KNU CSE

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

MORE SQL:

COMPLEX QUERIES, TRIGGERS, VIEWS, AND SCHEMA MODIFICATION

Chapter 7

Chapter Outline

- More Complex SQL Retrieval Queries
- Specifying Semantic Constraints as Assertions and Actions as Triggers
- Views (Virtual Tables) in SQL
- Schema Modification in SQL

MORE COMPLEX SQL RETRIEVAL QUERIES

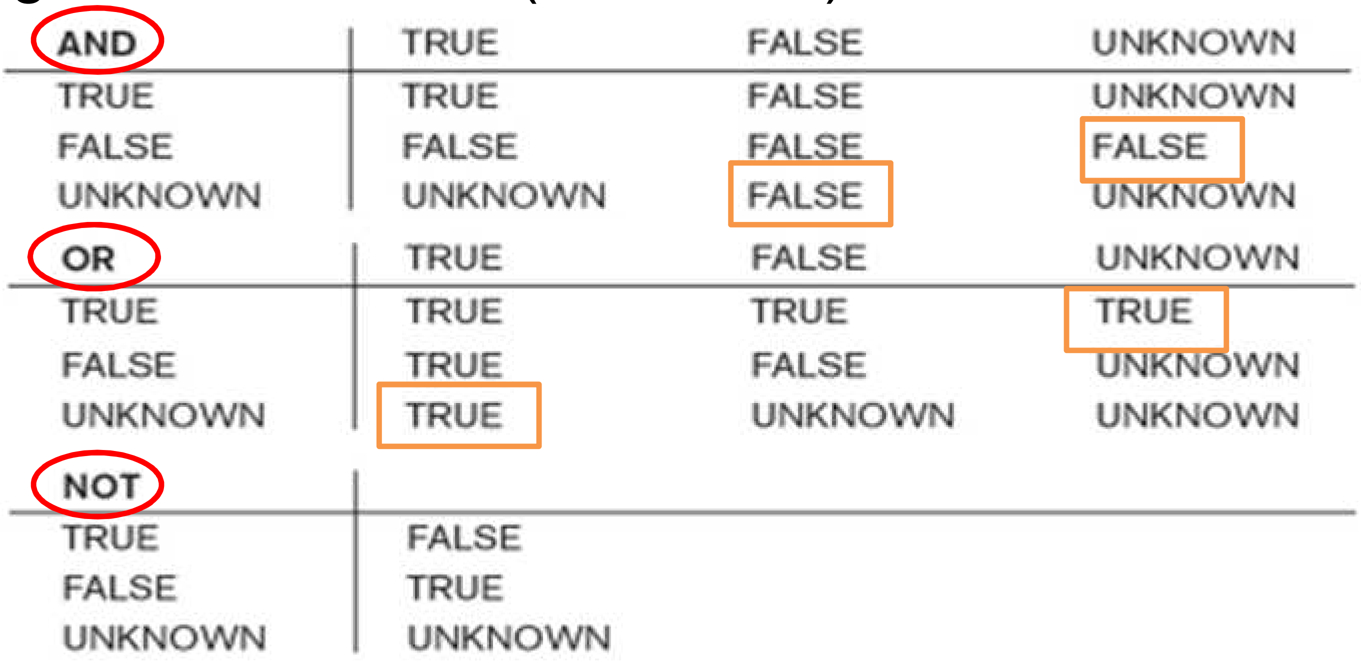
Chapter 7.1

More Complex SQL Retrieval Queries

- Additional feature allow users to specify more complex and interesting retrievals from database.
  - Such features:
    - Nested queries (중첩 질의)
    - Joined tables (Natural Join)
    - *Outer* joins in the `FROM` clause
    - Views (Derived Tables), Assertions, Triggers
    - Aggregate functions (집계 함수)
    - Grouping
- This chapter focuses on learning what they are and how to use them in SQL.

Comparisons Involving `NULL` and Three-Valued Logic

- SQL uses a *three-valued* logic.
    - The result of evaluating an expression falls in:
        - `TRUE`, `FALSE`, and `UNKNOWN`
    - `NULL = NULL` cannot be evaluated.
- Logical connectives (truth table) in the three-Valued Logic



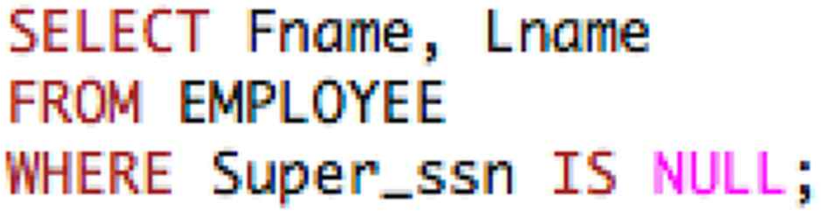
AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

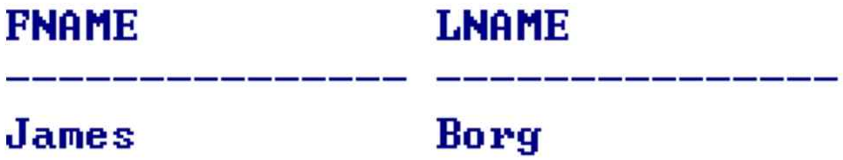
Comparisons Involving NULL and Three-Valued Logic (Cont'd)

- SQL allows queries checking whether an attribute value **IS** NULL.



```
SELECT Fname, Lname
FROM EMPLOYEE
WHERE Super_ssn IS NULL;
```

  - The above query looks for the names of all employees who do not have supervisors.

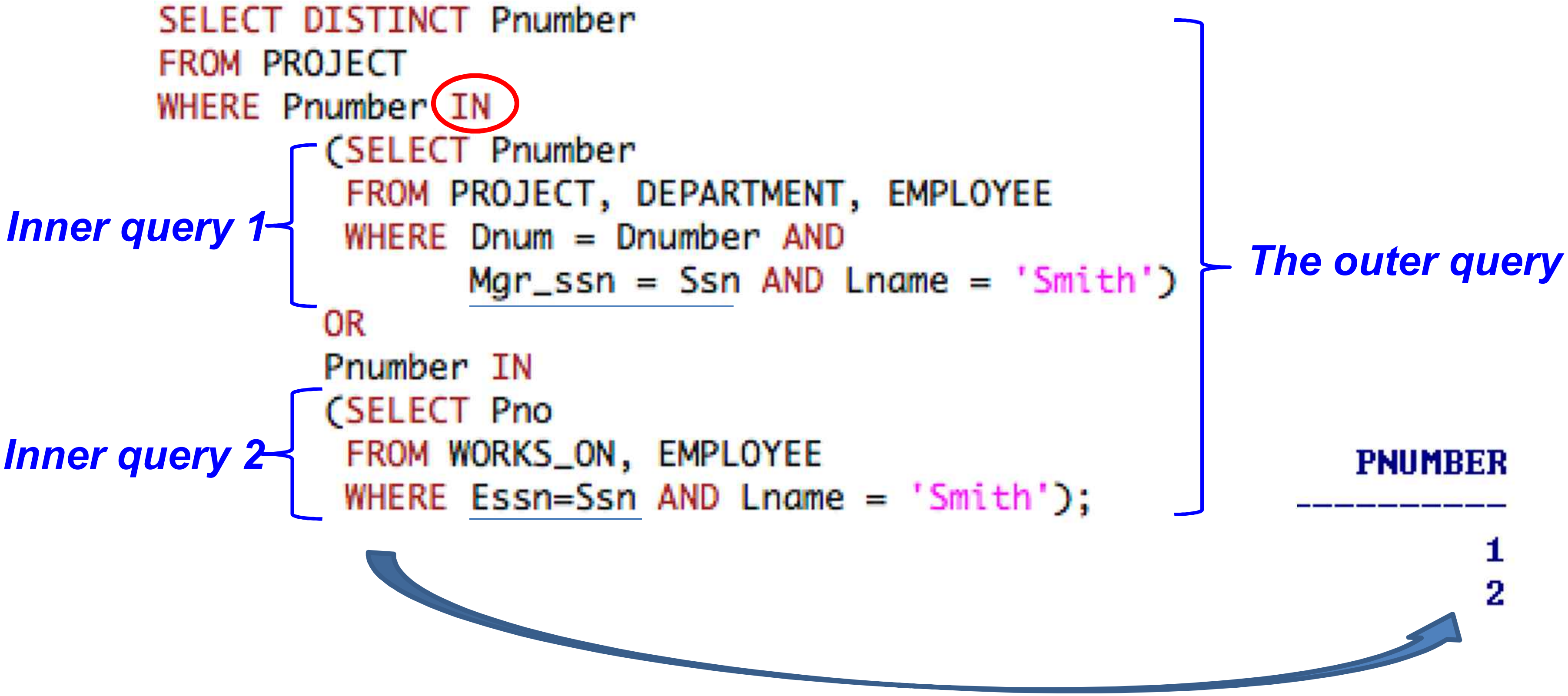


```
FNAME                LNAME
---------------      ---------------
James                Borg
```

  - For this result, we assume that we restore the COMPANY database before the update and deletion done in Lab #5-4.

Nested Queries

- Have complete a *select-from-where* block(s), called a (nested) ***subquery*** or an ***inner query***, within `WHERE` clause of another query, called an ***outer query***



```
SELECT DISTINCT Pnumber
FROM PROJECT
WHERE Pnumber IN
       (SELECT Pnumber
        FROM PROJECT, DEPARTMENT, EMPLOYEE
        WHERE Dnum = Dnumber AND
              Mgr_ssn = Ssn AND Lname = 'Smith')
       OR
       Pnumber IN
       (SELECT Pno
        FROM WORKS_ON, EMPLOYEE
        WHERE Essn=Ssn AND Lname = 'Smith');
```

Inner query 1

Inner query 2

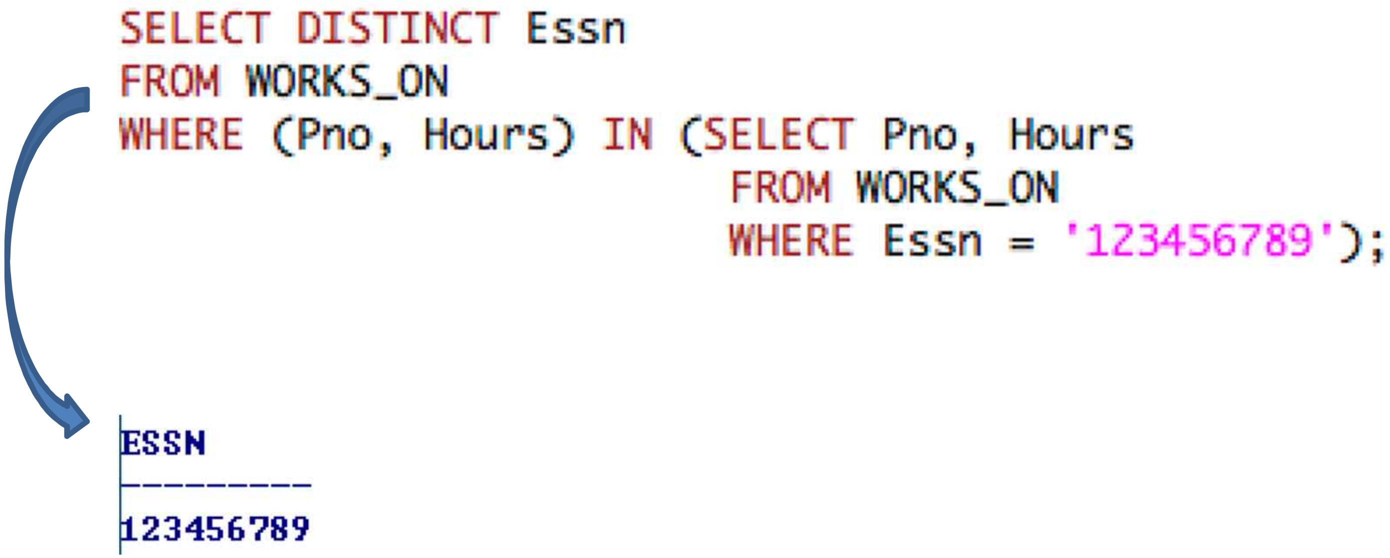
The outer query

```
   PNUMBER
----------
         1
         2
```

(Nested Queries (Cont'd))

Set/Multiset Comparison Operator: IN

- Compares value *v* with a set (or multiset) of values *V*
- Evaluates to TRUE if *v* is one of the elements in *V*
- Can be used for comparing "tuples of values"
  - To do the comparison, place the tuples within ' ( ) '.

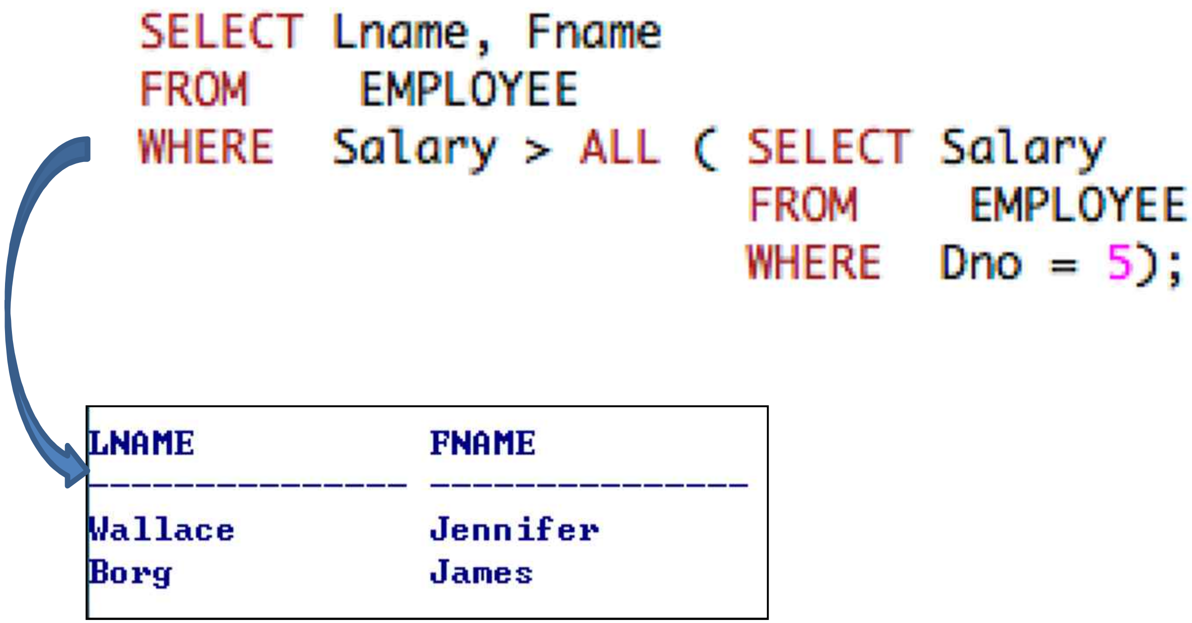


```
SELECT DISTINCT Essn
FROM WORKS_ON
WHERE (Pno, Hours) IN (SELECT Pno, Hours
                       FROM WORKS_ON
                       WHERE Essn = '123456789');
```

```
ESSN
---------
123456789
```

