# CHAPTER 7

# More SQL: Complex Queries, Triggers, Views, and Schema Modification

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Chapter 7 Outline

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

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# More Complex SQL Retrieval Queries

- Additional features allow users to specify more complex retrievals from database:
  - Nested queries, joined tables, and outer joins (in the FROM clause), aggregate functions, and grouping

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Comparisons Involving NULL and Three-Valued Logic

- Meanings of `NULL`
  - **Unknown value**
  - **Unavailable or withheld value**
  - **Not applicable attribute**
- Each individual `NULL` value considered to be different from every other `NULL` value
- SQL uses a three-valued logic:
  - `TRUE`, `FALSE`, and `UNKNOWN` (like Maybe)
- **NULL = NULL comparison is avoided**

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Comparisons Involving NULL and Three-Valued Logic (cont'd.)

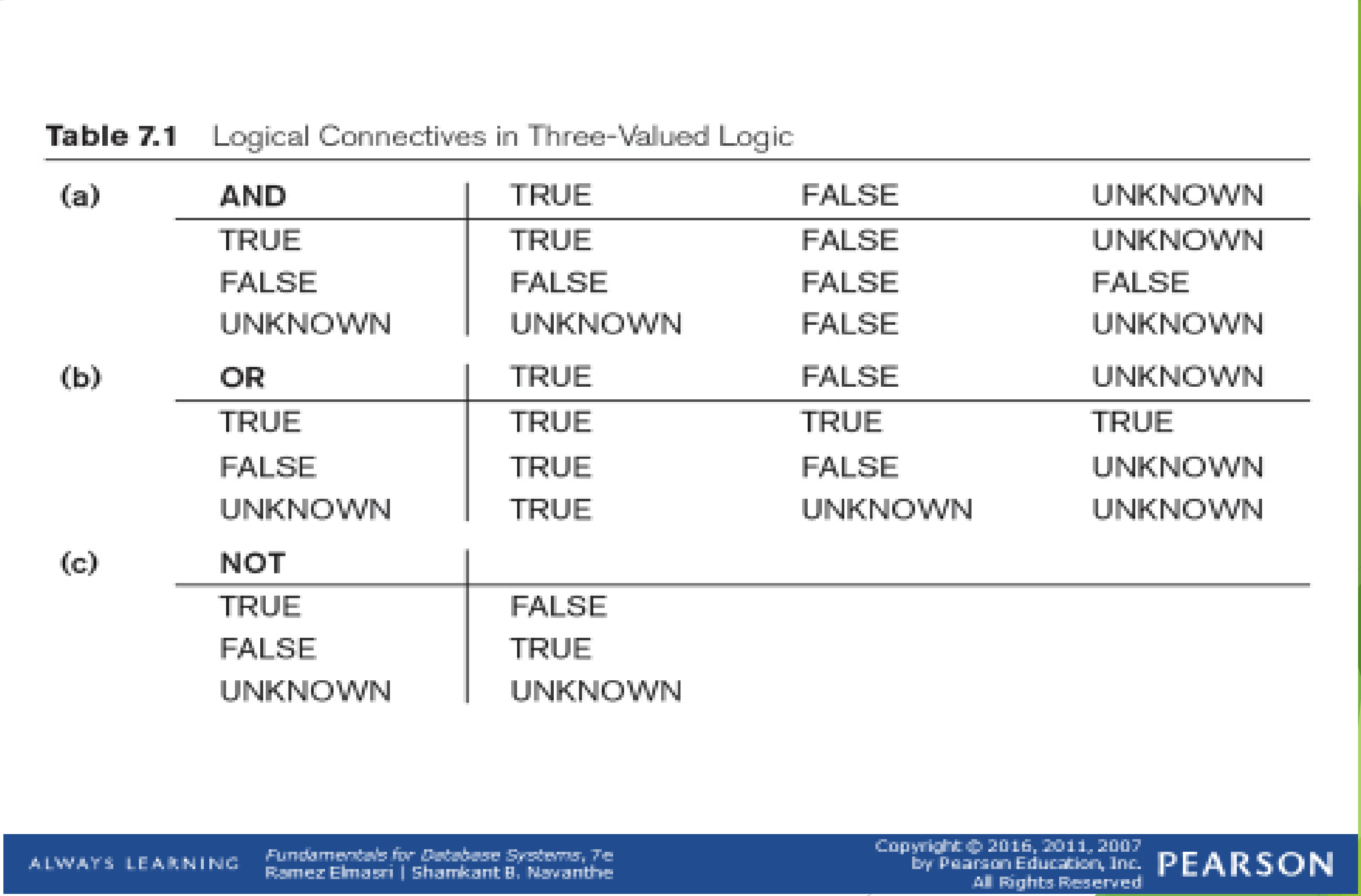
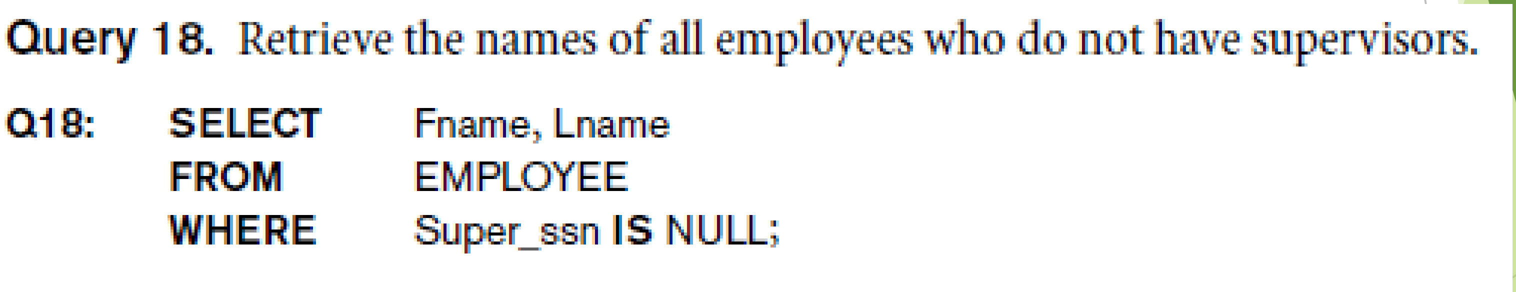


