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
Table 7.1  Logical Connectives in Three-Valued Logic

<table>
<thead>
<tr>
<th></th>
<th>AND</th>
<th>TRUE</th>
<th>FALSE</th>
<th>UNKNOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td></td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td>FALSE</td>
<td>UNKNOWN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b)</th>
<th>OR</th>
<th>TRUE</th>
<th>FALSE</th>
<th>UNKNOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>TRUE</td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(c)</th>
<th>NOT</th>
<th>TRUE</th>
<th>FALSE</th>
<th>UNKNOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

```sql
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 (i.e., outer 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`

- If a nested query returns a single attribute and a single tuple, the query result will be a single value. In such a case, we can use `=` instead of `IN` for the comparison operator.
Q4A: Make a list of 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.

Q4A: 

\[
\begin{align*}
\text{SELECT} & \quad \text{DISTINCT Pnumber} \\
\text{FROM} & \quad PROJECT \\
\text{WHERE} & \quad \text{Pnumber} \text{ IN} \\
& \quad (\text{SELECT} \quad \text{Pnumber} \\
& \quad \text{FROM} \quad \text{PROJECT, DEPARTMENT, EMPLOYEE} \\
& \quad \text{WHERE} \quad \text{Dnum} = \text{Dnumber AND} \\
& \quad \text{Mgr_ssn} = \text{Ssn AND Lname} = \text{‘Smith’} ) \\
\text{OR} & \quad \text{Pnumber} \text{ IN} \\
& \quad (\text{SELECT} \quad \text{Pno} \\
& \quad \text{FROM} \quad \text{WORKS_ON, EMPLOYEE} \\
& \quad \text{WHERE} \quad \text{Essn} = \text{Ssn AND Lname} = \text{‘Smith’} );
\end{align*}
\]
Without a nested query

**Query 4.** 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.

Q4A:  
\[
\begin{align*}
\text{(SELECT} & \quad \text{DISTINCT Pnumber} \\
\text{FROM} & \quad \text{PROJECT, DEPARTMENT, EMPLOYEE} \\
\text{WHERE} & \quad \text{Dnum=Dnumber AND Mgr_ssn=Ssn} \\
& \quad \text{AND Lname='Smith')} \\
\text{UNION} & \\
\text{(SELECT} & \quad \text{DISTINCT Pnumber} \\
\text{FROM} & \quad \text{PROJECT, WORKS_ON, EMPLOYEE} \\
\text{WHERE} & \quad \text{Pnumber=Pno AND Essn=Ssn} \\
& \quad \text{AND Lname='Smith')} 
\end{align*}
\]
Nested Queries (cont’d.)

- Use **tuples** of values in comparisons
  - Place them within parentheses
    - Type-compatible tuples

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

The query selects the SSNs of all employees who **work the same (project, hours) combination** on some project that the employee with SSN=‘123456789’ works on.
Nested Queries (cont’d.)

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

```sql
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > ALL
  ( SELECT Salary
      FROM EMPLOYEE
      WHERE Dno=5 );
```
Nested Queries (cont’d.)

- In general, we can have several levels of nested queries (may cause possible ambiguity among attribute names)
- A reference to an *unqualified attribute* refers to the relation declared in the *innermost nested query*
- To refer an attribute of the relation specified in the outer query, we can specify and refer an *alias* for that relation
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:

```sql
SELECT E.Fname, E.Lname
FROM EMPLOYEE AS E
WHERE E.Ssn IN ( SELECT Essn
  FROM DEPENDENT AS D
  WHERE E.Fname = DDEPENDENT_name
  AND E.Sex = D.Sex );
```
Correlated Nested Queries

- If a condition in the WHERE-clause of a nested query references an attribute of a relation declared in the outer query, the two queries are said to be correlated.

- The result of a correlated nested query is different for each tuple (or combination of tuples) of the relation(s) in the outer query.

- Correlated nested query
  - Evaluated once for each tuple in the outer query.
Correlated Nested Queries

- We can understand a correlated query better by considering *the nested query is evaluated once for each tuple (or combination of tuples) in the outer query.*
  - Q16: for *each* EMPLOYEE tuple, evaluate the nested query, if the SSN value of the EMPLOYEE tuple is *in* the result of the nested query, then select that EMPLOYEE tuple (see slide 14)

- 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;`
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
THE EXISTS FUNCTION (contd.)

- Query 12: Retrieve the name of each employee who has a dependent with the same first name as the employee.

Q12B: SELECT FNAME, LNAME FROM EMPLOYEE WHERE EXISTS (SELECT * FROM DEPENDENT WHERE SSN=ESSN AND FNAME=DEPENDENT_NAME);

- In general, EXISTS(Q) returns TRUE if there is at least one tuple in the result of the nested query, and it returns FALSE otherwise.
Query 6: Retrieve the names of employees who have no dependents.

