Database

Lecture 4-2. Structured Query Language (Part 2)

Spring 2024

Prof. Jik-Soo Kim, Ph.D.

E-mail: jiksoo@mju.ac.kr

Notes

- **Readings**
  - Chapter 3: Introduction to SQL (Database System Concepts 7th Edition)

DATA MANIPULATION LANGUAGE

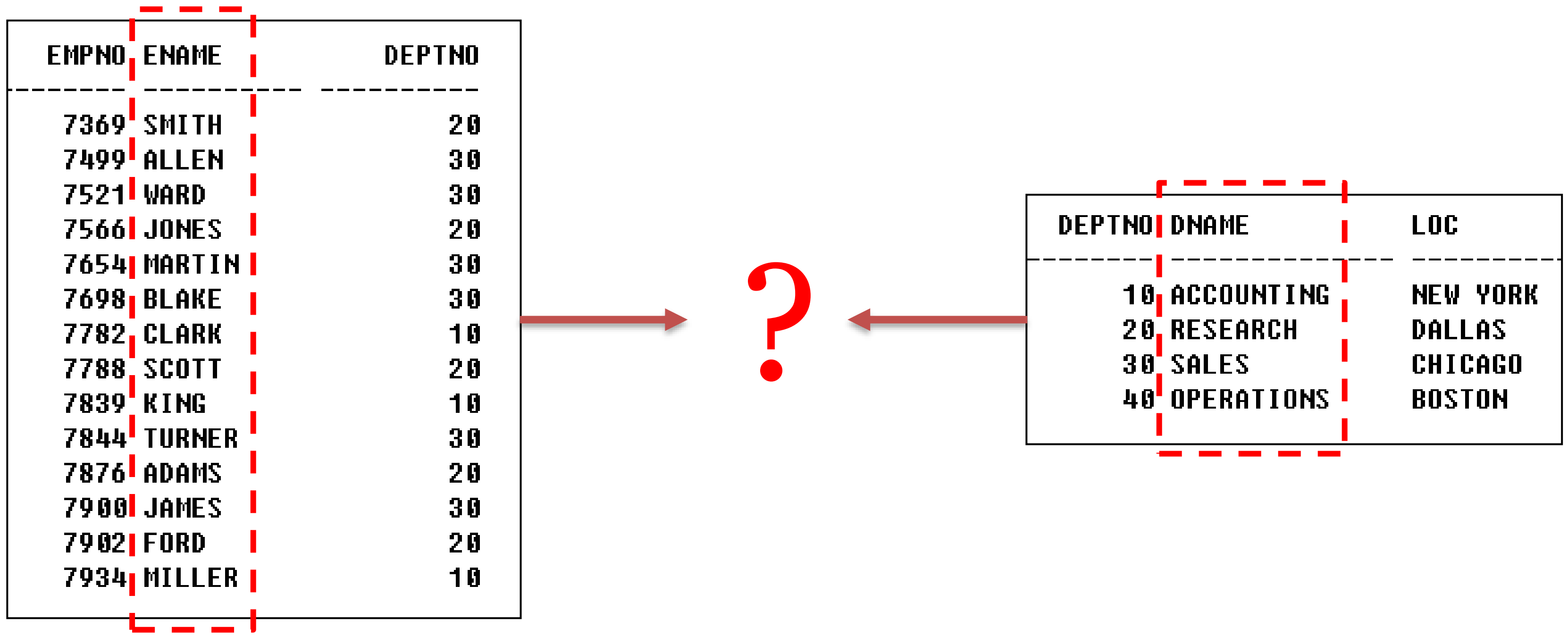
Data Manipulation Language (DML)

- SELECT
- DELETE
- INSERT
- UPDATE

JOIN OPERATIONS

Join Operation

- 두개 이상의 테이블을 합쳐서 하나의 큰 테이블로 만드는 방법
- 필요성
  - 관계형 모델에서는 데이터의 일관성이나 효율을 위하여 데이터의 중복을 최소화("**정규화**") → 주로 **Foreign Key**를 이용하여 참조
  - 정규화된 테이블로부터 결합된 형태의 정보를 추출할 필요가 있음
  - **예) 직원의 이름과 직원이 속한 부서명을 함께 보고 싶으면???**

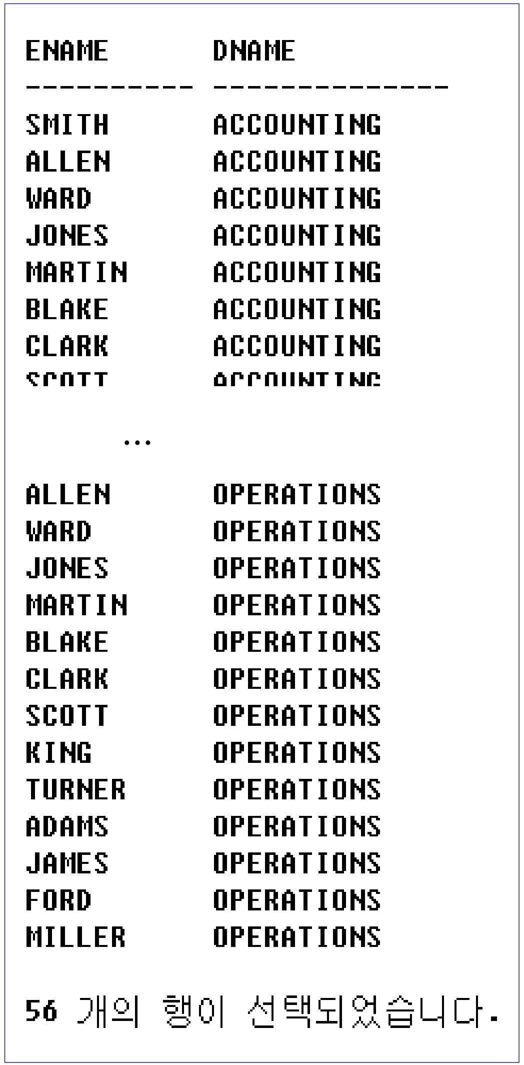


```
 EMPNO ENAME          DEPTNO
------ ---------- ----------
  7369 SMITH              20
  7499 ALLEN              30
  7521 WARD               30
  7566 JONES              20
  7654 MARTIN             30
  7698 BLAKE              30
  7782 CLARK              10
  7788 SCOTT              20
  7839 KING               10
  7844 TURNER             30
  7876 ADAMS              20
  7900 JAMES              30
  7902 FORD               20
  7934 MILLER             10
```

```
DEPTNO DNAME          LOC
------ -------------- ----------
    10 ACCOUNTING     NEW YORK
    20 RESEARCH       DALLAS
    30 SALES          CHICAGO
    40 OPERATIONS     BOSTON
```

