

Database Systems

Lecture05 – Intermediate SQL (Ch.4)

Beomseok Nam (남범석)
bnam@skku.edu

Outline

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

Joined Relations

▪ Join operations

- take two relations
- combine rows from two tables, using related columns
- return another relation

▪ Three types of joins:

- Natural join
- Inner join
- Outer join

Natural Join in SQL

공통특성을 찾는다

- Natural join

- matches tuples using all common attributes
- retains only one copy of each common column.

- E.g. List the names of instructors along with the course ID of the courses that they teaches

Instructors

ID	name	dept_name
10	Nam	Comp. Sci.
20	Alice	Electrical Eng.

Teaches

ID	c_id	semester	year
10	swe3003	Spring	2025
20	swe3003	Fall	2024
10	swe3021	Spring	2024

```
select name, course_id  
from instructors natural join teaches;
```

Natural Join in SQL

Instructors

ID	name	dept_name
10	Alice	Comp. Sci.
20	Bob	Electrical Eng.

Teaches

ID	c_id	semester	year
10	swe3003	Spring	2025
20	swe3003	Fall	2024
10	swe3021	Spring	2024

ID	name	dept_name	c_id	semester	year
10	Alice	Comp. Sci.	swe3003	Spring	2025
10	Alice	Comp. Sci.	swe3021	Spring	2024
20	Bob	Electrical Eng.	swe3003	Fall	2024

*select name, course_id
from instructors natural join teaches;*

Natural Join in SQL

Instructors

ID	name	dept_name
10	Alice	Comp. Sci.
20	Bob	Electrical Eng.

Teaches

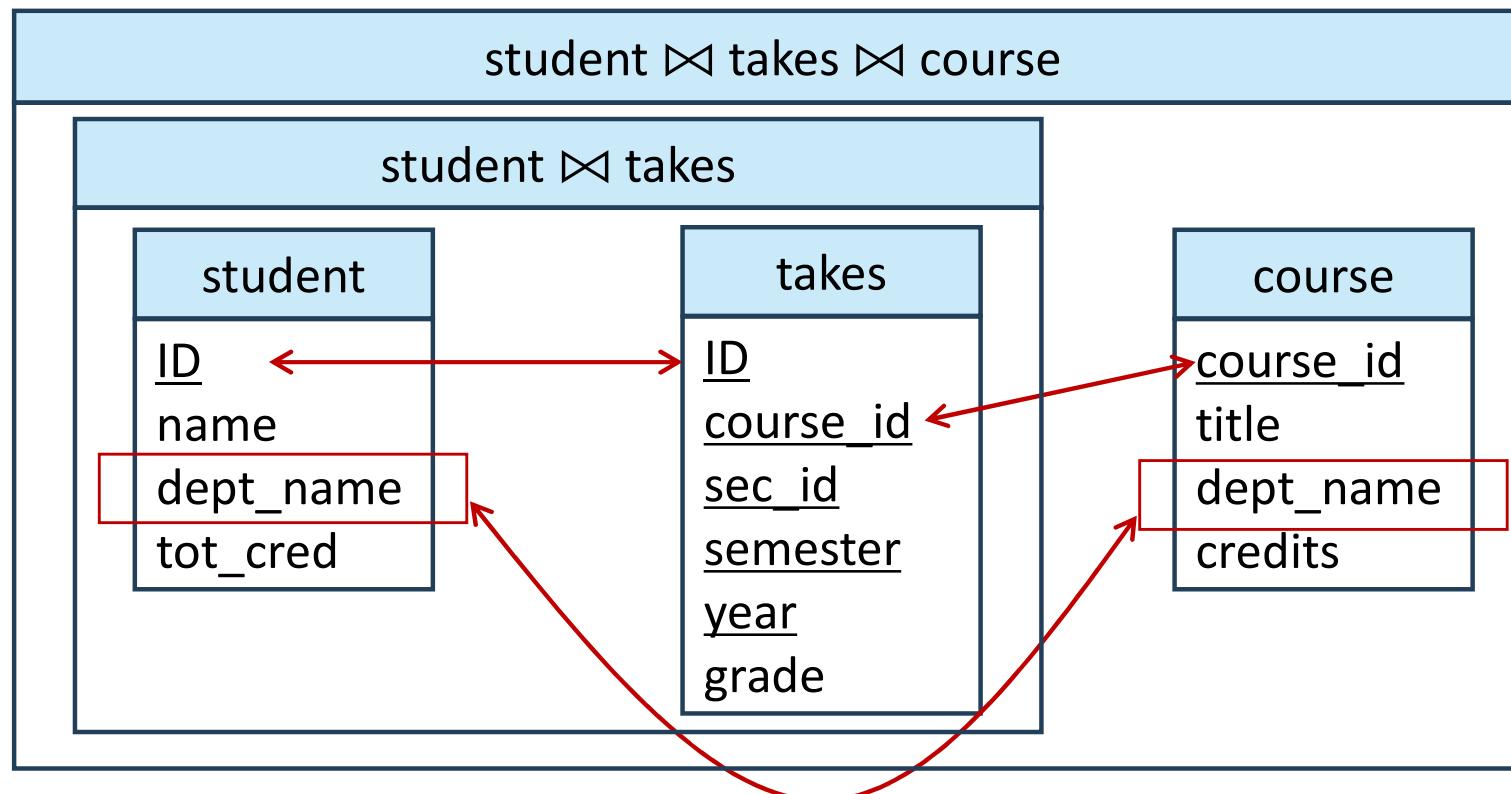
ID	c_id	semester	year
10	swe3003	Spring	2025
20	swe3003	Fall	2024
10	swe3021	Spring	2024

