Advanced SQL (Lecture 3)

- Index
- Transaction

- Integrity
  - Constraints
  - Referential Integrity
  - Assertions
- Trigger
- Functions and Procedures

- Authorization

Index Creation

- Many queries reference only a small proportion of the records in a table.
- It is inefficient for the system to read every record to find a record with particular value
- An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.

- We create an index with the **create index** command

    create index <name> on <relation-name> (attribute);

Index Creation Examle

- **create table** *student*
  (*ID* **varchar** (5),
  name **varchar** (20) **not null**,
  dept_name **varchar** (20),
  tot_cred **numeric** (3,0) **default** 0,
  primary key (*ID*))

  create index *studentName_Idx* **on** *student(name)*

- Indices are data structures used to speed up access to records with specified values for index attributes
  - e.g. **select ***
    from *student*
    where *ID* = ‘12345’

  can be executed by using the index to find the required record, without looking at all records of *student*

Transactions

- A **transaction** consists of a sequence of query and/or update statements.
- Atomic transaction: either fully executed or rolled back as if it never occurred.
  - Commit work. The updates performed by the transaction become permanent in the database.
  - Rollback work. All the updates performed by the SQL statements in the transaction are undone.
- Isolation from concurrent transactions
- Transactions begin implicitly, by default on most databases: each SQL statement commits automatically
  - Can turn off auto commit for a session (e.g. using API)
  - In SQL:1999, can use: **begin atomic** …. **end**

Integrity

- 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 $12.00 an hour
  - A customer must have a (non-null) phone number

Constrains

- Constraints on a Single Relation
  - Primary Key, Unique, Not Null;
  - CHECK - User defined constraints on the table
  - Domain Constraints
  - Referential Integrity

- Constrains on entire database (a set of relations)
  - Assertions --Constrains to check the consistency of database
  - Triggers --Active rules to maintain the consistency of database

Not Null and Unique Constraints

- **not null**
  - Declare *name* and *budget* to be **not null**

    name **varchar**(20) **not null**
    budget **numeric**(12,2) **not null**

- **unique** ($A_1, A_2, \ldots, A_m$)
  - The unique specification states that the attributes *A*1, *A*2, … *A*m form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

CHECK

- **check** (*P*), where *P* is a predicate

  Example: ensure that semester is one of fall, winter, spring or summer:

```
create table section (
    course_id varchar (8),
    sec_id varchar (8),
    semester varchar (6),
    year numeric (4,0),
    building varchar (15),
    room_number varchar (7),
    time slot id varchar (4),
    primary key (course_id, sec_id, semester, year),
    check (semester in ('Fall', 'Winter', 'Spring', 'Summer'))
);
```

- *P* can be any complex conditions as in WHERE clause. However, most of DBMS does not support subquery in CHECK statement.

Domains

- **create domain** construct in SQL-92 creates user-defined domain types from existing data types

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

  create domain *Dollars* **numeric**(12, 2)
  create domain *Pounds* **numeric**(12,2)

- Domains can have constraints, such as **not null**, and more complex constrains:

  create domain *degree_level* **varchar**(10)
  constraint *degree_level_test*
  check (**value in** ('Bachelors', 'Masters', 'Doctorate'));

- Domains are not strongly typed. Values of one domain type can be assigned to values of another domain type as long as the underlying types are compatible.
  - We CAN assign or compare a value of type Dollars to a value of type Pounds.

Domain Constraints

- The **check** clause in SQL-92 permits domains to be restricted:
  - 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* > = 12.00)
  - The domain has a constraint that ensures that the hourly-wage is greater than 12.00
  - The clause **constraint** *value-test* is optional; useful to indicate which constraint an update violated.
- Can have complex conditions in domain check
  - **create domain** *dept_name_D* **varchar**(20)
        constraint *dept_name__test*
          check (*value* **in** (**select** *dept-name* **from** *department*))

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.
  - Example: If "Biology" is a department name appearing in one of the tuples in the instructor relation, then there exists a tuple in the department relation for "Biology".
- Formal Definition
  - $r_1(R_1)$ and $r_2(R_2)$ are two relations, Let $r_1(R_1)$ with primary keys K_1.
  - 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

