CHAPTER 7

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

Note: Slides, content, web links and end chapter questions are prepared from Pearson textbook and other Internet resources.
Additionally many information added from another Pearson book of Kroenke, Auer and others of 2019 – 15th edition.

Topics of Discussion

A. More Complex SQL Retrieval Queries
B. Specifying Semantic Constraints as Assertions and Actions as Triggers
C. Views (Virtual Tables) in SQL
D. Schema Modification in SQL

More Complex SQL Retrieval Queries

- This chapter describes more advanced features of the SQL language for relational databases.
- 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
  - This chapter is a continuation of Chapter 6, and it is identified as self learning chapter.

Comparisons Involving NULL and 3-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
- SQL allows queries that check whether an attribute value is NULL
  - IS NULL or IS NOT NULL

Comparisons Involving NULL and 3-Valued Logic

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

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

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

Nested Queries, Tuples, 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*

```
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, Tuples, Set/Multiset Comparisons

- 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' );
```

- 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

Nested Queries, Tuples, Set/Multiset Comparisons

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

- 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 );
```

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

```
Q7: SELECT Fname, Lname  FROM Employee
    WHERE EXISTS (SELECT *
                  FROM DEPENDENT
                  WHERE Ssn= Essn)
          AND EXISTS (SELECT   *  FROM Department
                      WHERE Ssn= Mgr_Ssn)
```

The EXISTS and UNIQUE Functions in SQL for correlating queries

- To achieve the “for all” effect, we use double negation following way in the 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:**

```
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:    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 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

Example: LEFT OUTER JOIN

- In the SQL standard, by explicitly specifying the keyword OUTER JOIN in a joined table, as illustrated in

  Q8B: 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: In some systems, to specify outer joins by using the comparison operators + =, = +, and + = + for left, right, and full outer join, respectively. For ex, Q8B we could write the query Q8C as follows:

  Q8C:

  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:

```
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:

```
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

- 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';
```

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)

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;
```

The GROUP BY and HAVING Clauses

- **`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

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 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
- See an alternate approach to doing Q28:
- **Q28':**

```
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;
```

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
- 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
```

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

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 Constraints as Assertions and Actions as Triggers

- **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 ) );
```

Introduction to Triggers in SQL

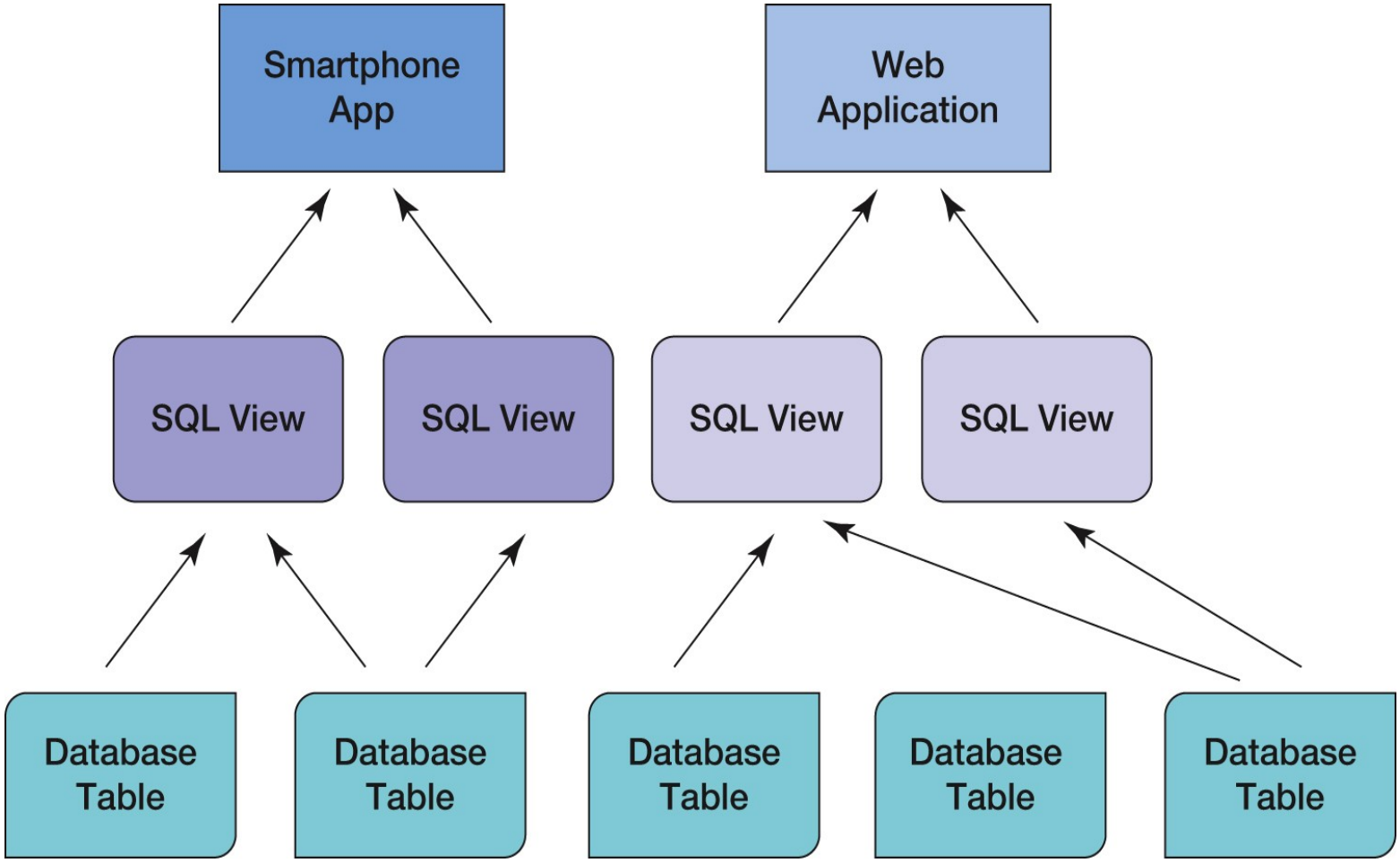
- `CREATE TRIGGER` statement
  - Used to monitor the database
- Typical trigger has three components which make it a rule for an “active database “ (more on active databases in section 26.1) :
  - **1) Event(s) 2) Condition 3) Action**
- Complete code of trigger explained in Ch-10.
  An Example with standard 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
  - An SQL view is a virtual table that is constructed from other tables or views.
  - It has no data of its own but obtains data from tables or other views.
  - SELECT statements are used to define views:
- A view definition may not include an ORDER BY clause
- SQL views are a subset of the external views:
- They can be used only for external views that involve one multivalued path through the schema

