Lecture 4 Advanced SQL

Shuigeng Zhou

March 19, 2014

School of Computer Science
Fudan University

Outline

- SQL data types and schemas
- Domain Constraints (域约束)
- Referential Integrity (引用完整性)
- Assertions (断言)
- Triggers (触发器)
- Security
- Authorization
- Authorization in SQL
- Embedded SQL
- Dynamic SQL
- ODBC and JDBC
- Functions and procedures
- Recursive SQL
- Advanced features

Built-in Data Types in SQL

- **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

Built-in Data Types in SQL (cont'd)

- Can extract values of individual fields from date/time/timestamp
  - Example: **extract** (**year from** r.starttime)

- Can cast string types to date/time/timestamp
  - Example: **cast** <string-valued-expression> **as date**
  - Example: **cast** <string-valued-expression> **as time**

Large-Object Types

- Large objects (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

User-Defined Types

- **create type** construct in SQL creates user-defined type [SQL: 1999]

  create type *Dollars* **as numeric (12,2) final**

- **create domain** construct in SQL-92 creates user-defined domain types [SQL: 1992]

  create domain *person_name* **char**(20) **not null**

- New domains can be created from existing data types
  - E.g. **create domain** *Dollars* **numeric**(12, 2)
    create domain *Pounds* **numeric**(12,2)
- Types and domains are some different
  - Domains can have constraints, such as **not null**, specified on them
  - Domains are not strongly typed, which means that values of one domain can be assigned to values of another domain as long as the underlying types are compatible

Domain 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

- Domain constraints are the most elementary form of integrity constraint

Domain Constraints (Cont.)

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

  create domain *hourly-wage* **numeric(5,2)**
  constraint *value-test* **check**(*value* > = 4.00)

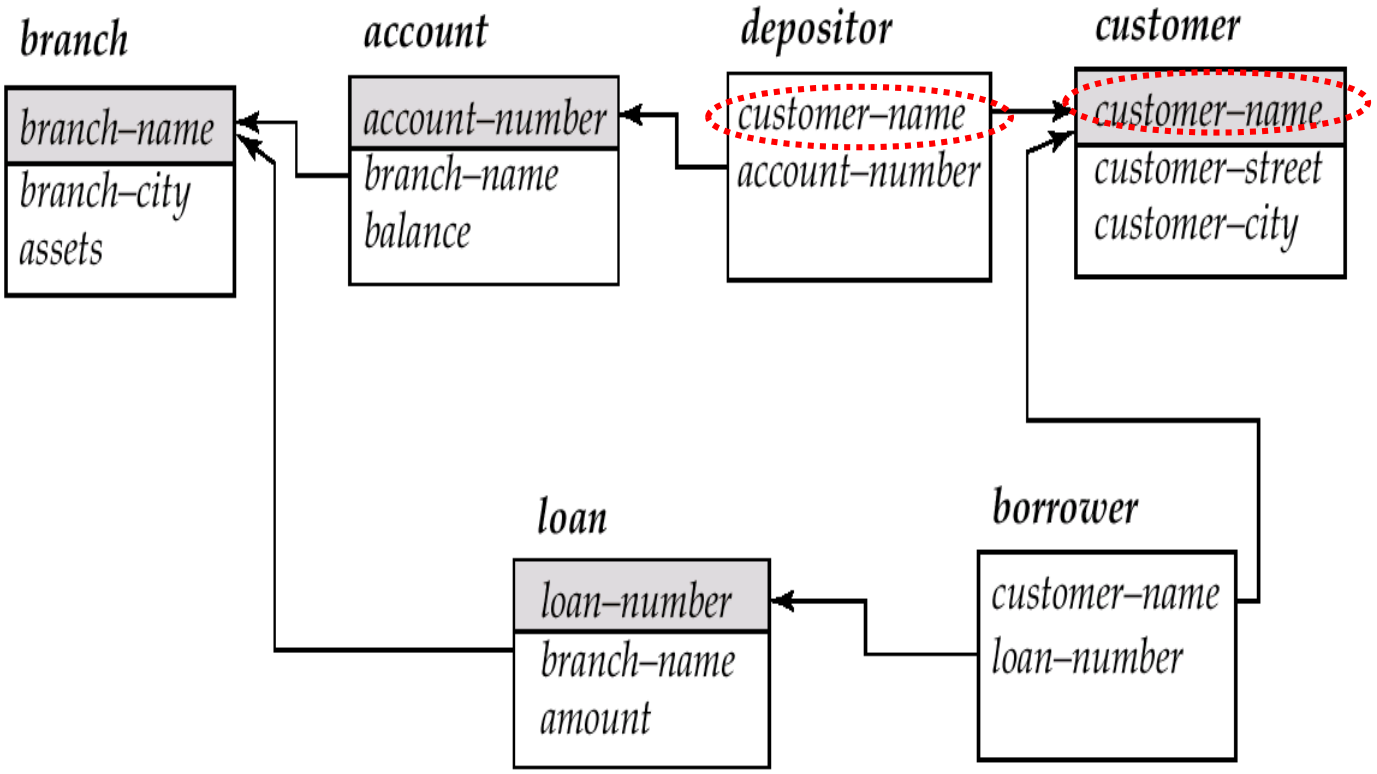
- Can have complex conditions in domain check
  - **create domain** *AccountType* **char**(10)
    constraint *account-type-test*
    check (**value in** ('Checking', 'Saving'))
  - **check** (*branch-name* **in** (**select** *branch-name* **from** *branch*))

