Nested Queries (= Subqueries)

- A query that is part of another query is called a **subquery** (or **nested query**);

- Subquery can have many nested subquries as we want;

- There are many other ways that subqury can be used;

(1) Subquery can return constant value(s), and this value can be compared with another value in WHERE clause;

(2) Subquery can return relation (a set of tuples) that can be used in various ways in WHERE clause;

(3) Subquery can have their relations appear in FROM clause;

Nested Queries

- A nested query can be specified within the **WHERE**-clause of another query, called the outer query
- Single query :

```
SELECT   SSN
FROM     EMP,  DEPT
WHERE    (DNAME = 'research') AND DNO = DNUM
```

- Nested Query : Having executed nested query returns 'research' number; Then, outer query returns SSN(s) only if its DNO is equal to result of nested query.

```
SELECT   SSN
FROM     EMP
WHERE    DNO =
    (SELECT   DNUM
     FROM     DEPT
     WHERE    DNAME = 'research')
```

Outer query : Select a EMP tuple if its DNO is equal to result of nested query

Nested query : research dept number

- Question: What if there are more than one research dept numbers?

Nested Queries : IN

- A nested query can be compared with outer query by using **IN** operator;

- (*v* **IN** *V*) compares a value *v* (in outer query) with the value(s) *V* (in nested query), and returns "true" if *v* is included in *V*. Otherwise, return "false";

```
SELECT   SSN
FROM     EMP
WHERE    DNO
   IN
SELECT   DNUM
FROM     DEPT
WHERE    DNAME = 'research'
```

Outer query : Select an EMP tuple if its DNO is included in result of nested query)

Nested query : research dept number(s)

Nested Queries : IN

- SQL allows use of (sub)**tuples** of values in comparisons by placing them within parenthesis;

- Get SSNs of employees who work for the same (project, hours) on some project that employee with SSN = '123456789' works on;

```
SELECT    DISTINCT ESSN
FROM      WORK-ON
WHERE     (PNO, hours)
          IN
          SELECT    PNO, hours
          FROM      WORK-ON
          WHERE     ESSN = '123456789'
```

outer query : Select an EMP tuple if its (pno, hours) is included in result of nested query)

nested query : a set of (pno, hours) worked by '1234567'

Nested Queries : IN

- Get names of employees who work for the same (project, hours) on some project that employee with SSN = '123456789' works on;

```
SELECT   ename
FROM     EMP
WHERE    SSN
         IN
SELECT   ESSN
FROM     WORK-ON
WHERE    (PNO, hours)
         IN
SELECT   PNO, hours
FROM     WORK-ON
WHERE    ESSN = '123456789'
```

nested query : Select an EMP tuple name if its SSN is included in result of nested query

nested query : Select an EMP ESSN if its (pno, hours) is included in result of nested query)

nested query :
a set of (pno, hours) worked by '1234567'

Nested Queries : NOT IN

- (*v* **NOT IN** *V*) compares a value *v* (in outer query) with the value(s) *V* (in nested query), and returns "true" if *v* is not included in *V*. Otherwise, return "false";

- Retrieve names of employees with no dependents.

```
SELECT    ename
FROM      EMP
WHERE     SSN
          NOT IN
SELECT    ESSN
FROM      DEPENDENT
```

Outer query : Select an EMP tuple if its SSN is not included in result of nested query)

Nested query : SSN(s) with dependents

- Retrieve names of employees whose names are neither Smith nor Jones.

```
SELECT    ename
FROM      EMP
WHERE     ename
          NOT IN ('Smith', 'Jone')
```

Nested Queries : Use of Tuple Variables

- Nested Queries also can use tuple variables;

- Retrieve names of employees who have a dependent with the same sex as the employees.

```
SELECT   name
FROM     EMP AS e
WHERE    e.SSN
         IN
         SELECT   d.ESSN
         FROM     DEPENDENT AS d
         WHERE    e.sex = d.sex
```

outer query : Select a EMP tuple if its SSN is included in result of nested query.

nested query : Set of employee SSNs with the same sex as their dependents;

Nested Queries : SOME (= ANY)

- **SOME** (= **ANY**) can be used with {<, >, =, <=, >=, <>} operators:
  - For example, (*v* > **SOME** *V*) returns "true" if the value *v* is greater than **some** of the values in a set *V*.

- Get SSNs of employees whose salary is greater than salary of some employees in department 5:

```
SELECT   SSN
FROM     EMP
WHERE    salary
         > SOME
         (SELECT   salary
          FROM     EMP
          WHERE    DNO = 5)
```

outer query : Select an EMP tuple if its salary is greater than some values in result of nested query

nested query : a set of all salaries of employees who work for DNO = 5

Example : Meaning of SOME

- Example :
  - ✓ (5 < **SOME** {0, 5, 7}) = true
  - ✓ (5 < **SOME** {1, 3, 5}) = false
  - ✓ (5 = **SOME** {0, 5, 7}) = true
  - ✓ (5 <> **SOME** {0, 5, 7}) = true

- 참조: (= **SOME**)은 **IN** 과 동일한 표현임. 반면에, (<> **SOME**)은 **NOT IN** 과 동일하지 않음.

Nested Queries : ALL

- **ALL** can be used with {<, >, =, <=, >=, <>} operators:
  - For example, (*v* **>** **ALL** *V*) returns "true" if the value *v* is greater than **all** the values in a set *V*.

- Get SSNs of employees whose salary is greater than salary of all the employees in department 5:

```
SELECT    SSN
FROM      EMP
WHERE     salary
        > ALL
          SELECT    salary
          FROM      EMP
          WHERE     DNO = 5
```

outer query : Select an EMP tuple if its salary is greater than all values in result of nested query

nested query : a set of all salaries of employees who work for DNO = 5

Example : Meaning of ALL

- Example :
  - ✓ (5 < **ALL** {0, 5, 6}) = false
  - ✓ (5 > **ALL** {0, 3, 4}) = true
  - ✓ (5 = **ALL** {4, 5, 7}) = false
  - ✓ (5 <> **ALL** {4, 6, 8}) = true

- 참조: (<> **ALL**)은 **NOT IN** 과 동일한 표현임. 반면에, (= **ALL**)은 **IN** 과 동일하지 않음.

