Lecture 4

Intermediate SQL

Outline

- Join Expressions
- Views
- Transactions
- SQL Data Types and Schemas
- Integrity Constraints
- Authorization

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- A join operation is a **Cartesian product** which requires that **tuples in the two relations match** (under some condition).
  - It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the **from** clause

Given relations: r(A1, A2, A3) s(A1, A4, A5)

select *r*.A1, A4 **from** *r* **natural join** *s*

Join operations – Example

- Relation *course*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- Relation *prereq*

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe that

  prereq information is missing for CS-315 and

  course information is missing for CS-347

Outer Join

- An extension of the join operation that **avoids loss of information**.
- Computes the join and then **adds tuples** form one relation **that does not match** tuples in the other relation to the result of the join.
- Uses *null* values.

Left Outer Join

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

course **natural left outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

Right Outer Join

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

course **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

Full Outer Join

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

course **natural full outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using $(A_1, A_2, \ldots, A_n)$

Joined Relations – Examples

- *course* **inner join** *prereq* **on** *course.course_id = prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prereq_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

course **left outer join** *prereq* **on** *course.course_id = prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prereq_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-315	Robotics	Comp. Sci.	3	*null*	*null*

Joined Relations – Examples

course **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

course **full outer join** *prereq* **using** (*course_id*)

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Joined Relations - Examples

- List all departments along with the number of instructors in each department

select department.dept_name, count(**distinct** instructor.id)

from department **left outer join** instructor **on** department.dept_name=instructor.dept_name

group by department.dept_name;

Oracle：

select department.dept_name, count(**distinct** instructor.id)
from department, instructor

where department.dept_name = instructor.dept_name**(+)**

group by department.dept_name;

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  select *ID*, *name*, *dept_name*
  from *instructor*

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.

View Definition

- A view is defined using the **create view** statement which has the form

  create view ***v*** as < query expression >

  where <query expression> is any legal SQL expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition **is not** the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the **saving of an expression**; the expression is *substituted* into queries using the view.

Example Views

- A view of instructors without their salary

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

- Find the names of all instructors in the Biology department

  select *name*
  from *faculty*
  where *dept_name* = ‘Biology’

- Create a view of department salary totals

  create view *departments_total_salary*(*dept_name*, *total_salary*) **as**
  select *dept_name*, **sum** (*salary*)
  from *instructor*
  group by *dept_name*;

Views Defined Using Other Views

- One view may be used in the expression defining another view
  - A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1
  - A view relation v_1 is said to *depend on* view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
  - A view relation v is said to be *recursive* if it depends on itself.

Views Defined Using Other Views

- **create view** *physics_fall_2009* **as**
  select *course*.*course_id*, *sec_id*, *building*, *room_number*
  from *course*, *section*
  where *course*.*course_id* = *section*.*course_id*
  and *course*.*dept_name* = 'Physics'
  and *section*.*semester* = 'Fall'
  and *section*.*year* = '2009';

- **create view** *physics_fall_2009_watson* **as**
  select *course_id*, *room_number*
  from *physics_fall_2009*
  where *building*= 'Watson';

View Expansion

- A way to define the meaning of views defined in terms of other views.
  - View expansion of an expression repeats the following replacement step:

    repeat
    Find any view relation v_i in e_1
    Replace the view relation v_i by the expression defining v_i
    until no more view relations are present in e_1

  - As long as the view definitions are not recursive, this loop will terminate

View Expansion

- Expand use of a view in a query/another view

create view *physics_fall_2009_watson* **as**
(**select** *course_id*, *room_number*
from (**select** *course.course_id*, *sec_id*, *building*, *room_number*
    from *course*, *section*
    where *course.course_id* = *section.course_id*
        and *course.dept_name* = 'Physics'
        and *section.semester* = 'Fall'
        and *section.year* = '2009')
where *building*= 'Watson';

physics_fall_2009

Update of a View

- Create a view of instructors without their salary

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

- Add a new tuple to the *faculty* view

  insert into *faculty* **values** ('30765', 'Green', 'Music');

This insertion must be represented by the insertion of the tuple

('30765', 'Green', 'Music', null)

into the *instructor* relation.

