

# 数据库技术-SQL高级

赵亚伟

**[zhaoyw@ucas.ac.cn](mailto:zhaoyw@ucas.ac.cn)**

中国科学院大学大数据分析技术实验室

**2017.11.26**

# 目录

- SQL数据类型与模式
- 完整性约束
- 嵌入式SQL
- 动态SQL (ODBC, JDBC)
- 安全性—授权
- 函数及过程\*\*
- 高级SQL特性 (SQL-03)\*\*
- 关于实验分析

# Built-in Data Types in SQL 内建类型

- **date:** Dates, containing a (4 digit) year, month and date
  - Example: **date** '2005-7-27'
- **time:** Time of day, in hours, minutes and seconds.
  - Example: **time** '09:00:30'      **time** '09:00:30.75'
- **timestamp:** date plus time of day
  - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval:** period of time
  - Example: **interval** '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

# Build-in Data Types in SQL

- Can extract values of individual fields from 抽取 date/time/timestamp
  - Example: **extract (year from r.starttime)**
  - 注意: SQL Server中用于提取的维度元素中的元组构成的集
- Can cast string types to date/time/timestamp 强制
  - Example: **cast <string-valued-expression> as date**
  - Example: **cast <string-valued-expression> as time**
  - 注意: SQL Server中是函数

# User-Defined Types

- 用户自定义类型（可视为简称）
- **create type** construct in SQL creates user-defined type  
类型定义  

```
create type Dollars as numeric (12,2) final
```
- **create domain** construct in SQL-92 creates user-defined  
domain types 域定义  

```
create domain person_name char(20) not null
```
- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.

域上可以指定约束

# Domain Constraints

- 域约束（型和值的约束）
- **Domain constraints** are the most elementary form of integrity constraint. They test values inserted in the database, and test queries to ensure that the comparisons make sense.
- New domains can be created from existing data types
  - Example: **create domain Dollars numeric(12, 2)**  
**create domain Pounds numeric(12,2)**
- We cannot assign or compare a value of type Dollars to a value of type Pounds.
  - However, we can convert type as below  
(**cast *r.A* as Pounds**)  
(Should also multiply by the dollar-to-pound conversion-rate)

# 类型与域之间的区别

- 尽管二者在应用上很相近，但还是有两个重要区别：
  - 在域上可以定义约束，也可以指定默认值，但在类型上却不可以
  - 域不是强类型的，只要相容，域类型可以相互赋值

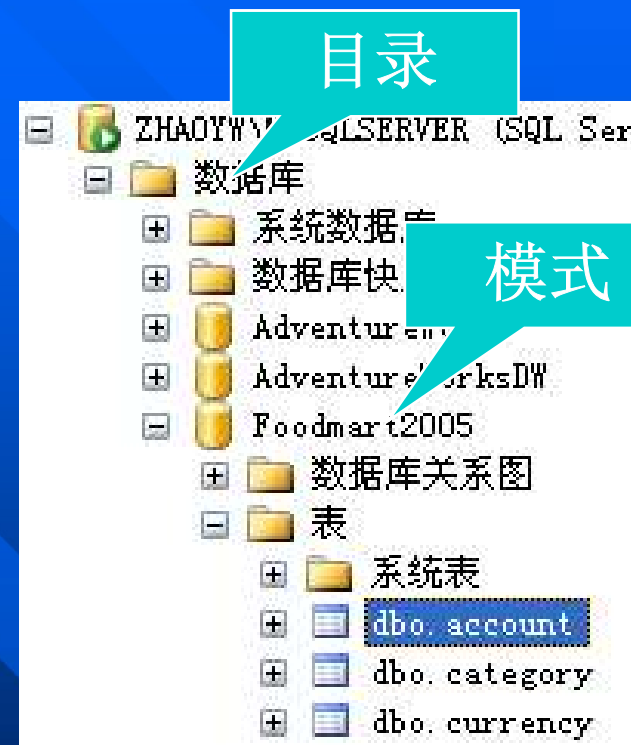
# Large-Object Types

- **大对象类型** Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
- **二进制大对象**
  - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system) 图像、电影
- **字符大对象**
  - **clob**: character large object -- object is a large collection of character data 文本, 小说
  - When a query returns a large object, a pointer is returned rather than the large object itself.



# 模式、目录与环境

- 现代数据库系统提供一个三层结构的关系命名机制：
  - 目录(**catalog**): 最顶层, 数据库模式
  - 模式: 每个目录都包含一个模式(**schema**), 数据库
  - 环境: 默认目录和模式是为每个连接建立的SQL环境(**SQL environment**), 环境还包括用户标识。



标示一个关系:

数据库. **Foodmart2005. account**

# 目录

- SQL数据类型与模式
- 完整性约束
- 嵌入式SQL
- 动态SQL (ODBC, JDBC)
- 安全性—授权
- 函数及过程\*\*
- 高级SQL特性 (SQL-03)\*\*
- 关于实验分析

# 完整性与安全性

- 完整性所要解决的问题是数据被授权用户修改时保证数据的一致性。
- 安全性所解决的问题是防止未经授权的访问和恶意破坏。
- 因此，完整性与安全性是两种对数据库的保护措施。前者也称完整性保护，后者也称安全性保护。
- 完整性保护还包括并发控制与恢复。
- 安全性保护访问包括控制技术、密码技术和审计技术

# 完整性约束



# 表完整约束(1)-唯一性

- 唯一性约束(**unique constraint**): 唯一性约束满足的充分必要条件是, 在表中不存在任何两行在清单所列的列上拥有相同的非空值, 若唯一性说明为**PRIMARY KEY**, 则进一步要求指定列不能取空值。
- SQL语法实现

*primary key | unique*

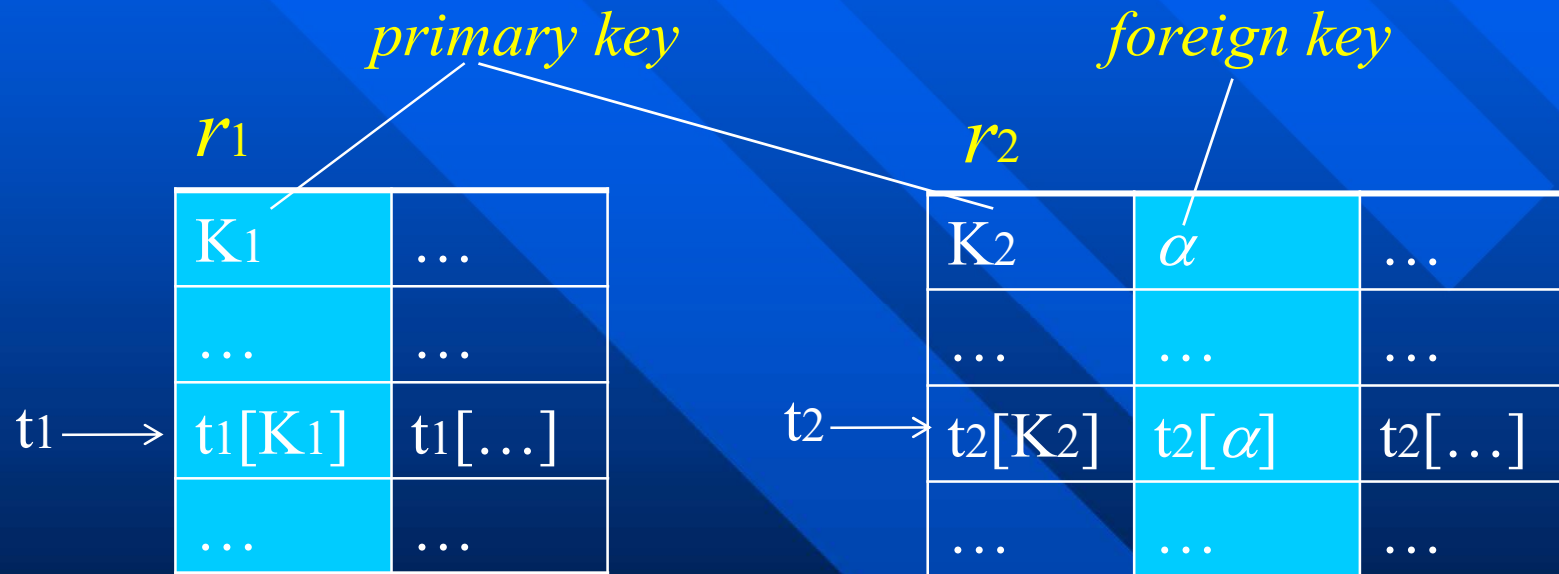
```
create table student  
(name char(15),  
student-id char(10),  
Degree-level char(15),
```

```
primary key | unique(student-id ))
```

