Database Systems
Lecture03 & 04 – Ch 3. Introduction to SQL

Beomseok Nam (남범석)
bnam@skku.edu

PostgreSQL

- A Relational Database Management System
- PostgreSQL (pronounced “POST-gress” and QL is silent)
- ANSI-SQL:2008 compatible (Most Oracle-compatible)

DBA creates Users and Databases

- When PostgreSQL is installed, DBA account 'postgres' is created
- 'postgres' can login to PostgreSQL DBMS by running 'psql'

```
postgres@swui:~$ psql
psql (14.7 …)
postgres=#
```

- To create a new user, DBA (postgres) runs the following statements in psql

```
postgres=> CREATE USER db20333999 with
encrypted password 'changethis';
```

- To create a database, DBA runs the following statements in psql

```
postgres=> CREATE DATABASE db20333999;
postgres=> GRANT ALL PRIVILEGES on database
db20333999 TO db20333999;
```

User Login to PostgreSQL

- Users can login to psql as follows

```
20203333999@swye:~$ psql -h localhost -U db20333999
Password for user db20333999: changethis <ENTER>
psql (14.7 (…))
SSL connection (protocol: TLSv1.3, …)
Type "help" for help.

db20333999=>
```

PostgreSQL: psql Command Line options

- -A, --no-align: set output to non-aligned, no padding
- -c sql, --command sql: execute the sql command
- -d name, --dbname name: name of database
- -f name, --file name: use name as the source of commands
- -o name, --output name: put the output in name
- -q, --quiet: suppress welcome messages
- -t, --tuples-only: suppress print column names, result row counters, etc
- -?, --help: get command line help

psql Meta-commands

- \l: list existing databases
- \c db_*name*: connect to a database name
- \d: list all tables (display)
- \d *table*: show information about a *table*
- \copy *table*: copy a *table* to/from a file
- \e: edit the query buffer
- \p: display the query buffer
- \g: execute the query buffer (go)
- \r: resets (clears) the query buffer
- \w *name*: write the query buffer to *name*

Listing & Choosing Databases in PostgreSQL

- Use the `\l` command to find out which databases currently exist on the server:

```
db20333999 => \l
 List of databases
    Name     |  Owner   | Encoding |   Collate   |    Ctype    | privileges
-------------+----------+----------+-------------+-------------+-------------
 db16311252 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 db16311648 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 db20333999 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
…
```

```
db20333999=> \c db20333999
psql (14.7 (…))
SSL connection (protocol: TLSv1.3, …)
You are now connected to database "db20333999" ...
db20314193=>
```

psql Meta-commands

- \i file_*name*: read *file* into query buffer (input)
- \q: quit the program
- \! *Command*: execute the Linux *command*
- \h command: display help on command
- \?: help on meta-commands

Creating a Table

- Once you choose a database, you can display tables in the database:

```
db20333999 => \dt
  Did not find any relations.
```

- An empty set indicates that you have not created any table yet.

Loading Sample Data

- Create a text file `pet.txt' containing one record per line.
- Values must be separated by commas or tabs, and given in the order in which the columns were listed in the CREATE TABLE statement.
- Then load the data via the COPY Command.

Sample Data File

Fluffy	Harold	cat	f	1993-02-04	\N
Claws	Gwen	cat	m	1994-03-17	\N
Buffy	Harold	dog	f	1989-05-13	\N
Fang	Benny	dog	m	1990-08-27	\N
Bowser	Diane	dog	m	1979-08-31	1995-07-29
Chirpy	Gwen	bird	f	1998-09-11	\N
Whistler	Gwen	bird	\N	1997-12-09	\N
Slim	Benny	snake	m	1996-04-29	\N

To Load pet.txt:

```
db20333999 => COPY myTable FROM 'pet.txt' (FORMAT CSV,
DELIMITER('\t'));
```

Run SQL source code

- You can create a SQL source file `test.sql' containing SQL statements using a text editor and run it in psql shell.

To Run test.sql:

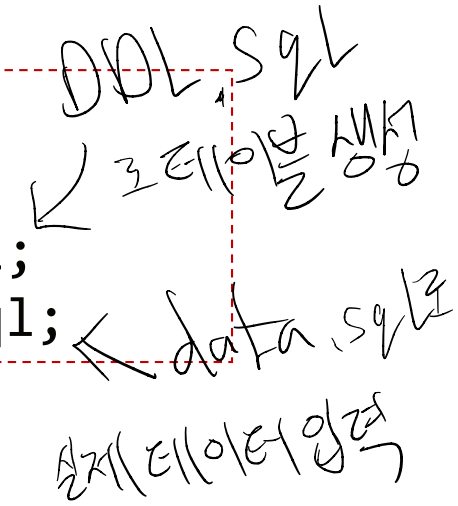
```
db20333999 => \i test.sql;
```

University Database Example

- in ~swe3003/sample_db

To Create and populate University Database:

```
db20333999 => \i /home/swe3003/sample_db/DDL.sql;
db20333999 => \i /home/swe3003/sample_db/data.sql;
```



Chapter 3: Introduction to SQL

DDL: Data Definition Language

- The SQL **data-definition language (DDL)** allows the specification about relations, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- And as we will see later, also other information such as
  - The set of indexes to be maintained for each relations.
  - Security and authorization information for each relation.
  - The physical storage structure of each relation on disk.

Domain Types in SQL

- **char(n).** Character string, with fixed length *n*.
- **varchar(n).** Variable length string, with maximum length *n*.
- **int.** Integer
- **smallint.** Small integer
- **numeric(p,d).** Fixed point number
  - with precision of *p* digits, with *d* digits to the right of decimal point.
- **real.** Floating point number
- **float(n).** Floating point number
  - with user-specified precision of at least *n* digits.

Create Table

create table *TableName* (*Attribute*$_1$ *DataType*$_1$,
Attribute$_2$ *DataType*$_2$,
...,
Attribute$_n$ *DataType*$_n$,
[integrity-constraint],
...)