Some Updates cannot be Translated Uniquely

- **create view** *instructor_info* **as**
  select *ID*, *name*, *building*
  from *instructor*, *department*
  where *instructor*.*dept_name*= *department*.*dept_name*;
- **insert into** *instructor_info* **values** ('69987', 'White', 'Taylor');
  - which department, if multiple departments in Taylor?
  - what if no department is in Taylor?
- Most SQL implementations allow updates only on simple views
  – The **from** clause has only one database relation.
  – The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
  – Any attribute not listed in the **select** clause can be set to null
  – The query does not have a **group** by or **having** clause.

And Some Not at All

- **create view** *history_instructors* **as**
  select *
  from *instructor*
  where *dept_name*= ’History’;
- What happens if we insert (’25566’, ’Brown’, ’Biology’, 100000) into *history_instructors?*

Materialized Views **

- **Materializing a view**: create a physical table containing all the tuples in the result of the query defining the view
- If relations used in the query are updated, the materialized view result becomes out of date
  - Need to **maintain** the view, by updating the view whenever the underlying relations are updated.

Transactions

- Unit of work
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions
- Transactions begin implicitly
  - Ended by **commit work** or **rollback work**
- But default on most databases: each SQL statement commits automatically
  - Can turn off auto commit for a session (e.g., using an API)
  - In SQL:1999, can use: **begin atomic** …. **end**

Built-in Data Types

- Basic Data Types
  - char(n), varchar(n)
  - int , smallint, numeric(p,d)
  - real, double precision, float(n)
- Addition Built-in Data Types
  - date: **date ‘2006-09-04’**
  - time: **time ’08:55:00’**
  - timestamp: **timestamp ‘2006-09-04 11:25:09.77’**
  - interval: Indicates a period of time, usu. implemented as a fixed point number
    - 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

- Operating with date/time/timestamp values
  - Extract values of individual fields

    extract (year from r.starttime)
  - Type coercion

    cast '2006-12-25' as date
- Implementation may vary in real systems
  - Oracle supports addition/subtraction between datetime types and numeric values

User-Defined Types

- Creating types or domains
  - Examples:
    - **create type** Dollars **as** numeric(12,2) final
    - **create domain** person_name **as** char(20) not null
    - Then we can specify attributes with types and domains (if they are supported by DBMS)
  - Domains can have constraints specified on them while types cannot
  - Domain constraints are the most elementary form of integrity constraint

User-Defined Types

- Domain Constraints
  - New domains can be created from existing data types
  - Domains are not strongly typed, while types are.
    - Example

      create type USD numeric(12, 2)

      create type CNY numeric(12, 2)

      We cannot assign or compare a value of type USD to a value of type CNY.
    - Type coercion is applicable.

      (**cast** r.A **as** CNY)

Large-Object Types

- Large objects (photos, videos, etc.) are stored as a *large object*
  - **blob**: binary large object – stored as uninterpreted binary data
  - **clob**: character large object – stored as characters
  - When a query returns a large object, a *pointer* is returned rather than the large object itself.
    - Returned data should be handled by an application **outside of** the database system

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

Integrity Constraints

- Constraints on a Single Relation
  - **not null**
  - **primary key**
  - **unique**
  - **check(***P***)**
- Referential Integrity
  - **foreign key**
- Assertion across relations

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 $A_1, A_2, \ldots A_m$ form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

The **check** clause

- **check** (P)

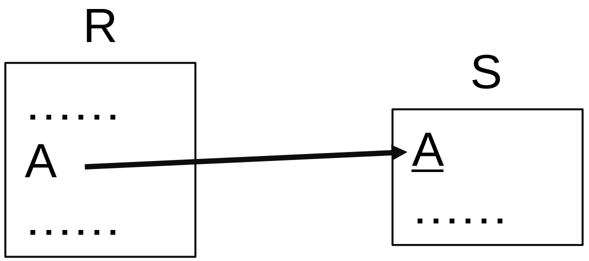
  where P is a predicate

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'))
);

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" department
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.