Nested Queries : EXISTS

- **EXISTS**(Q) returns "true" if there is **at least one** tuple (= **not empty**) in the result of a nested query Q. Otherwise, it returns "false"

- Retrieve SSNs of employees who have dependent(s).

```
SELECT    SSN
FROM      EMP
WHERE
```

outer query : For each employee, if result of its related nested query is not empty, select that employee SSN.

EXISTS

```
SELECT    *
FROM      DEPENDENT
WHERE     SSN = ESSN
```

nested query : For each EMP tuple, select all dependent tuples whose ESSN matches SSN.

Nested Queries : EXISTS

- Retrieve names of employees who have a dependent with the same sex as the employees.

```
SELECT    name
FROM      EMP AS e
WHERE
    EXISTS
        ( SELECT    *
          FROM    DEPENDENT AS d
          WHERE   (e.SSN = d.ESSN)
                AND (e.sex = d.sex) )
```

outer query : For each employee tuple, if result of its related nested query is not empty, select that employee name.

nested query :Set of dependent tuples with the same sex as their employees;

Nested Queries : NOT EXISTS

- **NOT EXISTS**(Q) returns "true" if there **no tuple** (= **empty**) in the result of a nested query Q. Otherwise, it returns "false".

- Retrieve SSNs of employees who have no dependent.

```
SELECT    SSN
FROM      EMP
WHERE
```

outer query : For each employee tuple, if result of its related nested query is empty, select that employee SSN.

NOT EXISTS

```
SELECT    *
FROM      DEPENDENT
WHERE     SSN = ESSN
```

nested query : For each EMP tuple, select all dependent tuples whose ESSN matches SSN.

Nested Queries : EXISTS

- Retrieve names of employees whose sex are not the same with any dependent or with no dependent.

```
SELECT     name
FROM       EMP AS e
WHERE
    NOT EXISTS
        ( SELECT    *
          FROM    DEPENDENT AS d
          WHERE  (e.SSN = d.ESSN)
                AND (e.sex = d.sex) )
```

outer query : For each employee tuple, if result of its related nested query is not empty, select that employee name.

nested query :Set of dependent tuples with the same sex as their employees;

Nested Queries : NOT EXISTS

- Retrieve names of employees whose salary are greater than salary of all the employees with age < 25.

```
SELECT    e1.ename
FROM      EMP AS e1
WHERE
      NOT EXISTS
```

```
SELECT      *
FROM      EMP AS e2
WHERE   e2.age < 25 AND
            e1.salary <= e2.salary
```

- Question: Suppose that age of all employees are >= 25; Then, what is the query result?

Division Query : NOT EXISTS and EXCEPT

- Retrieve name of each employee who work on **all** the projects controlled by department 5.

```
SELECT    name
FROM      EMP
WHERE
```

outer query : For each employee, if query 1 – query 2 is empty (that means, that employee works all projects by dept 5) then the answer is selected

NOT EXISTS

```
SELECT    Pnumber
FROM      PROJECT
WHERE     DUM = 5
```

nested query 1 : Select all project numbers controlled by dept 5

EXCEPT

```
SELECT    PNO
FROM      WORK-ON
WHERE     SSN = ESSN
```

nested query 2 : Select all project numbers performed currently by each employee

EXISTS and NOT EXISTS

- **INTERSECT** vs **EXISTS**

```
( SELECT   R.A, R.B
  FROM     R  )
    INTERSECT
( SELECT   S.A, S.B
  FROM     S)
```

=

```
SELECT   R.A, R.B
FROM     R
WHERE
   EXISTS
     ( SELECT   *
       FROM     S
       WHERE  R.A=S.A
            AND R.B=S.B )
```

- **EXCEPT** vs **NOT EXISTS**

```
( SELECT   R.A, R.B
  FROM     R  )
    EXCEPT
( SELECT   S.A, S.B
  FROM     S)
```

=

```
SELECT   R.A, R.B
FROM     R
WHERE
   NOT EXISTS
     ( SELECT   *
       FROM     S
       WHERE  R.A=S.A
            AND R.B=S.B )
```

Aggregate Functions

- Aggregate Functions
  - COUNT : count the number of values
  - SUM : sum of values
  - AVG : average of values
  - MAX : maximum of values
  - MIN : minimum of values

- These functions can be used in **SELECT** clause;

COUNT (*)
COUNT ([**DISTINCT**] A)
SUM ([**DISTINCT**] A)
AVG ([**DISTINCT**] A)
MAX (A)
MIN (A)

Aggregate Functions

- Count the number of employees in our company.

```
SELECT   COUNT(*)
FROM     EMP
```

  - Any tuples with any NULL values are counted.

- Count the number of salaries of employees.

```
SELECT   COUNT(salary)
FROM     EMP
```

  - Any tuples with duplicate salary values are counted.
  - Any tuples with NULL salary values are not counted.

- Count the number of distinct salaries of employees.

```
SELECT   COUNT(DISTINCT salary)
FROM     EMP
```

  - Any tuples with duplicate salary values are not counted.
  - Any tuples with NULL salary values are not counted.

Aggregate Functions

- Get total of salaries of employees working in dept number 5.

```
SELECT   SUM(salary)
FROM     EMP
WHERE    DNO = 5
```

- Get total, max, avg. of salaries of employees working in research dept.

```
SELECT   SUM(salary), MAX(salary), AVG(salary)
FROM     EMP JOIN DEPT ON DNO = DNUM
WHERE    Dname = 'Research'
```

- In general, NULL values are ignored when aggregate functions are applied to particular column(s).

- But NULL values are not ignored when COUNT(*) are applied.

Nested Queries : Aggregate Functions

- Aggregate functions can also be used in SELECT clause conditions involving nested queries.

