Chapter 3 – Part 1 Review (Introduction to SQL (1))

3.2 SQL Data Definition

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

- The schema for each relation
- The domain of values associated with each attribute
- Integrity constraints

SQL **data-manipulation language (DML)** provides the ability to query information, and insert, delete and update tuples

Basic Schema Definition – Create Table

An SQL relation is defined using the **create table** command:

```
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)
```

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

Note: **primary key** declaration on an attribute automatically ensures **not null**

Drop and Alter Table Constructs

drop table *student*

- Deletes the table and its contents

delete from *student (Where절 포함가능)*

- Deletes all contents of table, but retains table
  - e.g. delete from student where dept_name =‘Comp. Sci.’;

alter table

- **alter table** *r* **add** *A D*
  - where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A.*
  - All tuples in the relation are assigned *null* as the value for the new attribute.
- **alter table** *r* **drop** *A*
  - where *A* is the name of an attribute of relation *r*
  - Dropping of attributes not supported by many databases

참고 Integrity Constraints by Alter Table(1)

Add Primary Key by ALTER TABLE

ALTER TABLE *table_name ADD CONSTRAINT*

constraint_name_pk PRIMARY KEY (column1,column2,..column_n);

- ALTER TABLE supplier ADD CONSTRAINT supplier_pk
  PRIMARY KEY (supplier_id); (Supplier 테이블에 supplier_id를 주키로)

Disable Primary Key by ALTER TABLE

ALTER TABLE *table_name DISABLE CONSTRAINT*

constraint_name_pk;

- ALTER TABLE supplier DISABLE CONSTRAINT supplier_pk;

Drop Primary Key by ALTER TABLE

ALTER TABLE *table_name DROP CONSTRAINT*

constraint_name_pk;

- ALTER TABLE supplier DROP CONSTRAINT supplier_pk;

참고 Integrity Constraints by Alter Table(2)

Add Foreign Key by ALTER TABLE

ALTER TABLE *table_name ADD CONSTRAINT*

fk_constraint_name FOREIGN KEY (column1,column2,..,column_n)

REFERENCES parent_table (column1, column2,…,column_n);

- ALTER TABLE products ADD CONSTRAINT fk_supplier
  FOREIGN KEY (supplier_id) REFERECNES supplier (supplier_id);

Disable Foreign Key by ALTER TABLE

ALTER TABLE *table_name DISABLE CONSTRAINT*

fk_constraint_name;

- ALTER TABLE products DISABLE CONSTRAINT fk_supplier;

Drop Foreign Key by ALTER TABLE

ALTER TABLE *table_name DROP CONSTRAINT*

fk_constraint_name;

- ALTER TABLE products DROP CONSTRAINT fk_supplier;

3.3 Basic Structure of SQL Queries

SQL **data-manipulation language (DML)** provides the ability to query information, and insert, delete and update tuples

A typical SQL query has the form:

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

```
SELECT  <attributes>
FROM    <one or more relations>
WHERE   <conditions>
```

A_i represents an attribute

R_i represents a relation

P is a predicate (condition).

