MODULE 4
CHAPTER 1 SQL: Advanced Queries

More Complex SQL Retrieval Queries

1.1 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.

- **Unknown value**. A person's date of birth is not known, so it is represented by NULL in the database.
- **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.
- **Not applicable attribute.** An attribute Last College Degree 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.
- Hence, SQL does not distinguish between the different meanings of NULL.
- 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 5.1 shows the resulting values.

Table 5.1 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

- In Tables 5.1(a) and 5.1(b), 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.
- For example, the result of (FALSE AND UNKNOWN) is FALSE, whereas the result of (FALSE OR UNKNOWN) is UNKNOWN.
- 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.
- Query 18. Retrieve the names of all employees who do not have supervisors.

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

1.2 Nested Queries, Tuples, and Set/Multiset Comparisons

- Some 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.
- IN operator
- Which is a comparison operator that 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.

Ex: Query 17. Retrieve the Social Security numbers of all employees who work on project numbers 1, 2, or 3.

SELECT DISTINCT Essn

FROM WORKS_ON WHERE Pno **IN** (1, 2, 3);

Ex: 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');

- In this example, the IN operator compares the sub tuple 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 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.

Ex: 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);

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.Fame, E.Lname   FROM EMPLOYEE AS E

WHERE E.Ssn IN (SELECT Essn   FROM DEPENDENT AS D

WHERE E.Fame=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 Fame and Ssn of EMPLOYEE if they appeared in the nested query because the DEPENDENT relation does not have attributes called Fame and Ssn, so there is no ambiguity.

1.3 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. We can understand a correlated query better by considering that the nested query is evaluated once for each tuple (or combination of tuples) in the outer query.

Ex: Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```
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 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:

```
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;
```

1.4 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 EXIST 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);
```

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.

Ex. Query 6. Retrieve the names of employees who have no dependents.

```
SELECT Fname, Lname
FROM EMPLOYEE
WHERE NOT EXISTS (SELECT * FROM DEPENDENT
WHERE Ssn=Essn);
```

1.5 Explicit Sets and Renaming of Attributes in SQL

Explicit Sets

- 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.

Ex. Query 17. Retrieve the Social Security numbers of all employees who work on project numbers 1,2 or 3.

SELECT DISTINCT Essn FROM WORKS_ON WHERE Pno IN (1, 2, 3);

Renaming of Attributes

- In SQL, 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. Hence, the AS construct can be used to alias both attribute and relation names, and it can be used in both the SELECT and FROM clauses.

Ex. to retrieve the last name of each employee and his or her supervisor, while renaming the resulting attribute names as Employee_name and Supervisor_name. The new names will appear as column headers in the query result.

SELECT E.Lname **AS** Employee_name, S.Lname **AS** Supervisor_name

FROM EMPLOYEE AS E, EMPLOYEE AS S

WHERE E.Super_ssn=S.Ssn;

1.6 Joined Tables in SQL and Outer Joins

- The concept of a joined table (or joined relation) was incorporated into SQL to permit users to specify a table resulting from a join operation in the FROM clause of a query. This construct may be easier to comprehend than mixing together all the select and join conditions in the WHERE clause.

Ex. consider query Q1, which retrieves the name and address of every employee who works for the 'Research' department. It may be easier to specify the join of the EMPLOYEE and DEPARTMENT relations first, and then to select the desired tuples and attributes. This can be written in SQL as in Q1A:

Q1A**:** **SELECT** Fname, Lname, Address

FROM (EMPLOYEE **JOIN** DEPARTMENT ON **Dno=Dnumber**)

WHERE Dname='Research';

- 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 JOIN**.
- In a **NATURAL JOIN** on two relations R and S, no join condition is specified; an implicit **EQUIJOIN** condition for each pair of attributes with the same name from R and S is created. Each such pair of attributes is included only once in the resulting relation.

Q1B: SELECT Fname, Lname, Address FROM
(EMPLOYEE **NATURAL JOIN** (DEPARTMENT AS DEPT (Dname, Dno, Mssn,Msdate)))
WHERE Dname='Research';

- The default type of join in a joined table is called an **inner join**, where a tuple is included in the result only if a matching tuple exists in the other relation

There are a variety of outer join operations.

- **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 NULLvalues 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 NULLvalues for the attributes of the left table).
- **FULL OUTER JOIN:** It is a combination of left and right outer joins.
- The keyword **CROSS JOIN** is used to specify the **CARTESIAN PRODUCT** operation although this should be used only with the utmost care because it generates all possible tuple combinations.
- It is also possible to nest join specifications; that is, one of the tables in a join may itself be a joined table. This allows the specification of the join of three or more tables as a single joined table, which is called a **multiway join**.

EX: SELECT Pnumber, Dnum, Lname, Address, Bdate
FROM ((PROJECT **JOIN** DEPARTMENT ON Dnum=Dnumber)
JOIN EMPLOYEE ON Mgr_ssn=Ssn) WHERE Plocation='Stafford';

Not all SQL implementations have implemented the new syntax of joined tables. In some systems, a different syntax was used to specify outer joins by using the comparison operators **+=, =+, and +=+** for left, right, and full outer join, respectively, when specifying the join condition. For example, this syntax is available in **Oracle.** To specify the **left outer join** using this syntax, we could write the query as follows:

Ex. SELECT E.Lname, S.Lname
FROM EMPLOYEE E, EMPLOYEE S
WHERE E.Super_ssn += S.Ssn;

1.7 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.
- 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.

Ex. Query 19. Find the sum of the salaries of all employees, the maximum salary, the minimum salary and the average salary.

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

Ex. 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.