ID	name	dept_name	ID	c_id	semester	year
10	Alice	Comp. Sci.	10	swe3003	Spring	2025
10	Alice	Comp. Sci.	20	swe3003	Fall	2024
10	Alice	Comp. Sci.	10	swe3021	Spring	2024
20	Bob	Electrical Eng.	10	swe3003	Spring	2025
20	Bob	Electrical Eng.	20	swe3003	Fall	2024
20	Bob	Electrical Eng.	10	swe3021	Spring	2024

**select name, course_id
from instructors, teaches
where instructors.ID = teaches.ID;**

Recall: Join with *unrelated attributes with same name*

- if two columns **accidentally** happen to have the **same name**, **NATURAL JOIN** will use the column as the join condition,



- Example:

```
select name, title
```

```
from student natural join takes natural join course;
```

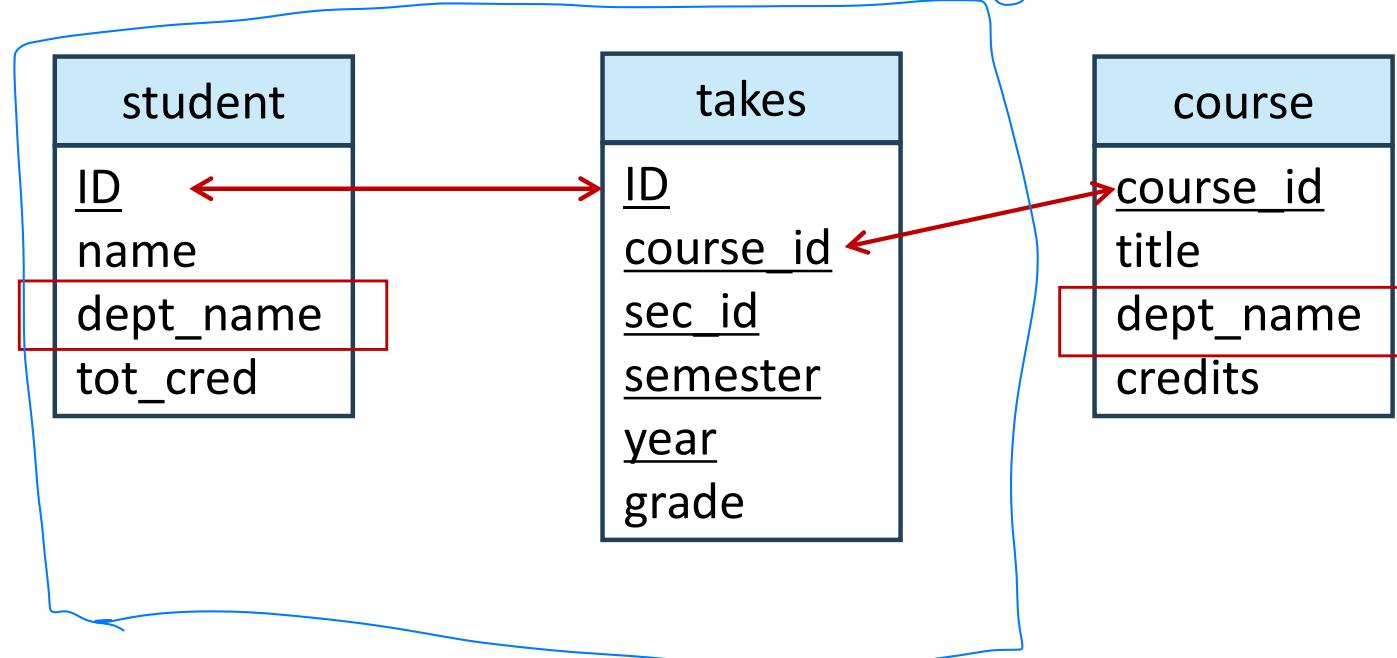
Natural Join with Using

- Join condition can be explicitly specified by the “**using**” construct.
 - To avoid the problem of *unrelated attributes with same name*

