

21121350

Database System

Lecture 4: Advanced SQL

Lu Chen (陈璐)

College of Computer Science

Zhejiang University

Spring & Summer 2023

luchen@zju.edu.cn/18868818726

Outline

- ❑ SQL Data Types and Schemas
- ❑ Integrity Constraints
 - Domain Constraints
 - Referential Integrity
 - Assertions
 - Triggers
- ❑ Authorization
- ❑ Embedded SQL
- ❑ Dynamic SQL
- ❑ ODBC and JDBC



SQL Data Types and Schemas

- ❑ Built-in data types in SQL
- ❑ User-defined types
 - Structured data types
 - Distinct types
 - Create type *person_name* as varchar (20)
 - Create table *student*
 - (*sno* char(10) primary key,
 - sname* *person_name*,
 - ssex* char(1),
 - birthday* date)
 - Drop type *person_name*

SQL Data Types and Schemas (Cont.)

❑ Create new domain

Create domain **Dollars** as numeric(12, 2) **not null**;

Create domain **Pounds** as numeric(12,2);

Create table employee

(eno char(10) primary key,

ename varchar(15),

job varchar(10),

salary **Dollars**,

comm **Pounds**);

❑ Cf. domain vs. type

- Domain: Constraints, not strongly typed

SQL Data Types and Schemas (Cont.)

❑ Large-object types

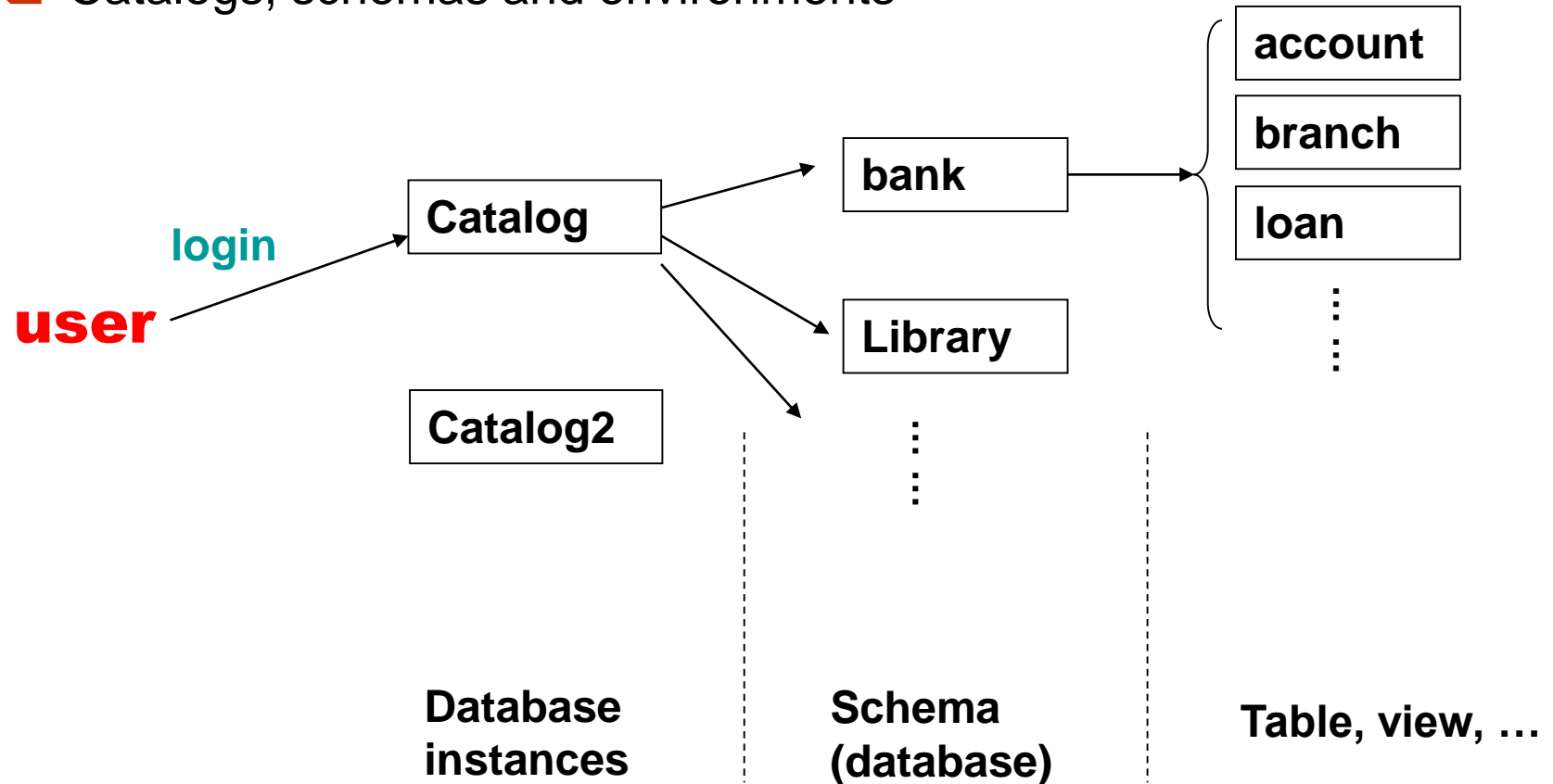
➤ Large objects (e.g., photos, videos, CAD files, etc.) are stored as a *large object*:

- **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)
- **clob**: character large object -- object is a large collection of character data
- When a query returns a large object, a **pointer** is returned rather than the large object itself.
- E.g., Create table students

```
(sid char(10) primary key,  
 name varchar(10),  
 gender char(1),  
 photo blob(20MB),  
 cv clob(10KB))
```

SQL Data Types and Schemas (Cont.)

❑ Catalogs, schemas and environments



Outline

- ❑ SQL Data Types and Schemas
- ❑ Integrity Constraints
 - Domain Constraints
 - Referential Integrity
 - Assertions
 - Triggers
- ❑ Authorization
- ❑ Embedded SQL
- ❑ Dynamic SQL
- ❑ ODBC and JDBC



Integrity Constraints

- ❑ Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
 - A checking account must have a balance greater than \$10,000.00
 - A salary of a bank employee must be at least \$4.00 an hour
 - A customer must have a (non-null) phone number
- ❑ 实体完整性、参照完整性和用户定义的完整性约束
- ❑ 完整性约束是数据库实例(Instance)必须遵循的
- ❑ 完整性约束由DBMS维护

Integrity Constraints (Cont.)

❑ Constraints on a **single relation**

- Not null
- Primary key
- Unique
- Check (P), where P is a predicate

❑ E.g.,

Create table *branch2*

```
(branch_name varchar(30) primary key,  
branch_city varchar(30),  
assets integer not null,  
check (assets >= 100))
```

Domain Constraints

- ❑ The **check** clause in SQL-92 permits **domains to be restricted**:

- E.g., Use check clause to ensure that an hourly-wage domain allows only values greater than a specified value.

Create domain *hourly-wage* numeric(5, 2)

Constraint *value-test* check(*value* > = 4.00)

- ❑ The clause **constraint *value-test*** is optional; useful to indicate which constraint an update violated.

Referential Integrity

Formal Definition:

- Let $r_1(R_1)$ and $r_2(R_2)$ be the relations with primary keys K_1 and K_2 , respectively.
- The subset α of R_2 is a *foreign key* referencing K_1 in relation r_1 , if for every t_2 in r_2 there must be a tuple t_1 in r_1 such that $t_1[K_1] = t_2[\alpha]$.
- Referential integrity constraint also called *subset dependency*, since its can be written as

$$\Pi_{\alpha}(r_2) \subseteq \Pi_{K_1}(r_1)$$

Referential Integrity (Cont.)

