MODULE 3
SQL DML, Physical Data Organization

Sindhu Jose, CSE Dept, VJCET

SYLLABUS

- **SQL DML (Data Manipulation Language)**
  - SQL queries on single and multiple tables, Nested queries (correlated and non-correlated), Aggregation and grouping, Views, assertions, Triggers, SQL data types.
- **Physical Data Organization**
  - Review of terms: physical and logical records, blocking factor, pinned and unpinned organization. Heap files, Indexing, Singe level indices, numerical examples, Multi-level-indices, numerical examples, B-Trees & B+-Trees (structure only, algorithms not required), Extendible Hashing, Indexing on multiple keys – grid files

Data-manipulation language(DML)

- DML is short name of **Data Manipulation Language** which deals with data manipulation and includes most common SQL statements such SELECT, INSERT, UPDATE, DELETE, etc., and it is used to store, modify, retrieve, delete and update data in a database.
  - SELECT - retrieve data from a database
  - INSERT - insert data into a table
  - UPDATE - updates existing data within a table
  - DELETE - Delete all records from a database table

SQL queries on single and multiple tables

- SQL has one basic statement for retrieving information from a database; the SELECT statement

SELECT **<attribute list>**

FROM **<table list>**

WHERE **<condition>**

- <attribute list>
  - is a list of attribute names whose values are to be retrieved by the query
- <table list>
  - is a list of the relation names required to process the query
- <condition>
  - is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATIONS

DNUMBER	DLOCATION

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORKS_ON

ESSN	PNO	HOURS

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
	Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

DEPT_LOCATIONS	DNUMBER	DLOCATION
	1	Houston
	4	Stafford
	5	Bellaire
	5	Sugarland
	5	Houston

DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
	Research	5	333445555	1988-05-22
	Administration	4	987654321	1995-01-01
	Headquarters	1	888665555	1981-06-19

WORKS_ON	ESSN	PNO	HOURS
	123456789	1	32.5
	123456789	2	7.5
	666884444	3	40.0
	453453453	1	20.0
	453453453	2	20.0
	333445555	2	10.0
	333445555	3	10.0
	333445555	10	10.0
	333445555	20	10.0
	999887777	30	30.0
	999887777	10	10.0
	987987987	10	35.0
	987987987	30	5.0
	987654321	30	20.0
	987654321	20	15.0
	888665555	20	null

PROJECT	PNAME	PNUMBER	PLOCATION	DNUM
	ProductX	1	Bellaire	5
	ProductY	2	Sugarland	5
	ProductZ	3	Houston	5
	Computerization	10	Stafford	4
	Reorganization	20	Houston	1
	Newbenefits	30	Stafford	4

DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
	333445555	Alice	F	1986-04-05	DAUGHTER
	333445555	Theodore	M	1983-10-25	SON
	333445555	Joy	F	1958-05-03	SPOUSE
	987654321	Abner	M	1942-02-28	SPOUSE
	123456789	Michael	M	1988-01-04	SON
	123456789	Alice	F	1988-12-30	DAUGHTER
	123456789	Elizabeth	F	1967-05-05	SPOUSE

Q0.Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

```
SELECT   Bdate, Address
FROM     EMPLOYEE
WHERE    Fname='John' AND Minit='B' AND Lname='Smith';
```

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

Bdate	Address
1965-01-09	731 Fondren, Houston, TX

- Q1. Retrieve the name and address of all employees who work for the 'Research' department.

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

• Q2. 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

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

Dealing with Ambiguous Attribute Names and Renaming (Aliasing)

- In SQL the same name can be used for two (or more) attributes as long as the attributes are in different relations.
- If this is the case, and a query refers to two or more attributes with the same name, we must qualify the attribute name with the relation name, to prevent ambiguity.
- This is done by prefixing the relation name to the attribute name and separating the two by a period (.)

