More SQL: Complex Queries

This chapter describes more advanced features of the SQL language standard for relational databases.

5.1 More Complex SQL Retrieval Queries

Because of the generality and expressive power of the language, there are many additional features that allow users to specify more complex retrievals from the database.

Comparisons Involving NULL and Three-Valued Logic

SQL has various rules for dealing with NULL values. NULL is used to represent a missing value, but that it usually has one of three different interpretations—value *unknown* (exists but is not known), value *not available* (exists but is purposely withheld), or value *not applicable* (the attribute is undefined for this tuple). Consider the following examples to illustrate each of the meanings of NULL.

1. **Unknown value.** A person's date of birth is not known, so it is represented by NULL in the database.
2. **Unavailable or withheld value.** A person has a home phone but does not want it to be listed, so it is withheld and represented as NULL in the database.
3. **Not applicable attribute.** An attribute LastCollegeDegree would be NULL for a person who has no college degrees because it does not apply to that person. It is often not possible to determine which of the meanings is intended; for example, a NULL for the home phone of a person can have any of the three meanings.

SQL does not distinguish between the different meanings of NULL.

In general, each individual NULL value is considered to be different from every other NULL value in the various database records.

When a NULL is involved in a comparison operation, the result is considered to be UNKNOWN (it may be TRUE or it may be FALSE). Hence, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead of the standard two-valued (Boolean) logic with values TRUE or FALSE.
It is therefore necessary to define the results (or truth values) of three-valued
logical expressions when the logical connectives AND, OR, and NOT are used. Table shows the resulting values.
Logical Connectives in Three-Valued Logic

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN
OR			
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN
NOT	TRUE	TRUE	TRUE
TRUE	FALSE		
FALSE	TRUE		
UNKNOWN	UNKNOWN		

The rows and columns represent the values of the results of comparison conditions, which would typically appear in the WHERE clause of an SQL query. Each expression result would have a value of TRUE, FALSE, or UNKNOWN.

In select-project-join queries, the general rule is that only those combinations of tuples that evaluate the logical expression in the WHERE clause of the query to TRUE is selected.

Tuple combinations that evaluate to FALSE or UNKNOWN are not selected.

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 operators **IS** or **IS NOT**. This is because SQL considers each NULL value as being distinct from every other NULL value, so equality comparison is not appropriate. It follows that when a join condition is specified, tuples with NULL values for the join attributes are not included in the result

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 Comparison:

SQL Join Operators:

Inner Join:
The relational join operation merges rows from two tables and returns the rows with one of the following conditions:

- Have common values for common attributes , this is called a natural join, and usually refers to attributes linked by a foreign key constraint.
- Meet a given join condition (equality or inequality)
- Have common values in common attributes or have no matching values , this is called an outer join.
  Example:

SELECT Pnumber
FROM P as PROJECT, D as DEPARTMENT, E as EMPLOYEE
WHERE P. Dnum=D.Dnumber AND E.Ssn= 12345

- The FROM clause indicates which tables are to be joined. If three or more tables are included, the join operation takes place two tables at a time, from left to right. If we are joining table T1, T2, T3; the first join is T1 with T2; the results of that join are then joined with T3.
- The join condition in the WHERE clause tells the SELECT statement which rows will be returned, i.e., those tuples that satisfy the condition
- The number of join conditions is always equal to the number of tables being joined minus one. For example if we are joining T1, T2 and T3, we need to have two join conditions J1 and J2.
- All join conditions are connected with an AND logical operator.
- Generally, the join condition will be an equality comparison of the primary key in one table and the related foreign key in the second or any additional tables (equijoin) or an inequality condition (theta join)

CROSS JOIN:

A cross join performs a Cartesian product of two tables.
Syntax:

SELECT column_list FROM table1 CROSS JOIN table2

SELECT * FROM P CROSS JOIN V;

NATURAL JOIN:

Natural join returns all rows with matching values in the matching columns and eliminates duplicates.
Syntax:
SELECT column-list FROM table1 NATURAL JOIN table 2
The natural join will perform the following tasks:

- Determine the common attributes by looking for attributes with identical names and compatible data types.
- Select only the rows with common values in the common attribute
- If there are no common attributes, return the relational product of the two tables.

