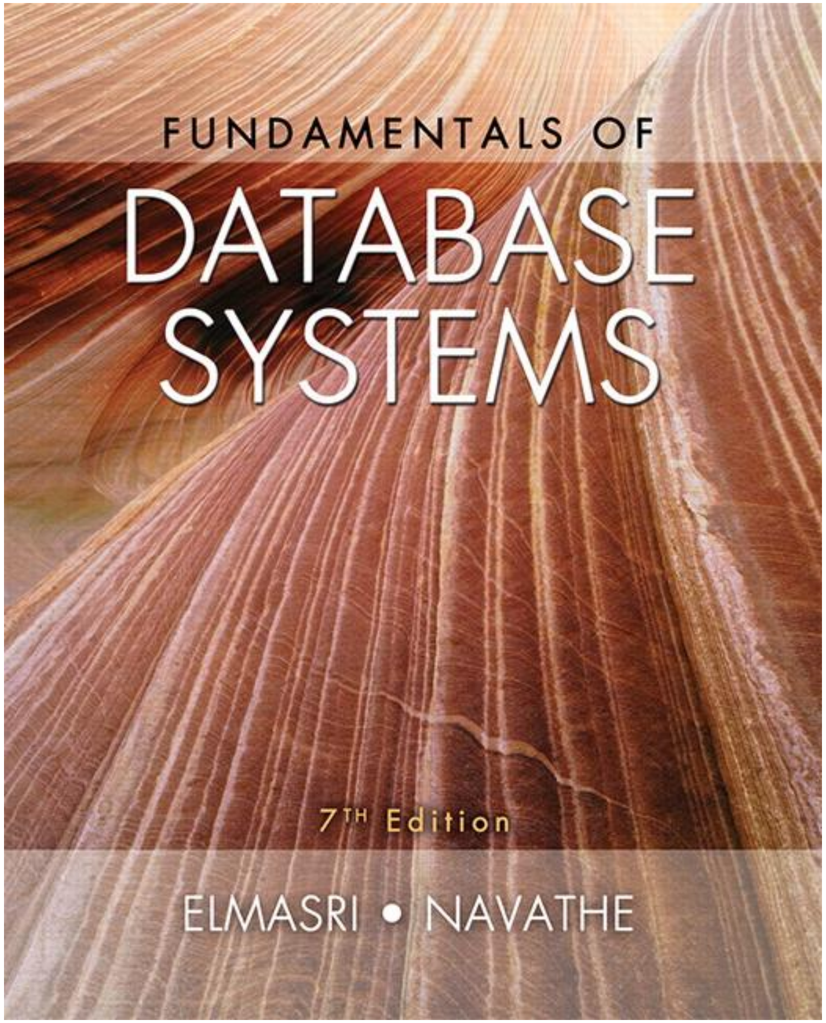

FUNDAMENTALS OF
DATABASE
SYSTEMS
7TH Edition
ELMASRI • NAVATHE

# CHAPTER 7

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

# 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

# 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

# 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**

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

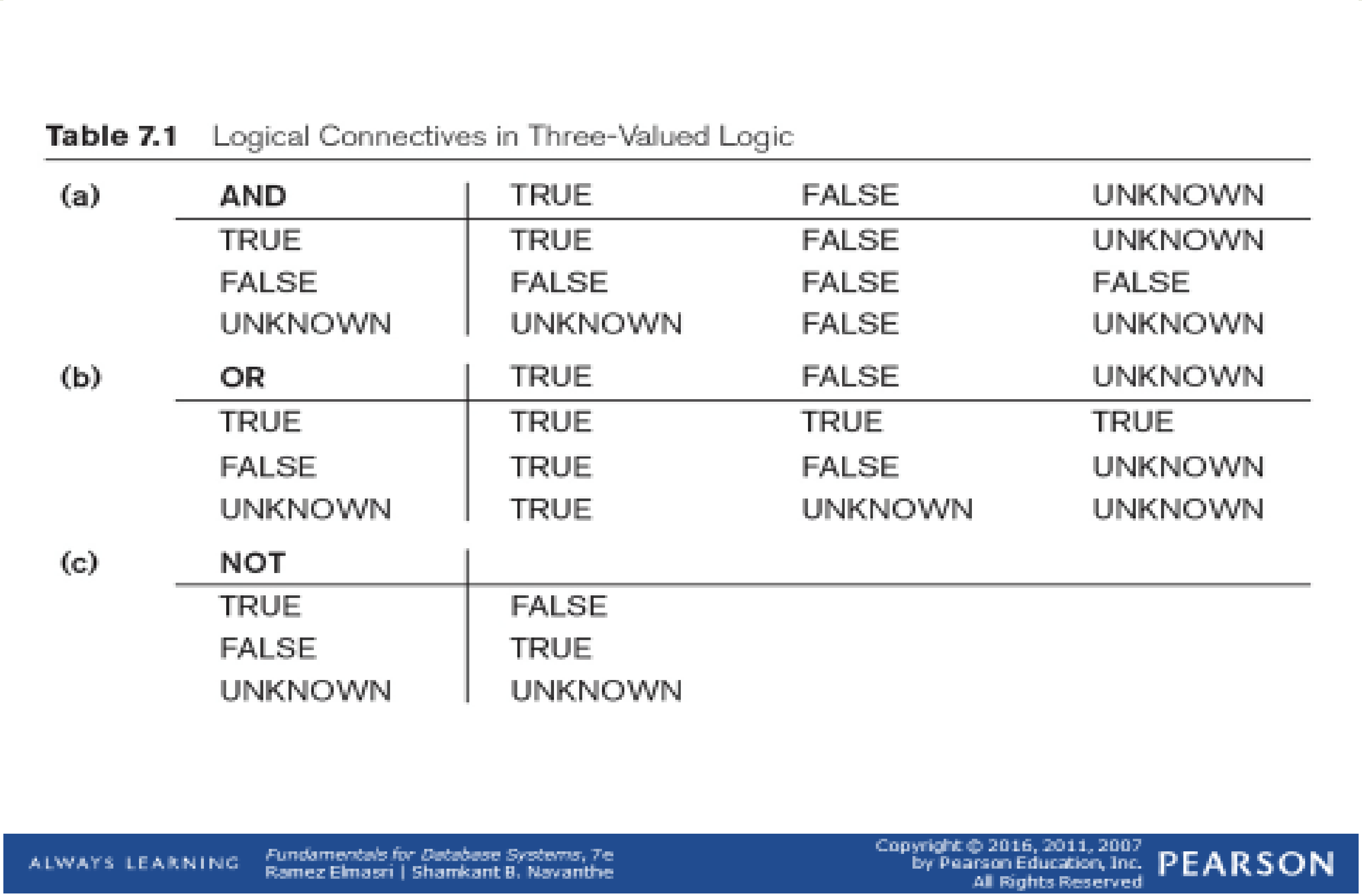


**Table 7.1** Logical Connectives in Three-Valued Logic

(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)