- Query example

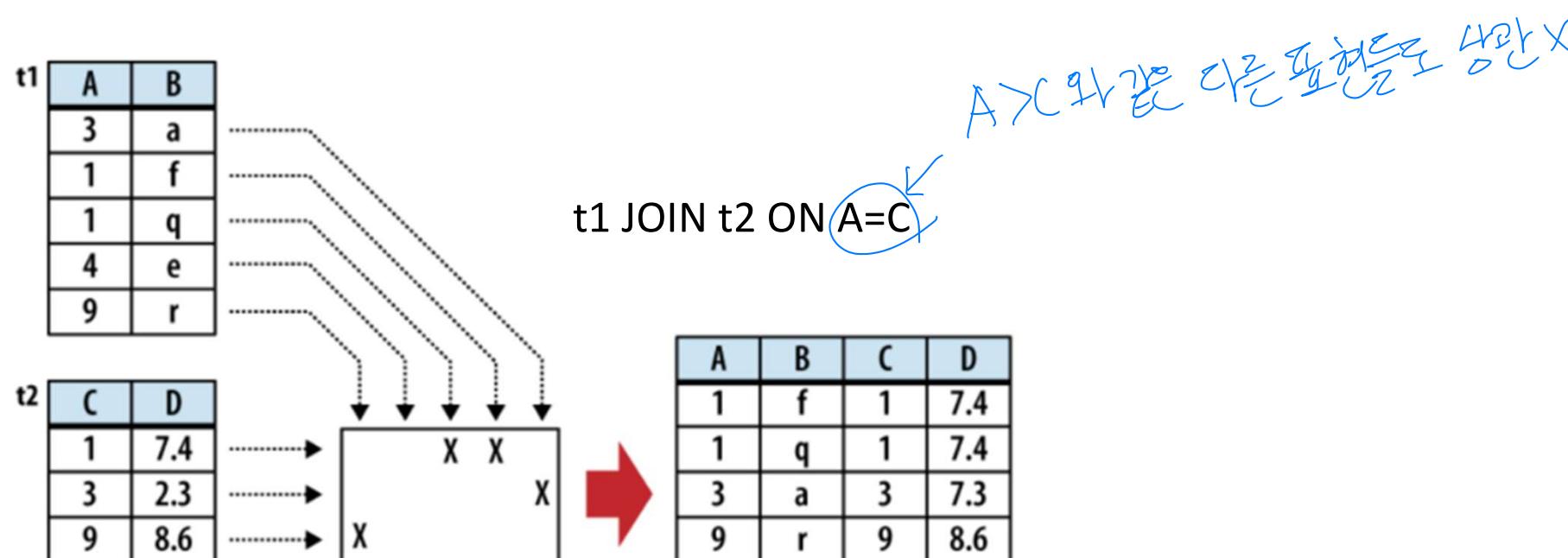
```
select name, title
```

```
from (student natural join takes) join course using (course_id)
```



Join with On

- “on” specifies a predicate as a join condition
 - It allows join using columns that have different names