JOIN USING CLAUSE:
Another way to express a natural join is through USING keyword. The query returns only the rows with matching values in the column indicated in the USING clause- that column must exist.
Syntax:
SELECT column-list FROM table1 JOIN table2 USING(common-column)

JOIN ON CLAUSE:
When the tables have no common attribute names we use the JOIN ON operator. The query will only return the rows that meet the indicated join condition. The join condition will typically include an equality comparison expression of two columns.
Syntax:
SELECT column-list FROM table1 JOIN table2 ON join-condition

```
SELECT      INVOICE.INV_NUMBER, PRODUCT.P_CODE, P_DESCRIPT,
LINE_UNITS, LINE_PRICE
FROM              INVOICE JOIN LINE ON
INVOICE.INV_NUMBER=LINE.INV_NUMBER
              JOIN PRODUCT ON LINE.P_CODE= PRODUCT.P_CODE;

SELECT      E.EMP_MGR, M.EMP_LNAME
FROM              EMP E JOIN EMP M ON E.EMP_MGR= M.EMP_NUM;
```

OUTER JOIN:

An outer join returns not only the rows matching the join condition, it returns the rows with unmatched values as well.

There are three types of outer joins: left, right and full.
The left and right designations reflect the order in which the tables are processed by the DBMS. The first table named in the FROM clause will be the left side, and the second table named will be the right side and so on.

The left outer join returns not only the rows matching the join condition, it returns the rows in the left table with unmatched values in the right table.
Syntax:

```
SELECT       column_list
FROM              table1 LEFT[OUTER} JOIN table2 ON join_condition
```

Example: this following query lists the product code, vendor code and vendor name for all products and includes those vendors with no matching products:

```
SELECT      P_CODE, V_CODE, V_NAME
FROM              VENDOR LEFT JOIN PRODUCT ON V.V_CODE=
PRODUCT.V_CODE;
```

The right outer join returns only the rows matching the join condition; it returns the rows in the right table with unmatched values in the left table

```
SELECT       column_list
FROM       table1 RIGHT [OUTER} JOIN table2 ON join_condition

SELECT      P_CODE, VENDOR_CODE, V_NAME
FROM              VENDOR RIGHT JOIN PRODUCT ON VENDOR.V_CODE=
PRODUCT.V_CODE;
```

Outer queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using **nested queries**, which are complete select-from-where blocks within the WHERE clause of another query. That other query is called the **outer query**.

Q4A introduces the comparison operator **IN**, which compares a value *v* with a set (or multiset) of values *V* and evaluates to **TRUE** if *v* is one of the elements in *V*.
The first nested query selects the project numbers of projects that have an employee
with last name ‘Smith’ involved as manager, while the second nested query selects the project numbers of projects that have an employee with last name ‘Smith’ involved as worker. In the outer query, we use

the **OR** logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested 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' );
```

If a nested query returns a single attribute *and* a single tuple, the query result will be a single (scalar) value. In such cases, it is permissible to use = instead of IN for the comparison operator. In general, the nested query will return a **table** (relation),which is a set or multiset of tuples. 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. In this example, the IN operator compares the subtuple of values in parentheses (Pno, Hours) within each tuple in WORKS_ON with the set of type-compatible tuples produced by the nested query.

The = ANY (or = SOME) operator :

A number of other comparison operators can be used to compare a single value *v* (typically an attribute name) to a set or multiset *v* (typically a nested query). 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*.
An example is the following query, which 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 );
```

(Notice that this query can also be specified using the MAX aggregate function)

In general, we can have several levels of nested queries. We can be faced with possible ambiguity among attribute names if attributes of the same name exist—one in a relation in the FROM clause of the *outer query,* and another in a relation in the FROM clause of the *nested query.*

The rule is that a reference to an *unqualified attribute* refers to the relation declared in the **innermost nested query**.