```
Q6: SELECT FNAME, LNAME
    FROM EMPLOYEE
    WHERE NOT EXISTS (SELECT * 
                      FROM DEPENDENT
                      WHERE SSN = ESSN);
```

In Q6, the correlated nested query retrieves all DEPENDENT tuples related to an EMPLOYEE tuple. If none exist, the EMPLOYEE tuple is selected.

- EXISTS is necessary for the expressive power of SQL
USE of EXISTS

Query 7: List the names of 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)

In general, EXISTS(Q) returns TRUE if there is at least one tuple in the result of the nested query, and it returns FALSE otherwise.
USE OF NOT EXISTS

To achieve the “for all” (universal quantifier- see Ch.8) 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.
Double Negation to accomplish “for all” in SQL

- **Q3B**: 
  ```sql
  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.
Explicit Sets and Renaming of Attributes in SQL

- Can use explicit set of values in WHERE clause
  
  Q17:
  ```sql
  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:
  ```sql
  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';
```

*Note: If a join condition is specified, tuples with NULL values for the join attributes are **not** included in the result.*
Different Types of JOINed Tables in SQL

- Specify different types of join
  - NATURAL JOIN
  - Various types of OUTER JOIN (LEFT, RIGHT, FULL, CROSS)
- 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
  - The result will eliminate the duplicated fields of join attributes
- This construct may be easier to comprehend than mixing together all the select and join condition in the WHERE clause
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.
Joined Tables in SQL and Outer Joins (cont’d.)

- **Examples:**
  
  Q1: `SELECT FNAME, LNAME, ADDRESS
  FROM EMPLOYEE, DEPARTMENT
  WHERE DNAME='Research' AND DNUMBER=DNO;`

- **could be written as: (using inner join)**
  
  Q1: `SELECT FNAME, LNAME, ADDRESS
  FROM (EMPLOYEE JOIN DEPARTMENT
  ON DNUMBER=DNO)
  WHERE DNAME='Research';`

- **or as: (using natural join)**
  
  Q1: `SELECT FNAME, LNAME, ADDRESS
  FROM (EMPLOYEE NATURAL JOIN DEPARTMENT AS DEPT(DNAME, DNO, MSSN, MSDATE)
  WHERE DNAME='Research';`
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
Employee INNER JOIN Department
ON Employee.DepartmentID = Department.DepartmentID

<table>
<thead>
<tr>
<th>Employee table</th>
<th>Department table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LastName</strong></td>
<td><strong>DepartmentID</strong></td>
</tr>
<tr>
<td>Rafferty</td>
<td>31</td>
</tr>
<tr>
<td>Jones</td>
<td>33</td>
</tr>
<tr>
<td>Heisenberg</td>
<td>33</td>
</tr>
<tr>
<td>Robinson</td>
<td>34</td>
</tr>
<tr>
<td>Smith</td>
<td>34</td>
</tr>
<tr>
<td>Williams</td>
<td>NULL</td>
</tr>
</tbody>
</table>

```sql
SELECT *
FROM employee
JOIN department
ON employee.DepartmentID = department.DepartmentID;
```
### Employee NATURAL JOIN Department