- Example:

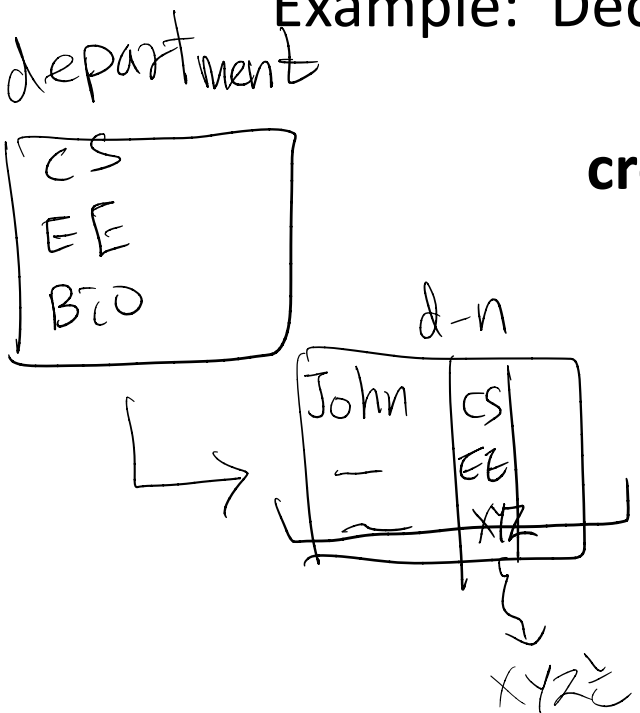
create table *instructor* (
ID **char**(5),
name **varchar**(20) **not null**,
dept_name **varchar**(20),
salary **numeric**(8,2))

name에 null 이들어갈수X

Integrity Constraints in Create Table

- **not null**
- **primary key** (A_1, ..., A_n)
- **foreign key** (A_m, ..., A_n) **references** *r*

Example: Declare *dept_name* as the primary key for *department*




```
create table instructor (
        ID              char(5),
        name            varchar(20) not null,
        dept_name       varchar(20),
        salary          numeric(8,2),
        primary key (ID),
        foreign key (dept_name) references department)
```

primary key automatically ensures **not null**

And a Few More Relation Definitions

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

- **create table** *takes* (
  ID **varchar**(5),
  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 (*ID*) **references** *student,*
  foreign key (*course_id, sec_id, semester, year*) **references** *section*);

행이 구분되기위한 최소조건

primary key

And more still

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

Drop and Alter Table Constructs

- **drop table** *student*
  - Deletes the table schema and its contents
- **delete from** *student* -- *DML*
  - Deletes all contents of table, but retains table schema
- **alter table**
  - **alter table** *r* **add** *A D*
    - *A* is the name of the attribute to be added to relation *r*
    - *D* is the domain of *A.*
    - All existing tuples in the relation get *null* for the new attribute.
  - **alter table** *r* **drop** *A*
    - Dropping of attributes is not supported by many databases

sql standard

DML (Data-Manipulation Language)

- DML
  - select / insert / update / delete
  - set operations
  - ordering
  - aggregate functions
  - nested subqueries

sum, max, min

R, A (Redundant

DML (Data-Manipulation Language)

- A typical SQL query :

select $A_1, A_2, \ldots, A_n$
from $r_1, r_2, \ldots, r_m$
where P

→ set
Relation

- Note: The result of an SQL query is also a relation.

The select Clause

- The **select** clause list the attributes
  - corresponds to the projection operation (Π)

- Example: find the names of all instructors:

select *name*

from *instructor*

- NOTE: SQL are case insensitive
  - i.e., upper- or lower-case letters do not matter.
  - E.g. *Name* ≡ *NAME* ≡ *name*

The select Clause (Cont.)

- SQL allows duplicates in relations
- To eliminate duplicates, use **distinct** after select.

- Find all department names of instructor, and remove duplicates

```
select distinct dept_name
from instructor
```

- The keyword **all** allows duplicates to be retained.

```
select all dept_name
from instructor
```

The select Clause (Cont.)

- An asterisk (*) denotes "all attributes"

모든 속성

```
select *
from instructor
```

- The **select** clause can use arithmetic expressions
  - +, –, *, and /

- E.g.:

```
select ID, name, salary/12
from instructor
```

salary/12 한 값을 출력 (as ~~)로 "~~"이라는 열이름으로

The where Clause

- The **where** specifies conditions that the result must satisfy
  - boolean operators (**and or not** …)
  - comparison operators (<, >, =, …)
  - and more…

- Find all instructors in Comp. Sci. dept with salary > 80000

```
select name
from instructor
where dept_name = 'Comp. Sci.' and
        salary > 80000
```

The from Clause

- The **from** clause lists the relations
  - Corresponds to the Cartesian product (*X*)

select *
from *instructor, teaches*

Generates Cartesian product *instructor X teaches*, with all attributes from both relations

<기능>
select : 출력
where : 조건
from : 어떤 테이블에서? (, 는 X 곱셈으로)

Join (⋈)

- $R \bowtie S = \Pi \sigma_{R.common\_attrs=S.common\_attrs} (R \times S)$
- Find the names of instructors and the course ID of the courses they taught.

```
select name, course_id
 from instructor, teaches
 where   instructor.ID = teaches.ID
```

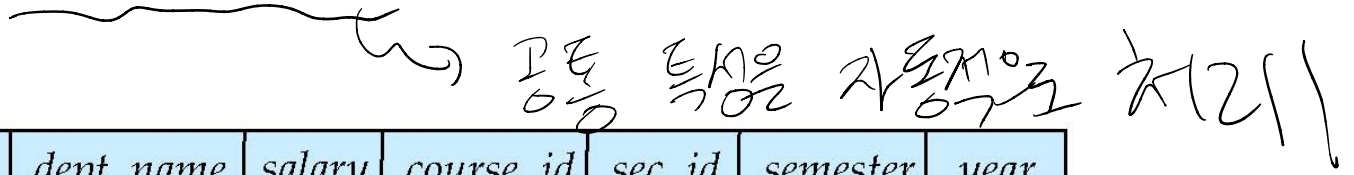
Join (⋈) - Example

- Find the course ID, semester, year and title of each course offered by the Comp. Sci. department

```
select section.course_id, semester, year, title
  from section, course
 where section.course_id = course.course_id
   and dept_name = 'Comp. Sci.'
```

Natural Join

- Natural join retains only one copy of common column
- **select** *
  from *instructor* **natural join** *teaches*;



ID	*name*	*dept_name*	*salary*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010

Natural Join Example

- List the names of instructors along with the course ID of the courses that they taught.