SQL Views as the Basis for Application Data



Uses of SQL Views

- **`Uses of SQL Views`**
- Hide columns or rows.
- Display results of computations.
- Hide complicated SQL syntax.
- Layer built-in functions.
- Provide level of isolation between table data and users' view of data.
- Assign different processing permissions to different views of the same table.
- Assign different triggers to different views of the same table.

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;
```

SQL VIEW Example (1 of 2)

- Create the 'EmployeeNameView' on the EMPLOYEE table that displays the employee's FirstName and LastName.

```
USE [COMPANY]
GO
/****** Object:  View [dbo].[EmployeeNameView]    Script Date: 9/19/2020 10:19:08 PM ******/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO
/* SQL View for showing correct heading for name of employee */
CREATE view [dbo].[EmployeeNameView] AS
select Fname AS Emp_First_Name, Lname As Emp_Last_Name
from employee;
GO
```

SQL VIEW Example (2 of 2)

- To see the results, use an SQL SELECT statement with the view name as the table name in the FROM clause as shown below.

```
use company
go
select * from EmployeeNameView
order by Emp_Last_Name, Emp_First_Name;
```

36 %

Results | Messages

	Emp_First_Name	Emp_Last_Name
1	Red	Bacher
2	Nandita	Ball
3	Bonnie	Bays
4	Bob	Bender
5	Alec	Best
6	James	Borg
7	James	Borg
8	Tom	Brand
9	Chris	Carter
10	Jeff	Chase

The SQL ALTER VIEW Statement

- You can use the **ALTER VIEW Statement** to change an SQL view after it has been created.

```
/* *** EXAMPLE CODE – DO NOT RUN *** */
ALTER VIEW EmployeeNameView AS
    SELECT  FName AS Employee_FirstName,
            LName AS Employee_LastName,
    FROM    Employee;
```

- In the Oracle Database or MySQL 5.7, use the SQL CREATE OR REPLACE VIEW statement.
  - This allows creation and modification of SQL VIEW code

View Update

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

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

View Update and Inline Views

- View involving joins
  - Often not possible for DBMS to determine which of the updates is intended
- 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
- **In-line view**
  - Defined in the `FROM` clause of an SQL query (e.g., we saw its used in the WITH example)

Guidelines for Updating Views

- Updatable Views

- View based on a single table with no computed columns and all non-null columns
- present in the view.
- View based on any number of tables, with or without computed columns, and
- INSTEAD OF trigger defined for the view.

- Possibly Updatable Views

- Based on a single table, primary key in view, some required columns missing from
- view, update and delete may be allowed. Insert is not allowed.
- Based on multiple tables, updates may be allowed on the most subordinate table in
- the view if rows of that table can be uniquely identified.

Specification of Views in SQ

- 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

View Implementation, View Update, and Inline Views

- 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
  - 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

View Materialization

- 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.

Views as authorization mechanism

- SQL query authorization statements (GRANT and REVOKE) are described in detail in Ch -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
- `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.

Schema Change Statements in SQL

- **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);`
- Change constraints specified on a table
  - Add or drop a named constraint

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

Schema Change Statements in SQL

- 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';

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

Summary of SQL Syntax

Table 7.2 Summary of SQL Syntax

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

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

End Chapter Questions

- 7.1. Describe the six clauses in the syntax of an SQL retrieval query. Show what type of constructs can be specified in each of the six clauses.
- 7.4. Discuss how each of the following constructs is used in SQL, and discuss the various options for each construct. Specify what each construct is useful for.
  - Nested queries
  - Joined tables and outer joins
  - Aggregate functions and grouping
  - Assertions and how they differ from triggers
  - The SQL WITH clause
  - SQL CASE construct
  - Views and their updatability
- 7.5. Explain Schema change commands, with examples.