Cartesian Product

- 두 테이블에서 그냥 결과를 선택하면?
  - SELECT ename, dname FROM emp, dept
  - 결과: 두 테이블의 행들의 가능한 모든 쌍이 추출됨
  - <u>일반적으로 사용자가 원하는 결과가 아님</u>



```
ENAME      DNAME
---------- --------------
SMITH      ACCOUNTING
ALLEN      ACCOUNTING
WARD       ACCOUNTING
JONES      ACCOUNTING
MARTIN     ACCOUNTING
BLAKE      ACCOUNTING
CLARK      ACCOUNTING
SCOTT      ACCOUNTING

      ...

ALLEN      OPERATIONS
WARD       OPERATIONS
JONES      OPERATIONS
MARTIN     OPERATIONS
BLAKE      OPERATIONS
CLARK      OPERATIONS
SCOTT      OPERATIONS
KING       OPERATIONS
TURNER     OPERATIONS
ADAMS      OPERATIONS
JAMES      OPERATIONS
FORD       OPERATIONS
MILLER     OPERATIONS

56 개의 행이 선택되었습니다.
```

- Cartesian Product

$$X \times Y = \{(x, y) | x \in X \text{ and } y \in Y\}$$

- Cartesian Product를 막기 위해서는 <u>올바른 **Join 조건**을 **WHERE** 절에 부여</u>해야 함

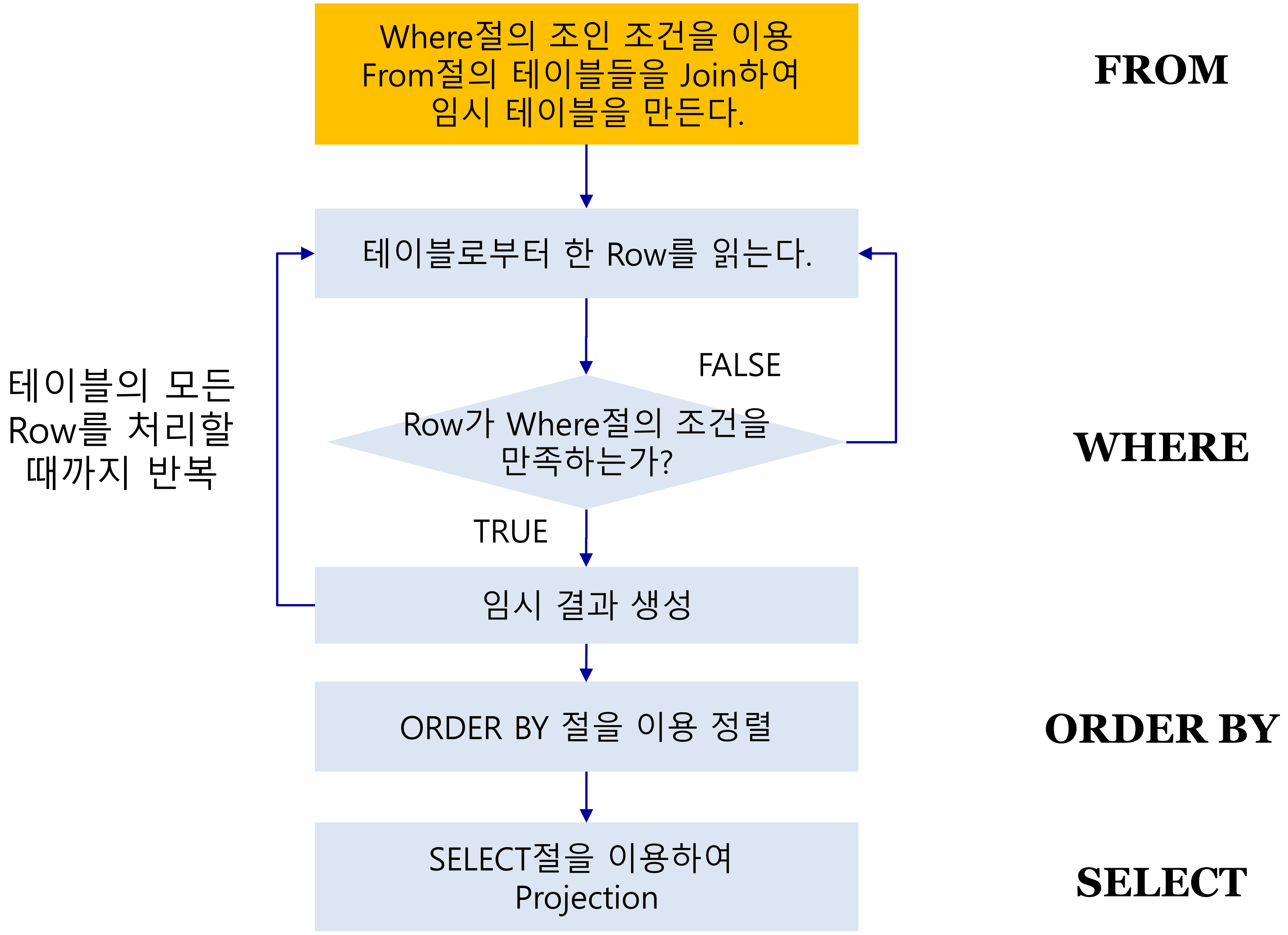
Simple Join

- Syntax

```
SELECT t1.col1, t1.col2, t2.col1 …
FROM Table1 t1, Table2 t2
WHERE t1.col3 = t2.col3
```

- 설명
  - **FROM** 절에 필요로 하는 테이블을 모두 적는다
  - 컬럼 이름의 모호성을 피하기 위해 Table 이름에 Alias 사용 가능 (테이블 이름으로 직접 지칭 가능)
  - 적절한 **Join 조건**을 **WHERE** 절에 부여 (일반적으로 "테이블 개수 -1" 개의 조인 조건이 필요하며, **PK**와 **FK**간의 = 이 붙는 경우가 많음)

Join 처리 개념



Where절의 조인 조건을 이용 From절의 테이블들을 Join하여 임시 테이블을 만든다.
FROM
테이블로부터 한 Row를 읽는다.
테이블의 모든 Row를 처리할 때까지 반복
FALSE
Row가 Where절의 조건을 만족하는가?
WHERE
TRUE
임시 결과 생성
ORDER BY 절을 이용 정렬
ORDER BY
SELECT절을 이용하여 Projection
SELECT

실제 모든 SQL이 이렇게 처리되는 것은 아닙니다. SQL의 처리 순서는 DBMS가 질의 최적화 과정을 통하여 결정합니다. 질의의 종류, 데이터의 분포 등에 따라 질의의 실제 순서는 달라질 수도 있습니다.