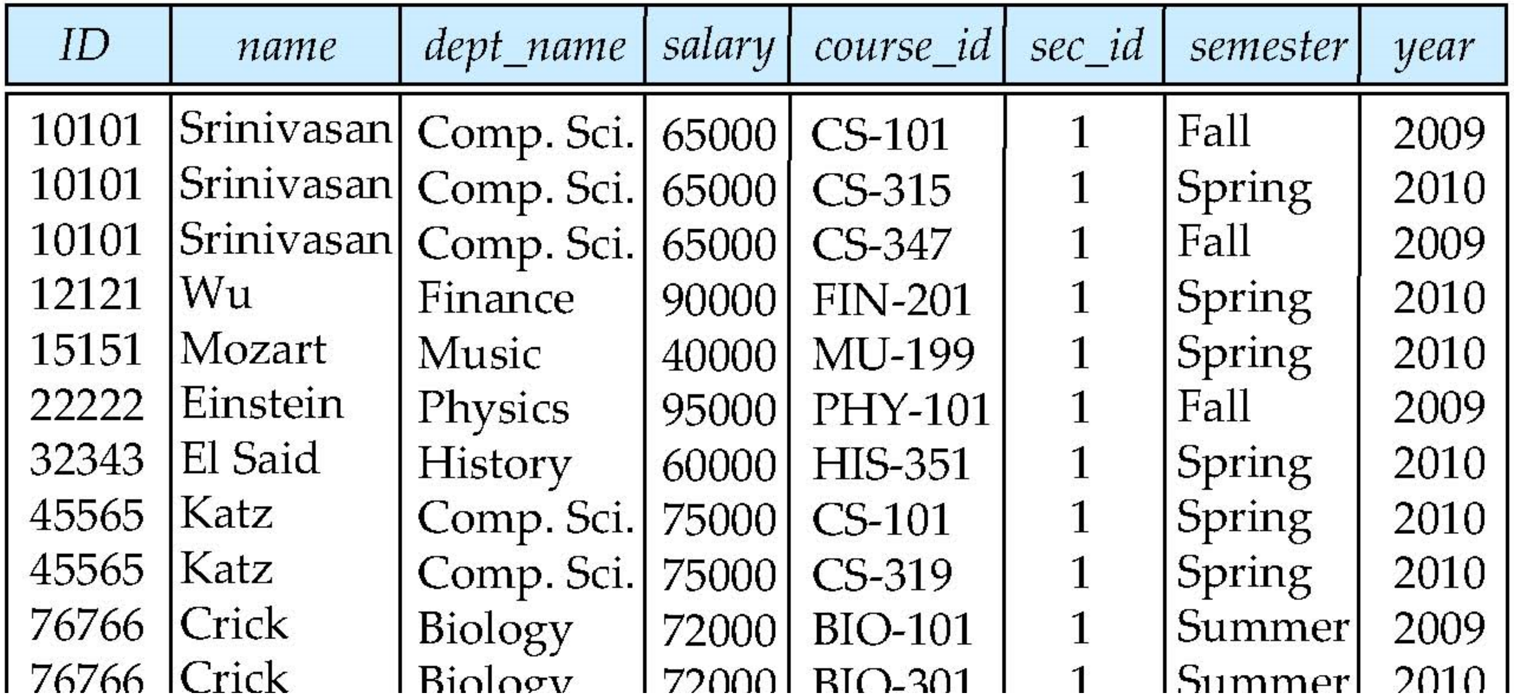
The result of an SQL query is a relation.

Natural Join

Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column

select *
from *instructor* **natura**

1. 자연조인시 하나의 table이 생성된다
2. 조인에 사용된 속성 중 하나는 사라진다(중복되기 때문)
3. 자연조인 결과에 있는 속성을 가리키기 위해 원래의 릴레이션 이름을 포함한 속성이름 사용은 불가! (*instructor.name*)



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

3.4 Additional Basic Operations - Rename Operation

The SQL allows renaming relations and attributes using the **as** clause:

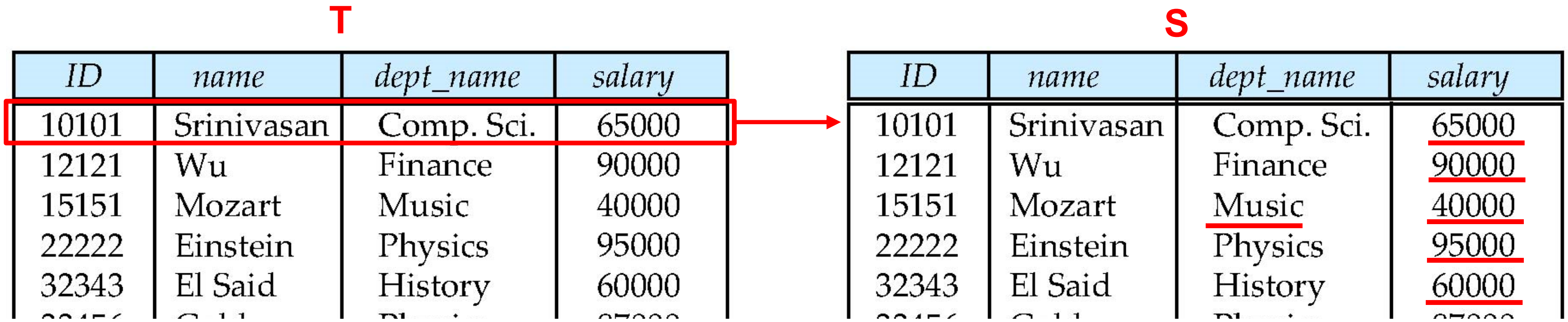
old-name **as** *new-name*

E.g.