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

  - **select** *name*, *course_id*
    from *instructor* **natural join** *teaches*;

Danger in Natural Join

- Danger in natural join: beware of unrelated attributes with same name which get equated incorrectly

Instructor

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

Course

c_id	title	dept_name
swe3003	DB	'Comp. Sci'
swe3021	Multicore	'Comp. Sci'

List the names of instructors along with the the titles of courses that they teach

select *name*, *title*
from *instructor* **natural join** *teaches* **natural join** *course*;

Danger in Natural Join

Instructor

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	swe3004	Spring	2024

List the names of instructors along with the the titles of courses that they teach

Course

c_id	title	dept_name
swe3003	DB	'Comp. Sci'
swe3004	OS	'Comp. Sci'

select *name*, *title*
from *instructor* **natural join** *teaches* **natural join** *course*;

name	title
Alice	DB
Alice	OS

Uh oh, (Bob, DB) is missing because course.dept_name is different from instructor.dept_name

Danger in Natural Join

Instructor

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	swe3004	Spring	2024

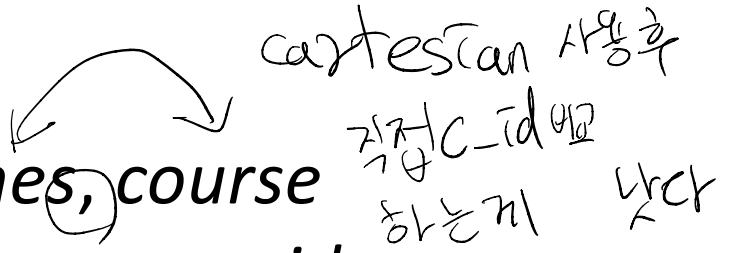
List the names of instructors along with the the titles of courses that they teach

Course

c_id	title	dept_name
swe3003	DB	'Comp. Sci'
swe3004	OS	'Comp. Sci'

Correct version:

select *name*, *title*
from *instructor* **natural join** *teaches*, *course*
where *teaches.course_id* = *course.course_id*;



Rename (ρ) Operation

- **as** clause renames relations and attributes :

  old-name **as** *new-name*

- E.g.
  - **select** *ID, name, salary/12* **as** *monthly_salary*
    from *instructor*

- Keyword **as** is optional and may be omitted

  instructor **as** *T* ≡ *instructor T*

  - Keyword **as** must be omitted in Oracle DBMS

String Operations

- string-matching operators: = and **like**
  - percent % - matches any substring.
  - underscore _ - matches any single character.

- Find the names of all instructors whose name includes the substring “dar”.

```
select name
from instructor
where name like '%dar%'
```

- Match the string “100 %”

```
like ‘100 \%'  escape  '\'
```

String Operations (Cont.)

- Patterns are case sensitive.
- Pattern matching examples:
  - 'Intro%' matches any string beginning with "Intro".
  - '%Comp%' matches any string containing "Comp"
  - '_ _ _' matches any string of exactly three characters.
  - '_ _ _ %' matches any string of at least three characters.

- SQL supports a variety of string operations such as
  - concatenation (using "||")
  - converting from upper to lower case (and vice versa)
    - UPPER(), LOWER()
  - finding string length, extracting substrings, etc.
    - LENGTH(), SUBSTRING(str, position, length)

Sorting: order by Clause

- List in alphabetic order the names of all instructors

  select *name*
  from *instructor*
  order by *name*

- **asc** for ascending order (default)

ID	name	dept_name	salary
20	Alice	Comp. Sci.	8,000
30	Charlie	Biology	null
40	Bob	Comp. Sci.	6,000

name
Alice
Bob
Charlie

- **desc** for descending order
  - **select** *name*
    from *instructor*
    order by *name* **desc**

name
Charlie
Bob
Alice

Sorting: order by Clause

- Can sort on multiple attributes (lexicographic sorting)
  - **select** *dept_name, name*
    from *instructor*
    order by *dept_name, name*

select하지않아도 order by 가능

1st

ID	name	dept_name	salary
20	Alice	Comp. Sci.	8,000
30	Charlie	Biology	null
40	Bob	Comp. Sci.	6,000

dept_name	name
Biology	Charlie
Comp. Sci.	Alice
Comp. Sci.	Bob

null은 가장 뒤로 정렬됨

Range query and tuple comparison

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between $90,000 and $100,000 (that is, ≥ $90,000 and ≤ $100,000)
  - **select** *name*
    from *instructor*
    where *salary* **between** 90000 **and** 100000

- Tuple comparison
  - **select** *name*, *course_id*
    from *instructor*, *teaches*
    where (*instructor.ID*, *dept_name*) =
    (*teaches.ID*, 'Biology');

Set Operations

- Set operations **union**, **intersect**, and **except**
  - Each of the above operations automatically eliminates duplicates

- To retain all duplicates use the corresponding multiset operations **union all**, **intersect all** and **except all**.

Set Operations

- Find courses that were offered in Fall 2024 or in Spring 2025

(**select** *c_id* **from** *course* **where** *semester* = 'Fall' **and** *year* = 2024)
union
(**select** *c_id* **from** *course* **where** *semester* = 'Spring' **and** *year* = 2025)

- Find courses that were offered in Fall 2024 and in Spring 2025

(**select** *c_id* **from** *section* **where** *semester* = 'Fall' **and** *year* = 2024)
intersect
(**select** *c_id* **from** *section* **where** *semester* = 'Spring' **and** *year* = 2025)

- Find courses that were offered in Fall 2024 but not in Spring 2025

(**select** *c_id* **from** *section* **where** *semester* = 'Fall' **and** *year* = 2024)
except
(**select** *c_id* **from** *section* **where** *semester* = 'Spring' **and** *year* = 2025)

Null Values

- Tuples may have a null value, denoted by *null*
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null
- The predicate **is null** can be used to check for null values.
  - Example: Find all instructors whose salary is null.

null값을 확인하기위해

```
select name
from instructor
where salary is null
```

unknown value라
사칙연산을 해도 null임

Null Values and Three Valued Logic

- Any comparison with *null* returns *unknown*
  - Example*: 5 < null or null <> null or null = null*
- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*) = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - AND: *(true* **and** *unknown) = unknown,*
    (false **and** *unknown) = false,*
    (unknown **and** *unknown) = unknown*
  - NOT*: (***not** *unknown) = unknown*
  - "*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Aggregate Functions

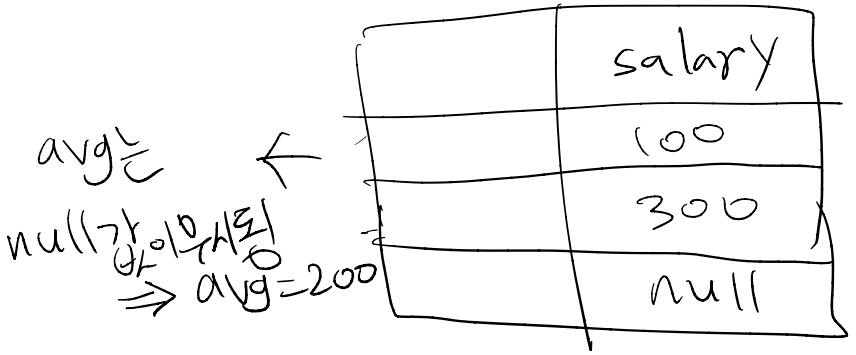
- These functions operate on the multiset of values of a column of a relation, and return a value

avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of values

select 다음구절에 사용가능

Aggregate Functions (Cont.)

