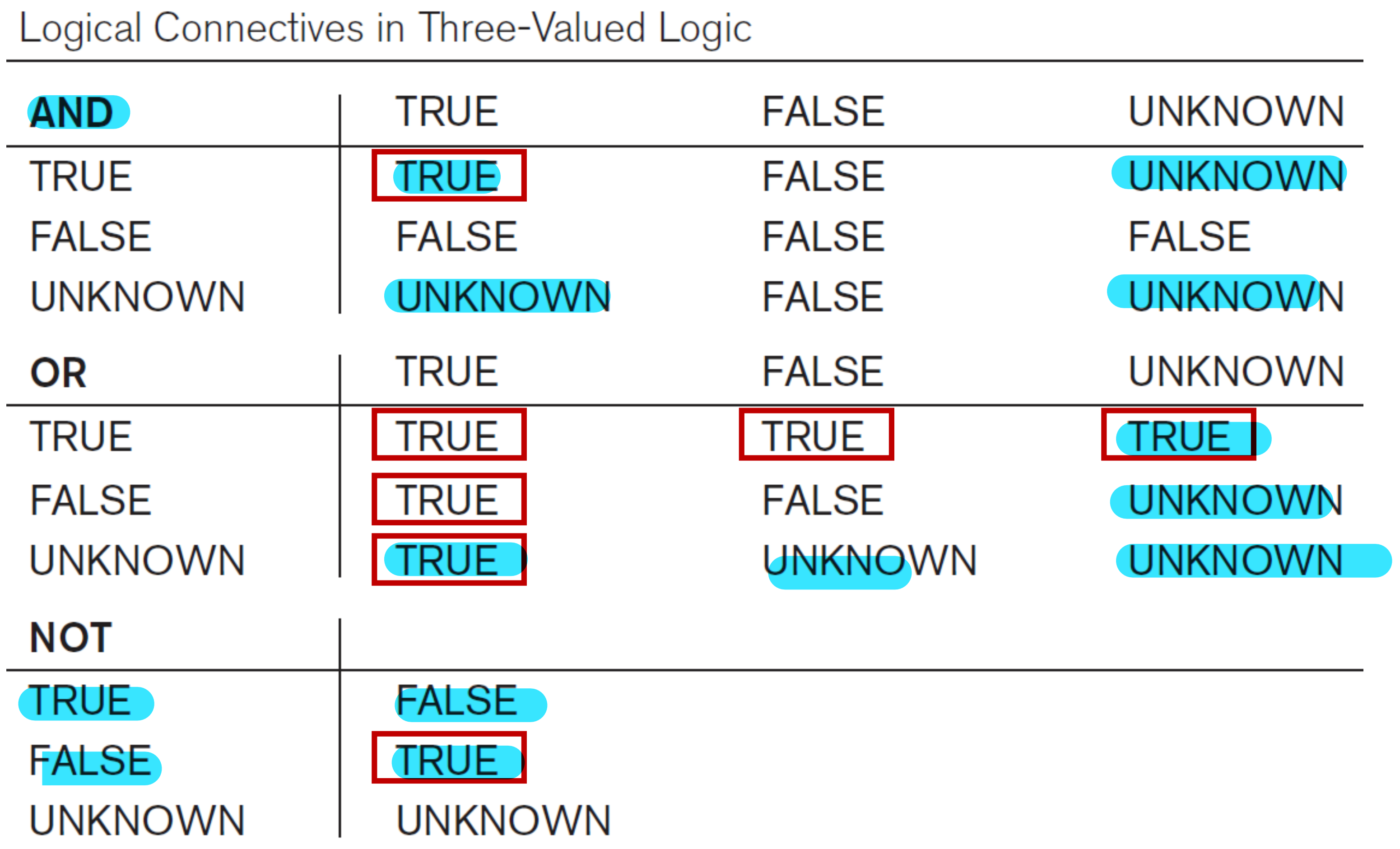
Lecture 5: Complex SQL (Structured Query Language)

CS3402 Database Systems

Comparisons Involving NULL and Three-Valued Logic (1/3)

- When a record with NULL in one of its attributes is involved in a comparison operation, the result is considered to be UNKNOWN (it may be TRUE or it may be FALSE).
- SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead of the standard two-valued (Boolean) logic with values TRUE or FALSE.
- It is therefore necessary to define the results (or truth values) of three-valued logical expressions when the logical connectives AND, OR, and NOT are used.