Nested Queries (Cont'd)

- We can use other comparison operator to compare a single value ***v***.
- **ALL**: value must exceed "all" values from nested query

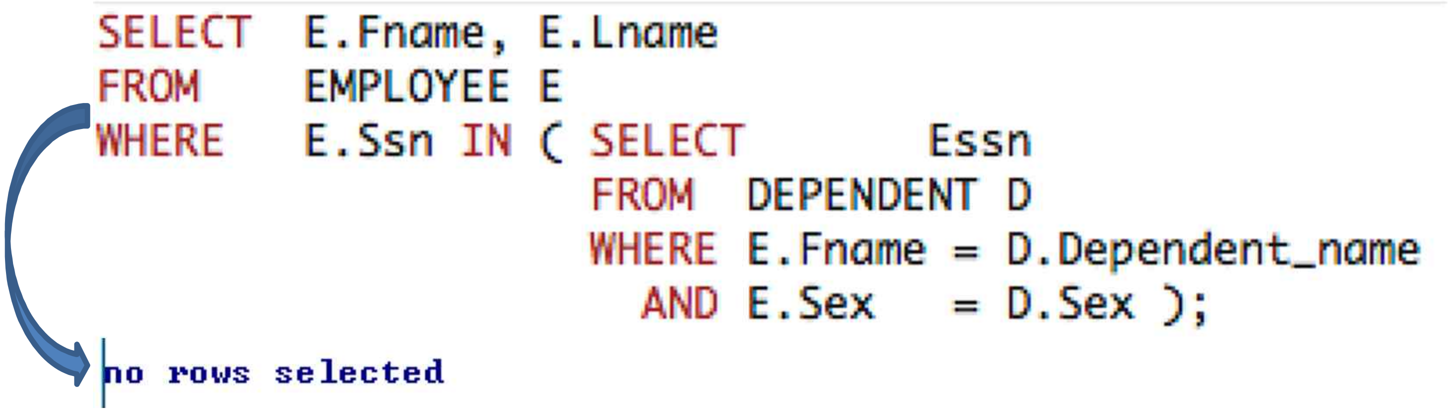


```
SELECT Lname, Fname
FROM    EMPLOYEE
WHERE  Salary > ALL ( SELECT Salary
                      FROM    EMPLOYEE
                      WHERE  Dno = 5);
```

```
LNAME                FNAME
-------------------- --------------------
Wallace              Jennifer
Borg                 James
```

Nested Queries (Cont'd)

- To avoid potential errors and ambiguities, create "aliases" for all tables referenced in an SQL query
- Example) "Retrieve the name of each employee who
  1) Has a dependent with the same first name, and
  2) Is the same gender as that of the employee.



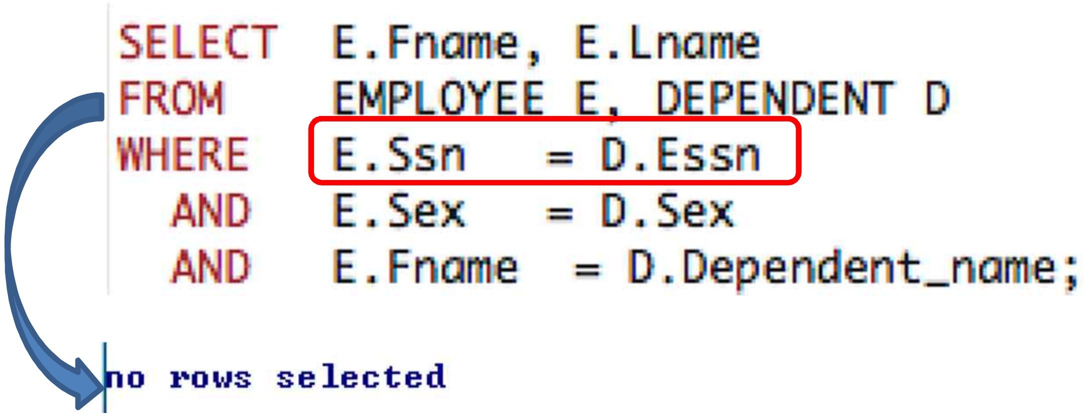
```
SELECT   E.Fname, E.Lname
FROM     EMPLOYEE E
WHERE    E.Ssn IN ( SELECT       Essn
                    FROM   DEPENDENT D
                    WHERE  E.Fname = D.Dependent_name
                      AND  E.Sex   = D.Sex );
```

```
no rows selected
```

  - Called a ***correlated nested query*** (상호 연관된 중첩 질의)
    - Evaluated once for each tuple in the outer query

Nested Queries (Cont’d)

- Nested queries using the ‘**=**’ or ‘**IN**’ can be rewritten into one single query with a join condition.
- Ex) The previous query can be written in the following:

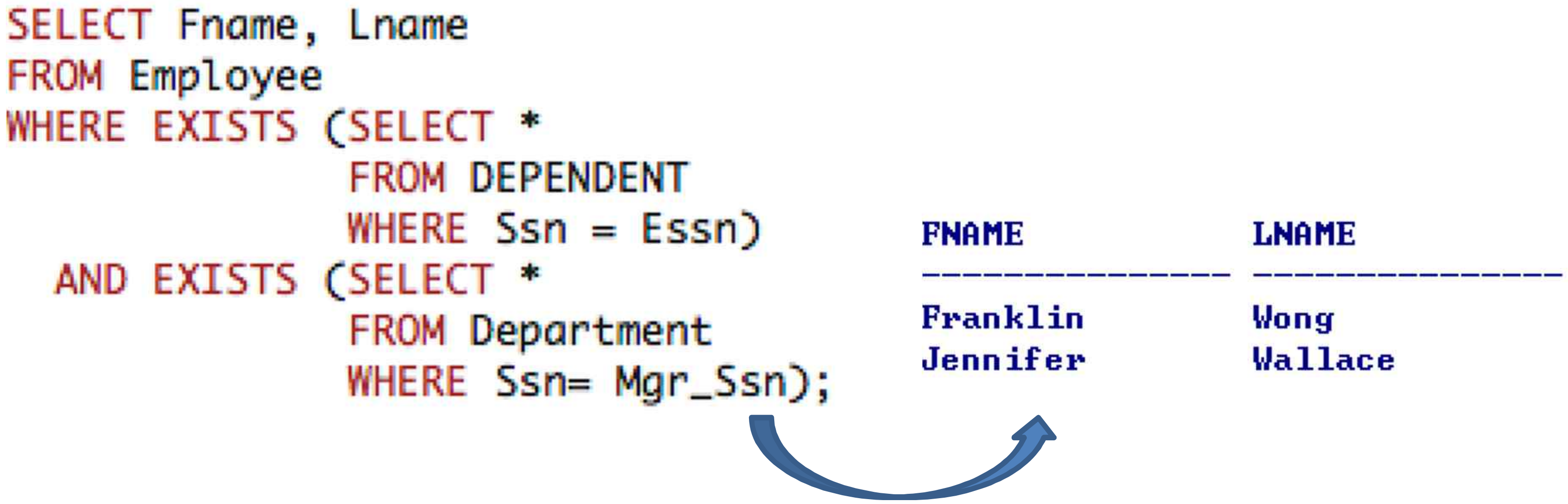


```
SELECT   E.Fname, E.Lname
FROM     EMPLOYEE E, DEPENDENT D
WHERE    E.Ssn    = D.Essn
  AND    E.Sex    = D.Sex
  AND    E.Fname   = D.Dependent_name;
```

```
no rows selected
```

The (NOT) EXISTS Functions in SQL for Correlating Queries

- (NOT) EXISTS function
  - Check whether the result of a correlated nested query is ‘empty’ or not.
  - Can be used in conjunction with a correlated nested query
  - Is a Boolean function that returns a TRUE or FALSE result.
    - If there is no tuple returned by the correlated nested query, then EXISTS (NOT EXISTS) returns TRUE (or FALSE).

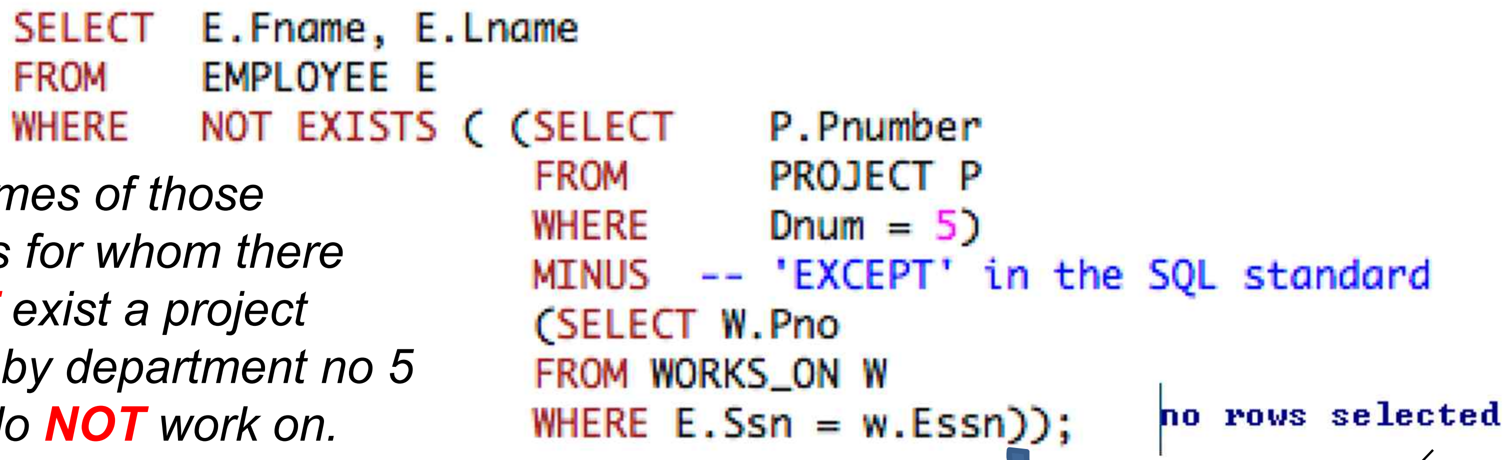


```
SELECT Fname, Lname
FROM Employee
WHERE EXISTS (SELECT *
              FROM DEPENDENT
              WHERE Ssn = Essn)
  AND EXISTS (SELECT *
              FROM Department
              WHERE Ssn= Mgr_Ssn);
```

```
FNAME           LNAME
--------------- ---------------
Franklin        Wong
Jennifer        Wallace
```

Use of **NOT EXISTS**

- To achieve the “for all” (∀), a universal quantifier (정량자: “모든 것에 대해), effect, we may use double negation this way in SQL:
(…하지 않은 튜플들은 존재하지 않는다 -> …한 튜플들만 존재한다.)
- “Retrieve the name of employees working on <u>“ALL” projects controlled by **Dno = 5**</u>.”



```
SELECT   E.Fname, E.Lname
FROM     EMPLOYEE E
WHERE    NOT EXISTS ( (SELECT    P.Pnumber
                       FROM      PROJECT P
                       WHERE     Dnum = 5)
                       MINUS  -- 'EXCEPT' in the SQL standard
                       (SELECT W.Pno
                       FROM WORKS_ON W
                       WHERE E.Ssn = w.Essn));
```

```
no rows selected
```

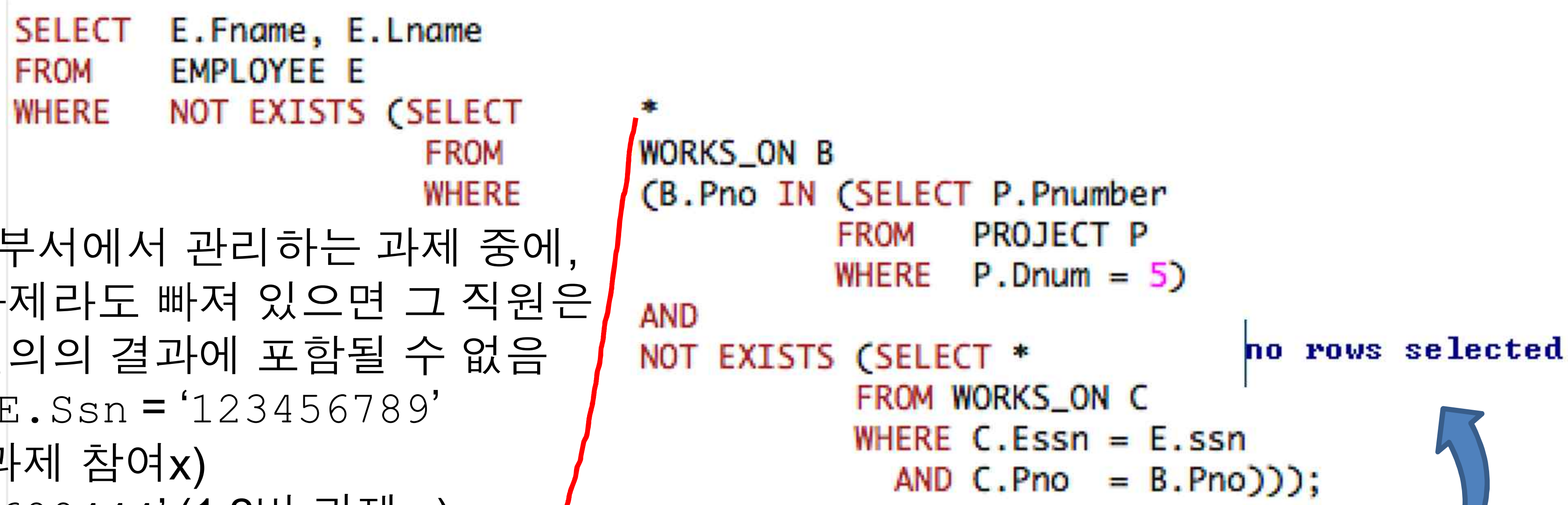
*=> List names of those employees for whom there does **NOT** exist a project controlled by department no 5 that they do **NOT** work on.*

(직원들이 <u>일도 하지 않는</u> 5번 부서에서 관리되는 과제가 <u>없는</u> 그러한 직원들의 이름을 나열하라.)

What does this mean?