```sql
SELECT * 
FROM employee NATURAL JOIN department;
```

<table>
<thead>
<tr>
<th>DepartmentID</th>
<th>Employee.LastName</th>
<th>Department.DepartmentName</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>Smith</td>
<td>Clerical</td>
</tr>
<tr>
<td>33</td>
<td>Jones</td>
<td>Engineering</td>
</tr>
<tr>
<td>34</td>
<td>Robinson</td>
<td>Clerical</td>
</tr>
<tr>
<td>33</td>
<td>Heisenberg</td>
<td>Engineering</td>
</tr>
<tr>
<td>31</td>
<td>Rafferty</td>
<td>Sales</td>
</tr>
</tbody>
</table>
Employee LEFT OUTER JOIN Department
ON Employee.DepartmentID = Department.DepartmentID;

<table>
<thead>
<tr>
<th>Employee table</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastName</td>
</tr>
<tr>
<td>Rafferty</td>
</tr>
<tr>
<td>Jones</td>
</tr>
<tr>
<td>Heisenberg</td>
</tr>
<tr>
<td>Robinson</td>
</tr>
<tr>
<td>Smith</td>
</tr>
<tr>
<td>Williams</td>
</tr>
</tbody>
</table>

```
SELECT *
FROM employee
LEFT OUTER JOIN
department
ON employee.DepartmentID = department.DepartmentID;
```

<table>
<thead>
<tr>
<th>Employee.LastName</th>
<th>Employee.DepartmentID</th>
<th>Department.DepartmentID</th>
<th>Department.DepartmentName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>33</td>
<td></td>
<td>Engineering</td>
</tr>
<tr>
<td>Rafferty</td>
<td>31</td>
<td></td>
<td>Sales</td>
</tr>
<tr>
<td>Robinson</td>
<td>34</td>
<td></td>
<td>Clerical</td>
</tr>
<tr>
<td>Smith</td>
<td>34</td>
<td></td>
<td>Clerical</td>
</tr>
<tr>
<td>Williams</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Heisenberg</td>
<td>33</td>
<td></td>
<td>Engineering</td>
</tr>
</tbody>
</table>
Employee RIGHT OUTER JOIN Department ON Employee.DepartmentID = Department.DepartmentID;

SELECT *
FROM employee RIGHT OUTER JOIN department ON employee.DepartmentID = department.DepartmentID;

<table>
<thead>
<tr>
<th>Employee.LastName</th>
<th>Employee.DepartmentID</th>
<th>Department.DepartmentID</th>
<th>Department.DepartmentName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>34</td>
<td>Clerical</td>
<td>34</td>
</tr>
<tr>
<td>Jones</td>
<td>33</td>
<td>Engineering</td>
<td>33</td>
</tr>
<tr>
<td>Robinson</td>
<td>34</td>
<td>Clerical</td>
<td>34</td>
</tr>
<tr>
<td>Heisenberg</td>
<td>33</td>
<td>Engineering</td>
<td>33</td>
</tr>
<tr>
<td>Rafferty</td>
<td>31</td>
<td>Sales</td>
<td>31</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>Marketing</td>
<td>35</td>
</tr>
</tbody>
</table>
Ex: Only employees who have a supervisor (inner join)

Q8: SELECT E.FNAME, E.LNAME, S.FNAME, S.LNAME
    FROM EMPLOYEE E, S
    WHERE E.SUPERSSN = S.SSN;

If all employees are included (using outer join):

Q8B: SELECT E.FNAME, E.LNAME, S.FNAME, S.LNAME
    FROM (EMPLOYEE E LEFT OUTER JOIN
           EMPLOYEE S ON
           E.SUPERSSN = S.SSN);

SELECT E.LNAME, S.LNAME
FROM EMPLOYEE E, EMPLOYEE S
WHERE E.SUPERSSN += S.SSN
Example: LEFT OUTER JOIN

- Some systems specify outer joins with different syntax
  - +=, =+, and +=+ (left, right, and full outer joins)

```sql
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:**

```sql
SELECT E.Lname, S.Lname
FROM EMPLOYEE E, EMPLOYEE S
WHERE E.Super_ssn + = S.Ssn
```
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:
  - Another example: Q2 could be written as follows; this illustrates multiple joins (nest join) in the joined tables

Q2A:  

```
SELECT Pnumber, Dnum, Lname, Address, Bdate
FROM ((PROJECT JOIN DEPARTMENT ON Dnum=Dnumber) JOIN EMPLOYEE ON Mgr_ssn=Ssn)
WHERE Plocation='Stafford';
```
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:
  - Query 19: Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary among all employees.