Join 종류

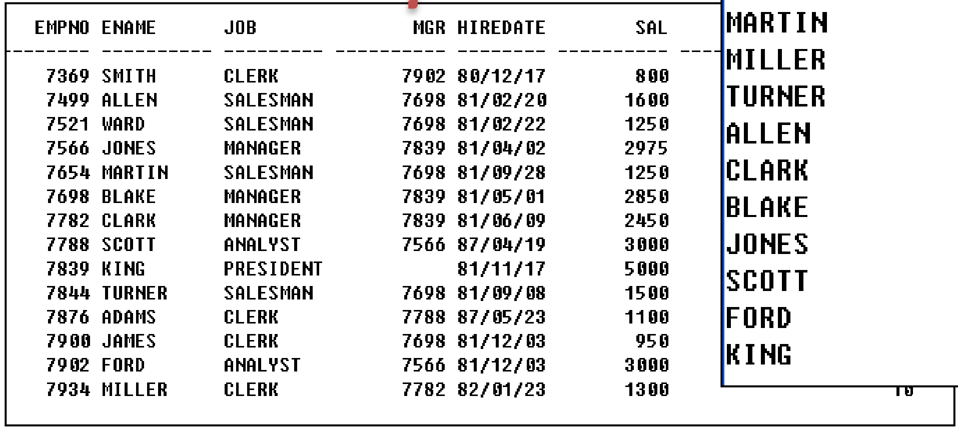
- **용어**
  - Cross Join (Cartesian Product): 모든 가능한 쌍이 나타남
  - Inner Join: Join 조건을 만족하는 튜플만 나타남
  - Outer Join: Join 조건을 만족하지 않는 튜플(짝이 없는 튜플)도 Null과 함께 나타남 ⟗ ⟖ ⟕
  - Theta Join: 임의의 조건(theta)에 의한 조인 $r \bowtie_\theta s = \sigma_{\underline{\theta}}(r \times s)$
  - Equi-Join: Theta Join & 조건이 Equal (=)
  - Natural Join: Equi-join & 동일한 Column명 합쳐짐 └조건
  - Self Join: 자기 자신과 조인

Theta Join

- 정의
  - 임의의 조건을 Join 조건으로 사용 (Equi-Join도 Theta Join의 한 형태)
  - "="외의 조건 사용 시 Non-Equi Join이라고도 함
- 예)

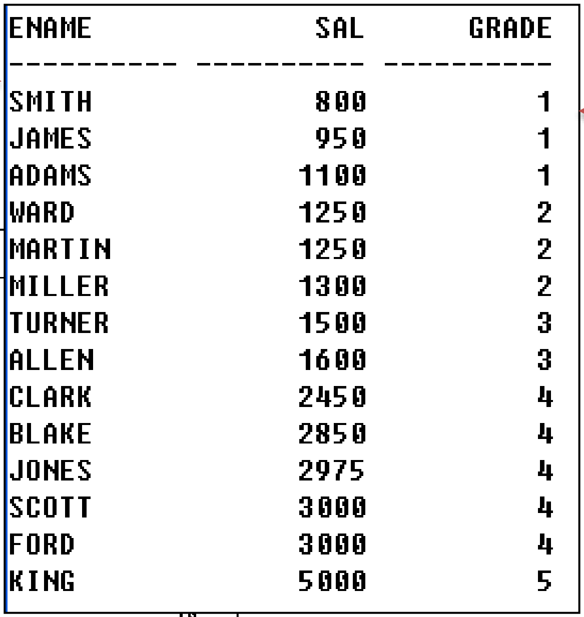
```
SELECT e.ename, e.sal, s.grade
FROM emp e, salgrade s × 연산
WHERE e.sal BETWEEN s.losal AND s.hisal
```

s.losal이상 s.hisal이하

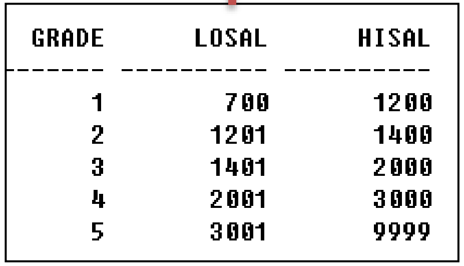


```
 EMPNO ENAME      JOB              MGR HIREDATE        SAL
------ ---------- --------- ---------- -------- ---------- ---
  7369 SMITH      CLERK           7902 80/12/17        800
  7499 ALLEN      SALESMAN        7698 81/02/20       1600
  7521 WARD       SALESMAN        7698 81/02/22       1250
  7566 JONES      MANAGER         7839 81/04/02       2975
  7654 MARTIN     SALESMAN        7698 81/09/28       1250
  7698 BLAKE      MANAGER         7839 81/05/01       2850
  7782 CLARK      MANAGER         7839 81/06/09       2450
  7788 SCOTT      ANALYST         7566 87/04/19       3000
  7839 KING       PRESIDENT            81/11/17       5000
  7844 TURNER     SALESMAN        7698 81/09/08       1500
  7876 ADAMS      CLERK           7788 87/05/23       1100
  7900 JAMES      CLERK           7698 81/12/03        950
  7902 FORD       ANALYST         7566 81/12/03       3000
  7934 MILLER     CLERK           7782 82/01/23       1300                  10
```

emp



```
ENAME             SAL      GRADE
---------- ---------- ----------
SMITH             800          1
JAMES             950          1
ADAMS            1100          1
WARD             1250          2
MARTIN           1250          2
MILLER           1300          2
TURNER           1500          3
ALLEN            1600          3
CLARK            2450          4
BLAKE            2850          4
JONES            2975          4
SCOTT            3000          4
FORD             3000          4
KING             5000          5
```



```
 GRADE      LOSAL      HISAL
------ ---------- ----------
     1        700       1200
     2       1201       1400
     3       1401       2000
     4       2001       3000
     5       3001       9999
```

salgrade

Equi-Join

theta join이면서 동시에 이퀴조인

EMPNO	ENAME	……	DEPTNO
7839	KING		10
7566	JONES		20
7900	JAMES		30
7369	SMITH		20
7499	ALLEN		30

EMP **FK**

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATION	BOSTON

PK **DEPT**

=

EMPNO	ENAME	……	DEPTNO	DEPTNO	DNAME	LOC
7839	KING		10	10	ACCOUNTING	NEW YORK
7566	JONES		20	20	RESEARCH	DALLAS
7900	JAM					CHICAGO
7369	SMI					DALLAS
7499	ALLEN		30	30	SALES	CHICAGO

```
SELECT * FROM EMP, DEPT
WHERE EMP.DEPTNO = DEPT.DEPTNO
```