Appendix: Use of **NOT EXISTS** (Cont'd)

- The previous query can be rewritten in a more complex way of using two-level nesting:



```
SELECT  E.Fname, E.Lname
FROM    EMPLOYEE E
WHERE   NOT EXISTS (SELECT  *
                    FROM    WORKS_ON B
                    WHERE   (B.Pno IN (SELECT  P.Pnumber
                                       FROM    PROJECT P
                                       WHERE   P.Dnum = 5)
                    AND
                    NOT EXISTS (SELECT *
                                FROM WORKS_ON C
                                WHERE C.Essn = E.ssn
                                  AND C.Pno  = B.Pno)));
```

```
no rows selected
```

* 5번 부서에서 관리하는 과제 중에,
 한 과제라도 빠져 있으면 그 직원은
 본 질의의 결과에 포함될 수 없음
(e.g., E.Ssn = '123456789'
(3번 과제 참여x)
or '66688444' (1,2번 과제 x)
or '453453453' (3번 과제 x)
or '33445555' (1번 과제 x).
E.Ssn = '123456789' 의 경우,

666884444	3	40.0

(in WORKS_ON)
가 존재하게 하게 되어, 선택받지 못함

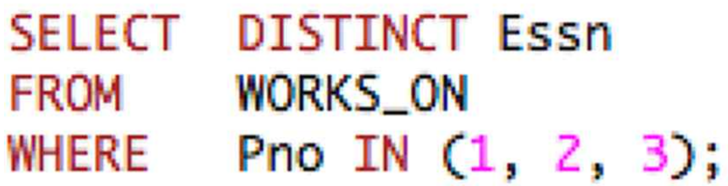
*"Select each employee, such that there does **NOT EXIST** a project controlled by department 5 that the employee does **NOT WORK ON**."*

SQL Function: **UNIQUE** (Q)

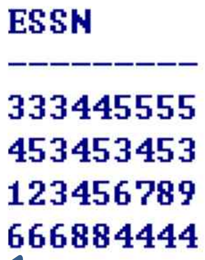
- Returns
  - TRUE if there are no duplicate tuples in the result query Q
  - FALSE, otherwise.
- Can be used to test whether the result of a nested query is a set (no duplicates) or a multiset (duplicates exist).

Explicit Sets and Renaming in SQL

- An *explicit* set of values can be used in the `WHERE` clause.

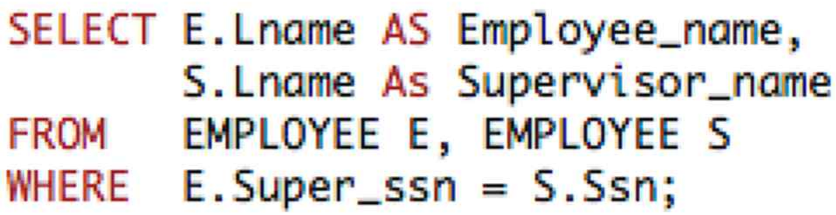


```
SELECT   DISTINCT Essn
FROM     WORKS_ON
WHERE    Pno IN (1, 2, 3);
```

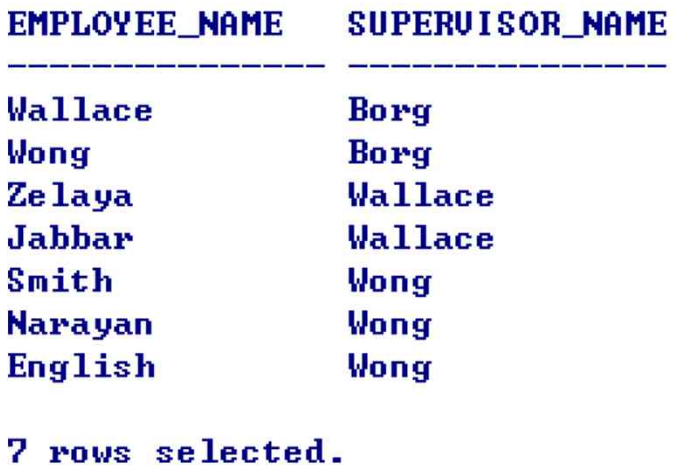


```
ESSN
---------
333445555
453453453
123456789
666884444
```

- Attribute renaming: use “`AS`” followed by whatever name is legal. (Discussed last time)



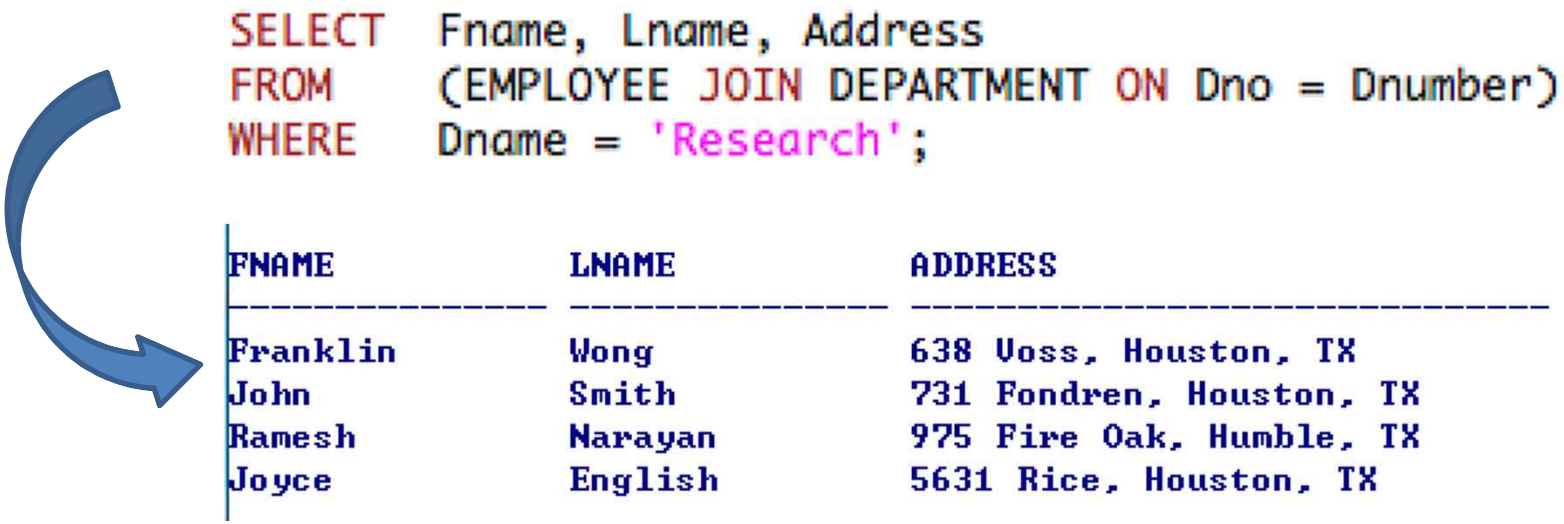
```
SELECT E.Lname AS Employee_name,
       S.Lname As Supervisor_name
FROM   EMPLOYEE E, EMPLOYEE S
WHERE  E.Super_ssn = S.Ssn;
```



```
EMPLOYEE_NAME   SUPERVISOR_NAME
--------------- ---------------
Wallace         Borg
Wong            Borg
Zelaya          Wallace
Jabbar          Wallace
Smith           Wong
Narayan         Wong
English         Wong

7 rows selected.
```

Joined Tables in SQL and Inner Joins

- ***Joined Tables***
  - Concept: Users can be permitted to specify a table resulting from a **join** operation *in the `FROM` clause* of a query.



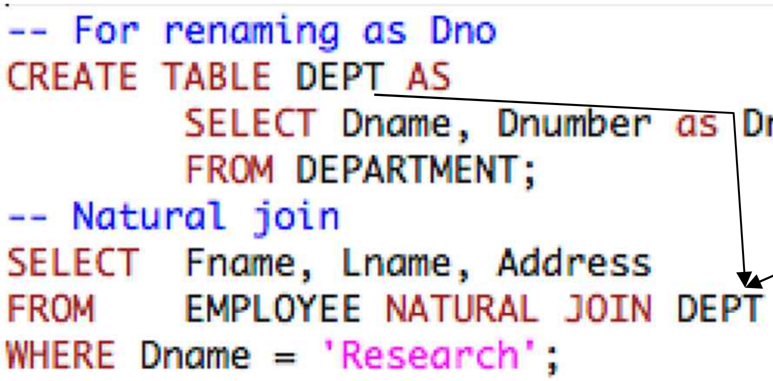
```
SELECT  Fname, Lname, Address
FROM    (EMPLOYEE JOIN DEPARTMENT ON Dno = Dnumber)
WHERE   Dname = 'Research';
```

```
FNAME           LNAME           ADDRESS
--------------- --------------- ------------------------------
Franklin        Wong            638 Voss, Houston, TX
John            Smith           731 Fondren, Houston, TX
Ramesh          Narayan         975 Fire Oak, Humble, TX
Joyce           English         5631 Rice, Houston, TX
```

- Contains a single joined table; Such join may also be called ***inner join*** (to be discussed later) (working for only matching tuples).

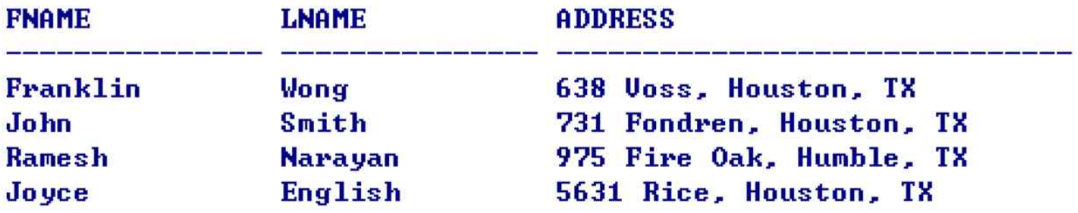
Different Types of JOINed Tables in SQL

- Users can specify *different* types of join:
  - **NATURAL JOIN** (most representative of an inner join)
    - For R (left table) ⋈ S (right table), *no* join condition is specified.
    - Same as creating an implicit **EQUIJOIN** condition for *each pair of attributes with the same name* from R and S



```
-- For renaming as Dno
CREATE TABLE DEPT AS
        SELECT Dname, Dnumber as Dno, Mgr_ssn, Mgr_start_date
        FROM DEPARTMENT;
-- Natural join
SELECT  Fname, Lname, Address
FROM    EMPLOYEE NATURAL JOIN DEPT
WHERE Dname = 'Research';
```

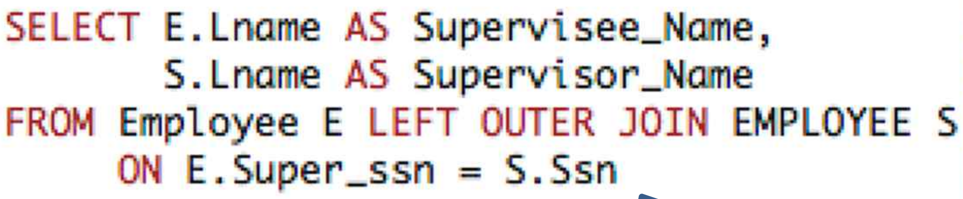
You should rename attributes of one relation so it can be joined with another using NATURAL JOIN.



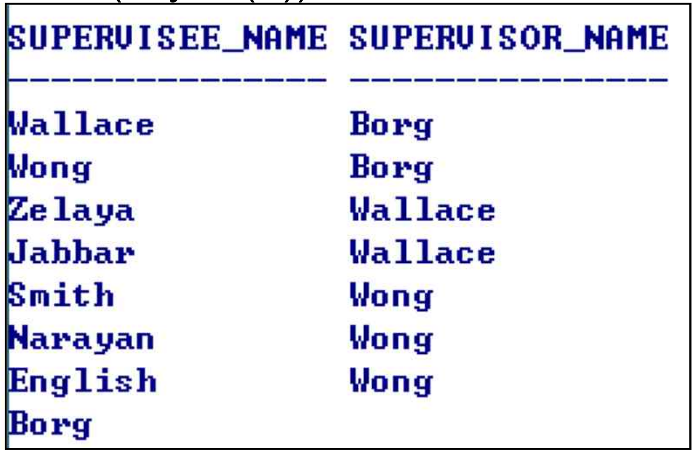
```
FNAME           LNAME           ADDRESS
--------------- --------------- ------------------------------
Franklin        Wong            638 Voss, Houston, TX
John            Smith           731 Fondren, Houston, TX
Ramesh          Narayan         975 Fire Oak, Humble, TX
Joyce           English         5631 Rice, Houston, TX
```

Different Types of JOINed Tables in SQL (Cont'd)

- Users can specify different types of join (Cont'd):
  - **INNER JOIN** (vs. OUTER JOIN)
    - Default type of join in a joined table
    - Tuple is included in the result *only* if a matching tuple exists in the other relation. What if we'd like to see the result including non-matching tuples?
  - **LEFT (RIGHT) OUTER JOIN**
    - **EVERY** tuple in left (right) table (say, R (S)) must appear in the result.
    - If no matching tuple,
      - Paddled with `NULL` values for attributes of right table (say, S (R))
    - Example



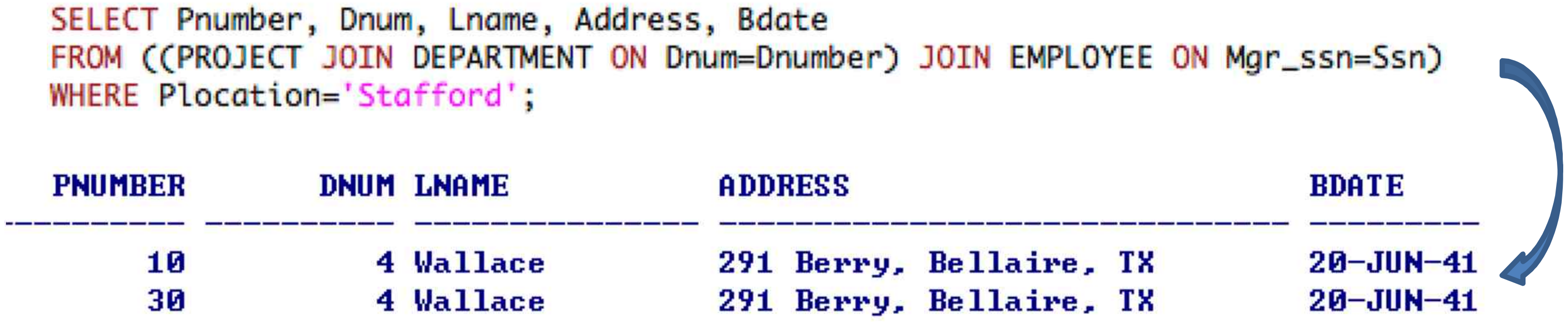
```
SELECT E.Lname AS Supervisee_Name,
       S.Lname AS Supervisor_Name
FROM Employee E LEFT OUTER JOIN EMPLOYEE S
     ON E.Super_ssn = S.Ssn
```



```
SUPERVISEE_NAME SUPERVISOR_NAME
--------------- ---------------
Wallace         Borg
Wong            Borg
Zelaya          Wallace
Jabbar          Wallace
Smith           Wong
Narayan         Wong
English         Wong
Borg
```

Multiway JOIN in the `FROM` clause

- **FULL OUTER JOIN**: combines the result if LEFT and RIGHT OUTER JOIN
- A “multiway” join can be specified by nesting JOIN specifications.
  - Example)



```
SELECT Pnumber, Dnum, Lname, Address, Bdate
FROM ((PROJECT JOIN DEPARTMENT ON Dnum=Dnumber) JOIN EMPLOYEE ON Mgr_ssn=Ssn)
WHERE Plocation='Stafford';
```

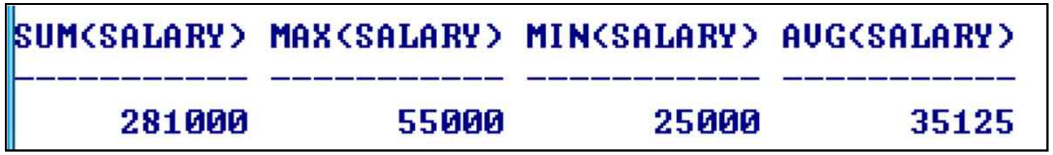
```
   PNUMBER       DNUM LNAME           ADDRESS                        BDATE
---------- ---------- --------------- ------------------------------ ---------
        10          4 Wallace         291 Berry, Bellaire, TX        20-JUN-41
        30          4 Wallace         291 Berry, Bellaire, TX        20-JUN-41
```

Aggregate Functions in SQL

- Why? To summarize information from multiple tuples into a **single-tuple** summary
- Built-in aggregate functions: `COUNT`, `SUM`, `MAX`, `MIN`, `AVG`
- Typically, grouping via `GROUP BY` clause
  - Create **subgroups** of tuples before summarizing
- To select (or, apply condition to) entire groups, `HAVING` clause is used.
- Aggregate functions can be used in the `SELECT` clause and a `HAVING` clause.