- We can specify a nested query with aggregate function and then use the nested query in WHERE clause of outer query.

- Get names of employees who have two or more than dependents.

```
SELECT    ename
FROM      EMP
WHERE
    ( SELECT    COUNT(*)
      FROM      DEPENDENT
      WHERE     SSN = 'ESSN  ) >= 2
```

- The nested query counts the number of dependents that each employee has. If the count >= 2, the employee is selected.

Aggregate Functions

- Find name and age of the oldest employee(s).

- Q1 is incorrect!

Q1 :

```
SELECT  e.name, MAX(e.age)
FROM    EMP AS e
```

- Q2 is correct!

Q2 :

```
SELECT  e1.name, e1.age
FROM    EMP AS e1
WHERE   e1.age =
        ( SELECT  MAX (e2.age)
          FROM  EMP AS e2 )
```

- Note : Tuple variable e1 and e2 refer each copy of EMP, respectively.

GROUP BY

- We want to apply the aggregate functions to subgroups of tuples in a relation
- **GROUP BY** partitions the table into disjoint groups of tuples based on grouping attributes(s).
- Each group consists of the set of tuples that have the same value for the grouping attribute(s)
- We can then apply the aggregate function to each group independently.
- The grouping attributes must also appear in **SELECT**-clause.

GROUP BY

- For each department, retrieve DNO, number of employees in the department, and their average salary.

```
SELECT      DNO, COUNT (*), AVG (salary)
FROM        EMP
GROUP BY    DNO
```

  - Each employee group has the same value for the grouping attribute DNO.
  - Note : Group attribute 'DNO' must appear in SELECT clause
  - COUNT and AVG functions are applied to each such group.

GROUP BY

- For each project, retrieve the project number, project name, and the number of employees who work on that project.

```
SELECT      PNUM, PNAME, COUNT (*)
FROM        PROJECT, WORK_ON
WHERE       PNUM = PNO
GROUP BY    PNUM
```

  - In this case, a join condition is used together with grouping.

  - The grouping and functions are applied after the joining of the two relations.

- If null values exist in the grouping attribute, then separate group is created for all tuples with a NULL value for the grouping attribute.

GROUP BY

WORK-ON

SSN	PNO	hours
11111	p1	15
11111	p2	20
22222	p1	18
22222	p2	25
22222	p3	10
33333	p2	30
33333	p3	40
44444	p3	30
55555	p3	20

PROJECT

PNUM	pname	budget
p1	laptop	500M
p2	printer	700M
p3	mp3	400M
p4	chip	800M

앞 query 결과

PNUM	pname	count(*)
p1	laptop	2
p2	printer	3
p3	mp3	4

HAVING

- Sometimes, we want to retrieve the values of these functions for only those *groups that satisfy certain conditions*

- **HAVING** is used for specifying a selection condition on groups. (rather than on individual tuples)

- For each project on which more than two employees work, retrieve the project number, project name, and the number of employees who work on that project.

```
SELECT      PNUM, PNAME, COUNT(*)
FROM        PROJECT, WORK-ON
WHERE       PNUM = PNO
GROUP BY    PNUM, PNAME
HAVING      COUNT(*) > 2
```

HAVING

WORK-ON

SSN	PNO	hours
11111	p1	15
11111	p2	20
22222	p1	18
22222	p2	25
22222	p3	10
33333	p2	30
33333	p3	40
44444	p3	30
55555	p3	20

PROJECT

PNUM	pname	budget
p1	laptop	500M
p2	printer	700M
p3	mp3	400M
p4	chip	800M

앞 query 결과

PNUM	pname	count(*)
p2	printer	3
p3	mp3	4

HAVING, GROUP BY and Nested Queries

- Find the project number with the highest average working hours.

```
SELECT    PNO
FROM      WORK-ON
GROUP BY  PNO
HAVING    AVG(hours)
      >= ALL
SELECT    AVG(hours)
FROM      WORK-ON
GROUP BY  PNO
```

WORK-ON

SSN	PNO	hours
11111	p1	15
11111	p2	20
22222	p1	18
22222	p2	25
22222	p3	10
33333	p2	30
33333	p3	40
44444	p3	30
55555	p3	20

- SQL cannot compose of aggregate functions (function of a function), so it must be written in a certain way.

HAVING, GROUP BY and Nested Queries

- Find the age of youngest employee with age > 35 for each department with at least 20 employees of any age.

```
SELECT      e1.DNO, MIN(e1.age)
FROM        EMP AS e1
WHERE       e1.age > 35
GROUP BY    DNO
HAVING
```

```
    >= 20
```

```
SELECT    COUNT(*)
FROM      EMP AS e2
WHERE     e1.DNO = e2.DNO
```

ORDER BY

- **ORDER BY** is used to sort the tuples in a query result based on the values of some attribute(s);

  - **ASC** (usually, omitted) : increasing order
  - **DESC** : decreasing order

- ```
  SELECT    salary, SSN
  FROM      EMP
  ORDER BY  salary, SSN
  ```

- ```
  SELECT    salary, SSN
  FROM      EMP
  WHERE     age > 35
  ORDER BY  salary,  DESC SSN
  ```

Summary of SQL Queries

- SQL Query consists of 6 clauses. [ ] is optional.

```
SELECT      <attribute list>
FROM        <table list>
[ WHERE     <condition>]
[ GROUP BY  <grouping attributes>]
[ HAVING    <group condition>]
[ ORDER BY  <attribute list>]
```

- Order of query is evaluated conceptually as follows:
  - First, evaluate the FROM-clause,
  - Then, evaluate the WHERE-clause,
  - Then, GROUP BY and HAVING, and
  - Finally, the SELECT-clause

- However, this method may be inefficient in real systems;
  - Each DBMS has its own query optimization;

INSERT

- **Single** tuple Insertion

  INSERT INTO table name
  VALUES attribute values

- Attribute value들은 table에 정의된 attribute들의 순서와 일치가 되어야 함.