MYONGJI UNIVERSITY

Outer Join

- 정의
  - Join 조건을 만족하지 않는(짝이 없는) 튜플의 경우 Null을 포함하여 결과를 생성
  - 모든 행이 결과 테이블에 나타남

- 종류
  - **Left** Outer Join: 왼쪽의 모든 튜플이 결과 테이블에 나타남
  - **Right** Outer Join: 오른쪽의 모든 튜플이 결과 테이블에 나타남
  - **Full** Outer Join: 양쪽 모두 결과 테이블에 나타남

- 표현 방법
  - NULL이 올수 있는 쪽 조건에 (+)를 붙인다(오라클)

Outer Join

EMP (FK: DEPTNO)

EMPNO	ENAME	……	DEPTNO
7839	KING		10
7566	JONES		20
7900	JAMES		30
7369	SMITH		20

DEPT (PK: DEPTNO)

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATION	BOSTON

40에 해당하는건 EMP에 없어 원래 이퀴조인에선
안나오지만 +를 통해 나온다.
이거같은 경우 right area

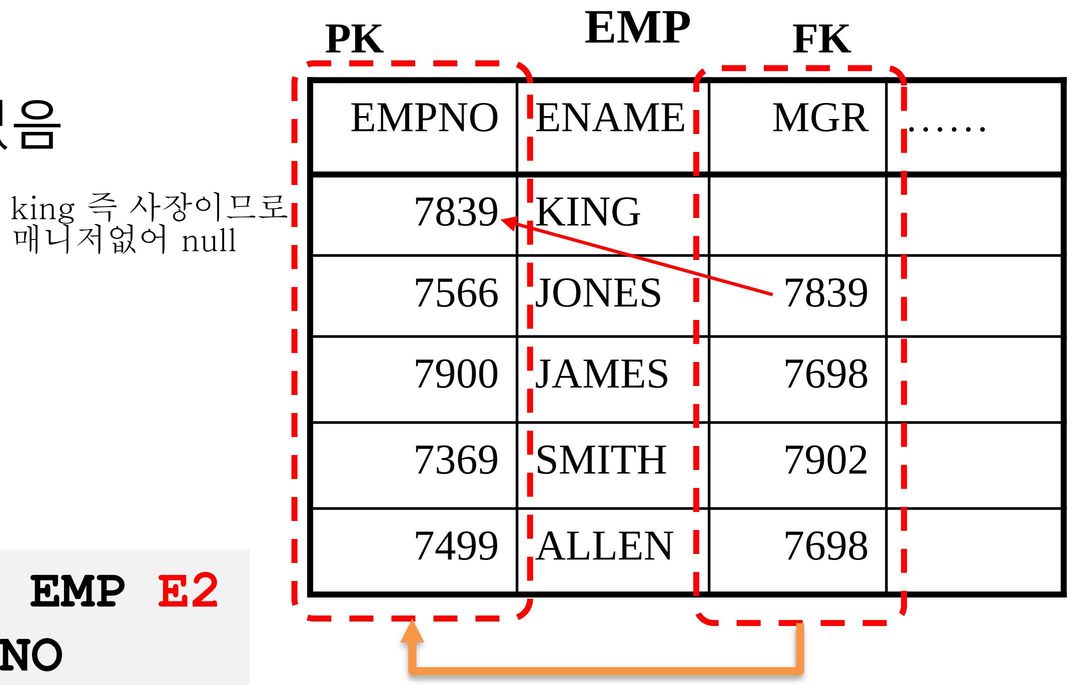
EMPNO	ENAME	……	DEPTNO	DEPTNO	DNAME	LOC
7839	KING		10	10	ACCOUNTING	NEW YORK
7566	JONES					S
7900	JAMES		30	30	SALES	CHICAGO
7369	SMITH		20	20	RESEARCH	DALLAS
7499	ALLEN		30	30	SALES	CHICAGO
				40	OPERATION	BOSTON

```
SELECT * FROM EMP, DEPT
WHERE EMP.DEPTNO (+)= DEPT.DEPTNO
```

MYONGJI UNIVERSITY

Self Join

- 자기자신과 Join
- Alias를 사용할 수 밖에 없음



EMPNO	ENAME	MGR	……
7839	KING		
7566	JONES	7839	
7900	JAMES	7698	
7369	SMITH	7902	
7499	ALLEN	7698	

```
SELECT * FROM EMP E1, EMP E2
WHERE E1.MGR = E2.EMPNO
```

EMPNO	ENAME	MGR	……	EMPNO	ENAME
7566	JONES	7839		7839	KING
7900	JAMES	7698		7698	BLAKE
7369	SMITH	7902		7902	FORD
7499	ALLEN	7698		7698	BLAKE

SQL:1999 Syntax

- From절에서 바로 Join을 명시적으로 정의

```
SELECT table1.column, table2.column
FROM table1
[CROSS JOIN table2] |
[NATURAL JOIN table2] |
[JOIN table2 USING (column_name)] |
[JOIN table2
ON(table1.column_name = table2.column_name)] |
[LEFT|RIGHT|FULL OUTER JOIN table2
ON (table1.column_name = table2.column_name)];
```

- 예)
  - SELECT * FROM emp **NATURAL JOIN** dept;
  - SELECT * FROM emp **JOIN** dept **USING** (deptno);
  - SELECT * FROM emp **JOIN** dept **ON** emp.deptno = dept.deptno;
  - SELECT * FROM emp **RIGHT OUTER JOIN** dept **ON** (emp.deptno = dept.deptno);

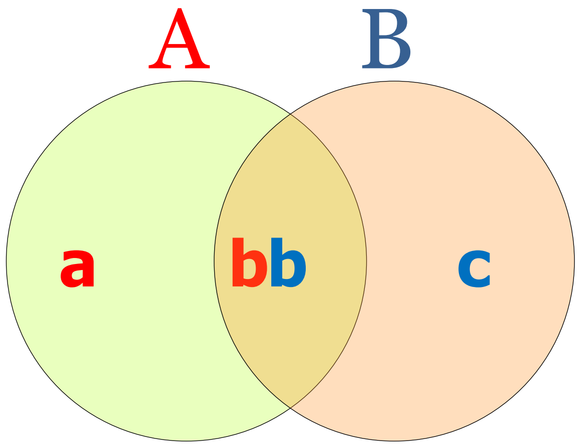
같은 결과값을 낸다.