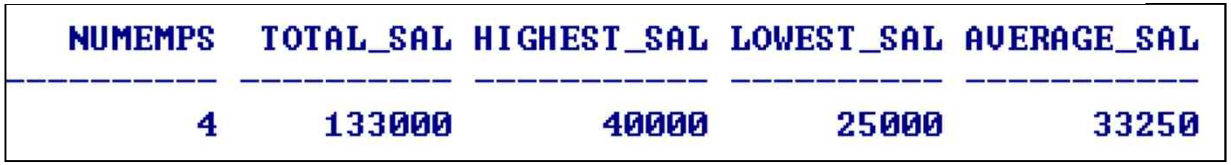
Aggregations Applied for Entire Tuples

```
SELECT   SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
FROM     EMPLOYEE;
```



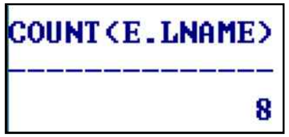
```
SUM(SALARY) MAX(SALARY) MIN(SALARY) AVG(SALARY)
----------- ----------- ----------- -----------
     281000       55000       25000       35125
```

```
SELECT   COUNT(*) as NumEmps, SUM (Salary) AS Total_Sal, MAX (Salary) AS Highest_Sal,
         MIN (Salary) AS Lowest_Sal, AVG (Salary) AS Average_Sal
FROM     EMPLOYEE, DEPARTMENT
WHERE    Dno = Dnumber AND Dname = 'Research';
```



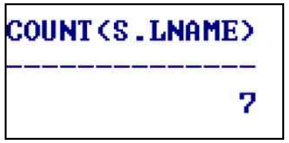
```
   NUMEMPS  TOTAL_SAL HIGHEST_SAL LOWEST_SAL AVERAGE_SAL
---------- ---------- ----------- ---------- -----------
         4     133000       40000      25000       33250
```

```
SELECT COUNT (E.Lname)
FROM Employee E LEFT OUTER JOIN EMPLOYEE S
     ON E.Super_ssn = S.Ssn;
```



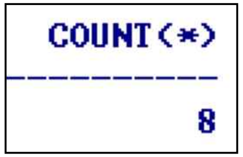
```
COUNT(E.LNAME)
--------------
             8
```

```
SELECT COUNT (S.Lname)
FROM Employee E LEFT OUTER JOIN EMPLOYEE S
     ON E.Super_ssn = S.Ssn;
```



```
COUNT(S.LNAME)
--------------
             7
```

```
SELECT COUNT (*)
FROM Employee E LEFT OUTER JOIN EMPLOYEE S
     ON E.Super_ssn = S.Ssn;
```

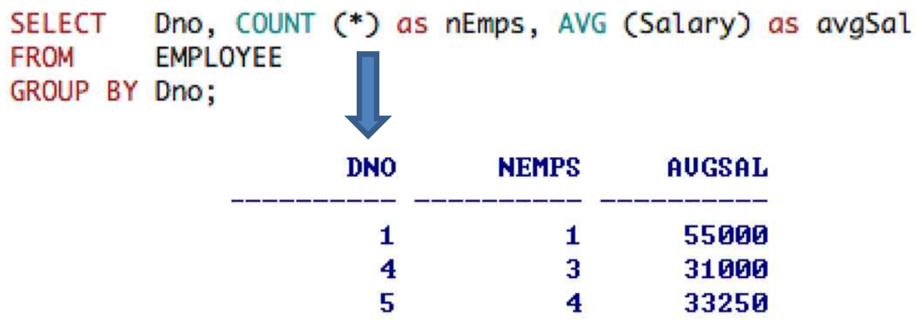


```
  COUNT(*)
----------
         8
```

NULLs are discarded for counting values, EXCEPT for **tuples**.

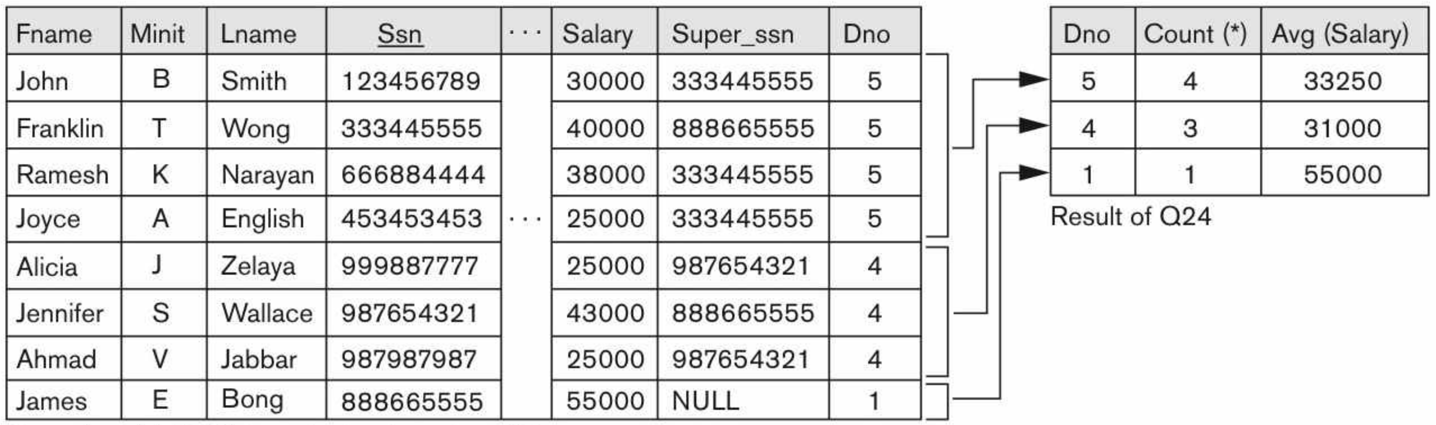
Grouping: The `GROUP BY` Clause

- **Partition** relation into subsets of tuples
  - Based on *grouping attributes*: having the same value for them
  - Apply function to each such group independently
- `GROUP BY` clause
  - Specifies grouping attributes
  - The grouping attribute **MUST** appear in the `SELECT` clause.



```
SELECT   Dno, COUNT (*) as nEmps, AVG (Salary) as avgSal
FROM     EMPLOYEE
GROUP BY Dno;
```

```
       DNO      NEMPS     AVGSAL
---------- ---------- ----------
         1          1      55000
         4          3      31000
         5          4      33250
```



Fname	Minit	Lname	Ssn	. . .	Salary	Super_ssn	Dno
John	B	Smith	123456789		30000	333445555	5
Franklin	T	Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	. . .	25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Bong	888665555		55000	NULL	1

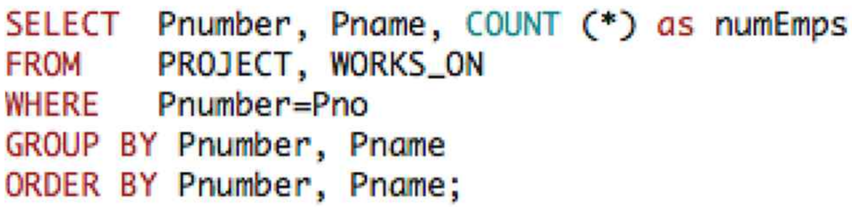
Grouping EMPLOYEE tuples by the value of Dno

Dno	Count (*)	Avg (Salary)
5	4	33250
4	3	31000
1	1	55000

Result of Q24

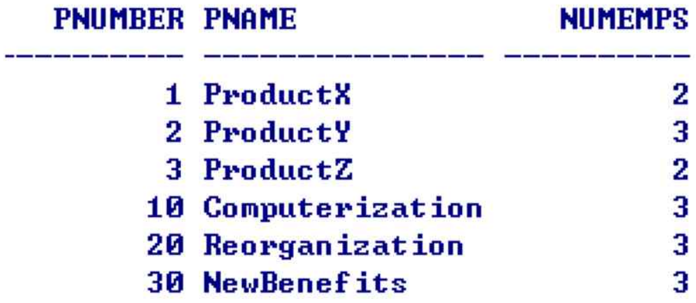
Grouping: The **GROUP BY** Clause

- GROUP BY (GB) clause (Cont'd)
  - Can be applied to the result of JOIN



```
SELECT  Pnumber, Pname, COUNT (*) as numEmps
FROM    PROJECT, WORKS_ON
WHERE   Pnumber=Pno
GROUP BY Pnumber, Pname
ORDER BY Pnumber, Pname;
```

GB *applied to the result of JOIN and then sorted*

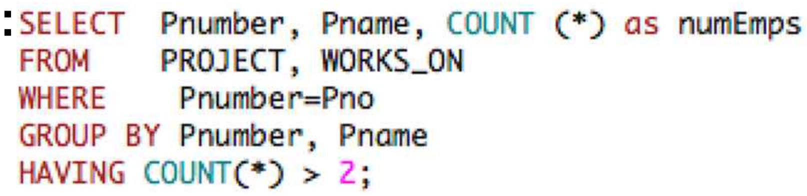


```
   PNUMBER PNAME                NUMEMPS
---------- --------------- ----------
         1 ProductX                 2
         2 ProductY                 3
         3 ProductZ                 2
        10 Computerization          3
        20 Reorganization           3
        30 NewBenefits              3
```

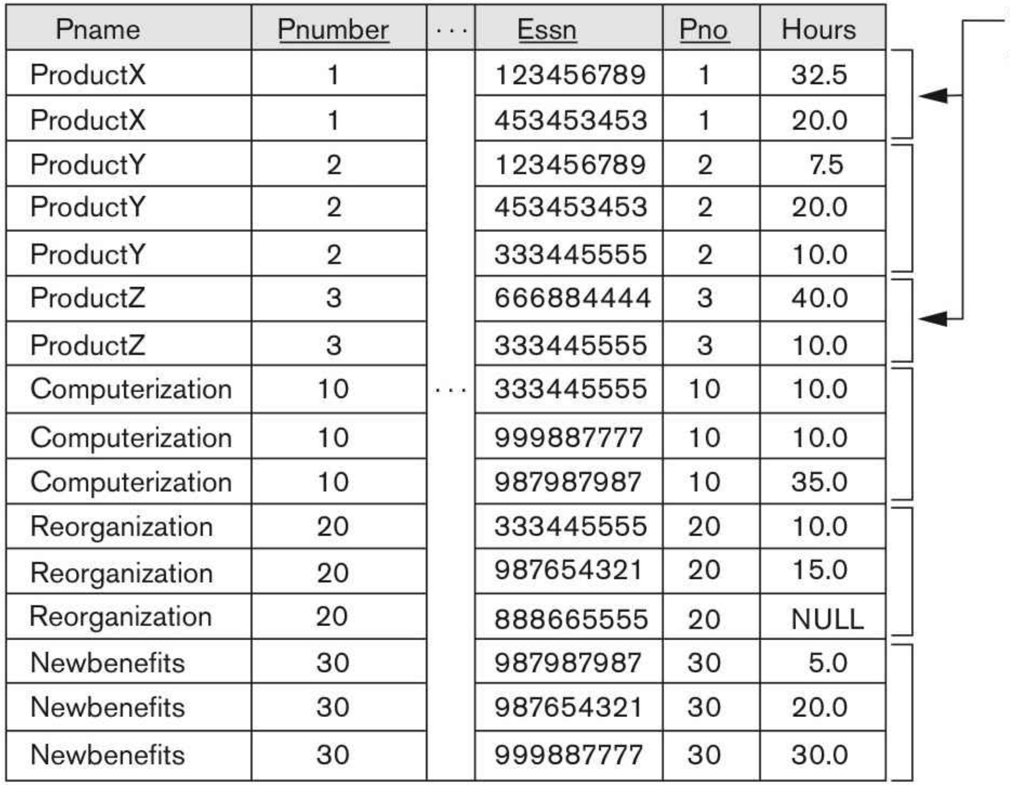
Grouping: The GROUP BY Clause with HAVING Clause

- HAVING clause
  - Provides a condition to select or reject an entire group

Q26:



```
SELECT   Pnumber, Pname, COUNT (*) as numEmps
FROM     PROJECT, WORKS_ON
WHERE     Pnumber=Pno
GROUP BY Pnumber, Pname
HAVING COUNT(*) > 2;
```



Pname	Pnumber	. . .	Essn	Pno	Hours
ProductX	1		123456789	1	32.5
ProductX	1		453453453	1	20.0
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
ProductZ	3		666884444	3	40.0
ProductZ	3		333445555	3	10.0
Computerization	10	. . .	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

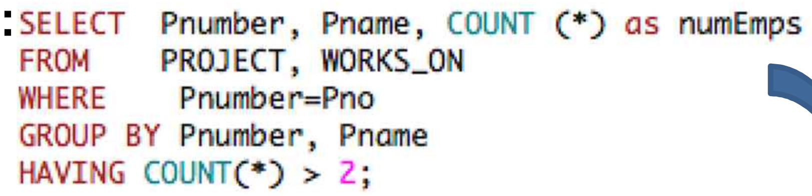
These groups are not selected by the HAVING condition of Q26.