- **Multiple** tuples Insertion

  INSERT INTO table name
  (**SELECT FROM WHERE**)

- 이 경우 SELECT query 결과를 저장하기 위한 새로운 임시 table (**CREATE TABLE**을 이용)를 미리 만들어 야 함.

INSERT : Single Tuple

- Insert a new project tuple into PROJECT table;

  INSERT INTO PROJECT

  VALUES (‘notebook', ‘1234', ‘LA', ‘D3’)

- Insert another new project tuple into PROJECT table;

  INSERT INTO PROJECT (PNAME, PNO, DNO)

  VALUES (‘memory', ‘2345', ‘2')

- 단, 이 project의 location은 알려지지 않았음
- 이 경우 각 값들과 상응하는 attribute들을 명시해야 함.

INSERT : Multiple Tuples

- Insert DEPT tuples with the name, number of employees, and total salaries for each department; In this case, we have to prepare a temporary table.

```
CREATE TABLE  DEPT-INFO
   (    Dept-Name        VARCHAR(10),
        No-of-Emps       INTEGER,
        Total-Sal        INTEGER    );
```

```
INSERT INTO    DEPT-INFO (Dept-Name, No-of-Emps, Total-Sal)
SELECT         Dname, COUNT (*), SUM(Salary)
FROM           DEPT  JOIN  EMP  ON  Dnumber = DNO
GROUP BY       Dname
```

- We can now query DEPT-INFO as any other tables;
- Note that if we update the tuples in EMP or DEPT, DEPT-INFO may not be up-to-date.

DELETE

- Removes tuple(s) from a table.

  DELETE FROM table name
  [**WHERE** [**SELECT FROM WHERE**]]

- A missing WHERE-clause specifies that all tuples in the table are to be deleted;
  - The table then becomes an empty, but the table itself still exists;

- Tuples are deleted from only *one table* at a time.
  (unless CASCADE is specified on a referential constraint)

- Referential integrity should be verified.

DELETE

- **DELETE FROM** EMP
  WHERE age > 60

- **DELETE FROM** EMP
  WHERE DNO **IN**
  (**SELECT** DNUMBER
  FROM DEPT
  WHERE DNAME = 'research')

  - All employees tuples in 'research' department are deleted.

- **DELETE FROM** EMP

  - All employee tuples are deleted, but, the EMP table itself (= definition) still exists.

UPDATE

- Used to modify attribute values of one or more selected tuples
- **WHERE**-clause selects the tuples to be modified.
- **SET**-clause specifies the attributes to be modified and their new values
- Each update command modifies tuples in a single relation.
- If primary key value is modified, referential constraints should be verified
- Change the location and controlling department number of project number 10 to ‘Chicago' and 5, respectively.

```
UPDATE    PROJECT
SET       PLOCATION = ‘Chicago', DNUM = 5
WHERE     PNUMBER =10
```

UPDATE

- Give employees in the ‘research' department a 10% raise in salary.

```
UPDATE      EMP
SET         salary = salary * 1.1
WHERE  DNO  IN
      (  SELECT     DNUMBER
         FROM       DEPT
         WHERE      DNAME = 'Research‘  )
```

- Salary column on the right side of = refers to the old SALARY value before update.

- SALARY column on the left side of = refers to the new SALARY value after update

Views

- A view is a **single** table that is derived from other table(s);
- These other tables can be **base tables**, or previously defined **views**;
- A view is defined using **CREATE VIEW** as follows;

  CREATE VIEW View Name **AS**

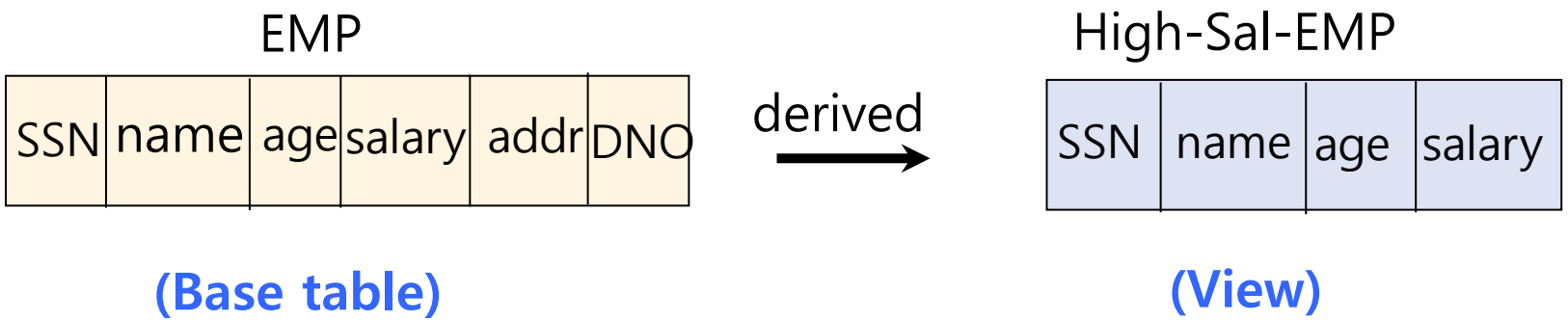
  SELECT FROM WHERE

- We can define a view as a table that we reference "frequently";
- A view is considered a **virtual table** because view does not necessary physically exists;
- The view is realized at request time for querying the view.

Views : From Single Table

- We only want to keep information with SSN, name, salary, and age of employees with high salaried (say, > $50,000) employees. (We don't want about all salaries.)

```
CREATE VIEW   HIGH_SAL_EMP   AS
    SELECT    SSN, name, age, salary
    FROM      EMP
    WHERE     salary > 50000
```



Querying Views

- The view "High-Sal-EMP" does not contain tuples in the usual sense. Rather, if we query this view, the corresponding tuples are obtained from the base table "EMP".

- Retrieve the names of high salaried employees with age < 45.

```
SELECT   name
FROM     High-Sal-EMP
WHERE    age < 45
```

- This query is translated into underlying base table as follows.