- Find the average salary of instructors in the Computer Science department
  - **select avg** (*salary*)
    from *instructor*
    where *dept_name*= 'Comp. Sci.';



- Find the total number of instructors who teach a course in the Spring 2014 semester
  - **select count** (**distinct** *ID*)
    from *teaches*
    where *semester* = 'Spring' **and** *year* = 2014

중복제거

- Find the number of tuples in the *course* relation
  - **select count** (*)
    from *course*;

전체

count(salary)는 null제외

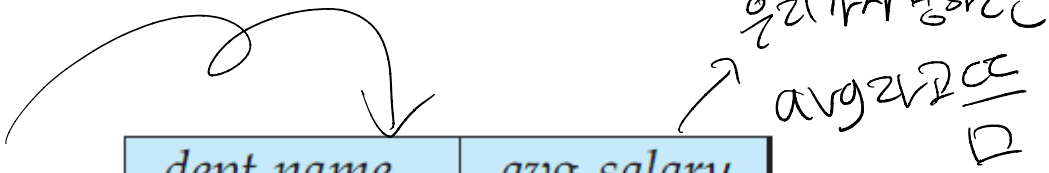
count

Aggregate Functions – group by

- Find the average salary of instructors in each department
  - **select** *dept_name*, **avg** (*salary*)
    from *instructor*
    group by *dept_name*;
  - Note: departments with no instructor will not be shown

aggregate : 합계
합계할 수 있는 값일 때 가능하다.

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000



dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Aggregation (Cont.)

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list

  - /* erroneous query */
    select *dept_name*, *ID*, **avg** (*salary*)
    from *instructor*
    group by *dept_name*;

(Instructor)

dept_name G dept_name, avg(salary)

$$_{G_1, G_2, \ldots, G_n}\mathcal{G}_{F_1(A_1), F_2(A_2, \ldots, F_n(A_n)}(E)$$

E is any relational-algebra expression

- $G_1, G_2 \ldots, G_n$ is a list of attributes on which to group (can be empty)
- Each F_i is an aggregate function
- Each A_i is an attribute name

Aggregate Functions – having Clause

- Find the names and average salaries of all departments whose average salary is greater than 42000

```
select dept_name, avg (salary)
from instructor
group by dept_name
having avg (salary) > 42000;
```

Salary가 10000이상인 애들만 avg계산

where Salary > 1000

group을 한 후에 필터 적용됨 (where은 group전에 적용됨)

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

Null Values and Aggregates

- The following query

  select sum (*salary*), **avg** (*salary*)
  from *instructor*

  - ignores tuples with salary=null
  - Result is *null* if there is no non-null amount

ID	name	dept_name	salary
20	Alice	Comp. Sci.	8,000
30	Charlie	Biology	null
40	Bob	Comp. Sci.	6,000

sum(salary)	avg(salary)
14,000	7,000

- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes

Nested subquery

- SQL allows nesting of subqueries.
- A **subquery** is a **select-from-where** expression within another query.
- Common use cases of subqueries
  - set membership test (IN , NOT IN)
  - set comparisons (SOME, ALL)
  - set cardinality

Example Query

- Find IDs of students who take DB class

```
select student_id        → 20
from Take                → 40
where  course_id IN (
        select course_id
        from Course
        where title = 'DB'
)
```

sub query 먼저 실행

```
select course_id
from Course
where title = 'DB'
```

Student

ID	name
20	Alice
30	Charlie
40	Bob

Take

	ID	c_id
o	20	swe3003
X	20	swe3004
X	30	swe3004
o	40	swe3003
X	40	swe3021

Course

c_id	title
swe3003	DB
swe3004	OS
swe3021	Multicore

c_id
swe3003

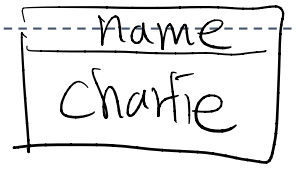
subquery의 include 되어있는지 확인

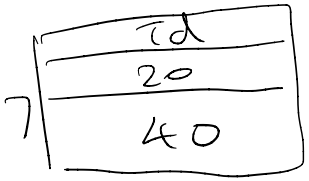
∈ 포함되어 있으면 null

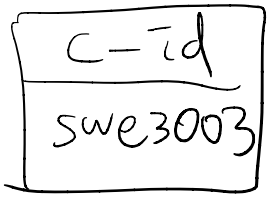
Example Query

- Find IDs of students who do not take DB class

```
select student_name
from Student
where student_id NOT IN (
        select student_id
        from Take
        where  course_id IN (
                select course_id
                from Course
                where title = 'DB'
        )
)
```







Student

ID	name
20	Alice
30	Charlie
40	Bob

Take

ID	c_id
20	swe3003
20	swe3004
30	swe3004
40	swe3003
40	swe3021

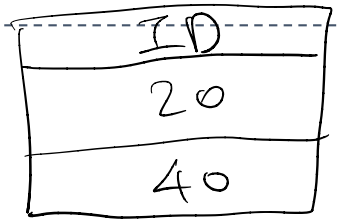
Course

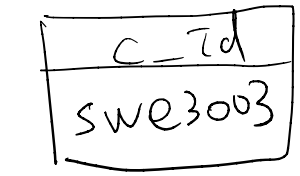
c_id	title
swe3003	DB
swe3004	OS
swe3021	Multicore

Example Query

- Find the total number of (distinct) students take course taught by the instructor with *ID* 10101

```
select count (distinct ID)
from takes
where c_id IN
        (select course_id
         from teaches
         where teaches.ID= 10101);
```





- Note: This query can be written in a much simpler manner.