```
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=NUMBER AND DNAME = 'Research';
```

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, as in the next example.

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.

1.8 Grouping: The GROUP BY and HAVING Clauses

GROUP BY clause

- SQL has a GROUP BY clause. 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, 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.

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

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

In Q24, the EMPLOYEE tuples are partitioned into groups- each group having the same value for the grouping attribute Dno. Fence, each group contains the employees who work in the same department. The COUNT and AVG functions are grouping attribute and the aggregate functions to be applied on each group of tuples.

Figure 5.1 (a) illustrates how grouping works on Q24; it also shows the result of Q24.

Figure 5.1
Results of GROUP BY and HAVING. (a) Q24. (b) Q26.

(a)

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

Grouping EMPLOYEE tuples by the value of Dno

Dno	Count (*)	Avg (Salary)
5	4	33250
4	3	31000
1	1	55000

Result of Q24

HAVING clause

- SQL provides a **HAVING** clause, which can appear in conjunction with a **GROUP BY** clause. HAVING provides a condition on the summary information regarding the **group** of tuples associated with each value of the grouping attributes. Only the groups that satisfy the condition are retrieved in the result of the query. This is illustrated by Query 26.

Query 26. For each project on which more than to 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;
```

Notice that while selection conditions in the WHERE cause limit the tuples to which functions are applied, the HAVING clause serves to choose whole groups. Figure 5.1b illustrates the use of HAVING and displays the result of Q26.

(b)

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

These groups are not selected by the HAVING condition of Q26.

After applying the WHERE clause but before applying HAVING

Pname	Pnumber	...	Essn	Pno	Hours
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
Computerization	10		333445555	10	10.0
Computerization	10	...	999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

After applying the HAVING clause condition

Pname	Count (*)
ProductY	3
Computerization	3
Reorganization	3
Newbenefits	3

Result of Q26
(Pnumber not shown)

1..9 Discussion and Summary of SQL Queries

A retrieval query in SQL can consist of up to six clauses, but only the first two SELECT and FROM- -are mandatory. The query can span several lines, and is ended by a semicolon. Query terms are separated by spaces, and parentheses can be used to group relevant parts of a query in the standard way. The clauses are specified in the following order, with the clauses between square brackets [...] being optional:

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

The SELECT clause lists the attributes or functions to be retrieved. The FROM clause specifies all relations (tables) needed in the query, including joined relations, but not those in nested queries. The WHERE clause specifies the conditions for selecting the tuples from these relations, including join conditions if needed. GROUP BY

Specifies grouping attributes, whereas HAVING specifies a condition on the groups being selected rather than on the individual tuples. The built-in aggregate functions COUNT, SUM, MIN, MAX, and AVG are used in conjunction with grouping, but they can also be applied to all the selected tuples in a query without a GROUP BY clause. Finally, ORDER BY specifies an order for displaying the result of a query.