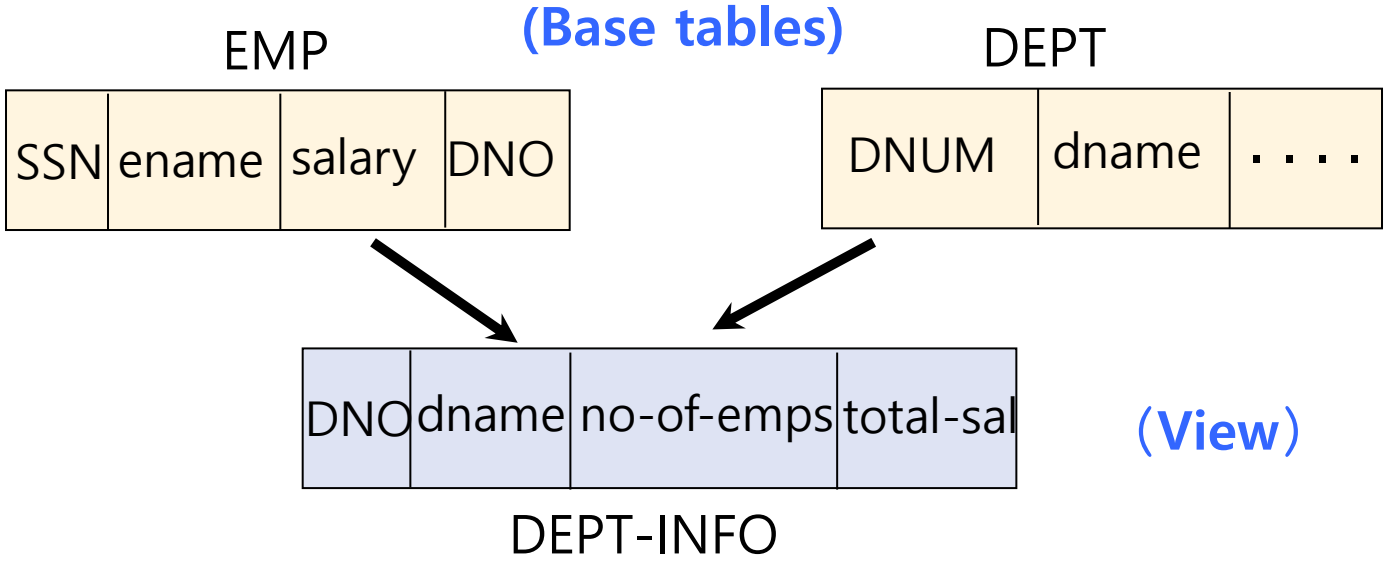
```
SELECT   name
FROM     EMP
WHERE    age < 45 AND salary > 50000
```

- As a result, we can ask the same query about "High-Sal-EMP" twice or more, we may get different answers, because base table "EMP" may have changed in the interim.

Views : From Multiple Tables

- We only want to keep information with DNO, dept. name, number of employees, and total salary for each department;

```
CREATE VIEW   DEPT-INFO (DNO, dname, no-of-emps, total-sal)  AS
      SELECT      DNO, dname, COUNT(*), SUM(salary)
      FROM        DEPT, EMP
      WHERE       DNUM = DNO
      GROUP BY    DNO
```

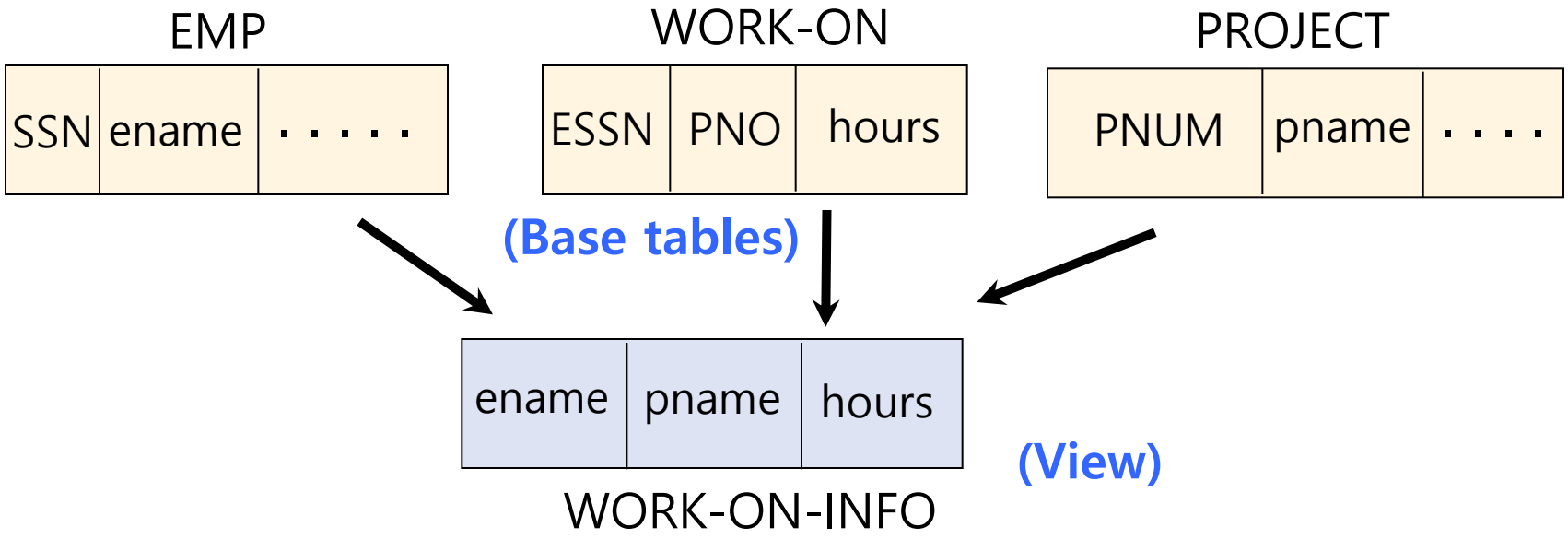


- Note: COUNT(*), SUM(salary) are renamed as no-of-emps, total-sal.

