Chapter 6.2

More SQL: Complex Queries, Views, and Schema Modification

Addison-Wesley
is an imprint of
PEARSON

Handling NULLs in SQL

- SQL allows queries that check if an attribute is **NULL** (missing or undefined or not applicable)
- SQL uses **IS** or **IS NOT** to compare an attribute to NULL because it considers each NULL value distinct from other NULL values, so *equality comparison is not appropriate*.
- For example, to retrieve first name and last of name of all employees who do not have supervisors, we write

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

3-valued Logic in SQL

- Standard 2-valued logic assumes a condition can evaluate to either TRUE or FALSE
- With NULLs a condition can evaluate to UNKNOWN, leading to 3-valued logic
- Example: Consider a condition EMPLOYEE.DNO = 5; this evaluates for individual tuples in EMPLOYEE as follows:
  - TRUE for tuples with DNO=5
  - UNKNOWN for tuples where DNO is NULL
  - FALSE for other tuples in EMPLOYEE
- Combining individual conditions using AND, OR, NOT logical connectives must consider UNKNOWN in addition to TRUE and FALSE

Table 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

Nesting of Queries in SQL

- A complete SELECT ... query, called a *nested query*, can be specified within the WHERE-clause of *another query*
  - The other query is called the *outer query*
  - Many of the previous queries can be specified in an alternative form using nesting
- For example, to retrieve the name and address of all employees who work for the 'Research' department, we write

```
SELECT    Fname, Lname, Address
FROM      EMPLOYEE
WHERE     Dno IN  (SELECT  Dnumber
                   FROM     DEPARTMENT
                   WHERE    Dname='Research' ) ;
```

Nesting of Queries (cont.)

- In this query, the *nested query* selects the Dnumber of the 'Research' department and the *outer query* select an EMPLOYEE tuple if its Dno value is in the result of the nested query.
- The comparison operator **IN** compares a value v with a set (or multi-set) of values V, and evaluates to TRUE if v is one of the elements in V
- In general, can have several levels of nested queries

Nesting of Queries (cont.)

- SQL allows the use of tuples of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:

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

  This query will select the Essns of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose Ssn = '123456789') works on.

Nesting of Queries (cont.)

- In addition to the IN operator, a number of other comparison operators can also be used
- The = 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. The two keywords ANY and SOME have the same effect. Other operators that can be combined with ANY (or SOME) include >, >=, <, <=, and <>.
- The keyword ALL can also be combined with each of these operators. For example, the comparison condition (v > ALL V) returns TRUE if the value v is greater than all the values in the set (or multiset) V.

Nesting of Queries (cont.)

- The query below returns the names of employees whose salary is greater than the salary of all the employees in department 5.

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

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) the outer query
- Example: Retrieve the name of each employee who has a dependent with the same first name as the employee.

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

Correlated Nested Queries (cont.)

- Here, the nested query has a different result *for each tuple* in the outer query (because it refers to E.Fname)
- A query written with nested SELECT... FROM... WHERE... blocks and using the = or IN comparison operators can ***always*** be expressed as a single block query. For example, the query above may be written as

```
SELECT    E.Fname, E.Lname
FROM      EMPLOYEE E, DEPENDENT D
WHERE     E.Ssn=D.Essn AND E.Fname =
          D.Dependent_name;
```

The EXISTS Function in SQL

- EXISTS is used to check whether the result of a query is empty (contains no tuples) or not (contains one or more tuples)
  - Applied to a query, but returns a boolean result (TRUE or FALSE)
  - Can be used in the WHERE-clause as a condition
  - EXISTS (Q) evaluates to TRUE if the result of Q has one or more tuple; evaluates to FALSE if the result of Q has no tuples
  - Opposite is NOT EXISTS

The EXISTS Function (cont.)

- For example, to retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee, we write

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

The EXISTS Function (cont.)

- To retrieve the names of employees who have no dependents, we write

```
SELECT   E.FNAME, E.LNAME
FROM     EMPLOYEE E
WHERE    NOT EXISTS   (SELECT  *
                       FROM      DEPENDENT D
                       WHERE     E.SSN=D.ESSN) ;
```

- EXISTS and NOT EXISTS are necessary for the expressive power of SQL
- SQL also has UNIQUE(Q) function that returns TRUE if there are no duplicate tuples in the result of query Q.

Explicit (Literal) Sets in SQL

- An **explicit (enumerated) set of values** is enclosed in parentheses
- To retrieve the social security numbers of all employees who work on project number 1, 2, or 3

```
SELECT   DISTINCT ESSN
FROM   WORKS_ON
WHERE    PNO IN  (1, 2, 3) ;
```

Renaming of Attributes

- It is possible to rename any attribute that appears in the result of a query by adding the qualifier AS followed by the desired new name.

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

Joined Tables (Relations) in SQL

- Permit users to specify a table resulting from a join operation in the FROM clause of a query. For example, to retrieve the name and address of every employee who works for the 'Research' department, we write

  SELECT Fname, Lname, Address

  FROM (EMPLOYEE JOIN DEPARTMENT ON Dno = Dnumber)

  WHERE Dname = 'Research';

  The FROM clause contains a single joined table. The attributes of such a table are all the attributes of the first table, EMPLOYEE, followed by all the attributes of the second table, DEPARTMENT.