Eg) Suppose that the DNO and LNAME attributes of the EMPLOYEE relation were called DNUMBER and NAME and the DNAME attribute of DEPARTMENT was also called NAME; then, to prevent ambiguity, query Q1 would be rephrased as

```
Q1:   SELECT   Fname, Lname, Address
      FROM     EMPLOYEE, DEPARTMENT
      WHERE    Dname='Research' AND Dnumber=Dno;
```

```
Q1A:  SELECT   Fname, EMPLOYEE.Name, Address
      FROM     EMPLOYEE, DEPARTMENT
      WHERE    DEPARTMENT.Name='Research' AND
               DEPARTMENT.Dnumber=EMPLOYEE.Dnumber;
```

Q3.For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

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

- Alternative relation names E and S are called aliases or tuple variables, for the EMPLOYEE relation.
- An alias follow the keyword AS
- It is also possible to rename the relation attributes within the query in SQL by giving them aliases.

```
Q1A:  SELECT   Fname, EMPLOYEE.Name, Address
      FROM     EMPLOYEE, DEPARTMENT
      WHERE    DEPARTMENT.Name='Research' AND
               DEPARTMENT.Dnumber=EMPLOYEE.Dnumber;
```

```
SELECT   E.Fname, E.LName, E.Address
FROM     EMPLOYEE E, DEPARTMENT D
WHERE    D.DName='Research' AND D.Dnumber=E.Dno;
```

Use of the Asterisk (*)

- To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL;
- we just specify an asterisk (*), which stands for all the attributes

Q) Retrieves all the attribute values of any EMPLOYEE who works in DEPARTMENT number 5

```
SELECT    *
FROM      EMPLOYEE
WHERE     Dno=5;
```

Q)Retrieves all the attributes of an EMPLOYEE and the attributes of the DEPARTMENT in which he or she works for every employee of the 'Research' department

```
SELECT    *
FROM      EMPLOYEE, DEPARTMENT
WHERE     Dname='Research' AND Dno=Dnumber;
```

Tables as Sets in SQL

- SQL usually treats a table not as a set but rather as a multiset;
  - duplicate tuples can appear more than once in a table, and in the result of a query.
- SQL does not automatically eliminate duplicate tuples in the results of queries.

DISTINCT Keyword

- to eliminate duplicate tuples from the result of an SQL querys we use the keyword **DISTINCT** in the **SELECT** clause
- only distinct tuples should remain in the result
- a query with **SELECT DISTINCT eliminates duplicates**, whereas a query with **SELECT ALL does not**.
- SELECT with neither ALL nor DISTINCT is equivalent to SELECT ALL

Q) Retrieves the salary of every employee without distinct

```
SELECT    ALL Salary
FROM      EMPLOYEE;
```

→

(a)

Salary
30000
40000
25000
43000
38000
25000
25000
55000

Q) Retrieves the salary of every employee using keyword DISTINCT

```
Q11A:  SELECT    DISTINCT Salary
       FROM      EMPLOYEE;
```

→

(b)

Salary
30000
40000
25000
43000
38000
55000

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

UNION, EXCEPT and INTERSECT

- set union (UNION), set difference (EXCEPT), and set intersection (INTERSECT) operations.
- The relations resulting from these set operations are sets of tuples; that is, duplicate tuples are eliminated from the result.
- These set operations apply only to union-compatible relations, so we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations

UNION ALL

- The UNION ALL command combines the result set of two or more SELECT statements (allows duplicate values).

Example 1: UNION

Example of UNION

The **First** table,

ID	Name
1	abhi
2	adam

The **Second** table,

ID	Name
2	adam
3	Chester

Union SQL query will be,

```
SELECT * FROM First
UNION
SELECT * FROM Second;
```

The resultset table will look like,

ID	NAME
1	abhi
2	adam
3	Chester

Example 2: UNION ALL

Example of Union All

The **First** table,

ID	NAME
1	abhi
2	adam

The **Second** table,

ID	NAME
2	adam
3	Chester

Union All query will be like,

```
SELECT * FROM First
UNION ALL
SELECT * FROM Second;
```

The resultset table will look like,

ID	NAME
1	abhi
2	adam
2	adam
3	Chester

Eg 3) Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

```
(SELECT   DISTINCT Pnumber
 FROM     PROJECT, DEPARTMENT, EMPLOYEE
 WHERE    Dnum=Dnumber AND Mgr_ssn=Ssn
          AND Lname='Smith' )
 UNION
( SELECT  DISTINCT Pnumber
 FROM     PROJECT, WORKS_ON, EMPLOYEE
 WHERE    Pnumber=Pno AND Essn=Ssn
          AND Lname='Smith' );
```

- The first SELECT query retrieves the projects that involve a 'Smith' as manager of the department that controls the project, and the second retrieves the projects that involve a 'Smith' as a worker on the project. Applying the UNION operation to the two SELECT queries gives the desired result.