Take

ID	c_id
20	swe3003
20	swe3004
30	swe3004
40	swe3003
40	swe3021

Teaches

ID	c_id
10101	swe3003
20202	swe3004
30303	swe3021

Set Comparison

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept_name = 'Biology';
```

Cartesian product (T x S)

T.name	dept	T.salary	S.name	S.dept	S.salary	
~~Alice~~	~~Comp. Sci.~~	~~8,000~~	~~Alice~~	~~Comp. Sci.~~	~~8,000~~	
Alice	Comp. Sci.	8,000	Charlie	Biology	2,000	8000>2000
~~Alice~~	~~Comp. Sci.~~	~~8,000~~	~~Bob~~	~~Comp. Sci.~~	~~6,000~~	
Alice	Comp. Sci.	8,000	David	Biology	7,000	8000>7000
~~Charlie~~	~~Biology~~	~~2,000~~	~~Alice~~	~~Comp. Sci.~~	~~8,000~~	
~~Charlie~~	~~Biology~~	~~2,000~~	~~Charlie~~	~~Biology~~	~~2,000~~	
~~Charlie~~	~~Biology~~	~~2,000~~	~~Bob~~	~~Comp. Sci.~~	~~6,000~~	
~~Charlie~~	~~Biology~~	~~2,000~~	~~David~~	~~Biology~~	~~7,000~~	
~~Bob~~	~~Comp. Sci.~~	~~6,000~~	~~Alice~~	~~Comp. Sci.~~	~~8,000~~	
...						

name	dept	salary
Alice	Comp. Sci.	8,000
Charlie	Biology	2,000
Bob	Comp. Sci.	6,000
David	Biology	7,000

Set Comparison

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.
  - Same query using > **some** clause

```
select name
from instructor
where salary > some (select salary
                     from instructor
                     where dept_name = 'Biology');
```

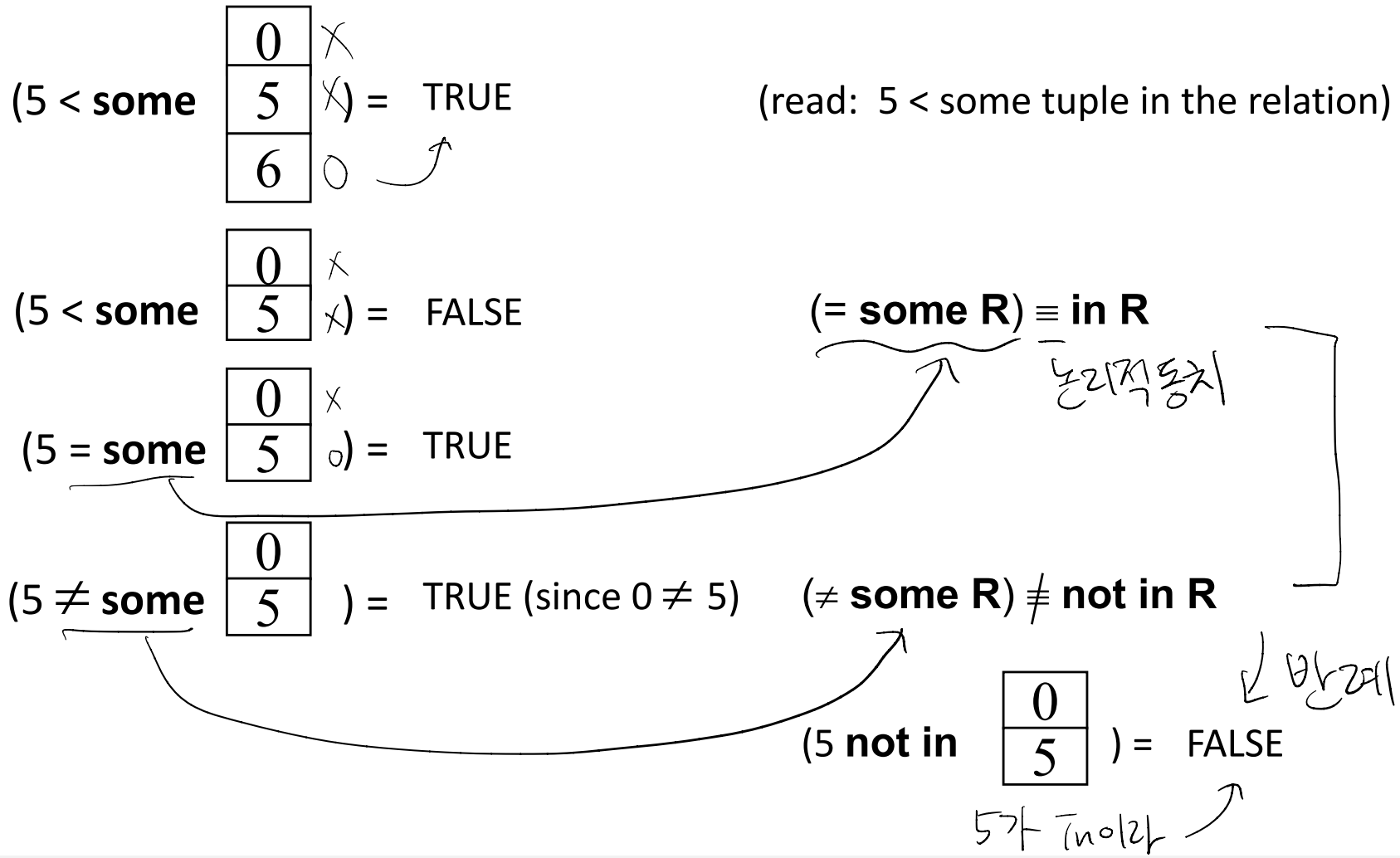
name	dept	salary
Alice	Comp. Sci.	8,000
Charlie	Biology	2,000
Bob	Comp. Sci.	6,000
David	Biology	7,000

> some

name	dept	salary
Charlie	Biology	2,000
David	Biology	7,000

Definition of some Clause

- F <comp> **some** $r \Leftrightarrow \exists\, t \in r$ such that (F <comp> t)
  Where <comp> can be: $<,\ \leq,\ >,\ =,\ \neq$



Example Query

- Find the names of all instructors whose salary is greater than the salary of **all** instructors in the Biology department.

