

# **Chapter 4: Intermediate SQL**

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



- **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



# Join operations – Example

## ■ Relation *course*

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

## ■ Relation *prereq*

<i>course_id</i>	<i>prereq_id</i>
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 from one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.



# Left Outer Join

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

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



# Right Outer Join

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

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



- **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  $(A_1, A_1, \dots, A_n)$





# Full Outer Join

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

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



# Joined Relations – Examples

- **course inner join prereq on**  
*course.course\_id = prereq.course\_id*

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>	<i>course_id</i>
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- What is the difference between the above, and a natural join?
- **course left outer join prereq on**  
*course.course\_id = prereq.course\_id*

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>	<i>course_id</i>
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	<i>null</i>	<i>null</i>



# Joined Relations – Examples

## ■ *course* natural right outer join *prereq*

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

## ■ *course* full outer join *prereq* using (*course\_id*)

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



- 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**.



- 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.



- A view of instructors without their salary

**create view** *faculty* **as**

**select** *ID, name, dept\_name*

**from** *instructor*

- Find 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 *DataBase System Concepts*

---

- **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';*



- 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, 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';
```





- 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.



- A way to define the meaning of views defined in terms of other views.
- Let view  $v_1$  be defined by an expression  $e_1$  that may itself contain uses of view relations.
- 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



- Add a new tuple to *faculty* view which we defined earlier  
**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




- **create view** *history\_instructors* **as**  
**select** \*  
**from** *instructor*  
**where** *dept\_name*= 'History';
- What happens if we insert ('25566', 'Brown', 'Biology', 100000) into *history\_instructors*?



- **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.



- 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 API)
  - In SQL:1999, can use: **begin atomic .... en** ' 
  - ▶ Not supported on most databases

- 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 on a Single Relation*

---

- - **not null**
  - **primary key**
  - **unique**
  - **check (P)**, where P is a predicate
- 
- 
- 
- 
- 
- 



## ■ not null

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

*name* **varchar(20) not null**

*budget* **numeric(12,2) not null**

## ■ unique ( $A_1, A_2, \dots, A_m$ )

- The unique specification states that the attributes  $A_1, A_2, \dots, A_m$  form a candidate key.
- Candidate keys are permitted to be null (in contrast to primary keys).



- **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'))  
);
```

- 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”.
- 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* (  
    *course\_id* **char**(5) **primary key**,  
    *title* **varchar**(20),  
    *dept\_name* **varchar**(20) **references** *department*  
)
- **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**

# Integrity Constraint Violation During Transactions

■ E.g.

```
create table person (  
    ID char(10),  
    name char(40),  
    mother char(10),  
    father char(10),  
    primary key ID,  
    foreign key father references person,  
    foreign key mother references person)
```

■ How to insert a tuple without causing constraint violation ?

- insert father and mother of a person before inserting person
- OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
- OR defer constraint checking (next slide)



- **check** (*time\_slot\_id* in (**select** *time\_slot\_id* from *time\_slot*))
  - why not use a foreign key here?
- Every section has at least one instructor teaching the section.
  - how to write this?
- Unfortunately: subquery in check clause not supported by pretty much any database
  - Alternative: triggers (later)
- **create assertion** <assertion-name> **check** <predicate>;
  - Also not supported by anyone



- **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/time values





- **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** *studentID\_index* **on** *student*(*ID*)
  - 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*
- More on indices in Chapter 11*



- **create type** construct in SQL creates user-defined type

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

- **create table *department***  
**(*dept\_name* varchar (20),**  
***building* varchar (15),**  
***budget Dollars*);**