Join with On

- Query example

```
select *  
from student join project on ID = student_ID
```

- Equivalent to:

```
select *  
from student, project where ID = student_ID
```

student

ID	name
1	Alice
2	Bob
3	Charlie
4	David

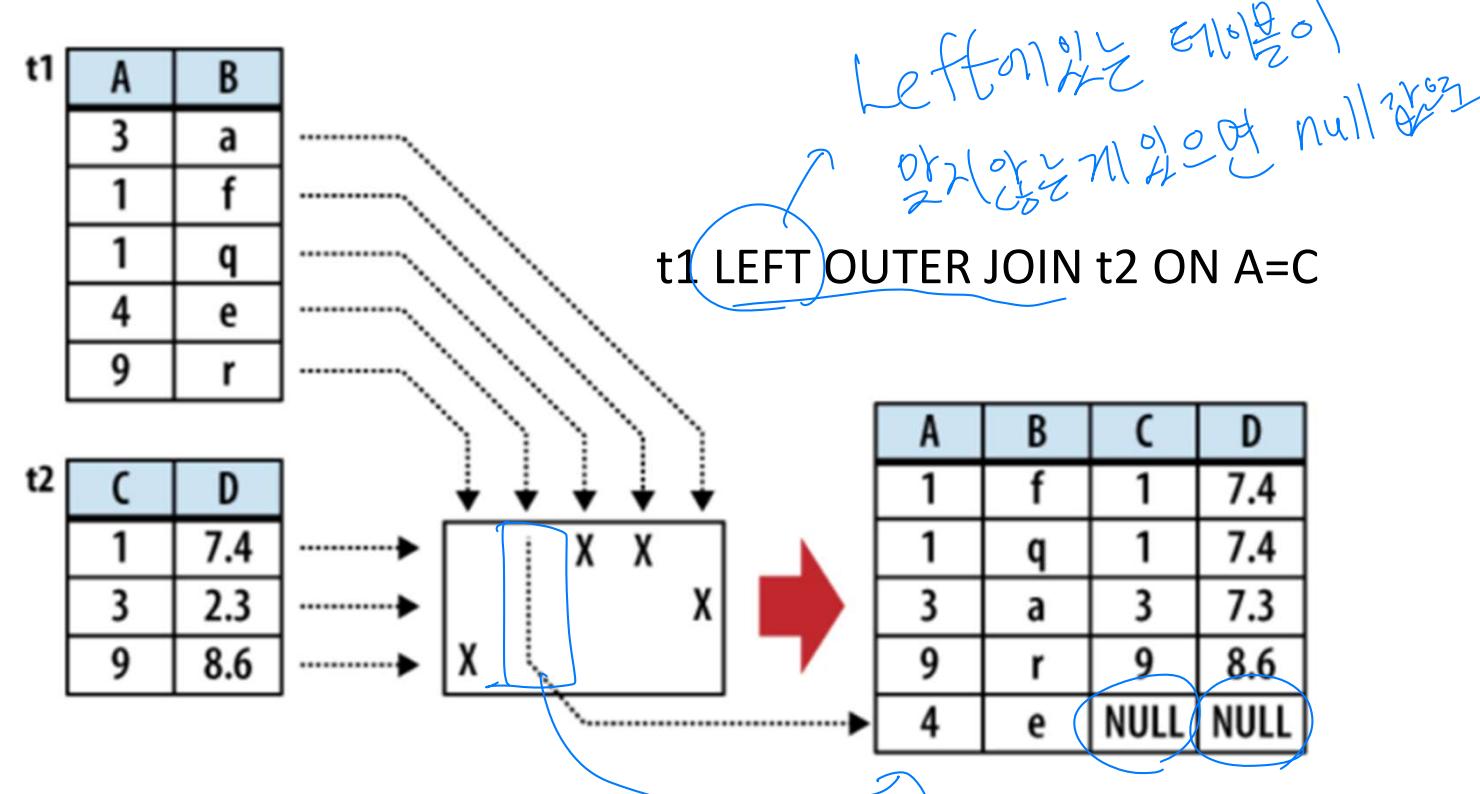
project

student_ID	project_ID
1	Prj1
3	Prj1
3	Prj2
5	Prj2

ID	name	student_ID	project_ID
1	Alice	1	Prj1
3	Charlie	3	Prj1
3	Charlie	3	Prj2

Outer Join

- Compute the join and then adds tuples that do not match any tuple
- Use *null* (unknown) values for missing attributes
- Three forms of outer join:
 - left outer join
 - right outer join
 - full outer join



Outer Join Examples

- Observe that

student is missing student_ID 5

project is missing student_ID 2 & 4

student

ID	name
1	Alice
2	Bob
3	Charlie
4	David

project

student_ID	project_ID
1	Prj1
3	Prj1
3	Prj2
5	Prj2

Left Outer Join

- Observe that

student is missing student_ID 5

project is missing student_ID 2 & 4

student

ID	name
1	Alice
2	Bob
3	Charlie
4	David

project

student_ID	project_ID
1	Prj1
3	Prj1
3	Prj2
5	Prj2

- student left outer join project on ID=student_ID***

ID	name	student_ID	project_ID
1	Alice	1	Prj1
3	Charlie	3	Prj1
3	Charlie	3	Prj2
2	Bob	NULL	NULL
4	David	NULL	NULL

Natural Left Outer Join

- Observe that

student is missing student_ID 5

project is missing student_ID 2 & 4

student

ID	name
1	Alice
2	Bob
3	Charlie
4	David

project

ID	project_ID
1	Prj1
3	Prj1
3	Prj2
5	Prj2

- student natural left outer join project*

ID	name	project_ID
1	Alice	Prj1
3	Charlie	Prj1
3	Charlie	Prj2
2	Bob	NULL
4	David	NULL

- In relational algebra:

student \bowtie *project*
left join

Right Outer Join

- Observe that

student is missing student_ID 5

project is missing student_ID 2 & 4

student

ID	name
1	Alice
2	Bob
3	Charlie
4	David

project

student_ID	project_ID
1	Prj1
3	Prj1
3	Prj2
5	Prj2

- *student right outer join project on ID=student_ID*

ID	name	student_ID	project_ID
1	Alice	1	Prj1
3	Charlie	3	Prj1
3	Charlie	3	Prj2
NULL	NULL	5	Prj2

Natural Right Outer Join

- Observe that

student is missing student_ID 5

project is missing student_ID 2 & 4

student

ID	name
1	Alice
2	Bob
3	Charlie
4	David

project

ID	project_ID
1	Prj1
3	Prj1
3	Prj2
5	Prj2

- *student natural right outer join project*

ID	name	project_ID
1	Alice	Prj1
3	Charlie	Prj1
3	Charlie	Prj2
5	NULL	Prj2

- In relational algebra:

student \bowtie *project*

Full Outer Join

- Observe that

student is missing student_ID 5

project is missing student_ID 2 & 4

student

ID	name
1	Alice
2	Bob
3	Charlie
4	David

project

student_ID	project_ID
1	Prj1
3	Prj1
3	Prj2
5	Prj2

- student right outer join project on ID=student_ID*

ID	name	student_ID	project_ID
1	Alice	1	Prj1
3	Charlie	3	Prj1
3	Charlie	3	Prj2
2	Bob	NULL	NULL
4	David	NULL	NULL
NULL	NULL	5	Prj2

Natural Full Outer Join

- Observe that

student is missing student_ID 5

project is missing student_ID 2 & 4

student

ID	name
1	Alice
2	Bob
3	Charlie
4	David

project

ID	project_ID
1	Prj1
3	Prj1
3	Prj2
5	Prj2

- *student natural full outer join project*

ID	name	project_ID
1	Alice	Prj1
3	Charlie	Prj1
3	Charlie	Prj2
2	Bob	NULL
4	David	NULL
5	NULL	Prj2

- In relational algebra:

student  *project*

Join Conditions and Types

- **Join condition** – Natural vs. On vs. Using

- defines how to match tuples

- **Join type** – Inner Join vs. {Left/Right/Full} Outer Join

- defines how tuples that do not match any tuple in the other relation are treated.