```
select name
from instructor
where salary > all (select salary
                    from instructor
                    where dept_name = 'Biology');
```

name	dept	salary
Alice	Comp. Sci.	8,000
Charlie	Biology	2,000
Bob	Comp. Sci.	6,000
David	Biology	7,000

> all

name	dept	salary
Charlie	Biology	2,000
David	Biology	7,000

→ 2000, 7000 둘 다보다 커야됨

Definition of all Clause

- F <comp> **all** *r* $\Leftrightarrow$ $\forall$ *t* $\in$ *r* (F <comp> *t)*

(5 < **all** | 0, 5, 6 |) = false

(5 < **all** | 6, 10 |) = true

(5 = **all** | 4, 5 |) = false

(5 $\neq$ **all** | 4, 6 |) = true (since 5 $\neq$ 4 and 5 $\neq$ 6)

전부아니다 = 모든 속해 있지 않다

($\neq$ **all**) $\equiv$ **not in**

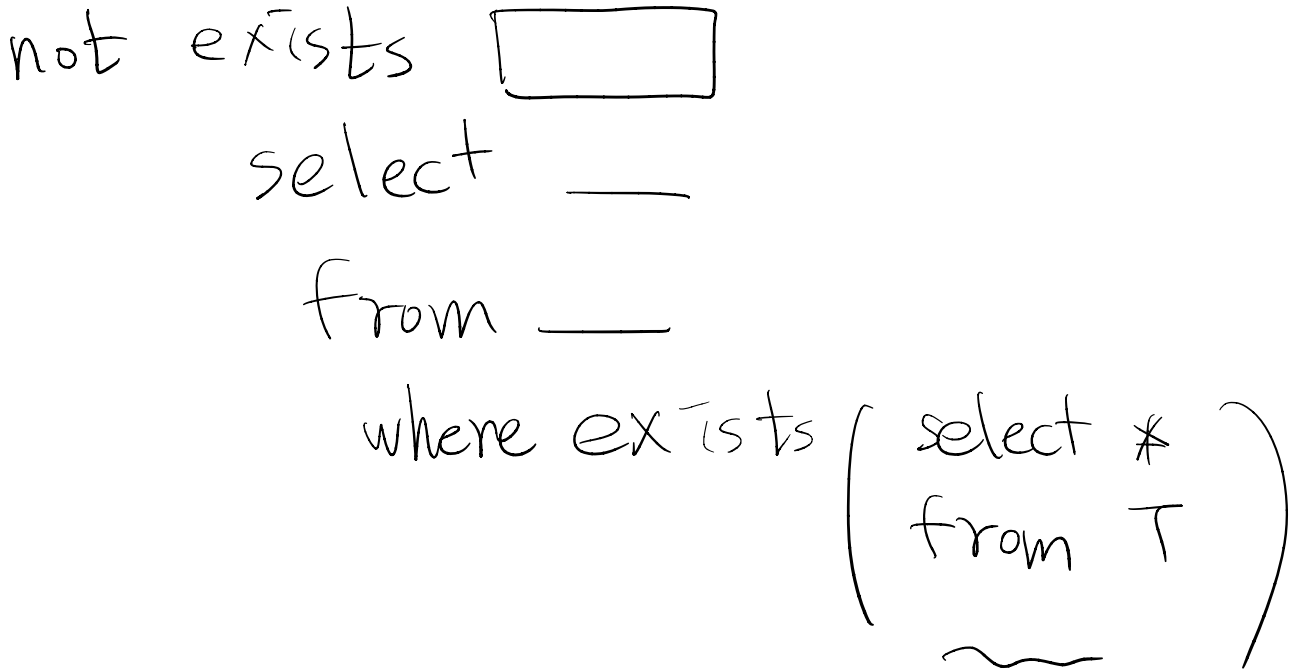
However, (= **all**) $\neq$ **in**

4가지 개념하기

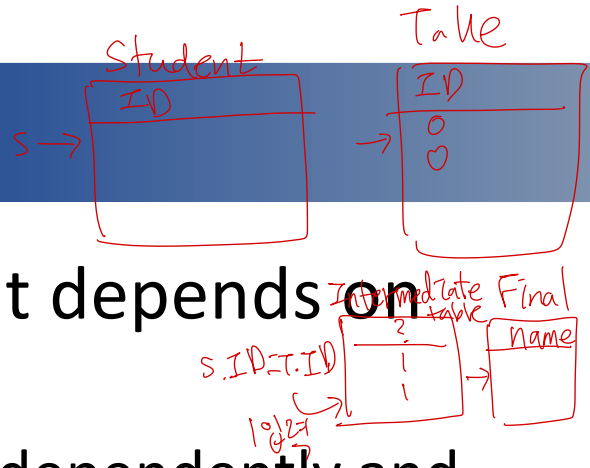
- = some R $\equiv$ in R
- $\neq$ some R $\not\equiv$ not in R
- $\neq$ all R $\equiv$ not in R
- = all R $\not\equiv$ in R

Test for Empty Relations: exists

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$



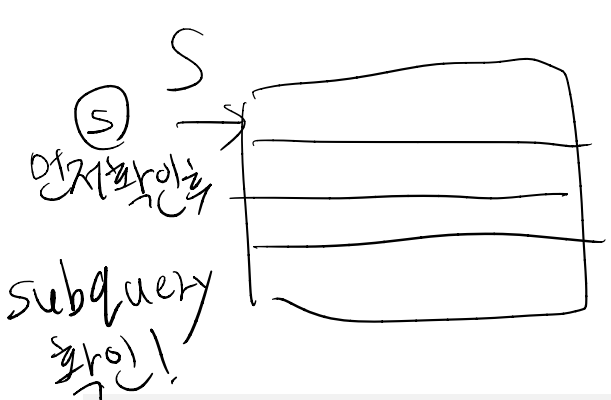
Correlation Variables

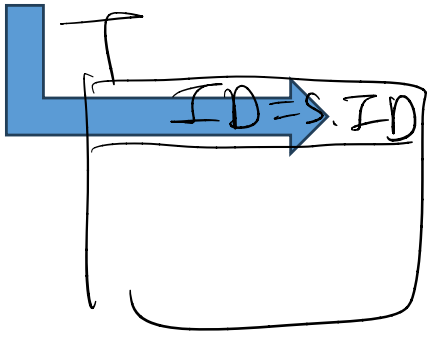


- **correlated subquery** is a subquery that depends on the outer query for its values.
  - Unlike a **regular subquery** (which runs independently and returns a fixed result), a **correlated subquery is executed once for each row** in the outer query.
  - Eg. Find names of student who have taken at least one course

nested for loop

```
select distinct name
from Student natural join Take;
```






```
select distinct name
from Student s
where EXISTS (
   select 1
   from Take t
   where t.ID = s.ID
);
```

S가 있는지 알려면 outer 문에 확인해봐

Inner query

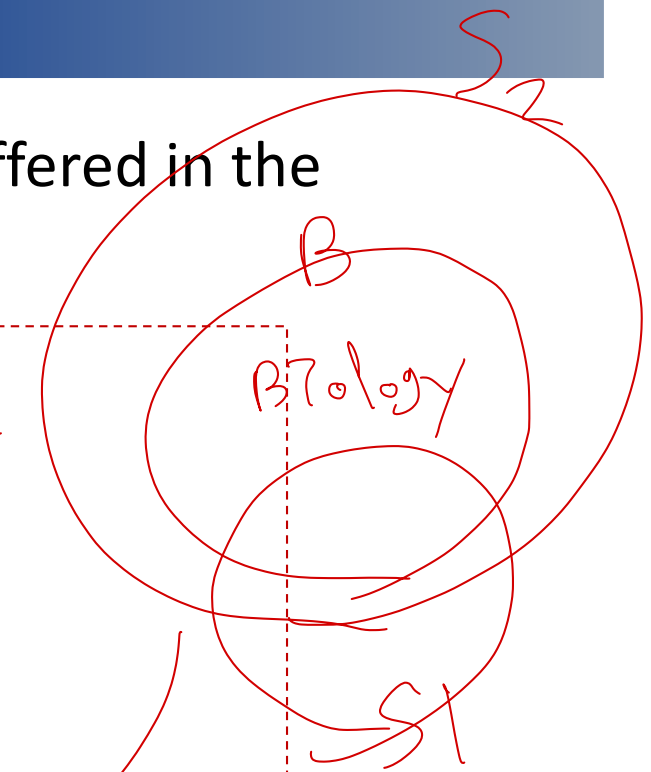
얘네들이 다 1로바뀜

S : **correlation variable**

Test for all: not exists

- Find all students who have taken **all** courses offered in the Biology department.