*unique* 约束用于强制非主键列的唯一性。*primary key*约束列自动包含唯一性限制, *unique* 约束允许存在空值 (对非主码属性, 本例不可以为空)。

## 表完整约束(2)-参照完整性

- 被引用的列应在被引用表定义中说明为具有唯一性约束的列，即用 *primary key* 说明



$\alpha$ 是 $K_1$ 的 *foreign key*, 存在 $t_1[K_1] = t_2[\alpha]$ , 则 $\Pi_{\alpha}(r_2) \subseteq \Pi_{K_1}(r_1)$

# 参照完整性的SQL实现

```
create table customer
(customer-name
      char(20),
customer-street char(30),
customer-city  char(30),
primary key (customer-
name))
```

```
create table branch
(branch-name char(15),
branch-city  char(30),
assets integer,
primary key (branch-name))
```

```
create table account
(account-number char(10),
branch-name char(15),
balance integer,
primary key (account-number),
foreign key (branch-name)
references branch)
```

```
create table depositor
(customer-name char(20),
account-number char(10),
primary key (customer-name,
account-number),
foreign key (account-number)
references account,
foreign key (customer-name)
references customer)
```

Customer

***Customer\_name***  
Customer\_street  
Customer\_city

Depositor

***Customer\_name***  
***Account\_number***

Branch

***Branch\_name***  
Branch\_street  
Assets

Account

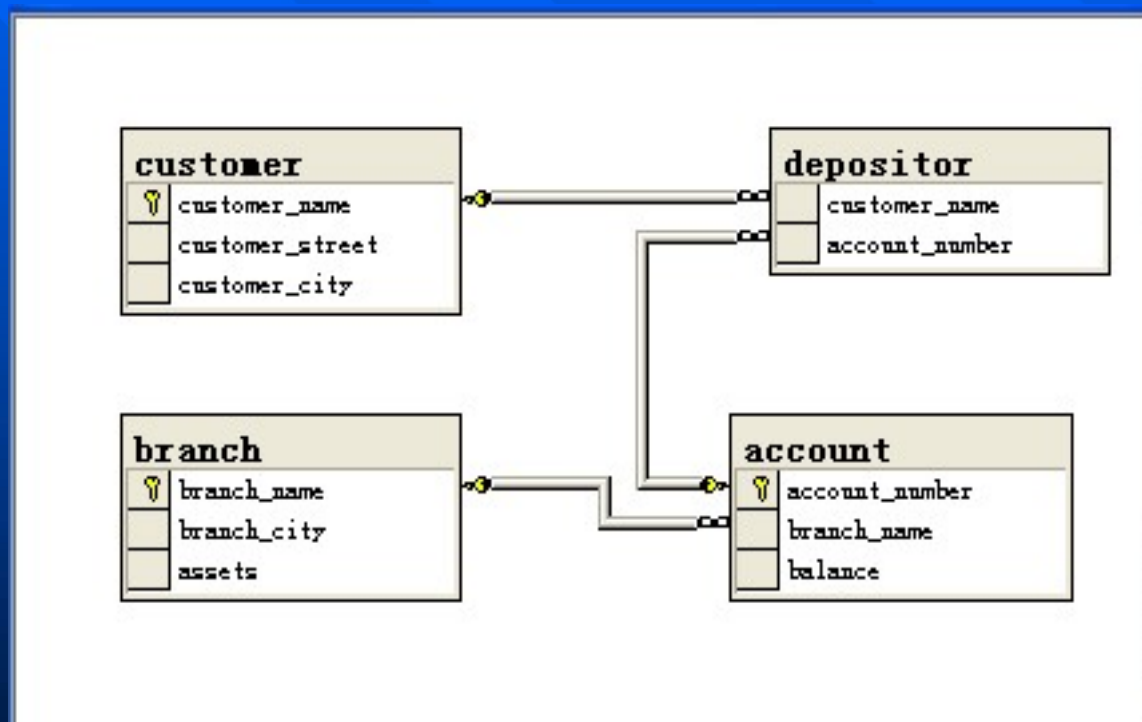
***Account\_number***  
***Branch\_name***  
Balance



一般的DBMS都能提供可视化的  
完整性等功能设置

# SQL Server 2000中的模式图

## ■ 带约束的模式图



# 数据库修改

- 已知参照完整性约束:

$$\Pi_{\alpha}(r_2) \subseteq \Pi_K(r_1)$$

- 插入: 向中 $r_2$ 插入 $t_2$ , 则必须保证 $r_1$ 中存在元组使得 $t_1[K]=t_2[\alpha]$ , 即

$$t_2[\alpha] \in \Pi_K(r_1)$$

- 删除: 从 $r_1$ 中删除元组 $t_1$ , 则系统必须计算 $r_2$ 中参照 $t_1$ 的元组集合

$$\sigma_{\alpha = t_1[K]}(r_2)$$

如果该集合非空, 或删除命令报错并撤消, 或参照 $t_1$ 的元组本身被删除。

## ■ 更新：考虑两种情况

1. 对关系 $r_2$ 中的元组 $t_2$ 更新，且需要修改外码 $\alpha$ ，则进行类似插入操作，若令 $t_2'$ 表示元组 $t_2$ 的新值，则系统必须保证

$$t_2'[\alpha] \in \Pi_K(r_1)$$

2. 对关系 $r_1$ 中元组 $t_1$ 更新，且需要修改主码 $K$ 的值，则进行类似删除操作

$$\sigma_{\alpha = t_1[K]}(r_2)$$

如果该集合非空，则更新被拒绝，或参照 $t_1$ 的元组本身被删除。

# 关于级联-删除及更新

- 当用户试图删除或更新外键所指向的键时，如何保证参照完整性？
- 级联是保证表间参照完整性的一种方法。
- **Create**或**Alter**一个表时，可以约定级联操作，在指定**foreign key**和**reference**时约定。SQL Server的语法如下：

**FOREIGN KEY**

**[ ( *column* [ ,...*n* ] ) ]**

**REFERENCES *ref\_table* [ ( *ref\_column* [ ,...*n* ] ) ]**

**[ ON DELETE { CASCADE | NO ACTION } ]**

**[ ON UPDATE { CASCADE | NO ACTION } ]**

- 另外,在授权中也存在级联授权和级联收回的问题

## ON DELETE { CASCADE | NO ACTION }

- 含义：指定当要创建的表中的行具有引用关系，并且从父表中删除该行所引用的行时，要对该行采取的操作。默认设置为 **NO ACTION**。
- 如果指定 **CASCADE**（小瀑布），则从父表中删除被引用行时，也将从引用表中删除引用行。如果指定 **NO ACTION**（无动作），SQL Server 将产生一个错误并回滚父表中的行删除操作。

