Chapter 7: More SQL

Content

- More complex features of SQL retrieval queries
  - Nested queries aggregate functions, grouping
- Defining VIEWs on the database
- DROP and ALTER TABLE statement
- CREATE ASSERTION statement, CREATW TRIGGER statement

Aggregate Functions in SQL

- Aggregate functions
  - Compute an **aggregate** value from a **set of values**
  - The **aggregate functions** available in SQL are:

1. **SUM( )**: sums a set of values
2. **AVG( )**: takes the average of a set of values
3. **MAX( )**: finds the maximum value of a set of values
4. **MIN( )**: finds the minimum value of a set of values
5. **COUNT( )**: returns the number of elements in a set (**duplicates** are counted multiple times !!!)
6. **COUNT(DISTINCT ...)**: returns the number of **distinct** elements in a set (***duplicates* are counted *ONCE***)

Applying Set of Function on a Set of Values

- Syntax:
  - SELECT AggFunc(attribute)
  - Apply the aggregate function **AggFunction** on the selected set of tuples returned by the SELECT command

```
SELECT salary
FROM  employee
```

SALARY
30000
40000
25000
43000
38000
25000
25000
55000

```
SELECT SUM(salary), AVG(salary), MAX(salary),
       MIN(salary), COUNT(salary), COUNT( DISTINCT salary )
FROM   employee
```

SUM(SALARY)	AVG(SALARY)	MAX(SALARY)	MIN(SALARY)	COUNT(SALARY)	COUNT(DISTINCT salary)
281000	35125	55000	25000	8	6

Using the Result of Aggregate Functions in Queries

- Query
  - Find the fname and lname of the employee in the **'Research' department** that earns more than the average salary within the 'Research' department

```
SELECT   fname, lname
FROM     employee
WHERE    salary > ( SELECT  AVG(salary)
                    FROM   employee, department
                    WHERE  dno = dnumber AND  dname = 'Research')
```

Correct? No!

- Reas
  - Th f the Research de

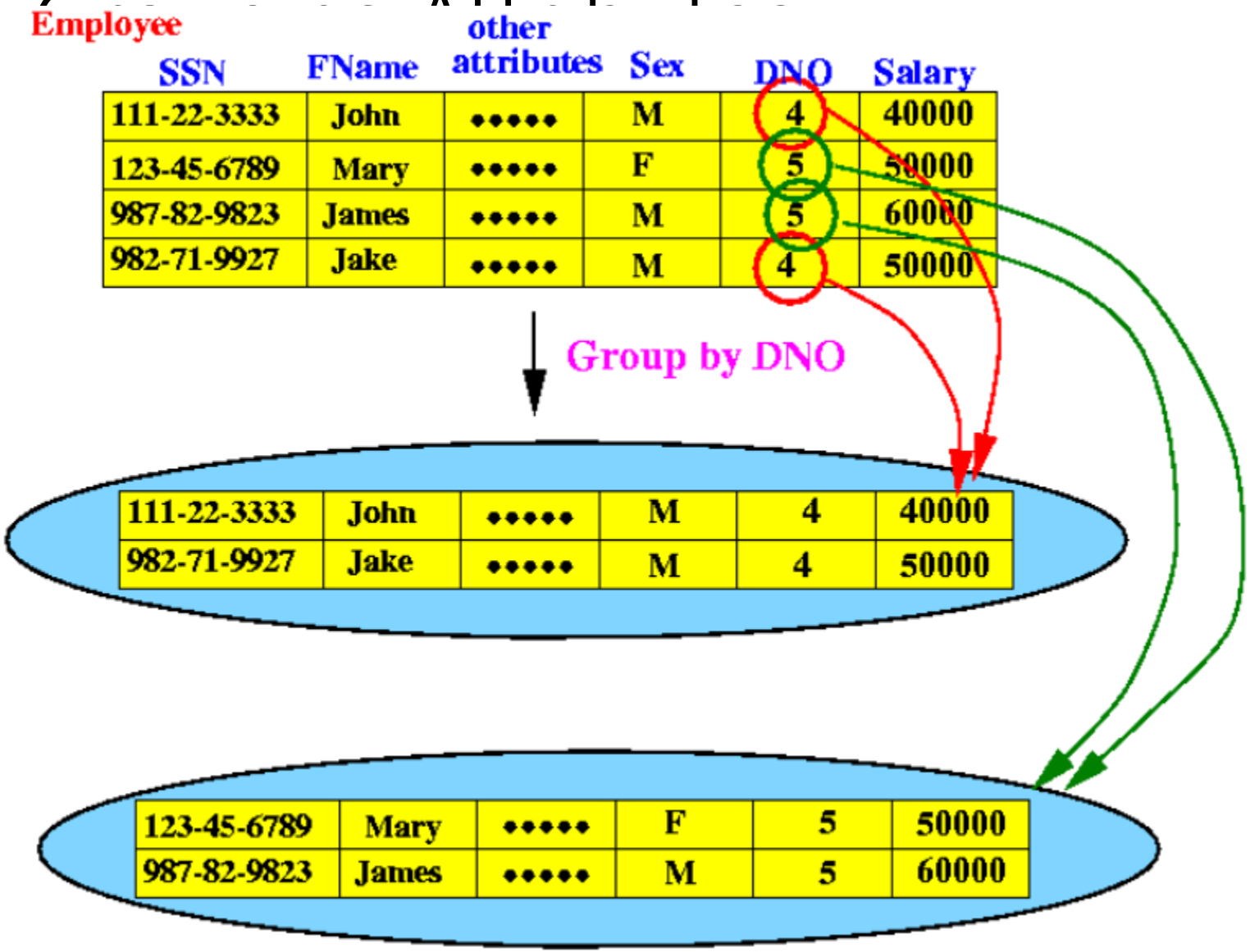
```
SELECT fname, lname
FROM   employee, department
WHERE  dno = dnumber          /* The employee must work for */
  AND  dname = 'Research'     /* the Research department !! */
  AND  salary > (SELECT  AVG(salary)
                 FROM   employee, department
                 WHERE  dno = dnumber
                   AND  dname = 'Research')
```