```
select distinct S.ID, S.name
from student as S
where not exists ( (select course_id
                    from course
                    where dept_name = 'Biology')
                   except
                   (select T.course_id
                    from takes as T
                    where S.ID = T.ID));
```



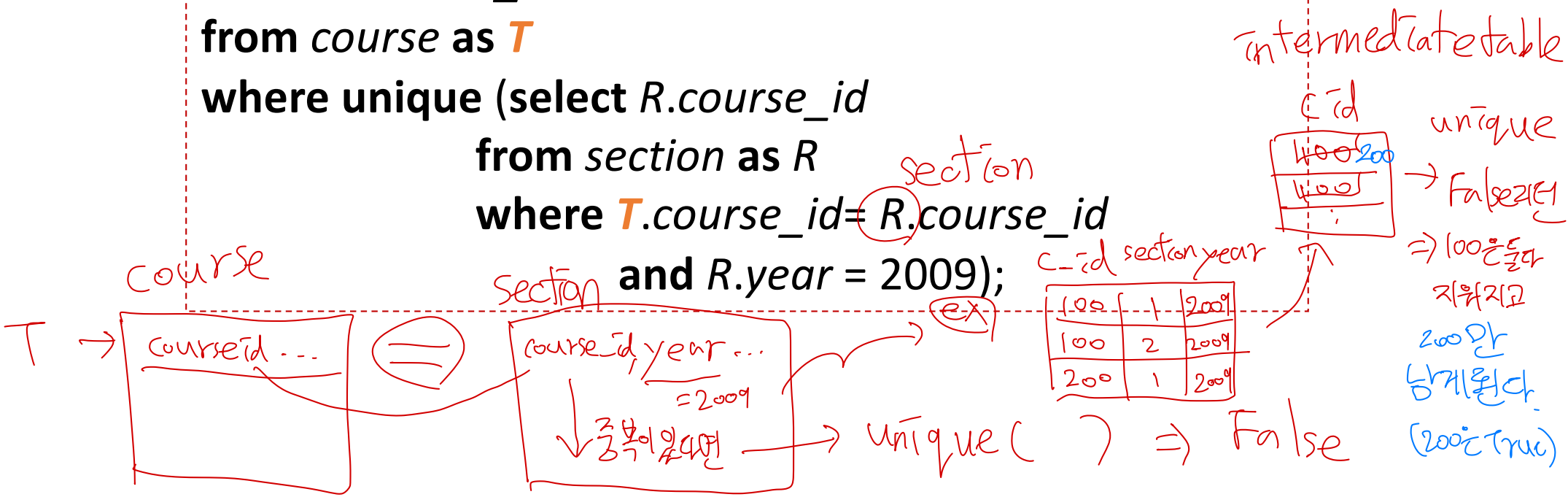
S : correlation variable

- If 'ALL' is included in the query description, consider 'not exists' and 'except'.
- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- *Note:* Cannot write this query using = **all** and its variants

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
  - (Evaluates to "true" on an empty set)
- Find all courses that were offered at most once in 2009

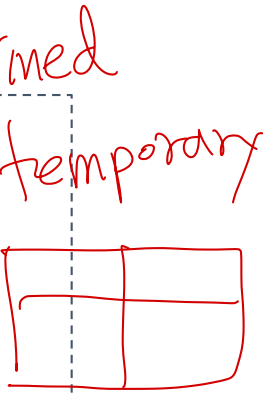
```
select T.course_id
from course as T
where unique (select R.course_id
              from section as R
              where T.course_id= R.course_id
                    and R.year = 2009);
```



Subqueries in the From Clause

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than $42,000.

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
      from instructor
      group by dept_name)
where avg_salary > 42000;
```



or

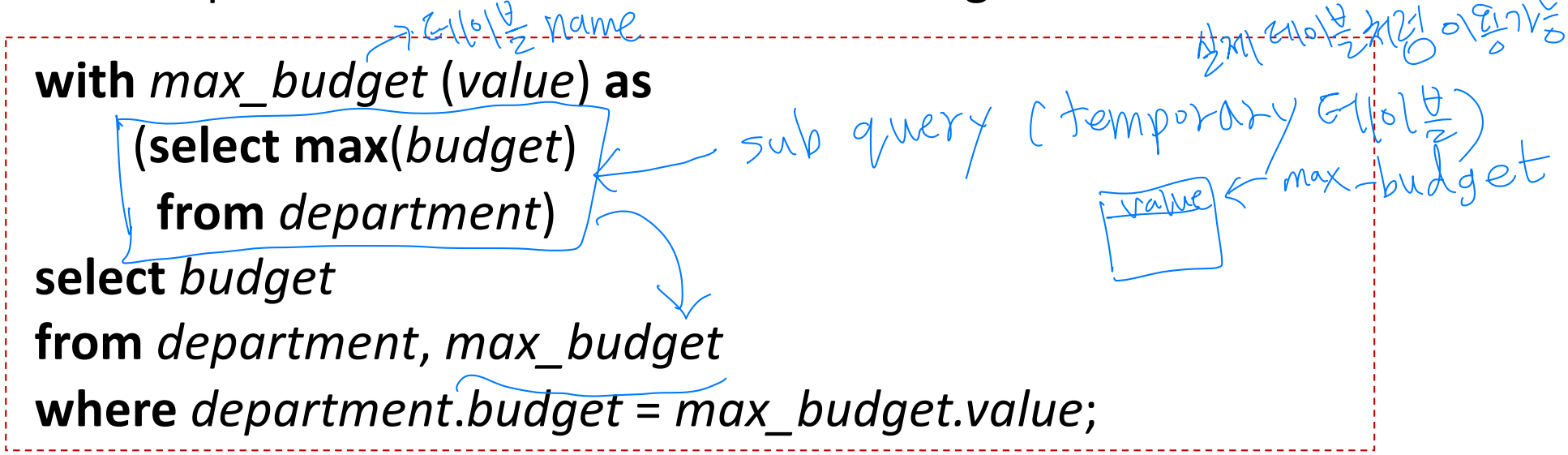
```
with dept_avg(d_name, avg_sal)
as select d_n, avg
   from Inst
   group by dept_name
select dept_name, avg_salary
from (select dept_name, avg (salary)
      from instructor
      group by dept_name)
      as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