## ON UPDATE { CASCADE | NO ACTION }

- 含义：指定当要创建的表中的行具有引用关系，并且在父表中更新该行所引用的行时，要对该行采取的操作。默认设置为 **NO ACTION**。
- 如果指定 **CASCADE**，则在父表中更新被引用行时，也将在引用表中更新引用行。如果指定 **NO ACTION**，**SQL Server** 将产生一个错误并回滚父表中的行更新操作。

# 多级级联

- 当r1被r2引用，r2被r3引用，r3被r4引用时，并均使用 **ON DELETE|UPDATE CASCADE** 定义相应的主键和外键。当对r1进行 **DELETE** 或 **UPDATE** 时，就涉及多级级联问题。单独的 **DELETE** 或 **UPDATE** 语句可启动一系列级联引用操作
- 级联引用操作必须构成不包含循环引用的树。在 **DELETE** 或 **UPDATE** 所产生的所有级联引用操作的列表中，每个表只能出现一次。级联引用操作树到任何给定表的路径必须只有一个。树的任何分支在遇到指定了 **NO ACTION** 或默认为 **NO ACTION** 的表时终止。



# 表完整约束(3)-表检验

- **CHECK**: 是通过限制可输入到一系列或多列中的可能值强制域完整性的约束。

- 语法:

```
create table student
(name char(15) not null,
student-id char(10),
Degree-level char(15),
primary key(student-id ),
check (degree-level in ('bachelors','masters','doctorate')))
```

要求属性 '**degree-level**'必须在下列枚举中  
(**'bachelors','masters','doctorate'**)

**Check**也可以将属性限制在某查询集合范围内, 需要注意的是,  
**check**的系统开销比较大

# 域完整约束(domain)

- 域约束是完整性约束的一种基本形式，用于约束属性的取值范围。
- 域约束包括物理上的和语义上的约束。
- SQL语法：
  - Create domain** *Dollars* numeric(12,2)
  - Create domain** *Pounds* numeric(12,2)
  - 物理上相同，但语义上不同，不能直接转换，语义上需要乘上相关系数。
- 一些商业产品不支持上述语法

# 断言 (Assertion)

- 断言是一种命名的约束，可能涉及某表中一些独立的行，也可能涉及一张表的所有内容，甚至涉及到与若干个表有关的一个数据库状态。上述域约束和参照完整性约束是断言的一种特殊形式
- SQL中断言语法：  
Create assertion <断言名称> check <谓词>  
只要<谓词>部分为“假”，不满足条件，就被拒绝执行，与域完整性、参照完整性相似

- 例子：每个支行的贷款金额总和必须少于该支行账户余额的总和

create assertion *sum-constraint* check

(not exists (select \* from *branch*

where ( select sum(*amount*) from *loan*

where *loan.branch-name* =

*branch.branch-name*)

>= (select sum(*balance*) from *account*

where *account.branch-name* =

*branch.branch-name*)))

not exists(null)=true

若sum(account)>=sum(balance),  
贷款>=余额, 则为“not null”, 则  
not exists(not null)=false, 断言被  
破坏, 拒绝执行

- 断言创建后，系统会检查其有效性，如果有效，则系统以后只有不破坏断言的数据库修改才会被允许。
- 一些商品化产品不完全支持断言

# 触发器

- 触发器是一种特殊类型的**存储过程**，当数据被修改时，触发器会生效。主要用于**强制复杂的业务规则或要求**。应用程序设计中很有用，这里只探讨它对关系的完整性约束功能。
- 设置触发器的两个要求：
  - 指明触发器执行的条件，事件—条件
  - 指明触发器执行时采取的动作
  - 模型：**事件—条件—动作**
- 语法：
  - **create trigger** *trigger\_name* **after update on** *account*
  - **create trigger** *trigger\_name* **on** *account* **after insert, update** as 触发器执行的动作 (SQL Server)

# 触发器

- **SQL-99**前触发器不是标准，但商品化的数据库系统广泛使用的触发器，各自定义，相互不兼容。
- **SQL-99**中的触发器语法与**IBM DB2**及**Oracle**系统的语法类似。
- 触发器与断言类似，消耗系统资源大，实际中可采用存储过程取代触发器。

# 触发器—例子

- Jones是该银行的客户，具有存款账户，Jones 透支操作将触发一些列的动作。首先，系统读取账户余额（负数），建立一笔新贷款将余额（转正，透支额）存入，将账户余额置为“0”。

```
create trigger overdraft_trigger after update on account  
referencing new row as nrow  
for each row  
when nrow.balance < 0  
begin atomic
```

```
  insert into borrower
```

```
    (select customer_name, account_number  
     where nrow.account_number = depositor.account_number);
```

复合语句 

```
  insert into loan values
```

```
    (n.row.account_number, nrow.branch_name, - nrow.balance);
```

```
  update account set balance = 0
```

```
    where account.account_number = nrow.account_number
```

```
end
```

在借款人插入一个新元组，该用户是存款人

在贷款中插入一个新元组，其贷款值为 -*nrow.balance* (*balance* 为负数)

将该用户的账户余额置为



# SQL Server 中调试 trigger

```
/*CREATE TRIGGER reminder
ON account
FOR INSERT, UPDATE
AS RAISERROR ('呵呵，触发器已执行', 16, 10)*/

insert into account(account_number, branch_name, balance)
values('100', '汉武帝', 1998)

/*drop trigger reminder*/

select *
from account
```

服务器: 消息 50000, 级别 16, 状态 10, 过程 trigger\_test  
呵呵，触发器已执行

(所影响的行数为 11 行)

(所影响的行数为 11 行)

	account_number	branch_name	balance
1	100	汉武帝	1998
2	A-101	Downtown	500
3	A-102	Perryridge	400
4	A-201	Brighton	900
5	A-215	Mianus	700
6	A-217	Brighton	700
7	A-222	Redwood	700
8	A-305	Round hill	349
9	b-100	zhaoyawei	17
10	b-102	汉武帝	18
11	b-103	汉武帝1	188



# 触发链

- 写触发器应特别小心，一个触发器的动作可能会引发另一个触发器，这就形成一个**触发链**。
- 相互触发会出现：**死链**，如A关系“插入触发器”一个动作引起A关系上另一个“插入触发器”新的插入动作，**自锁了**。
- 解决死链的办法：限制链长度，如不允许超过**16或32次**，达到最大限制就强制停止。
- 一般情况下，长触发器链被视为系统错误，尽量避免出现死链。

# 目录

- SQL数据类型与模式
- 完整性约束
- 嵌入式SQL
- 动态SQL (ODBC, JDBC)
- 安全性—授权
- 函数及过程\*\*
- 高级SQL特性 (SQL-03)\*\*
- 关于实验分析

# 嵌入式SQL

- 交互式SQL是在DBMS上执行，**DBA**
- 嵌入式SQL与应用程序融为一体，**Developer**
- 需要将SQL嵌入到其他高级语言中，**嵌入式SQL**
- 一般格式：  
    EXEC SQL      //在C语言(主语言)中嵌入SQL的格式  
        <embedded SQL statement>  
    END-EXEC
- 现在的集成开发环境已经大大方便了SQL的嵌入。
  - 如在Delphi和C++ Builder中使用Tquery组件可以方便嵌入SQL  
        Query1->SQL->Add( "Select loan\_number from loan");  
        Query1->ExecSQL( );
  - 如在VB中用OLEDB等组件来嵌入SQL
  - ...