After applying the WHERE clause but before applying HAVING

Grouping: The GROUP BY Clause with HAVING Clause (Cont’d)

- HAVING clause (Cont’d)
  - Provides a condition to select or reject an entire group

Q26:



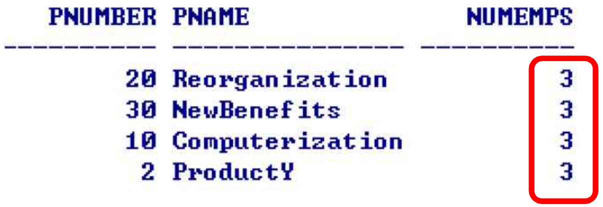
```
SELECT   Pnumber, Pname, COUNT (*) as numEmps
FROM     PROJECT, WORKS_ON
WHERE     Pnumber=Pno
GROUP BY Pnumber, Pname
HAVING COUNT(*) > 2;
```

Pname	Pnumber	. . .	Essn	Pno	Hours
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
Computerization	10		333445555	10	10.0
Computerization	10	. . .	999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

After applying the HAVING clause condition

Pname	Count (*)
ProductY	3
Computerization	3
Reorganization	3
Newbenefits	3

Result of Q26
(Pnumber not shown)

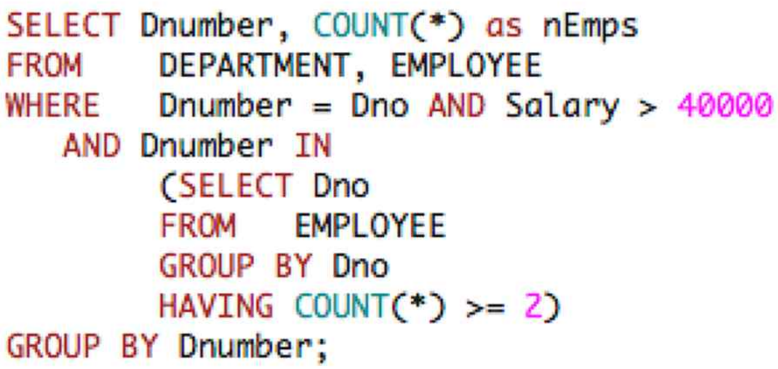


```
   PNUMBER PNAME                       NUMEMPS
---------- --------------- ----------
        20 Reorganization                    3
        30 NewBenefits                       3
        10 Computerization                   3
         2 ProductY                          3
```

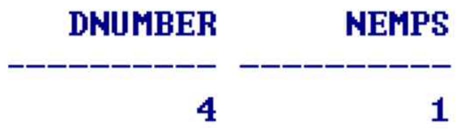
Grouping: The **GROUP BY** Clause with **HAVING** Clause (Cont'd)

- `HAVING` clause (Cont'd)
  - Provides a condition to select or reject an entire group

Q28: "For each department with **>= 2** employees, retrieve the *department number* and *the number of its employee* who're earning > **$40,000**.



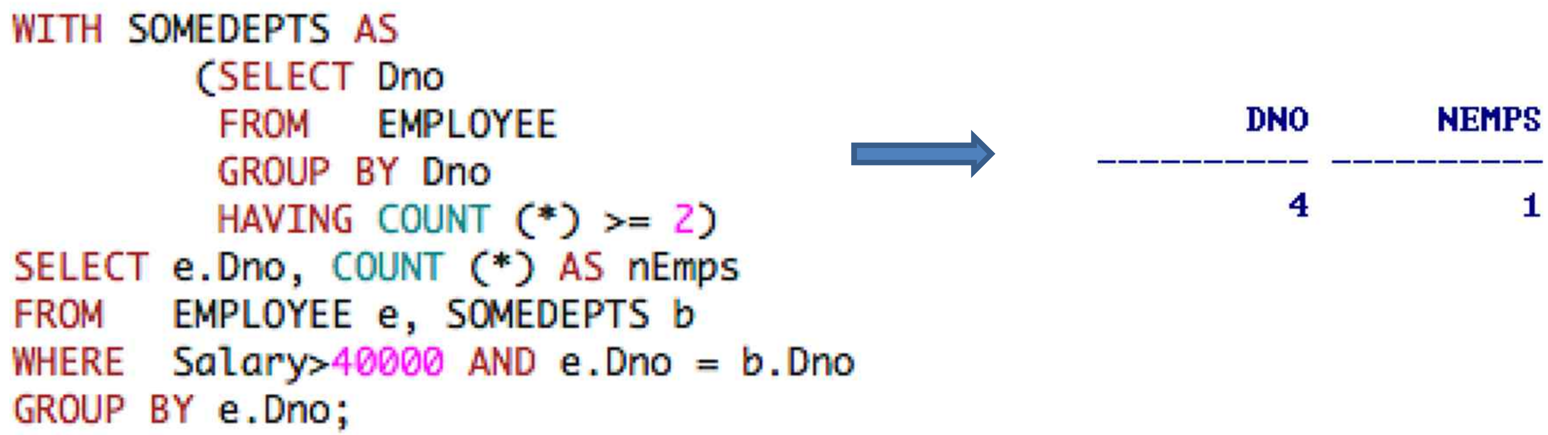
```
SELECT Dnumber, COUNT(*) as nEmps
FROM    DEPARTMENT, EMPLOYEE
WHERE   Dnumber = Dno AND Salary > 40000
   AND Dnumber IN
        (SELECT Dno
        FROM    EMPLOYEE
        GROUP BY Dno
        HAVING COUNT(*) >= 2)
GROUP BY Dnumber;
```



```
   DNUMBER      NEMPS
---------- ----------
         4          1
```

WITH Clause

- Allows a user to define a (temporary) table that will only be used in a “particular” query.
- Used for convenience to create a temporary “View” and use that immediately in a query: called an *in-line view*
- Q28 can be rewritten with the `WITH` clause:

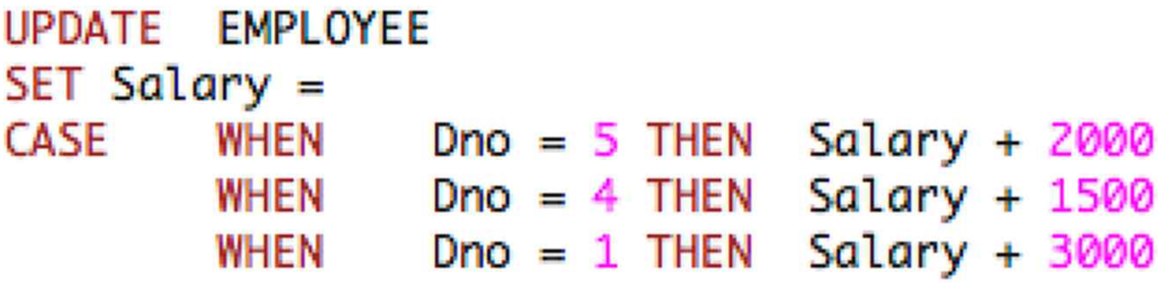


```
WITH SOMEDEPTS AS
        (SELECT Dno
         FROM   EMPLOYEE
         GROUP BY Dno
         HAVING COUNT (*) >= 2)
SELECT e.Dno, COUNT (*) AS nEmps
FROM   EMPLOYEE e, SOMEDEPTS b
WHERE  Salary>40000 AND e.Dno = b.Dno
GROUP BY e.Dno;
```

```
       DNO      NEMPS
---------- ----------
         4          1
```

Use of CASE Clause

- Used when a value can be different based on “certain conditions”
- Can be used in *any* part of an SQL query where a value is expected
- Applicable when *querying*, *inserting*, or *updating* tuples



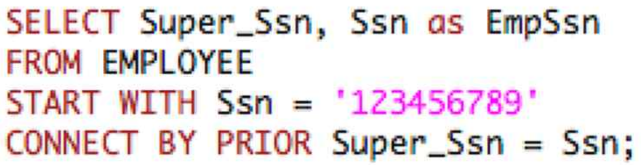
```
UPDATE  EMPLOYEE
SET Salary =
CASE    WHEN    Dno = 5 THEN  Salary + 2000
        WHEN    Dno = 4 THEN  Salary + 1500
        WHEN    Dno = 1 THEN  Salary + 3000
```

Recursive Queries in SQL

- Can be used to keep track of the relationship between tuples of the same type: e.g., employee vs. supervisor
  - Such relationship is described by the FK, `Super_ssn` of `EMPLOYEE`.

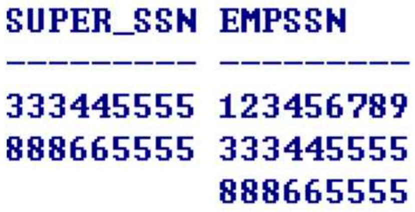
```
WITH RECURSIVE SUP_EMP (SupSsn,
EmpSsn) AS
(SELECT Super_Ssn, Ssn
FROM    EMPLOYEE
UNION
SELECT E.Ssn, S.SupSsn
FROM   EMPLOYEE E, SUP_EMP S
WHERE E.Super_Ssn = S.EmpSsn)
SELECT  *
FROM    SUP_EMP;
```

[In the SQL standard]



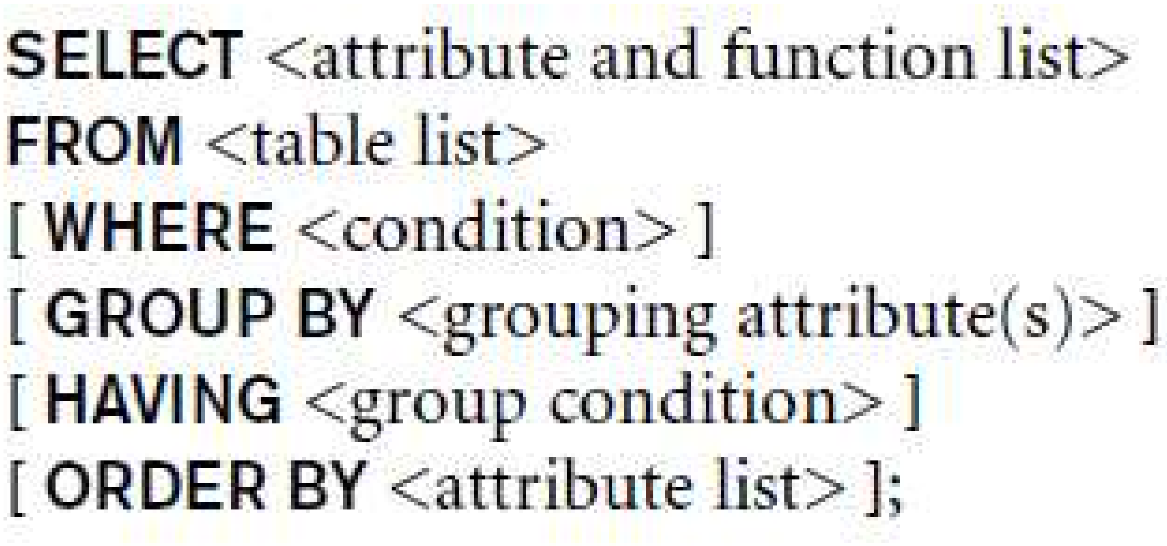
```
SELECT Super_Ssn, Ssn as EmpSsn
FROM EMPLOYEE
START WITH Ssn = '123456789'
CONNECT BY PRIOR Super_Ssn = Ssn;
```

[In Oracle]



```
SUPER_SSN EMPSSN
--------- ---------
333445555 123456789
888665555 333445555
          888665555
```

Reminder: EXPANDED Block Structure of SQL Queries



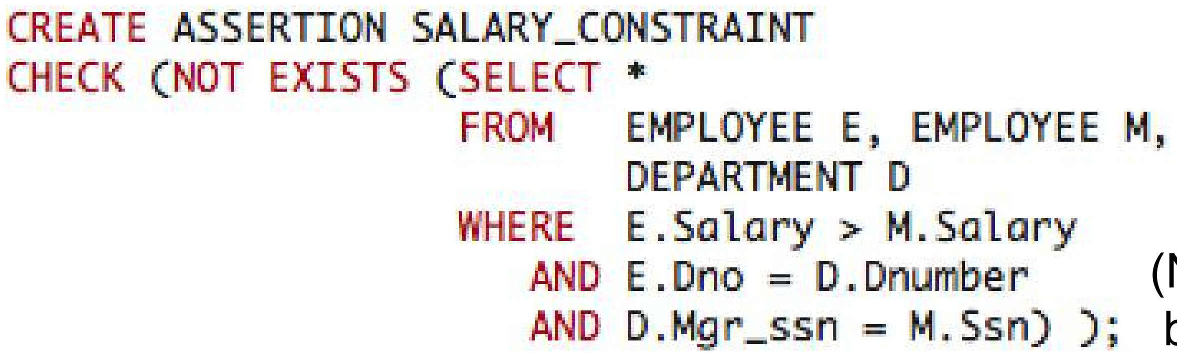
```
SELECT <attribute and function list>
FROM <table list>
[ WHERE <condition> ]
[ GROUP BY <grouping attribute(s)> ]
[ HAVING <group condition> ]
[ ORDER BY <attribute list> ];
```

SPECIFYING CONSTRAINTS AS ASSERTIONS AND ACTIONS AS TRIGGERS

Chapter 7.2

CREATE ASSERTION

- Can allow users to specify “general” constraints via declarative assertions.
    - The constraints do not fall into any of the categories of key (or unique), entity, not-null, referential integrity constraints.
- Specifies a query that selects any tuples violating the desired condition (set by the users).
- Use *only* in cases that cannot be specified by a simple `CHECK` which applies to individual attributes and domains.