Referential Integrity

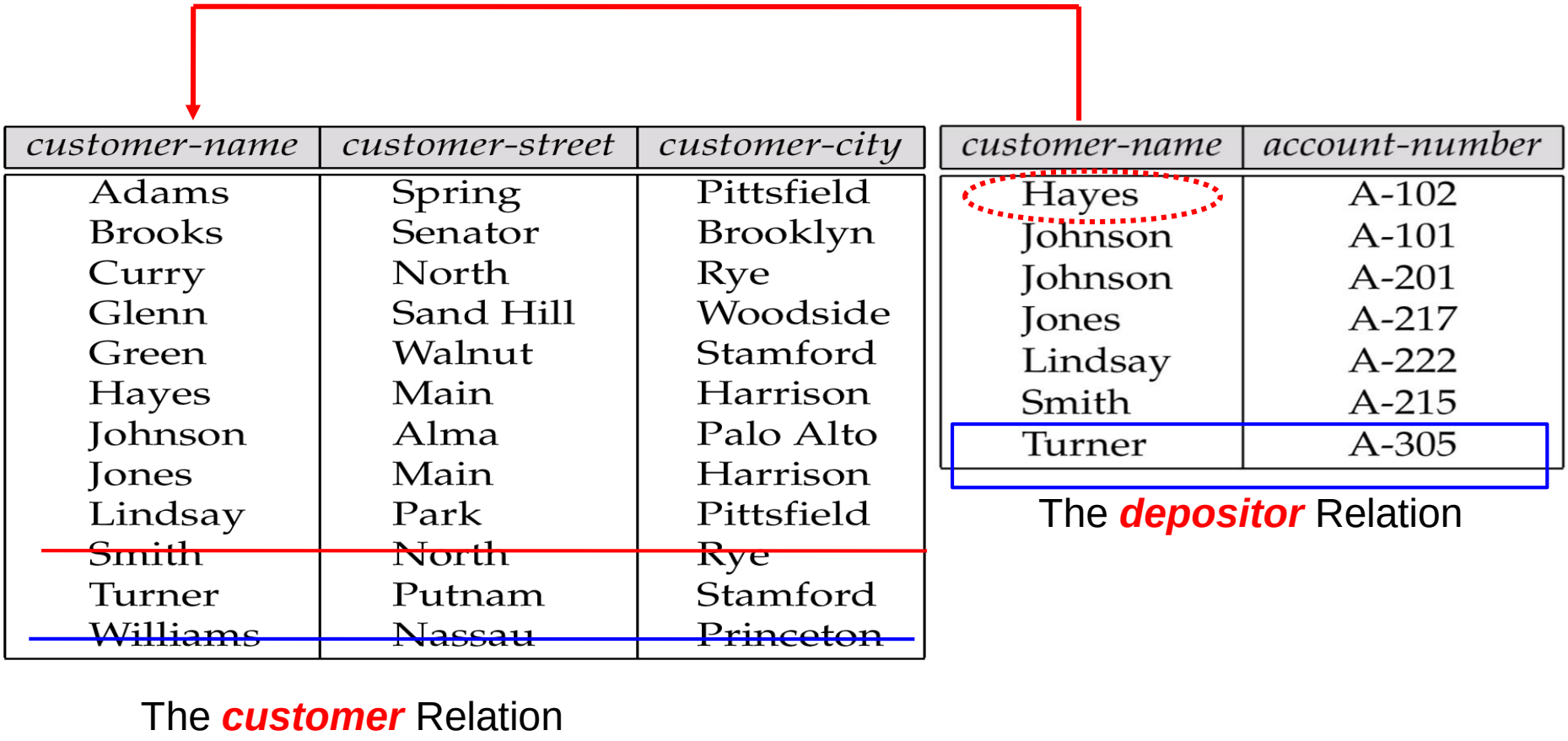
- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation
- Formal Definition
  - Let $r_1(R_1)$ and $r_2(R_2)$ be relations with primary keys K_1 and K_2 respectively
  - The subset α of R_2 is a ***foreign key*** 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