Cascading Actions in Referential Integrity

- **create table** *course* (

    …

    dept_name **varchar**(20),

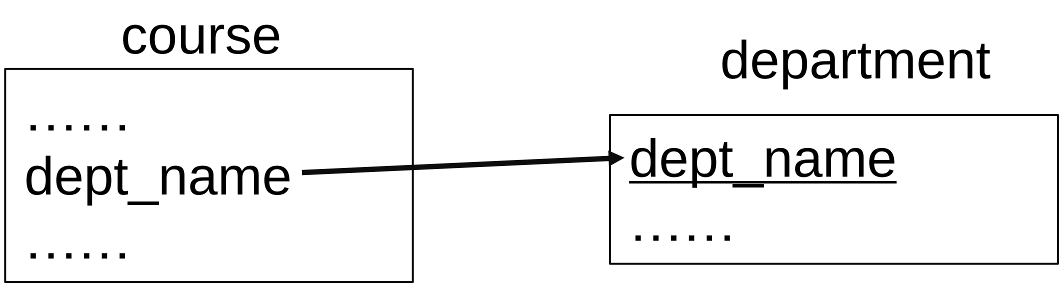
    foreign key (*dept_name*) **references** *department*

    on delete cascade

    on update cascade,

    . . .

)



- alternative actions to **cascade**: **set null**, **set default**
  - e.g., on delete set null

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>

- The system tests an assertion
  - when an assertion is made
  - when an update may violate the assertion
  - that may introduce a significant amount of overhead
    - USE with GREAT CARE!

Some implementation of DBMS does not support assertion.
Refer to the user’s manual for instructions.

Authorization

- Authorizations to …
  - read data
  - insert new data
  - update data
  - delete data
- Privileges
  - select
  - insert
  - update
  - delete

Authorization Specification in SQL

- The **grant** statement

  grant <privilege list>

  on <relation name or view name> **to** <user / role list>

  – several privileges can be granted in one command

  – <privilege list> may be **all privileges**, indicating all allowable privileges to be granted

  – <user / role list> may be **public**, allowing all current and future users the privilege granted *implicitly*

  – 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).

Revoking Authorization in SQL

- The **revoke** statement

  revoke <privilege list>

  on <relation name or view name> **from** <user list>

  - <privilege list> may be all to revoke all privileges revokee may hold
  - <revokee-list> may include **public**, meaning 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

Privileges in SQL

- **select:** allows read 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 *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

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

- 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
  - **create role** instructor;
  - **grant** *instructor* **to** Amit**;**
  - **grant select on** *takes* **to** *instructor*;
  - **create role** *teaching_assistant*
  - **grant** *teaching_assistant* **to** *instructor*;
    - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles

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 precisely limit a user's access to the data that user needs.

Authorization on Views

- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = ’Geology’);
- **grant select on** *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  - **select** *
    from *geo_instructor*;
- What if
  - *geo_staff* does not have permissions on *instructor?*
  - creator of view did not have some permissions on *instructor?*