select *ID, name, salary/12* **as** *monthly_salary*
from *instructor*

Find the names of all instructors who have a higher salary than some instructor in ‘Comp. Sci’.

select distinct *T. name*
from *instructor* **as** *T, instructor* **as** *S*
where *T.salary* > *S.salary* **and** *S.dept_name* = *‘Comp. Sci.’*



T

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000

S

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000

String Operations

SQL includes a string-matching operator for comparisons on character strings. The operator “like” uses patterns that are described using two special characters:

percent (%). The % character matches any substring.

underscore (_). The _ character matches any character.

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

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

Pattern matching examples:

‘Intro%’ matches any string beginning with “Intro”.

‘%Comp%’ matches any string containing “Comp” as a substring.

‘_ _ _’ matches any string of exactly three characters.

‘_ _ _ %’ matches any string of at least three characters.

Ordering the Display of Tuples

List in alphabetic order the names of all instructors

```
select distinct name
from    instructor
order by name
```

We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.

Example: **order by** *name* **desc**

Can sort on multiple attributes

Example: **order by** *dept_name, name*

Where Clause Predicates

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
```

- where salary >= 90000 and salary <= 100000

Tuple comparison

```
select name, course_id
from instructor, teaches
where (instructor.ID, dept_name) = (teaches.ID, ’Biology’);
```

- where instructor.ID = teaches.ID and dept_name = ‘Biology’

참고: not between 연산자도 있음

3.5 Set Operations

Find courses that ran in Fall 2009 or in Spring 2010

(**select** *course_id* **from** *section* **where** *sem* = ‘Fall’ **and** *year* = 2009)
union
(**select** *course_id* **from** *section* **where** *sem* = ‘Spring’ **and** *year* = 2010)

Find courses that ran in Fall 2009 and in Spring 2010

(**select** *course_id* **from** *section* **where** *sem* = ‘Fall’ **and** *year* = 2009)
intersect
(**select** *course_id* **from** *section* **where** *sem* = ‘Spring’ **and** *year* = 2010)

Find courses that ran in Fall 2009 but not in Spring 2010

(**select** *course_id* **from** *section* **where** *sem* = ‘Fall’ **and** *year* = 2009)
except
(**select** *course_id* **from** *section* **where** *sem* = ‘Spring’ **and** *year* = 2010)

MySQL에서는 **except,** 오라클에서는 **MINUS**

Chapter 3: Introduction to SQL (2)

Revision by Gun-Woo Kim

Dept. of Computer Science and Engineering

Hanyang University

Database System Concepts, 6th Ed.

Contents

3.6 Null Values

3.7 Aggregate Functions

3.8 Nested Subqueries

3.9 Modification of the Database

3.6 Null Values

It is possible for tuples to have a null value, denoted by *null*, for some of their attributes

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.

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

잠깐, "salary = null" 과연 유효한 표현인가 ?
SQL에서 사용가능한가?

null은 값을 모르거나 존재하지 않을 경우를 의미함. So, salary = null은 salary 값이 null이라는 것을 알고 있음을 의미. 의미적으로 = 은 맞지 않음. Oracle에서는 질의 자체는 인식 하지만, 결과를 출력하지는 못함

Null Values and Three Valued Logic

true, false, unknown

이때 5<null은 불값이아니라
unknown리턴

Any comparison with *null* returns *unknown*

Example*: 5 < null or null <> null or null = null*

Three-valued logic using the *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*

3.7 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

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 2010 semester