# Example Query

From within a host language, find the names and cities of customers with more than the X dollars in some account. 找出存款多于X元的账户

**EXEC SQL**

**declare** *c* **cursor for**

**select** *customer-name, customer-city*

**from** *depositor, customer, account*

**where** *depositor.customer-name =*

*customer.customer-name*

**and** *depositor account-number =*

*account.account-number*

**and** *account.balance > :X*

连接  
条件

结果是集合，游标c  
指向其中当前元组

X是变量

**END-EXEC**

# Embedded SQL

The **open** statement causes the query to be evaluated  
EXEC SQL **open** *c* END-EXEC

把查询结果元组放进主语言用**fetch**，（可循环调用）

The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.

EXEC SQL **fetch** *c into* *:cn, :cc* END-EXEC

Repeated calls to **fetch** get successive tuples in the query result

.

# Embedded SQL

通讯区SQLCA，SQL与主语言共享，互通信息

A variable called SQLSTATE in the SQL Communication Area (SQLCA) gets set to '02000' to indicate no more data is available

结束，删除临时结果

The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

EXEC SQL **close** *c* END-EXEC

**Note: Java** 中 具体语句 有所不同

# Updates Through Cursors (通过游标更新)

光标 C 指向结果集中的当前元组

```
declare c cursor for
  select *
  from account
  where branch-name = 'Perryridge'

for update // 以上准备好查询结果集合,下面更新

To update tuple at the current location of cursor

update account
  set balance = balance + 100 //余额加100
  where current of c
```

# 目录

- SQL数据类型与模式
- 完整性约束
- 嵌入式SQL
- 动态SQL (ODBC, JDBC)
- 安全性—授权
- 函数及过程\*\*
- 高级SQL特性 (SQL-03)\*\*
- 关于实验分析



# 动态SQL

- 在实际应用中，查询需求变化经常发生，在DBMS中进行交互式查询十分方便，但不适合一般用户
- 嵌入式SQL便于应用，但查询在编译时就已经写死
- 动态SQL可以具有一定的灵活性，又具有嵌入式SQL的优点
- 例子：  

```
char *sqlprog="select loan_number from loan where amount>?";  
EXEC SQL  
  sqlprog;  
END-EXEC
```

# 例子：动态SQL

## 动态SQL实验

请输入SQL命令：（如： select \* from testmdb.dbf）

select \* from DemiClass.DBF

执行

	CLSCODE	CTY_BIG	CTY_SML
▶	0101	食品	糖果、饼干
	0102	食品	罐头、调味品
	0103	食品	蜜饯、膨化、小食品
	0104	食品	茶叶、保健品
	0105	食品	冲饮
	0106	食品	面制品
	0107	食品	干果
	0108	食品	宠物用品
	0109	食品	冷冻品
	0110	食品	巧克力类

# 数据共享问题

- 异构型数据库之间的数据共享问题提出
- SQL标准为应用程序的移植带来了希望，但各个DBMS定义了各自的SQL“方言”，这就导致不同的DBMS之上的应用软件之间难以实现数据通信和共享。

问题如何解决？

# ODBC的提出

- **ODBC(Open Database Connectivity)**:是由 **Microsoft** 公司于1991 年提出的一个用于访问数据库的统一**标准**。
- 应用程序和数据库系统之间的**中间件**。
- 通过使用相应应用平台上和所需数据库对应的驱动程序与应用程序的交互来实现对数据库的操作，避免了在应用程序中直接调用与数据库相关的操作，实现了数据的**独立性**。

# ODBC的基本思想

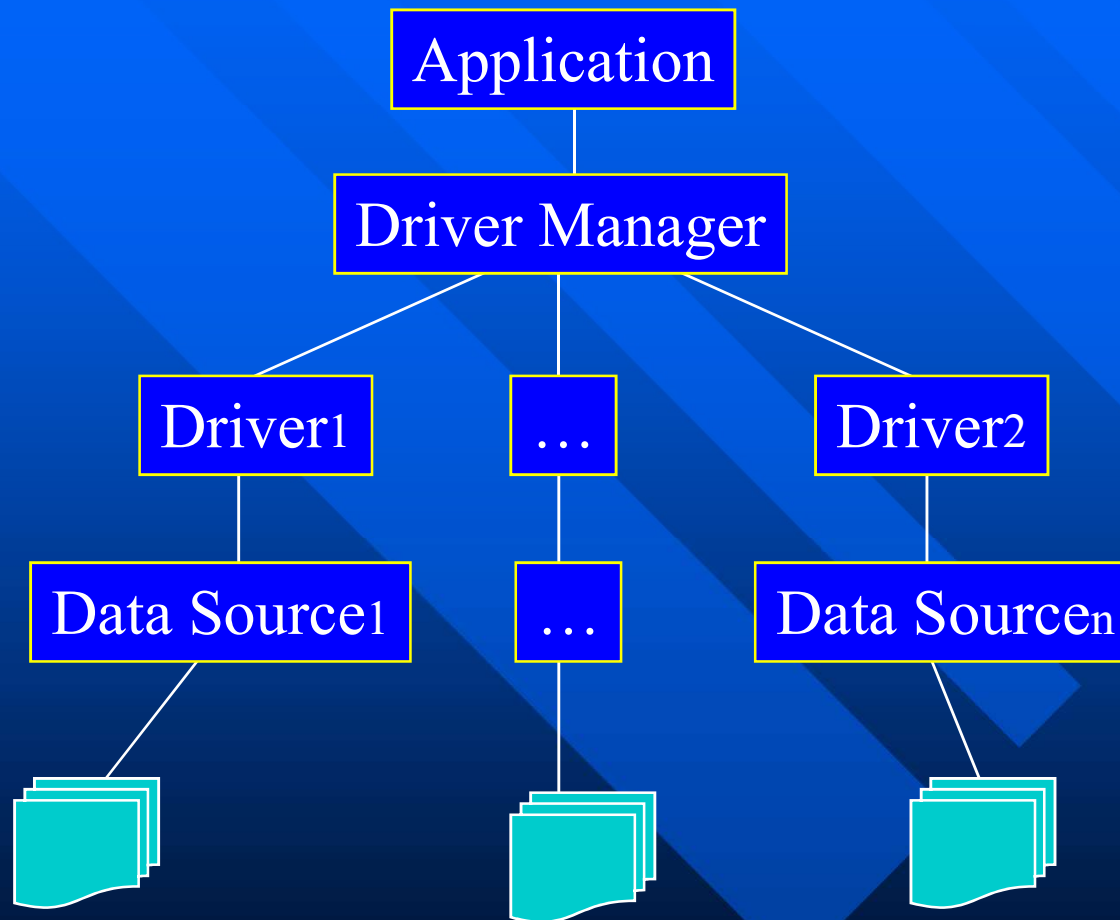
- ODBC定义了一个统一的数据接口：数据库驱动程序 (Database Drivers)，该驱动程序是一个动态链接库 (Dynamic Link Libraries, DLL)，就像在Windows下的打印机驱动程序一样，应用程序可以根据需要来选择一个数据源。
- ODBC提供了一个标准接口，使应用程序可在各种应用和数据源之间传递数据。

# ODBC的组成

## ■ ODBC包括四个组件：

- (1) 应用程序(Application): 负责调用ODBC函数来提交SQL语句，提取结果。
- (2) 驱动程序管理器(Driver Manager): 为应用程序加载驱动程序
- (3) 驱动程序(Driver): 处理ODBC函数调用，向数据源提交SQL请求，向应用程序返回结果，必要时驱动程序将SQL语法翻译成符合DBMS语法规定的格式。
- (4) 数据源(Data source): 由用户想要存取的数据。

# ODBC驱动程序组件





# ODBC的符合性级别

- 最小级SQL语法:

- (1) DDL: CREATE TABLE和DROP TABLE
- (2) DML: 简易SQL, INSERT, UPDATE和DELETE。
- (3) 表达式: 简易型(例如 $A > B + C$ )。
- (4) 数据类型: CHAR, VARCHAR, LONG VARCHAR

- 核心级SQL语法:

- (1) 最小级SQL语法和数据类型。
- (2) DDL: ALTER TABLE, CREATE INDEX, DROP INDEX, CREATE VIEW, DROP VIEW, GRANT和REVOKE
- (3) DML: 全部SELECT。
- (4) 表达式: 子查询及一组函数(如SUM和MIN)。
- (5) 数据类型: DECIMAL, NUMERIC等

## ■ 扩充SQL语法:

- (1) 最小级和核心级SQL语法和数据类型。
- (2) DML: 外连接、定位UPDATE、定位DELETE等
- (3) 表达式: 扩充了函数, 如(SUBSTRING、日期、时间等)。
- (4) 数据类型: **BIT, TINYINT, BIGINT, DATE, TIME, TIMESTAMP**等。
- (5) 批量SQL语句。
- (6) 过程调用。

# 应用ODBC的基本流程

- (1) 连接数据源。指定数据源名字，及所必需的信息
- (2) 处理一组SQL语句。
- (3) 以提交或撤销的方式结束事务
- (4) 完成对数据源的操作后结束与数据源的连接。

# 在C++中的ODBC

```
#include <SQL.H>
#include <WINDOWS.H>
#include <SQLEXT.H>

int ODBCexample()
{
    HENV env; //定义环境句柄
    HDBC conn; //定义连接句柄

    SQLAllocEnv(&env); //分配内存存储空间
    SQLAllocConnect(env,&conn); //为连接信息分配存储空间
    SQLConnect(conn,"COPERATION_DATA",SQL_NTS,"avi",
    SQL_NTS,"avipasswd", SQL_NTS); //建立连接
```

```

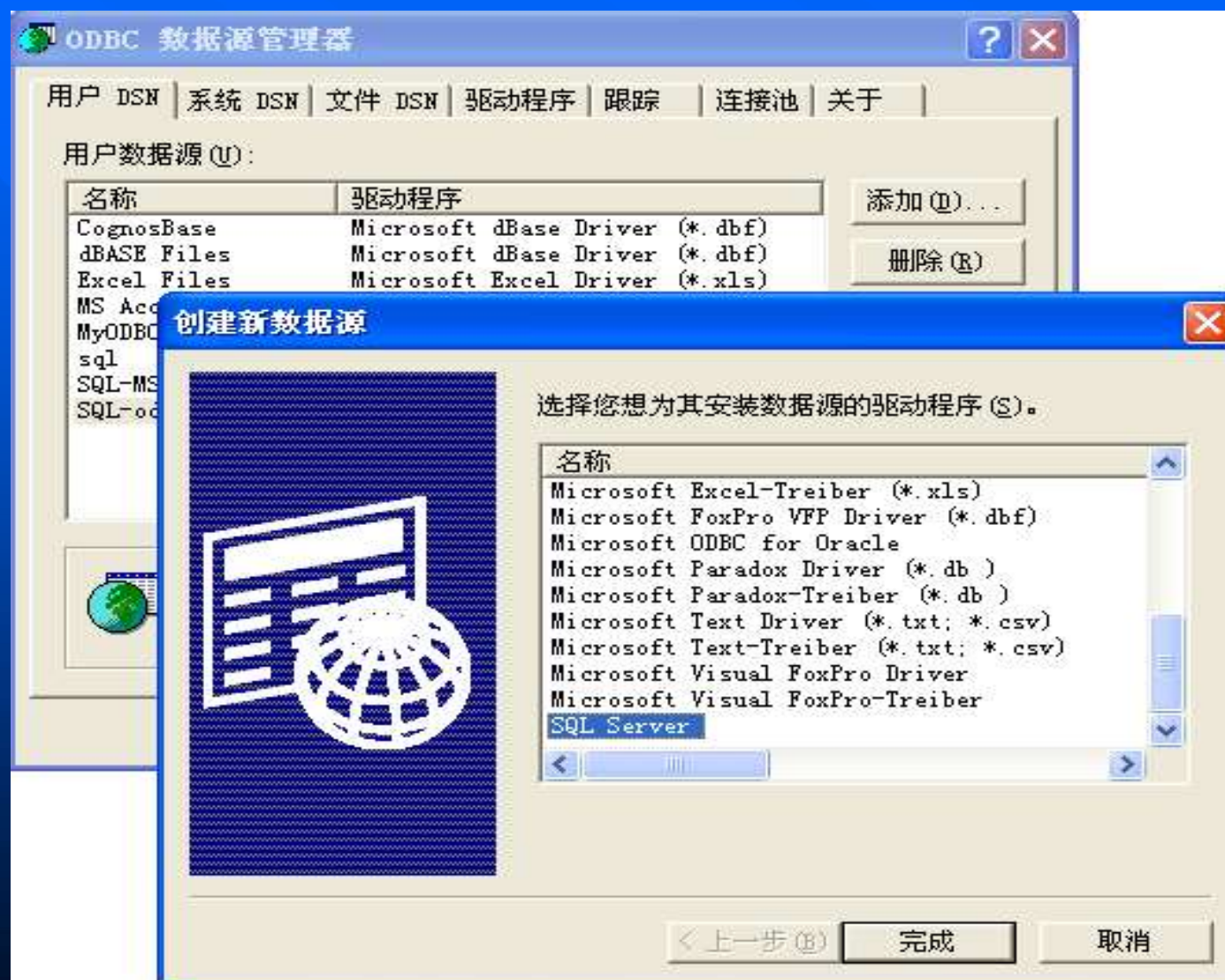
{ char branchname[80];
  float balance;
  int lenOut1, lenOut2;
  HSTMT stmt; //定义语句句柄
  SQLAllocStmt(conn, &stmt); //分配语句句柄
  char * sqlquery = "select branch_name, sum (balance)
                    from account
                    group by branch_name";
  error = SQLExecDirect(stmt, sqlquery, SQL_NTS); //执行(处理)SQL
  if (error == SQL_SUCCESS) {
    SQLBindCol(stmt, 1, SQL_C_CHAR, branchname, 80, &lenOut1);
    SQLBindCol(stmt, 2, SQL_C_FLOAT, &balance, 0, &lenOut2);
    //为结果分配缓冲区，并指定数据类型
    while (SQLFetch(stmt) >= SQL_SUCCESS) {
      printf(" %s %g\n", branchname, balance);
    } //结果输出
  }
  SQLFreeStmt(stmt, SQL_DROP); //释放语句句柄
  SQLDisconnect(conn); //断开连接
  SQLFreeConnect(conn); //释放连接句柄
  SQLFreeEnv(env); //释放环境句柄
}

```

# 在C++ Builder和Delphi中使用ODBC方法

- 连接到SQL Server
- 在ODBC中创建一个新的数据源
  1. 在ODBC中新建一个数据源，选择适当的驱动程序，如SQL Server
  2. 数据源命名（如SQL-ODBC），以及服务器选择
  3. 数据库选择