Authorization on Views

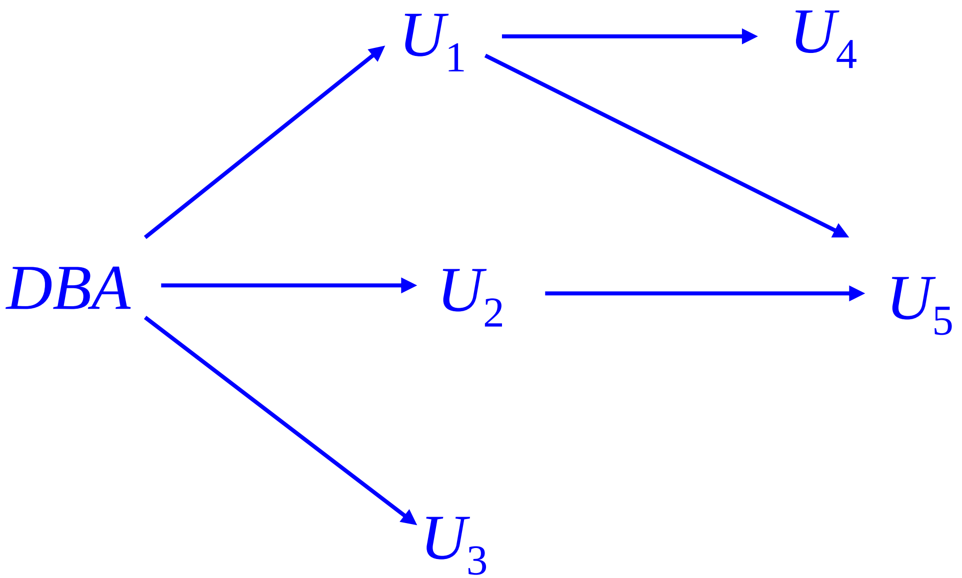
- Creation of view does not require **resources** authorization since no real relation is being created
- The creator of a view gets only those privileges that provide no additional authorization beyond that he already had.
  - e.g. if creator of view *geo_instructor* had only **read** authorization on *instructor*, he gets only **read** authorization on *geo_instructor*

Other Authorization Features

- **references** privilege to create foreign key
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;
  - why is this required?
- transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;

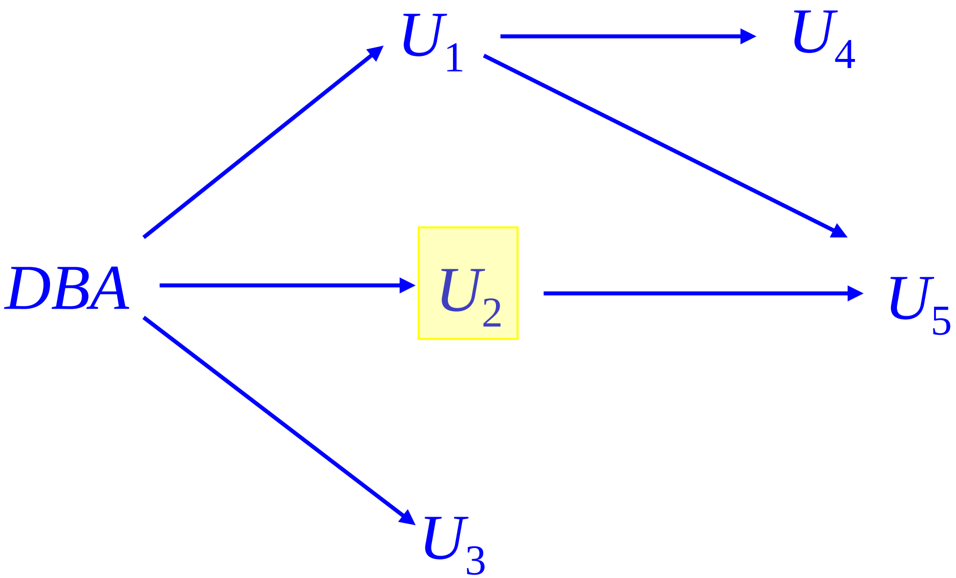
Granting of Privileges

- 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.
- Graph below
  - An edge $U_i \rightarrow U_j$ indicates that user U_i has granted update authorization on loan to U_j.

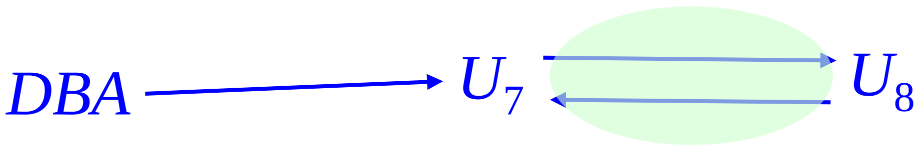


Authorization Grant Graph

- *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.



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
- 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

- 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

Next Lecture

- Advanced SQL

End of Lecture 4