Forming Groups with Common Attribute Values: G

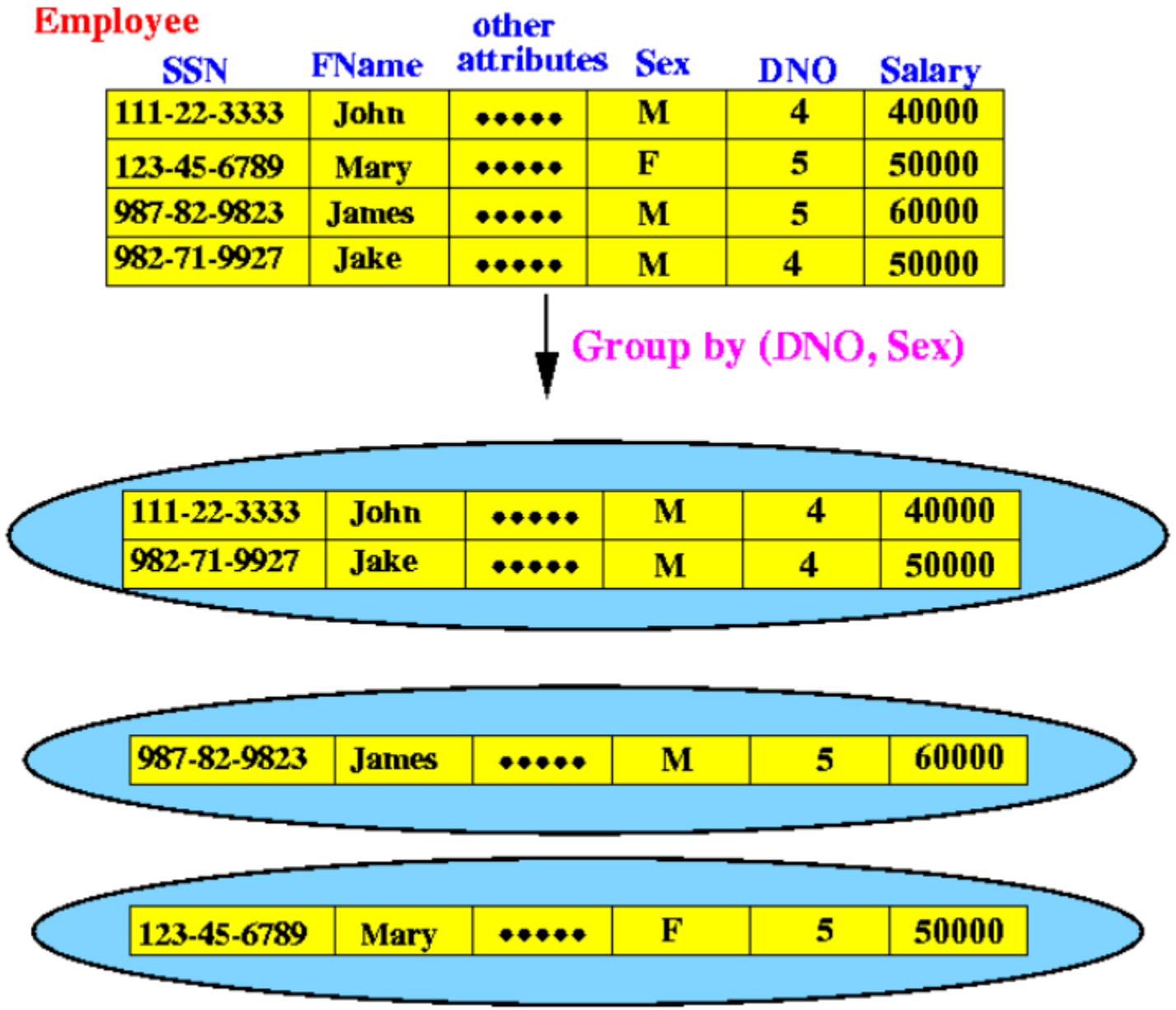
- **Grouping**
- Forming
  - Each gr
    some a
  - Examp