<i>Join types</i>
inner join
left outer join
right outer join
full outer join

<i>Join conditions</i>
natural
on < predicate >
using (A_1, A_2, \dots, A_n)

→ 앞 2가지의 조건을 만족하는 풀어쓰기

→ 뒤 2가지의 조건을 만족하는 풀어쓰기

using
or
on

Join Conditions and Types – More Examples

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

- *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

Join Conditions and Types – More Examples

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

- *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

- *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>

Join Conditions and Types – More Examples

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

- ***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

Outline

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

View

- In some cases, not all users should see the entire table.
- A **view** provides a way of hiding certain data
- A view is defined as follows:

create view *view_name* as <select statement>

- E.g. Let's create a view of instructors without their salary
 - Certain users do not need to know the salary.

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

faculty (logical view)

ID	name	dept_name
1	Alice	Comp. Sci.
2	Bob	Biology

Instructor (physical table)

ID	name	dept_name	salary
1	Alice	Comp. Sci.	20000
2	Bob	Biology	10000

View Examples

- Let's create a view of all instructors in the Biology department

 – create view *bio_faculty* as

- **select** *name*

from *faculty*

where *dept_name* = 'Biology'

- Note: One view may be used in the expression defining another view

- Expand the view :

select *

from *bio_faculty*

where name like '%dar%';

- To

```
select *
```

from (select name

from faculty

where *dept_name* = 'Biology')

where name like '%dar%';

Note#1: A view is purely logical; it is not a real table that stores data. Instead, it stores only the definition (i.e., the SELECT statement).

skip

~~Note#2: Materialized views can be physically stored on disks~~

Update of a View

- Let's add a new tuple to *faculty* view which we defined earlier

insert into *faculty*

values ('3', 'Charlie', 'Music');

→ *instructor* (logical)
↪ *faculty* (logical)

faculty view (logical)

ID	name	dept_name
1	Alice	Comp. Sci.
2	Bob	Biology

- This inserts into *instructor* (*ID*, *name*, *dept_name*, *salary*)
- Must have a value for *salary*.

- Two approaches

- Reject the insert
- Insert the tuple

('3', 'Charlie', 'Music', **null**)

into the *instructor* relation

Instructor (physical table)

ID	name	dept_name	salary
1	Alice	Comp. Sci.	20000
2	Bob	Biology	10000
3	Charlie	Music	????

Note: PostgreSQL uses NULL

Some Updates Cannot be Translated Uniquely

instructor

ID	name	dept_name	salary
1	Alice	Comp. Sci.	20,000
2	Bob	Biology	10,000

department

dept_name	building
Comp. Sci.	Taylor
Biology	Taylor
Elec. Eng.	Watson

- **create view instructor_info as**
select ID, name, building, budget
from instructor natural join department;

instructor_info (logical view)

ID	name	building
1	Alice	Taylor
2	Bob	Taylor

- **insert into instructor_info**
values ('4', 'David', 'Watson');
- Issue
 - Which department? Is EE dept is the only one in Watson?

DB27M 대입이정우
Insert 하는 경우는 차용문제X

부정한 View입니다

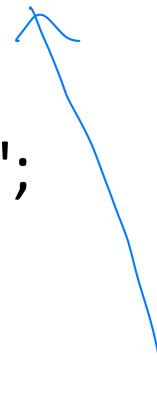
And Some Updates Should Not Be Allowed at All

- **create view history_instructors as**

```
select *
```

```
from instructor
```

```
where dept_name= 'History';
```



history
dept_name

- What happens if we insert

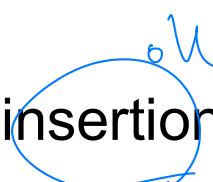
```
('25566', 'Brown', 'Biology', 100000)
```

into *history_instructors*?



insert X

If we allow the insertion, the tuple will not appear in the view



View Updates in SQL

- Most DBMSs allow updates only on simple views.
 - The **from** clause has only one database relation.
 - The **select** clause does not have any WHERE predicates, aggregates, or **distinct** specification.
 - The query does not have a **group by** or **having** clause.
 - Attributes not listed in the **select** clause can be set to null