on이 using보다 좀더 상위개념이라보면 됨.(using은 equal밖에 못 씀)

SET OPERATIONS

SET Operator

- 두 질의의 결과를 가지고 집합 연산(합집합, 교집합, 차집합 등)
- UNION, UNION ALL, INTERSECT, MINUS, …



- A UNION B = {a, b, c} 중복제거
- A UNION ALL B = {a, b, b, c}
- A INTERSECT B = {b}
- A MINUS B = {a}

```
SELECT ename FROM emp
UNION
SELECT dname FROM dept;
```

안먹힐수도 있다.(IDE에 따라)

* The **MINUS** operator is not supported in all SQL databases!
It can be used in Oracle. For databases such as SQL Server, PostgreSQL, and SQLite, use the **EXCEPT** operator to perform this type of query.

Set Operations

- Set operations **union**, **intersect**, and **except** 합, 교, 차
  - each of the operations automatically eliminates duplicates

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

- Suppose a tuple occurs *m* times in *r* and *n* times in *s,* then it occurs:
  - *m* + *n* times in *r* **union all** *s*
  - min(*m, n)* times in *r* **intersect all** *s*
  - max(0, *m* – *n)* times in *r* **except all** *s*

Set Operations

- Find courses(*course_id*) that ran in Fall 2009 *or* in Spring 2010

  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2009)
  union
  string은 작은따옴표!
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2010)
  2010년 &봄학기에 단하나라도 열린 강좌의 강좌번호

- Find courses(*course_id*) 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(*course_id*) 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)

Set Operations

- Find the salaries of all instructors that are less than the largest salary

  누군가보다는 연봉이 작은 값

  – **select distinct** *T.salary*
    from *instructor* **as** *T, instructor* **as** *S*
    where *T.salary < S.salary*

  rename을 통해 같은 두집합 합치기

- Find all the salaries of all instructors
  – **select distinct** *salary*
    from *instructor*

- Find the largest salary of all instructors
  – ("second query")
    except
    ("first query")

  전체−누군가보다는연봉이 작은 값 = 누구보다 작지않은값
  = 가장 큰 값

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 a value that does not exist
- The result of any arithmetic expression involving *null* is *null*
  - e.g., 5 + *null* returns *null*
- The predicate **is null** can be used to check for null values
  - e.g., Find all instructors whose salary is *null*

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

  cf. = null로 쓰면 unknown이라는 다른 vaule값이 나오기때문에 쓰면 안된다.

Null Values and Three Valued Logic

- Three values – *true*, *false*, *unknown*
- Any comparison with *null* returns *unknown*
  - e.g., *5 < null* or *null <> null* or *null = null*
- Three-valued logic using the 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*

- **Result of where clause predicate is treated as *false* if it evaluates to *unknown*** 즉, unknown은 false로 가정한다.

AGGREGATE FUNCTIONS

Aggregate Functions

- Aggregate functions operate on the multiset of values of a column of a relation, and return a *single* value

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

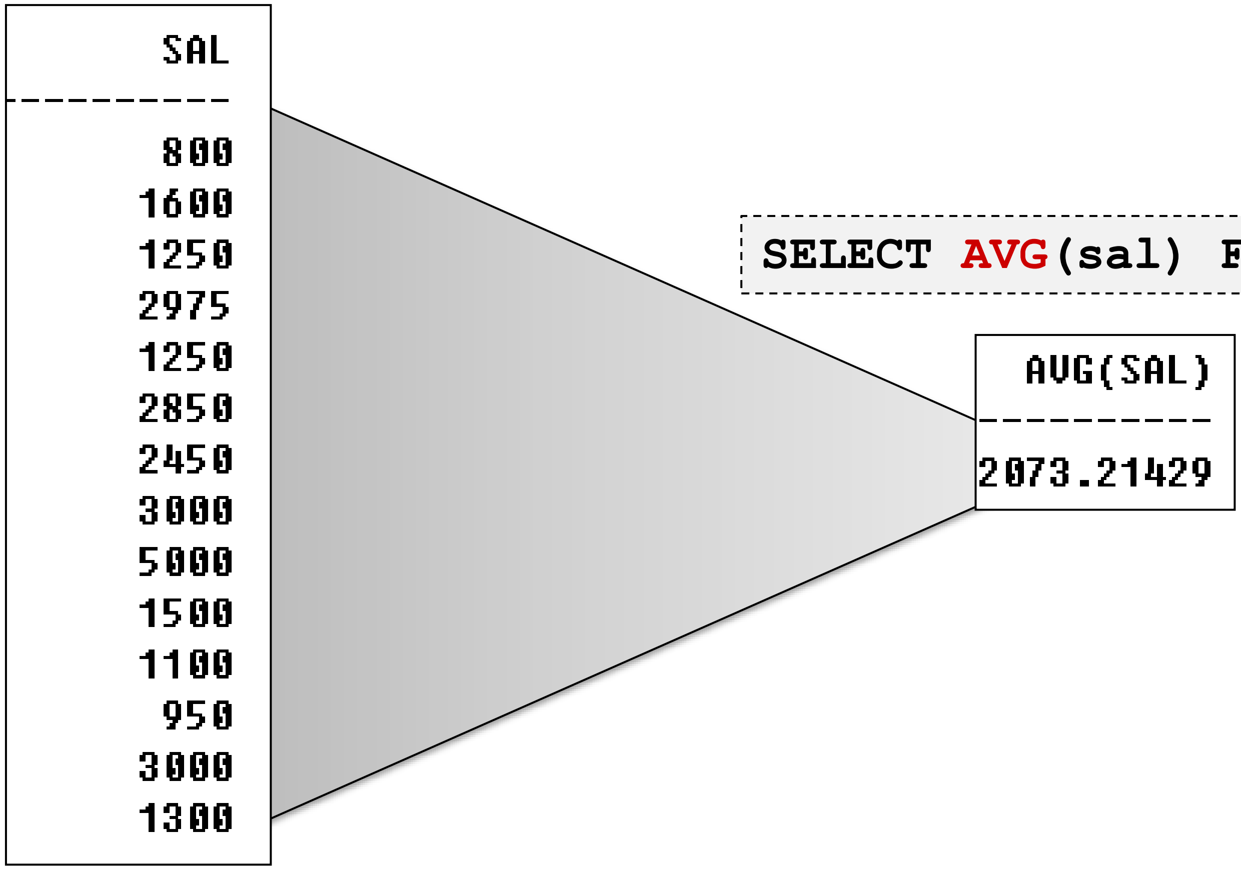
Aggregate Functions

- 여러 행으로부터 하나의 결과값을 반환
- 종류
  - AVG
  - COUNT
    - null포함 행 개수
    - COUNT(*): number of rows in a table (NULL도 count된다)
    - COUNT(expr): non-null value (NULL은 빠진다)
    - COUNT(DISTINCT expr): distinct non-null
    - null빼고 + 중복제거
  - MAX
  - MIN
  - SUM
  - STDDEV 표준편차
  - VARIANCE 분산

만약 튜플에 null만 있다면 count 제외 연산은 모두 null이다.