Comparisons Involving NULL and Three-Valued Logic (2/3)



Logical Connectives in 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 (3/3)

- SQL allows queries that check whether an attribute value is NULL. Rather than using = or <> to compare an attribute value to NULL, SQL uses the comparison operators IS or IS NOT.
- When a join condition is specified, tuples with NULL values for the join attributes are not included in the result (unless it is an OUTER JOIN).
- For example, retrieve the names of all employees who do not have supervisors:

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

Nested Queries (1/6)

- Some queries require that existing values in the database be fetched and then used in a comparison condition.
- Such queries can be conveniently formulated by using **nested queries**, which are complete select-from-where blocks within another SQL query.
- That other query is called the **outer query**.
- These nested queries can also appear in the WHERE clause or the FROM clause or the SELECT clause or other SQL clauses as needed.

Nested Queries (2/6)

- Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.
- The first nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as manager.
- The second nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as worker.
- In the outer query, we use the OR logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested 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');
```

Nested Queries (3/6)

- If a nested query returns a single attribute and a single tuple, the query result will be a single (**scalar**) value. In such cases, it is permissible to use = instead of IN for the comparison operator.
- In general, the nested query will return a **table** (relation), which is a set or multiset of tuples.

Nested Queries (4/6)

- SQL allows the use of **tuples** of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:

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

- This query will select the Essns of all employees who work the same (project, hours) combination on some project that employee ‘John Smith’ (whose Ssn = ‘123456789’) works on. In this example, the IN operator compares the subtuple of values in parentheses (Pno, Hours) within each tuple in WORKS_ON with the set of type-compatible tuples produced by the nested query.

Nested Queries (5/6)

- In addition to the IN operator, a number of other comparison operators can be used to compare a single value v (typically an attribute name) to a set or multiset v (typically a nested query). The = ANY (or = SOME) operator returns TRUE if the value v is equal to some value in the set V and is hence equivalent to IN. The two keywords ANY and SOME have the same effect.
- Other operators that can be combined with ANY (or SOME) include >, >=, <, <=, and <>.

Nested Queries (6/6)

- The keyword ALL can also be combined with each of these operators. For example, the comparison condition (v > ALL V) returns TRUE if the value v is greater than all the values in the set (or multiset) V.
- For example, return the names of employees whose salary is greater than the salary of all the employees in department 5:

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

Correlated Nested Queries (1/2)

- Whenever a condition in the WHERE clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be **correlated**.
- We can understand a correlated query better by considering that the nested query is evaluated once for each tuple (or combination of tuples) in the outer query.

Correlated Nested Queries (2/2)

- For example, for each EMPLOYEE tuple, evaluate the nested query, which retrieves the Essn values for all DEPENDENT tuples with the same sex and name as that EMPLOYEE tuple; if the Ssn value of the EMPLOYEE tuple is in the result of the nested query, then select that EMPLOYEE tuple.

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

EXISTS and NOT EXISTS Functions (1/5)

- EXISTS and NOT EXISTS are Boolean functions that return TRUE or FALSE; hence, they can be used in a WHERE clause condition.
- The EXISTS function in SQL is used to check whether the result of a nested query is empty (contains no tuples) or not. The result of EXISTS is a Boolean value TRUE if the nested query result contains at least one tuple, or FALSE if the nested query result contains no tuples.
- For example, retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

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

EXISTS and NOT EXISTS Functions (2/5)

- EXISTS (and NOT EXISTS) are typically used in conjunction with a correlated nested query. In the example, the nested query references the Ssn, Fname, and Sex attributes of the EMPLOYEE relation from the outer query.
- For each EMPLOYEE tuple, evaluate the nested query, which retrieves all DEPENDENT tuples with the same Essn, Sex, and Dependent_name as the EMPLOYEE tuple; if at least one tuple EXISTS in the result of the nested query, then select that EMPLOYEE tuple. EXISTS(Q) returns TRUE if there is at least one tuple in the result of the nested query Q, and returns FALSE otherwise.

EXISTS and NOT EXISTS Functions (3/5)

- On the other hand, NOT EXISTS(Q) returns TRUE if there are no tuples in the result of nested query Q, and returns FALSE otherwise.
- For example, retrieve the names of employees who have no dependents:

```
SELECT Fname, Lname
FROM EMPLOYEE
WHERE NOT EXISTS (SELECT *
                  FROM DEPENDENT
                  WHERE Ssn=Essn);
```

- The correlated nested query retrieves all DEPENDENT tuples related to a particular EMPLOYEE tuple. If none exist, the EMPLOYEE tuple is selected because the WHERE-clause condition will evaluate to TRUE in this case.
- For each EMPLOYEE tuple, the correlated nested query selects all DEPENDENT tuples whose Essn value matches the EMPLOYEE Ssn; if the result is empty, no dependents are related to the employee, so we select that EMPLOYEE tuple and retrieve its Fname and Lname.

EXISTS and NOT EXISTS Functions (4/5)

- List the names of managers who have at least one dependent.

```
SELECT Fname, Lname
FROM EMPLOYEE
WHERE EXISTS (SELECT *
              FROM DEPENDENT
              WHERE Ssn=Essn)
      AND
      EXISTS (SELECT *
              FROM DEPARTMENT
              WHERE Ssn=Mgr_ssn);
```

- We specify two nested correlated queries; the first selects all DEPENDENT tuples related to an EMPLOYEE, and the second selects all DEPARTMENT tuples managed by the EMPLOYEE. If at least one of the first and at least one of the second exists, we select the EMPLOYEE tuple.

EXISTS and NOT EXISTS Functions (5/5)

- Retrieve the name of each employee who works on all the projects controlled by department number 5:

```
SELECT Fname, Lname
FROM EMPLOYEE
WHERE NOT EXISTS ((SELECT Pnumber
                   FROM PROJECT
                   WHERE Dnum = 5)
                   EXCEPT (SELECT Pno
                          FROM WORKS_ON
                          WHERE Ssn = Essn));
```

- The first subquery (which is not correlated with the outer query) selects all projects controlled by department 5, and the second subquery (which is correlated) selects all projects that the particular employee being considered works on.
- If the set difference of the first subquery result MINUS (EXCEPT) the second subquery result is empty, it means that the employee works on all the projects and is therefore selected.

Aggregate functions (1/6)

- **Aggregate functions** are used to summarize information from multiple tuples into a single-tuple summary.
- Built-in aggregate functions: COUNT, SUM, MAX, MIN, and AVG.
  - The COUNT function returns the number of tuples or values as specified in a query.
  - The functions SUM, MAX, MIN, and AVG can be applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values.
  - These functions can be used in the SELECT clause or in a HAVING clause.

Aggregate functions (2/6)

- Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

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

- This query returns a single-row summary of all the rows in the EMPLOYEE table. We could use AS to rename the column names in the resulting single-row table; for

```
SELECT SUM(Salary) AS Total_Sal, MAX(Salary) AS Highest_Sal,
MIN(Salary) AS Lowest_Sal, AVG(Salary) AS Average_Sal
FROM EMPLOYEE;
```

Aggregate functions (3/6)

- Find the sum of the salaries of all employees of the ‘Research’ department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
SELECT SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
FROM EMPLOYEE, DEPARTMENT
WHERE Dno=Dnumber  AND Dname=‘Research’;
```

- Retrieve the total number of employees in the company:

```
SELECT COUNT (*)
FROM EMPLOYEE;
```

- Retrieve the number of employees in the ‘Research’ department .

```
SELECT COUNT (*)
FROM EMPLOYEE, DEPARTMENT
WHERE DNO = DNUMBER AND DNAME = ‘Research’;
```

- Here the asterisk (*) refers to the rows (tuples), so COUNT (*) returns the number of rows in the result of the query.

Aggregate functions (4/6)

- We may also use the COUNT function to count values in a column rather than tuples.
- For example, count the number of distinct salary values in the database:

```
SELECT COUNT (DISTINCT Salary)
FROM EMPLOYEE;
```

- If we write COUNT(SALARY) instead of COUNT(DISTINCT SALARY), then duplicate values will not be eliminated.
- However, any tuples with NULL for SALARY will not be counted. In general, NULL values are **discarded** when aggregate functions are applied to a particular column (attribute); the only exception is for COUNT(*) because tuples instead of values are counted.

Aggregate functions (5/6)

- The general rule is as follows: when an aggregate function is applied to a collection of values, NULLs are removed from the collection before the calculation; if the collection becomes empty because all values are NULL, the aggregate function will return NULL (except in the case of COUNT, where it will return 0 for an empty collection of values).