```
select count (distinct ID)
from teaches
where semester = ’Spring’ and year = 2010
```

Find the number of tuples in the *course* relation

이때 count(distinct *)는 불가하고count(distinct <attr name>)은 가능

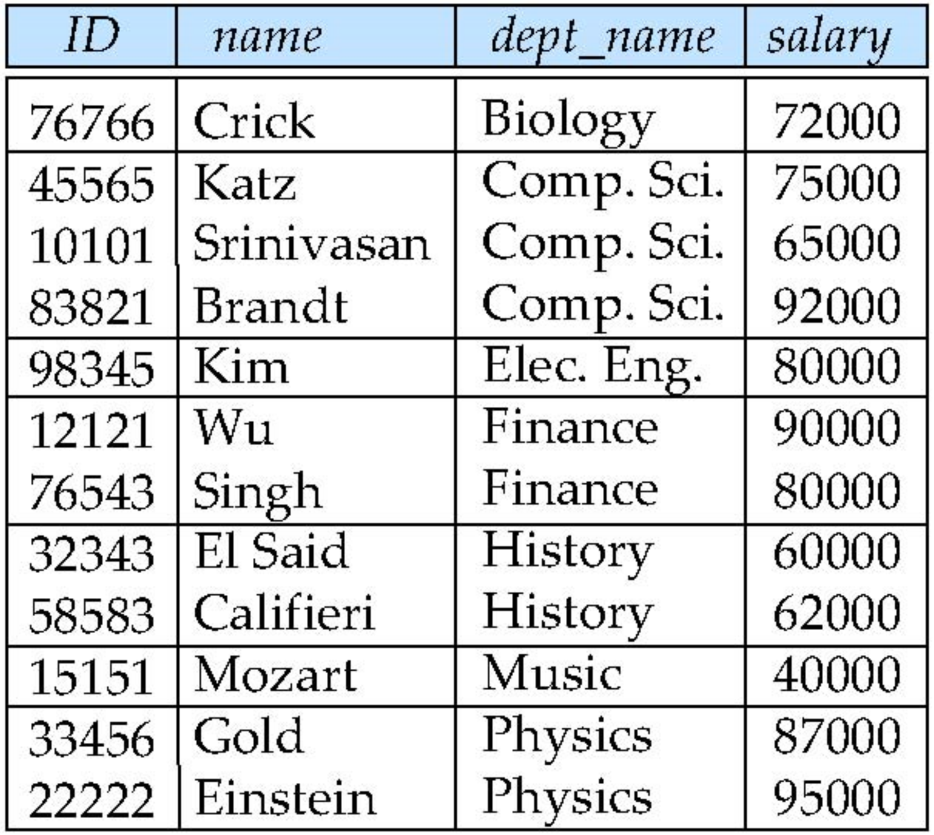
```
select count (*)
from course;
```

Aggregate Functions – Group By

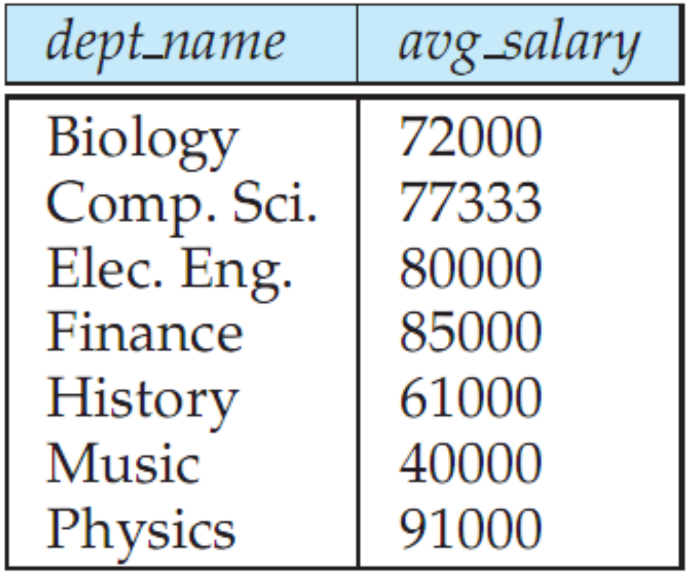
Find the average salary of instructors in each department

select *dept_name*, **avg** (*salary*)
from *instructor*
group by *dept_name*;

Note: departments with no instructor will not appear in result



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

Aggregate Functions – Group By (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;
```

dept_name으로 그룹화되면 한 그룹 안에
여러 교수 ID가 존재함
→ 그룹의 세부 가지수를 표현할 수 없음
→ 그래서, select절에 집계함수(avg) 바깥쪽에
있는 attribute들이 group by 뒤에도 나와야 한다.

