序通过使用SQLExecDirect向数据库发送SQL命令
- 使用SQLFetch()获取结果元组。
- SQLBindCol()将C语言变量绑定到查询结果的属性
 - 当获取元组时，其属性值会自动存储在相应的C变量中

ODBC代码(续)

- SQLBindCol()的参数
 - ODBC stmt变量, 属性在查询结果中的位置
 - 从SQL到C的类型转换
 - 变量的地址
 - 对于字符数组等变长类型
 - 变量的最大长度
 - 获取元组时存储实际长度的位置
 - **注意:length字段返回的负值表示空值**
- 好的编程需要检查每个函数调用的结果是否有错误;为简洁起见, 我们省略了大部分检查

更多ODBC特性

- **事先准备好的声明中**
 - SQL语句准备:在数据库中编译
 - 可以有占位符():例如插入到帐户值(?, ?, ?)
 - 用占位符的实际值重复执行
- 默认情况下, 每个SQL语句都被视为自动提交的单独事务
 - 能在连接上关闭自动提交吗
 - `SQLSetConnectOption (conn, SQL_AUTOCOMMIT, 0)`
 - 事务必须由。显式地提交或回滚
 - `SQLTransact (conn, SQL_COMMIT)`或
 - `SQLTransact (conn, SQL_ROLLBACK)`

嵌入式SQL

- SQL标准在各种编程语言(如C和Java)中定义了SQL的嵌入
- 嵌入SQL查询的语言称为宿主语言, 宿主语言中允许的SQL结构包含嵌入式SQL
- EXEC SQL语句用于识别向预处理器发出的嵌入式SQL请求
EXEC SQL <嵌入式SQL语句> END_EXEC

注意:这因语言而异(例如, Java嵌入使用# SQL {...}),)

嵌入式SQL与JDBC或ODBC

- 嵌入式SQL程序在编译之前必须经过特殊的预处理器处理。预处理器用允许运行时执行数据库访问的主机语言声明和过程调用替换嵌入式SQL请求。
- 然后，生成的程序由主机语言编译器编译。
- 这是嵌入式SQL与JDBC或ODBC之间的主要区别。
 - 在JDBC中，SQL语句是在运行时解释的(即使它们是首先使用预处理语句特性准备的)。
 - 当使用嵌入式SQL时，一些与SQL相关的错误(包括数据类型错误)可能会在编译时被捕获。

示例查询

- 查找某个帐户中金额大于变量美元的客户的姓名和城市
- 在SQL中指定查询并为其声明游标

执行SQL

为select depositor 声明游标。 Customer_name, customer_city from depositor, customer, account where depositor。 Customer_name = customer。 Customer_name 和存款人account_number = account。 Account_number 和account。 Balance >:amount

END_EXEC

嵌入式SQL(续)

- open语句导致对查询求值
执行SQL打开c END_EXEC
- fetch语句导致查询结果中一个元组的值被放在主机语言变量上。
执行SQL `fetch c into: cn, :cc END_EXEC`重复调用获取查询结果中的连续元组
- close语句导致数据库系统删除保存查询结果的临时关系
执行SQL关闭c END_EXEC
- **注:以上细节因语言而异。例如, Java嵌入定义了Java迭代器来遍历结果元组。**

通过游标进行更新

- 可以通过声明游标为更新来更新由游标获取的元组吗

声明 `c cursor for select * from account where branch_name = 'Perryridge' for update`

- 更新游标 `c` 当前位置的元组

更新账户集 `balance = balance + 100`，其中当前位置为 `c`

动态SQL

- 允许程序在运行时构造和提交SQL查询
- 在C程序中使用动态SQL的示例。
`Char * sqlprog = " update account
set balance = balance * 1.05 where account_number = ? "`执行SQL准备
`dynprogchar account [10] = " A-101 ";`执行SQL执行dynprog
- 动态SQL程序包含一个?, 它是执行SQL程序时提供的值的占位符