**Table 7.1** Logical Connectives in Three-Valued Logic

|     |         |          |         |         |
|-----|---------|----------|---------|---------|
| (a) | **AND** | TRUE | FALSE | UNKNOWN |
|     | TRUE | TRUE | FALSE | UNKNOWN |
|     | FALSE | FALSE | FALSE | FALSE |
|     | UNKNOWN | UNKNOWN | FALSE | UNKNOWN |
| (b) | **OR** | TRUE | FALSE | UNKNOWN |
|     | TRUE | TRUE | TRUE | TRUE |
|     | FALSE | TRUE | FALSE | UNKNOWN |
|     | UNKNOWN | TRUE | UNKNOWN | UNKNOWN |
| (c) | **NOT** | | | |
|     | TRUE | FALSE | | |
|     | FALSE | TRUE | | |
|     | UNKNOWN | UNKNOWN | | |

# Comparisons Involving NULL and Three-Valued Logic (cont’d.)

- SQL allows queries that check whether an attribute value is `NULL`
  - `IS` or `IS NOT NULL`



**Query 18.** Retrieve the names of all employees who do not have supervisors.

```
Q18:    SELECT    Fname, Lname
        FROM      EMPLOYEE
        WHERE     Super_ssn IS NULL;
```

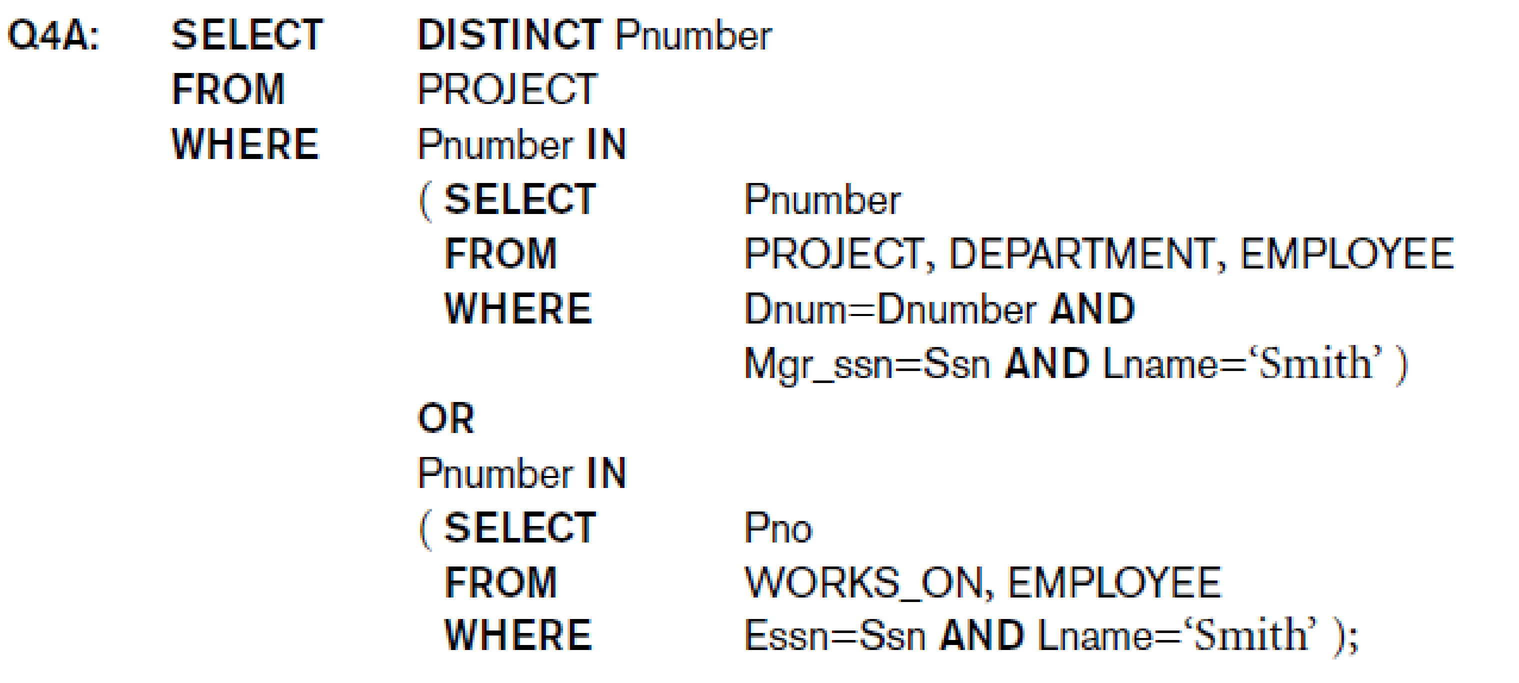
Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Nested Queries, Tuples, and Set/Multiset Comparisons