# 在ODBC中创建一个新的数据源



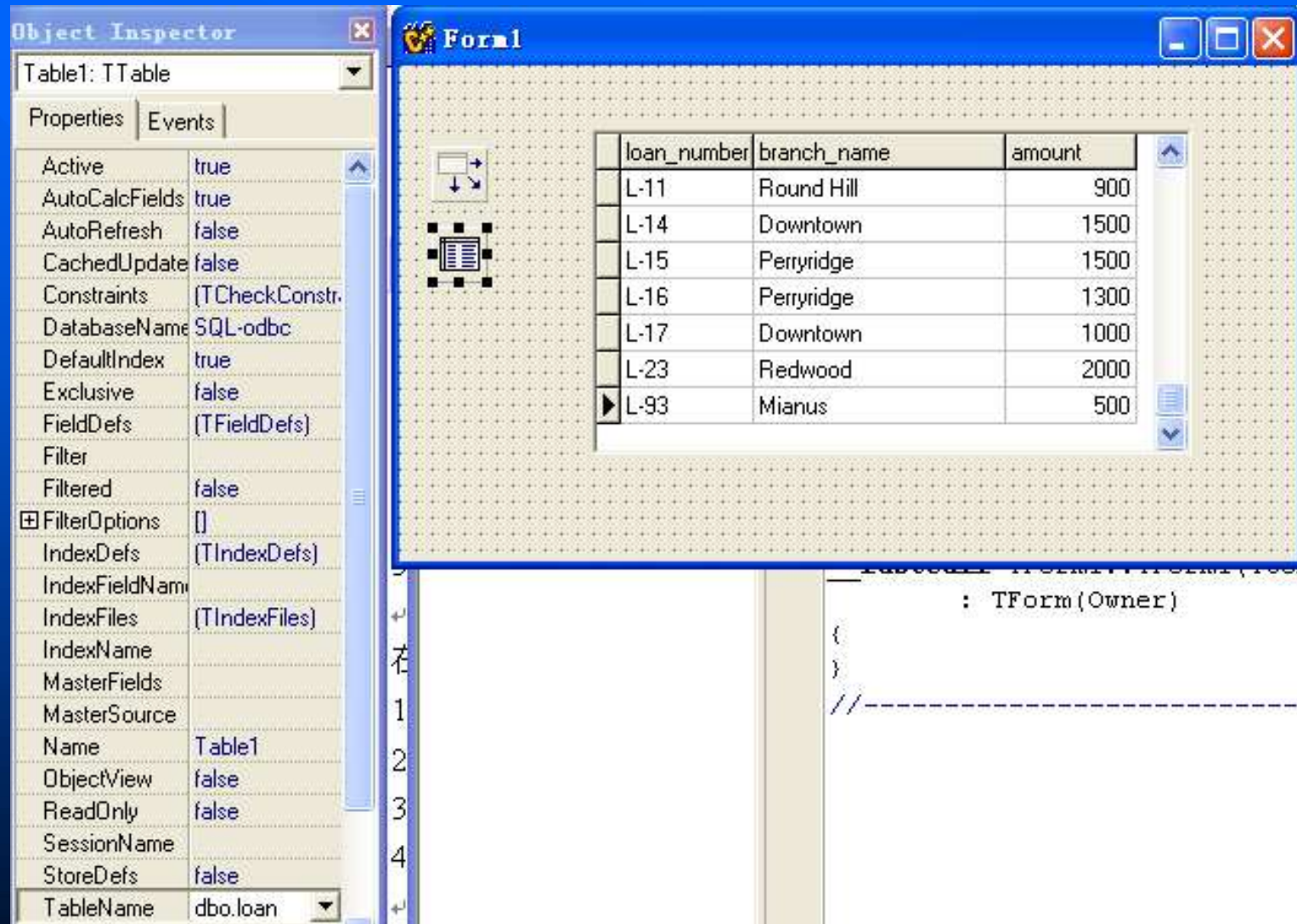


- 使用TTable调用ODBC数据源
  1. 在窗体中加入TTable组件
  2. 在DataBaseName属性中会看到SQL-ODBC数据源的名字
  3. 在TableName属性中选择相应的表
  4. 至此，通过ODBC连接到新数据源
- 使用TQuery调用ODBC数据源
  1. 在窗体中加入TQuery组件
  2. 在DataBaseName属性中会看到SQL-ODBC数据源的名字
  3. 在SQL属性中书写SQL语句如下：

```
use bank
select *
from loan
order by amount desc
```
  4. 至此，通过ODBC连接到新数据源



## 使用TTable调用ODBC数据源



## 使用TQuery调用ODBC数据源

The screenshot shows the Delphi IDE. On the left, the 'Object Inspector' for 'Query1: TQuery' is visible. The 'SQL' property is set to '(TStrings)'. On the right, the 'Form1' contains a table with the following data:

loan_number	branch_name	amount
L-23	Redwood	2000
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-11	Round Hill	900
L-93	Mianus	500

```
use bank
select *
from loan
order by amount desc
```

# ODBC解决了数据共享性的问题

# 目录

- SQL数据类型与模式
- 完整性约束
- 嵌入式SQL
- 动态SQL (ODBC, JDBC)
- 安全性—授权
- 函数及过程\*\*
- 高级SQL特性 (SQL-03)\*\*
- 关于实验分析

# 系统安全分级

- 20世纪70年代初以来，欧美等发达国家就开始重视计算机系统的安全问题。1983年美国率先发布了“可信计算机系统评价标准(TDSEC)”，1994年4月美国国家计算机安全中心(NCSC)颁布TDI，即“可信计算机系统评估标准在数据库管理系统中的解释”，它将TDSEC扩展到数据库领域。在TDSEC中，系统安全划分为七个不同的安全级别，即D、C1、C2、B1、B2、B3、A1，其中A1级别最高，D级别最低。
- TDI从安全策略、责任、保证和文档四个方面来描述每级安全性。一般认为，数据库及其它处理敏感商业信息的系统应达到C2级。处理保密的和要求更高敏感度的信息系统，安全级应达到B1级。

# 安全性违例及措施

- 恶意访问形式：
  - 未经授权的读取信息
  - 未经授权修改数据
  - 未经授权破坏数据
- 安全性措施：
  - 数据库系统层次：授权
  - 操作系统层次：授权
  - 网络层次：防火墙
  - 物理层次：设施保护
  - 人员层次：管理



# 授权

- 数据库访问授权:

- Read**: 允许读取数据, 但不允许修改数据
- Insert**: 允许插入新数据, 但不允许修改已经存在的数据
- Update**: 允许修改数据, 但不允许删除数据
- Delete**: 允许删除数据

- 数据库模式修改授权:

- Index**: 允许创建和删除索引
- Resource**: 允许创建新关系
- Alternation**: 允许添加或删除关系中的属性
- Drop**: 允许删除关系

# SQL实现安全性的方法-授权

## ■ 权限授予

**Grant** <权限列表> **on** <关系名或视图名> **to** <用户/角色列表>

可以同时多个权限一次授予多个用户或角色

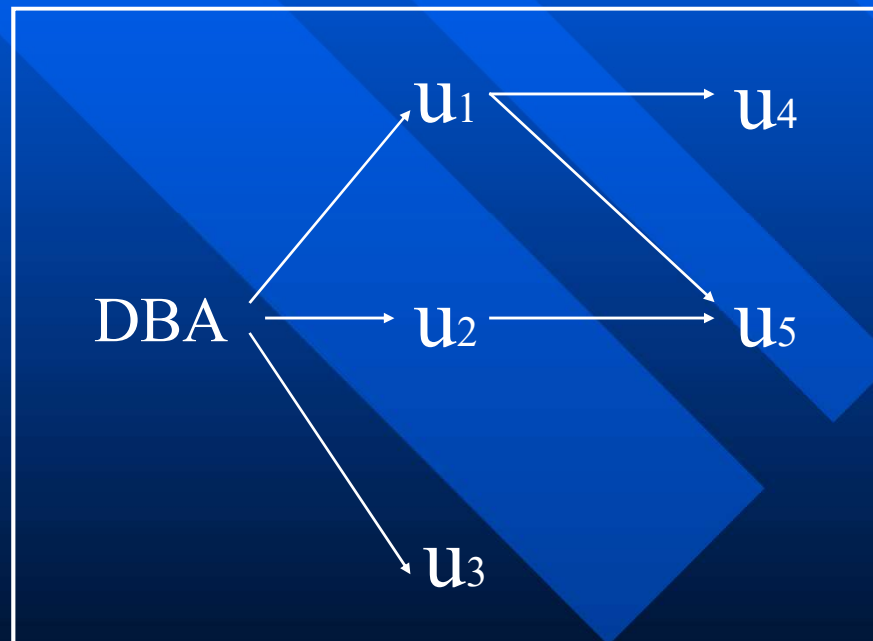
**Grant select on *account* to U1,U2,U3**