- 从编程语言访问DB

函数和过程

- 触发器
- SQL *中的递归
- 高级SQL功能*

函数和过程

- **SQL:1999支持函数和过程**
 - 函数/过程可以用SQL本身编写，也可以用外部编程语言编写
 - **过程和函数允许将“业务逻辑”存储在数据库中，并从SQL语句执行。**
 - 函数对于特殊的数据类型，如图像和几何对象，特别有用
 - 例如:检查多边形是否重叠的函数，或者比较图像的相似性的函数
 - 一些数据库系统支持表值函数，它可以返回一个关系作为结果
- SQL:1999还支持一组丰富的命令式结构，包括
 - 循环, if-then-else, 赋值
- 许多数据库都有与SQL:1999不同的专有的SQL过程扩展

SQL函数

- 定义一个函数，给定客户的名字，该函数返回该客户拥有的帐户数的计数。

创建函数 `account_count(customer_name varchar(20))` *返回整数*
`begin declare a_count integer; Select count (*) into a_count from depositor where depositor.
Customer_name = Customer_name return a_count;` *结束*

- 查找拥有多个帐户的每个客户的姓名和地址

选择 `customer_name, customer_street, customer_city` *from* `customer` *where*
`account_count(customer_name) > 1`

表函数

- SQL:2003增加了返回关系结果的函数
 - 示例:返回给定客户拥有的所有帐户
- 创建函数accounts_of (customer_name char(20))
返回表(account_number char(10), branch_name char(15) balance
numeric(12,2))
- (select * from depositor D where D.customer_name =
accounts_of(customer_name) and D.account_number = A.account_number))
- 用法:select * from table (accounts_of('史密斯'))

过程扩展和存储过程

- **SQL提供了一种模块语言**
 - 允许在SQL中定义过程，使用if-then-else语句，for和while循环等。
- **存储过程**
 - 可以在数据库中存储过程吗
 - 然后用call语句执行它们
 - 允许外部应用程序在不知道内部细节的情况下操作数据库

程序结构

- 复合语句:begin...end
 - begin和end之间可能包含多个SQL语句
 - 局部变量可以在复合语句中声明
- **While和repeat语句:**

声明 n 整数默认0;

而 $n < 10$ 做

 设 $n = n + 1$

结束while重复

 设 $n = n - 1$

直到 $n = 0$

最后重复一遍

程序构造(续)

- **For循环**

- 允许对查询的所有结果进行迭代
- 例如，找出佩里里奇分行所有余额的总和

声明n个整数默认为0;

For r as select balance

从branch_name = ' 佩里里奇 '的帐户查询

设置n = n + r.balance end for

过程构造(Cont.)

- 条件语句(if-then-else)

- 例如:查找三类账户的余额之和(余额<1000, ≥1000和<5000, ≥5000)

如果 $r.balance < 1000$ 则设 $l = l + r.balance$ elseif $r.balance < 5000$ 则设 $m = m + r.balance$ else设 $h = h + r.balance$ end If

过程构造(Cont.)

- 发出异常条件的信号，并声明异常处理程序

声明out_of_stock条件声明out_of_stock begin的退出处理程序信号缺货结束

- 这里的处理程序是exit——导致封闭begin...结束被退出
- 异常时可能的其他操作

SQL过程

- account_count函数可以写成procedure:

创建过程account_count_proc (in customer_name varchar(20), out a_count integer) begin

Select count() into a_count from depositor where depositor. Customer_name =
account_count_proc. customer_name*

结束

- 可以使用调用语句从SQL过程或从嵌入式SQL调用过程。
声明a_count integer;call account_count_proc('史密斯', a_count);
- 过程和函数也可以从动态SQL中调用。
- SQL:1999允许多个同名函数/过程(称为名称重载), 只要参数的数量不同, 或者至少参数的类型不同。

外部语言函数/过程

- SQL:1999允许使用用其他语言(如C或c++)编写的函数和过程
- 声明外部语言过程和函数

*创建过程*account_count_proc (in customer_name varchar (20), out count integer)*语言C外部名称* ' / usr / avi /bin/ account_count_proc '*创建函数*
account_count (customer_name varchar(20))*返回整数语言C外部名称* ' / usr / avi /bin/ account_count '

外部语言例程(Cont.)