- **Nested queries**
  - Complete select-from-where blocks within WHERE clause of another query
  - **Outer query and nested subqueries**
- 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*

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

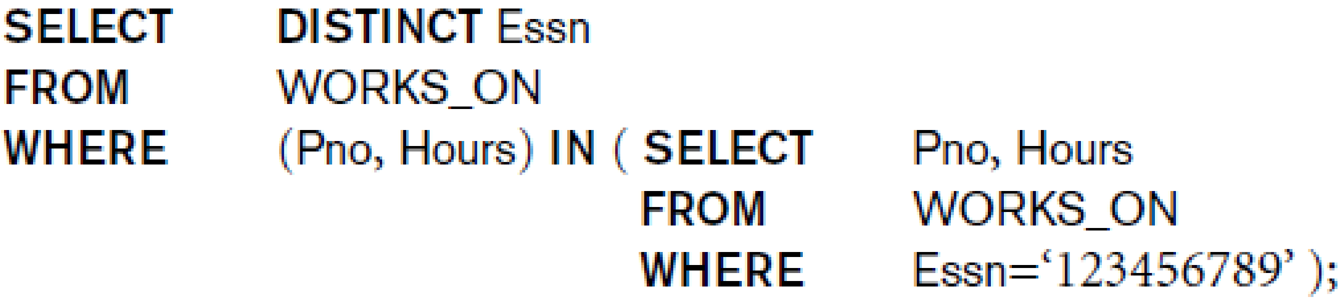
# Nested Queries (cont’d.)



```
Q4A:    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 (cont'd.)

- Use tuples of values in comparisons
  - Place them within parentheses

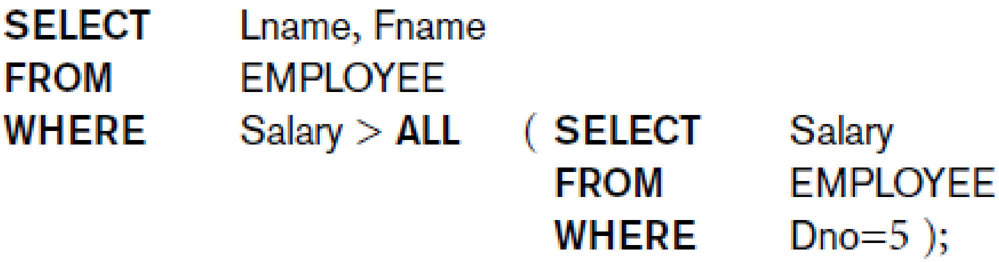


```
SELECT     DISTINCT Essn
FROM       WORKS_ON
WHERE      (Pno, Hours) IN ( SELECT     Pno, Hours
                             FROM       WORKS_ON
                             WHERE      Essn='123456789' );
```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Nested Queries (cont'd.)

- Use other comparison operators to compare a single value *v*
  - = 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
  - Other operators that can be combined with ANY (or SOME): >, >=, <, <=, and <>
  - ALL: value must exceed all values from nested query

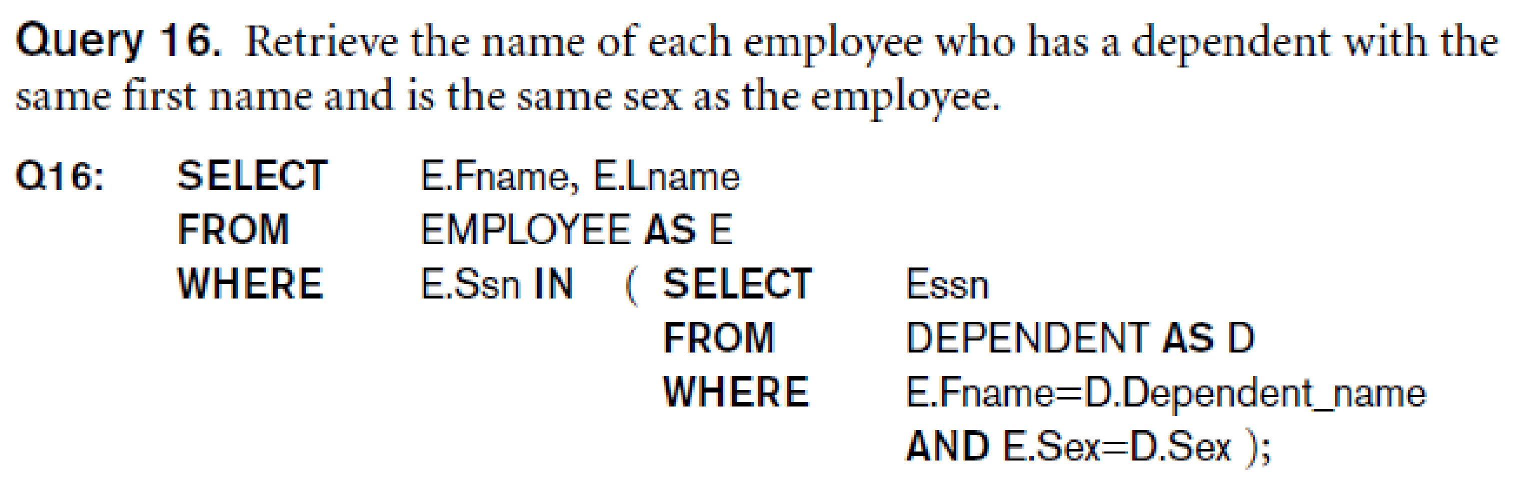