- ❑ Assume there exists relations r and s : $r(\underline{A}, B, C)$, $s(\underline{B}, D)$, we say attribute B in r is a **foreign key** from relation r , and r is called **referencing relation** (参照关系), and s is called **referenced relation** (被参照关系).
 - E.g., 学生(学号, 姓名, 性别, 专业号, 年龄) --- 参照关系
专业(专业号, 专业名称) --- 被参照关系 (目标关系)
其中属性专业号称为关系学生的外码。
 - Account(account-number, branch-name, balance) --- 参照关系
Branch(branch-name, branch-city, assets) --- 被参照关系
- ❑ 参照关系中外码的值必须在被参照关系中实际存在，或为null.

Checking Referential Integrity on Database Modification

- ❑ The following tests must be made in order to preserve the following **referential integrity constraint**:

$$\Pi_{\alpha}(r_2) \subseteq \Pi_K(r_1) \quad \text{----} \quad \alpha \text{ in } r_2 \text{ is a Foreign Key}$$

- ❑ **Insert**: If a tuple t_2 is inserted into r_2 , the system must ensure that there is a tuple t_1 in r_1 such that $t_1[K] = t_2[\alpha]$, i.e.,

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

Checking Referential Integrity on Database Modification (Cont.)

- ❑ **Delete**: If a tuple t_1 is deleted from r_1 , the system must compute the set of tuples in r_2 that reference t_1 :

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

如果 r_2 中存在与 r_1 中被删记录匹配的元组

If this set is **not empty**, then

- **Either the delete command is rejected as an error, or the tuples in t_2 that references t_1 must themselves be deleted (cascading deletions are possible).**

Checking Referential Integrity on Database Modification (Cont.)

□ **Update:** If a tuple t_2 is updated in relation r_2 and the update modifies values for foreign key α , then a test similar to the insert case is made:

➤ Let t_2' denote the new value of tuple t_2 . The system must ensure that

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

Referential Integrity in SQL

- ❑ Primary, candidate keys, and foreign keys can be specified as part of the SQL create table statement:
 - The **primary key** clause lists attributes that comprise the primary key.
 - The **unique key** clause lists attributes that comprise a **candidate key**.
 - The **foreign key** clause lists the attributes that comprise the foreign key, and the name of the relation referenced by the foreign key.

Referential Integrity in SQL (Cont.)

- ❑ By default, a foreign key references the primary key attributes of the referenced table:
 - E.g., foreign key (*account-number*) references *account*
- ❑ Short form for specifying a single column as foreign key:
 - E.g., *account-number* char (10) references *account*
- ❑ Reference columns in the referenced table can be explicitly specified
 - but must be declared as primary/candidate keys:
 - ... foreign key (*account-number*) references *account* (*account-number*)



Integrity constraint in SQL – Example

Create table *customer*

```
(customer-name    char(20),  
  customer-street varchar(30),  
  customer-city   varchar(30),  
  primary key (customer-name));
```

Create table *branch*

```
(branch-name      varchar(15),  
  branch-city     varchar(30),  
  assets           integer,  
  primary key (branch-name));
```

Integrity constraint in SQL – Example (Cont.)

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);
```

Cascading Actions in SQL

Create table *account* (

. . .

foreign key (*branch-name*) references *branch*

[on delete cascade]

[on update cascade]

. . .);

- ❑ Due to the on delete cascade clauses, if a delete of a tuple in *branch* results in referential-integrity constraint violation, the delete “cascades” to the *account* relation, deleting the tuple in *account* that refers to the *branch* which was deleted.
- ❑ Cascading updates are similar.

Cascading Actions in SQL (Cont.)

- ❑ If there is a chain of foreign-key dependencies across multiple relations, with on delete cascade specified for each dependency, **a deletion or update at one end of the chain can propagate across the entire chain.**
- ❑ But, if a cascading update or delete causes a **constraint violation** that cannot be handled by a further cascading operation, the system **aborts the transaction.**
 - As a result, **all the changes** caused by the transaction and its cascading actions are **undone.**

Cascading Actions in SQL (Cont.)

- ❑ Alternative to cascading:

- on delete set null
- on delete set default

- ❑ Null values in foreign key attributes complicate SQL referential integrity semantics, and are best prevented using not null.

- if any attribute of a foreign key is null, the tuple is defined to satisfy the foreign key constraint!

Cascading Actions in SQL (Cont.)

❑ Note: A constraint can alternatively be specified as not deferrable, which means it is checked immediately by default, but can be deferred when desired. For constraints declared as deferrable referential integrity is checked at the end of a transaction.

- That is: Intermediate steps are allowed to violate referential integrity provided later steps remove the violation by the end of the transaction.
- Otherwise, it would be impossible to create some database states, e.g., insert two tuples whose foreign keys point to each other
 - E.g., *spouse* attribute of relation:
marriedperson (name, address, spouse)

Assertions

- ❑ An *assertion* is a predicate expressing a condition that we wish the database always to satisfy. --- for complex check condition on several relations!

- ❑ An assertion in SQL takes the form:

```
CREATE ASSERTION <assertion-name>  
CHECK <predicate>;
```

- ❑ When an assertion is made, the system tests it for validity on every update that may violate the assertion. (when the *predicate* is true, it is Ok, otherwise report error.)
 - This testing may introduce a significant amount of overhead; hence assertions should be used with great care.

Assertions (Cont.)

❑ E.g., if we require “the sum of all loan amounts for each branch must be less than the sum of all account balances at the branch”.

❑ But SQL does not provide a construct for asserting:
for all X , $P(X)$

So it is achieved in a round-about fashion, using:

not exists X , such that not $P(X)$

$$\therefore (\forall x) P(x) = \neg (\exists x) \neg P(x)$$

Assertions (Cont.)

- ❑ **Example 1:** The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