Views : From Multiple Tables

- We only want to keep WORK-ON information with employee name, project name, and hours.

```
CREATE VIEW    WORK-ON-INFO    AS
    SELECT     ename, pname, hours
    FROM       EMP, PROJECT, WORK-ON
    WHERE      (SSN = ESSN) AND (PNO = PNUM)
```



Advantages of Views

- By using views, we can specify queries "more concisely".

- Retrieve names of employees who work on the project 'notebook':

```
SELECT    ename
FROM      WORK-ON-INFO
WHERE     pname = 'notebook'
```

- This query is equivalent to the following query;

```
SELECT    ename
FROM      EMP, WORK-ON, PROJECT
WHERE     (SSN = ESSN)  AND
          (PNO= PNUM)  AND
          (pname = 'notebook')
```

  - In case of not using view, we need to specify two joins on the base tables; This is more complex!

Advantages of Views

- Retrieve the number of working employees and total salary for department 5.

(Using view)

```
SELECT    no-of-emps, total-sal
FROM      DEPT-INFO
WHERE     DNO = 5
```

(Without using view)

? ?

Modifying Views

- Sometimes, users need to modify (insert, delete, update) views;
  - Insert a new tuple into a view
  - Delete some tuple from a view
  - Update some tuple from a view

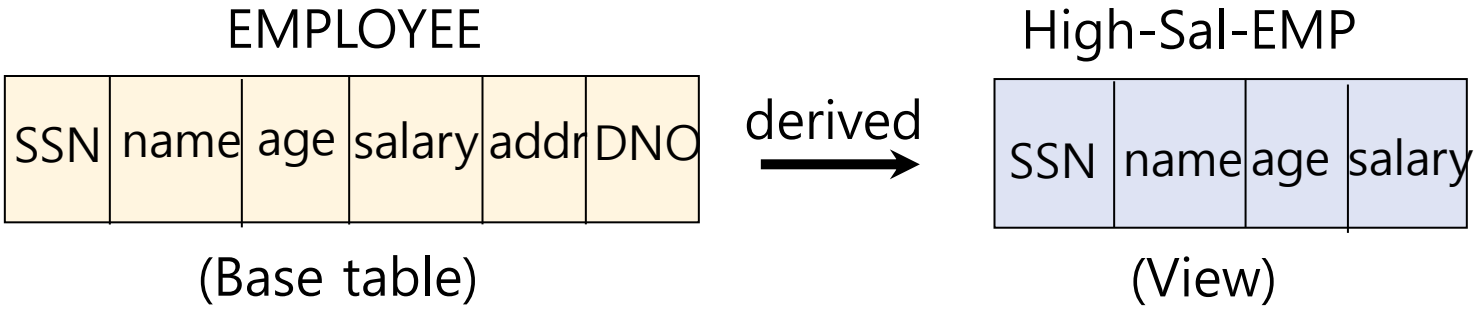
- Is it possible to modify views?
  (Note : Views does not exist physically, but base table does.

  - Answer : Yes, but only for some cases; Very restricted!

- For simple views (called, "**updatable** views"), it is possible to translate the modification of the view into equivalent modification on the underlying base table.

Modifying Views

- Consider the following view;



- Increase salary for high salaried employees with age >= 60 by 5%.

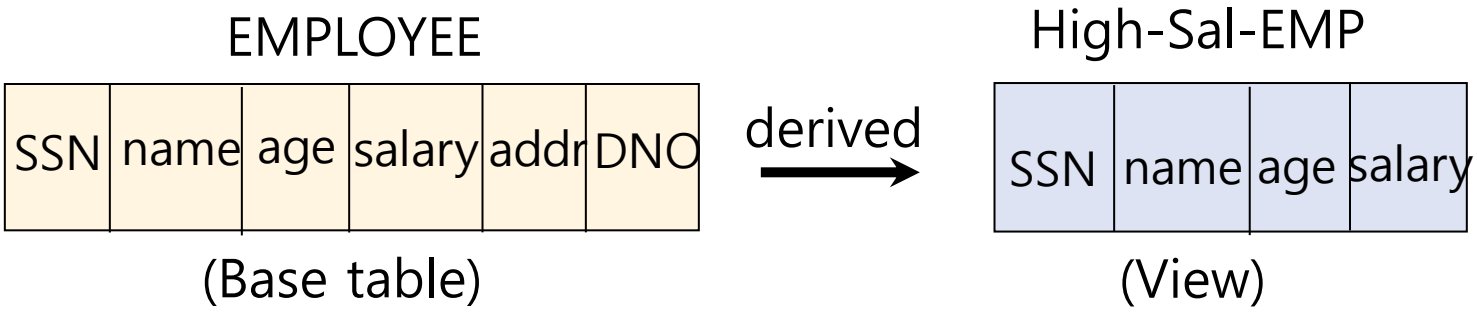
```
UPDATE   High-Sal-EMP
SET      salary = salary * 1.05
WHERE    age >= 60
```

- Is this view update possible? "Yes", because we can actually modify base table EMP through the view; Thus, this view update is translated as follows;

```
UPDATE   EMP
SET      salary = salary * 1.05
WHERE    age >= 60 AND salary > 50000
```

Modifying Views

- Consider the following view;



- Delete all high salaried employees with salary >= 100,000.