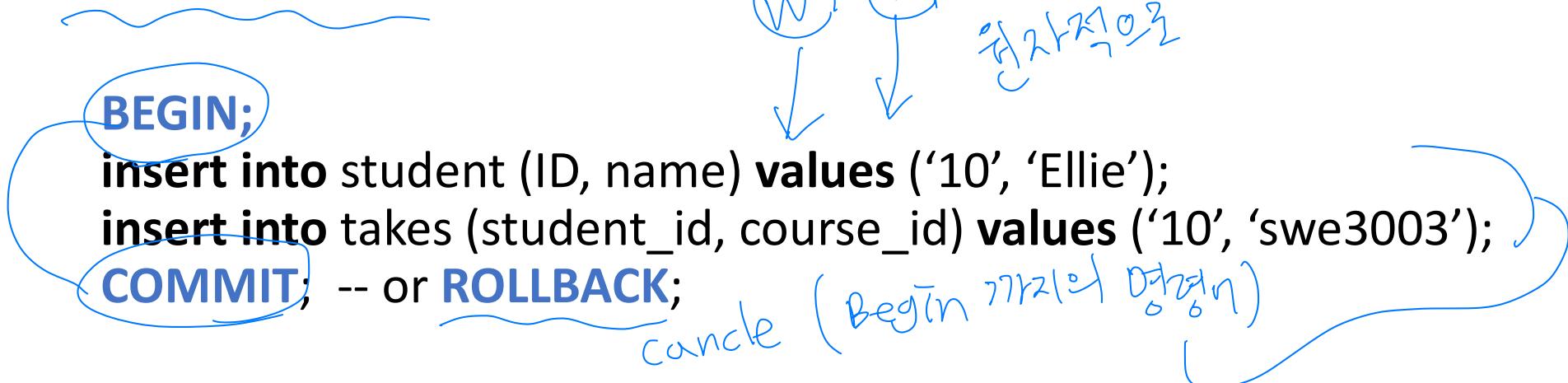
가장 먼저 이해해야 할 원칙
view는 실제 데이터의 일부로만 간주
(null - 실제 데이터의 일부로만 간주)
특정 속성을 신중하게 사용해야

Outline

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

Transactions

- A **transaction** consists of a sequence of SQL statements but is an atomic work



- Two insert statements will be either fully executed or rolled back as if both never occurred
- Isolation from concurrent transactions
 - More details in Chapter 17.

Outline

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

Integrity Constraints

무결성 제약

- Integrity constraints protect databases

- E.g.)

- A bank account must have a balance greater than \$0.00
 - A customer must have a (non-null) phone number
 - Minimum hourly wage is KRW 9,620
 - New legal cap on weekly work hours is 69
 - Etc.

Constraints on a Single Relation

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

(Boolean)

Not Null Constraints

- **not null**

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

name **varchar(20) not null**

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

Unique Constraints

- Candidate key is a set of attributes that uniquely identifies each record in the table.

- **unique** (A_1, A_2, \dots, A_m)

- Attributes (A_1, A_2, \dots, A_m) of a tuple must be unique
- Candidate keys are permitted to be null (in contrast to primary keys)
 - NULL is never equal to other NULL values
 - NULL is not considered a duplicate value

→ null은 어떤 값이든
우리하고 같아

- **primary key** (A_1, A_2, \dots, A_m)

- Primary key is the chosen candidate key.
- It cannot have NULL (unknown) values.

Check Constraints

- **check (P)** specifies a predicate P that must be satisfied.
- 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')))
```

만약 고려할
Insert → reject

Referential Integrity

- A value that appears in one relation must appear in another relation.
 - **Foreign** key is the primary key of another relation S.
 - Any values in foreign key A of R must appear in S.
- Example: If a customer ID appears in in the *Order* relation, there must be a tuple in the *customer* relation for the customer ID.
 - DBMS **rejects insert queries** that violate referential integrity.
 - E.g., **insert into order values('4', 'invalid_id', 346);** -- reject

Foreign Key

order_ID	customer_ID	item_ID
1	ByteBoi	224
2	FizzBug	356
3	FizzBug	128

Order Table

Primary Key

customer_ID	name	address
ByteBoi	Bob	... Suwon, KR
FizzBug	Charlie	... Seoul, KR

Customer Table

ByteBoi
FizzBug

Customer Table
reject

Referential Integrity (Cont.)

- Referential integrity is enforced via Foreign key constraint
 - Principle
 - method
 - Foreign keys are specified in SQL **create table** statement
 - foreign key (*customer_ID*) references *customer*
 - By default, a foreign key references the primary-key of the referenced table.
 - 기본적으론
 - SQL allows a list of attributes of the referenced relation to be specified explicitly.

foreign key (*cust_name*)

references *customer*(*customer name*)

↳ 주요 국가와 비교해
국내 시장은 그러나 타이어
제작 기관을 찾는 편이다.

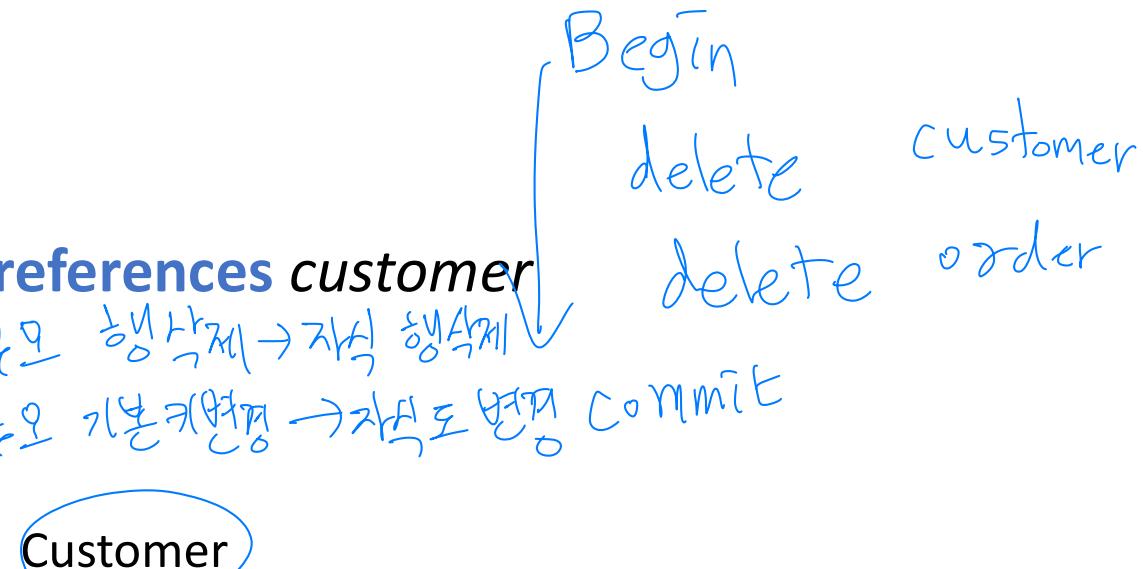
Cascading Actions in Referential Integrity

- In case of **delete** or **update**, cascade the action

