COMPSCI 351
Fundamentals of Database Systems

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

Outline

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

Comparisons Involving NULL and Three-Valued Logic

- Meanings of `NULL`
  - **Unknown value**
    - Value exists but is not known, or it is not known whether or not the value exists
  - **Unavailable or withheld value**
    - Value exists but is purposely withheld
  - **Not applicable attribute**
    - The attribute does not apply to this tuple, or is undefined for this tuple
- SQL does not distinguish among different meanings of NULL.

Comparisons Involving NULL and Three-Valued Logic

- When a NULL value is involved in a comparison, the result is considered to be UNKNOWN.
- SQL uses a three-valued logic:
  - `TRUE`, `FALSE`, and `UNKNOWN` (like Maybe)

Table 7.1 Logical Connectives in Three-Valued Logic

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

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

- SQL allows queries that check whether an attribute value is `NULL`
- Rather than using = or <> to compare an attribute value to NULL, SQL uses the comparison operator:
  - `IS NULL` or `IS NOT NULL`
  - Each individual `NULL` value is considered to be distinct from every other `NULL` value

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

- Some queries require that existing values in the database be fetched and then used in a comparison condition.
  - can be formulated using nested queries
  - the other query is called the outer query.

```
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

- SQL allows the use of tuples of values in comparison by:
  - Place them within parentheses

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

- In addition to the IN operator, a number of other comparison operators can be used to compare a single value v to a set (or multiset) of values 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`

Nested Queries

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

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

Nested Queries

- In the case of several levels of nested queries, we may have possible ambiguity among the attribute names (if attributes of the same name exist).
- 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

- Two queries are **correlated:**
  - A condition in the WHERE clause of a nested query references some attributes of a relation declared in the outer query.
  - Is valuated once for each tuple in the outer query
- 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

- `EXIST` and `UNIQUE` are Boolean functions that return `TRUE` or `FALSE`, hence can be used in the `WHERE` clause
- `EXISTS` and `NOT EXISTS` functions
  - Check whether the result of a correlated nested query is empty or not.
- SQL function `UNIQUE(Q)`
  - Returns `TRUE` if there are no duplicate tuples in the result of query Q

USE of EXISTS

- `EXIST(Q)` returns `TRUE` if there is at least one tuple in the results of the nested query Q, and return `FALSE` otherwise.

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 );
```

USE OF NOT EXISTS

- To achieve the "for all" (universal quantifier) effect, we use *double negation* this way in SQL:
- Query: List first and last name of employees who work on ALL projects controlled by Deparment 5.

```
Q3A:  SELECT   Fname, Lname
      FROM     EMPLOYEE
      WHERE    NOT EXISTS ( ( SELECT   Pnumber
                              FROM     PROJECT
                              WHERE    Dnum = 5)
                              EXCEPT   ( SELECT   Pno
                                         FROM     WORKS_ON
                                         WHERE    Ssn = Essn) );
```

Explicit Sets and Renaming of Attributes in SQL

- can use explicit set of values in WHERE clause,
  - e.g., retrieve the SSN of all employees who work on project numbers 1, 2, or 3.

```
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 (or joined relation)**
  - 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, which has all the attributes of the first table EMPLOYEE followed by all the attributes of the second table DEPARTMENT.

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

Different Types of JOINed Tables in SQL

- Allow users to specify different types of join, such as
  - 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
  - Each such pair of attributes is included only once in the resulting relation.

NATURAL JOIN

- If the names of the join attributes are not the same in the base relations, it is possible to **rename** attributes so that they match, and then to apply the 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, added 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

- LEFT OUTER JOIN

```
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);
```

- In some DBMS, a different syntax was used to specify outer joins by using the comparison operators, e.g.,
  - In Oracle, '+=', '=+' and '+=+' symbols are used for left, right and full outer join respectively.

```
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 results of the LEFT and RIGHT OUTER JOIN
- Can nest JOIN specifications for a multiway join, e.g.,

Query 2. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

```
Q2:   SELECT   Pnumber, Dnum, Lname, Address, Bdate
      FROM     PROJECT, DEPARTMENT, EMPLOYEE
      WHERE    Dnum = Dnumber AND Mgr_ssn = Ssn AND
               Plocation = 'Stafford'

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.

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

- The result can be presented with new names:

We could use AS to rename the column names in the resulting single-row table; for example, as in Q19A.

```
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 in SQL

- Use the `COUNT` function to count values in a column (attribute) rather than tuples, e.g.,

Query 23. Count the number of distinct salary values in the database.