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

**create domain** *person\_name* **char(20) not null**

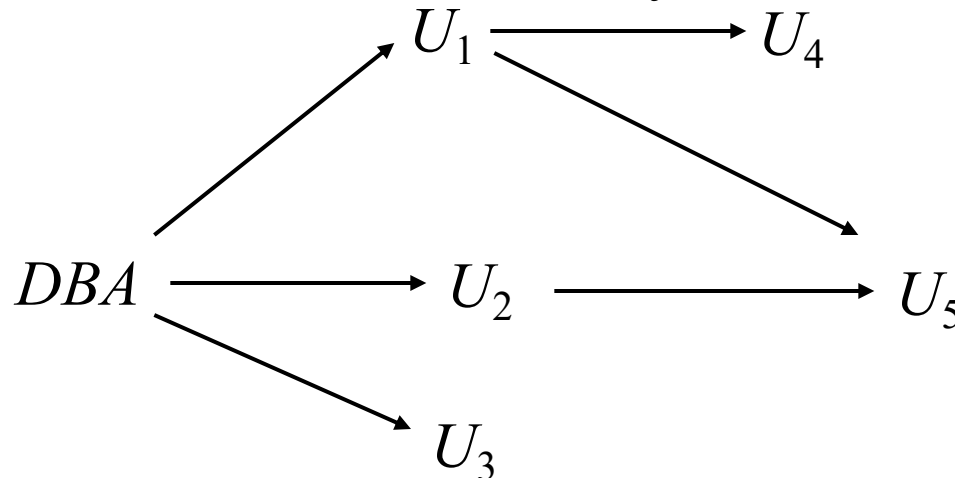
- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- **create domain** *degree\_level* **varchar(10)**  
**constraint** *degree\_level\_test*  
**check (value in ('Bachelors', 'Masters', 'Doctorate'));**



- 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.

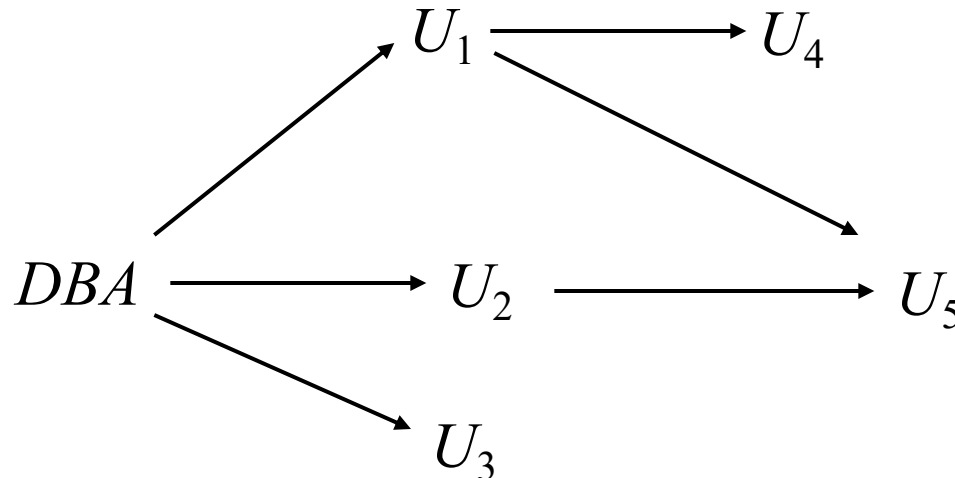


- The passage of authorization from one user to another may be represented by an authorization graph.
- The nodes of this graph are the users.
- The root of the graph is the database administrator.
- Consider graph for update authorization on loan.
- An edge  $U_i \rightarrow U_j$  indicates that user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



# Authorization Grant Graph

- *Requirement:* All edges in an authorization graph must be part of some path originating with the database administrator
- If DBA revokes grant from  $U_1$ :
  - Grant must be revoked from  $U_4$  since  $U_1$  no longer has authorization
  - Grant must not be revoked from  $U_5$  since  $U_5$  has another authorization path from DBA through  $U_2$



Forms of authorization on parts of the database:

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

Forms of authorization to modify the database schema

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



# Authorization Specification in SQL

## DataBase System Concepts

- 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).





- **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 *instructor* relation:

**grant select on *instructor* 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.
- **all privileges**: used as a short form for all the allowable privileges



# ● *Revoking Authorization in SQL*

- ■ The **revoke** statement is used to revoke authorization.  
**revoke** <privilege list>  
**on** <relation name or view name> **from** <user list>  
**revoke select on branch from**  $U_1, U_2, U_3$
- ■ <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.



- **create role** instructor;
- **grant** *instructor* **to** Amit;
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
  - **create role** *teaching\_assistant*
  - **grant** *teaching\_assistant* **to** *instructor*;
    - ▶ *Instructor* inherits all privileges of *teaching\_assistant*
- Chain of roles
  - **create role** *dean*;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;



- **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*?



- **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**;
- Etc. read Section 4.6 for more details we have omitted here.



- 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.



- Triggering event can be **insert**, **delete** or **update**;
- Triggers can be activated **before/after** an event, which can serve as extra constraints.

**create trigger *setnull-trigger* before/after update  
on *r***

**referencing new row as *nrow*  
for each row**

**when *nrow.phone-number* = ' '  
set *nrow.phone-number* = null**



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





# 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;**