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=DMnumber) 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=DMNUMBER AND DNAME=‘Research’;

(*) refers to the rows (tuples)
Aggregate Functions in SQL (cont’d.)

- **Queries 23:** Count the number of distinct salary values in the database

  Q23: `SELECT COUNT (DISTINCT SALARY) FROM EMPLOYEE ;`

- If we write `COUNT(SALARY)` instead of `COUNT(DISTINCT SALARY)`, the duplicate values will **NOT** be eliminated. However, any tuples with NULL for SALARY will not be counted.
The aggregate functions can also be used in the selection conditions involving nested queries

Queries 5: Retrieve the names of all employees who have two or more dependents

Q5: SELECT LNAME, FNAME
    FROM EMPLOYEE
    WHERE (SELECT COUNT(*)
           FROM DEPENDENT
           WHERE SSN=ESSN) >= 2;
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)

```
SELECT month, some(amount>100000)
FROM sales
GROUP BY month;
```
Grouping: The GROUP BY Clause

- **Partition** relation into subsets of tuples
  - Based on **grouping attribute(s)**
    - Each subgroup of tuples consists of the set of tuples that have the same value for the grouping attribute(s)
  - Apply function to each such group independently

- **GROUP BY** clause
  - Specifies **grouping attributes**, which **must** also appear in the **SELECT**-clause

- **COUNT (**) counts** the number of rows in the group

- If **NULLs** exist in grouping attribute
  - **Separate group** created for all tuples with a NULL value in grouping attribute
Grouping: The GROUP BY and HAVING Clauses (cont’d.)

- Query 24: For each department, retrieve the department number, the number of employees in the department, and their average salary.

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

- In Q24, the EMPLOYEE tuples are divided into groups-
  - Each group having the same value for the grouping attribute DNO
- The COUNT and AVG functions are applied to each such group of tuples separately
- The SELECT-clause includes only the grouping attribute and the functions to be applied on each group of tuples

- A join condition can be used in conjunction with grouping (Q25).
  - In this case, the grouping and functions are applied after the joining of the two relations.
Examples of GROUP BY

<table>
<thead>
<tr>
<th>Fname</th>
<th>Minit</th>
<th>Lname</th>
<th>Ssn</th>
<th>Salary</th>
<th>Super_ssn</th>
<th>Dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>B</td>
<td>Smith</td>
<td>123456789</td>
<td>30000</td>
<td>333445555</td>
<td>5</td>
</tr>
<tr>
<td>Franklin</td>
<td>T</td>
<td>Wong</td>
<td>333445555</td>
<td>40000</td>
<td>888665555</td>
<td>5</td>
</tr>
<tr>
<td>Ramesh</td>
<td>K</td>
<td>Narayan</td>
<td>666884444</td>
<td>38000</td>
<td>333445555</td>
<td>5</td>
</tr>
<tr>
<td>Joyce</td>
<td>A</td>
<td>English</td>
<td>453453453</td>
<td>25000</td>
<td>333445555</td>
<td>5</td>
</tr>
<tr>
<td>Alicia</td>
<td>J</td>
<td>Zelaya</td>
<td>999887777</td>
<td>25000</td>
<td>987654321</td>
<td>4</td>
</tr>
<tr>
<td>Jennifer</td>
<td>S</td>
<td>Wallace</td>
<td>987654321</td>
<td>43000</td>
<td>888665555</td>
<td>4</td>
</tr>
<tr>
<td>Ahmad</td>
<td>V</td>
<td>Jabbar</td>
<td>987987987</td>
<td>25000</td>
<td>987654321</td>
<td>4</td>
</tr>
<tr>
<td>James</td>
<td>E</td>
<td>Bong</td>
<td>888665555</td>
<td>55000</td>
<td>NULL</td>
<td>1</td>
</tr>
</tbody>
</table>

Grouping EMPLOYEE tuples by the value of Dno

<table>
<thead>
<tr>
<th>Dno</th>
<th>Count (*)</th>
<th>Avg (Salary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td>33250</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>31000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>55000</td>
</tr>
</tbody>
</table>

If NULL exist in the grouping attributes, then a separate group is created for all tuples with a NULL value in the grouping attributes.
Query 25: For each project, retrieve the project number, project name, and the number of employees who work on that project.