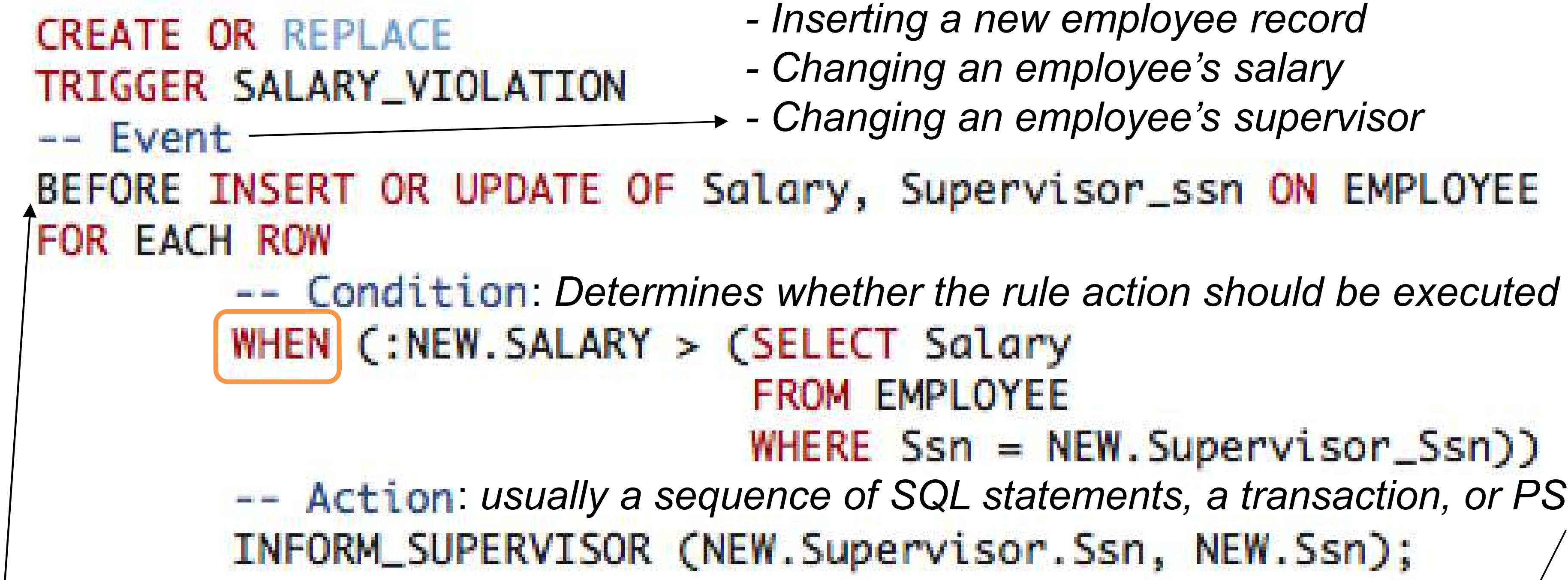
```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK (NOT EXISTS (SELECT *
                   FROM   EMPLOYEE E, EMPLOYEE M,
                          DEPARTMENT D
                   WHERE  E.Salary > M.Salary
                     AND  E.Dno = D.Dnumber
                     AND  D.Mgr_ssn = M.Ssn) );
```

(Not implemented by Oracle)

Triggers (트리거)

- Convenient to specify the *type of action* to be taken when *certain events occur* and *certain conditions are satisfied*
  - “If an employee exceeds a travel expense limit, notify his manager.”
  - Used to monitor the database
- Typical trigger has three components:
  - ***Event(s)***, ***Condition***, ***Action*** (**ECA**)
    - So the trigger is regarded as an ECA rule.
  - These make it a rule for an “active” database, which is out of scope in this course.
    - For those who are further interested, refer to Section 26.1.

Triggers – How to Use? (Cont'd)



```
CREATE OR REPLACE
TRIGGER SALARY_VIOLATION
-- Event
BEFORE INSERT OR UPDATE OF Salary, Supervisor_ssn ON EMPLOYEE
FOR EACH ROW
        -- Condition
        WHEN (:NEW.SALARY > (SELECT Salary
                            FROM EMPLOYEE
                            WHERE Ssn = NEW.Supervisor_Ssn))
        -- Action
        INFORM_SUPERVISOR (NEW.Supervisor.Ssn, NEW.Ssn);
```

- Inserting a new employee record
- Changing an employee's salary
- Changing an employee's supervisor

Condition: *Determines whether the rule action should be executed*

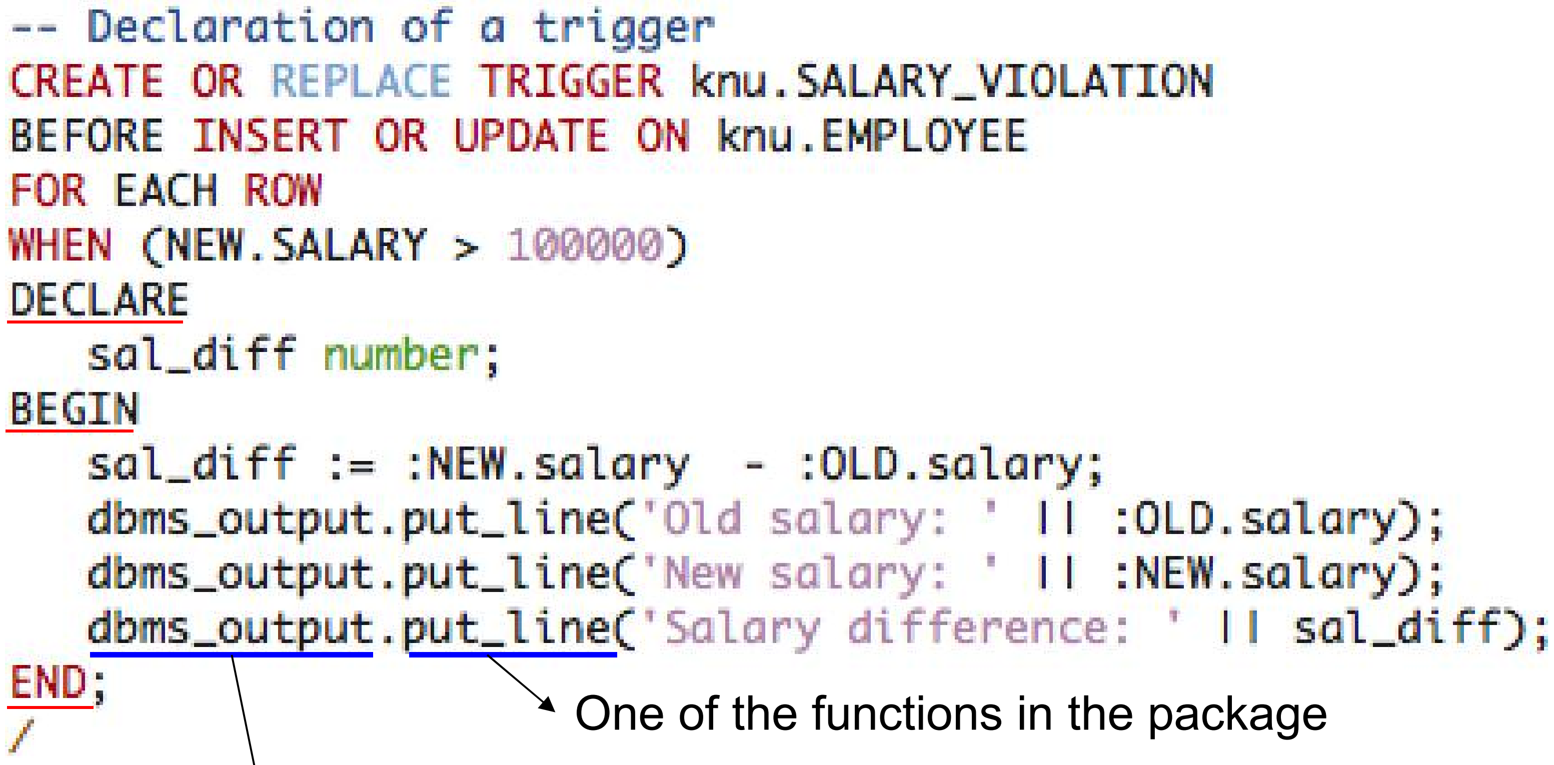
Action: *usually a sequence of SQL statements, a transaction, or PSM*

- Means that the trigger should be executed "BEFORE" the triggering operation is executed.

Called a ***stored procedure*** (저장 프로시저): a *program module* stored by the DBMS at the database server; in the SQL standard, called ***persistent stored modules*** (**PSM**) (영속 저장 모듈)

Not executed in Oracle as it is….

Triggers – Another Example (Cont'd)



```
-- Declaration of a trigger
CREATE OR REPLACE TRIGGER knu.SALARY_VIOLATION
BEFORE INSERT OR UPDATE ON knu.EMPLOYEE
FOR EACH ROW
WHEN (NEW.SALARY > 100000)
DECLARE
   sal_diff number;
BEGIN
   sal_diff := :NEW.salary  - :OLD.salary;
   dbms_output.put_line('Old salary: ' || :OLD.salary);
   dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

A Block of **Oracle PL/SQL**

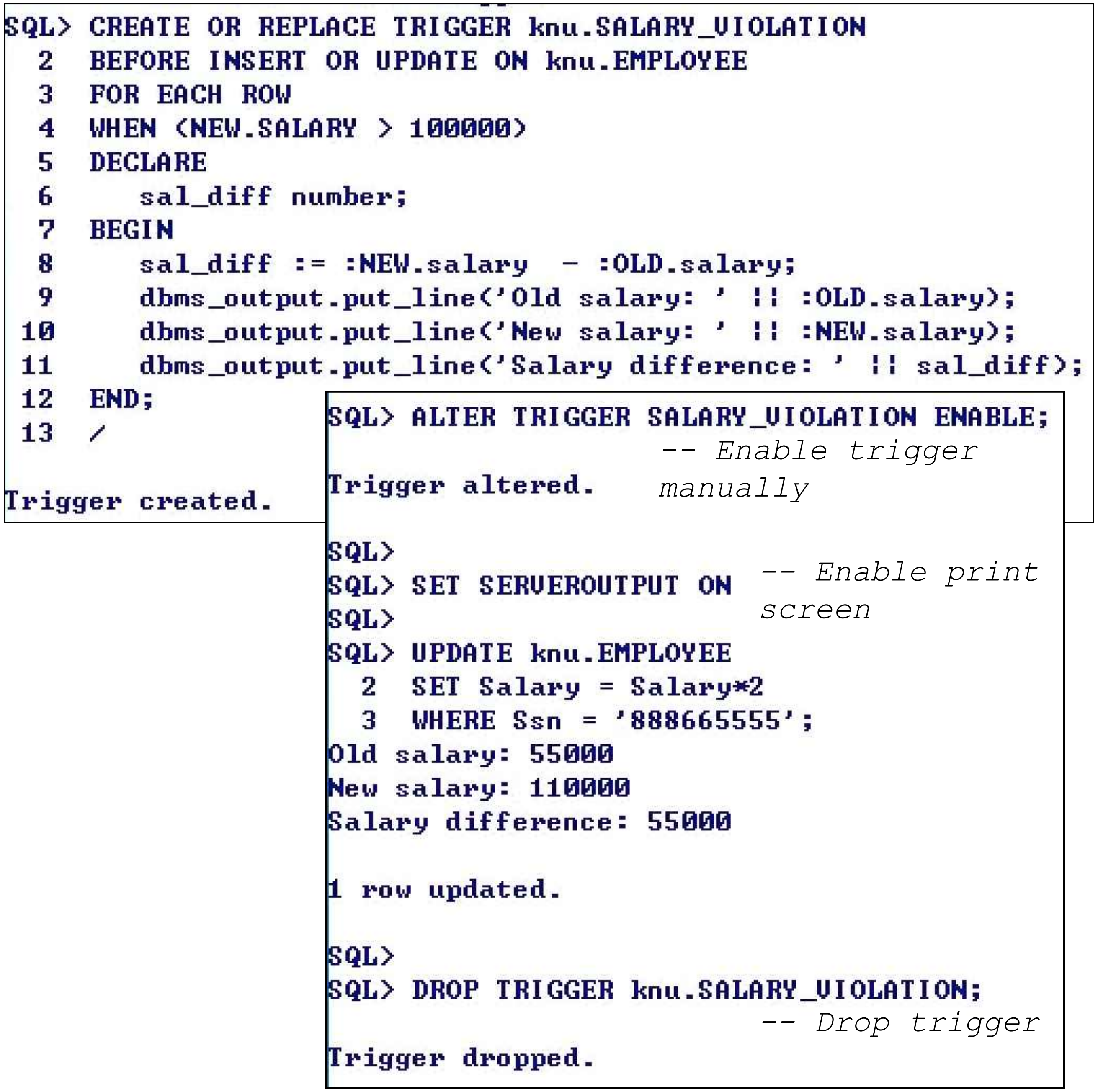
Execute

One of the functions in the package

A system package provided by Oracle

"Report when a employee's new salary exceeds a salary cap, $100K."

Triggers – On Oracle (Cont'd)



```
SQL> CREATE OR REPLACE TRIGGER knu.SALARY_VIOLATION
  2  BEFORE INSERT OR UPDATE ON knu.EMPLOYEE
  3  FOR EACH ROW
  4  WHEN (NEW.SALARY > 100000)
  5  DECLARE
  6     sal_diff number;
  7  BEGIN
  8     sal_diff := :NEW.salary  - :OLD.salary;
  9     dbms_output.put_line('Old salary: ' || :OLD.salary);
 10     dbms_output.put_line('New salary: ' || :NEW.salary);
 11     dbms_output.put_line('Salary difference: ' || sal_diff);
 12  END;
 13  /

Trigger created.
```

```
SQL> ALTER TRIGGER SALARY_VIOLATION ENABLE;
                    -- Enable trigger
Trigger altered.    manually

SQL>
SQL> SET SERVEROUTPUT ON      -- Enable print
SQL>                          screen
SQL> UPDATE knu.EMPLOYEE
  2  SET Salary = Salary*2
  3  WHERE Ssn = '888665555';
Old salary: 55000
New salary: 110000
Salary difference: 55000

1 row updated.