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

- We can specify a correlated nested query with an aggregate function, e.g., to retrieve the names of all employees who have two or more dependents (Query 5)

```
Q5:     SELECT    Lname, Fname
        FROM      EMPLOYEE
        WHERE     ( SELECT    COUNT (*)
                    FROM      DEPENDENT
                    WHERE     Ssn = Essn ) >= 2;
```

Grouping: The GROUP BY Clause

- **Partition** the relation into nonoverlapping subsets (or groups) of tuples
  - Each group (partition) consists of the tuples that have the same value of some attributes, called **grouping attribute(s)**
  - Apply function to each such group independently to produce summary information about each group
- **`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

- **`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;
```

  - `WHERE` clause is executed first to select individual or joined tuples
  - `HAVING` clause is applied later to select individual groups of tuples.

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

Other SQL Construct: WITH

- The `WITH` clause allows a user to define a table that will only be used in a particular query (*may not be available in all SQL based DBMS 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
- This temporary "View" (or table) will be discarded after the query is executed.

Example of WITH

- See an alternate approach to performing Q28:

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′:   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;
```

- We define in the `WITH` clause a temporary table `BIGDEPTS`, and use this table in the subsequent query.

Other SQL Construct: 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
  - For example, 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
- A example of a **recursive operation** is to retrieve all supervisees of a supervisory employee e at all levels, i.e.,
  - 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 (SupSn, EmpSsn) AS
                ( SELECT    SupervisorSsn, Ssn
                  FROM      EMPLOYEE
                            UNION
                  SELECT    S.EmpSsn, E.Ssn
                  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.

Summary of 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>];

- `SELECT` clause lists the attributes or functions to be retrieved
- `FROM` clause specifies all the tables needed
- `WHERE` clause specifies the conditions for selection
- `GRROUP BY` specifies grouping attributes
- `HAVING` specifies a condition on the grouping
- `ORDER BY` specifies an order for displaying the result

Specifying Constraints as Assertions and Actions as Triggers

- **`CREATE ASSERTION`**
  - Specify additional types of constraints that are outside scope of the built-in relational model constraints (i.e., *primary and unique keys, entity integrity, and referential integrity*).
- **`CREATE TRIGGER`**
  - Specify automatic actions that database system will perform when certain events and conditions occur
  - This type of functionality is generally referred as ***active databases***.

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
  - E.g., a constraint specifies that the salary of an employee must not be greater than that of the manager in the same department

```
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**
  - Specify the type of action to be taken when certain events occur and when certain conditions are satisfied.
    - E.g., a manager may want to be informed if an employee's travel expenses exceed a certain limit in the database state
  - Used to monitor the database, and implement such actions in DBMS
- Typical trigger has three components which make it a rule for an "active database" :
  - **Event(s)**: usually database update operations
  - **Condition**: determines whether rule actions should be executed
  - **Action**: to be taken, usually SQL statements or transactions, etc.

Example use of TRIGGERs

- Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database.
  - Several events can trigger this rule: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor.
  - Suppose that the action to take would be to call an external stored procedure SALARY_VIOLATION,5 which will notify the supervisor.

```
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 (can be base tables or previously defined views)
  - Does not necessary exist in physical form, is considered to be a **virtual table** (in contrast to base tables)
  - As a way of specifying a table that we need to reference frequently, even though it may not exist physically.
- 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

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;
```

- **`DROP VIEW`** command
  - Dispose of a view

View Implementation, View Update, and Inline Views

- Complex problem of efficiently implementing a view for querying
- **Strategy 1: 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 or created
  - 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.

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)

Views as authorization mechanism

- SQL query authorization statements (GRANT and REVOKE) are described in later topics.
- 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 or she may only access the view DEPT5EMP:

```
CREATE VIEW   DEPT5EMP  AS
SELECT        *
FROM          EMPLOYEE
WHERE         Dno = 5;
```

Schema Change Statements in SQL

- **Schema evolution commands**
  - Can be used to alter a schema by adding or dropping tables, attributes, constraints and other schema elements.
  - DBA may want to change the schema while the database is operational
  - Does not require recompilation of the database schema
  - Certain checks must be performed by the DBMS to ensure that the changes do not affect the rest of the database and cause inconsistencies.

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.
  - `DROP TABLE DEPENDENT RESTRICT;`
  - Table is dropped only if not referenced in any constraints

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);`
- We must still enter a value for the new attribute Job for each individual EMPLOYEE tuple.
  - By using the default clause or the UPDATE command

Dropping Columns, Default Values

- Drop a column must choose either `CASCADE` or `RESTRICT`
  - `CASCADE`: all constraints and views that reference the column are dropped automatically
  - `RESTRICT`: the drop is successful only when no views or constraints (or other schema elements) reference the column

```
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';
```

Adding and Dropping Constraints

- We can change the constraints specified on a table by
  - Add or drop a named constraint
  - For example, to drop the constraint named EMPSUPERFK from the EMPLOYEE table

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

- After removing the constraint, we can redefine a replacement constraint using the ADD CONSTRAINT keyword in the ALTER TABLE statement, e.g.,

```
ALTER TABLE COMPANY.EMPLOYEE
ADD CONSTRAINT EMPSUPERFK
  FOREIGN KEY(Super_ssn) REFERENCES EMPLOYEE(Ssn)
                ON DELETE SET NULL  ON UPDATE CASCADE;
```

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.

- Notes on the BNF notation:
  - <...> for nonterminal symbols, and [...] for optional parts;
  - {...} for repetitions, and (...|...|...) for alternatives.

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.