$$\Pi_{\alpha}(r_2) \subseteq \Pi_{K1}(r_1)$$

Database Modification



Database Modification



customer-name	*customer-street*	*customer-city*
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

The ***customer*** Relation

customer-name	*account-number*
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

The ***depositor*** Relation

Database Modification

- **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 $t_1[K] = t_2[\alpha]$. That is $t_2[\alpha] \in \Pi_K(r_1)$
- **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: $\sigma_{\alpha = t1[K]}(r_2)$

  If this set is not empty
  - either the delete command is rejected as an error, or
  - the tuples that reference t_1 must themselves be deleted (cascading deletions are possible)

Database Modification (Cont.)

- **Update.** There are two cases:
  - 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.
  - If a tuple t_1 is updated in r_1, and the update modifies values for the primary key (K), then a test similar to the delete case is made:
    1. The system must compute

       $\sigma_{\alpha = t1[K]} (r_2)$

       using the old value of t_1
    2. If this set is not empty
       1. the update may be rejected as an error, or
       2. the update may be cascaded to the tuples in the set, or
       3. the tuples in the set may be deleted.

Referential Integrity in SQL

- Primary and candidate keys and foreign keys can be specified as part of the SQL **create table** statement:
  - The **primary key** clause
  - The **unique key** clause
  - The **foreign key** clause
- By default, a foreign key references the primary key attributes of the referenced table

  foreign key (*account-number*) **references** *account*
- Short form for specifying a single column as foreign key

  account-number **char** (10) **references** *account*
- Reference columns in the referenced table can be explicitly specified

  foreign key (*account-number*) **references** *account*(*account-number*)

Referential Integrity in SQL – Example

```
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)
```

```
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))
```

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 that refers to the branch that 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**
- Referential integrity is only checked at the **end** of a transaction
  - Intermediate steps are allowed to violate referential integrity provided later steps remove the violation
  - 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)

Referential Integrity 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!

Assertions

- An *assertion* is a predicate expressing a condition that we wish the database always to satisfy
- An assertion in SQL takes the form

  create assertion <assertion-name> **check** <predicate>
- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion
  - This testing may introduce a significant amount of overhead; hence assertions should be used with great care

Assertion Example

- 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
               where (select sum(amount) from loan
                      where loan.branch-name =
                            branch.branch-name)
                     >= (select sum(amount) from account
                         where loan.branch-name =
                               branch.branch-name)))
```

Assertion Example

- Every loan has at least one borrower who maintains an account with a minimum balance at least $1000.00

```
create assertion balance-constraint check
  (not exists (
      select * from loan
    where not exists (
        select *
        from borrower, depositor, account
        where loan.loan-number = borrower.loan-number
            and borrower.customer-name = depositor.customer-name
            and depositor.account-number = account.account-number
            and account.balance >= 1000)))
```

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 introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases

Trigger Example

- Suppose that instead of allowing negative account balances, the bank deals with overdrafts by
  - 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
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
```

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attr.
  - **E.g. ... 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
- Triggers can be activated before an event, which can serve as extra constraints

```
create trigger setnull-trigger before update on r
referencing new row as nrow
for each row
    when nrow.phone-number = ' '
    set nrow.phone-number = null
```

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,
- 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
- E.g. Suppose a warehouse has the following tables
  - *inventory(item, level)*: How much of each item is in the warehouse
  - *minlevel(item, level)* : What is the minimum desired level
  - *reorder(item, amount)*: What quantity should we re-order
  - *orders(item, amount)* : Orders to be placed