Aggregate functions (6/6)

- Aggregate functions can also be used in selection conditions involving nested queries. We can specify a correlated nested query with an aggregate function, and then use the nested query in the WHERE clause of an outer query.
- For example, to retrieve the names of all employees who have two or more dependents:

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE (SELECT COUNT(*)
       FROM DEPENDENT
       WHERE Ssn = Essn) >= 2;
```

- The correlated nested query counts the number of dependents that each employee has; if this is greater than or equal to two, the employee tuple is selected.

GROUP BY Clause (1/4)

- In many cases we want to apply the aggregate functions to subgroups of tuples in a relation, where the subgroups are based on some attribute values.
- For example, we may want to find the average salary of employees in each department or the number of employees who work on each project.
- In these cases we need to **partition** the relation into nonoverlapping subsets (or **groups**) of tuples. Each group (partition) will consist of the tuples that have the same value of some attribute(s), called the **grouping attribute(s)**. We can then apply the function to each such group independently to produce summary information about each group.

GROUP BY Clause (2/4)

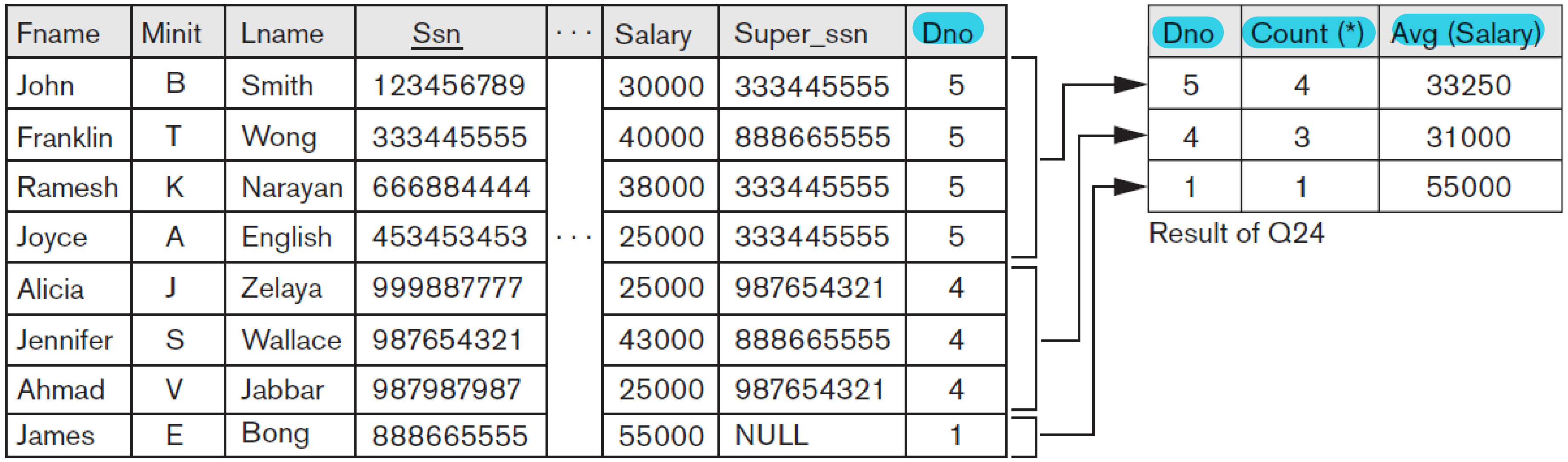
- SQL has a GROUP BY clause for this purpose. The GROUP BY clause specifies the grouping attributes, which should also appear in the SELECT clause, so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attribute(s).
- For example, for each department, retrieve the department number, the number of employees in the department, and their average salary:

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

- In this query, the EMPLOYEE tuples are partitioned into groups—each group having the same value for the GROUP BY attribute Dno. Hence, each group contains the employees who work in the same department. The COUNT and AVG functions are applied to each such group of tuples.

GROUP BY Clause (3/4)

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



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

GROUP BY Clause (4/4)