For example, in the SELECT clause and WHERE clause of the first nested query of Q4A, a reference to any unqualified attribute of the PROJECT relation refers to the PROJECT relation specified in the FROM clause of the nested query. To refer to an attribute of the PROJECT relation specified in the outer query, we specify and refer to an *alias* (tuple variable) for that relation.
These rules are similar to scope rules for program variables in most programming languages that allow nested procedures and functions. To illustrate the potential ambiguity of attribute names in nested queries, consider Query 16.
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 );
```

In the nested query of Q16, we must qualify E.Sex because it refers to the Sex attribute of EMPLOYEE from the outer query, and DEPENDENT also has an attribute called Sex.

If there were any unqualified references to Sex in the nested query, they would refer to the Sex attribute of DEPENDENT. However, we would not *have to* qualify the attributes Fname and Ssn of EMPLOYEE if they appeared in the nested query because the DEPENDENT relation does not have attributes called Fname and Ssn, so there is no ambiguity.

It is generally advisable to create tuple variables (aliases) for *all the tables referenced in an SQL query* to avoid potential errors and ambiguities, as illustrated in Q16.

Correlated Nested Queries

Whenever a condition in the WHERE clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be **correlated**.

A correlated query is defined by considering that the *nested query is evaluated once for each tuple (or combination of tuples) in the outer query.*

For example, we can think of Q16 as follows:

For *each* EMPLOYEE tuple, evaluate the nested query, which retrieves the Essn values for all DEPENDENT tuples with the same sex and name as that EMPLOYEE tuple; if the Ssn value of the EMPLOYEE tuple is *in* the result of the nested query, then select that EMPLOYEE tuple

In general, 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,
Q16 may be written as in Q16A:

```
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

The EXISTS function in SQL is used to check whether the result of a correlated nested query is *empty* (contains no tuples) or not. The result of EXISTS is a Boolean value **TRUE** if the nested query result contains at least one tuple, or **FALSE** if the nested query result contains no tuples. We illustrate the use of EXISTS—and NOT EXISTS—with some examples. First, we formulate Query 16 in an alternative form that uses EXISTS as in Q16B:

```
Q16B: 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);
```

EXISTS and NOT EXISTS are typically used in conjunction with a correlated nested query.

Q16B: 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);

In Q16B, the nested query references the Ssn, Fname, and Sex attributes of the EMPLOYEE relation from the outer query. We can think of Q16B as follows:

For each EMPLOYEE tuple, evaluate the nested query, which retrieves all DEPENDENT tuples with the same Essn, Sex, and Dependent_name as the EMPLOYEE tuple; if at least one
tuple EXISTS in the result of the nested query, then select that EMPLOYEE tuple.
In general, EXISTS(Q) returns **TRUE** if there is *at least one tuple* in the result of the nested query Q, and it returns **FALSE** otherwise.

On the other hand, NOT EXISTS(Q) returns **TRUE** if there are *no tuples* in the result of nested query Q, and it returns **FALSE** otherwise.

Next, we illustrate the use of NOT EXISTS.
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 a particular EMPLOYEE tuple. If *none exist,* the EMPLOYEE tuple is selected because the
WHERE-clause condition will evaluate to **TRUE** in this case. We can explain Q6 as
follows:
For *each* EMPLOYEE tuple, the correlated nested query selects all DEPENDENT tuples whose Essn value matches the EMPLOYEE Ssn; if the result is empty, no dependents are related to the employee, so we select that EMPLOYEE tuple and retrieve its Fname and Lname.

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 );
```

One way to write this query is shown in Q7, where we specify two nested correlated queries; the first selects all DEPENDENT tuples related to an EMPLOYEE, and the second selects all DEPARTMENT tuples managed by the EMPLOYEE. If at least one of the first and at least one of the second exists, we select the EMPLOYEE tuple.
Can you rewrite this query using only a single nested query or no nested queries?

The query Q3: *Retrieve the name of each employee who works on* all *the projects controlled by department number 5*

can be written using EXISTS and NOT EXISTS in SQL systems. We show two ways of specifying this query Q3 in SQL as Q3A and Q3B.
This is an example of certain types of queries that require *universal quantification*. One way to write this query is to use the construct (*S2* EXCEPT *S1*) as explained next, and checking whether the result is empty. This option is shown as Q3A.

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) );
```