Aggregate Functions

```
SELECT sal FROM emp;
```



```
       SAL
----------
       800
      1600
      1250
      2975
      1250
      2850
      2450
      3000
      5000
      1500
      1100
       950
      3000
      1300
```

```
SELECT AVG(sal) FROM emp;
```

```
  AVG(SAL)
----------
2073.21429
```

Examples of Aggregate Functions

- 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*) 각 교수당 하나의 수업만 하면 상관없으나 그게 아니므로 distinct를 써서 중복을 제거한다.
    from *teaches*
    where *semester* = 'Spring' **and** *year* = 2010;

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

MYONGJI UNIVERSITY

집계함수에서의 일반적인 오류

```
SELECT deptno, AVG(sal) FROM emp;
```

- 주의
  - 집계함수의 결과는 한 row만 남게 된다
  - deptno는 하나의 row에 표현될 수 없다
  - 부서별과 같은 내용이 필요할 때는 **Group by**절 사용

GROUP BY

```
SELECT deptno, sal
FROM emp
ORDER BY deptno;
```

정렬해라(오름차순)`

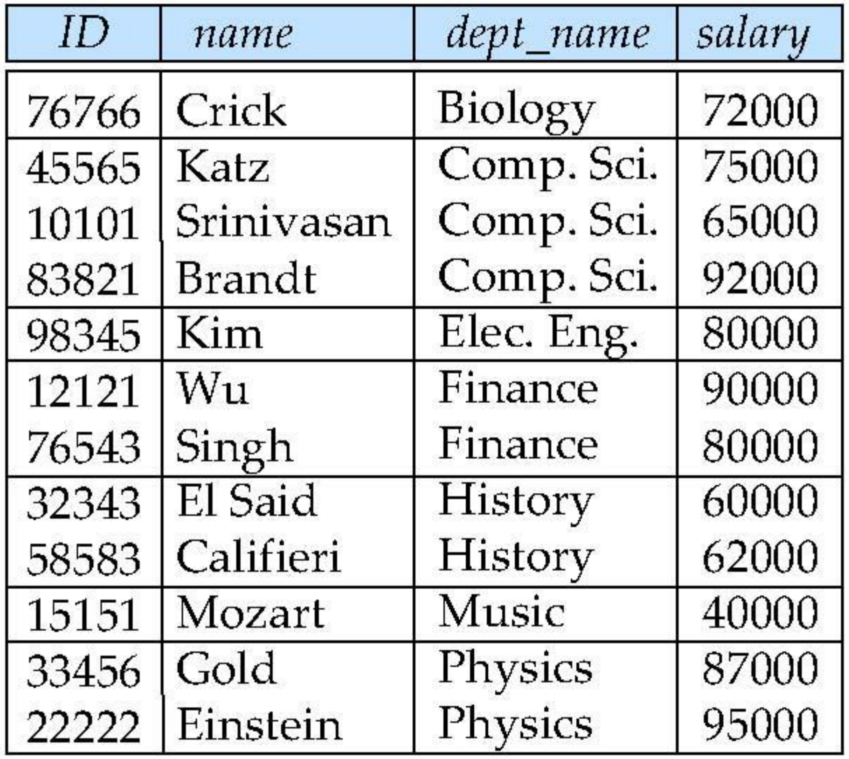
```
    DEPTNO        SAL
---------- ----------
        10       2450
        10       5000
        10       1300
        20       2975
        20       3000
        20       1100
        20        800
        20       3000
        30       1250
        30       1500
        30       1600
        30        950
        30       2850
        30       1250
```

```
SELECT deptno, AVG(sal)
FROM emp
GROUP BY deptno
ORDER BY deptno;
```

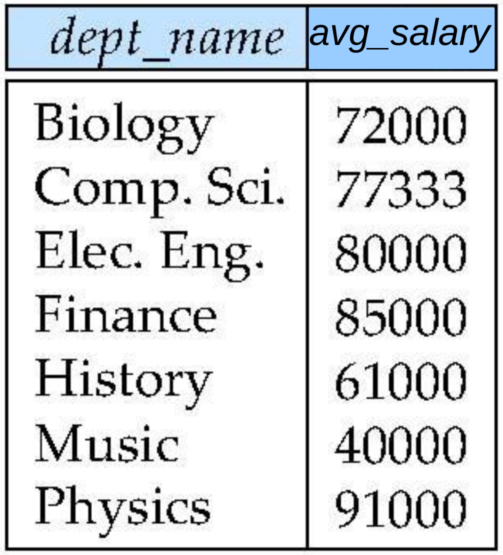
```
    DEPTNO   AVG(SAL)
---------- ----------
        10 2916.66667
        20       2175
        30 1566.66667
```

GROUP BY

- Find the average salary of instructors in each department
  - **select** *dept_name*, **avg** (*salary*) **as** *avg_salary*
    from *instructor*
    group by *dept_name*;



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

Group By에서의 일반적인 오류

- 부서별 월급에서 부서명도 출력?

```
SELECT deptno, dname, AVG(sal)
FROM emp
GROUP BY deptno
ORDER BY deptno;
```

  - 비록 부서번호에 따라 부서명은 하나로 결정될 수 있지만, <u>dname은 grouping에 참여하지 않았으므로</u> 하나의 row로 aggregate될 수 있다고 볼 수 없음

- 주의
  - **SELECT의 컬럼 리스트에는 Group by에 참여한 필드나 Aggregate 함수만 올 수 있다!**
  - Group by가 수행된 이후에는 <u>Group by에 참여한 필드나 Aggregate 함수만 남아있는 셈</u> (∴ dname을 project 할 수 없음)
    - HAVING, ORDER BY도 마찬가지 즉 원본테이블이 아닌 Groupby 테이블이 기본이라 생각해야됨.