```
SELECT    Lname, Fname
FROM      EMPLOYEE
WHERE     Salary > ALL    ( SELECT    Salary
                            FROM      EMPLOYEE
                            WHERE     Dno=5 );
```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Nested Queries (cont'd.)

- Avoid potential errors and ambiguities
  - Create tuple variables (aliases) for all tables referenced in SQL query



**Query 16.** Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```
Q16:   SELECT   E.Fname, E.Lname
       FROM     EMPLOYEE AS E
       WHERE    E.Ssn IN   ( SELECT   Essn
                             FROM     DEPENDENT AS D
                             WHERE    E.Fname=D.Dependent_name
                                      AND E.Sex=D.Sex );
```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Correlated Nested Queries

- **Queries that are nested using the = or IN comparison operator** can be collapsed into one single block: E.g., Q16 can be written as:

-
  ```
  Q16A:   SELECT      E.Fname, E.Lname
    FROM        EMPLOYEE AS E, DEPENDENT AS D
    WHERE       E.Ssn=D.Essn AND E.Sex=D.Sex
                      AND
    E.Fname=D.Dependent_name;
  ```
- **Correlated** nested query
  - Evaluated once for each tuple in the outer query

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# The EXISTS and UNIQUE Functions in SQL for correlating queries

- `EXISTS` function
  - Check whether the result of a correlated nested query is empty or not. They are Boolean functions that return a TRUE or FALSE result.
- `EXISTS` and `NOT EXISTS`
  - Typically used in conjunction with a correlated nested query
- SQL function `UNIQUE(Q)`
  - Returns `TRUE` if there are no duplicate tuples in the result of query Q

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# USE of EXISTS

**Q7:**

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

              AND EXISTS (SELECT   *
                                 FROM Department
                                 WHERE Ssn= Mgr_Ssn)
```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# USE OF NOT EXISTS

To achieve the “for all” (universal quantifier) effect, we use double negation this way in SQL:

Query: List first and last name of employees who work on ALL projects controlled by Dno=5.

```
SELECT Fname, Lname
FROM Employee
WHERE NOT EXISTS ( (SELECT  Pnumber
                           FROM PROJECT
                          WHERE Dno=5)

                 EXCEPT (SELECT   Pno
                           FROM WORKS_ON
                           WHERE Ssn= ESsn)
```

The above is equivalent to double negation: List names of those employees for whom there does NOT exist a project managed by department no. 5 that they do NOT work on.

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Double Negation to accomplish “for all” in SQL

```
Q3B:    SELECT          Lname, Fname
FROM                EMPLOYEE
WHERE  NOT EXISTS (      SELECT      *
              FROM WORKS_ON B
              WHERE      ( B.Pno IN  (  SELECT Pnumber
                             FROM PROJECT
                             WHERE Dnum=5                    AND

NOT EXISTS (SELECT *
                   FROM  WORKS_ON C
                   WHERE  C.Essn=Ssn
                   AND       C.Pno=B.Pno )));
```

The above is a direct rendering of: List names of those employees for whom there does NOT exist a project managed by department no. 5 that they do NOT work on.

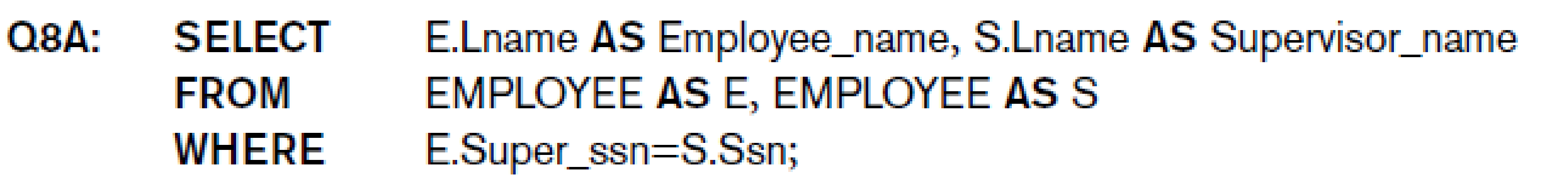
Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Explicit Sets and Renaming of Attributes in SQL

- Can use explicit set of values in WHERE clause