- **外部语言函数/过程的好处:**

- 很多操作更有效率, 表达能力更强

- **缺点**

- 实现功能的代码可能需要加载到数据库系统中, 并在数据库系统的地址空间中执行
 - **有意外破坏数据库结构的危险**
 - **安全风险, 允许用户访问未经授权的数据**
- 当效率比安全更重要时, 在数据库系统的空间中直接执行

外部语言例程的安全性

- 处理安全问题
 - 使用沙盒技术
 - 即使用像Java这样的安全语言，它不能被用来访问/破坏数据库代码的其他部分
 - 或者，在单独的进程中运行外部语言函数/过程，不能访问数据库进程的内存
 - 参数和结果通过进程间通信进行通信
- 两者都有性能开销
- 许多数据库系统既支持上述两种方法，也支持在数据库系统地址空间中直接执行

大纲

- 从编程语言访问DB
- 函数和过程

☞ 触发器

- SQL *中的递归
- 高级SQL功能*

触发器

- 触发器是由系统自动执行的语句，作为修改数据库的副作用
- 要设计一个触发器机制，我们应该：
 - 指定要执行触发器的条件
 - 指定触发器执行时要采取的操作
- 上述触发器模型被称为触发器的事件-条件-动作(ECA)模型

触发的例子

- 假设银行不允许账户余额为负，而是通过(动作)处理透支。
 - 将账户余额设置为零
 - 在透支金额上创建贷款
 - 给这笔贷款一个与透支账户的账号相同的贷款号
- 执行触发器的条件是对导致负余额值的帐户关系进行更新(事件)

触发器的例子在SQL:1999

创建触发器`overdraft_trigger`更新后的帐户引用新行为`nrow`

对于每一行，当`nrow. 余额 < 0`开始原子插入到借款人(`select customer_name, account_number from depositor where nrow. Account_number = 存款人. account_number`);插入到贷款值(`nrow. Account_number, nrow. Branch_name, - nrow. 平衡`);更新账户`set balance = 0 where account. Account_number = nrow. account_number`结束

在SQL中触发事件和操作

- 触发事件可以是插入、删除或更新
- 更新时的触发器可以限制为特定的属性
 - 例如，在更新账户余额之后
- 更新前后的属性值可以被引用
 - 将旧行引用为:用于删除和更新
 - 引用新行作为:用于插入和更新

在SQL中触发事件和操作

- 触发器可以在事件发生之前被激活，这可以作为额外的约束

创建触发器setnull_trigger在update on r上引用新行为nrow时的每一行之前。

Phone_number = ' ' set nrow。 Phone_number = null

语句级触发器

- 可以对受事务影响的所有行执行单个操作，而不是对每个受影响的行执行单独的操作
 - 对每条语句使用，而不是对每一行使用
 - 使用引用旧表或引用新表来引用包含受影响行的临时表(称为过渡表)
 - 在处理更新大量行的SQL语句时是否更有效

外部世界操作

- 我们有时需要在数据库更新时触发外部世界动作
 - 例如，重新订购仓库中数量变少的物品，或者打开警报灯，
- 触发器不能用来直接执行外部世界的动作， BUT
 - 触发器可以用来在一个单独的表中记录将要执行的动作
 - 有一个外部进程，反复扫描表，执行外部世界动作，从表中删除动作