Group By에서의 일반적인 오류

- 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*;

ID는 groupby테이블에 없기때문에 오류

여기에 텍스트 입력

HAVING

- Aggregation 결과에 대해 다시 condition을 적용할 때 사용

- 일반적인 오류
  – 평균 월급이 2000 이상인 부서는?

```
SELECT deptno, AVG(sal)
FROM emp
WHERE AVG(sal) >= 2000
GROUP BY deptno;
```

보통 오류가 나기때문에 이렇게 안쓴다.

```
SELECT deptno, AVG(sal)
FROM emp
GROUP BY deptno
HAVING AVG(sal) >= 2000;
```

- 주의
  – WHERE 절은 Aggregation 이전, HAVING 절은 Aggregation 이후의 Filtering
  – **Having절에는 Group by에 참여한 컬럼이나 Aggregate 함수만 사용 가능!**

집계함수는 Groupby참여 이외도 가능하다.

HAVING

- 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;

위 조건 대신 count(ID) > 2;도 가능하다.

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

where절은 grouping이전에 먼저 적용 후 goruping한다.

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 모두 null일시 결과도 null

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

- What if collection has only null values?
  - count returns 0
  - all other aggregates return null

SQL문 실행 개념

JOIN을 통하여 큰 테이블을 만든다	**FROM**
테이블로부터 한 Row씩 읽어 조건을 만족하는 결과만 뽑는다	**WHERE**
원하는 그룹별로 행들을 Grouping한다	**GROUP BY**
원하는 조건을 만족하는 그룹만 남긴다	**HAVING**
주어진 조건에 따라 정렬 한다	**ORDER BY**
원하는 결과만 Projection한다	**SELECT**

SQL 작성법

① 최종 출력될 정보에 따라 원하는 컬럼들을 SELECT 절에 추가
② 원하는 정보를 가진 테이블들을 FROM 절에 추가
☆③ WHERE절에 알맞은 Join 조건 추가
④ WHERE절에 알맞은 검색 조건 추가
⑤ 필요에 따라 GROUP BY, HAVING 등을 통해 Grouping/Filtering
⑥ 정렬 조건 ORDER BY에 추가

기본

SELECT [ALL | DISTINCT] 열_리스트
[**FROM** 테이블_리스트]
[**WHERE** 조건]
[**GROUP BY** 열_리스트 [HAVING 조건]]
[**ORDER BY** 열_리스트 [ASC | DESC]];

NESTED SUBQUERIES

Nested Subqueries

- SQL provides a mechanism for *nesting* of subqueries. A **subquery** is a **select-from-where** expression that is nested within another query
- The nesting can be done in the following SQL query

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

  as follows:
  - A_i can be replaced by a subquery that generates a single value
  - r_i can be replaced by any valid subquery
  - P can be replaced with an expression of the form:

    B <operation> (subquery)

    where B is an attribute and <operation> to be defined later

MYONGJI UNIVERSITY

Subqueries in the Select Clause

- **Scalar subquery** is the one which is used where a *single* value is expected → select 절에서 함수처럼 사용되는 Query

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

최종 테이블

dept_name	num_inst
⋮	⋮

```
select dept_name,
        (select count(*)
         from instructor
         where department.dept_name = instructor.dept_name)
         as num_instructors
from department;
```

행을 지날때마다 계속 바뀌는 것 (for int i느낌)
이유는 select문은 결국 튜플 한행 씩 보기 때문이다.
ex. 컴공, 정통 아예 하나를 쭉이어감

- Runtime error if subquery returns more than one result tuple

Subqueries in the From Clause

- SQL allows a subquery expression to be used in the **from** clause
  - from절에서 사용되며 임시 공간에 테이블을 생성하여 사용하는 View와 비슷함 → **Inline View**라고도 표현

- Find the average instructors' salaries of those departments where the average salary is greater than $42,000 cf.34page having절

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

- Note that we do *not* need to use the **having** clause

Subqueries in the From Clause

- Another way to write the query, “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)
          from instructor
          group by dept_name) as dept_avg (dept_name, avg_salary)
 where avg_salary > 42000;
```

원래 이름이없던 임시테이블을 이름을 주었음.

Subqueries in the Where Clause

- A common use of subqueries is to perform tests:
  - for set membership
  - for set comparisons
  - for set cardinality

→ Where 조건 절에서 주로 **비교형태**로 사용됨

Set Membership

< in
not in

- Find courses offered in Fall 2009 and in Spring 2010

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

조건을 term이라 하는데 현재 3개의 term이 있다. + 이 조건들이 모두 and로 결합

2010년 봄학기에 하나라도 열린 강좌번호의 테이블

- Find courses offered in Fall 2009 but not in Spring 2010 집합에선 except연산

```
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);
```

Set Membership

- Name all instructors whose name is neither “Mozart” nor Einstein”

cf. neither A nor B : A B 둘다 아닌

```
select distinct name
from instructor
where name not in ('Mozart', 'Einstein')
```

+a 'mozart'소문자일 시엔 나온다.(String이기때문에 완벽일치해야됨.)

- Find the total number of (distinct) students who have taken course sections taught by the instructor with ID 10101

이 교수가 가르친걸 수강한 학생의 수

이 ID는 학번

기P.K

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year) in
                    (select course_id, sec_id, semester, year
                     from teaches
                     where teaches.ID = 10101);
```

학생의 수를 세는 것이기 때문에
중복을 제외하는 것

이 아이디는 교수교번

Set Comparison – "some" Clause

- 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';
```

- Same query using > **some** clause

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

아무거나 골랐을때 그거보다 작은게 있으면 된다.
(하나라도 만족하면 가능)

Definition of "some" Clause

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

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

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

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

$$(5 \neq \textbf{some} \begin{array}{|c|}\hline 0\\\hline 5\\\hline\end{array}) = \text{true (since } 0 \neq 5)$$

(= **some**) ≡ **in** 존재 O
However, (≠ **some**) $\not\equiv$ **not in**

Set Comparison – “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’);
```

모든 즉 그집합에선 이조건을 모두 다 만족해야한다.

Definition of "all" Clause

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

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

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

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

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

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

However, (= **all**) $\not\equiv$ **in**

Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty
- **exists** $r \Leftrightarrow r \neq \varnothing$ 존재하냐? 즉 공집합이 아닌 것
- **not exists** $r \Leftrightarrow r = \varnothing$ 존재하지않냐 = 즉 공집합이냐

Use of "exists" Clause

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

```
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);
```

조건 총 3개

만약 course_id로 작성시
T.course_id로 인식한다.