```
CREATE ASSERTION sum-constraint CHECK
    (not exists (select * from branch B
        where (select sum(amount) from loan
            where loan.branch-name = B.branch-name)
            > (select sum(balance) from account
                where account.branch-name =
                    B.branch-name)))
```

- ❑ If the check() in assertion gets true, it is ok, otherwise rejection.

Assertions (Cont.)

- ❑ **Example 2:** Every loan has at least one borrower who maintains an account with a minimum balance of \$1000. (对每一笔借款, 至少有一个借款人有存款 \$1000 以上).

CREATE ASSERTION *balance-constraint* CHECK

(not exists (select * from *loan L*

where not exists (select *

For each loan-number,
to see if exists

from *borrower B*, *depositor D*, *account A*

where *L.loan-number* = *B.loan-number*

and *B.customer-name* = *D.customer-name*

and *D.account-number* = *A.account-number*

and *A.balance* >= 1000)))

loan(*loan-number*, *branch-name*, *amount*)

borrower(*customer-name*, *loan-number*)

account(*account-number*, *branch-name*, *balance*)

depositor(*customer-name*, *account-number*)

Triggers

- ❑ A trigger is a statement that is executed **automatically** by the system **as a side-effect of a modification** to the database.
- ❑ To design a trigger mechanism, we must:
 - Specify the **conditions** under which the trigger is to be executed.
 - Specify the **actions** to be taken when the trigger executes.
- ❑ Triggers were introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
- ❑ A kind of stored procedures

Trigger Example

- ❑ Example: Suppose that instead of allowing negative account balances, the **bank deals with overdrafts** by (the **actions**):
 - Setting the account balance to zero
 - Creating a loan in the amount of the overdraft, giving this loan a loan number identical to the account number of the overdrawn account
- ❑ The **condition** for executing the trigger is an **update** to the *account* relation that results in a **negative balance** value.

Trigger Example in SQL:1999

CREATE TRIGGER *overdraft-trigger* **after** update on *account*
referencing **new row** as *nrow* for each row
when *nrow.balance* < 0

actions

- begin **atomic**
- insert into *borrower*
(select *customer-name*, *account-number* from *depositor*
where *nrow.account-number* = *depositor.account-number*)
- insert into *loan* values
(*nrow.account-number*, *nrow.branch-name*, - *nrow.balance*)
- update *account* set *balance* = 0
where *account.account-number* = *nrow.account-number*
- end

loan(*loan-number*, *branch-name*, *amount*)
borrower(*customer-name*, *loan-number*)
account(*account-number*, *branch-name*, *balance*)
depositor(*customer-name*, *account-number*)

Triggering Events and Actions in SQL

- ❑ Triggering event can be **insert**, **delete** or **update**.
- ❑ Triggers **on update** can be **restricted to specific attributes**:
 - E.g., Create trigger *overdraft-trigger*
after update of *balance* **on** *account* ...
- ❑ Values of attributes **before** and **after** an update can be referenced:
 - Referencing old row as: for deletes and updates
 - Referencing new row as: for inserts and updates

Statement Level Triggers

- ❑ Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction.
 - Use *for each statement* instead of *for each row*
 - Use *referencing old table* or *referencing new table* to refer to temporary tables (called *transition tables*) containing the affected rows
 - Can be more efficient when dealing with SQL statements that update a large number of rows

External World Actions

- ❑ We sometimes require external world actions to be triggered on a database update
 - E.g., Re-ordering an item whose quantity in a warehouse has become small, or turning on an alarm light. （当库存低于警戒线，增加订货或报警）
- ❑ Triggers cannot be used to directly implement external-world actions, BUT
 - Triggers can be used to record actions-to-be-taken in a separate table
 - Have an external process that repeatedly scans the table, carries out external-world actions and deletes action from table.

External World Actions (Cont.)

□ E.g., Suppose a warehouse has the following tables

当前库存量 ➤ *inventory(item, level)*: How much of each item is in the warehouse presently

警戒线 ➤ *minlevel(item, level)*: What is the minimum desired level of each item

增订量 ➤ *reorder(item, amount)*: What quantity should we re-order at a time

定单 ➤ *orders(item, quantity)*: Orders to be placed (to be read by external process)

External World Actions (Cont.)

CREATE TRIGGER *reorder-trigger* 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* = *nrow.item*)
and *orow.level* > (select *level*
from *minlevel*
where *minlevel.item* = *orow.item*)

仅本次操作
造成越界



begin
insert into *orders*
(select *item*, *amount*
from *reorder*
where *reorder.item* = *orow.item*)
end

Trigger Example in MS-SQLServer Syntax

CREATE TRIGGER *overdraft-trigger* on *account*

for update as

if **inserted**.*balance* < 0

begin

insert into *borrower*

(select *customer-name*, *account-number*

from *depositor*, **inserted**

where **inserted**.*account-number* = *depositor.account-number*)

insert into *loan* values

(**inserted**.*account-number*, *inserted.branch-name*,

– *inserted.balance*)

update *account* set *balance* = 0

from *account*, **inserted**

where *account.account-number* = **inserted**.*account-number*

end

Inserted, deleted 相当于前法的 *nrow* (称为过渡表, *transition table*) 和 *orow*

Trigger Example in Oracle

Create or replace trigger *secure_student* **before insert or delete or update on student**

Begin

```
IF (to_char(sysdate, 'DY') in ('星期六', '星期日' ))  
    OR (to_char(sysdate, 'HH24') NOT Between 8 and 17)  
THEN raise_application_error(-20506, '你只能在上班时间  
修改数据' );
```

```
END IF;
```

```
END;
```

. (退出插入状态)

注：运行该程序，实际是对其进行编译，若出错，可查看数据字典中 *user_errors* 的出错信息。*User_triggers* 登记已建立了哪些触发器及所定义的内容。

❑ **删除触发器：** drop trigger <触发器名>

When Not To Use Triggers

- ❑ Triggers were used earlier years for tasks such as:
 - **Maintaining summary data** (e.g., total salary of each department)
 - **Replicating databases** by recording changes to special relations (called **change** or **delta relations**) and having a separate process that applies the changes over to a replica.

- ❑ There are better ways of doing these now:
 - Databases today provide **built in materialized view** facilities to maintain summary data;
 - Databases provide **built-in support for replication**.

- ❑ **Compare:**
 - Check
 - Assertion
 - Trigger

Outline

- ❑ SQL Data Types and Schemas
- ❑ Integrity Constraints
 - Domain Constraints
 - Referential Integrity
 - Assertions
 - Triggers
- ❑ **Authorization**
- ❑ Embedded SQL
- ❑ Dynamic SQL
- ❑ ODBC and JDBC



Authorization

❑ Security - protection from malicious attempts to steal or modify data.

- Database system level
 - Authentication and authorization mechanisms allow specific users access only to required data.
 - We concentrate on authorization in the rest of this chapter.
- Operating system level
 - Operating system super-users can do anything they want to the database! Good operating system level security is required.
- Network level: must use encryption to prevent
 - Eavesdropping (unauthorized reading of messages)
 - Masquerading (pretending to be an authorized user or sending messages supposedly from authorized users)
- Physical level
 - Physical access to computers allows destruction of data by intruders; traditional lock-and-key security is needed
 - Computers must also be protected from floods, fire, etc. -- (Recovery)
- Human level
 - Users must be screened to ensure that an authorized users do not give access to intruders
 - Users should be trained on password selection and secrecy

Authorization (Cont.)

❑ Forms of authorization on parts of the database:

- **Read authorization** - allows reading, but not modification of data.
- **Insert authorization** - allows insertion of new data, but not modification of existing data.
- **Update authorization** - allows modification, but not deletion of data.
- **Delete authorization** - allows deletion of data.

❑ Forms of authorization to modify the database schema:

- **Index authorization** - allows creation and deletion of indices.
- **Resources authorization** - allows creation of new relations.
- **Alteration authorization** - allows addition or modifying of attributes in a relation.
- **Drop authorization** - allows deletion of relations.

Authorization and Views

- ❑ Users can be given **authorization on views**, without being given any authorization on the relations used in the view definition.
- ❑ Ability of **views** to **hide data** serves both to **simplify usage** of the system and to **enhance security** by allowing users access only to data they need for their job.
- ❑ A **combination of relational-level security and view-level security** can be used to limit a user's access to precisely the data that user needs.

View Example

- ❑ Suppose a bank clerk needs to know the names of the customers of each branch, but is not authorized to see specific loan information.
 - Approach: **Deny direct access to the *loan*** relation, but **grant access to the view *cust-loan***, which consists only of the names of customers and the branches at which they have a loan.
 - The *cust-loan* view is defined in SQL as follows:

```
CREATE VIEW cust-loan as
SELECT branchname, customer-name
FROM borrower, loan
WHERE borrower.loan-number = loan.loan-number
```

View Example (Cont.)

- ❑ The clerk is authorized to see the result of the query:

```
SELECT *  
FROM cust-loan
```

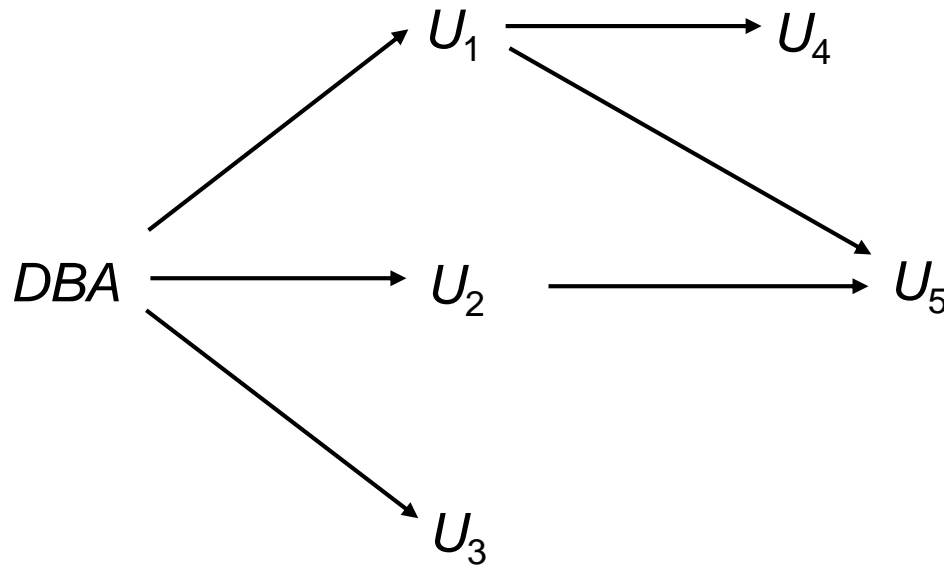
- ❑ When the query processor translates the result into a query on the actual relations in the database, we obtain a query on *borrower* and *loan*.
- ❑ Authorization must be checked on the clerk's query before query processing replaces a view by the definition of the view.

Authorization on Views

- ❑ Creation of view **does not require resources authorization** since no real relation is being created.
- ❑ The **creator of a view** gets only those privileges that provide no additional authorization beyond that he already had.
 - E.g., If creator of view *cust-loan* had only read authorization on *borrower* and *loan*, he gets only read authorization on *cust-loan*.

Granting of Privileges

- ❑ The passage of authorization from one user to another may be represented by an authorization graph.
 - The nodes of this graph are the users.
 - The root of the graph is the database administrator.
 - Consider graph for update authorization on loan.
 - An edge $U_i \rightarrow U_j$ indicates that user U_i has granted *update* authorization on *loan* to U_j .



Authorization Grant Graph

- ❑ **Requirement:** All edges in an authorization graph must be part of some path originating with the database administrator.
- ❑ If DBA **revokes** grant from U_1 :
 - Grant must be revoked from U_4 since U_1 no longer has authorization.
 - Grant must not be revoked from U_5 , since U_5 has another authorization path from DBA through U_2 .
- ❑ Must prevent cycles of grants with no path from the root:
 - DBA grants authorization to U_7
 - U_7 grants authorization to U_8
 - U_8 grants authorization to U_7
 - DBA revokes authorization from U_7
- ❑ Must revoke grant U_7 to U_8 and from U_8 to U_7 , since there is **no path** from DBA to U_7 or to U_8 anymore.

Security Specification in SQL

- ❑ The grant statement is used to confer authorization

```
GRANT <privilege list> ON <table | view>  
TO <user list>
```

- ❑ <user list> is:
 - **user-ids**
 - **public**, which allows all valid users the privilege granted
 - **A role** (more details about this later)
- ❑ Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- ❑ The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- ❑ **Select**: allows read access to relation, or the ability to query using the view

➤ Example: To grant users U_1 , U_2 , and U_3 , select and insert authorization on the *branch* relation:

```
SQL: GRANT select, insert ON branch TO  $U_1$ ,  $U_2$ ,  $U_3$ ;
```

- ❑ **Insert**: the ability to insert tuples.
- ❑ **Update**: the ability to update using the SQL update statement.
- ❑ **Delete**: the ability to delete tuples.
- ❑ **References**: ability to declare foreign keys when creating relations.
- ❑ **All privileges**: used as a short form for all the allowable privileges.
- ❑ **All**

Privileges in SQL

- ❑ **With grant option:** Allows a user who is granted a privilege to pass the privilege on to other users.

➤ Example:

grant select on *branch* to U_1 with grant option;
gives U_1 the select privileges on *branch* and
allows U_1 to grant this privilege to others.

Roles

- ❑ Roles permitting **common privileges** for a class of users can be specified just once, by creating a corresponding “**role**”.
- ❑ Privileges can be granted to or revoked from roles, just like user; roles can be assigned to users, and even to other roles.
- ❑ SQL:1999 supports roles

Create role **teller**;

Create role **manager**;

Grant select on *branch* to **teller**;

Grant update (*balance*) on *account* to **teller**;

Grant all privileges on *account* to **manager**;

Grant **teller** to **manager**;

Grant **teller** to *alice*, *bob*;

Grant **manager** to *avi*;

Revoking Authorization in SQL

- ❑ The revoke statement is used to revoke authorization.

```
REVOKE <privilege list> ON <table | view>  
FROM <user list> [restrict | cascade]
```

- ❑ Example:

Revoke select on *branch* from U_1 , U_3 cascade;

- ❑ Revocation of a privilege from a user may cause other users also to lose that privilege, which is referred to as **cascading of the revoke**.
- ❑ We can **prevent cascading** by specifying restrict:

Revoke select on *branch* from U_1 , U_3 restrict;

With restrict, the revoke command fails if cascading revokes are required.

Revoking Authorization in SQL (Cont.)

- ❑ <privilege-list> may be **ALL**, to revoke all privileges the revokee may hold.
- ❑ If <revokee-list> includes **PUBLIC**, all users lose the privilege except those granted it explicitly.
- ❑ If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- ❑ All privileges that depend on the privilege being revoked are also revoked.

Limitations of SQL Authorization

- ❑ SQL does not support authorization at a tuple level.
 - E.g., we cannot restrict students to see only (the tuples storing) their own grades by grant.
- ❑ With the growth in Web access to databases, database accesses come primarily from application servers.
 - End users don't have database user ids, they are all mapped to the same database user id.
- ❑ All end-users of an application (such as a web application) may be mapped to a single database user.

Limitations of SQL Authorization (Cont.)

- ❑ The task of authorization in above cases **falls on the application program, with no support from SQL**.
 - **Benefit**: Fine grained authorizations, such as to individual tuples, can be implemented by the application.
 - **Drawback**: Authorization must be done in application code, and may be dispersed all over an application.
 - **Checking for absence of authorization loopholes** becomes **very difficult** since it requires reading large amounts of application code.

Audit Trails

- ❑ An audit trail is **a log of all changes** (inserts/deletes/updates) to the database along with information such as **which user** performed the change, and **when** the change was performed.
- ❑ Used to **track erroneous/fraudulent updates**.
- ❑ Can be implemented using triggers, but **many database systems provide direct support**.

Example: Audit in Oracle

□ 语句审计:

- E.g., **audit table by scott by access whenever successful** ---- 审计用户scott每次成功地执行有关table的语句 (create table, drop table, alter table)。
- 格式: **AUDIT <st-opt> [BY <users>] [BY SESSION | ACCESS] [WHENEVER SUCCESSFUL | WHENEVER NOT SUCCESSFUL]**
 - 当 BY <users> 缺省, 对所有用户审计。
 - BY SESSION每次会话期间, 相同类型的需审计的SQL语句仅记录一次。
 - 常用的<St-opt>: table, view, role, index, ...
 - 取消审计: NOAUDIT ...(其余同audit语句)。

Example: Audit in Oracle (Cont.)

❑ 对象(实体)审计:

- E.g., **audit delete, update on student** --- 审计所有用户对student表的delete和update操作。
- 格式: **AUDIT <obj-opt> ON <obj> | DEFAULT [BY SESSION | BY ACCESS] [WHENEVER SUCCESSFUL | WHENEVER NOT SUCCESSFUL]**
 - obj-opt: insert, delete, update, select, grant, ...
- 实体审计对所有的用户起作用。
- ON <obj> 指出审计对象表、视图名。
- ON DEFAULT 对其后创建的所有对象起作用。
- 取消审计: **NOAUDIT ...**

❑ 怎样看审计结果:

- 审计结果记录在数据字典表: sys.aud\$中, 也可从dba_audit_trail, dba_audit_statement, dba_audit_object中获得有关情况。
- 上述数据字典表需在DBA用户 (system) 下才可见。

Examples

❑ Example 1: The **Person** table is defined as following:

Create table **Person**

(id char(10) primary key,
name varchar(12) not null,
age int,

gender char(1),

spouse char(10),

foreign key(spouse) references **Person**

on update cascade on delete set NULL,

Check(gender in {'f', 'm'});

id		spouse
A	...	E
B	...	D
C	...	
D	...	B
E	...	A

Write a constraint on **Person** to carry out the following action: **After set the spouse attribute of a person to NULL, set the spouse attribute of the person's spouse to NULL accordingly.** (For example, originally Adam and Eve are a couple. After Adam's spouse attribute is set to NULL, Eve's spouse attribute should also be set to NULL.)

Examples (Cont.)

❑ Answer to example 1:

Create trigger *chk_couple* after update of *spouse* on *Person* referencing new row as *nrow*, old row as *orow* for each row when *nrow.spouse* is null and *orow.spouse* is not null

Begin

 update *Person* set *spouse* = NULL

 where *id* = *orow.spouse* , *spouse* = *nrow.id*

End

Examples (Cont.)

- Example 2: Define a constraint over the relation Person (as given in Example 1) to indicate that the spouse relationship is **one to one** between two **heterosexual** persons.

Create assertion *spouse_assert1* **check**

(not exists (select * from *Person* as p_1 , *Person* as p_2
where $p_1.spouse = p_2.id$ and ($p_1.id \neq p_2.spouse$
or $p_1.gender = p_2.gender$)))

id	gender	spouse
A	m	E
B	f	D
C	m	G
D	f	B
E	f	A
G	m	D

Examples (Cont.)

Create assertion *spouse_assert2* check

(not exists (select * from *Person* as p_1 , *Person* as p_2
where $p_1.spouse = p_2.id$ and $p_1.gender = p_2.gender$))

and

not exists (select count(*) from *Person*
group by *spouse*
having count(*) > 1))

Examples (Cont.)

❑ Following is an instance of Person:

id	name	age	gender	spouse
01	Zhao	20	m	02
02	Qian	21	f	01
03	Sun	22	f	04
04	Li	20	m	03
05	Wang	24	m	08
06	Zhang	24	m	
07	Liu	24	f	
08	Ling	25	f	05
09	Jiang	25	m	

- ❑ What's the output of executing the following two SQL statements on the *Person* table above?
- Delete from *Person* where name = 'Wang'
 - Select count(*spouse*) from *Person*

Outline

- ❑ SQL Data Types and Schemas
- ❑ Integrity Constraints
 - Domain Constraints
 - Referential Integrity
 - Assertions
 - Triggers
- ❑ Authorization
- ❑ **Embedded SQL**
- ❑ Dynamic SQL
- ❑ ODBC and JDBC



Embedded SQL

- ❑ SQL的功能不完备性。（计算，资源...）
- ❑ The SQL standard defines embeddings of SQL in a variety of programming languages such as Pascal, PL/I, Fortran, C, and Cobol.
- ❑ A language in which SQL queries are embedded is referred to as a *Host language* (宿主语言), and the SQL structures permitted in the host language comprise *embedded SQL*.
- ❑ **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor:

EXEC SQL <embedded SQL statement> **END_EXEC**

Note: This varies by language, e.g., the Java embedding uses **# SQL { }**

Query

❑ 单行查询