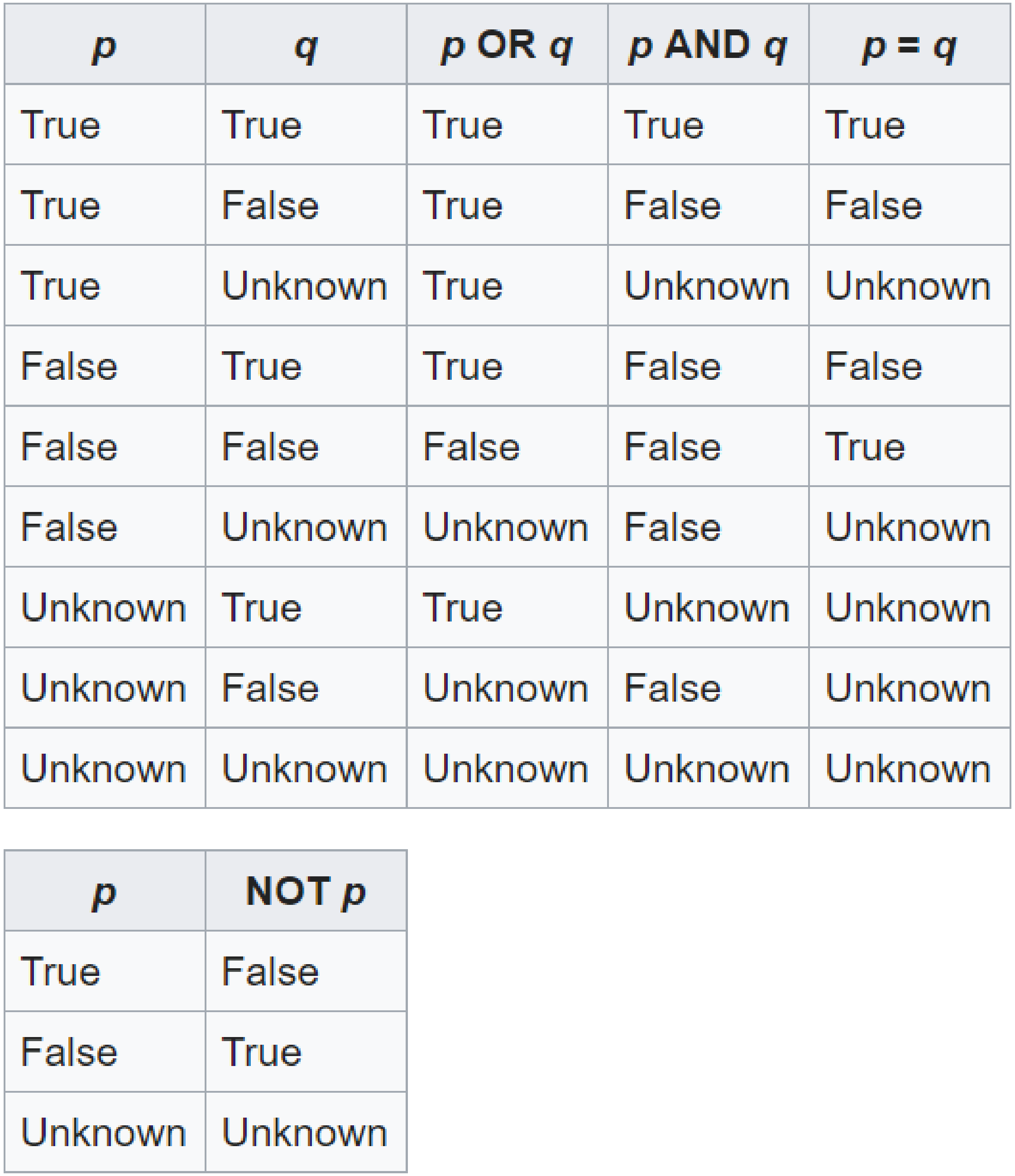
| NOT | |
|---|---|
| TRUE | FALSE |
| FALSE | TRUE |
| UNKNOWN | UNKNOWN |

ALWAYS LEARNING

*Fundamentals for Database Systems, 7e*
Ramez Elmasri | Shamkant B. Navathe

PEARSON

# Three-Valued Logic

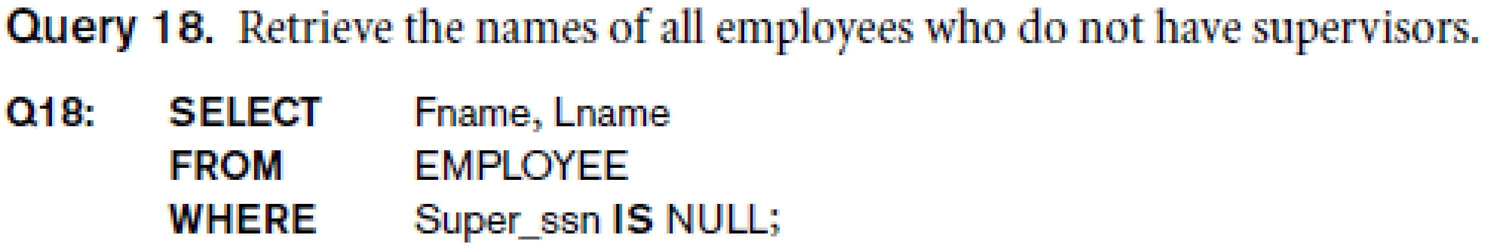


| *p* | *q* | *p* OR *q* | *p* AND *q* | *p* = *q* |
|---|---|---|---|---|
| True | True | True | True | True |
| True | False | True | False | False |
| True | Unknown | True | Unknown | Unknown |
| False | True | True | False | False |
| False | False | False | False | True |
| False | Unknown | Unknown | False | Unknown |
| Unknown | True | True | Unknown | Unknown |
| Unknown | False | Unknown | False | Unknown |
| Unknown | Unknown | Unknown | Unknown | Unknown |

| *p* | NOT *p* |
|---|---|
| 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.