```
select ...
from instructor
group by d_n
having avg_sal > 42000
```

with Clause

- The **with** clause provides a way of defining a temporary **view** whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget

```
with max_budget (value) as
        (select max(budget)
          from department)
select budget
from department, max_budget
where department.budget = max_budget.value;
```



Complex Queries using with Clause

- With clause is very useful for writing complex queries
- Supported by most database systems, with minor syntax variations
- Find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept_total (dept_name, value) as
        (select dept_name, sum(salary)
         from instructor
         group by dept_name),
     dept_total_avg(value) as
        (select avg(value)
         from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value >= dept_total_avg.value;
```

Scalar Subquery

- Scalar subquery is used where a single value is expected
- E.g. **select** *dept_name*,
  (**select count**(*)
  from *instructor*
  where *department.dept_name* = *instructor.dept_name*)
  as *num_instructors*
  from *department*;

- E.g. **select** *name*
  from *instructor*
  where *salary * 10 >*
  (**select** *budget* **from** *department*
  where *department.dept_name* = *instructor.dept_name*)

- Runtime error if subquery returns more than one result tuple

Modification of the Database

- Deletion of tuples from a given relation
- Insertion of new tuples into a given relation
- Updating values in some tuples in a given relation

Modification of the Database – delete from

- Delete all instructors

  delete from *instructor*

- Delete all instructors from the Finance department

  delete from *instructor*
  where *dept_name*= 'Finance';

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.

```
delete from instructor
where dept_name in (select dept_name
                    from department
                    where building = 'Watson');
```

delete from (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors

```
delete from instructor
where salary< (select avg (salary) from instructor);
```

- Problem: as we delete tuples from deposit, the average salary changes
- Solution used in SQL:
  1. First, compute **avg** salary and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Modification of the Database – insert into

- Add a new tuple to *course*

```
insert into course
        values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

- or equivalently

```
insert into course (course_id, title, dept_name, credits)
        values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

- Add a new tuple to *student* with *tot_creds* set to null

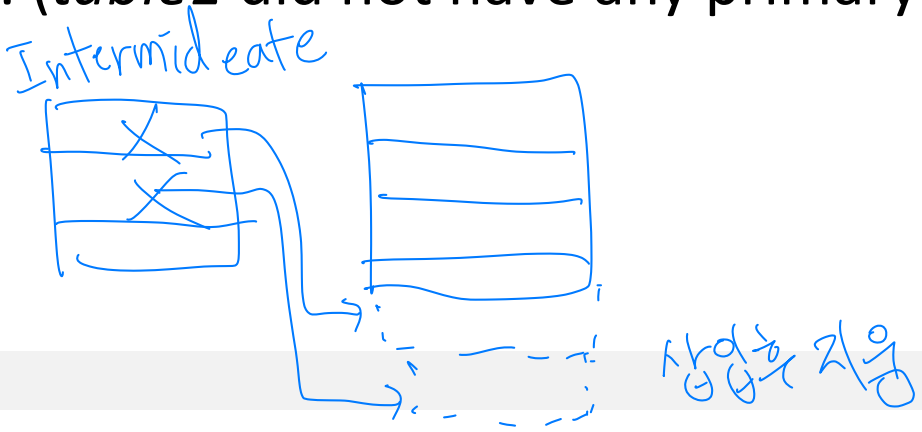
```
insert into student
        values ('3003', 'Green', 'Finance', null);
```

insert into (Cont.)

- Add all instructors to the *student* relation with tot_creds set to 0

```
insert into student
        select ID, name, dept_name, 0
        from   instructor
```

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation
  - otherwise queries like
    insert into *table*1 **select** * **from** *table*1
    would cause problems. (*table1* did not have any primary key defined)



Modification of the Database – update set

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others receive a 5% raise
  - Write two **update** statements:

```
update instructor
    set salary = salary * 1.03
    where salary > 100000;

update instructor
    set salary = salary * 1.05
    where salary <= 100000;
```

  - What if we change the order of the two statements?
    - The order is important
  - Can be done better using the **case** statement (next slide)

case Statement for Conditional Updates

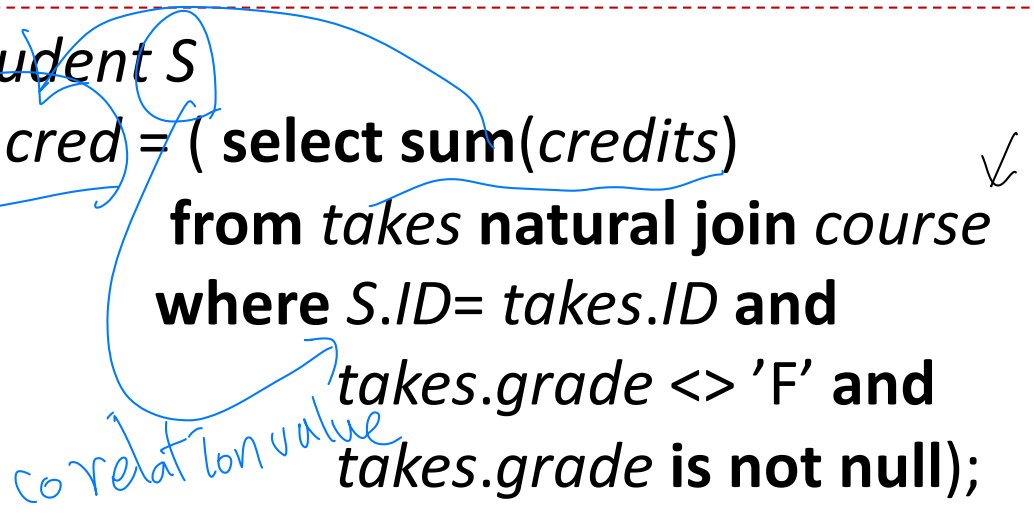
- Same query as before but with case statement

```
update instructor
    set salary = case
                 when salary <= 100000 then salary * 1.05
                 else salary * 1.03
                 end
```

Updates with Scalar Subqueries

- Recompute and update tot_creds value for all students

```
update student S
   set tot_cred = ( select sum(credits)
                    from takes natural join course
                    where S.ID= takes.ID and
                          takes.grade <> 'F' and
                          takes.grade is not null);
```



- Sets *tot_creds* to null for students who have not taken any course
- Instead of **sum**(*credits*), use:

```
case
  when sum(credits) is not null then sum(credits)
  else 0
end
```