SQL标准包括**delete**、**insert**、**select**、**update**和**references**等权限



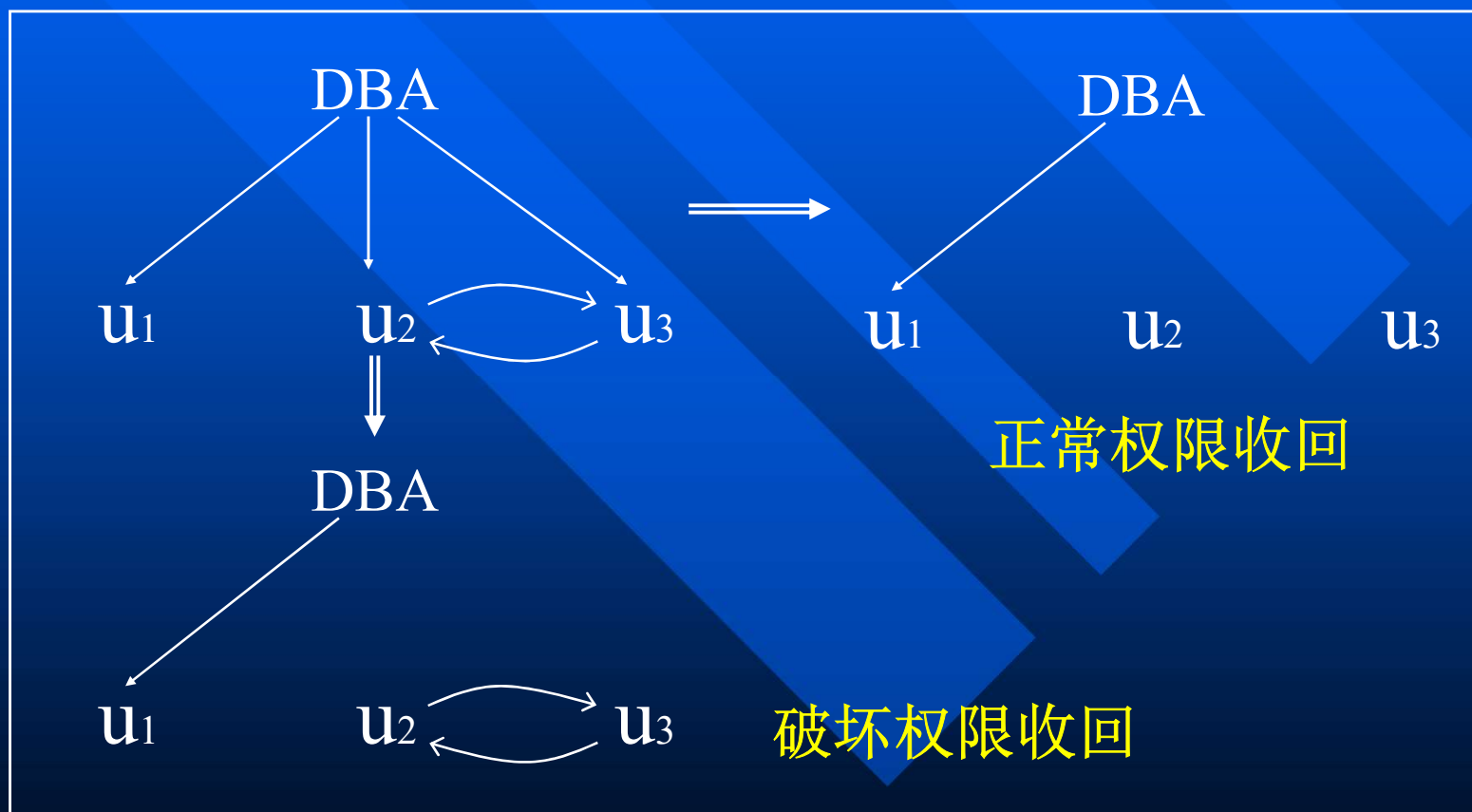
# 权限授予图

- 获得某种形式授权的用户可能将此授权授予其他用户，因此，授权可能在用户之间传递，这就可能形成权限授予图。假设DBA将update权限授予了用户u1,u2,u3，若收回u1的权限，则u5仍将具有该权限



# 破坏权限回收

## ■ 相互授权可能破坏权限回收



# 权限授权及收回

- SQL允许用户/角色将权限授予其他用户/角色, 使用**with grant option**子句

**Grant select on *branch* to U1 with grant option**

- 在默认状态SQL不允许用户权限授权
- 权限收回

**revoke <权限列表> on <关系名或视图名> from <用户/角色列表> [restrict][cascade]**

**Cascade**表示权限可级联收回, **restrict**表示不可以级联收回。**Cascade**是默认状态可不予指定

**revoke create table from liuce**

# 元组授权问题

- 问题描述：学生成绩或员工工资的查询应该只能看到用户个人信息而不能看到他人的
- 授权还未达到元组级别
- SQL的授权机制存在缺陷，解决这一问题一般要通过应用程序实现

# 角色(Role)

- 问题：考虑一个有很多出纳的银行，每个出纳必须对同一组关系具有相同的权限，当有一个新的出纳时，应该单独授予这些权限，有没有更简洁方式？
- 一个更好的机制是指明所有出纳都具有相同的权限，只要标识出哪些用户是出纳。即一个新出纳出现，只要给他一个标识及标示为出纳。
- **角色**是一个具有相同权限的用户集合，任何授权给用户都可以授权给角色。
- **SQL Server**，创建角色后可将用户添加到该角色

# 角色(Role)

- SQL-99 supports roles

- `create role teller` //创建角色
  - `create role manager`

- 角色授权：同用户授权

- `grant select on branch to teller`
  - `grant update (balance) on account to teller`
  - `grant all privileges on account to manager`

- 角色可以分配给用户，也可以分配为其他角色

- `grant teller to manager`
  - `grant teller to alice, bob`

# 审计追踪

- 审计追踪是一种根据数据库修改(插入/删除/更新)日志进行回溯追踪的一种方法
- 一些错误操作和欺骗性行为在审计追踪中可以发现
- 当日志的数据量十分庞大的时候,一般的搜索方法效率会很低,问题如何解决?