```
EXEC SQL BEGIN DECLARE SECTION;
char V_an[20], bn[20];
float bal;
EXEC SQL END DECLARE SECTION;

.....

scanf("%s", V_an); // 读入账号,然后据此在下面的语句获得bn, bal的值
EXEC SQL SELECT branch_name, balance INTO :bn, :bal FROM
account WHERE account_number = :V_an;
END_EXEC
printf("%s, %s, %s", V_an, bn, bal);

.....
```

:*V_an*, :*bn*, :*bal*是宿主变量，可在宿主语言程序中赋值，从而将值带入SQL。宿主变量在宿主语言中使用时不加:号。

Query (Cont.)

❑ 多行查询

- E.g., From within a host language, find the names and cities of customers with more than the **variable amount** dollars in some account.

❑ Step1: Specify the query in SQL and declare a *cursor* for it

EXEC SQL

DECLARE *c* CURSOR FOR

SELECT *customer_name*, *customer_city*

FROM *depositor D*, *customer B*, *account A*

WHERE *D.customer_name* = *B.customer_name*

and *D.account_number* = *A.account_number*

and *A.balance* > :*v_amount*

END_EXEC

游标

Query (Cont.)

- ❑ **Step 2:** The OPEN statement causes the query to be evaluated:

EXEC SQL OPEN *c* END_EXEC

- ❑ **Step 3:** The FETCH statement causes the values of one tuple in the result of query to be placed on host language variables.

EXEC SQL FETCH *c* INTO :*cn*, :*ccity* END_EXEC

Repeated calls to fetch of successive tuples in the query result.

- A variable called SQLSTATE in the **SQLCA** (SQL communication area) gets set to '02000' to indicate no more data is available.

- ❑ **Step 4:** 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:** The above details vary with language, e.g., the Java embedding defines Java iterators to step through result tuples.

Example for embedded SQL – For Oracle Pro*c

Exec SQL include SQLCA; // SQL通讯区，是存放语句的执行状态的数据结构，其中//有一个变量sqlcode指示每次执行SQL语句的返回代码（success, not_success）。

```
Exec SQL BEGIN DECLARE SECTION;
```

```
    char bn[20], bc[30];
```

```
Exec SQL END DECLARE SECTION;
```

```
Exec SQL DECLARE branch_cur CURSOR FOR
```

```
    Select branch_name, branch_city From branch;
```

```
.....
```

```
Exec SQL OPEN branch_cur;
```

```
    While (1) {Exec SQL FETCH branch_cur INTO :bn, :bc;
```

```
        if (sqlca.sqlcode <> SUCCESS) BREAK;
```

```
        ..... // 由宿主语句对bn, bc中的数据进行相关处理
```

```
    }
```

```
Exec SQL CLOSE branch_cur;
```

```
.....
```

(Oracle+C: Pro*C程序 → 预编译pcc → compiler cc → linker → .exe)

(SQL Server2000+VC++:

nsqlprep ESQL_File /SQLACCESS /DB server_name.database_name /PASS
login.password)

Updates

❑ 单行的修改

```
Exec SQL BEGIN DECLARE SECTION;
```

```
    char an[20];
```

```
    float bal;
```

```
Exec SQL END DECLARE SECTION;
```

```
.....
```

```
scanf("%s, %d", an, &bal); // 读入账号及要增加的存款额
```

```
EXEC SQL update account set balance = balance + :bal
```

```
    where account_number = :an;
```

```
.....
```

Updates (Cont.)

- ❑ 多行的修改: Can update tuples fetched by cursor by declaring that the cursor is for update.

```
Exec SQL BEGIN DECLARE SECTION;
```

```
    char an[20];
```

```
    float bal;
```

```
Exec SQL END DECLARE SECTION;
```

```
EXEC SQL DECLARE csr CURSOR FOR
```

```
    SELECT *
```

```
    FROM account
```

```
    WHERE branch_name = 'Perryridge'
```

```
    FOR UPDATE OF balance;
```

```
.....
```

Updates (Cont.)

(To update tuple at the current location of cursor)

```
EXEC SQL OPEN csr;
```

```
While (1) {
```

```
EXEC SQL FETCH csr INTO :an, :bn, :bal;
```

```
if (sqlca.sqlcode <> SUCCESS) BREAK;
```

```
..... // 由宿主语句对an, bn, bal中的数据进行相关处理(如打印)
```

```
EXEC SQL update account
```

```
set balance = balance + 100
```

```
where CURRENT OF csr;
```

```
}
```

```
.....
```

```
EXEC SQL CLOSE csr;
```

```
.....
```

Or:

```
EXEC SQL delete from account
```

```
where current of csr
```



Outline

- ❑ SQL Data Types and Schemas
- ❑ Integrity Constraints
 - Domain Constraints
 - Referential Integrity
 - Assertions
 - Triggers
- ❑ Authorization
- ❑ Embedded SQL
- ❑ **Dynamic SQL**
- ❑ ODBC and JDBC



Dynamic SQL

- ❑ Allows programs to construct and submit SQL queries at run time.
- ❑ Example: use dynamic SQL from within a C program.