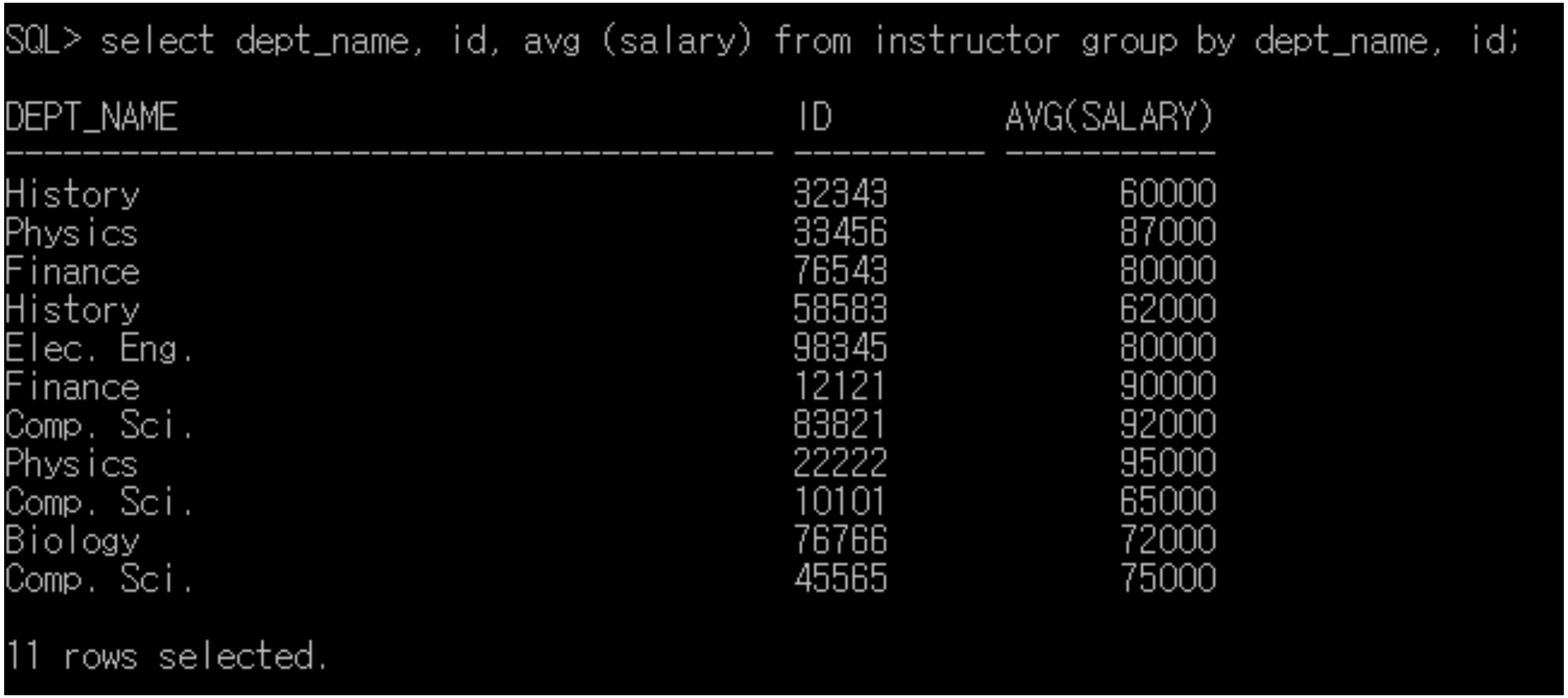
즉, 그룹으로 묶이지 않은 속성들도 집계함수를 이용해 값을 산출하는 경우가 아니면 같이 group해주어야함