- If NULLs exist in the grouping attribute, then a **separate group** is created for all tuples with a NULL value in the grouping attribute.
- For another example, for each project, retrieve the project number, the project name, and the number of employees who work on that project.

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

- In this query, we use a join condition in conjunction with GROUP BY. The grouping and functions are applied after the joining of the two relations in the WHERE clause.

Having Clause (1/5)

- Sometimes we want to retrieve the values of these functions only for groups that satisfy certain conditions.
- SQL provides a HAVING clause, which can appear in conjunction with a GROUP BY clause, for this purpose.
- HAVING provides a condition on the summary information regarding the group of tuples associated with each value of the grouping attributes. Only the groups that satisfy the condition are retrieved in the result of the query.

Having Clause (2/5)

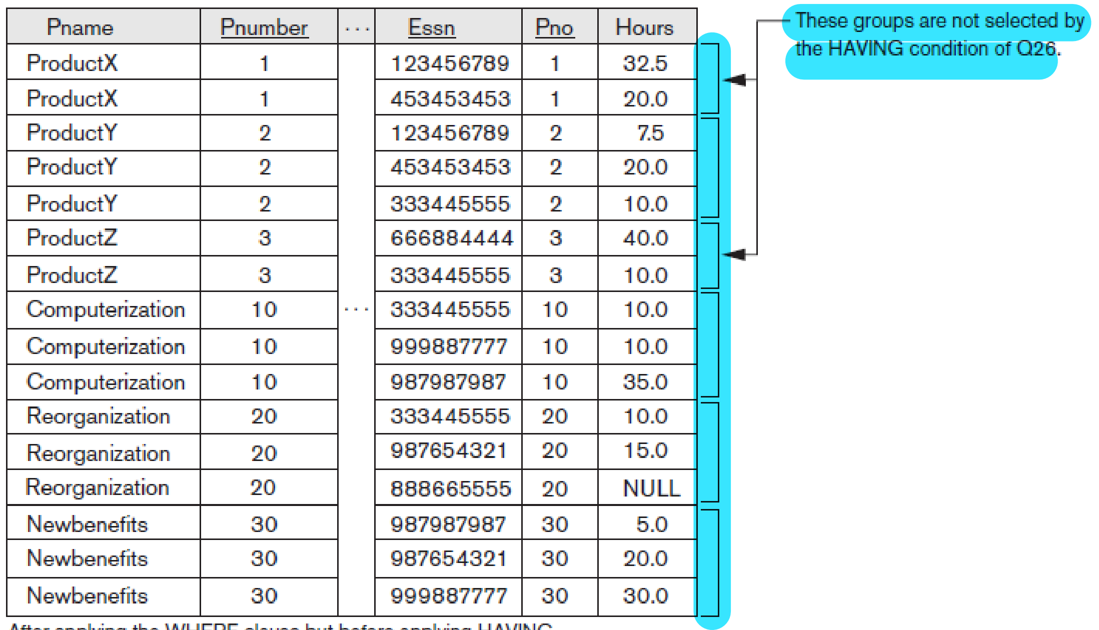
- For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on the project.

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

- Notice that although selection conditions in the WHERE clause limit the tuples to which functions are applied, the HAVING clause serves to choose whole groups.
- Notice that we must be extra careful when two different conditions apply (one to the aggregate function in the SELECT clause and another to the function in the HAVING clause).

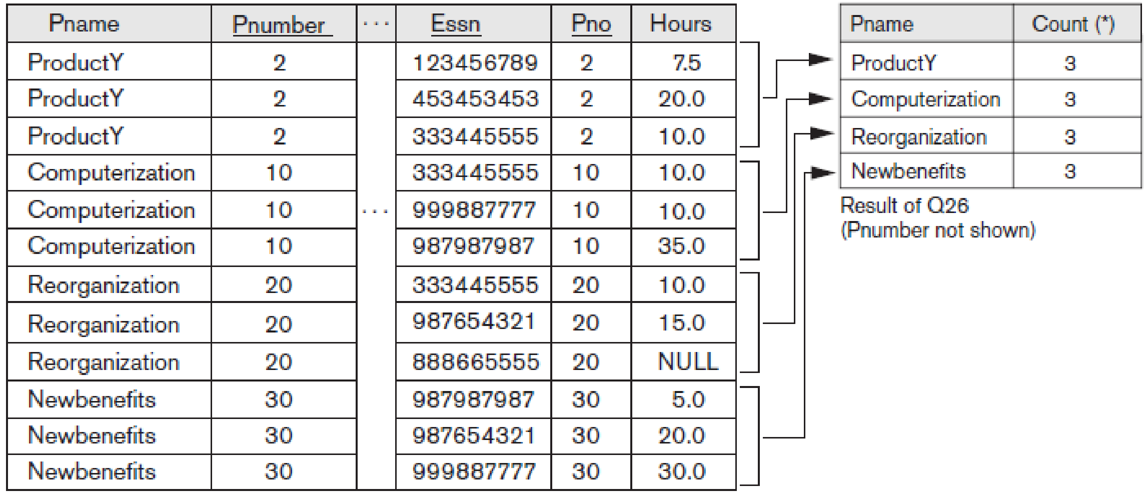
Having Clause (3/5)

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


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

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

After applying the WHERE clause but before applying HAVING



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)

Having Clause (4/5)

- For example, suppose that we want to count the total number of employees whose salaries exceed $40,000 in each department, but only for departments where more than five employees work. Here, the condition (SALARY > 40000) applies only to the COUNT function in the SELECT clause. Suppose that we write the following **incorrect** query:

```
SELECT Dno, COUNT(*)
FROM EMPLOYEE
WHERE Salary>40000
GROUP BY Dno
HAVING COUNT(*)>5;
```

- This is **incorrect** because it will select only departments that have more than five employees who each earn more than $40,000.

Having Clause (5/5)

- The rule is that the WHERE clause is executed first, to select individual tuples or joined tuples; the HAVING clause is applied later, to select individual groups of tuples. In the incorrect query, the tuples are already restricted to employees who earn more than $40,000 before the function in the HAVING clause is applied.
- For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

```
SELECT Dno, COUNT(*)
FROM EMPLOYEE
WHERE Salary>40000 AND Dno IN
                        (SELECT Dno
                         FROM EMPLOYEE
                         GROUP BY Dno
                         HAVING COUNT(*)>5)
GROUP BY Dno;
```

Views (Virtual Tables) in SQL (1/4)

- A **view** in SQL terminology is a single table that is derived from other tables. These other tables can be base tables or previously defined views. A view does not necessarily exist in physical form; it is considered to be a **virtual table**, in contrast to **base tables**, whose tuples are always physically stored in the database. This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view.
- We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically.

Views (Virtual Tables) in SQL (2/4)

Views (Virtual Tables) in SQL (3/4)

- In SQL, the command to specify a view is CREATE VIEW. The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view. If none of the view attributes results from applying functions or arithmetic operations, we do not have to specify new attribute names for the view, since they would be the same as the names of the attributes of the defining tables in the default case.

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

WORKS_ON1

Fname	Lname	Pname	Hours

- In this view, we did not specify any new attribute names for the view WORKS_ON1 (although we could have); in this case, WORKS_ON1 inherits the names of the view attributes from the defining tables EMPLOYEE, PROJECT, and WORKS_ON.

Views (Virtual Tables) in SQL (4/4)

```
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;
```

DEPT_INFO

Dept_name	No_of_emps	Total_sal

- This view explicitly specifies new attribute names for the view DEPT_INFO, using a one-to-one correspondence between the attributes specified in the CREATE VIEW clause and those specified in the SELECT clause of the query that defines the view.

View Implementation (1/3)

- 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 problem of how a DBMS can efficiently implement a view for efficient querying is complex. Two main approaches have been suggested.
- One strategy, called **query modification**, involves modifying or transforming the view query (submitted by the user) into a query on the underlying base tables. The disadvantage of this approach is that it is inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple view queries are going to be applied to the same view within a short period of time.

View Implementation (2/3)

- The second strategy, called **view materialization**, involves physically creating a temporary or permanent view table when the view is first queried or created and keeping that table on the assumption that other queries on the view will follow. In this case, an efficient strategy for automatically updating the view table when the base tables are updated must be developed in order to keep the view up-to-date.
- **Incremental update** strategy
    - The DBMS determines 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.
    - The view is generally kept as a materialized (physically stored) table as long as it is being queried.
    - If the view is not queried for a certain period of time, the system may then automatically remove the physical table and recompute it from scratch when future queries reference the view.

View Implementation (3/3)

- **Immediate update** strategy
  - The DBMS update a view as soon as the base tables are changed.
- **Lazy update** strategy
  - The DBMS updates the view when needed by a view query.
- **Periodic update** strategy
  - The DBMS updates the view periodically (a view query may get a result that is not up-to-date).