In order to formulate queries correctly, it is useful to consider the steps that define the meaning or semantics of each query. A **query is evaluated** conceptually by first applying the FROM clause (to identify all tables involved in the query or to materialize any joined tables), followed by the WHERE clause to select and join tuples, and then by GROUP BY and HAVING. Conceptually, ORDER BY is applied at the end to sort the query result.

Specifying Constraints as Assertions and Actions as Triggers

- In this section, we introduce two additional features of SQL: the **CREATE ASSERTION** statement and the **CREATE TRIGGER** statement.
- CREATE ASSERTION, which 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) that we presented early.
- CREATE TRIGGER, which can be used to specify automatic actions that the database system will perform when certain events and conditions occur. This type of functionality is generally referred to as active databases.

Specifying General Constraints as Assertions in SQL

ASSERTIONS

- In SQL, users can specify general constraints—those that do not fall into any of the categories described 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 data- base 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.

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

A typical trigger has **three components:**

Event: When this event happens, the trigger is activated

Condition (optional): If the condition is true, the trigger executes, otherwise skipped

Action: The actions performed by the trigger

The **action** is to be executed **automatically** if the **condition** is satisfied when **event** occurs.

Trigger: Events

Three event types

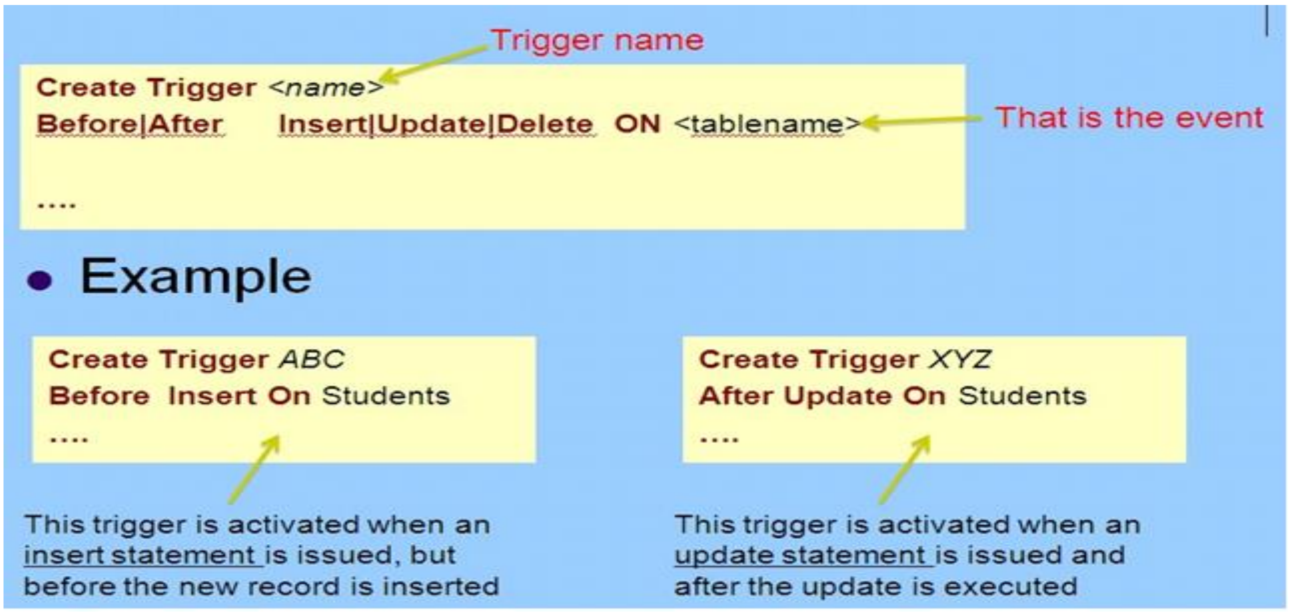
- Insert
- Update
- Delete

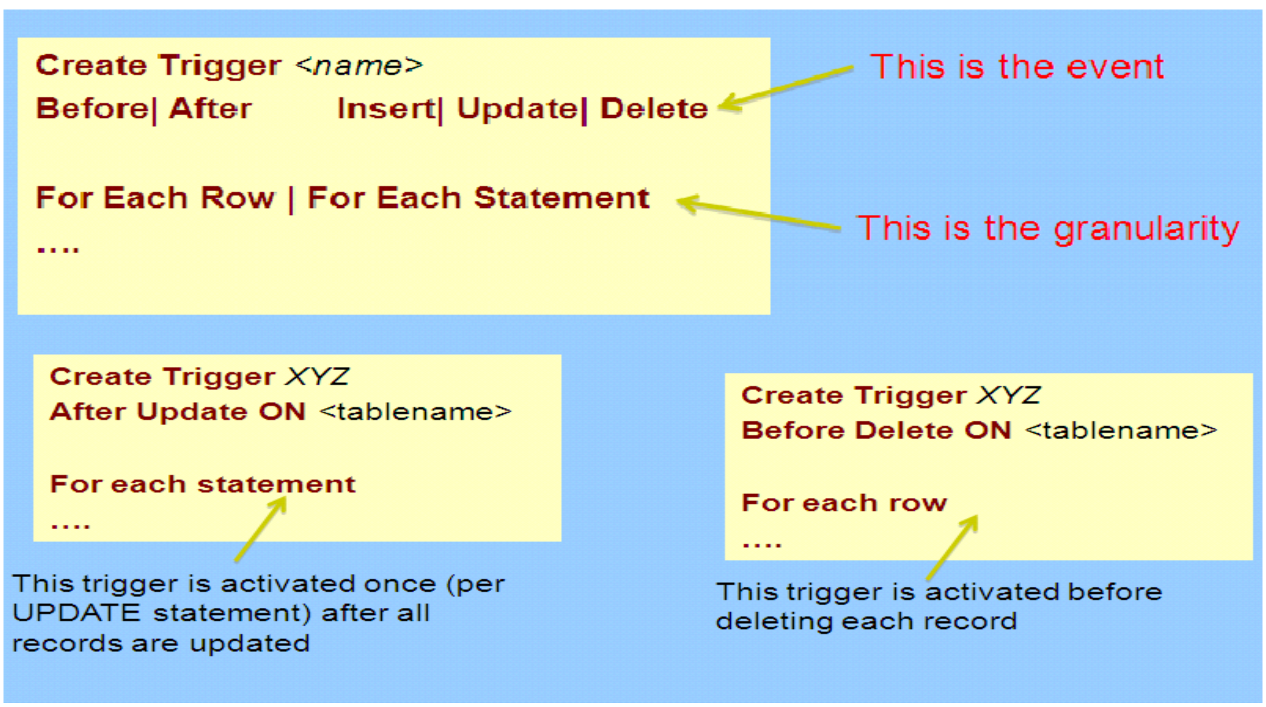
Two triggering times

- Before the event
- After the event

Two granularities

- Execute for each row
- Execute for each statement





Trigger: Condition

```
Create Trigger <name>
Before| After       Insert| Update| Delete On <tableName>
For Each Row | For Each Statement

When <condition>
...
```

That is the condition

If the employee salary > 150,000 then some actions will be taken

```
Create Trigger EmpSal
After Insert or Update On Employee
For Each Row
When (new.salary >150,000)
...
```