```
char *sqlprog = "update account  
        set balance = balance * 1.05  
        where account_number = ?"  
  
EXEC SQL PREPARE dynprog FROM :sqlprog;  
  
char v_account [10] = "A_101";  
  
.....
```

- ❑ EXEC SQL EXECUTE *dynprog* USING :*v_account*;
- ❑ The dynamic SQL program contains a **?**, which is a *place holder* (占位符) for a value that is provided by '**using**' **variable** when the SQL program is executed.

Outline

- ❑ SQL Data Types and Schemas
- ❑ Integrity Constraints
 - Domain Constraints
 - Referential Integrity
 - Assertions
 - Triggers
- ❑ Authorization
- ❑ Embedded SQL
- ❑ Dynamic SQL
- ❑ ODBC and JDBC



ODBC

❑ Open DataBase Connectivity (ODBC, 开放数据库互连)

- A standard for application program to communicate with a database server.
- By application program interface (API) to
 - Open a connection with a database,
 - Send queries and updates,
 - Get back results.

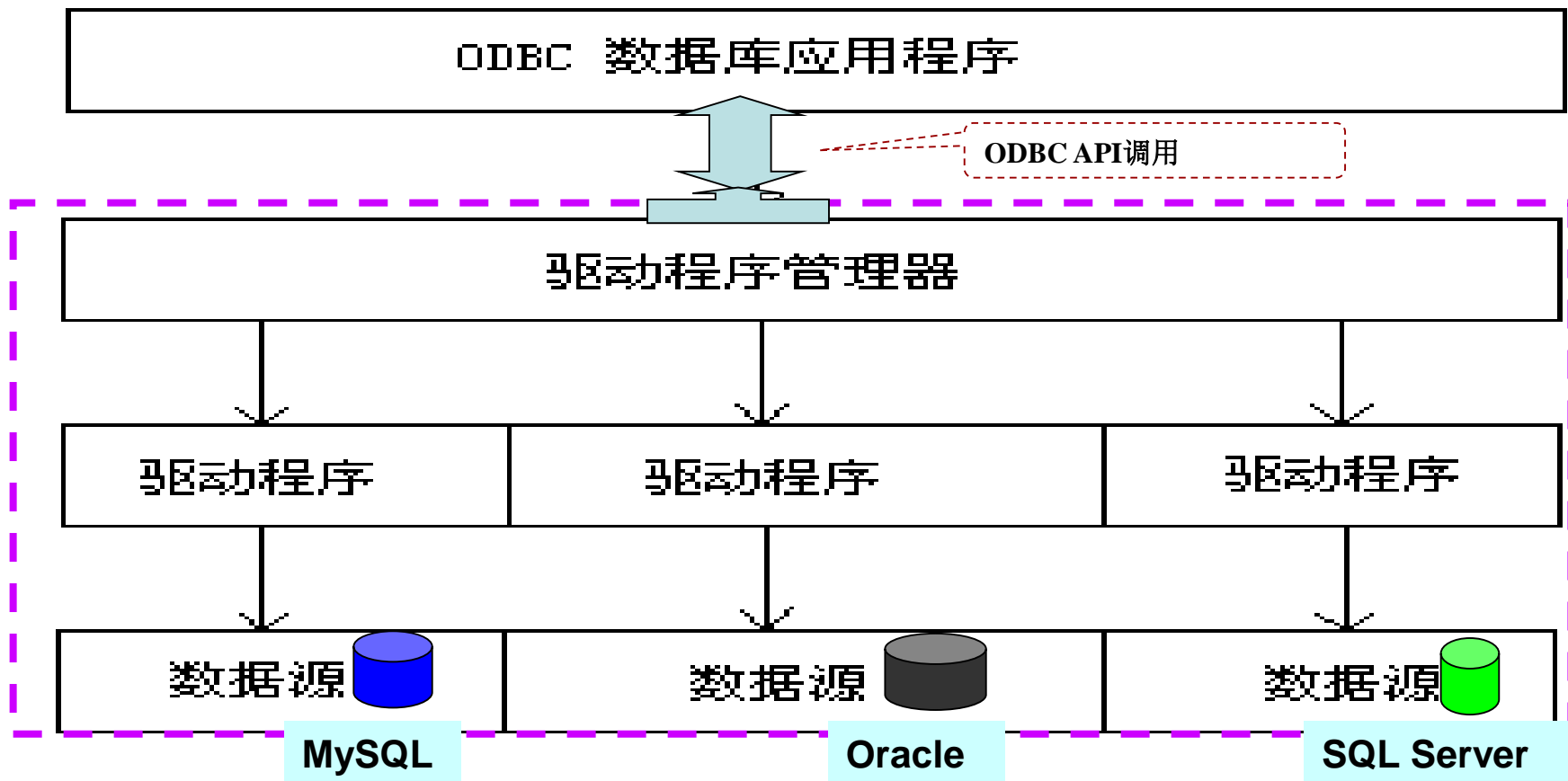
❑ Applications such as GUI, spreadsheets, etc. can use ODBC

❑ Embedded SQL vs. ODBC

- Embedded SQL: The pre_compiler is DBMS_specific.
- ODBC provides a standardized way for connection of database to the application programmers through API.
 - Not DBMS_specific.
 - Need not precompiling.

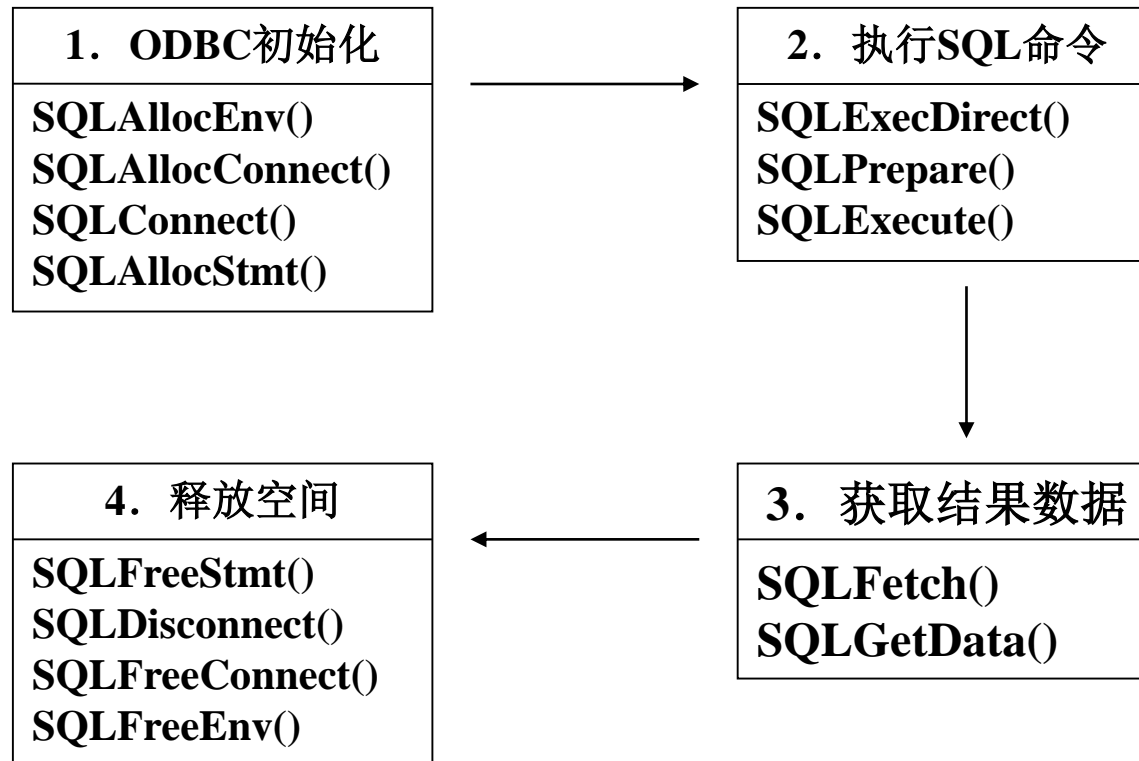