narization

e value of



Another Example:



Grouping attribute: dno and sex

Forming Groups with Common Attribute Values: Grouping Attributes (cont.)

- **GRO**
  - S
- Exa
  - F

department

Here is the employee relation:

```
SELECT fname, lname, sex, dno, salary
FROM   employee
```

fname	lname	sex	dno	salary
James	Borg	M	1	55000.00
Alicia	Zelaya	F	4	25000.00
Jennif	Wallace	F	4	43000.00
Ahmad	Jabbar	M	4	25000.00
John	Smith	M	5	30000.00
Frankl	Wong	M	5	40000.00
Ramesh	Narayan	M	5	38000.00
Joyce	English	F	5	25000.00

Here is the solution:

```
SELECT    dno, sum(salary)
FROM      employee
GROUP BY  dno
```

DNO	SUM(SALARY)
1	55000
4	93000
5	133000

Grouping with Multiple Common Attribute Values

- Query:
  - Find the total salary paid to **male** and **female** employees (separate total) for **each department**
- Solution:

```
SELECT    dno, sex, sum(salary)
FROM      employee
GROUP BY  dno, sex
```

DNO	S	SUM(SALARY)
1	M	55000
4	F	68000
4	M	25000
5	F	25000
5	M	108000

Note on Output Format

- See an example first:

```
SUM(SALARY)
-------------------
          25000
          55000
         108000
          68000
          25000
```

```
SELECT      sum(salary)    /* Omit dno, sex */
FROM        employee
GROUP BY    dno, sex
```

What does each row mean?
Group by What ?

You don't have a clue what each row mean...

- You don't **need** to specify the **grouping attributes** in the **SELECT clause**

- **However**, if you **omit** , you will have **no idea what** the **result** ***mean*** !!!

Execution of an SQL Query with GROUP BY Clause

- A query with a GROUP BY clause is processed as follows:
  - First, select the **tuples** that satisfies the tuple (WHERE) condition
  - Then, the **selected tuples** in step 1 are grouped based on their value in the grouping attributes
  - Finally, one or more aggregate functions is applied to the groups

- Example:

How is the following query processed?

```
SELECT    dno, sum(salary)
FROM      employee
WHERE     sex = 'M'
GROUP BY  dno
```

The complete Employee relation:

SSN	DNO	S	SALARY
123456789	5	M	30000
333445555	5	M	40000
999887777	4	F	25000
987654321	4	M	43000
666884444	5	M	38000
453453453	5	F	25000
987987987	4	M	25000
888665555	1	M	55000

(1) The WHERE clause is processed first:

SSN	DNO	S	SALARY
123456789	5	M	30000
333445555	5	M	40000
987654321	4	M	43000
666884444	5	M	38000
987987987	4	M	25000
888665555	1	M	