外部世界动作

- 例如，假设一个仓库有以下表格
 - 库存(物品, 等级):每件物品在仓库中的数量
 - `minlevel (item, level)`:期望的最低水平是多少
 - `reorder(item, amount)`:我们需要再订购多少数量
 - 订单(项目, 数量):要下的订单

外部世界行动(续)

在库存数量更新后创建触发器`reorder_trigger`

引用旧行为`orow`，新行为`nrow`

对于每一行

 当`nrow.Level <= (select Level
 从minlevel
 minlevel的地方。Item = orow。项)`
 和`orow.Level > (select Level
 从minlevel
 minlevel的地方。Item = orow。项)`

开始

 插入订单
 (选择项目，金额
 从重新排序
 重新排序。`Item = orow。项`)

结束

什么时候不要使用触发器

- 触发器以前被用于诸如
 - 维护汇总数据(例如, 每个部门的总工资)
 - 通过记录特殊关系的变化来复制数据库, 并有一个单独的过程将这些变化应用到副本上
- 现在有更好的方法来实现这些:
 - 今天的数据库提供内置的物化视图设施来维护汇总数据
 - 数据库提供内置的复制支持
- 在许多情况下, 可以使用封装设施来代替触发器
 - 定义更新字段的方法
 - 执行动作作为更新方法的一部分, 而不是通过触发器

大纲

- 从编程语言访问DB
- 函数和过程
- 触发器

SQL递归*

- 高级SQL功能*

SQL中的递归

- SQL:1999允许递归视图定义
 - 例如, 用递归empl (employee_name, manager_name)作为(select employee_name, manager_name from manager /*a base query */ union select manager)查找所有员工-经理对, 其中员工直接或间接向经理报告(即经理的经理, 经理的经理的经理等)。
Employee_name, empl.;Manager_name from manager, empl /*递归查询*/ where manager。 Manager_name = empl。 Employee_name) select * from empl

注意:这个示例视图empl被称为经理关系的传递闭包()

递归的威力

- 递归视图使得编写查询成为可能，例如传递闭包查询，这些查询如果没有递归或迭代就无法编写。
 - 直觉:没有递归，一个非递归的非迭代程序只能执行固定数量的manager与自身的联接
 - 这只能给出固定数量的管理器层次
 - 给定一个程序，我们可以构建一个数据库，其中包含更多级别的管理人员，程序将无法在这些级别上运行

递归的力量

- 计算传递闭包
 - 下一张幻灯片展示了一个管理者关系
 - 迭代过程的每一步都从`empl`的递归定义构建一个扩展版本。
 - 最终的结果称为递归视图定义的不动点。
- 递归视图要求是单调的。也就是说，如果我们向`manager`添加元组，视图将包含它之前包含的所有元组，可能还会加上更多

定点计算的例子

<i>employee_name</i>	<i>manager_name</i>
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

<i>Iteration number</i>	<i>Tuples in empl</i>
0	
1	(Duarte), (Estovar)
2	(Duarte), (Estovar), (Barinsky), (Corbin)
3	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)
4	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)

大纲

- 从编程语言访问DB
- 函数和过程
- 触发器
- SQL *中的递归

高级SQL功能*

高级SQL特性

- 创建一个与现有表模式相同的表:

创建一个类似account的表temp_account

- SQL:2003允许在任何需要值的地方进行子查询, 只要子查询只返回一个值。这也适用于更新
- SQL2003允许from子句中的子查询使用横向结构访问from子句中其他关系的属性:

select count(*) from account a where A.customer_name =
C.customer_name作为this_customer(num_accounts)

高级SQL功能(续)

- 合并结构允许批量处理更新
 - 例如, 关系funds_received (account_number, amount)有一批存款要添加到帐户关系中适当的帐户中

当匹配时, 使用(*select * from funds_received*)作为F on
(*A.account_number = F.account_number*)合并到账户作为A, 然后更新set
balance = balance + F.amount

家庭作业

- **进一步的阅读**
 - 第五章

第五讲结束