# 数据库加密种类

- 数据库的数据加密分为三种形式：

- 表加密
- 元组加密
- 属性加密

表加密是将整个表进行加密，元组加密是指对某些记录进行加密，属性加密则是对某些敏感字段进行加密



# 加密技术简介

## ■ 对称加密

- 密钥只有一个，加密和解密用同一密钥，密钥管理和传送是一个十分重要的问题
- 对称加密最著名的是美国数据加密标准**DES**、**AES**(高级加密标准)和 欧洲数据加密标准**IDEA**

## ■ 非对称加密（公钥加密）

- 密钥有两个：公钥和私钥，私钥用于加密，公钥用于解密。所谓公钥是指公开的密钥，而私钥则是不公开的，只有拿到公钥和私钥才能对数据进行修改和加密。一般用于身份认证，数字签名等
- 著名的公钥加密算法是**RSA**

# 身份认证

## ■ 身份认证所采用的方法:

- 密码认证, 用户直接输入密码进行验证, 但在网络上容易被窃取
- 问答式认证, 系统提问, 用户用密码作为私钥加密相应字符串后发送给系统, 系统用公钥进行解密, 若通过则为合法用户

## ■ 数字签名

- 基本思想和问答式身份认证相同, 即用私钥对签名(明文)进行加密, 其他人可以用公钥对签名(密文)进行解密来实现签名的验证

# PKI

- **PKI (Public Key Infrastructure)** 公钥基础设施是提供公钥加密和数字签名服务的系统或平台，目的是为了管理密钥和证书
- **PKI**是目前信息安全技术的一个研究和应用上的热点，感兴趣的同学可以参考一下相关资料

# 目录

- SQL数据类型与模式
- 完整性约束
- 嵌入式SQL
- 动态SQL (ODBC, JDBC)
- 安全性—授权
- 函数及过程\*\*
- 高级SQL特性 (SQL-03)\*\*
- 关于实验分析

# 函数及过程

- SQL99开始提供函数、过程和方法，可以通过过程组件来定义，也可以通过外部的程序设计语言来定义。
- 商用的数据库系统支持自己的过程语言，如Oracle中的PL/SQL和Microsoft SQL Server中的Transact SQL，这些自定义的过程部分类似SQL99的过程部分，但在语法和语义上有区别


函数及过程的主要功能：  
提高程序的开发效率及可读性

# 函数定义

- 通过一个例子说明函数定义。给定一个顾客的名字，返回该客户拥有的账户数。

复合语句 {

```
create function account_count(customer_name varchar(20))  
  returns integer  
  begin  
    declare a_count integer; 声明一个变量  
    select count(*) into a_count  
    from depositor  
    where depositor.customer_name=customer.name  
    return a_count;  
  end
```



# 过程定义

- 将上述函数写为一个过程如下：

```
create procedure account_count_proc(in customer_name  
varchar(20), out a_count integer)
```

复合  
语句 {

```
begin  
    select count(*) into a_count  
    from depositor  
    where depositor.customer_name =  
    account_count_proc.customer_name  
end
```



# 例子—函数及过程调用

- 函数调用

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

- 过程调用


```
declare a_count integer  
call account_count_proc('Smith',a_count)
```

# 表函数

- SQL2003支持返回以表为结果的函数，这样的函数称为表函数
- 通常，表函数可以被视为带参数的视图
- 通过允许带参数把视图的概念更加一般化

# 例子：表函数定义

```
create function accounts_of(customer_name char(20))  
returns table( //定义表函数结构  
    account_number char(10),  
    branch_name char(15),  
    balance numeric(12,2))  
return table //表函数数据  
    (select account_number,branch_name,balance  
    from account  
    where exists(  
        select *  
        from depositor  
        where depositor.customer_name = accounts_of.  
            customer_name and depositor.account_number =  
            account.account_number))
```



参数customer\_name引用

# 例子：表函数应用

- 表函数可以作为一个表调用

```
select *  
from table (accounts_of ('Smith'))
```

# 过程化结构

- 从SQL99开始，SQL支持多种过程化结构，与通用程序设计语言相当的几乎所有的功能
- 如begin...end的复合语句；支持变量的声明；也支持循环语句 while...end while、repeat...end repeat、for...end for；同时也支持if...then...else、case的选择结构
- 上述内容不一一论述，同学们可以参考某一种程序设计语言进行学习，注意这些过程化结构中可以对表进行操作，这与一般的程序设计语言有所不同。

# 目录

- SQL数据类型与模式
- 完整性约束
- 嵌入式SQL
- 动态SQL (ODBC, JDBC)
- 安全性—授权
- 函数及过程\*\*
- 高级SQL特性 (SQL-03)\*\*
- 关于实验分析

# Advanced SQL Features\*\*

- Create a table with the **same schema** as an existing table:  
**create table temp\_account like account** 模式定义
- SQL:2003 allows subqueries to occur *anywhere* a value is required provided the subquery returns only one value. This applies to updates as well
- SQL:2003 allows subqueries in the **from** clause to access attributes of other relations in the **from** clause using the **lateral** construct: 查询, 允许访问前面子查询的属性

```
select C.customer_name, num_accounts
from customer C,
    lateral (select count(*)
             from account A
             where A.customer_name = C.customer_name )
as this_customer (num_accounts )
```

# Advanced SQL Features

- **Merge construct** allows batch processing of updates.
- Example: relation *funds\_received* (*account\_number*, *amount*) has batch of deposits to be added to the proper account in the *account* relation 更新的高级结构

```
merge into account as A  
using (select *  
        from funds_received as F)  
on (A.account_number = F.account_number)  
when matched then  
    update set balance = balance + F.amount
```

- 这些高级结构并没有从本质上增加新功能，但简化了操作，上面的例子都可以用前面的知识解决，但可能要麻烦一些



# 目录

- SQL数据类型与模式
- 完整性约束
- 嵌入式SQL
- 动态SQL (ODBC, JDBC)
- 安全性—授权
- 函数及过程\*\*
- 高级SQL特性 (SQL-03)\*\*
- 关于实验分析

# 如何做实验？

## 这里给出一个基本的方法

# 问题提出—发现问题

- 针对查询表达式:

```
SELECT  $A_1, A_2, \dots, A_n$   
FROM  $r_1, r_2, \dots, r_n$   
WHERE  $P$ 
```

- 有这样的描述:“SQL先构造from子句中关系的笛卡尔积, 根据where子句中的谓词进行关系代数的选择运算, 然后将结果投影到select子句的属性上”, SQL Server2000中是怎样实现的? 是按上述顺序实现的吗?

# 分析问题

- 找个具体例子:

```
select *  
from loan,branch  
where loan.branch_name=branch.branch_name
```

- 如果上述按描述的方式执行, 则运行时间应该是:

$T=t1+t2+t3$

t1: 计算笛卡尔积的时间(from)

t2: 选择时间(when)

t3: 投影时间(select)

因此,  $T \geq t1$ , 只要验证这个命题就可以了

# 实验步骤(验证)

## ■ 准备工作

- 数据准备: 两个足够大的表,并两表具有关联. 一个表有**316141**条记录, 另一个表具有**305**条记录
- 运行环境: **SQL Server 2000**, 查询分析器

## ■ 计算运行时间T

在查询分析器中执行下列语句:

```
select *  
from loan,branch  
where loan.branch_name=branch.branch_name
```

## ■ 计算运行时间t1

在查询分析器中执行下列语句:

```
select *  
from loan,branch
```

# 实验结果及结论

- 实验结果:

$T=46\text{ s}$

$t_1>1800\text{ s}$

- 实验结论

实验结果表明,  $T \geq t_1$  不成立, 而是  $T \leq t_1$ , 显然, 系统并不是按照上述描述的方式进行计算的

- 那么是如何计算的呢?

- 先做选择后做笛卡尔积?

- 等同连接操作?

- ...

# 小结

- SQL即是自含式语言又是嵌入式语言，前者一般是DBA用于交互式的数据数据库操作，后者则是嵌入到其他高级语言中与应用程序融为一体
- 动态SQL使宿主应用程序更灵活
- SQL提供了更多的与程序设计语言相当的功能，如函数、过程以及其他过程化结构
- ODBC是解决异构数据相互通信的一种标准,实现数据互操作的一种通用接口
- 完整性和安全性是实现数据保护的两种有效方法,前者保证了数据一致性,后者则保证了数据不被破坏