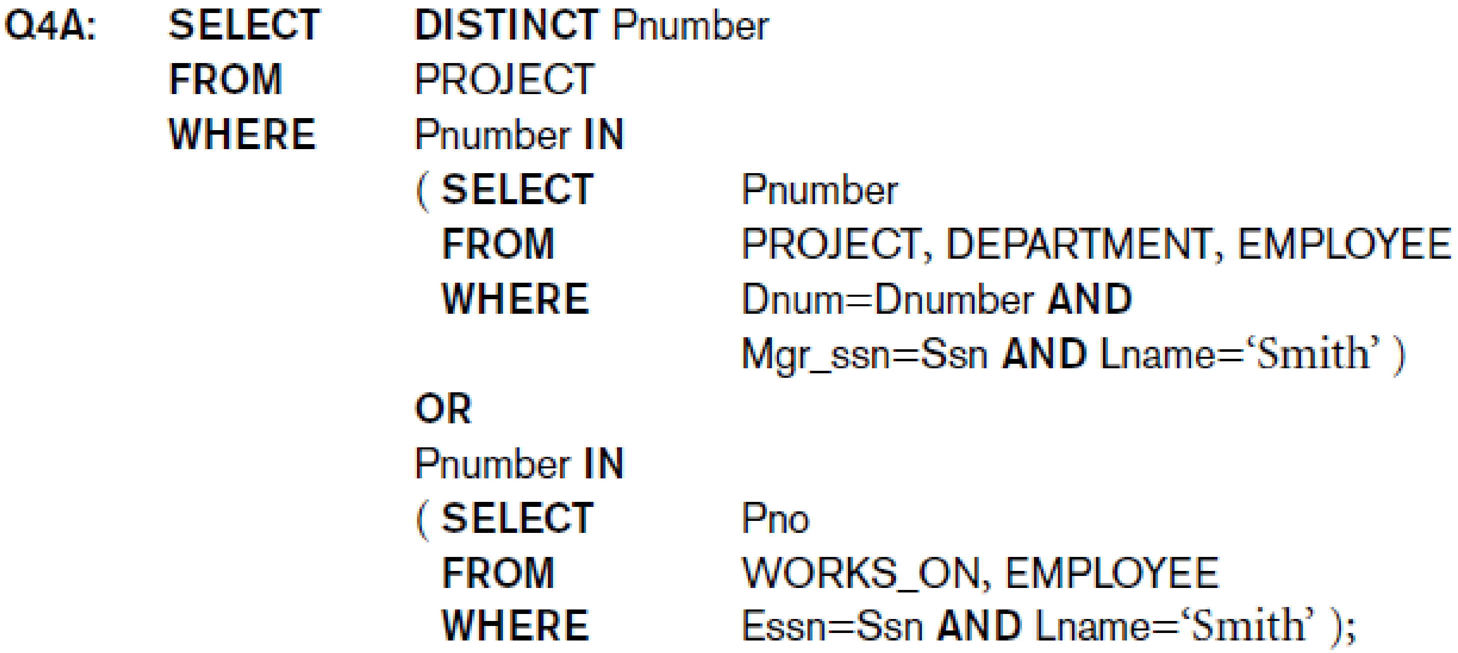
```
Q18:    SELECT    Fname, Lname
        FROM      EMPLOYEE
        WHERE     Super_ssn IS NULL;
```

# 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*

# Nested Queries (cont'd.)

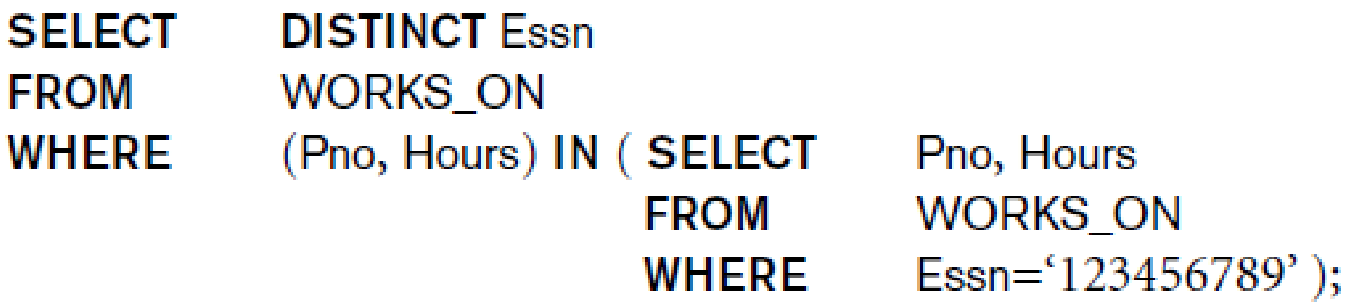
Make a list of all project numbers for projects that involve employee Smith either as worker or as a manager of the department that controls the project:



```
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.)

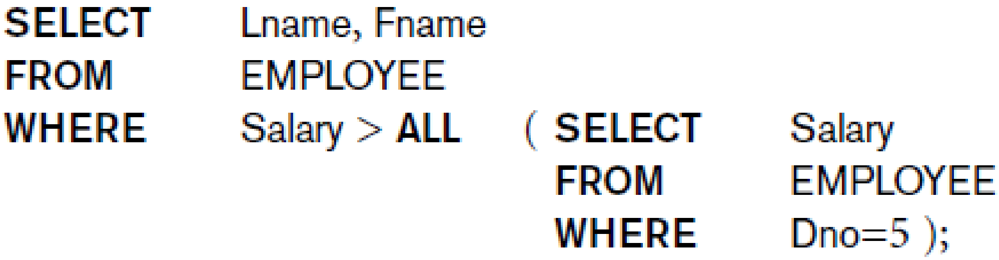
- Use tuples of values in comparisons
  - Place them within parentheses



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

# Nested Queries (cont'd.)

- Use other comparison operators to compare a single value *v*
  - `= ANY` (or `= SOME`) operator [equivalent to IN]
    - Returns `TRUE` if the value *v* is equal to some value in the set
  - Other operators that can be combined with `ANY` (or `SOME`): >, >=, <, <=, and <>
  - `ALL:` value must exceed all values from nested query



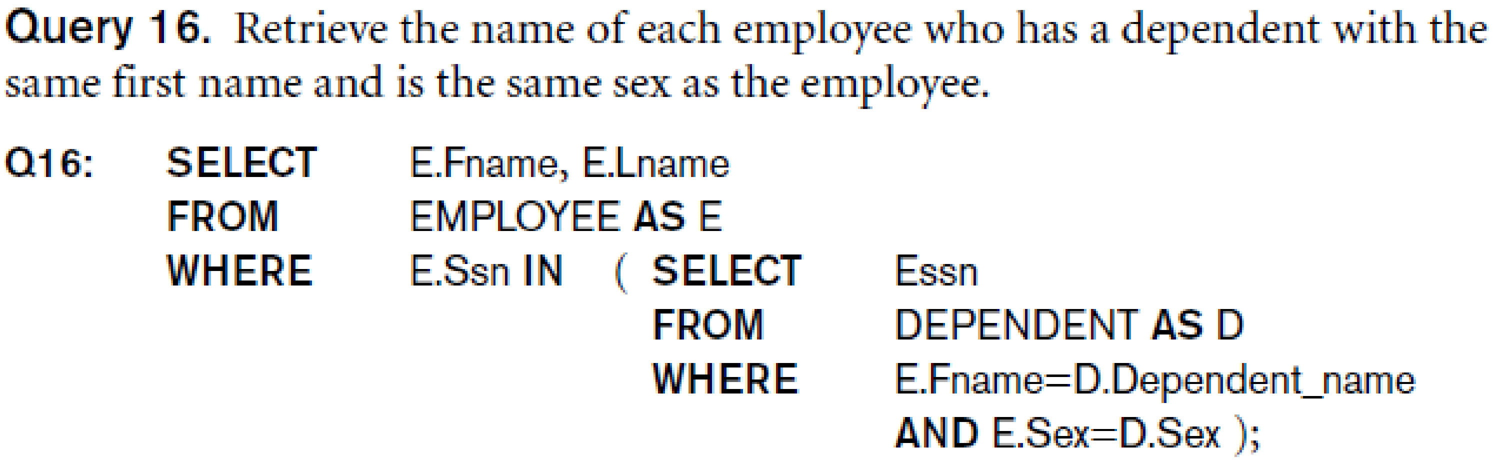
```
SELECT    Lname, Fname
FROM      EMPLOYEE
WHERE     Salary > ALL    ( SELECT    Salary
                            FROM      EMPLOYEE
                            WHERE     Dno=5 );
```

# General Form of ALL, ANY, SOME