External World Actions (Cont.)

```
create trigger reorder-trigger after update of amount 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
    insert into orders
        (select item, amount
         from reorder
         where reorder.item = orow.item)
end
```

Triggers 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
```

When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - maintaining summary data (e.g. total salary of each department)
  - Replicating databases by recording changes to special 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
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of through a trigger

Security

Security

- **Security** - protection from malicious attempts to steal or modify data.
  - Database system level
    - Authentication and Authorization mechanisms to allow specific users access only to required data
  - Operating system level
    - Operating system super-users can do anything they want to the database!
  - 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)

Security (Cont.)

- 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.
    - More in Chapter 17 (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

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

Authorization (Cont.)

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 deletion 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 or relational-level security and view-level security can be used to limit a user's access precisely to the data that user needs

View Example

- Suppose a bank clerk needs to know the names of the customers in 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
```

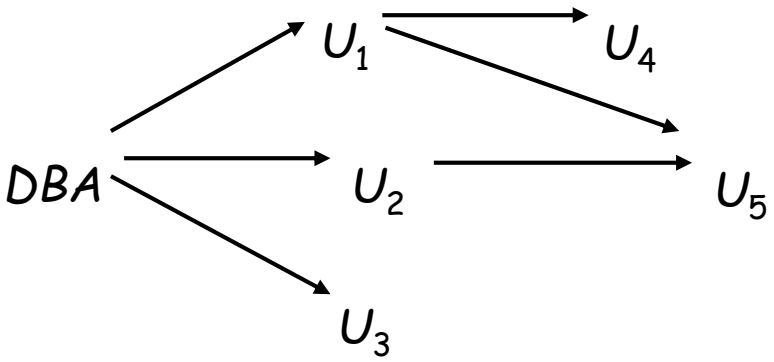
- When the query processor translates the result into a query on the actual relations in the database, we obtain a query on *borrower* and *loan*
- Authorization must be checked on the clerk's query before query processing replaces a view by the definition of the view.

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 transferring 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
- Eg. 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 root
- 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
  - U7 grants authorization to U_8
  - U8 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 <relation name or view name> to <user list>

- <user list> is:
  - a user-id
  - *public*, which allows all valid users the privilege granted
  - A role (more on 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

Privileges in SQL

- **select:** allows read access to relation,or the ability to query using the view
  - Example: **grant select 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
- **usage**: In SQL-92; authorizes a user to use a specified domain
- **all privileges**: used as a short form for all the allowable privileges

Privilege To Grant Privileges

- **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 (角色)

- Roles permit 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 <relation name or view name> **from** <user list> [**restrict**|**cascade**]

- Example:

  revoke select on *branch* **from** U_1, U_2, U_3 **cascade**

- Revocation of a privilege from a user may cause other users also to lose that privilege; referred to as cascading of the **revoke**

- We can prevent cascading by specifying **restrict**:

  revoke select on *branch* **from** U_1, U_2, 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 their own grades
- 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
- 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

Encryption

- Data may be *encrypted* when database authorization provisions do not offer sufficient protection
- Properties of good encryption technique:
  - Relatively simple for authorized users to encrypt and decrypt data
  - Encryption scheme depends not on the secrecy of the algorithm but on the secrecy of a parameter of the algorithm called the encryption key
  - Extremely difficult for an intruder to determine the encryption key

Encryption (Cont.)

- *Data Encryption Standard* (DES)
  - substitutes characters and rearranges their order on the basis of an encryption key which is provided to authorized users via a secure mechanism. Scheme is no more secure than the key transmission mechanism since the key has to be shared.
- Advanced Encryption Standard (AES)
  - a new standard replacing DES, and is based on the Rijndael algorithm, but is also dependent on shared secret keys
- *Public-key encryption*
  - based on each user having two keys:
    - *public key* – used to encrypt data, but cannot be used to decrypt data
    - *private key* -- used to decrypt data.
  - Encryption scheme is such that it is impossible or extremely hard to decrypt data given only the public key.
  - The RSA public-key encryption scheme is based on the hardness of factoring a very large number (100's of digits) into its prime components.