55000aaaa

(2) Resulting tuples are grouped by the **grouping attributes (dno):**

SSN	DNO	S	SALARY
888665555	1	M	55000
987654321	4	M	43000
987987987	4	M	25000
123456789	5	M	30000
333445555	5	M	40000
666884444	5	M	38000

(3) Apply the aggregate function(s) on each group

-----> (DNO=1) SUM = 55000

-----> (DNO=4) SUM = 68000

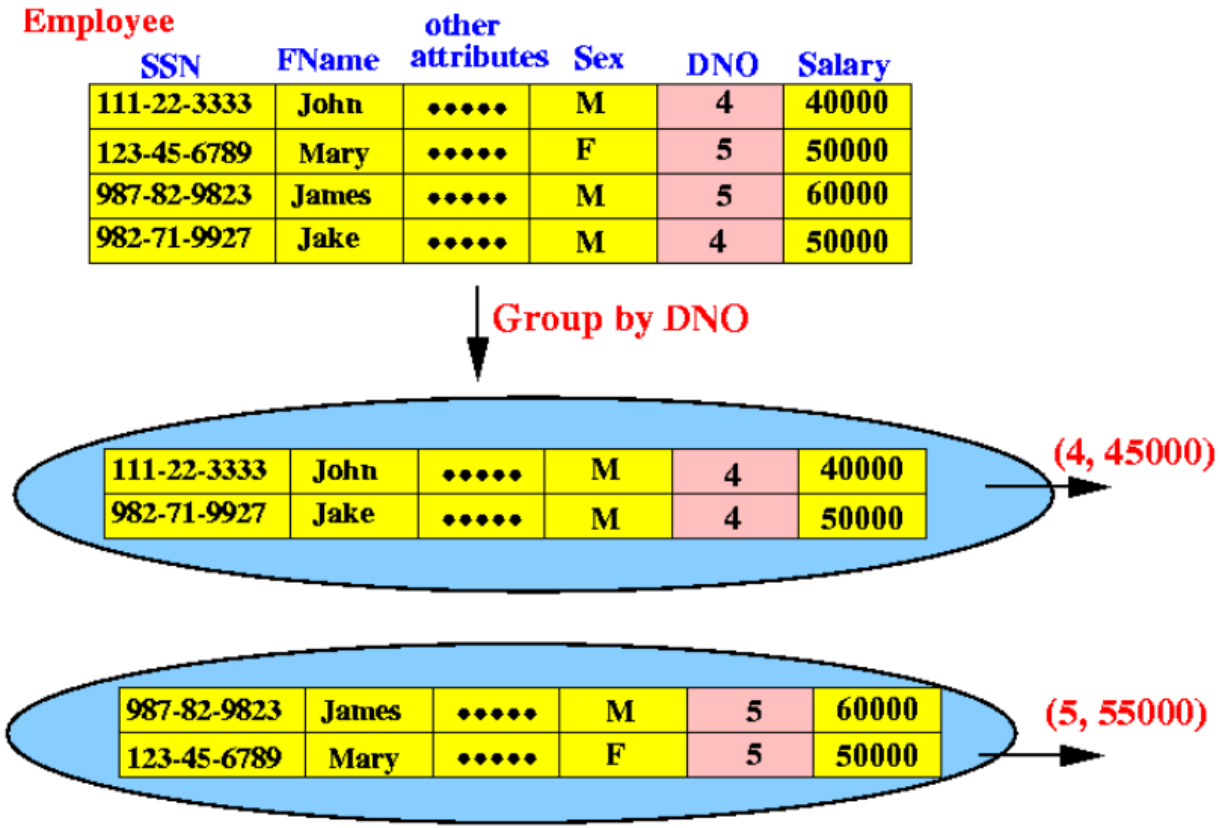
-----> (DNO=5) SUM = 108000

Output:

DNO	SUM(SALARY)
1	55000
4	68000
5	108000

Effect of the GROUP BY on Common Attribute Values

- Property:
  - After forming groups based on the **grouping attributes**, the attribute values other than the grouping attributes will be **lost**



Effect of the GROUP BY on Common Attribute Values (cont.)

- Why?