Joined Tables (Relations) in SQL

- This type of join is also called an inner join, where a tuple is included in the result only if a matching tuple exists in the other relation.
- The concept of a joined table also allows the user to specify different types of join, such as NATURAL JOIN and various types of OUTER JOINS.

Natural Join in SQL

- In NATURAL JOIN, no join condition is specified. An implicit EQUIJOIN condition for each pair of attributes with the same name is created. Each such pair of attributes is included only once in the resulting relation.
- If the names of the join attributes are not the same in the base relations, it is possible to rename the attributes so that they match, and then to apply NATURAL JOIN.

  SELECT Fname, Lname, Address

  FROM (EMPLOYEE NATURAL JOIN (DEPARTMENT AS DEPT (Dname, Dno, Mssn, Msdate)))

  WHERE Dname = 'Research';

Outer Joins in SQL

- LEFT OUTER JOIN – every tuple in the left table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the right table)
- RIGHT OUTER JOIN – every tuple in the right table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the left table
- FULL OUTER JOIN – the keyword OUTER may be omitted. Combines result if LEFT and RIGHT OUTER JOIN.

```
SELECT E.Lname AS Employee_name, S.Lname AS
        Supervisor_name
FROM (EMPLOYEE AS E LEFT OUTER JOIN EMPLOYEE
        AS S ON E.Super_ssn = S.Ssn);
```

Aggregate Functions

- Include **COUNT, SUM, MAX, MIN, and AVG**
- These can summarize information from multiple tuples into a single tuple. NULL values are discarded.
- To find the maximum salary, the minimum salary, and the average salary among all employees, we write

```
SELECT   MAX(Salary) AS High_sal, MIN(Salary)
                AS Low_sal, AVG(Salary) AS Mean_sal
FROM   EMPLOYEE ;
```

Aggregate Functions (cont.)

- To find the maximum salary, the minimum salary, and the average salary among employees who work for the 'Research' department, we write

```
SELECT MAX(E.Salary), MIN(E.Salary),
                AVG(E.Salary)
FROM EMPLOYEE E, DEPARTMENT D
WHERE E.Dno=D.Dnumber AND
                D.Dname='Research' ;
```

Aggregate Functions (cont.)

- To retrieve total number of employees in the company, we write

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

- To find the number of employees in the 'Research' department, we write,

```
SELECT    COUNT (*)
FROM      EMPLOYEE AS E, DEPARTMENT AS D
WHERE     E.Dno=D.Dnumber AND D.Dname='Research' ;
```

Aggregate Functions (cont.)

- Here the asterisk (*) refers to the rows (tuples), so COUNT (*) returns the number of rows in the result of the query.
- We may also use the COUNT function to count values in a column rather than tuples. For example, to count the number of distinct salary values in the database, we write

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

Aggregate Functions (cont.)

- We can specify a correlated nested query with an aggregate function, and then use the nested query in the WHERE clause of an outer query. For example, to retrieve the names of all employees who have two or more dependents, we write

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

Grouping: GROUP BY and HAVING

- In many cases we want to apply the aggregate functions to subgroups of tuples in a relation, where the subgroups are based on some attribute values.
- In these cases we need to partition the relation into nonoverlapping subsets (or groups) of tuples. Each group (partition) will consist of the tuples that have the same value of some attribute(s), called the grouping attribute(s).
- We can then apply the function to each such group independently to produce summary information about each group.

Grouping: GROUP BY and HAVING

- SQL has a GROUP BY clause for this purpose. The GROUP BY clause specifies the grouping attributes, which should also appear in the SELECT clause, so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attribute(s).
- For example, for each department, retrieve the department number, the number of employees in the department, and their average salary.

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

- If NULLs exist in the grouping attribute, then a separate group is created for all tuples with a NULL value in the grouping attribute.

Grouping: GROUP BY and HAVING

- We can also use a join condition in conjunction with GROUP BY. In this case, the grouping and aggregate functions are applied after the joining of the two relations in the WHERE clause. For example, for each project, retrieve the project number, the project name, and the number of employees who work on that project.

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

Grouping: GROUP BY and HAVING

- Sometimes we want to retrieve the values of these aggregate 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)
- 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.

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

Summary of SQL Queries

- A query in SQL can consist of up to six clauses, but only the first two, SELECT and FROM, are mandatory. The clauses are specified in the following order:

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

Summary of SQL Queries (cont.)