Aggregate Functions – Group By (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, ID*;

select절에 집계함수(avg) 바깥쪽에 있는 attribute들이 group by 뒤에도 나와야 한다.



```
SQL> select dept_name, id, avg (salary) from instructor group by dept_name, id;

DEPT_NAME                                ID         AVG(SALARY)
---------------------------------------- ---------- -----------
History                                  32343            60000
Physics                                  33456            87000
Finance                                  76543            80000
History                                  58583            62000
Elec. Eng.                               98345            80000
Finance                                  12121            90000
Comp. Sci.                               83821            92000
Physics                                  22222            95000
Comp. Sci.                               10101            65000
Biology                                  76766            72000
Comp. Sci.                               45565            75000

11 rows selected.
```

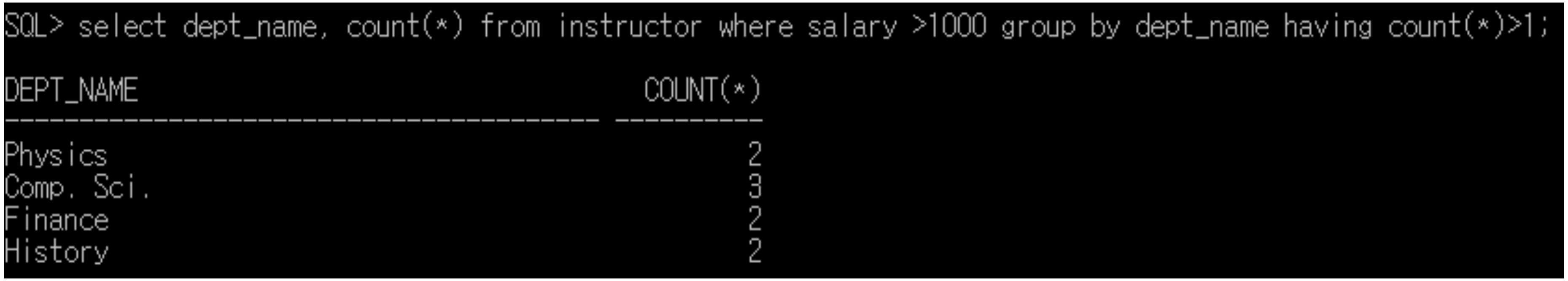
Aggregate Functions – Having Clause

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;

select dept_name,count(*) from instructor
where salary> 100
group by dept_name having avg(salary)>42000;
인데 where은 groupby전에 실행됨

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



```
SQL> select dept_name, count(*) from instructor where salary >1000 group by dept_name having count(*)>1;

DEPT_NAME                                  COUNT(*)
---------------------------------------- ----------
Physics                                           2
Comp. Sci.                                        3
Finance                                           2
History                                           2
```

Null Values and Aggregates

Total all salaries

```
select sum (salary )
from instructor
```

Above statement ignores null amounts

Result is *null* if there is no non-null amount

결국 모두 null 이면

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

What if collection has only null values?

→ empty collection인 경우

count returns 0

all other aggregates return null

3.8 Nested Subqueries

SQL provides a mechanism for the nesting of subqueries.

A **subquery** is a **select-from-where** expression that is nested within another query.

A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

Nested Subqueries - Set Membership

Find courses offered in Fall 2009 and in Spring 2010

```
select distinct course_id
from section
where semester = ’Fall’ and year= 2009 and
        course_id   in (select course_id
                        from section
                        where semester = ’Spring’ and year= 2010);
```

Find courses offered in Fall 2009 but not in Spring 2010

```
select distinct course_id
from section
where semester = ’Fall’ and year= 2009 and
        course_id   not in (select course_id
                        from section
                        where semester = ’Spring’ and year= 2010);
```