Q25: 

\[
\begin{align*}
\text{SELECT} & \quad \text{Pnumber, Pname, } \text{COUNT} (*) \\
\text{FROM} & \quad \text{PROJECT, WORKS_ON} \\
\text{WHERE} & \quad \text{Pnumber=Pno} \\
\text{GROUP BY} & \quad \text{Pnumber, Pname;} 
\end{align*}
\]

The grouping and functions are applied after the joining of the two relations.
Grouping: The GROUP BY and HAVING Clauses (cont’d.)

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

- The **HAVING**-clause is used for specifying a selection condition on groups (rather than on individual tuples):
  - **WHERE** clause applies to **the tuples**
  - **HAVING** clause applies to **the whole group**
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;`
Results of GROUP BY and HAVING.

Q26.

<table>
<thead>
<tr>
<th>Pname</th>
<th>Pnumber</th>
<th>...</th>
<th>Essn</th>
<th>Pno</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductX</td>
<td>1</td>
<td></td>
<td>123456789</td>
<td>1</td>
<td>32.5</td>
</tr>
<tr>
<td>ProductX</td>
<td>1</td>
<td></td>
<td>453453453</td>
<td>1</td>
<td>20.0</td>
</tr>
<tr>
<td>ProductY</td>
<td>2</td>
<td></td>
<td>123456789</td>
<td>2</td>
<td>7.5</td>
</tr>
<tr>
<td>ProductY</td>
<td>2</td>
<td></td>
<td>453453453</td>
<td>2</td>
<td>20.0</td>
</tr>
<tr>
<td>ProductY</td>
<td>2</td>
<td></td>
<td>333445555</td>
<td>2</td>
<td>10.0</td>
</tr>
<tr>
<td>ProductZ</td>
<td>3</td>
<td></td>
<td>666884444</td>
<td>3</td>
<td>40.0</td>
</tr>
<tr>
<td>ProductZ</td>
<td>3</td>
<td></td>
<td>333445555</td>
<td>3</td>
<td>10.0</td>
</tr>
<tr>
<td>Computerization</td>
<td>10</td>
<td>...</td>
<td>333445555</td>
<td>10</td>
<td>10.0</td>
</tr>
<tr>
<td>Computerization</td>
<td>10</td>
<td></td>
<td>999887777</td>
<td>10</td>
<td>10.0</td>
</tr>
<tr>
<td>Computerization</td>
<td>10</td>
<td></td>
<td>987987987</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>Reorganization</td>
<td>20</td>
<td></td>
<td>333445555</td>
<td>20</td>
<td>10.0</td>
</tr>
<tr>
<td>Reorganization</td>
<td>20</td>
<td></td>
<td>987654321</td>
<td>20</td>
<td>15.0</td>
</tr>
<tr>
<td>Reorganization</td>
<td>20</td>
<td></td>
<td>888665555</td>
<td>20</td>
<td>NULL</td>
</tr>
<tr>
<td>Newbenefits</td>
<td>30</td>
<td></td>
<td>987987987</td>
<td>30</td>
<td>5.0</td>
</tr>
<tr>
<td>Newbenefits</td>
<td>30</td>
<td></td>
<td>987654321</td>
<td>30</td>
<td>20.0</td>
</tr>
<tr>
<td>Newbenefits</td>
<td>30</td>
<td></td>
<td>999887777</td>
<td>30</td>
<td>30.0</td>
</tr>
</tbody>
</table>

After applying the WHERE clause but before applying HAVING.
Results of GROUP BY and HAVING.

Q26.