- The **group by** function will **produce *one* (1) tuple** output for ***each* group**

  Therefore, you can ***only*** have **a *single* value** as **output**

- ***Each* tuple** in a **group** has the ***same* value** for **DNO** (**grouping attribute**)

  Therefore, you ***can*** produce **a *single* value** for **DNO** as an **output value**

- ***Different* tuples** in the ***same* group** have ***different* value** for the ***other* attributes** (e.g., **SSN in group (DNO=4)**: 111-22-3333 and 982-71-9927)

  Therefore, you ***cannot*** produce **a *single* value** for ***non-grouping* attributes** as an **output value** !!!

Examples of GROUP BY Queries

- Query 1:
  - For ... of employees and ...
- Soluti...

```
SELECT
FROM
WHERE
```

...lary)

CORRECT:

```
SELECT   dname, COUNT(ssn), AVG(salary)
FROM     employee, department
WHERE    dno = dnumber
GROUP BY dno, dname
```

DNAME	COUNT(SSN)	AVG(SALARY)
Administration	3	31000
Headquarters	1	55000
Research	4	33250

```
+---------+
| ssn     |
+---------+
| 12345678|
| 33344555|
| 99988777|
| 98765432|
| 66688444|
| 45345345|
| 987987987 |   4 | Administration | 25000.00 |
| 888665555 |   1 | Headquarters   | 55000.00 |
+-----------+-----+----------------+----------+
```

Wrong! dname must be a grouping attribute !!

Flashback: Data Independency

- Physical data independence

> ▪ ***Physical* data independence** = the **ability** to **change** the ***Physical* storage format** of the **data files *without*** having to **change** the **program** (so you don't need to ***re-compile*** the code !!!)

  - You can change the **structure of the physical data records** without having to **change the programs** to access the new record structure
  - All you need to do is: provide a **new (updated) data description** (= meta data)

- Logical data independence

> ▪ ***Logical* data independence** = the **ability** to ***present*** the **stored data** in ***any* format** to the **user**

  - Data is stored in one format (called the **conceptual database schema**), the database system can present the data in different ways to different users
  - The view of the data can be adapted to the need of the user

Concept of a View in SQL

- A **view** is a **virtual relation** that is derived from
  - Relations
  - Other **virtual relations**
- Property of a virtual relation:
  - A virtual relation (= view) does not exist as in physical form
  - The tuples in a virtual relation (= view) are not stored physically in the database
  - The tuples in a virtual relation (= view) are computed as a **temporary relation** when the virtual relation is used

> - the **virtual relation** is **constructed** as a ***temporary*** **relation** at ***that*** **moment** in time

Example Temporary Rel

- Suppose a secretary maintains a lis
  - The list contains the following inform

FName	LName	ProjectName	#HoursWork

  - You could obtain the data using this S

Question: How to find the activities of "John Smith"?

```
SELECT fname, lname, pname, hours
FROM   employee, works_on, project
WHERE  ssn = essn
  AND  pno = pnumber
```

Output:

FNAME	LNAME	PNAME	HOURS
John	Smith	ProductX	32.5
John	Smith	ProductY	7.5
Frank	Wong	ProductY	10
Frank	Wong	Reorganization	10
Frank	Wong	ProductZ	10
Frank	Wong	Computerization	10
Joyce	English	ProductX	20
Joyce	English	ProductY	20
John	Doe	ProductZ	40
James	Borg	Reorganization	0
Jack	Wallace	Reorganization	15
Jack	Wallace	Newbenefits	20
Jake	Jones	Computerization	35
Jake	Jones	Newbenefits	5
Alice	Miller	Computerization	10
Alice	Miller	Newbenefits	30

Example Temporary Relation (cont.)

- To find the activities of "John Smith", we can use the following temporary relation construct:

```
SELECT *
FROM ( SELECT fname, lname, pname, hours
       FROM   employee, works_on, project
       WHERE  ssn = essn
         AND  pno = pnumber
     )   EmpActivity
WHERE  fname = 'John'
  AND  lname = 'Smith'


Output:

     FNAME  LNAME    PNAME                HOURS
     ------ -------- -------------------- ---------
     John   Smith    ProductX              32.5
     John   Smith    ProductY               7.5
```

The **temporal relation** EmpActivity is created on the fly !!!!