```
Q17:      SELECT DISTINCT Essn
  FROM        WORKS_ON
  WHERE Pno IN (1, 2, 3);
```

- Use qualifier AS followed by desired new name
  - Rename any attribute that appears in the result of a query

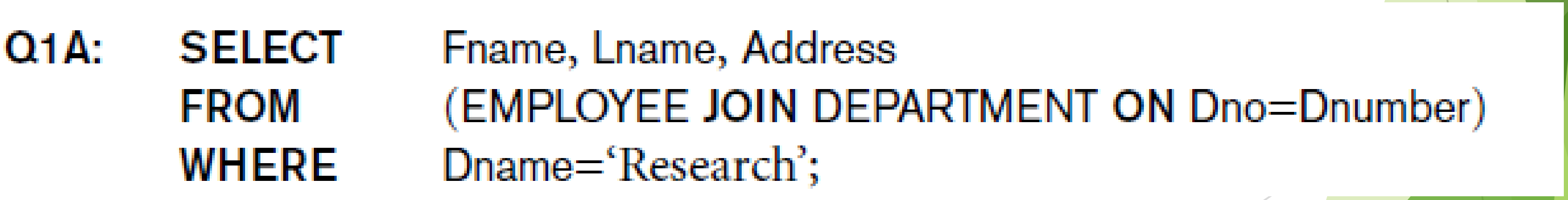


```
Q8A:    SELECT    E.Lname AS Employee_name, S.Lname AS Supervisor_name
        FROM      EMPLOYEE AS E, EMPLOYEE AS S
        WHERE     E.Super_ssn=S.Ssn;
```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Specifying Joined Tables in the FROM Clause of SQL

- **Joined table**
  - Permits users to specify a table resulting from a join operation in the FROM clause of a query
- The FROM clause in Q1A
  - Contains a single joined table. JOIN may also be called INNER JOIN



```
Q1A:    SELECT    Fname, Lname, Address
        FROM      (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
        WHERE     Dname='Research';
```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Different Types of JOINed Tables in SQL

- Specify different types of join
  - NATURAL JOIN
  - Various types of OUTER JOIN (LEFT, RIGHT, FULL )
- NATURAL JOIN on two relations R and S
  - No join condition specified
  - Is equivalent to an implicit EQUIJOIN condition for each pair of attributes with same name from R and S

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# NATURAL JOIN

- Rename attributes of one relation so it can be joined with another using NATURAL JOIN:

```
Q1B:    SELECT    Fname, Lname, Address
     FROM  (EMPLOYEE NATURAL JOIN
        (DEPARTMENT AS DEPT (Dname, Dno, Mssn,
                         Msdate)))
     WHERE     Dname='Research';
```

The above works with EMPLOYEE.Dno = DEPT.Dno as an implicit join condition

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# INNER and OUTER Joins

- INNER JOIN  **(versus** 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
- LEFT OUTER JOIN
  - Every tuple in left table must appear in result
  - If no matching tuple
    - Padded with NULL values for attributes of right table
- RIGHT OUTER JOIN
  - Every tuple in right table must appear in result
  - If no matching tuple
    - Padded with NULL values for attributes of left table

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Example: LEFT OUTER JOIN

```
SELECT E.Lname AS Employee_Name
             S.Lname AS Supervisor_Name

FROM Employee AS E LEFT OUTER JOIN EMPLOYEE AS S
                       ON E.Super_ssn = S.Ssn)
```

## ALTERNATE SYNTAX:

```
SELECT E.Lname , S.Lname
FROM  EMPLOYEE E, EMPLOYEE S
WHERE E.Super_ssn + = S.Ssn
```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Multiway JOIN in the FROM clause

- FULL OUTER JOIN – combines result if LEFT and RIGHT OUTER JOIN
- Can nest JOIN specifications for a multiway join:

```
Q2A:      SELECT Pnumber, Dnum, Lname, Address,
Bdate
 FROM       ((PROJECT JOIN DEPARTMENT ON
             Dnum=Dnumber)  JOIN EMPLOYEE ON
             Mgr_ssn=Ssn)
 WHERE    Plocation='Stafford';
```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Aggregate Functions in SQL

- Used to summarize information from multiple tuples into a single-tuple summary
- Built-in aggregate functions
  - **`COUNT`**, **`SUM`**, **`MAX`**, **`MIN`**, and **`AVG`**
- **Grouping**
  - Create subgroups of tuples before summarizing
- To select entire groups, `HAVING` clause is used
- Aggregate functions can be used in the `SELECT` clause or in a `HAVING` clause

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Renaming Results of Aggregation

- Following query returns a single row of computed values from EMPLOYEE table:

```
Q19:          SELECT    SUM (Salary), MAX (Salary), MIN
    (Salary), AVG                     (Salary)
         FROM      EMPLOYEE;
```

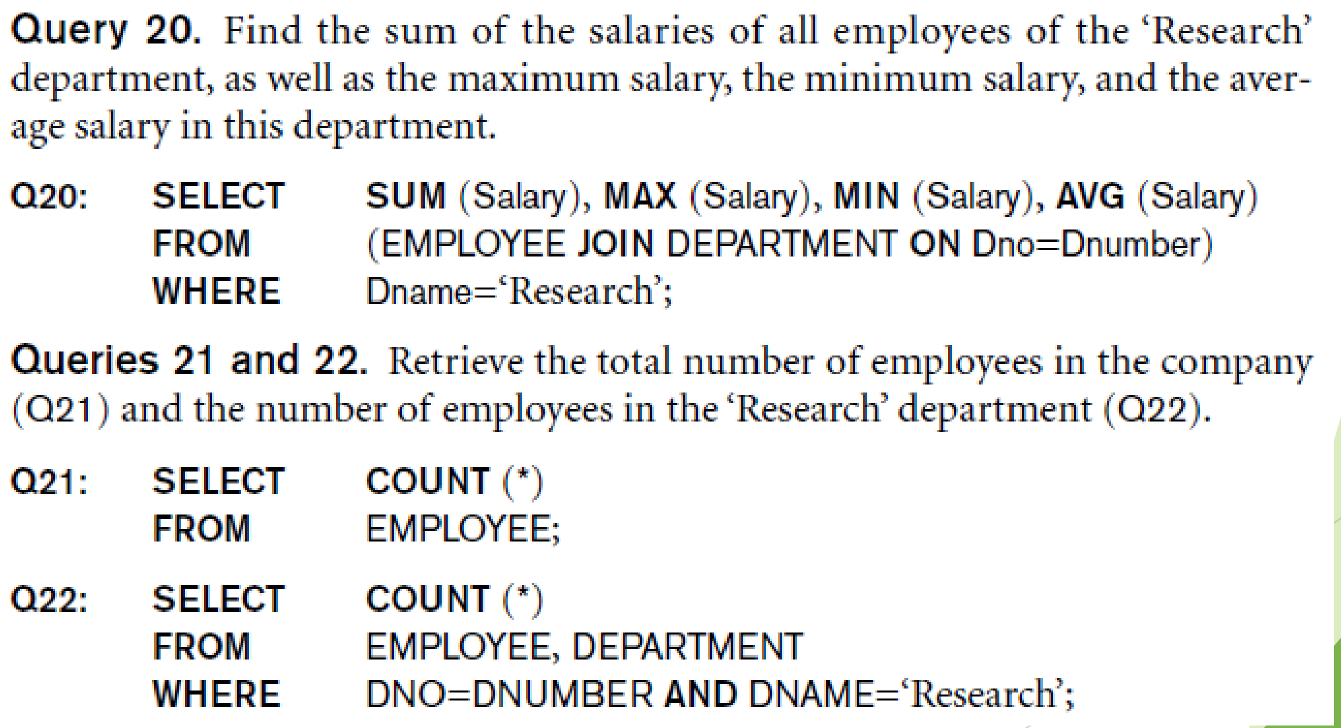
- The result can be presented with new names:

```
Q19A:        SELECT    SUM (Salary) AS Total_Sal, MAX (Salary)
    AS                 Highest_Sal, MIN (Salary) AS Lowest_Sal,
    AVG                (Salary) AS Average_Sal
         FROM      EMPLOYEE;
```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Aggregate Functions in SQL (cont'd.)

- NULL values are discarded when aggregate functions are applied to a particular column



**Query 20.** 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.

```
Q20:    SELECT    SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
        FROM      (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
        WHERE     Dname='Research';
```

**Queries 21 and 22.** Retrieve the total number of employees in the company (Q21) and the number of employees in the 'Research' department (Q22).

```
Q21:    SELECT    COUNT (*)
        FROM      EMPLOYEE;

Q22:    SELECT    COUNT (*)
        FROM      EMPLOYEE, DEPARTMENT
        WHERE     DNO=DNUMBER AND DNAME='Research';
```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Aggregate Functions on Booleans

- SOME and ALL  may be applied as functions on Boolean Values.
- SOME returns true if at least one element in the collection is TRUE (similar to OR)
- ALL returns true if all of the elements in the collection are TRUE (similar to AND)

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Grouping: The GROUP BY Clause

- **Partition** relation into subsets of tuples
  - Based on **grouping attribute(s)**
  - Apply function to each such group independently
- `GROUP BY` clause
  - Specifies grouping attributes
- COUNT (*) counts the number of rows in the group

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Examples of GROUP BY

- The grouping attribute must appear in the SELECT clause:

```
Q24:        SELECT     Dno, COUNT (*), AVG (Salary)
  FROM            EMPLOYEE
  GROUP BY Dno;
```

- If the grouping attribute has NULL as a possible value, then a separate group is created for the null value (e.g., null Dno in the above query)
- GROUP BY may be applied to the result of a JOIN:

```
Q25:   SELECT            Pnumber, Pname, COUNT (*)
  FROM       PROJECT, WORKS_ON
  WHERE           Pnumber=Pno
  GROUP BY   Pnumber, Pname;
```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Grouping: The GROUP BY and HAVING Clauses (cont’d.)

- **HAVING** clause
  - Provides a condition to select or reject an entire group:
- **Query 26.** 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.