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

Checking Referential Integrity on Database Modification

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

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

- **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:
    - Let t_2' denote the new value of tuple t_2. The system must ensure that

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

  - 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 (the value before the update is applied).
    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 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.
- Define Foreign Key
  - By default, a foreign key references the primary key attributes of the referenced table

    foreign key (*dept_name*) **references** *department*
  - Short form for specifying a single column as foreign key

    dept_name **varchar** (20) **references** *department*
  - Reference columns in the referenced table can be explicitly specified, but must be declared as primary/candidate keys

    foreign key (*dept_name*) **references** *department*(*dept_name*)

Referential Integrity in SQL – Example

- **create table** *department*(
  dept_name **varchar**(20) **primary key** ,
  building **varchar**(15),
  budget numeric(12,0));

- **create table** *course* (
  course_id **varchar**(8),
  title **varchar(**50),
  dept_name **varchar**(20),
  credits **numeric**(2,0),
  primary key *(course_id),*
  foreign key *(dept_name*) **references** *department*);

- **create table** *takes* (
  ID **varchar**(5) **references** *student*,
  course_id **varchar**(8),
  sec_id **varchar**(8),
  semester **varchar**(6),
  year **numeric**(4,0),
  grade **varchar**(2),
  primary key *(ID, course_id, sec_id, semester, year)* ,
  foreign key (*course_id, sec_id, semester, year*) **references** *section*);

Cascading Actions in SQL

create table *course*

    . . .

    foreign key(*dept_name*)   **references** *department*
on delete cascade
on update cascade

    . . . **)**

- Due to the **on delete cascade** clauses, if a delete of a tuple in *department* results in referential-integrity constraint violation, the delete "cascades" to the *course* relation, deleting the tuple that refers to the *department* that was deleted.
- Cascading updates are similar.

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!

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.
- If a cascading update to 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.
- 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)

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.
- Asserting

  for all *X*, *P*(*X*)

  is achieved in a round-about fashion using

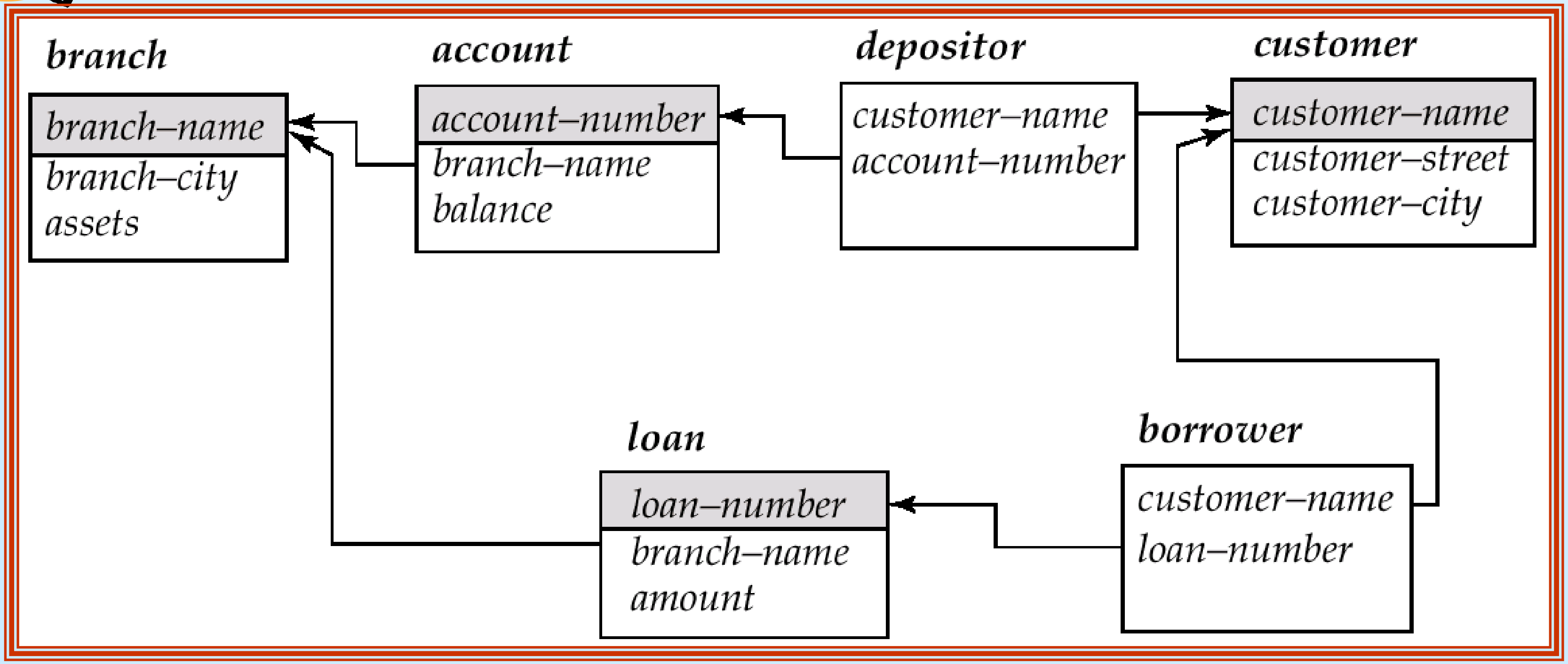
  not exists *X* such that not *P*(*X*)