<table>
<thead>
<tr>
<th>Pname</th>
<th>Pnumber</th>
<th>...</th>
<th>Essn</th>
<th>Pno</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductY</td>
<td>2</td>
<td></td>
<td>123456789</td>
<td>2</td>
<td>7.5</td>
</tr>
<tr>
<td>ProductY</td>
<td>2</td>
<td></td>
<td>453453453</td>
<td>2</td>
<td>20.0</td>
</tr>
<tr>
<td>ProductY</td>
<td>2</td>
<td></td>
<td>333445555</td>
<td>2</td>
<td>10.0</td>
</tr>
<tr>
<td>Computerization</td>
<td>10</td>
<td></td>
<td>333445555</td>
<td>10</td>
<td>10.0</td>
</tr>
<tr>
<td>Computerization</td>
<td>10</td>
<td></td>
<td>999887777</td>
<td>10</td>
<td>10.0</td>
</tr>
<tr>
<td>Computerization</td>
<td>10</td>
<td></td>
<td>987987987</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>Reorganization</td>
<td>20</td>
<td></td>
<td>333445555</td>
<td>20</td>
<td>10.0</td>
</tr>
<tr>
<td>Reorganization</td>
<td>20</td>
<td></td>
<td>987654321</td>
<td>20</td>
<td>15.0</td>
</tr>
<tr>
<td>Reorganization</td>
<td>20</td>
<td></td>
<td>888665555</td>
<td>20</td>
<td>NULL</td>
</tr>
<tr>
<td>Newbenefits</td>
<td>30</td>
<td></td>
<td>987987987</td>
<td>30</td>
<td>5.0</td>
</tr>
<tr>
<td>Newbenefits</td>
<td>30</td>
<td></td>
<td>987654321</td>
<td>30</td>
<td>20.0</td>
</tr>
<tr>
<td>Newbenefits</td>
<td>30</td>
<td></td>
<td>999887777</td>
<td>30</td>
<td>30.0</td>
</tr>
</tbody>
</table>

After applying the HAVING clause condition
Query 27: For each project, retrieve the project number, project name, and the number of employees from department 5 who work on the project.

Q27: 
```
SELECT PNUMBER, PNAME, COUNT(*)
FROM PROJECT, WORKS_ON, EMPLOYEE
WHERE PNUMBER=PNO AND SSN=ESSN
   AND DNO=5
GROUP BY PNUMBER, PNAME;
```

Restrict the tuples in the relation to those satisfy the WHERE condition – the employees work in department number 5.
Combining the WHERE and the HAVING Clause

- Consider the query Q28: 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;
```

since it will select only departments that have more than five employees who each earn more than $40,000.
Grouping: The GROUP BY and HAVING Clauses (cont’d.)

- Correct Specification of the Query:

```sql
Q28: SELECT DNUMBER, COUNT(*)
    FROM DEPARTMENT, EMPLOYEE
    WHERE DNUMBER=DNO AND SALARY>40000
    AND DNO IN (SELECT DNO
                 FROM EMPLOYEE
                 GROUP BY DNO
                 HAVING COUNT(*) > 5)
    GROUP BY DNUMBER;
```

- The WHERE clause is executed first, to select individual tuples, the HAVING clause is applied later
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 at a step-by-step query.
Example of WITH

See an alternate approach to doing Q28:

Q28':

```sql
WITH BIGDEPTS (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;
```

Copyright © 2017 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
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
    ELSE Salary + 0
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)): 
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.
EXPANDED Block Structure of SQL Queries

```sql
SELECT <attribute and function list>
FROM <table list>
[ WHERE <condition> ]
[ GROUP BY <grouping attribute(s)> ]
[ HAVING <group condition> ]
[ ORDER BY <attribute list> ];
```
Summary of SQL Queries (contd.)

- The SELECT-clause lists the attributes or functions to be retrieved.
- The FROM-clause specifies all relations (or aliases) needed in the query but not those needed in nested queries.
- The WHERE-clause specifies the conditions for selection and join of tuples from the relations specified in the FROM-clause.
- GROUP BY specifies grouping attributes.
- HAVING specifies a condition for selection of groups.
- ORDER BY specifies an order for displaying the result of a query.
  - A query is evaluated by first applying the WHERE-clause, then GROUP BY and HAVING, and finally the SELECT-clause.
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
  - Components include:
    - a constraint name,
    - followed by `CHECK`,
    - followed by a condition
    - Must hold true on every database state for the assertion to be satisfied

- **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*