```
Q26:      SELECT Pnumber, Pname, COUNT (*)
 FROM          PROJECT, WORKS_ON
 WHERE  Pnumber=Pno
 GROUP BY   Pnumber, Pname
 HAVING COUNT (*) > 2;
```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Combining the WHERE and the HAVING Clause

- Consider the query: 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.

- INCORRECT QUERY:

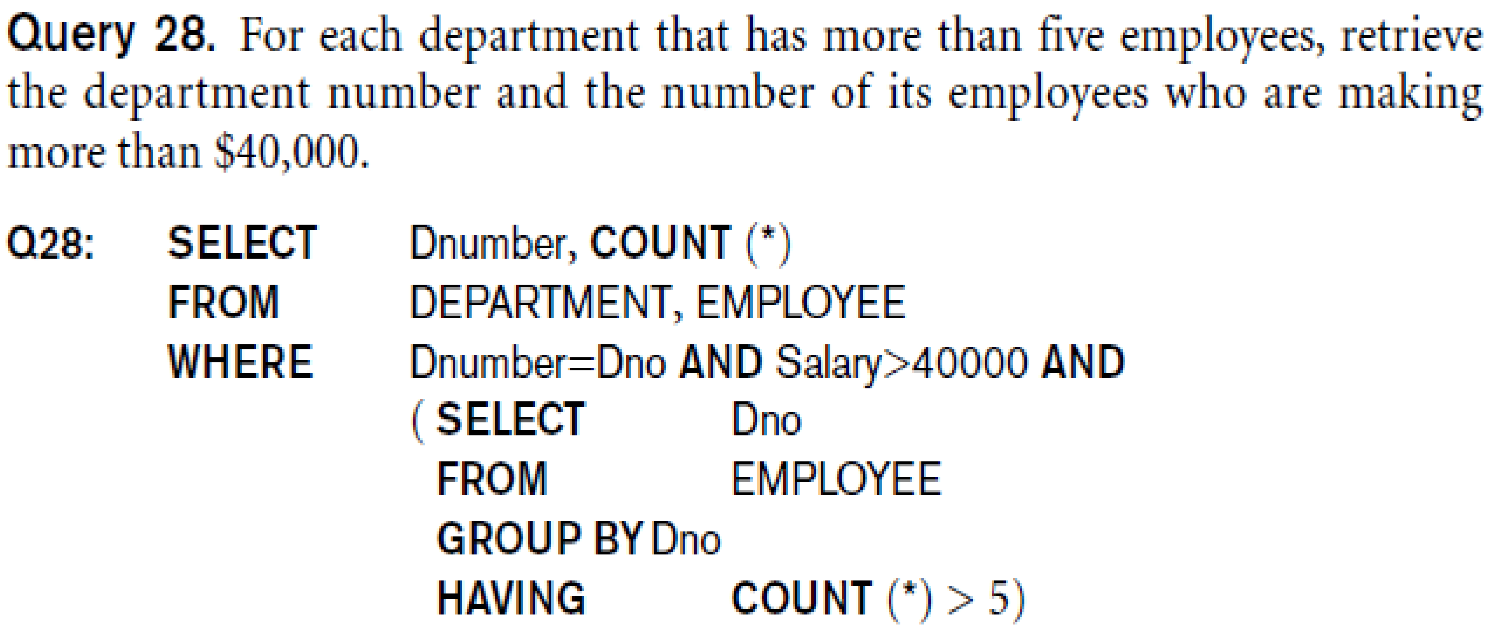
```
SELECT          Dno, COUNT (*)
FROM      EMPLOYEE
WHERE           Salary>40000
GROUP BYDno
HAVING          COUNT (*) > 5;
```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Combining the WHERE and the HAVING Clause (continued)

Correct Specification of the Query:

- Note: the WHERE clause applies tuple by tuple whereas HAVING applies to entire group of tuples



**Query 28.** 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.

```
Q28:   SELECT     Dnumber, COUNT (*)
       FROM       DEPARTMENT, EMPLOYEE
       WHERE      Dnumber=Dno AND Salary>40000 AND
                  ( SELECT      Dno
                    FROM        EMPLOYEE
                    GROUP BY Dno
                    HAVING      COUNT (*) > 5)
```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Use of WITH

- The WITH clause allows a user to define a table that will only be used in a particular query (not available in all SQL implementations)
- Used for convenience to create a temporary "View" and use that immediately in a query
- Allows a more straightforward way of looking a step-by-step query

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Example of WITH

- See an alternate approach to doing Q28:

- Q28’:

```
Q28’:              WITHBIGDEPTS (Dno) AS
               ( SELECT   Dno
            FROM      EMPLOYEE
            GROUP BY Dno
            HAVING   COUNT (*) > 5)
        SELECT          Dno, COUNT (*)
        FROM            EMPLOYEE
        WHERE           Salary>40000 AND Dno IN BIGDEPTS
        GROUP BY Dno;
```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Use of CASE

- SQL also has a CASE construct
- 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

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# EXAMPLE of use of CASE

- The following example shows that employees are receiving different raises in different departments (A variation of the update U6)