Assertion Example

- For each tuple in the *student* relation, the value of the attribute *tot_cred* must equal the sum of credits of course that the student has comleted successfully.

```
create assertion credits_earned_constraint check
  (not exists (
       select *
        from student  S
        where tot_cred <> (
             select  sum( credits)
             from takes nature join course
             where takes.ID = S.ID
                  and grade is not null and
             grade <> 'F'
       )
  )
```

Schema Diagram for the Banking Enterprise



- **Foreign Key**: **The attributes of a relation schema r1 is the primary key of another relation schema r2. The attributes is called a foreign key from r1, referencing r2.**
  - The attribute branch-name in Account is a foreign key referencing Branch.
  - Only values occurring in the primary key attribute of the **Referenced relation** may occur in the foreign key attribute of the **Referencing relation**

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 B
                   where (select sum(amount )
                             from loan
                               where loan.branch_name =
                                       B.branch_name )
                        >= (select sum (amount )
                             from account
                               where loan.branch_name =
                                       B.branch_name )
                   )
        )
```


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.
  - Syntax illustrated here may not work exactly on your database system; check the system manuals

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. after update of** *grade* **on** *takes*
- 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. E.g. convert blanks grade to null.

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

Trigger to Maintain credits_earned value

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

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 of each item
  - *reorder* (*item, amount*): What quantity should we re-order at a time
  - *orders* (*item, amount*) : Orders to be placed (read by external process)

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
```

Another program will keep watch on table *orders*

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 (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
- 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
- Triggers should be written with great care : cascading triggers

Functions and Procedures

- Functions and procedures allow "business logic" to be stored in the database and executed from SQL statements.
- These can be defined either by the procedural component of SQL or by an external programming language such as Java, C, or C++.
- The syntax we present here is defined by the SQL standard.
  - Most databases implement nonstandard versions of this syntax.

Declaring SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

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

- The function *dept*_count can be used to find the department names and budget of all departments with more that 12 instructors.

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

Table Functions

- The SQL standard supports functions that can return tables as results; such functions are called **table functions**
- Example: Return all instructors in a given department

create function *instructor_of* (*dept_name* **char**(20))
    returns table (
        ID **varchar**(5),
        name **varchar**(20),
        dept_name **varchar**(20),
        salary **numeric**(8,2))

    return table
        (**select** *ID, name, dept_name, salary*
        from *instructor*
        where *instructor.dept_name = instructor_of.dept_name*)

- Usage

    select *
    from table (*instructor_of* ('Music'))

External Language Routines

- SQL allows us to define functions in a programming language such as Java, C#, C or C++.
  - Can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL\can be executed by these functions.
- Declaring external language procedures and functions

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

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

Security with External Language Routines

- To deal with security problems, we can do on of the following:
  - Use **sandbox** techniques
    - That is, use a safe language like Java, which cannot be used to access/damage other parts of the database code.
  - 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.