```
SELECT [column_name ]
FROM [table_name]
WHERE expression operator
        {ALL | ANY | SOME} ( subquery )
```

# Nested Queries (cont'd.)

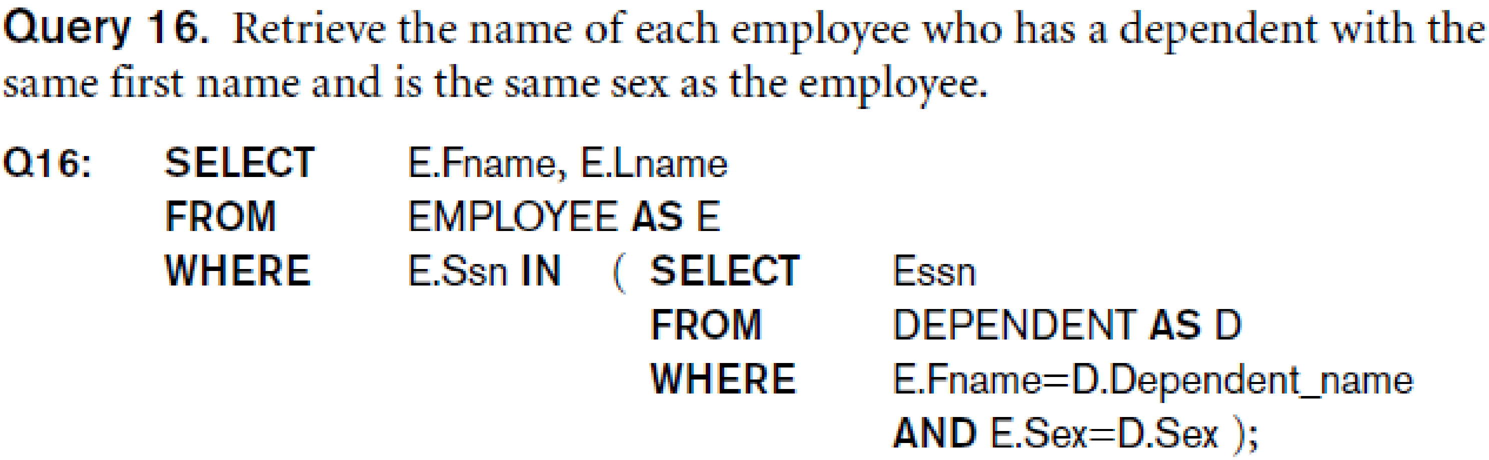
- Avoid potential errors and ambiguities
  - Create tuple variables (aliases) for all tables referenced in SQL query



**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 );
```

# Understanding a nested (correlated 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 );
```

For each E tuple,
  Evaluate the nested query
    which retrieves the Essn values of all D tuples
    with the same sex and name as E tuple
  If the Ssn value of E tuple is in the result,
    then select the E tuple

# 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

# 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

# USE of EXISTS

List the managers who have at least one dependent

**Q7:**

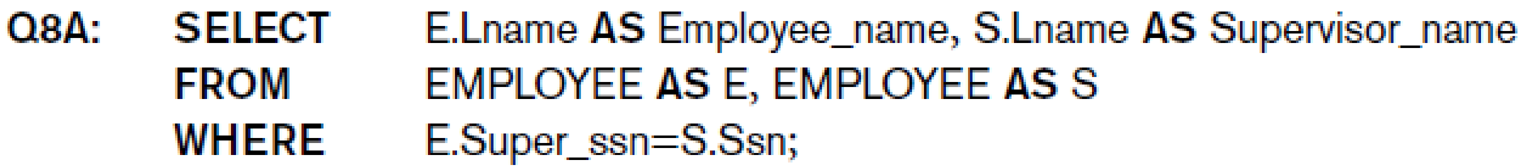
```
SELECT Fname, Lname
FROM Employee
WHERE EXISTS (SELECT *
               FROM DEPENDENT
               WHERE Ssn= Essn)