- 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, as well as joined tables
- 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 query result
  - Conceptually, a query is evaluated by first applying the WHERE-clause, then GROUP BY and HAVING, and finally the SELECT-clause and ORDER BY

Assertions in SQL

- Assertions can be used to specify additional types of constraints that are outside the scope of the built-in relational model constraints (primary and unique keys, entity integrity, and referential integrity) using CREATE ASSERTION statement.
- Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause.
- For example, to specify the constraint that the salary of an employee must not be greater than the salary of the manager of the department that the employee works for, we write

```
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 ) );
```

Triggers in SQL

- Triggers can be used to specify automatic actions that the database system will perform when certain events and conditions occur.
- Typical trigger has three components: Event (such as an insert, deleted, or update operation), Condition and Action (to be taken when the condition is satisfied).
- Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor.

```
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 in SQL

- A view is a single table that is derived from other tables (base tables or previously defined views).
- A view does not necessarily exist in physical form; it is considered to be a virtual table, in contrast to base tables, whose tuples are always physically stored in the database.
- 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.
- In SQL, the command to specify a view is **CREATE VIEW**. The view is given a (virtual) table name (or view name), a possible list of attribute names (when arithmetic operations and aggregate functions are specified or when we want the names to be different from the attributes in the base relations), and a query to specify the contents of the view.

Views in SQL

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

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;
```

Views in SQL

- We can now specify SQL queries on a view in the same way we specify queries involving base tables. For example,

  SELECT Fname, Lname

  FROM WORKS_ON1

  WHERE Pname = ‘ProductX’;
- Views are also used as a security and authorization mechanism.
- A view is supposed to be always up-to-date; if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes.
- If we do not need a view anymore, we can use the DROP VIEW command to dispose of it. For example,

  DROP VIEW WORKS_ON1;

View Implementation

- View implementation is hidden from the user. Two main approaches have been suggested.
  1. **Query modification:** Query modification, involves modifying or transforming the view query (submitted by the user) into a query on the underlying base tables. ***Disadvantage:*** Inefficient for views defined via complex queries, especially if many queries are to be applied to the view within a short time period.
  2. **View materialization:** Involves physically creating a temporary or permanent view table when the view is first queried or created and keeping that table on the assumption that other queries on the view will follow. An efficient strategy for automatically updating the view table when the base tables are updated must be developed in order to keep the view up-to-date. Techniques using the concept of incremental update have been developed for this purpose. If the view is not queried for a certain period of time, the system may then automatically remove and recompute it when future queries reference the view.

Updating of Views

- All views can be queried for retrievals, but many views cannot be updated.
- Update on a view on a single table without aggregate operations: If view includes key and NOT NULL attributes, view update may map to an update on the base table.
- Views involving joins and aggregate functions are generally not updatable unless they can be mapped to unique updates on the base tables.
- When a user intends to update a view, must add the clause **`WITH CHECK OPTION`** at the end of the CREATE VIEW statement. This allows the system to check for updatability. If view is not updatable, error will be generated. If view is updatable, system will create a mapping strategy to process view updates.

Schema Modification

- The **DROP** command can be used to drop named schema elements, such as tables, domains, types, or constraints.
- One can also drop a whole schema if it is no longer needed by using the DROP SCHEMA command. There are two drop behavior options: CASCADE and RESTRICT. The CASCADE option removes schema and all its tables, domains, and other elements.

  DROP SCHEMA COMPANY CASCADE;
- If the RESTRICT option is chosen in place of CASCADE, the schema is dropped only if it has no elements in it. To use the RESTRICT option, the user must first individually drop each element in the schema, then drop the schema itself.

Schema Modification

- If a base relation within a schema is no longer needed, the relation and its definition can be deleted by using the DROP TABLE command.

  DROP TABLE DEPENDENT CASCADE;
- If the RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is not referenced in any constraints .
- With the CASCADE option, all such constraints, views, and other elements that reference the table being dropped are also dropped automatically from the schema, along with the table itself. Notice that the DROP TABLE command not only deletes all the records in the table if successful, but also removes the table definition from the catalog.

Schema Modification

- The definition of a base table or of other named schema elements can be changed by using the **ALTER** command.
- For base tables, the possible alter table actions include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints. For example,

  ALTER TABLE EMPLOYEE ADD COLUMN Job VARCHAR(12);
- To drop a column, we must choose either CASCADE or RESTRICT for drop behavior.

  ALTER TABLE EMPLOYEE DROP COLUMN Address CASCADE;

Schema Modification

- If CASCADE is chosen, all constraints and views that reference the column are dropped automatically from the schema, along with the column. If RESTRICT is chosen, the command is successful only if no views or constraints (or other schema elements) reference the column.
- It is also possible to alter a column definition by dropping an existing default clause or by defining a new default clause. For example,

  ALTER TABLE DEPARTMENT ALTER COLUMN Mgr_ssn DROP DEFAULT;

  ALTER TABLE DEPARTMENT ALTER COLUMN Mgr_ssn SET DEFAULT ‘333445555’;

Schema Modification

- One can also change the constraints specified on a table by adding or dropping a named constraint. For example,

  ALTER TABLE EMPLOYEE DROP CONSTRAINT EMPSUPERFK CASCADE;
- We can also redefine a replacement constraint by adding a new constraint to the relation, if needed. This is specified by using the ADD CONSTRAINT keyword in the ALTER TABLE statement followed by the new constraint, which can be named or unnamed and can be of any of the table constraint types discussed.