Nested Subqueries - 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’;
```

name	salary	Department
Lee	5000	Biology
Kim	10000	Biology
Park	7000	Computer

Same query using > **some** clause

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

결과는 ?   → (5000, 10000)

Definition of some Clause

F <comp> **some** *r* ⇔ ∃ *t* ∈ *r* such that (F <comp> *t*)
Where <comp> can be: $<, \ \leq, \ >, \ =, \ \neq$

(5 < **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}$) = true

(read: 5 < some tuple in the relation)

(5 < **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$) = false

(5 = **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$) = true

(5 ≠ **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$) = true (since 0 ≠ 5)

Definition of all Clause

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

$(5 < \textbf{all} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{false}$

$(5 < \textbf{all} \begin{array}{|c|}\hline 6 \\ \hline 10 \\ \hline \end{array}) = \text{true}$

$(5 = \textbf{all} \begin{array}{|c|}\hline 4 \\ \hline 5 \\ \hline \end{array}) = \text{false}$

$(5 \neq \textbf{all} \begin{array}{|c|}\hline 4 \\ \hline 6 \\ \hline \end{array}) = \text{true (since } 5 \neq 4 \text{ and } 5 \neq 6)$

Example Query of all Clause

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’);
```

Test for Empty Relations

The **exists** construct returns the value **true** if the argument subquery is nonempty.

exists $r \Leftrightarrow r \neq \emptyset$

not exists $r \Leftrightarrow r = \emptyset$

Exists

Yet another way of specifying the query "Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester"

바깥질의

```
select course_id
from section as S
where semester = 'Fall' and year= 2009 and
        exists (select *
                from section as T
                where semester = 'Spring' and year= 2010
                        and S.course_id= T.course_id);
```

correlated

하위질의

Correlated subquery

Correlation name or **correlation variable**

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));
```

생물학과에서 개설된 모든 수업 → X

학생 S.ID가 수강하는 모든 수업 → Y

Note that $X - Y = \varnothing \iff X \subseteq Y$

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

기껏해야 한번 → 0 포함?

```
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);
```

unique (결과가 없거나 하나인 경우) → true

unique 를 쓰지 않고 표현하면 ?

```
...........
where 1 >= (select count(R.course_id)
                 from section as R
                 where T.course_id= R.course_id
                            and R.year = 2009);
```

note: 교재(pp. 95)에 오류 있음, not unique 도 있음

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;
```

select-from-where 결과가 결국 또 다른 relation이기 때문에 가능한 것임

Note that we do not need to use the **having** clause → 바깥쪽에 where 절 때문

Another way to write above query

```
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;
```

from 절 안에 있는 select 구문의 결과를 새로운 relation (즉, dept_avg)로 재명명 → Oracle에서는 지원되지 않음

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;
```

with절을 포함한 질의에서만 유효한 임시 relation (*max_budget*) 생성

Scalar Subquery

Scalar subquery is one which 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;
```

Subqeury의 결과가 table이 아니라 value인 경우

질의의 의미 ?   → 학과별 교수님의 수를 출력

3.9 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 – Deletion

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’);
```

Deletion (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 instructor, 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 – Insertion

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);
```

Insertion (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
```

insert 수행전에 select가
먼저 수행되어야 함

The **select from where** statement is evaluated fully before any of its results are inserted into the relation (otherwise queries like

```
insert into table1 select * from table1
```

would cause problems, if *table1* did not have any primary key defined)

만약 select from where 구문을 수행하면서 insert문을 수행하면 어떻게 될까?
즉, select를 해서 하나의 tuple을 뽑고 이를 다시 같은 table에 insert를 하고,
주키 제약조건마저 없다면 어떻게 될까 ?

→ 중복된 tuple들이 계속해서 들어가고 결국 무한 loop에 빠진다.

Modification of the Database – Updates

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
①  ┤ set salary = salary * 1.03
    └ where salary > 100000;

    ┌ update instructor
②  ┤ set salary = salary * 1.05
    └ where salary <= 100000;
```

②→① 순서로 수행되면 ?

$100000 보단 약간 적은 연봉
(즉, 5% 인상하면 $100000가
넘는 연봉)을 받는 사람들은
다시 3% 더 인상됨 → 본래 의도?

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
```

else 문이 없어도 실행되는데
if조건을 충족못하면 null값으로 처리

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 0 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
```

어떤 course도 듣지 않은 학생의 tot_creds는 null이 아니라 0 이어야 함