Creating View (Virtual Relations)

- A **view** (= **virtual relation**) is defined using the CREATE VIEW command
  - A (virtual) table name (or **view name**)
  - **A list of attribute names**, and
  - A **query** to specify the contents of the view.
- Example:

```
CREATE VIEW EmpActivity
AS
   ( SELECT Fname, Lname, Pname, Hours
     FROM   employee, works_on, project
     WHERE  ssn = essn
       AND  pno = pnumber
   )
```

EmpActivity

Fname	Lname	Pname	Hours

Results:

- The **CREATE VIEW** command defines an virtual table (relation) called EmpActivity
- This virtual table/relation will be construct when we use the EmpActivity relation in a query !!!

Creating View (Virtual Relations) (Example)

- We can now specify SQL queries on a **view**—or virtual table—in the same way we specify queries involving **base tables**
  - Example: Find all activities of John Smith

```
SELECT *
FROM   EmpActivity
WHERE  fname = 'John'
  AND  lname = 'Smith'

FNAME      LNAME      PNAME           HOURS
---------- ---------- --------------- ---------
John       Smith      ProductX           32.5
John       Smith      ProductY            7.5
```

Why Do We Introduce t

- Now thinking: why do we introduce the concept of **view**?
  - Simplify the specification of certain queries
  - Used as a security and authorization mechanism

```
SELECT fname, lname, pname, hours
FROM   employee, works_on, project
WHERE  ssn = essn
  AND  pno = pnumber
```

Output:

FNAME	LNAME	PNAME	HOURS
John	Smith	ProductX	32.5
John	Smith	ProductY	7.5
Frank	Wong	ProductY	10
Frank	Wong	Reorganization	10
Frank	Wong	ProductZ	10
Frank	Wong	Computerization	10
Joyce	English	ProductX	20
Joyce	English	ProductY	20
John	Doe	ProductZ	40
James	Borg	Reorganization	0
Jack	Wallace	Reorganization	15
Jack	Wallace	Newbenefits	20
Jake	Jones	Computerization	35
Jake	Jones	Newbenefits	5
Alice	Miller	Computerization	10
Alice	Miller	Newbenefits	30

Views as Authorization Mechanism

- **Views** can be used to hide certain attributes or tuples from unauthorized users
  - Example: suppose a certain user is only allowed to see **employee** information for employees who **work for department 5**;

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

Grant the user the privilege to query the **view** but not the base table EMPLOYEE itself.

  - Another example: restrict a user to only see certain co name, last name, and address of an employee

```
CREATE VIEW BASIC_EMP_DATA
AS
     ( SELECT  Fname, Lname, Address
       FROM   EMPLOYEE;
     )
```

Keep View be Up-to-Date

- A view is supposed to be always **up-to-date**
- If we modify the tuples in the base tables on which the **view** is defined
  - The view must automatically reflect these changes
  - The view does not have to be realized or materialized **at the time of view definition** but rather **at the time when we specify a query** on the **view**
  - Responsibility of the DBMS and not the user

```
CREATE VIEW EmpActivity
AS
   ( SELECT fname, lname, pname, hours
     FROM   employee, works_on, project
     WHERE  ssn = essn
       AND  pno = pnumber
   )
```

View is Computed on the fly

- Consider the following **view definition**

```
CREATE VIEW Dept_Info(dname, no_emps, total_sal)
AS
   ( SELECT      dname, count(ssn), sum(salary)
     FROM        department, employee
     WHERE       dnumber = dno
     GROUP BY    dname
   )
```

- Example query on this view:
  - Find all department with >2 employees:

```
SELECT *
FROM   Dept_Info
WHERE  no_emps > 2
```

DNAME	NO_EMPS	TOTAL_SAL
Administration	3	93000
Research	4	133000

View is Computed on the fly (cont.)