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

# 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);
```

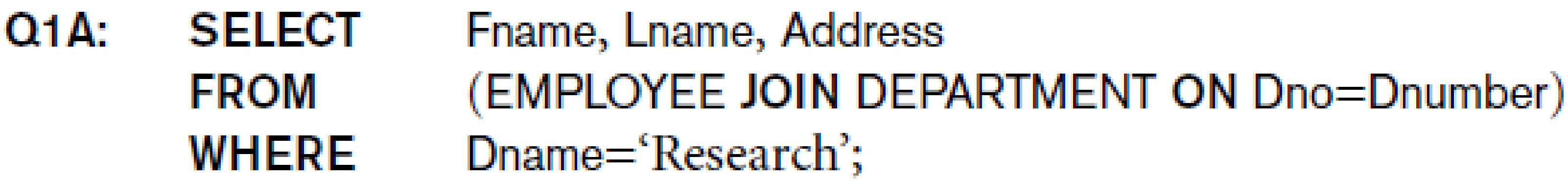
- Use qualifier AS followed by desired new name
  - Rename any attribute that appears in the result of a query



```
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;
```

# 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';
```

# 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

# 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

# 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

# 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

# Renaming Results of Aggregation

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

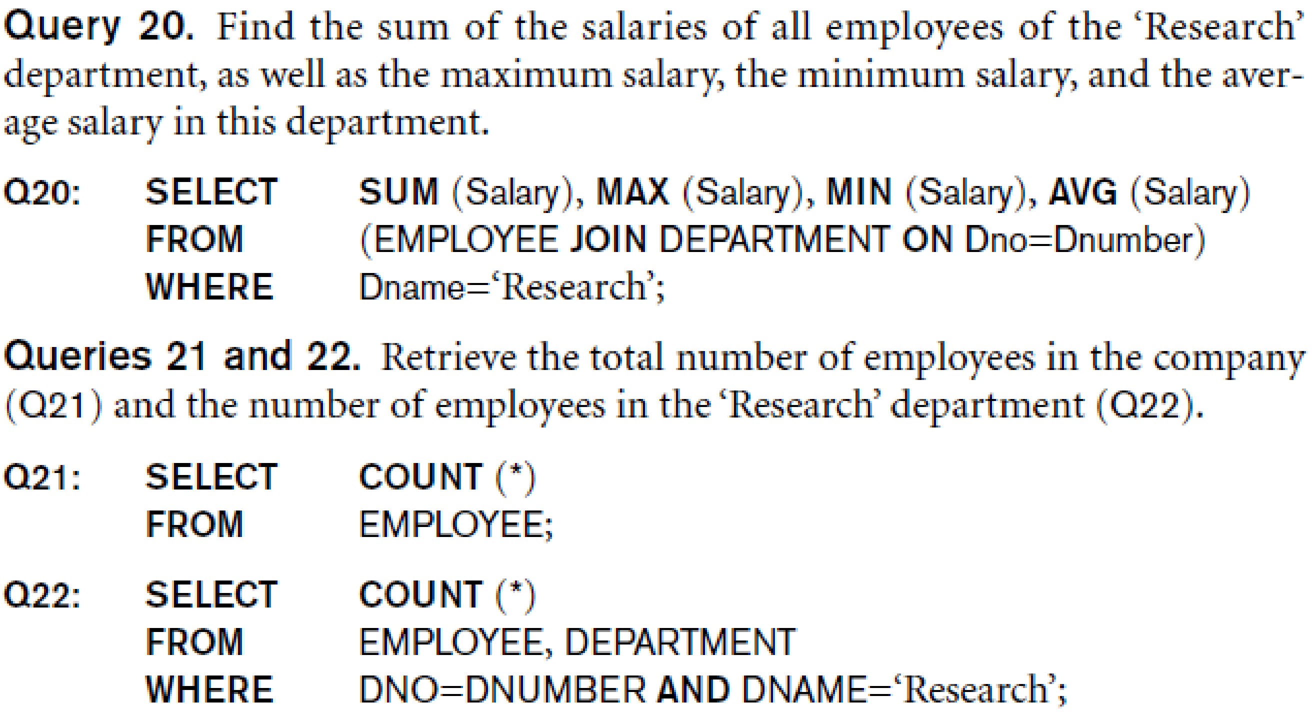
```
Q19:        SELECT    SUM (Salary), MAX (Salary), MIN (Salary), AVG
                      (Salary)
             FROM     EMPLOYEE;
```

- 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;
```

# 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';
```

# 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

# 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;
```

# 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;
```

# 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.

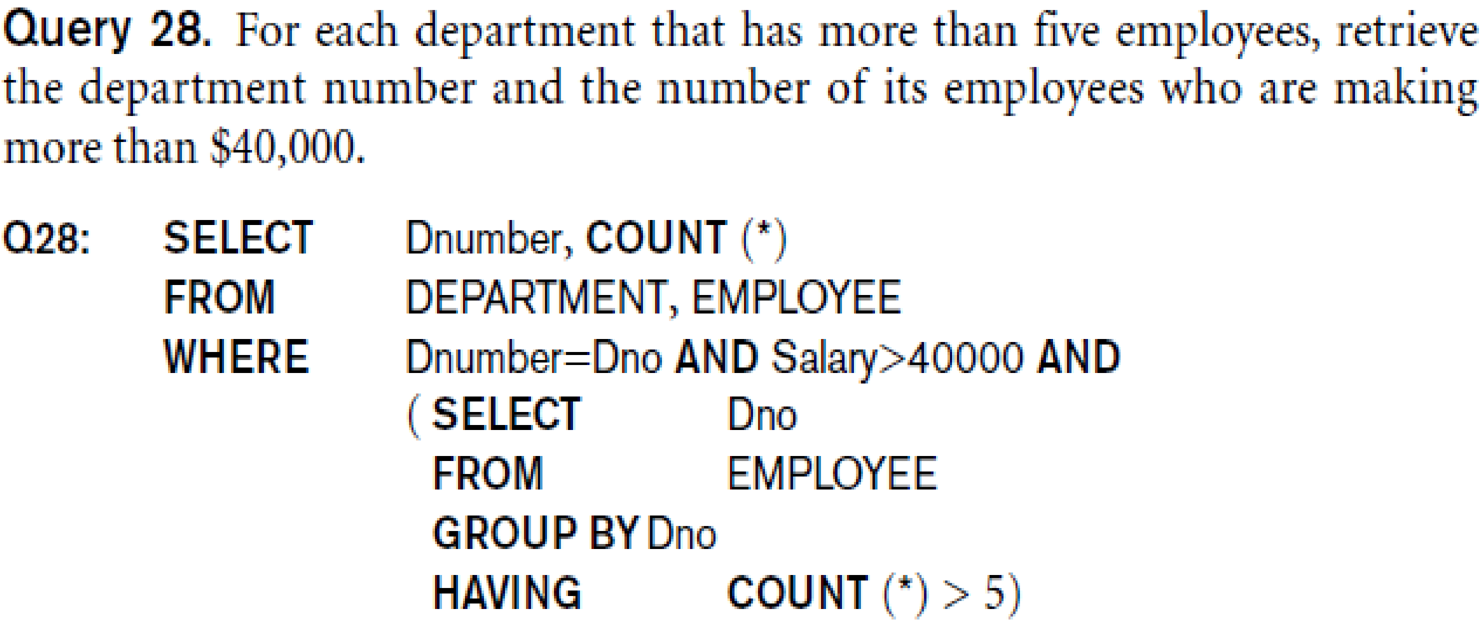
- INCORRECT QUERY:

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

# 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)
```

# 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

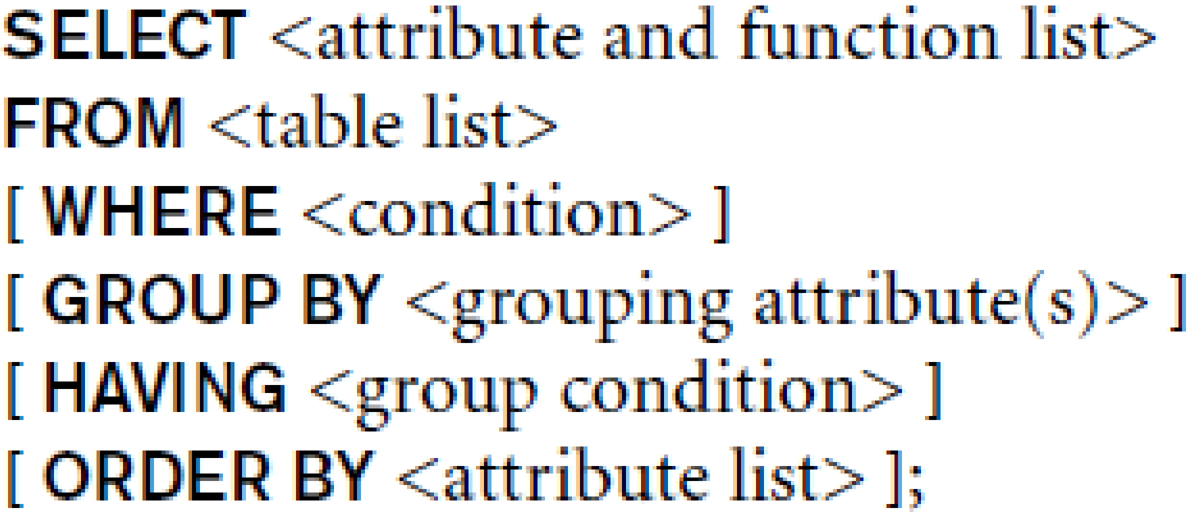
# EXAMPLE of use of CASE

- 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
```

# EXPANDED Block Structure of SQL Queries



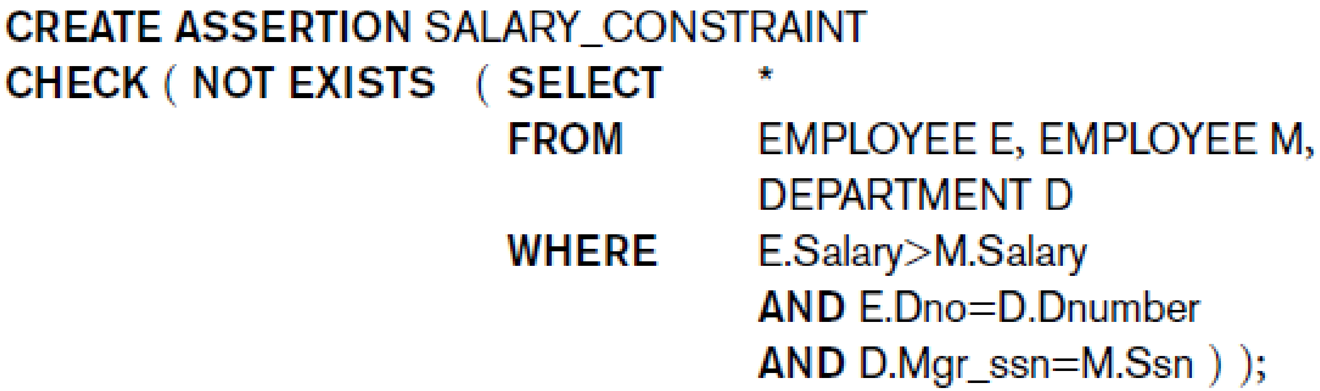
```
SELECT <attribute and function list>
FROM <table list>
[ WHERE <condition> ]
[ GROUP BY <grouping attribute(s)> ]
[ HAVING <group condition> ]
[ ORDER BY <attribute list> ];
```

# 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

# 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
  - Salary of an employee must be less than the manager



```
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 ) );
```

# 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 in section 26.1) :
  - **Event(s)**
  - **Condition**
  - **Action**