SQL>
SQL> DROP TRIGGER knu.SALARY_VIOLATION;
                              -- Drop trigger
Trigger dropped.
```

VIEWS(VIRTUAL TABLES) IN SQL

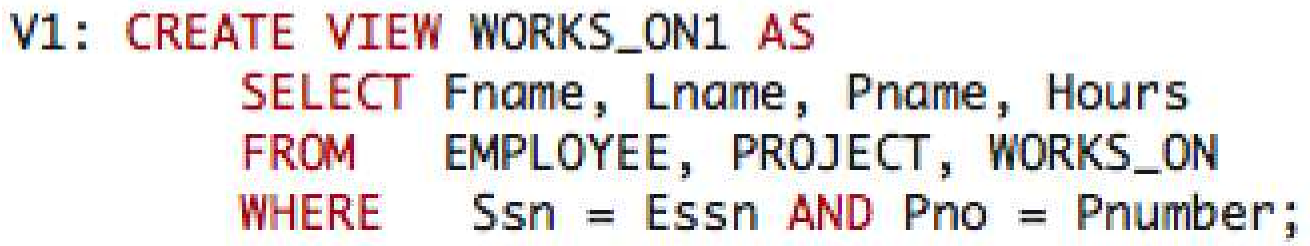
Chapter 7.3

Views (Virtual Tables) in SQL

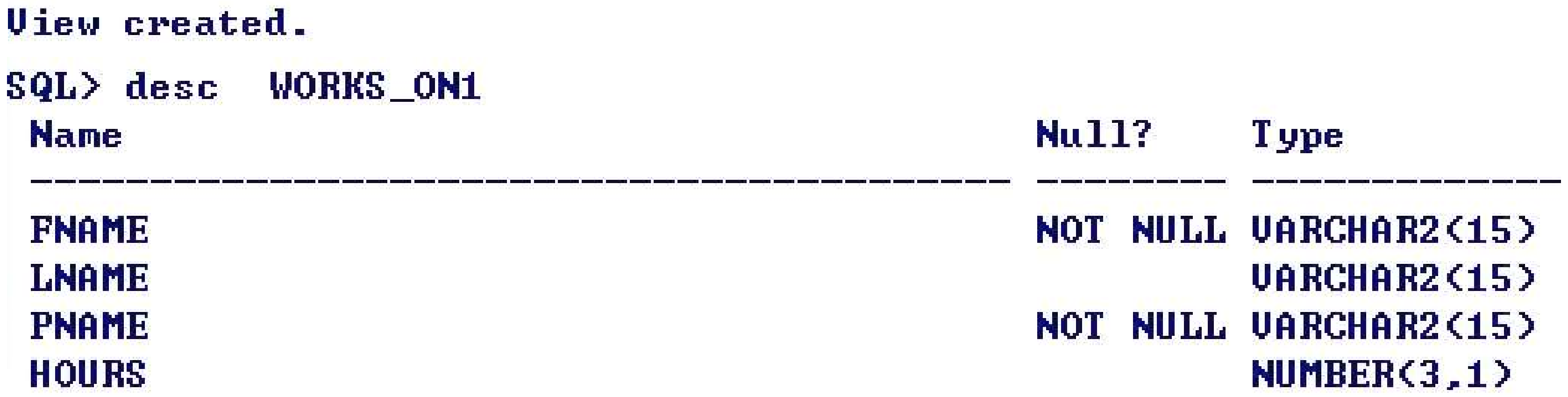
- Concept of a view in SQL
  - Single table **derived from** other tables
    - Somewhat different than “user view” involving many relations
  - Considered to be a *virtual* table that is not “necessarily populated”
- A view can be thought of as a way of specifying a table that needs to be referenced frequently (though it may not exist physically)
  - Often, the view is used for the purpose of caching the result of joins that are “frequently requested” (join cost vs. space (compromised))
  - Ex) “Retrieve the employee name and the project names that the employee works on.”
    - Joins `EMPLOYEE`, `WORKS_ON`, and `PROJECT` every time this query issued.
    - If a view is defined on these joins, then it works for “single-table” retrievals.

Specification of Views in SQL: CREATE VIEW

- Give view (virtual table) name, list attribute names, and include view definition−a “query” to specify the contents of the view



```
V1: CREATE VIEW WORKS_ON1 AS
         SELECT Fname, Lname, Pname, Hours
         FROM   EMPLOYEE, PROJECT, WORKS_ON
         WHERE   Ssn = Essn AND Pno = Pnumber;
```

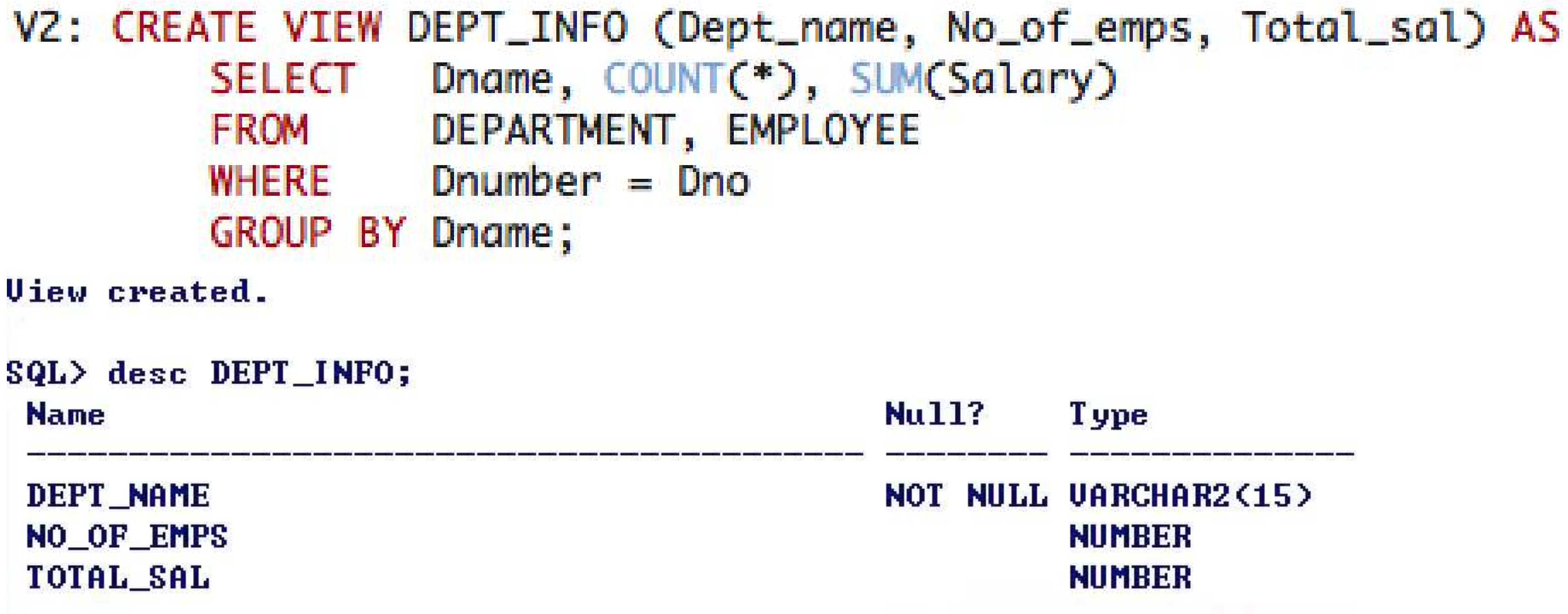


```
View created.

SQL> desc  WORKS_ON1
 Name                                      Null?    Type
 ----------------------------------------- -------- -------------
 FNAME                                     NOT NULL VARCHAR2(15)
 LNAME                                              VARCHAR2(15)
 PNAME                                     NOT NULL VARCHAR2(15)
 HOURS                                              NUMBER(3,1)
```

- V1 *inherits* the names of view attributes from the defining tables.

Specification of Views in SQL: CREATE VIEW (Cont'd)

- Give view (virtual table) name, list attribute names, and include view definition–a "query" to specify the contents of the view



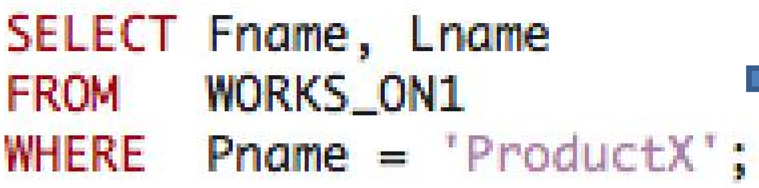
```
V2: CREATE VIEW DEPT_INFO (Dept_name, No_of_emps, Total_sal) AS
        SELECT    Dname, COUNT(*), SUM(Salary)
        FROM      DEPARTMENT, EMPLOYEE
        WHERE     Dnumber = Dno
        GROUP BY  Dname;
```

```
View created.

SQL> desc DEPT_INFO;
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------
 DEPT_NAME                                 NOT NULL VARCHAR2(15)
 NO_OF_EMPS                                         NUMBER
 TOTAL_SAL                                          NUMBER
```

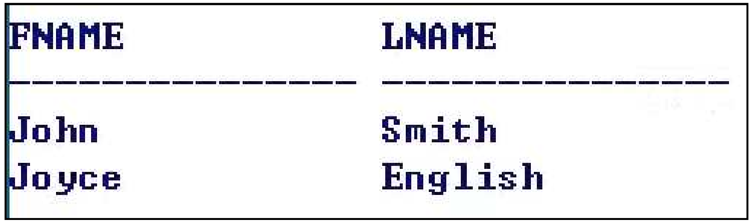
- `V2` explicitly specifies new attribute names for the view by a one-to-one correspondence.

Specification of Views in SQL (Cont'd)

- Once a View is defined, SQL queries can use the View relation in the `FROM` clause.
  - Ex) Accessing the defined view: `WORKS_ON1`



```
SELECT Fname, Lname
FROM   WORKS_ON1
WHERE  Pname = 'ProductX';
```

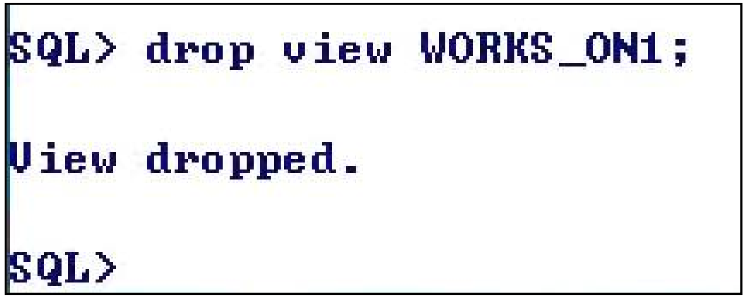


```
FNAME                LNAME
-----------------    -----------------
John                 Smith
Joyce                English
```

Specification of Views in SQL (Cont'd)

- Why using a view? Advantages of defining a view
  1) Simplification of the specification of certain queries.
  2) Provision of a security and authorization mechanism
  3) Saving (multiple) expensive join cost by space *if materialized*

- **DROP VIEW**
  - Dispose of a view.



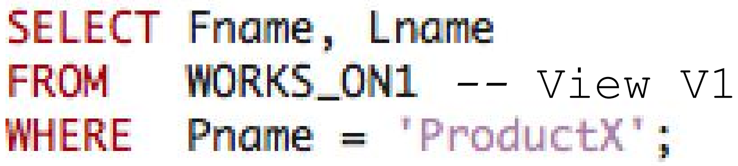
```
SQL> drop view WORKS_ON1;

View dropped.

SQL>
```

View Implementation

- A view is supposed to be always up-to-date. Why?
  - If we *modify* tuples in the base relation on which the view is defined, then the view must “automatically” reflect these changes.
  - The view should have to be realized or materialized
    - At the time of *specifying a query on the view* but not of defining the view
- The DBMS is responsible for keeping the view up-to-date
  - NOT the user to make sure that the view is newest.
  - Question: Then how does the DBMS let the view to be up-to-date?
    - Problem: not easy to “efficiently” implement a view for querying

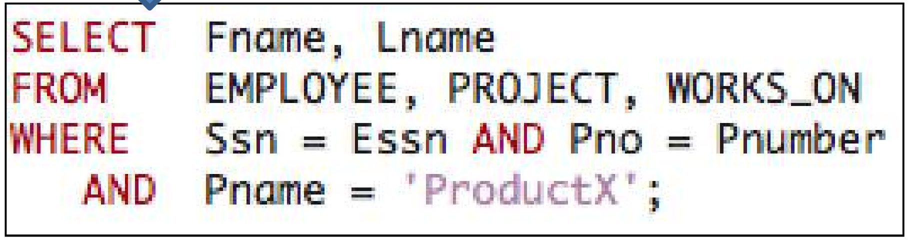
View Implementation (Cont'd)

- Strategy 1) **Query modification** approach
  - Compute the view as and when needed.
    - Let's not store the view permanently!
  - Modify view query into a query on underlying base tables.



```
SELECT Fname, Lname
FROM   WORKS_ON1 -- View V1
WHERE  Pname = 'ProductX';
```

transform



```
SELECT   Fname, Lname
FROM     EMPLOYEE, PROJECT, WORKS_ON
WHERE    Ssn = Essn AND Pno = Pnumber
    AND  Pname = 'ProductX';
```

- Any *problem*??
  - **Inefficient** for views defined via "complex" queries, which take so long, or *time-consuming*, to execute

View Implementation (Cont'd)

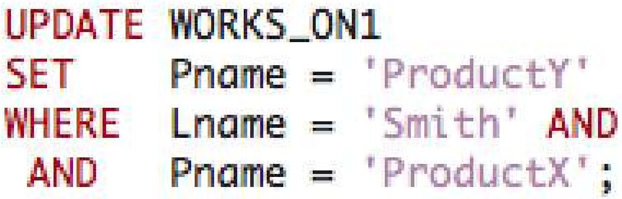
- Strategy 2) **View materialization** approach
  - *Physically create a temporary view table* when the view is *first* queried.
  - And keep that table on the assumption that other queries on the view will come later.
  - Requires efficient strategy for automatically updating the view table when the base tables are updated.
  - Incremental update strategy for materialized views
    - Means the DBMS can determine what new tuples must be inserted, deleted, or modified in a *materialized view table*
      - When a database update is applied *to one of the (defining) base tables*.
  - A materialized table is maintained as long as it is being queried.
    - If no query on the view for a certain period of time, then the table is automatically removed. Later recomputed from scratch, if accessed again.

View Implementation (Cont’d)

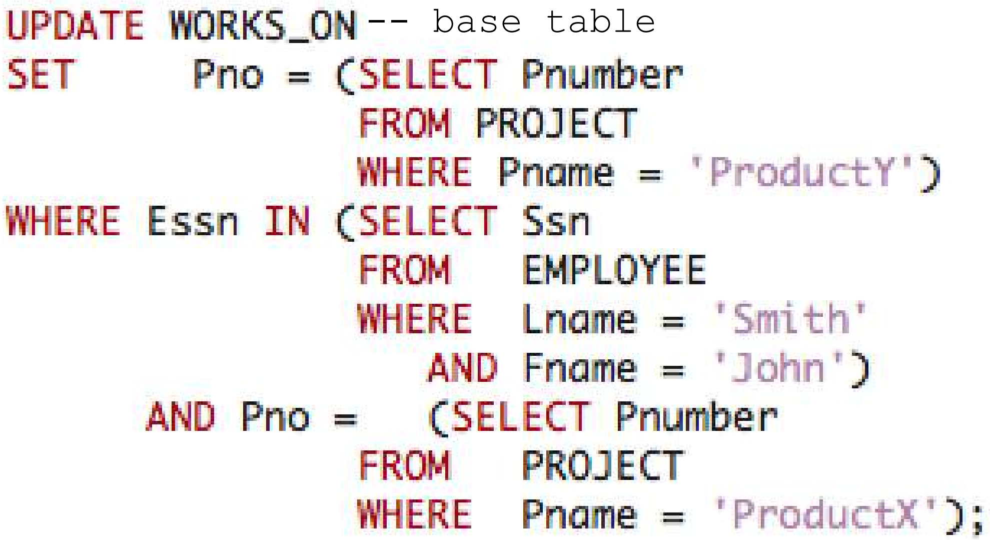
- Multiple ways to implement materialization:
  - *Immediate update* (c.f., “write-through”)
    - Updates a view *as soon as the base tables are changed*.
  - *Lazy update* (c.f., “write-back”)
    - Updates a view whenever a view query requests
  - *Periodic update*
    - Updates the view periodically
  - Note that in the *latter* strategy, a view query may not get an up-to-date result.
    - This is commonly used in Population Survey, Monthly Sale Record Retrievals, Banks, Retail store operations, etc.

View Update

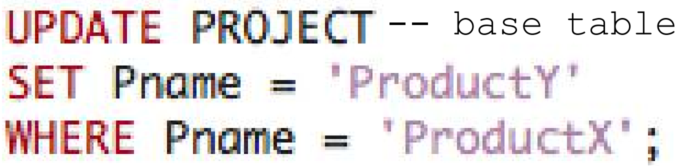
- In many cases, modifying a view table via `INSERT`/`DELETE`/ `UPDATE` command is not possible. Why? It may mean many...