```sql
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)**
    - Such as an insert, deleted, or update operation
  - **Condition**
  - **Action**
    - To be taken when the condition is satisfied
    - Usually a sequence of SQL statements, **stored procedure**, database transaction, or an external program
USE OF TRIGGERS

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

A trigger to compare an employee’s salary to his/her supervisor during insert or update operations:

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)
FIGURE 24.2
Specifying active rules as triggers in Oracle notation.
(a) Triggers for automatically maintaining the consistency of TOTAL_SAL of DEPARTMENT.
(b) Trigger for comparing an employee’s salary with that of his or her supervisor.
FIGURE 24.3
A syntax summary for specifying triggers in the Oracle system (main options only).

<trigger>::= CREATE TRIGGER <trigger name>
   (AFTER | BEFORE ) <triggering events> ON <table name>
   [ FOR EACH ROW ]
   [ WHEN <condition> ]
   <trigger actions> ;

<triggering events>::=<trigger event> {OR <trigger event> }
<trigger event>::=INSERT | DELETE | UPDATE [ OF <column name> {, <column name>} ]
<trigger action>::=<PL/SQL block>
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
- Allows for **limited update** operations
  - Since the table **may not physically be stored**
- Allows **full query** operations
- A convenience for expressing certain operations
  - 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
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=Esn 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

  \[ QV1: \texttt{SELECT FNAME, LNAME} \]
  \[ \texttt{FROM WORKS\_ON1} \]
  \[ \texttt{WHERE PNAME='ProjectX'} ; \]

- **View is always up-to-date**
  - Responsibility of the DBMS and not the user
  - The view is not realized or materialized at the time of **view definition** but rather at the time when we **specify a query** on the view

- **DROP VIEW** command
  - Dispose of a view
  - **DROP WORKS\_ON1**;
Complex problem of efficiently implementing a view for querying

Strategy1: Query modification approach
- Compute the view as and when needed. Do not store permanently
- Modify view query into a query on underlying base tables
  - E.g., QV1 will be modified as

```
SELECT  FNAME, LNAME
FROM    EMPLOYEE, PROJECT, WORK_ON
WHERE   SSN=ESSN AND PNO=PNUMBER  AND PNAME='ProjectX';
```

Disadvantage: inefficient for views defined via complex queries that are time-consuming to execute
View Materialization

- **Strategy 2: View materialization**
  - Physically create a temporary view table when the view is first queried
  - Keep that table on the assumption that other queries on the view will follow
  - Requires efficient strategy for automatically updating the view table when the base tables are updated

- **Incremental update strategy for materialized views**
  - 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 Materialization (contd.)

- Multiple ways to handle materialization:
  - **Immediate update** strategy updates a view as soon as the base tables are changed.
  - **Lazy update** strategy updates the view when needed by a view query.
  - **Periodic update** strategy updates the view periodically (in the latter strategy, a view query may get a result that is not up-to-date). This is commonly used in Banks, Retail store operations, etc.
View Update

- Updating of views is complicated and can be ambiguous
- Update on a view defined on a single table without any aggregate functions
  - Can be mapped to an update on underlying base table - possible if the primary key is preserved in the view
- Update not permitted on aggregate views. E.g.,

  UV2: 
  
  ```sql
  UPDATE DEPT_INFO
  SET Total_sal=100000
  WHERE Dname='Research';
  ```

  cannot be processed because Total_sal is a computed value in the view definition
Un-updateable Views

- A view with single defining table is **updatable** if the view attributes contain the **primary key** of the base relation, as well as attributes with the **NOT NULL** constraint that do **not have** default values specified.

- Views defined on **multiple tables using joins** are generally not updateable.
  - View involving **joins**
    - An update *may* map to an update on the underlying base relations *in multiple ways*
    - Often not possible for DBMS to determine which of the updates is intended.