In Q3A, the first subquery (which is not correlated with the outer query) selects all projects controlled by department 5, and the second subquery (which is correlated) selects all projects that the particular employee being considered works on.
If the set difference of the first subquery result MINUS (EXCEPT) the second subquery result is empty, it means that the employee works on all the projects and is therefore selected.
The second option is shown as Q3B. Notice that we need two-level nesting in Q3B and that this formulation is quite a bit more complex than Q3A, which uses NOT EXISTS and EXCEPT.

```
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 )));
```

In Q3B, the outer nested query selects any WORKS_ON (B) tuples whose Pno is of a project controlled by department 5, *if* there is not a WORKS_ON (C) tuple with the same Pno and the same Ssn as that of the EMPLOYEE tuple under consideration in the outer query. If no such tuple exists, we select the EMPLOYEE tuple.
The form of Q3B matches the following rephrasing of Query 3: Select each employee such that there does not exist a project controlled by department 5 that the employee does not work on.

There is another SQL function, UNIQUE(Q), which returns TRUE if there are no duplicate tuples in the result of query Q; otherwise, it returns FALSE.

Explicit Sets in SQL

We have seen several queries with a nested query in the WHERE clause. It is also possible to use an **explicit set of values** in the WHERE clause, rather than a nested query. Such a set is enclosed in parentheses in SQL.

Query 17. Retrieve the Social Security numbers 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);
```

Aggregate Functions in SQL

Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary.
Grouping is used to create subgroups of tuples before summarization. Grouping and aggregation are required in many database applications,
A number of built-in aggregate functions exist: **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**.
The COUNT function returns the number of tuples or values as specified in a query
The functions SUM, MAX, MIN, and AVG can be applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values.
These functions can be used in the SELECT clause or in a HAVING clause.
The functions MAX and MIN can also be used with attributes that have nonnumeric domains if the domain values have a *total ordering* among one another.3We illustrate the use of these functions with sample queries.
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;

Query 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) in Q23, then duplicate values will not be eliminated. However, any tuples with NULL for SALARY will not be counted.

In general NULL values are discarded when aggregate functions are applied.

Grouping: The GROUP BY and HAVING Clauses

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.

For example, we may want to find the average salary of employees *in each department* or the number of employees who work *on each project*. In these cases we need to **partition** the relation into non-overlapping 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. 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).

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;

Specifying General Constraints as Assertions in SQL

In SQL, users can specify general constraints via **declarative assertions**, using the
CREATE ASSERTION statement of the DDL.

Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.

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* in SQL, we can write the following assertion:

```
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 ) );
```

The constraint name SALARY_CONSTRAINT is followed by the keyword CHECK, which is followed by a **condition** in parentheses that must hold true on every database state for the assertion to be satisfied.
The constraint name can be used later to refer to the constraint or to modify or drop it. The DBMS is responsible for ensuring that the condition is not violated.
Any WHERE clause condition can be used, but many constraints can be specified using the EXISTS and NOT EXISTS style of SQL conditions.
Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE, the constraint is **violated**.
The constraint is **satisfied** by a database state if *no combination of tuples* in that database state violates the constraint.

The basic technique for writing such assertions is to specify a query that selects any tuples *that violate the desired condition*. By including this query inside a NOT EXISTS clause, the assertion will specify that the result of this query must be empty so that the condition will always be TRUE.

Thus, the assertion is violated if the result of the query is not empty.

In the preceding example, the query selects all employees whose salaries are greater than the salary of the manager of their department.
If the result of the query is not empty, the assertion is violated.

Note that the CHECK clause and constraint condition can also be used to specify constraints on *individual* attributes and domains and on *individual* tuples.

A major difference between CREATE ASSERTION and the individual domain constraints and tuple constraints is that the CHECK clauses on individual attributes, domains, and tuples are checked in SQL *only when tuples are inserted or updated*. Hence, constraint checking can be implemented more efficiently by the DBMS in these cases.

The schema designer should use CHECK on attributes, domains, and tuples only when he or she is sure that the constraint can *only be violated by insertion or updating of tuples*.

On the other hand, the schema designer should use CREATE ASSERTION only in cases where it is not possible to use CHECK on attributes, domains, or tuples, so that simple checks are implemented more efficiently by the DBMS.

Introduction to Triggers in SQL
Another important statement in SQL is CREATE TRIGGER.

In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied.
For example, it may be useful to specify a condition that, if violated, causes some user to be informed of the violation.

A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs.
The action that the DBMS must take in this case is to send an appropriate message to that user. The condition is thus used to **monitor** the database.

Other actions may be specified, such as executing a specific *stored procedure* or triggering other updates.

The CREATE TRIGGER statement is used to implement such actions in SQL.

Example:
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 which will notify the supervisor.

The trigger could then be written as in R5 below. Here we are using the syntax of the Oracle database system.

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);

The trigger is given the name SALARY_VIOLATION, which can be used to remove or deactivate the trigger later.

A typical trigger has three components:

1. The **event(s)**:
   These are usually database update operations that are explicitly applied to the database. In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor.
   The person who writes the trigger must make sure that all possible events are accounted for. In some cases, it may be necessary to write more than one trigger to cover all possible cases.
   These events are specified after the keyword **BEFORE** in our example, which means that the trigger should be executed before the triggering operation is executed.
   An alternative is to use the keyword **AFTER**, which specifies that the trigger should be executed after the operation specified in the event is completed.

2. The **condition** that determines whether the rule action should be executed:
   Once the triggering event has occurred, an *optional* condition may be evaluated.
   If *no condition* is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action

be executed. The condition is specified in the WHEN clause of the trigger.

3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed. In this example, the action is to execute the stored procedure INFORM_SUPERVISOR.

Triggers can be used in various applications, such as maintaining database consistency,
monitoring database updates, and updating derived data automatically. .

Views (Virtual Tables) in SQL

In this section we introduce the concept of a view in SQL.We show how views are specified, and then we discuss the problem of updating views and how views can be implemented by the DBMS.

Concept of a View in SQL

A **view** in SQL terminology is a single table that is derived from other tables. These other tables can be *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. This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view.

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.

For example, referring to the COMPANY database we may frequently issue queries that retrieve the employee name and the project names that the employee works on. Rather than having to specify the join of the three tables EMPLOYEE,WORKS_ON, and PROJECT

every time we issue this query, we can define a view that is specified as the result of these joins. Then we can issue queries on the view, which are specified as single table retrievals rather than as retrievals involving two joins on three tables.We call the EMPLOYEE,WORKS_ON, and PROJECT tables the **defining tables** of the view.

Specification of Views in SQL

In SQL, the command to specify a view is **CREATE VIEW**. The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view. If none of the view attributes results from applying functions or arithmetic operations, we do not have to specify new attribute names for the view, since they would be the same as the names of the attributes of the defining tables in the default case.

The views in V1 and V2 create virtual tables whose schemas are illustrated in Figure 5.2 when applied to the database schema of Figure 3.5.

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;

In V1, we did not specify any new attribute names for the view WORKS_ON1 (although we could have); in this case,WORKS_ON1 *inherits* the names of the view

attributes from the defining tables EMPLOYEE, PROJECT, and WORKS_ON.
View V2 explicitly specifies new attribute names for the view DEPT_INFO, using a one-to-one correspondence between the attributes specified in the CREATE VIEW clause and those specified in the SELECT clause of the query that defines the view.
We can now specify SQL queries on a view—or virtual table—in the same way we specify queries involving base tables.

For example, to retrieve the last name and first name of all employees who work on the 'ProductX' project, we can utilize the WORKS_ON1 view and specify the query as in QV1:

QV1: SELECT Fname, Lname
FROM WORKS_ON1
WHERE Pname='ProductX';

The same query would require the specification of two joins if specified on the base relations directly; one of the main advantages of a view is to simplify the specification of certain queries. 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.
Hence, 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. It is the responsibility of the DBMS and not the user to make sure that the view is kept up-to-date.

If we do not need a view any more, we can use the **DROP VIEW** command to dispose of it.
For example, to get rid of the view V1, we can use the SQL statement in V1A:

V1A: DROP VIEW WORKS_ON1;