Authentication

- Password based authentication is widely used, but is susceptible to sniffing on a network
- **Challenge-response** systems avoid transmission of passwords
  - DB sends a (randomly generated) challenge string to user
  - User encrypts string and returns result
  - DB verifies identity by decrypting result
  - Can use public-key encryption system by DB sending a message encrypted using user's public key, and user decrypting and sending the message back
- **Digital signatures** are used to verify authenticity of data
  - E.g. use private key (in reverse) to encrypt data, and anyone can verify authenticity by using public key (in reverse) to decrypt data. Only holder of private key could have created the encrypted data.
  - Digital signatures also help ensure **nonrepudiation:** sender cannot later claim to have not created the data

Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and Cobol
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL
- The basic form of these languages follows that of the System R embedding of SQL into PL/I.
- **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 (for example, the Java embedding uses

  # SQL { .... };)

Example Query

- From within a host language, find the names and cities of customers with more than the variable amount dollars in some account
- Specify the query in SQL and declare a *cursor* for it

```
EXEC SQL
  declare c cursor for
    select depositor.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 > :amount
END_EXEC
```

Embedded SQL (Cont.)

- The **open** statement causes the query to be evaluated

  EXEC SQL **open** *c* END_EXEC

- 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

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

- 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: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.

Updates Through Cursors

- Can update tuples fetched by cursor by declaring that the cursor is for update

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

- To update tuple at the current location of cursor *c*

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

Dynamic SQL

- Allows programs to construct and submit SQL queries at run time
- Example of the use of 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 account [10] = "A-101";
EXEC SQL execute dynprog using :account;
```

- The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed

ODBC and JDBC

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
  - Connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables
- ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
- JDBC (Java Database Connectivity) works with Java

ODBC

- Open DataBase Connectivity(ODBC) standard
  - standard for application program to communicate with a database server
  - 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

ODBC (Cont.)

- Each database system supporting ODBC provides a "driver" library that must be linked with the client program
- When client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.
- ODBC program first allocates an SQL environment, then a database connection handle
- Opens database connection using SQLConnect(). Parameters for SQLConnect:
  - 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 Code

```
int ODBCexample()
{
    RETCODE error;
    HENV    env;     /* environment */
    HDBC    conn;  /* database connection */
    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, "aura.bell-labs.com", SQL_NTS, "avi",
     SQL_NTS, "avipasswd", SQL_NTS);
    { .... Do actual work ... }

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

ODBC Code (Cont.)

- 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 Code (Cont.)

- Main body of program

```
 char branchname[80];
float  balance;
int  lenOut1, lenOut2;
HSTMT   stmt;
  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,
0 , &lenOut2);
        while (SQLFetch(stmt) >= SQL_SUCCESS) {
          printf (" %s  %g\n", branchname, balance);
     }
}
SQLFreeStmt(stmt, SQL_DROP);
```

More ODBC Features

- **Prepared Statement**
  - SQL statement prepared: compiled at the database
  - Can have placeholders: E.g. insert into account values(?,?,?)
  - Repeatedly executed with actual values for the placeholders
- **Metadata features**
  - finding all the relations in the database and
  - finding the names and types of columns of a query result or a relation in the database.
- By default, each SQL statement is treated as a separate transaction that is committed automatically.
  - Can turn off automatic commit on a connection
    - SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0)}
  - transactions must then be committed or rolled back explicitly by
    - SQLTransact(conn, SQL_COMMIT) or
    - SQLTransact(conn, SQL_ROLLBACK)

ODBC Conformance Levels

- Conformance levels specify subsets of the functionality defined by the standard.
  - Core
  - 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

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

Procedural Extensions and Stored Procedures

- SQL provides a **module** language
  - Permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.
  - more in Chapter 9
- Stored Procedures
  - Can store procedures in the database
  - then execute them using the **call** statement
  - permit external applications to operate on the database without knowing about internal details
- These features are covered in Chapter 9 (Object Relational Databases)

Functions and Procedures

- SQL:1999 supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external programming language
  - Functions are particularly useful with specialized data types such as images and geometric objects
    - Example: functions to check if polygons overlap, or to compare images for similarity
  - Some database systems support **table-valued functions**, which can return a relation as a result
- SQL:1999 also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999

SQL Functions

- Define a function that, given the name of a customer, returns the count of the number of accounts owned by the customer.

```
create function account_count (customer_name
varchar(20))
    returns integer
    begin
        declare a_count integer;
        select count (* ) into a_count
      from depositor
      where depositor.customer_name = customer_name
      return a_count;
    end
```

- Find the name and address of each customer that has more than one account.