# USE OF TRIGGERS

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

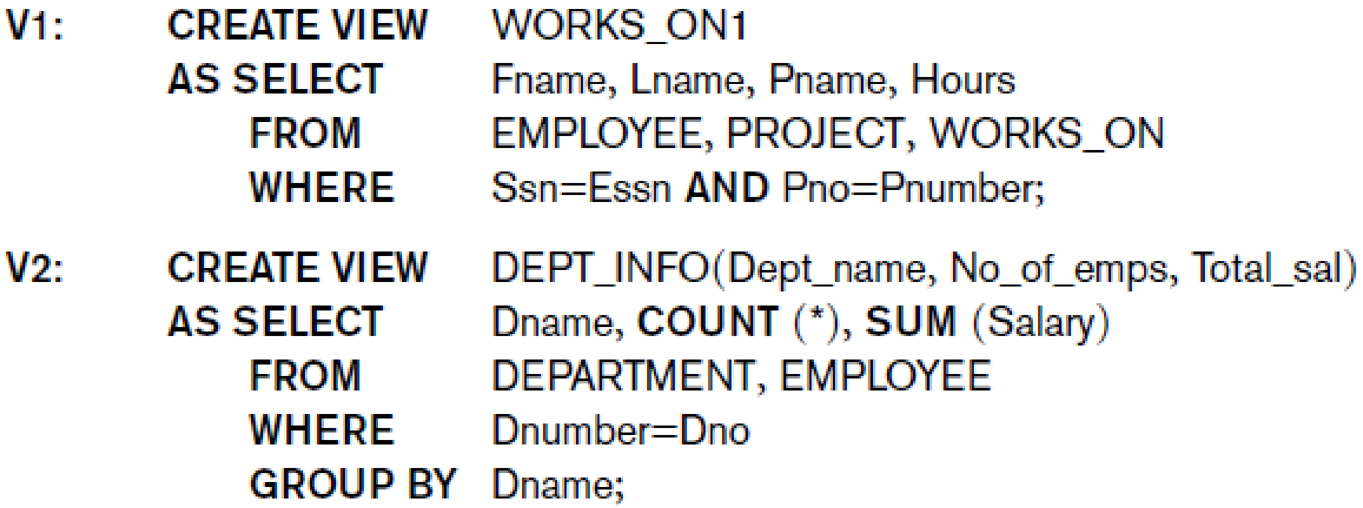
```
R5:
CREATE TRIGGER SALARY_VIOLATION
BEFORE INSERT OR UPDATE OF Salary, Supervisor_ssn ON
EMPLOYEE

FOR EACH ROW
WHEN (NEW.SALARY > ( SELECT Salary FROM EMPLOYEE
                     WHERE Ssn = NEW. Supervisor_Ssn))
INFORM_SUPERVISOR (NEW.Supervisor.Ssn, New.Ssn)
```

# Views (Virtual Tables) in SQL

- Concept of a view in SQL
  - Single table derived from other tables called the **defining tables**
  - Considered to be a virtual table that is not necessarily populated

# Specification of Views in SQL

- **CREATE VIEW** command
  - Give table name, list of attribute names, and a query to specify the contents of the view
  - In V1, attributes retain the names from base tables. In V2, attributes are assigned names



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

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

# Specification of Views in SQL (cont'd.)

- Once a View is defined, SQL queries can use the View relation in the FROM clause
- View is always up-to-date
  - Responsibility of the DBMS and not the user
- **`DROP VIEW`** command
  - Dispose of a view

# Schema Change Statements in SQL

- **Schema evolution commands**
  - DBA may want to change the schema while the database is operational
  - Does not require recompilation of the database schema

# The DROP Command

- `DROP` command
  - Used to drop named schema elements, such as tables, domains, or constraint
- Drop behavior options:
  - `CASCADE` and `RESTRICT`
- Example:
  - `DROP SCHEMA COMPANY CASCADE;`
  - This removes the schema and all its elements including tables, views, constraints, etc.
  - RESTRICT: drops only nothing in it

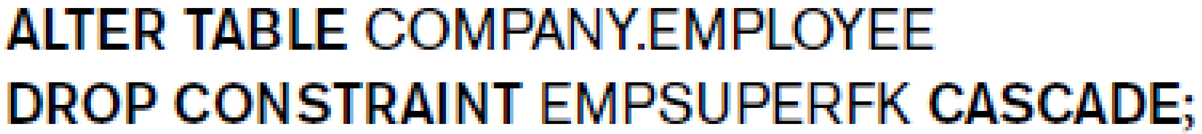
# The ALTER table command

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

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

# Adding and Dropping Constraints

- Change constraints specified on a table
  - Add or drop a named constraint



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

# Dropping Columns, Default Values

- To drop a column
  - Choose either `CASCADE` or `RESTRICT`
  - `CASCADE` would drop the column from views etc. `RESTRICT` is possible if no views refer to it.

    **ALTER TABLE** COMPANY.EMPLOYEE **DROP COLUMN** Address **CASCADE**;

- Default values can be dropped and altered :