이중for문처럼 한튜플당 상수취급해 조건처럼 들어감.
(ex. 첫번째가 db일시 db로 쭉 비교 그뒤에 두번째 튜플 ㄱ)

- **Correlation name** – variable *S* in the outer query
- **Correlated subquery** – the inner query

Use of "not exists" Clause

- Find all students who have taken *all* courses offered in the Biology department 생물학과 모든 교과목 수강한 학생들

÷ Division 연산자통해서 계산한 바있음.

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

조건 총 1개

S.ID가 다 수강한 강좌번호

- first nested query lists all courses offered in Biology
- second nested query lists all courses a particular student took

- Note that $\underline{X - Y = \emptyset}$ ⇔ $\boxed{X \subseteq Y}$

Test for Absence of Duplicate Tuples X

- The **unique** construct tests whether a subquery has any duplicate tuples in its result
- The **unique** construct evaluates to "true" if a given subquery contains no duplicates
- 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);
```

이제 안씀 참고만 하기

MYONGJI UNIVERSITY

With Clause

- The **with** clause provides a way of defining a *temporary* relation whose definition is available only to the query in which the **with** clause occurs

- Find all departments with the maximum budget

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

MODIFICATION OF THE DATABASE

Modification of the Database

- **Deletion** of tuples from a given relation

  삭제

- **Insertion** of new tuples into a given relation

  추가

- **Updating** of values in some tuples in a given relation

  업데이트

Modification of the Database

- 종류
  - Add new row(s) 한줄추가
    - **INSERT INTO *테이블이름* [(*컬럼리스트*)] VALUES (*값리스트*);**
  - Modify existing rows 변경하기
    - **UPDATE *테이블이름* SET *변경내용* [WHERE *조건*];**
  - Remove existing rows 삭제하기
    - **DELETE FROM *테이블이름* [WHERE *조건*];**

- **트랜잭션**의 대상
  - 트랜잭션은 DML의 집합으로 이루어짐

Deletion

- 조건을 만족하는 레코드 삭제
  - 이름이 'SCOTT'인 사원 삭제

```
DELETE FROM emp WHERE ename = 'SCOTT';
```

- 조건이 없으면 모든 레코드 삭제 (주의!)
  - 모든 직원 정보 삭제

```
DELETE FROM emp;
```

- Subquery를 이용한 DELETE
  - 'SALES'부서의 직원 모두 삭제

```
DELETE FROM emp WHERE deptno =
       (SELECT deptno FROM dept WHERE dname = 'SALES');
```

부서번호를 모르므로 한번더 조건건다.

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');
```

instructor테이블엔 building이 없으므로
조건 하나 더`

Deletion

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

둘이 같아 하나씩 삭제될수록 테이블도 똑같이 삭제되어 평균값이 계속 바뀐다.

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)

Insertion

- 묵시적 방법: 컬럼 이름/순서 지정하지 않음
  - 테이블 생성시 정의한 순서에 따라 값 지정 즉 순서에 맞게 처음에 맞춰 지정해줄 것 하나도 빼먹으면 안됨.

```
INSERT INTO dept VALUES (777, 'MARKETING', NULL);
```

dept테이블의 컬럼은 deptno, dname, contry 순서다.

- 명시적 방법: 컬럼 이름 명시적 사용
  - 지정되지 않은 컬럼 NULL 자동 입력

```
INSERT INTO dept(dname, deptno)
         VALUES ('MARKETING', 777);
```

- Subquery 이용: 타 테이블로부터 데이터 복사
  - 테이블은 이미 존재하여야 함

```
INSERT INTO deptusa
    SELECT deptno, dname FROM dept WHERE country = 'USA';
```

  - 참고: CREATE TABLE AS SELECT는 없는 테이블을 생성 & 데이터 복사

Insertion

- Add a new tuple to *course* 묵시적

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

- or equivalently 명시적

  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

- 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 table1 select * from table1
```

  would cause problem (might insert infinite number of tuples)

Updates

- 조건을 만족하는 레코드를 변경
  - 10번 부서원의 월급 100 인상 & 수수료 0으로 변경

```
UPDATE emp SET sal = sal + 100, comm = 0
WHERE deptno = 10;
```

- WHERE 절이 생략되면 모든 레코드에 적용
  - 모든 직원의 월급 10% 인상

```
UPDATE emp SET sal = sal * 1.1;
```

- Subquery를 이용한 변경
  - 담당업무가 'SCOTT'과 같은 사람들의 월급을 부서 최고액으로 변경

```
UPDATE emp SET sal = (SELECT MAX(sal) FROM emp)
WHERE job = (SELECT job FROM emp WHERE ename='SCOTT');
```

Updates

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

    update *instructor*
        set *salary* = *salary* * 1.03
        where *salary* > 100000;
    update *instructor*
        set *salary* = *salary* * 1.05
        where *salary* <= 100000;

    이 예시의 경우 순서바꿀시 안되는 예로 100000일때
    아래 5프로를 먼저쓰면 5프로오른상태에서 또 3프로
    오를 것이다.

  - the *order* is important! 순서중요!!
  - can be done better using the **case** statement

Updates

- Same query as before but with case statement

이러면 순서 상관없음.

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

참고

- 데이터 입력, 수정 시 자주 사용되는 Pseudo 컬럼
  - USER: Current user name
  - SYSDATE: Current date and time 현재시간
  - ROWID: Location information of rows

```
INSERT INTO emp(eno, hiredate) VALUES (200, SYSDATE);
```

- DEFAULT: default값이 정의된 컬럼에 기본값을 입력할 경우 사용할 수 있음 처음에 create시 default정의가능(null 초기값이 아닌 default값으로 가능하다는 말)

```
INSERT INTO book VALUES (200,'Gems', DEFAULT);
```

- **DELETE와 TRUNCATE의 차이점**
  - Delete는 Rollback 가능하나, 대량의 log 등을 유발하므로 Truncate보다 느림
- **모든 DML문은 Integrity Constraint를 어길 경우 에러 발생!**

1. drop table book; ->DDL / 시그마 날리고 내용도 날림
2. truncate table book; ->DDL / 내용만 날리기(시그마있음)
3. delete from book; -> DML / 내용만 날리기 (시그마있음)

cf. ddl은 undo(되돌리기)가 안된다. 즉 로그 안남김

THE END

MYONGJI UNIVERSITY