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

Table Functions

- SQL:2003 added functions that return a relation as a result
- Example: Return all accounts owned by a given customer

```
create function accounts_of (customer_name char(20)
        returns table (          account_number char(10),
                                 branch_name char(15)
                                 balance numeric(12,2))
return table
       (select account_number, branch_name, balance
        from account A
        where exists (
            select *
            from depositor D
            where D.customer_name =
accounts_of.customer_name
                        and D.account_number = A.account_number ))
```

Table Functions (cont'd)

- Usage

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

SQL Procedures

- The *author_count* function could instead be written as procedure:

```
create procedure 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
end
```

- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

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

  Procedures and functions can be invoked also from dynamic SQL

- SQL:1999 allows more than one function/procedure of the same name (called name **overloading**), as long as the number of arguments differ, or at least the types of the arguments differ

Procedural Constructs

- Compound statement: **begin** … **end**,
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statements
- **While** and **repeat** statements:

```
declare n integer default 0;
while n < 10 do
    set n = n + 1
end while

repeat
        set n = n - 1
until n = 0
end repeat
```

Procedural Constructs (Cont.)

- **For** loop
  - Permits iteration over all results of a query
  - Example: find total of all balances at the Perryridge branch

```
declare n integer default 0;
for r as
        select balance from account
      where branch_name = 'Perryridge'
do
        set n = n + r.balance
end for
```

Procedural Constructs (cont.)

- Conditional statements (**if-then-else**)
  E.g. To find sum of balances for each of three categories of accounts (with balance <1000, >=1000 and <5000, >= 5000)

```
if r.balance < 1000
    then set l = l + r.balance
elseif r.balance < 5000
    then set m = m + r.balance
else set h = h + r.balance
end if
```

- SQL:1999 also supports a **case** statement similar to C case statement
- Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_stock condition
declare exit handler for out_of_stock
begin
...
.. signal out-of-stock
end
```

  - The handler here is **exit** -- causes enclosing **begin..end** to be exited
  - Other actions possible on exception

External Language Functions/Procedures

- SQL:1999 permits the use of functions and procedures written in other languages such as C or C++
- Declaring external language procedures and functions

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

create function account_count(customer_name varchar(20))
returns integer
language C
external name '/usr/avi/bin/author_count'
```

External Language Routines (Cont.)

- Benefits of external language functions/procedures:
  - more efficient for many operations, and more expressive power
- Drawbacks
  - Code to implement function may need to be loaded into database system and executed in the database system's address space
    - risk of accidental corruption of database structures
    - security risk, allowing users access to unauthorized data
  - There are alternatives, which give good security at the cost of potentially worse performance
  - Direct execution in the database system's space is used when efficiency is more important than security

Security with External Language Routines

- To deal with security problems
  - Use **sandbox** techniques
    - that is use a safe language like Java, which cannot be used to access/damage other parts of the database code
  - Or, run external language functions/procedures in a separate process, with no access to the database process' memory
    - Parameters and results communicated via inter-process communication
- Both have performance overheads
- Many database systems support both above approaches as well as direct executing in database system address space

Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find all employee-manager pairs, where the employee reports to the manager directly or indirectly (that is manager's manager, manager's manager's manager, etc.)

```
with recursive empl (employee_name, manager_name ) as (
        select employee_name, manager_name
        from   manager
    union
        select manager.employee_name, empl.manager_name
        from   manager, empl
        where manager.manager_name = empl.employe_name)
select *
from   empl
```

This example view, *empl*, is called the *transitive closure* of the *manager* relation

The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *manager* with itself
    - This can give only a fixed number of levels of managers
    - Given a program we can construct a database with a greater number of levels of managers on which the program will not work
- Computing transitive closure
  - The next slide shows a *manager* relation
  - Each step of the iterative process constructs an extended version of *empl* from its recursive definition.
  - The final result is called the *fixed point* of the recursive view definition.
- Recursive views are required to be *monotonic*. That is, if we add tuples to *manger* the view contains all of the tuples it contained before, plus possibly more

Example of Fixed-Point Computation

employee_name	*manager_name*
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

Iteration number	*Tuples in empl*
0	
1	(Duarte), (Estovar)
2	(Duarte), (Estovar), (Barinsky), (Corbin)
3	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)
4	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)

Advanced SQL Features**

- Create a table with the same schema as an existing table:

  create table *temp_account* **like** *account*

- 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:

```
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 (cont'd)

- 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

```
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
```

The End of Lecture 5