```
UPDATE  WORKS_ON1
SET     Pname = 'ProductY'
WHERE   Lname = 'Smith' AND Fname = 'John'
 AND    Pname = 'ProductX';
```



```
UPDATE WORKS_ON -- base table
SET      Pno = (SELECT Pnumber
                FROM PROJECT
                WHERE Pname = 'ProductY')
WHERE Essn IN (SELECT Ssn
               FROM    EMPLOYEE
               WHERE   Lname = 'Smith'
                  AND  Fname = 'John')
      AND Pno =    (SELECT Pnumber
               FROM    PROJECT
               WHERE   Pname = 'ProductX');
```

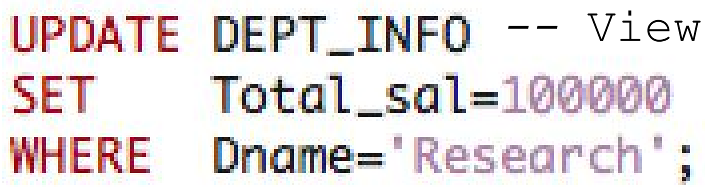


```
UPDATE  PROJECT -- base table
SET  Pname = 'ProductY'
WHERE  Pname = 'ProductX';
```

But Oracle says, "cannot modify a column which maps to a non key-preserved table" due to the existing tuple with the same attribute values as what the tuple to be updated has.

View Update (Cont'd)

- An update on a view defined on a *single table* without any *aggregate functions*
  - Can be translated to an update on underlying base table.
- What if there EXISTS such an aggregate function?

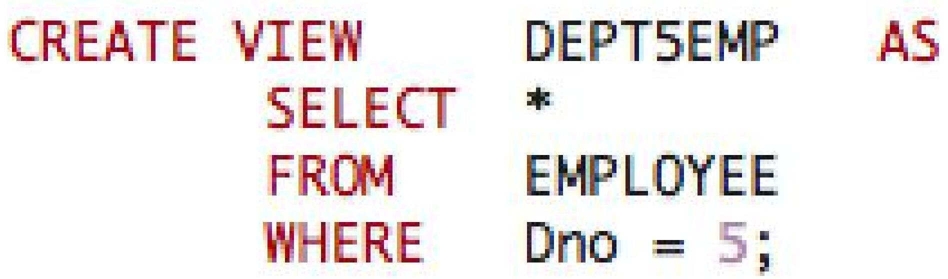


```
UPDATE DEPT_INFO -- View V2
SET    Total_sal=100000
WHERE  Dname='Research';
```

Do you think this update is permitted?
Why? Or Why ***NOT****?*

Views as Authorization Mechanism

- Suppose a certain user is only allowed to see employee information for employees that work for department 5.



```
CREATE VIEW      DEPT5EMP    AS
        SELECT   *
        FROM     EMPLOYEE
        WHERE    Dno = 5;
```

  - The DBA may grant to that user the privilege to query the view but not the base table `EMPLOYEE` itself.
  - This user then won’t be able to see other employee tuples when the view is queried, except the information of employees in DEPT #5.

- This way, view can be used to hide certain attributes or tuples from “unauthorized users.”

SCHEMA CHANGE STATEMENTS IN SQL

Chapter 7.4

Schema Evolution Commands

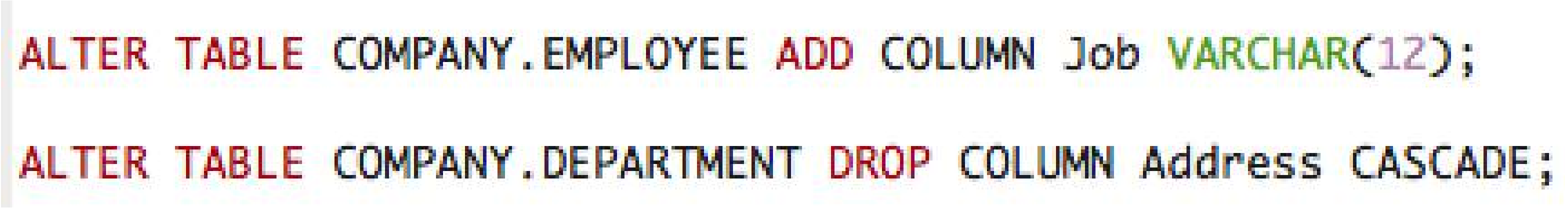
- Can be used to alter a schema by adding or dropping tables/views, attributes, constraints, and other schema constructs
  - Why? DBA may want to change the schema while the database is operational.
- Do not require recompiling the database schema;
  - Convenient, quick.
  - But ensures that the changes do not affect the rest of the database and make it consistent.

The **DROP** Command

- Used to drop named schema elements: tables, domains, or constraints
- DROP behavior options: **CASCADE** and **RESTRICT**
- Example
  - **DROP SCHEMA** COMPANY **CASCADE**; -- DON'T DO THIS UNLESS COMPLETE SURE
    - Removes the schema and all it s elements including tables, views, constraints, etc.
    - **RESTRICT**: Proceeds with the removal only if there's *no* element in it.
  - **DROP TABLE** DEPENDENT **CASCADE**;
    - Removes the relation and its definition from the catalog
    - **RESTRICT**: Proceeds with the removal only if no reference to it

The `ALTER TABLE` Command

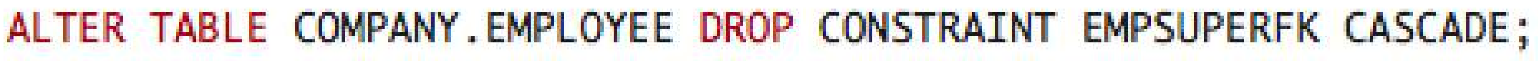
- The actions include
  - Adding or dropping a column (attribute)
  - Changing a column definition
  - Adding or dropping table constraints
- Example:



```
ALTER TABLE COMPANY.EMPLOYEE ADD COLUMN Job VARCHAR(12);

ALTER TABLE COMPANY.DEPARTMENT DROP COLUMN Address CASCADE;
```

- `CASCADE`: All constraints and views referencing the column (`Address`) are dropped with it.

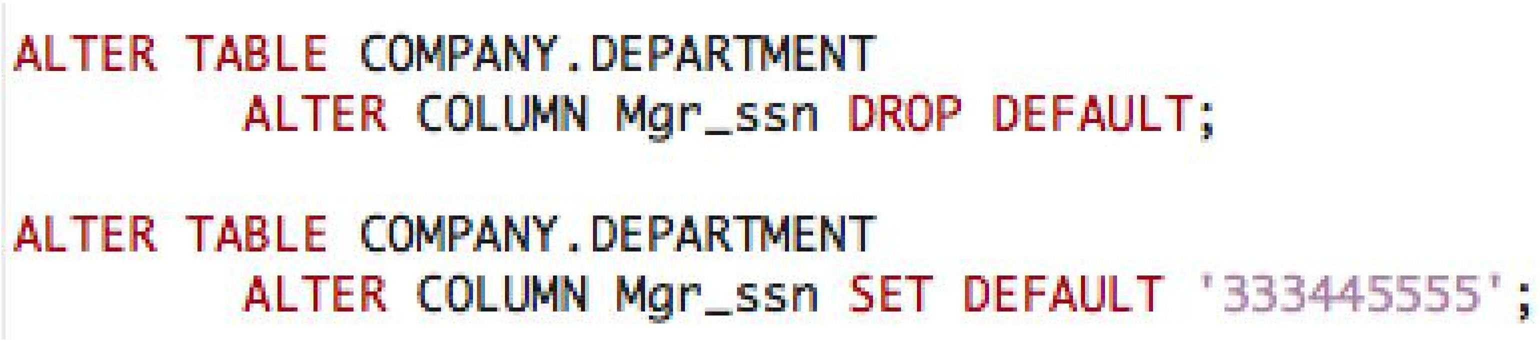
- `RESTRICT`: removes only if no views/constraints reference the column



```
ALTER TABLE COMPANY.EMPLOYEE DROP CONSTRAINT EMPSUPERFK CASCADE;
```

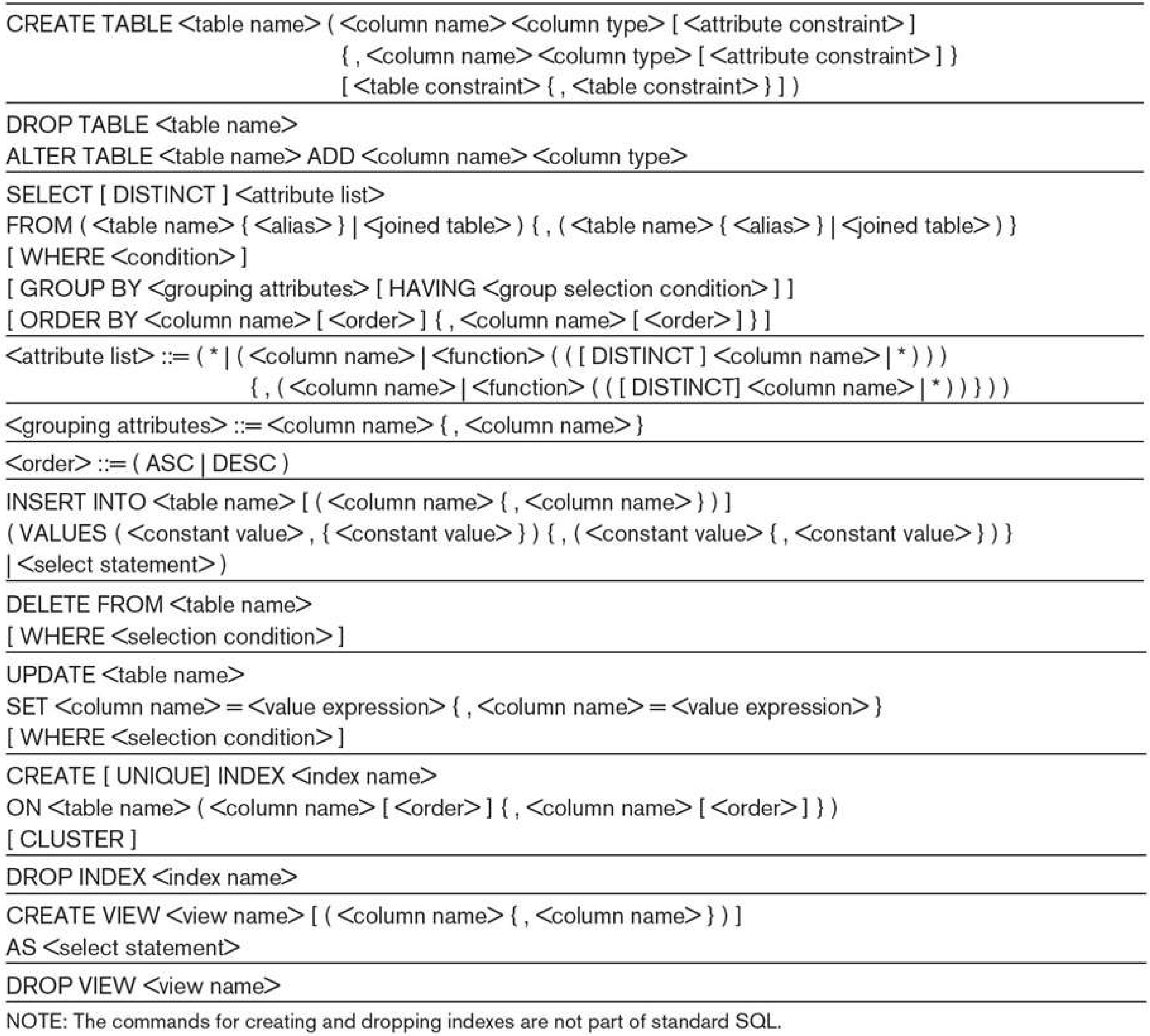
Default Values

- Can be dropped and altered:



```
ALTER TABLE COMPANY.DEPARTMENT
        ALTER COLUMN Mgr_ssn DROP DEFAULT;

ALTER TABLE COMPANY.DEPARTMENT
        ALTER COLUMN Mgr_ssn SET DEFAULT '333445555';
```

SUMMARY OF SQL SYNTAX



```
CREATE TABLE <table name> ( <column name> <column type> [ <attribute constraint> ]
                            { , <column name> <column type> [ <attribute constraint> ] }
                            [ <table constraint> { , <table constraint> } ] )

DROP TABLE <table name>
ALTER TABLE <table name> ADD <column name> <column type>

SELECT [ DISTINCT ] <attribute list>
FROM ( <table name> { <alias> } | <joined table> ) { , ( <table name> { <alias> } | <joined table> ) }
[ WHERE <condition> ]
[ GROUP BY <grouping attributes> [ HAVING <group selection condition> ] ]
[ ORDER BY <column name> [ <order> ] { , <column name> [ <order> ] } ]

<attribute list> ::= ( * | ( <column name> | <function> ( ( [ DISTINCT ] <column name> | * ) ) )
                     { , ( <column name> | <function> ( ( [ DISTINCT] <column name> | * ) ) } ) )

<grouping attributes> ::= <column name> { , <column name> }

<order> ::= ( ASC | DESC )

INSERT INTO <table name> [ ( <column name> { , <column name> } ) ]
( VALUES ( <constant value> , { <constant value> } ) { , ( <constant value> { , <constant value> } ) }
| <select statement> )

DELETE FROM <table name>
[ WHERE <selection condition> ]

UPDATE <table name>
SET <column name> = <value expression> { , <column name> = <value expression> }
[ WHERE <selection condition> ]

CREATE [ UNIQUE] INDEX <index name>
ON <table name> ( <column name> [ <order> ] { , <column name> [ <order> ] } )
[ CLUSTER ]

DROP INDEX <index name>

CREATE VIEW <view name> [ ( <column name> { , <column name> } ) ]
AS <select statement>

DROP VIEW <view name>
```

NOTE: The commands for creating and dropping indexes are not part of standard SQL.