ODBC (Cont.)

- ❑ ODBC提供了一个公共的、与具体数据库无关的应用程序设计接口API。它为开发者提供单一的编程接口，这样同一个应用程序就可以访问不同的数据库服务器。
- ❑ 使用ODBC访问数据库的方法：
 - ODBC API访问数据库
 - Visual C++的MFC提供了丰富的ODBC类，它们封装了大量的函数用以完成数据库的大部分应用
- ❑ 访问数据库的其他方法：
 - OLE DB (Object Link and Embedding DataBase) --- 是一套通过COM (Component Object Model,组件对象模型)接口访问数据库的ActiveX的底层接口技术,速度快，支持关系型和非关系型数据库，编程量大。
 - ADO---基于COM，建立在OLE DB 之上，更易于使用.
 - DAO (Data Access Objects)



- ❑ 首先要定义ODBC数据源：控制面板—>管理工具—> 数据源(ODBC)
—> DSN (data source name)

ODBC编程基本流程

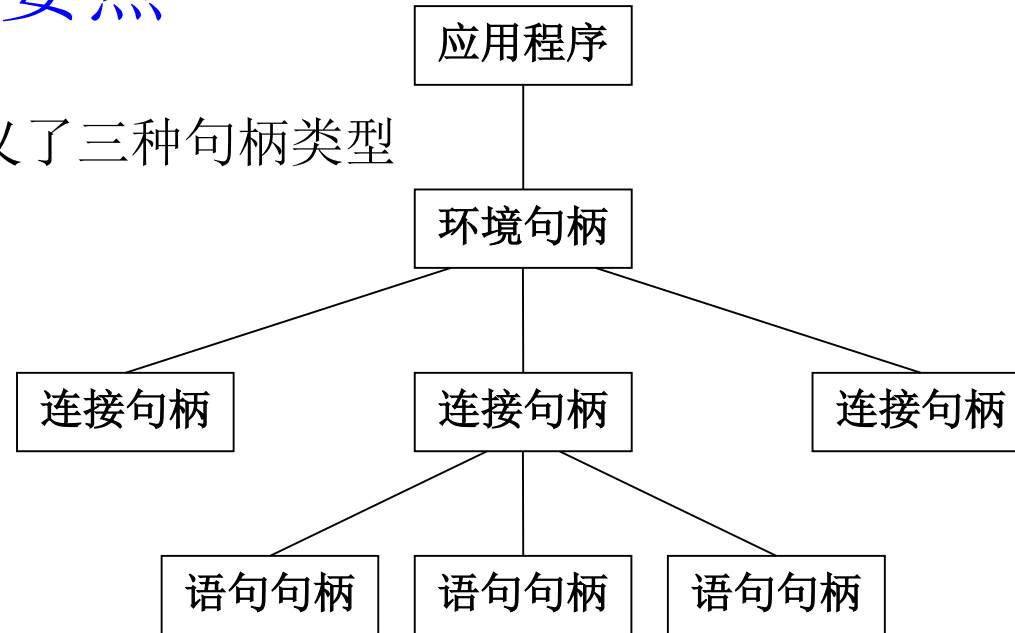


ODBC (Cont.)

- ❑ ODBC program first allocates an SQL environment, then a database connection **handle**.
- ❑ Opens database connection using SQLConnect(). Parameters for SQLConnect are as follows:
 - Connection handle
 - The server to which to connect
 - The user identifier
 - Password
- ❑ Must also specify types of arguments:
 - SQL_NTS denotes previous argument is a null_terminated string.

ODBC编程要点

❑ ODBC接口定义了三种句柄类型



1. 分配环境句柄
HENV henv;
SQLAllocEnv (&henv);

2. 分配连接句柄
HDBC hdbc;
SQLAllocConnect(henv, &hdbc);

3. 用已分配的连接句柄连接数据源
SQLConnect (hdbc, szDSN, cbDSN, szUID, cbUID, szAuthStr, cbAuthStr);
说明: hdbc是一个已分配的连接句柄;
szDSN和cbDSN分别表示系统所要连接的数据源名称字符串及其长度;
szUID和cbUID分别表示连接数据源的用户名字符串及其长度
szAuthStr和cbAuthStr分别表示连接数据源的权限字符串及其长度。

4. 分配语句句柄