Trigger: Action

- The new values of inserted or updated records *(:new)*
- The old values of deleted or updated records *(:old)*

```
Create Trigger EmpSal
After Insert or Update On Employee
For Each Row
When (new.salary >150,000)
Begin
    if (:new.salary < 100,000) ...
End;
```

Trigger body

Inside "When", the "new" and "old" should not have ":"

Inside the trigger body, they should have ":"

Example 1

```
CREATE TRIGGER init_count BEFORE INSERT ON student          /* Event*/
DECLARE
    Count integer;
BEGIN
    Count =0;                                               /* Action*/
END
```

Example 2

```
CREATE TRIGGER incr_count AFTER INSERT ON student           /* Event*/
WHEN (new.age<18)                                        /*Condition*/
FOR EACH ROW
BEGIN
    Count = count + 1;                                      /* Action*/
END
```

1.10 Views (virtual table) in SQL

Concept of a View in SQL

- A view is a single table that is derived from one or more base tables or other views
- Views neither exist physically nor contain data itself, it depends on the base tables for its existence
- A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

Specification of Views in SQL Syntax:

```
CREATE VIEW view_name AS SELECT column_name(s) FROM table_name
WHERE condition
```

Example

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

Ex. Retrieve the Last name and First name of all employees who work on ‘ProductX’

SELECT Fname, Lname FROM WORKS_ON1
WHERE Pname=‘ProductX’;

- A view always shows **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 any more, we can use the DROP VIEW command DROP VIEW WORKS_ON1;

View Implementation and View Update View Implementation

- The problem of efficiently implementing a view for quering is complex Two main approaches have been suggested

Query Modification

- Modifying the view query into a query on the underlying base tables

Disadvantage: inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple queries are applied to the view within a short period of time.

Example

The query example# would be automatically modified to the following query by the DBMS

SELECT Fname, Lname
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE Ssn=Essn AND Pno=Pnumber AND Pname=”ProductX’;

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, that is **Incremental Update**
- **Incremental Update** determines what new tuples must be inserted, deleted, or modified in a materialized view table when a change is applied to one of the defining base table

View Update

- Updating of views is complicated and can be ambiguous

- An update on view defined on a single table without any aggregate functions can be mapped to an update on the underlying base table under certain conditions.
- View involving joins, an update operation may be mapped to update operations on the underlying base relations in multiple ways.

Example

Update the Pname attribute of 'john smith' from 'ProductX' to 'ProductY' UPDATE WORKS_ON1

SET Pname= 'ProductY'

WHERE Lname='smith' AND Fname='john' AND Pname= 'ProductX'

This query can be mapped into several updates on the base relations to give the desired effect on the view.Two possible updates (a) and (b) on the base relations corresponding to above query.

```
UPDATE WORKS_ON
SET Pno = (SELECT  Pnumber
FROM PROJECT
WHERE Pname= 'ProductY')
WHERE Essn IN (SELECT Ssn FROM  EMPLOYEE
                    WHERE Lname='smith' AND Fname='john')
AND
Pno=  (SELECT Pnumber FROM  PROJECT
WHERE Pname= 'ProductX');
UPDATE PROJECT SET  Pname='ProductY'
WHERE Pname= 'ProductX';
```

- Update (a) relates 'john smith' to the 'ProductY' PROJECT tuple in place of the 'ProductX' PROJECT tuple and is the most likely update.
- Update (b) would also give the desired update effect on the view, but it accomplishes this by changing the name of the 'ProductX' tuple in the PROJECT relation to 'ProductY'

OBSERVATIONS ON VIEWS

- A view with a single defining table is updatable if the view attributes contain the primary key of the base relation, as well as all attributes with the NOT NULL constraint that do not have default values specified
- Views defined on multiple tables using joins are generally not updatable
- Views defined using grouping and aggregate functions are not updatable
- In SQL, the clause WITH HECK OPTION must be added at the end of the view definition if a view is to be updated.

Advantages of Views

- Data independence
- Currency
- Improved security
- Reduced complexity
- Convenience
- Customization
- Data integrity