- Change the salary of employees:

```
UPDATE employee
SET    salary = salary + 10000

SELECT *
FROM   Dept_Info
WHERE  no_emps > 2


DNAME              NO_EMPS  TOTAL_SAL
-------------- ---------- ----------
Research                4     173000
Administration          3     123000
```

Modifying Existing Relations: Altering and Dropping

- Modifying the *structure* of an existing relation
  - After you have defined a relation using the CREATE TABLE command, you can still make changes to its **structure**

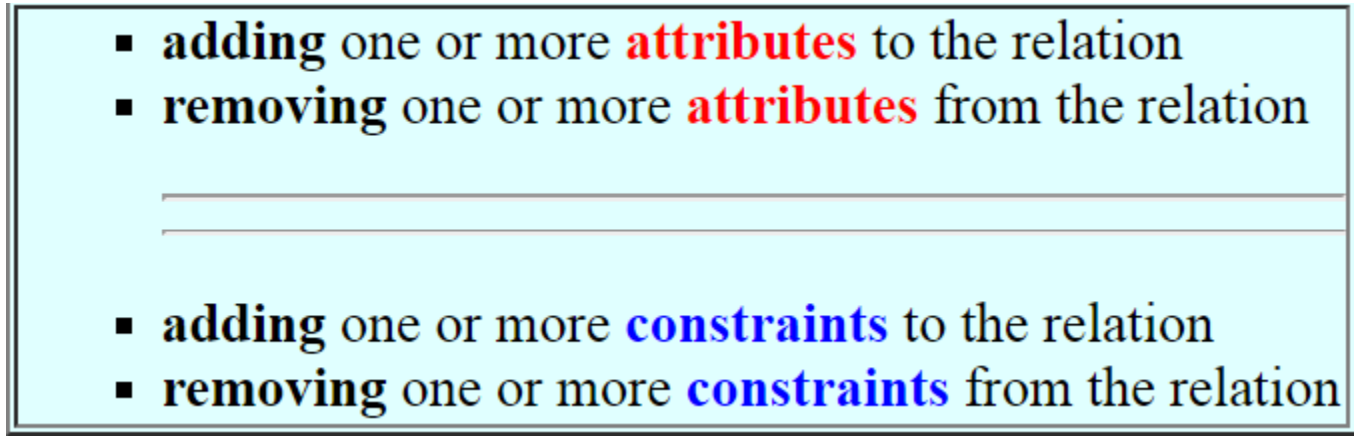
- **DROP TABLE**
  - Deleting **the definition of a table** from the **database schema**
  - Syntax: DROP TABLE relation_name
  - The **relation definition** is removed, along with **any tuples (data)** that you have previously inserted....
  - Here is the difference between **Drop** and **Delete**

Modifying Existing Relations: Altering and Dropping (cont.)

- Two types:
  - Example: if we no longer wish to keep track of dependents of employees in the COMPANY database, drop it:
  - **1) DROP TABLE DEPENDENT CASCADE;**
    - All constraints, views, and other elements that **reference the table** being dropped are also dropped automatically from the schema, along with the table itself.
  - **2) DROP TABLE** DEPENDENT **RESTRICT**;
    - A table is dropped only if it is not referenced in any constraints (for example, by foreign key definitions in another elation) or views (see Section 7.3) or by any other elements

Modifying Existing Relations: Altering and Dropping (cont.)

- **ALTER TABLE**: changing a relation schema.
  - SQL allows the owner of the database relation to change it by:



  - **ALTER TABLE** COMPANY.EMPLOYEE **ADD COLUMN** Job VARCHAR(12);
  - **ALTER TABLE** COMPANY.EMPLOYEE **DROP** COLUMN Address CASCADE;

Create Assertion

- **CREATE ASSERTION**
  - Used to specify **additional types of constraints** that are outside the scope of the **built-in** relational model **constraints**
  - Example: specify the constraint: *the salary of an employee must not be greater than the salary of the manager of the department that the employee works for*:

```
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 ) );
```

Create Trigger

- **CREATE TRIGGER**
  - Used to specify **automatic actions** that the database system will perform when certain **events and conditions** occur.
- Example:
  - *Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor*
  - **Trigger conditions**: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor
  - **Action**: suppose that the action to take would be to call an external stored procedure SALARY_VIOLATION, which will notify the supervisor.

Create Trigger (cont.)

- The trigger could then be written as

```
CREATE TRIGGER                    SALARY_VIOLATION
BEFORE INSERT OR UPDATE OF        SALARY, SUPERVISOR_SSN ON EMPLOYEE
FOR EACH ROW
        WHEN (  NEW.SALARY > ( SELECT SALARY FROM EMPLOYEE
                               WHERE SSN = NEW.SUPERVISOR_SSN
                               )
             )
         INFORM_SUPERVISOR(NEW.Supervisor_ssn, NEW.Ssn );
```

ECA (Event, Condition, Action)