```
HSTMT hstmt;  
SQLAllocStmt (hdbc, &hstmt);
```

5. 1直接执行SQL语句

```
SQLExecDirect ( hstmt, szSqlStr, cbSqlStr );
```

说明：hstmt是一个有效的语句句柄；

szSqlStr和cbSqlStr分别表示将要执行的SQL语句的字符串及其长度。

例子：retcode=SQLExecDirect(hstmt, “delete from book where ISBN=1”, SQL_NTS);

说明：删除book表中ISBN=1的记录。SQL_NTS是ODBC的一个常数，当字符串是以NULL结束时，可用它来表示字符串的长度。

5.2有准备地执行SQL语句

如果SQL语句需要执行几次，则采用有准备的执行更好，避免了SQL语句的多次分析。有准备的执行需要两个函数。

```
SQLPrepare ( hstmt, szSqlStr, cbSqlStr);
```

说明：SQL语句准备函数，参数同SQLExecDirect。

```
SQLExecute ( hstmt );
```

说明：SQL语句执行函数。

6. 查询结果的获取

```
SQLFetch(hstmt);
```

说明：把游标移到下一行，当查询语句执行后第一次调用时移到结果集的第一行。

```
SQLGetData(hstmt, icol, fCType, rgbValue, cbValueMax, pcbValue);
```

说明：读取游标指向行的列值。

icol和fCType分别表示结果集的列号和类型；

rgbValue和cbValueMax是接收数据存储区的指针和最大长度；

pcbValue是返回参数，表示本次调用后实际接收到的数据的字节数。

ODBC编程要点 (Cont.)

7.释放语句句柄

`SQLfreeStmt(hstmt, foption);`

说明：`foption`指定选项，一个选项是用`SQL_DROP`表示释放所有与该句柄相关的资源。

8.断开数据源连接

`SQLDisconnect(hdbc);`

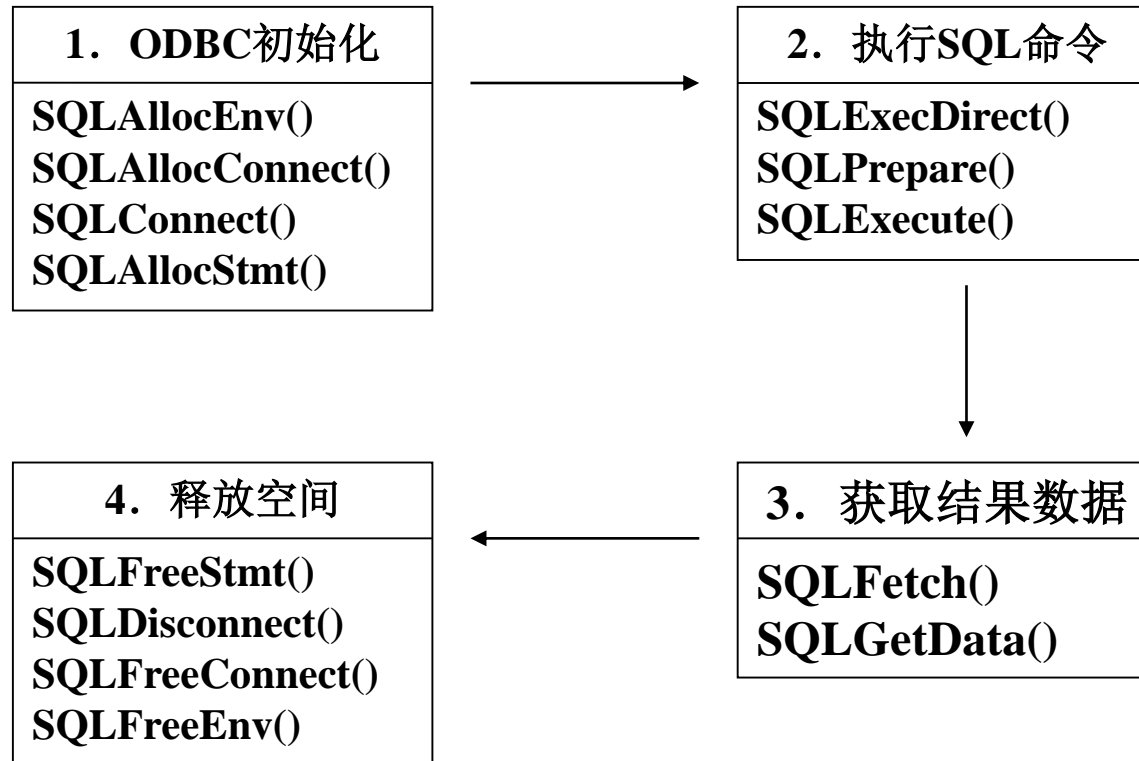
9.释放连接句柄

`SQLFreeConnect(hdbc);`

10.释放环境句柄

`SQLFreeEnv(henv);`

ODBC编程基本流程



//查询结果和自定义变量进行绑定
SQLBindCol()
SQLFetch()

Example of ODBC Code

```
❑ int ODBCexample() // 程序结构
{
    RETCODE error;
    HENV env; /* environment */
    HDBC conn; /* database connection */
    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn); /* 建立连接句柄 */
    SQLConnect (conn, "MySQLServer", SQL_NTS, "user", SQL_NTS, "password",
                SQL_NTS); /* 建立用户user与数据源的连接, SQL_NTS表示前
                           一参量以null结尾 */
    { .... Main body of program ... } // See next pages
    SQLDisconnect(conn);
    SQLFreeConnect(conn);
    SQLFreeEnv(env);
}
```

Example of ODBC Code (Cont.)

❑ Main body of program

```
.....  
{char branchname[80];  
float balance;  
int lenOut1, lenOut2;  
HSTMT stmt;  
    SQLAllocStmt(conn, &stmt); /* 为该连接建立数据区，将来存放查询结果 */  
char * sqlquery = "select branch_name, sum (balance) from account  
                    group by branch_name"; /* 装配SQL语句 */  
error = SQLExecDirect(stmt, sqlquery, SQL_NTS); /* 执行sql语句,查询结果存放到  
                    数据区stmt，同时sql语句执行状态的返回值送变量error*/
```

Example of ODBC Code (Cont.)

❑ Main body of program (Cont.)

```
{ .....  
    if (error == SQL_SUCCESS) {  
        SQLBindCol(stmt, 1, SQL_C_CHAR, branchname, 80, &lenOut1);  
        SQLBindCol(stmt, 2, SQL_C_FLOAT, &balance, 0, &lenOut2);  
        /* 对stmt中的返回结果数据加以分离，并与相应变量绑定。第1项数据转换为C的字符  
        类型，送变量branchname(最大长度为80)， lenOut1为实际字符串长度（若=-1代表  
        null），第2项数据转换为C的浮点类型送变量balance中 */  
        while ( SQLFetch(stmt) >= SQL_SUCCESS) { /* 逐行从数据区stmt中取数据，放到  
        绑定变量中 */  
            printf (" %s %d\n", branchname, balance);  
            /* 对取出的数据进行处理*/      ... ..      }  
        } .....  
    }  
    SQLFreeStmt(stmt, SQL_DROP); /* 释放数据区*/
```

ODBC Code

- ❑ Program sends SQL commands to the database by using `SQLExecDirect`.
- ❑ Result tuples are fetched using `SQLFetch()`.
- ❑ `SQLBindCol()` binds C language variables to attributes of the query result.
 - When a tuple is fetched, its attribute values are automatically stored in corresponding C variables.
 - Arguments to `SQLBindCol()`
 - ODBC stmt variable, attribute position in query result.
 - The type conversion from SQL to C.
 - The address of the variable.
 - For `variable_length` types like character arrays
 - The maximum length of the variable.
 - Location to store actual length when a tuple is fetched.
 - Note: A negative value returned for the length field indicates null value.
- ❑ Good programming requires checking results of every function call for errors; we have omitted most checks for brevity.

ODBC Conformance Levels

- ❑ ODBC **Conformance levels** specify subsets of the functionality defined by the standard. (不同版本ODBC提供不同等级的标准)
 - Core level.
 - Level 1 requires support for metadata querying.
 - Level 2 requires ability to send and retrieve arrays of parameter values and more detailed catalog information.
- ❑ SQL Call Level Interface (CLI) standard similar to ODBC interface, but with some minor differences.
 - E.g., Oracle call interface (OCI).
- ❑ 参考书：《Visual C++6.0数据库开发技术与工程实践》，求是科技，人民邮电出版社, 2003.1.

JDBC

- ❑ JDBC is a Java API for communicating with database systems supporting SQL.
- ❑ JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- ❑ JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.
- ❑ Model for communicating with the database:
 - Open a connection
 - Create a “statement” object
 - Execute queries using the Statement object to send queries and fetch results
 - Exception mechanism to handle errors

JDBC Code

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection
            ("jdbc:oracle:thin:@aura.bell_labs.com:2000:bankdb", userid, passwd);
        Statement stmt = conn.createStatement();
        ... Do Actual Work ....
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

JDBC Code (Cont.)

❑ Update to database:

```
try {  
    stmt.executeUpdate("insert into account values  
                        ('A_9732', 'Perryridge', 1200)");  
} catch (SQLException sqle) {  
    System.out.println("Could not insert tuple. " + sqle);  
}
```

❑ Execute query and fetch and print results:

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

JDBC Code Details

❑ Getting result fields:

- `rs.getString("branchname")` and `rs.getString(1)` equivalent if `branchname` is the first argument of select result.

❑ Dealing with Null values

```
int a = rs.getInt("a");
```

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


Prepared Statement

- ❑ Prepared statement allows queries to be compiled and executed multiple times with different arguments

```
PreparedStatement pstmt = conn.prepareStatement(  
    "insert into account values(?,?,?)"); pstmt.setString(1, "A_9732");  
pstmt.setString(2, "Perryridge");  
pstmt.setInt(3, 1200);  
pstmt.executeUpdate();  
pstmt.setString(1, "A_9733");  
pstmt.executeUpdate();
```

- ❑ Beware: If value to be stored in database contains a single quote or other special character, prepared statements work fine, but creating a query string and executing it directly would result in a syntax error!

Q & A



Your questions and
suggestions are
expected for me.

Thanks a lot!

Questions?
Questions?

Exercises: 4.7, 4.18, 5.4, 5.15, and 5.19 (see
Pages 177-179 and Pages 233-236)