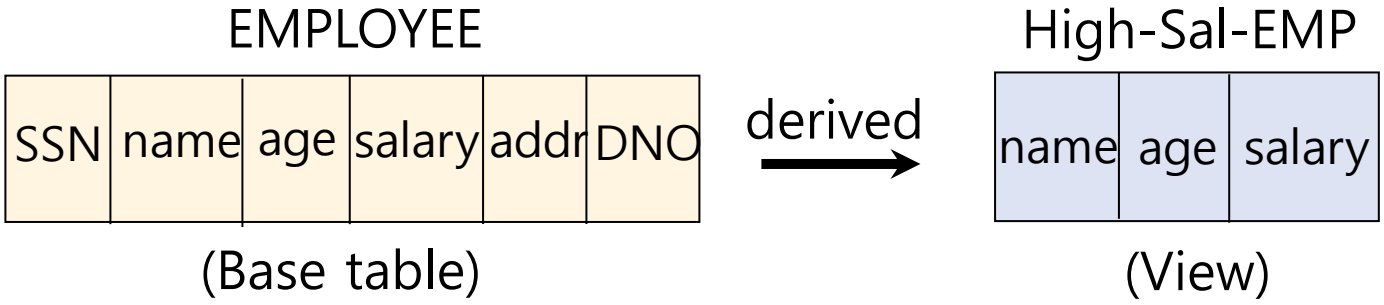
```
DELETE
FROM    High-Sal-EMP
WHERE   salary >= 100000
```

- Is this view update possible? "Yes", because we can actually modify base table EMP through the view; Thus, this view update is translated as follows;

```
DELETE
FROM    EMP
WHERE   salary > 50000 AND salary >= 100000
```

Modifying Views

- Consider the following view;



- Insert a new high salaried employee as follows;

  INSERT INTO High-Sal-EMP
  VALUES (Bob, 45, 60000)

- Is this view update possible? "No". Why? This view update is translated as follows; But this insertion is impossible, because primary key SSN has a NULL value.

  INSERT INTO EMP
  VALUES (NULL, Bob, 45, 60000, NULL, NULL)

View Update : Multiple Tables

- Update bob's current project name from 'printer' to 'laptop'.

```
UPDATE    WORK-ON-INFO
SET       pname = 'laptop'
WHERE     (ename = 'bob') AND (pname = 'printer')
```

(Base tables)

PROJECT

PNUM	pname
p1	printer
p2	memory
p3	laptop

WORK-ON

ESSN	PNO	hours
11111	p1	20
11111	p2	30
33333	p1	15
44444	p3	30

EMP

SSN	ename	age
11111	bob	22
22222	ann	18
33333	jim	33
44444	eve	27

WORK-ON-INFO

ename	pname	hours

(View)

View Update

- Is view update possible? No! It has two possible translations:

  1) Find bob's current PNO (say, p1), then find its pname (say, printer), then, change it by 'laptop'

  or

  2) Find laptop's PNUM (say, p3) and then find bob's current working project (say, printer, p1), and then, change it by 'p3'.

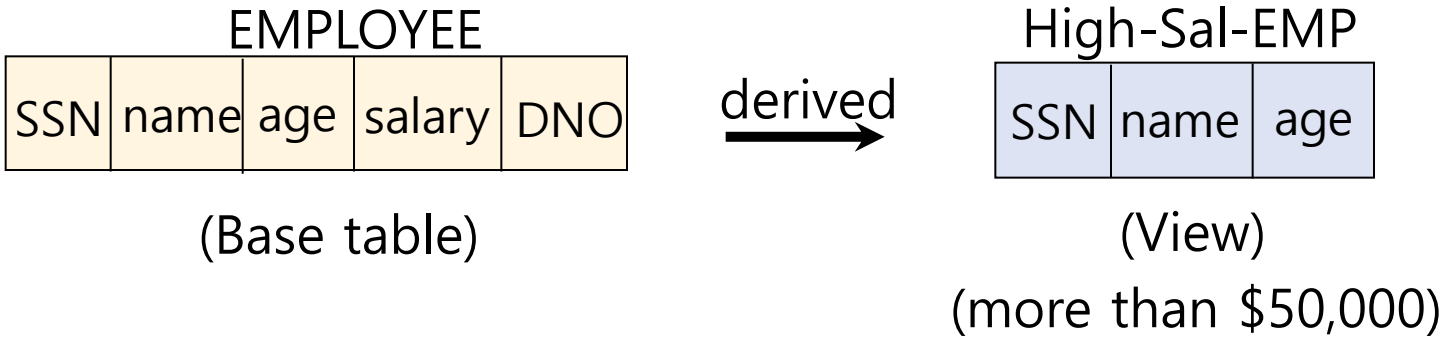
- This view update is ambiguous because it can not be translated uniquely:

- DBMS seems not smart to decide which one is better!

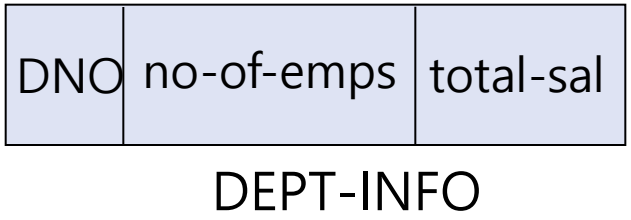
  In this example, option 2) looks better.

Modifying Views: Exercise

- Is each of the following view updates possible? (yes/no)
  - Delete all employees with age > 65 from High-Sal-EMP; **Yes!**
  - Insert <123456789, bob, 30> into High-Sal-EMP; **No!**
    (For null value in salary, we don't know bob's salary)



  - Increase total salary of for department 5 by 10%; **No!**



Un-updatable Views

- View update는 일반적으로 다음의 경우에는 update 불가.

✓ Aggregate 함수(sum, avg, group by 등)를 사용한 view

✓ 2 개 이상의 table들로 (join 등을 이용) 부터 유도된 view

✓ View의 **SELECT** clause에 Primary key가 명시되지 않은 경우

✓ 기타의 경우 : 사례별로 봐야 함.

- We can specify with **CHECK** option:
  - Must be added to the view definition if the view can be updated
  - To allow check for updatability and to plan for an execution strategy

Maintaining View: Up-to-Date

- A view always must maintain *up-to-date* information.
- If we update tuples in the base tables (on which the view is defined), the view must automatically reflect the changes.
- In general, view is not realized at the time of *view definition*, but rather at the time *we specify a query on the view*.
- Maintaining a view as up-to-date is the responsibility of DBMS (not the user)

View Implementation

(1) Query Modification (Most widely used)

- A view is not physically stored.
- Present the view query in terms of a query on the underlying base tables
- A view is computed at the time users specify *a query on the* view.
- DBMS must modify the user's view query into a query on its underlying base tables.
- Query processing is inefficient for views defined via complex queries:
  - Especially if additional queries are to be applied to the view within a short time period

View Implementation

(2) View Materialization

- A view is pre-computed and physically stored as keeping a temporary table.