```
create table order (
  ...
  customer_ID varchar(20),
  foreign key (customer_ID) references customer
    on delete cascade
    on update cascade,
  ...
)
```

Order

order_ID	customer_ID	item_ID
1	ByteBoi	224
2	FizzBug	356
3	FizzBug	128



```
delete customer where customer_ID = 'FizzBug';
```

- If a customer tuple is deleted/updated, corresponding order tuples are deleted/updated.

Cascading Actions in Referential Integrity

- In case of **delete** or **update**, cascade the action

```
create table order (
  ...
  customer_ID varchar(20),
  foreign key (customer_ID) references customer
    on delete cascade
    on update cascade,
  ...
)
```

Order

order_ID	customer_ID	item_ID
1	ByteBoi	224

Customer

customer_ID	name	address
ByteBoi	Bob	... Suwon, KR

```
delete customer where customer_ID = 'FizzBug';
```

- If a customer tuple is deleted/updated, corresponding order tuples are deleted/updated.

Cascading Actions in Referential Integrity

- Instead of **cascade**, we can use :

- set null**
- set default**

```
create table order (
```

```
(...)
```

```
customer_ID varchar(20) default 'pending',  
foreign key (customer_ID) references customer  
on delete set default, -- or on delete set null
```

```
...)
```

- E.g., **delete** customer **where** customer_ID = 'FizzBug';

Order

order_ID	customer_ID	item_ID
1	ByteBoi	224
2	FizzBug	356
3	FizzBug	128

Customer

customer_ID	name	address
ByteBoi	Bob	... Suwon, KR
FizzBug	Charlie	... Seoul, KR

Cascading Actions in Referential Integrity

- Instead of **cascade**, we can use :

- **set null**
- **set default**

```
create table order (
```

```
(...
```

```
    customer_ID varchar(20) default 'pending',  
    foreign key (customer_ID) references customer  
    on delete set default, -- or on delete set null
```

```
...)
```

- E.g., **delete** customer **where** customer_ID = 'FizzBug';

Order

order_ID	customer_ID	item_ID
1	ByteBoi	224
2	pending	356
3	pending	128

Customer

customer_ID	name	address
ByteBoi	Bob	... Suwon, KR

Integrity Constraint Violation During Transactions

- Consider:

```
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)
```

ID	Name	mother	father
10	Alice	null	null

- How to insert a tuple without causing constraint violation?
 - Set father and mother to **null** (unknown) initially
 - The first ancestor must have NULL in father and mother

Complex Check Conditions

- The predicate in the check clause can include a subquery.

```
create table section (
```

```
...
```

```
    time_slot_id varchar (4),
```

```
...
```

```
    check (time_slot_id in
           (select time_slot_id from time_slot)
        ))
```

- The condition is checked when a tuple is inserted or modified in *section*, also when the relation *time_slot* changes

Assertions

assert ()

- An **assertion** is a predicate expressing a condition that the database must satisfy.

- An assertion in SQL takes the form:

create assertion <assertion-name> **check** (<predicate>);

- Example assertions:

- An instructor cannot teach in two different classrooms in a semester in the same time slot