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

Forms of authorization to modify the database schema

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

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 (or be the database administrator).

Privileges in SQL

- **select:** allows read/query access to relation, or the ability to query using the view
  - Example: grant users U_1, U_2, and U_3 **select** authorization on the *branch* relation:

    grant select on *instructor* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **delete**: the ability to delete tuples.
- **update**: the ability to update using the SQL update statement
  - The attributes on which update authorization is to be granted can be listed.
  - Example: grant user U_1, U_2, and U_3 **update** authorization on the *amount* attribute of the *loan* relation.

    grant update(*budget***) on** *department* **to** U_1, U_2, U_3
- **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

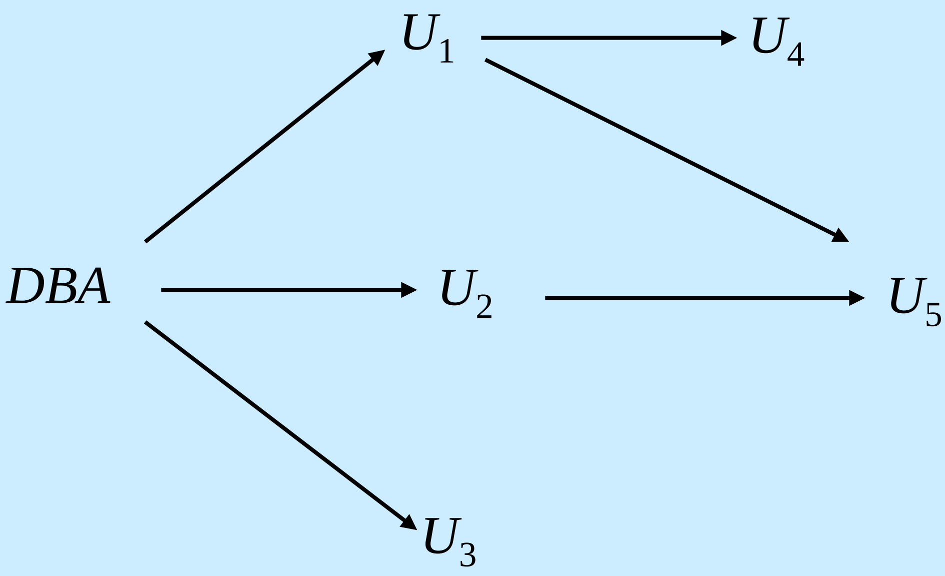
- **with grant option**: allows a user who is granted a privilege to pass the privilege on to other users.
  - Example:

    grant select on *instructor* **to** U_1 **with grant option**

    gives U_1 the **select** privileges on instructor and allows U_1 to grant this privilege to others

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 department.
- An edge $U_i \rightarrow U_j$ indicates that user U_i has granted update authorization on department 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
  - 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.

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.

- <privilege-list> may be **all** to revoke all privileges the revokee may hold

Roles

- 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 *instructor*
create role *dean*

grant select on *takes* **to** *instructor*
grant all privileges on *sections* **to** *instructor*

grant *instructor* **to** *dean*

grant update (*salary***) on** *instructor* **to** *dean*

grant *instructor* **to** *alice, bob*
grant *dean* **to** *avi*

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 to precisely the data that user needs.

View Example

- Suppose the staff in the university need to know all the information but not the salary of other staffs.
  - Approach: Deny direct access to the *instructor* relation, but grant access to the view *faculty*, which consists all the attributes exclude *salary* of *instructor*.

    create view *faculty* **as**
    select *ID, name, dept_name*
    from *instructor*

  - Defined the authorization in SQL as follows:

    grant select on *faculty* **to** *instructor*

    grant all privileges on *instructor* **to** *dean*

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*
- Authorization will be checked before query processing replaces a view by the definition of the view.

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
- Some database system provide mechanism for fine-gained authorization at the level of specific tuples within a relation.
  - Oracle Virtual Private Database feature
  - PostgreSQL and SQL Service have similar mechanism

Limitations of SQL Authorization

- 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
  - 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 loop holes becomes very difficult since it requires reading large amounts of application code

End of Lecture