-
  ```
  U6':        UPDATE      EMPLOYEE
   SET        Salary =
   CASE           WHEN   Dno = 5 THEN      Salary +
  2000
              WHEN   Dno = 4 THEN      Salary + 1500
              WHEN   Dno = 1 THEN      Salary + 3000
  ```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Recursive Queries in SQL

- An example of a **recursive relationship** between tuples of the same type is the relationship between an employee and a supervisor.
- This relationship is described by the foreign key Super_ssn of the EMPLOYEE relation
- An example of a **recursive operation** is to retrieve all supervisees of a supervisory employee *e* at all levels—that is, all employees *e*′ directly supervised by *e*, all employees *e*’’ directly supervised by each employee *e*′, all employees *e*‴ directly supervised by each employee *e*″, and so on. Thus the CEO would have each employee in the company as a supervisee in the resulting table. Example shows such table SUP_EMP with 2 columns (Supervisor,Supervisee(any level)):

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# An EXAMPLE of RECURSIVE Query

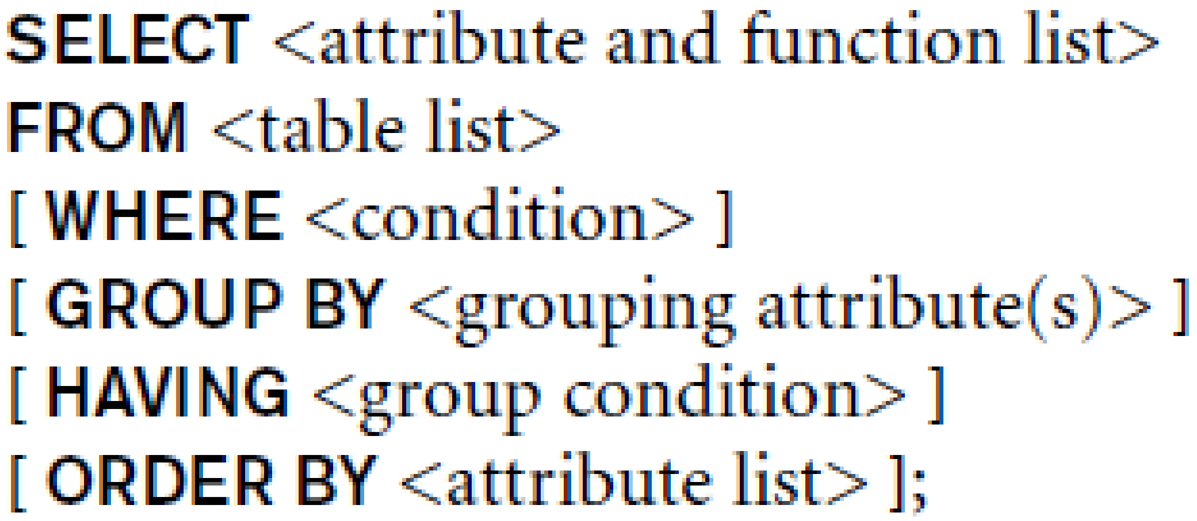
- **Q29:**

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

- The above query starts with an empty SUP_EMP and successively builds SUP_EMP table by computing immediate supervisees first, then second level supervisees, etc. until a **fixed point** is reached and no more supervisees can be added

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# 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> ];
```

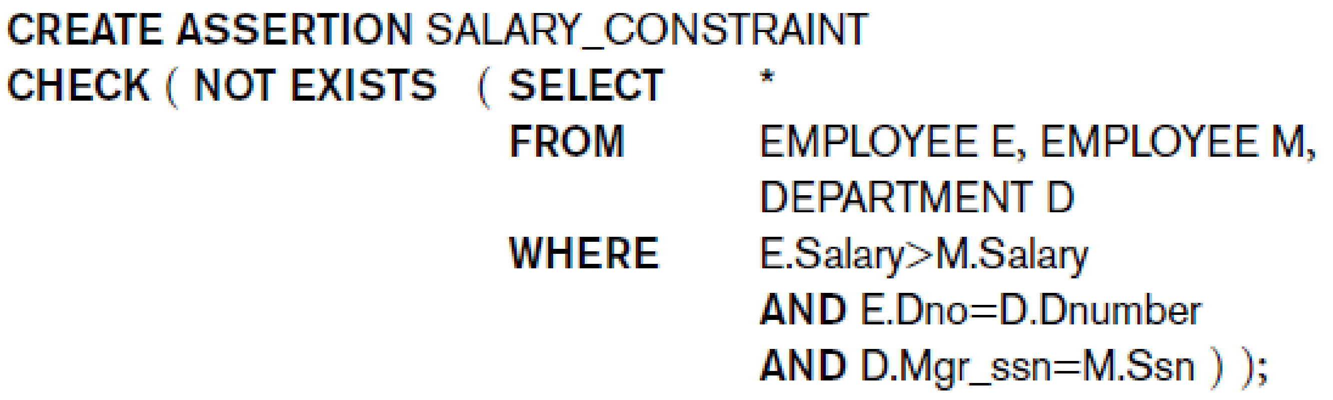
# Specifying Constraints as Assertions and Actions as Triggers

- Semantic Constraints: The following are beyond the scope of the EER and relational model
- **`CREATE ASSERTION`**
  - Specify additional types of constraints outside scope of built-in relational model constraints
- **`CREATE TRIGGER`**
  - Specify automatic actions that database system will perform when certain events and conditions occur

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Specifying General Constraints as Assertions in SQL

- `CREATE ASSERTION`
  - Specify a query that selects any tuples that violate the desired condition
  - Use only in cases where it goes beyond 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 ) );
```

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# Introduction to Triggers in SQL

- `CREATE TRIGGER` statement
  - Used to monitor the database
- Typical trigger has three components which make it a rule for an “active database “ (more on active databases) :
  - **Event(s)**
  - **Condition**
  - **Action**

Slide Credits: Ramez Elmasri and Shamkant B. Navathe

# USE OF TRIGGERS

- AN EXAMPLE with standard Syntax.(Note : other SQL implementations like PostgreSQL use a different syntax.)

**R5:**

```
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)
```

**Sample Example: http://www.mysqltutorial.org/create-the-first-trigger-in-mysql.aspx**

Slide Credits: Ramez Elmasri and Shamkant B. Navathe