- Views defined using **groups** and **aggregate functions** are **not updateable**.
**View Update and Inline Views**

- **Clause** `WITH CHECK OPTION`
  - Must be added at the end of the view definition if a view is to be updated to make sure that tuples being updated stay in the view
  - To allow check for updatability and to plan for an execution strategy

- **In-line view**
  - The view is defined within the query itself
  - Defined in the `FROM` clause of an SQL query (e.g., we saw its used in the WITH example)
Example of Inline View

- **Simplify a complex query** by removing join operations and condensing several separate queries into a single query

```sql
SELECT *
FROM (SELECT dept_no, count(*) emp_count
      FROM employees
      GROUP BY dept_no) employees, department
WHERE department.dept_no = employees.dept_no;
```
Views as authorization mechanism

- SQL query authorization statements (GRANT and REVOKE) are described in detail in Chapter 30
- Views can be used to hide certain attributes or tuples from unauthorized users
- E.g., For a user who is only allowed to see employee information for those who work for department 5, he may only access the view `DEPT5EMP`:

```
CREATE VIEW DEPT5EMP AS
SELECT * FROM EMPLOYEE WHERE Dno = 5;
```
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
  - Used to alter a schema by adding or dropping tables, attributes, constraints, and other schema elements
  - Certain checks must be done by the DBMS to ensure that the changes do not affect the rest of the database and make it consistent
The DROP Command

- **DROP** command
  - Used to drop named schema elements, such as tables, domains, or constraint
  - Remove its definition too
- Drop behavior options:
  - **CASCADE** and **RESTRICT**
- Example:
  - `DROP SCHEMA COMPANY CASCADE;`
  - This removes the schema and all its elements including tables, views, constraints, etc.
The DROP Command (cont’d.)

- DROP SCHEMA COMPANY CASCADE;
  - Remove COMPANY database schema and all its tables, domains, and other elements.

- DROP SCHEMA COMPANY RESTRICT;
  - Drop schema COMPANY only if it has no elements in it.

- DROP TABLE DEPENDENT CASCADE;
  - All constraints and views that reference the table are dropped automatically from the schema, along with the table itself.

- DROP TABLE DEPENDENT RESTRICT;
  - Drop the table if it is not referenced in any constraints or views
The ALTER table command

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

- Used to add an attribute to one of the base relations
  - The new attribute will have NULLs in all the tuples of the relation right after the command is executed; hence, the NOT NULL constraint is not allowed for such an attribute

- **Example:**
  ```sql
  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 \texttt{CASCADE} or \texttt{RESTRICT}
  - \texttt{CASCADE} would drop the column from views etc.
    - all constraints and views referencing the column are dropped automatically
  - \texttt{RESTRICT} is possible if no views refer to it.
    - the command succeeds if no views or constraints reference to the column

   \texttt{ALTER TABLE COMPANY.EMPLOYEE DROP COLUMN Address CASCADE;}

- Default values can be dropped and altered:
  \texttt{ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn DROP DEFAULT;}
  \texttt{ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn SET DEFAULT '333445555';}
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>
<thead>
<tr>
<th>SQL Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE FROM &lt;table name&gt;</td>
</tr>
<tr>
<td>[ WHERE &lt;selection condition&gt;]</td>
</tr>
<tr>
<td>UPDATE &lt;table name&gt;</td>
</tr>
<tr>
<td>SET &lt;column name&gt; = &lt;value expression&gt; { , &lt;column name&gt; = &lt;value expression&gt; }</td>
</tr>
<tr>
<td>[ WHERE &lt;selection condition&gt;]</td>
</tr>
<tr>
<td>CREATE [ UNIQUE] INDEX &lt;index name&gt;</td>
</tr>
<tr>
<td>ON &lt;table name&gt; ( &lt;column name&gt; [ &lt;order&gt; ] { , &lt;column name&gt; [ &lt;order&gt; ] } )</td>
</tr>
<tr>
<td>[ CLUSTER ]</td>
</tr>
<tr>
<td>DROP INDEX &lt;index name&gt;</td>
</tr>
<tr>
<td>CREATE VIEW &lt;view name&gt; [ ( &lt;column name&gt; { , &lt;column name&gt; } ) ]</td>
</tr>
<tr>
<td>AS &lt;select statement&gt;</td>
</tr>
<tr>
<td>DROP VIEW &lt;view name&gt;</td>
</tr>
</tbody>
</table>

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.