```
create assertion no_double_booking
check (
    not exists (
        select 1
        from schedule s1 join schedule s2 using (instructor_id)
        where s1.classroom_id != s2.classroom_id
            and s1.time_slot = s2.time_slot
    ));

```

- Note: Unfortunately, no DBMS supports assertions despite it is in SQL standard specification

자원 X

Outline

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

Built-in Data Types in SQL

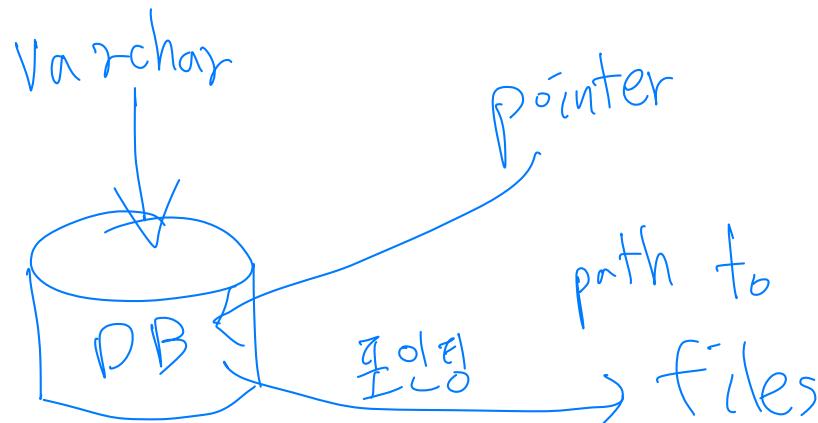
- **date:** Dates, containing a (4 digit) year, month and date
 - Example: `date '2005-7-27'`
- **time:** Time of day, in hours, minutes and seconds.
 - Example: `time '09:00:30'` `time '09:00:30.75'`
- **timestamp:** date plus time of day
 - Example: `timestamp '2005-7-27 09:00:30.75'`
- **interval:** 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

Large-Object Types

- Large objects (KB, MB, GB) are stored as :

- 자료형
- **blob**: binary large object (e.g., photos, videos, etc)
 - **clob**: character large object (e.g., large text files)

- When a query returns a large object, a pointer is returned rather than the large object itself.



User-Defined Types

- **create type** creates user-defined type

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

12.00
—
2x1r2

- Example:

```
create table department
( dept_name varchar (20),
  building varchar (15),
  budget Dollars);
```

Domains

- **create domain** *Creates user-defined domain types*

`create domain person_name char(20) not null`

or check와 같은 제한 가능

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.

- Example:

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

조건에 해당하는 것만 변수 만들기

Outline

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

Index Creation

BST Hash

- Many queries reference only a small proportion of the records in a table.
- It is inefficient for the system to scan the entire relation to find a record with particular value
전체를 스캔하는데 비효율적!
- An **index** is a data structure that allows DBMS to find tuples efficiently (e.g., B+tree, Hash Table).
- We create an index with the **create index** command

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

*Index를 부여하고 싶을 때
→ takes (student ID, course ID)*

Index Creation Example

- **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)** -- B+tree

- **create index studentID_index on student using HASH (ID)** -- Hash

- The query:

```
select *  
from student  
where ID = '12345'
```

will use the index and return the output relation faster

$O(\log N)$

↑
default

$O(1)$

Hash
faster!

Outline

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

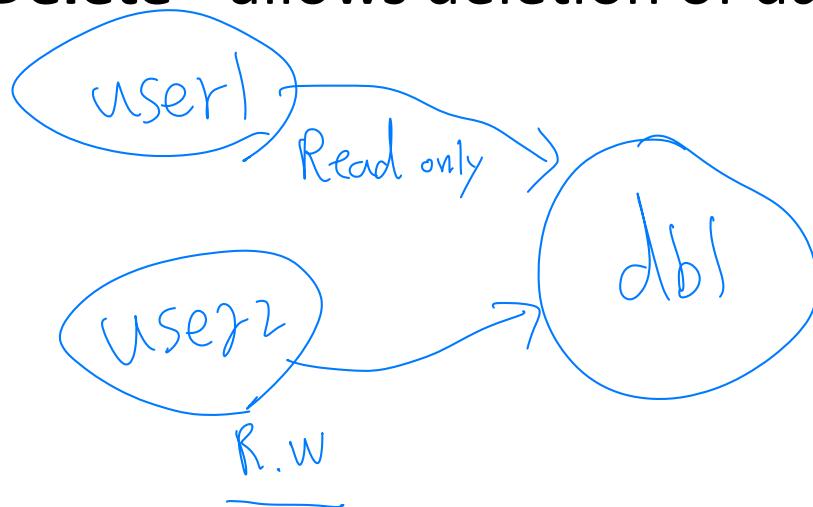
Authorization

- all, none, or a combination of the following **privileges** on a relation or a view can be authorized.

Super User ↴

특권

- **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.



Authorization Specification in SQL

- The **grant** statement is used to grant authorization

grant <privilege list> on <relation or view > to <user list>

- <user list> is:

- a user-id
→ 누구에게 주고 싶으면
- **public**, which allows all valid users the privilege granted
- A role (more on this later)

- Example:

- **grant all privileges on department to db_user1, db_user2**

- Granting a privilege on a view does not imply granting any privileges on the underlying relations.

Privilege List

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

- Example: grant users *db_user1* and *db_user2* **select** authorization on the *department* relation:

```
grant select on department to db_user1, db_user2
```

- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: all the allowable privileges

Revoking Authorization

- The **revoke** statement is used to revoke authorization.
revoke <privilege list> on <relation or view> from <user list>

- Example:

revoke select on student from U₁, U₂, U₃

- <privilege-list> may be **all** to revoke all privileges the revoker may hold.
- If <user list> includes **public**, all users lose the privilege except those granted it explicitly.