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

  **ALTER TABLE** COMPANY.DEPARTMENT **ALTER COLUMN** Mgr_ssn **SET DEFAULT** ‘333445555’;

# **Table 7.2** Summary of SQL Syntax

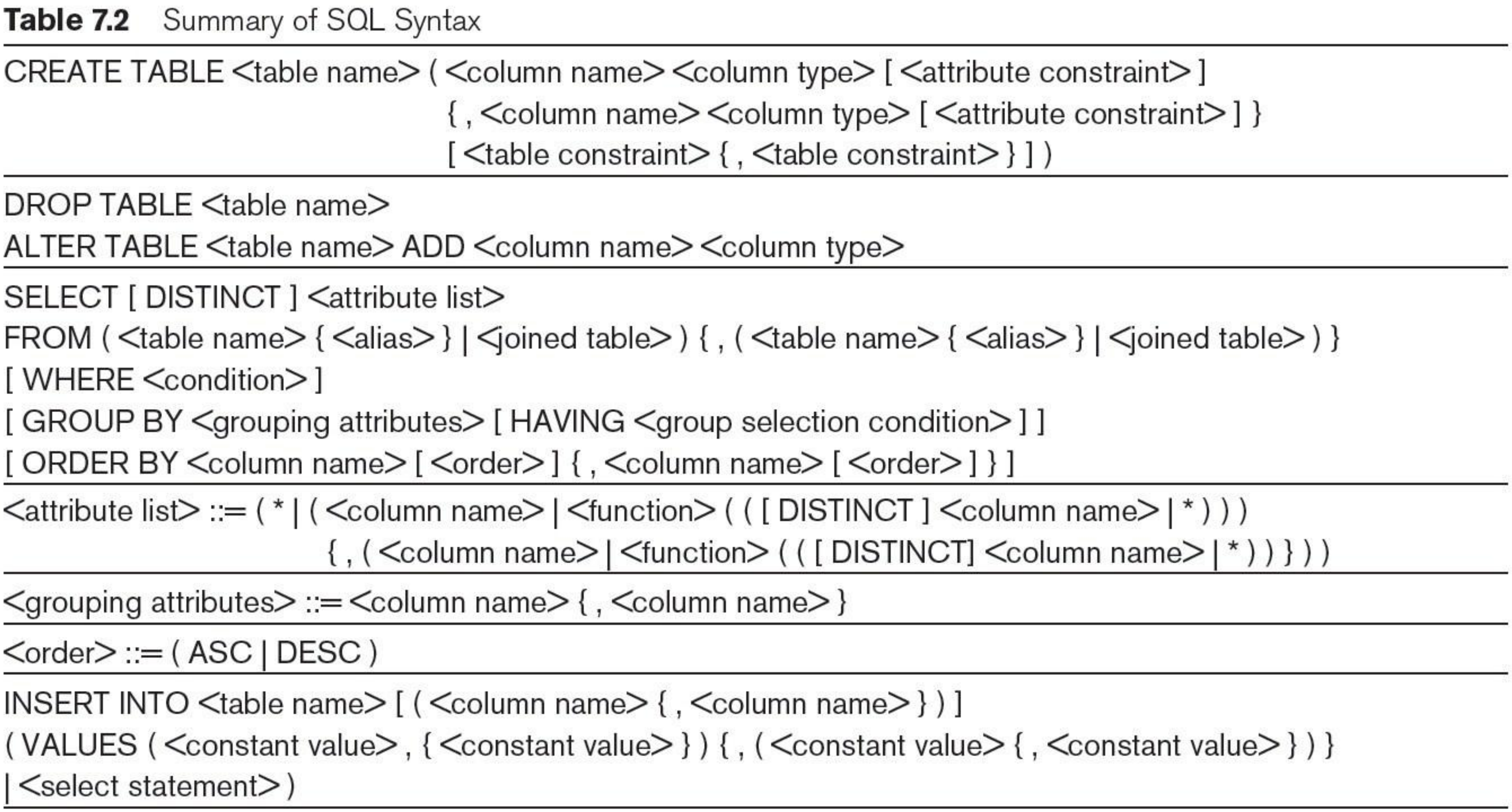


**Table 7.2** Summary of SQL Syntax

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

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

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

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

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

```
<order> ::= ( ASC | DESC )
```

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

*continued on next slide*

# Table 7.2 (continued)
## Summary of SQL Syntax

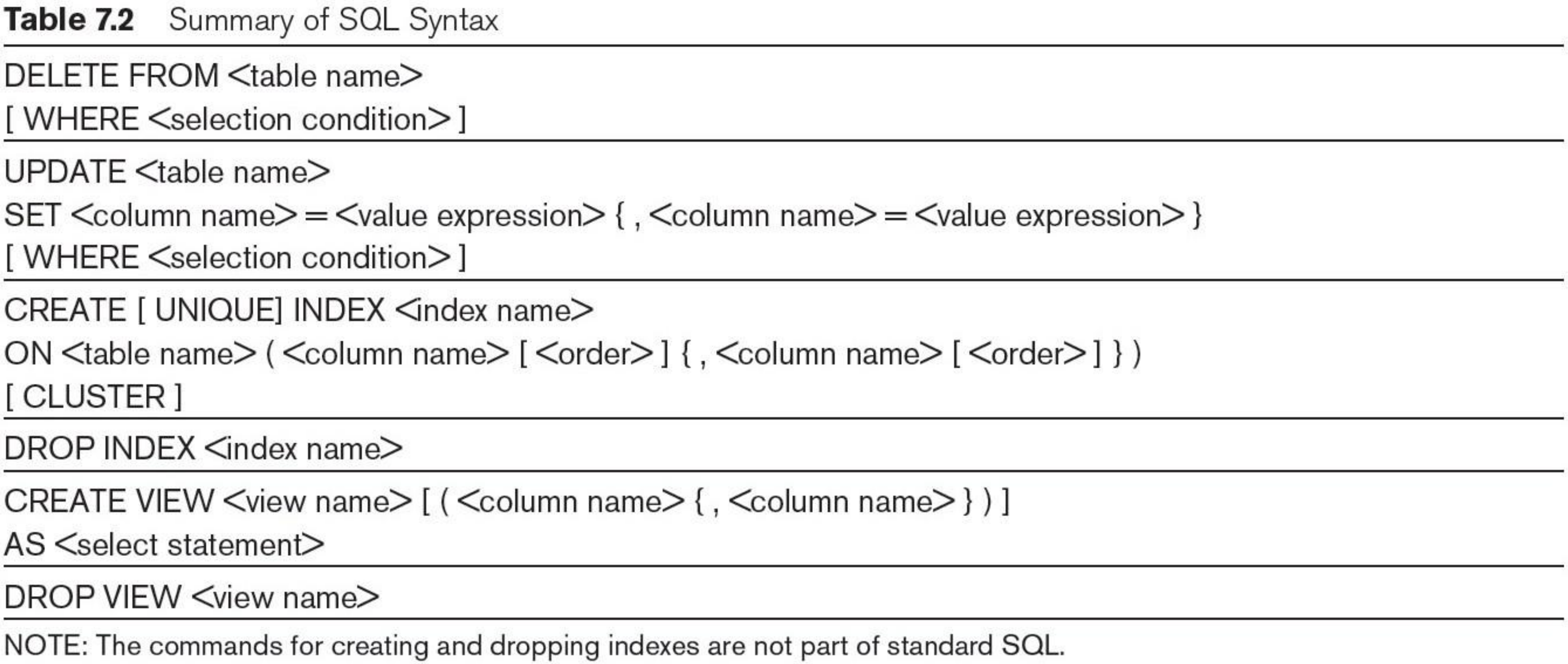


**Table 7.2** Summary of SQL Syntax

| |
|---|
| DELETE FROM <table name><br>[ WHERE <selection condition> ] |
| UPDATE <table name><br>SET <column name> = <value expression> { , <column name> = <value expression> }<br>[ WHERE <selection condition> ] |
| CREATE [ UNIQUE] INDEX <index name><br>ON <table name> ( <column name> [ <order> ] { , <column name> [ <order> ] } )<br>[ CLUSTER ] |
| DROP INDEX <index name> |
| CREATE VIEW <view name> [ ( <column name> { , <column name> } ) ]<br>AS <select statement> |
| DROP VIEW <view name> |

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

# Summary

- Complex SQL:
  - Nested queries, joined tables (in the FROM clause), outer joins, aggregate functions, grouping
- Handling semantic constraints with `CREATE ASSERTION` and `CREATE TRIGGER`
- `CREATE VIEW` statement and materialization strategies
- Schema Modification for the DBAs using `ALTER TABLE , ADD and DROP COLUMN, ALTER CONSTRAINT` etc.