- When a query is requested on the view, the unmodified query is executed directly on the pre-computed result.

- This is much faster than query modification; Faster access for expensive and complex joins

- Support applications that do not require current data or require data valid at a specific point in time (snapshot data).

- A major drawback is that we must maintain the consistency between the base table and the view when the base table is updated; Cost of maintaining view is high!

Advantages/Disadvantages of View

- Advantages of View

  (1) **Convenience**

  - A view can show data from many tables by single table;
  - Users construct multiple table query by **single table query**

  (2) **User View**

  - Each user can have his(her) own **personalized** view;
  - Provide data independence

  (3) **Security**

  - A view provides a mechanism to **hide certain data** from the view of certain users.

- Disadvantages of View

  (1) **Performance**

  - Query processing for view is **inefficient**;

  (2) **Update Restrictions**

  - Modifying view is **restricted** in most cases;

Checking Integrity Constraints

- **CHECK**
  - Attribute-based Constraints
  - Tuple-based Constraints

- **ASSERTION**
  - Specify additional types of constraints outside scope of built-in relational model constraints
  - Use only in cases where it is not possible to use CHECK on attributes and domains

- **TRIGGER**
  - Specify automatic actions that database system will perform when certain events and conditions occur

Attribute-based Constraints

- SQL provides constraints on the particular **attribute**;
- We can add **CHECK** (<condition>) for the attribute.
- Every employee's salary must be less than $80,000;

```
CREATE TABLE EMP
  ( SSN     CHAR(9),
    name    CHAR(20),
    age     INT,
    salary  REAL,   CHECK (salary < 80000) );
```

- Attribute-based checking is performed when a value for that attribute gets a new value (by **UPDATE** or **INSERT**)

  - **CHECK** (salary < 80000) checks every new salary value and rejects the modification for that tuple if the salary >= $80,000.

Tuple-Based Constraints

- SQL provides constraints on the particular tuples;
- **CHECK** (<condition>) can be added as a table element;
- This condition can refer to any attributes of the table.
- Salary of employees with age > 60 can be greater than $80,000;

```
CREATE TABLE EMP
    ( SSN     CHAR(9),
      name    CHAR(20),
      age      INT
      salary   REAL
      CHECK (age > 60) OR (salary < 80000) );
```

- Tuple-based checking is performed when a tuple is inserted or updated.
  - **CHECK** (. . . .) checks every new tuple; It is false if any tuples with (age <= 60) and (salary >= 80000) exist; Thus, rejected!

Assertions

- User는 더 일반적인 constraints들을 다음과 같이 명시할 수 있음.

  CREAT ASSERTION constraint name
  CHECK condition (**SELECT FROM WHERE**)

- **ASSERTION** is a condition which must be true at all times;

- (Difference with) Tuple-based Checking
  - Tuple-based checking refers to only the attributes of the table in whose declaration they appear;
  - But, by using assertions, we can refer any attributes (even if other tables) specified in the condition;.

Assertions

- "Total hours of projects worked by each employee must be less than 50 hours"

```
CREATE ASSERTION  Working-Total Hours
    CHECK
      (
          50 >= ALL
            ( SELECT  SUM(hours)
              FROM    WORK-ON
              GROUP BY  ESSN )
      )
```

- This condition refers to single table.

Assertions

- "Average hours of projects worked by each employee must be greater than 10 hours"

```
CREATE ASSERTION  Working-Average-Hours
    CHECK
      ( NOT EXISTS
          ( SELECT  *
            FROM    WORK-ON
            GROUP BY  ESSN
            HAVING  10 >= AVG(hours)
          )
      )
```

- This condition refers to single table.

Assertions

- "Salary of each employee must not be greater than the salary of the manager of the department that the employee works for.''

```
CREAT ASSERTION  Salary-Constraint
CHECK
    (
       NOT EXISTS
            (
                SELECT    *
                FROM     EMP AS e, EMP AS m, DEPT AS d
                WHERE   (e.salary > m.salary) AND
                             (e.DNO = d.DNUMBER) AND
                             (e.SSN = d.Mgr-SSN )

            )
     )
```

- This condition refers to multiple tables.

Assertions

- "Any bar can not sell beers by more than average price $5."

```
CREATE ASSERTION Cheap-Bars
CHECK
    (
      NOT EXISTS
        ( SELECT  bar
          FROM    SELL
          GROUP BY  bar
          HAVING  5.00 < AVG(price)
        )
    )
```

SQL Triggers

- Trigger always monitors a database and take action when a condition occurs
- It is called ECA(Event Condition Action) rule.
  - Event : Insert, Delete, Update
  - Condition : SQL Boolean-valued expression.
  - Action : SQL statements
- (1) When event occurs, test condition.

  (2) If condition is satisfied, execute action.

Triggers

- A trigger to create a new separate table for old aged new customers if a new customer's age is greater than 65.

```
CREATE TRIGGER   OldAge-New-Customer
AFTER INSERT ON  Customer
REFERENCING NEW TABLE  New-Customer
FOR EACH ROW
    INSERT INTO  OldAge-New-Customer
    SELECT
    FROM     New-Customer AS N
    WHERE  N.age >= 65
```

Triggers

- A trigger to increase a new employee's salary by 10 if his/her salary is less than 30,000.

```
CREATE TRIGGER   RaiseSalary
AFTER INSERT ON EMP
REFERENCING NEW ROW AS NewEMP
FOR EACH ROW
    WHEN  (NewEMP.Salary < 30000)
UPDATE  EMP
SET  NewEMP.Salary = Salray * 1.1
WHERE  SSN = NewEMP.SSN
```