If you had the *suppliers* table populated with the following records:

supplier_id	supplier_name
1000	Microsoft
2000	Oracle
3000	Apple
4000	Samsung

And the *orders* table populated with the following records

order_id	order_date	supplier_id
1	2015-08-01	2000
2	2015-08-01	6000
3	2015-08-02	7000
4	2015-08-03	8000

```
SELECT supplier_id
FROM suppliers
UNION ALL
SELECT supplier_id
FROM orders
ORDER BY supplier_id;
```

supplier_id
1000
2000
2000
3000
4000
6000
7000
8000

INTERSECT Operator

- INTERSECT operator is used to return the records that are in common between two SELECT statements or data sets.
- It is the intersection of the two SELECT statements.

Example 1: INTERSECT

Example of Intersect

The **First** table,

ID	NAME
1	abhi
2	adam

The **Second** table,

ID	NAME
2	adam
3	Chester

Intersect query will be,

```
SELECT * FROM First
INTERSECT
SELECT * FROM Second;
```

The resultset table will look like

ID	NAME
2	adam

EXAMPLE 2 : INTERSECT

Customers Table:

ID	Name	Address	Age	Salary
1	Harsh	Delhi	20	3000
2	Pratik	Mumbai	21	4000
3	Akash	Kolkata	35	5000
4	Varun	Madras	30	2500
5	Souvik	Banaras	25	6000
6	Dhanraj	Siliguri	22	4500
7	Riya	Chennai	19	1500

Orders Table:

Oid	Date	Customer_id	Amount
102	2017-10-08	3	3000
100	2017-10-08	3	1500
101	2017-11-20	2	1560
103	2016-5-20	4	2060

```
SELECT  ID, NAME, Amount, Date
      FROM Customers
      LEFT JOIN Orders
      ON Customers.ID = Orders.Customer_id
INTERSECT
      SELECT  ID, NAME, Amount, Date
      FROM Customers
      RIGHT JOIN Orders
      ON Customers.ID = Orders.Customer_id;
```

Output:

ID	Name	Amount	Date
3	Akash	3000	2017-10-08
3	Akash	1500	2017-10-08
2	Pratik	1560	2017-11-20
4	Varun	2060	2016-05-20

EXCEPT

- The SQL EXCEPT clause/operator is used to combine two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement.
- This means EXCEPT returns only rows, which are not available in the second SELECT statement.

Syntax

The basic syntax of **EXCEPT** is as follows.

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

EXCEPT

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

Here, the given condition could be any given expression based on your requirement.

Example 1: EXCEPT

SELECT * FROM First

EXCEPT SELECT * FROM Second;

The **First** table,

ID	NAME
1	abhi
2	adam

The **Second** table,

ID	NAME
2	adam
3	Chester

The resultset table will look like,

ID	NAME
1	abhi

Example

Consider the following two tables.

Table 1 – CUSTOMERS Table is as follows.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Table 2 – ORDERS table is as follows.

OID	DATE	CUSTOMER_ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   LEFT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
EXCEPT
   SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   RIGHT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL

Substring Pattern Matching and Arithmetic Operators

- LIKE comparison operator can be used to compare parts of a character string.
- This can be used for string **pattern matching**.
- Partial strings are specified using two reserved characters:
- % replaces an arbitrary number of zero or more characters, and the underscore (_) replaces a single character.

Q) Retrieve all employees name whose address is in Houston, Texas.

```
SELECT    Fname, Lname
FROM      EMPLOYEE
WHERE     Address LIKE '%Houston,TX%';
```

Q) Find all employees name who were born during the 1950s.

```
SELECT   Fname, Lname
FROM     EMPLOYEE
WHERE    Bdate LIKE '_ _ 5 _ _ _ _ _ _ _';
```

- To retrieve all employees who were born during the 1950s,
- Here, '5' must be the third character of the string (according to our format for date),
- so we use the value '_ _ 5 _ _ _ _ _ _ _', with each underscore serving as a placeholder for an arbitrary character.

BETWEEN

Q)Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.

```
Q14:  SELECT  *
      FROM    EMPLOYEE
      WHERE   (Salary BETWEEN 30000 AND 40000) AND Dno = 5;
```

instead of

The condition (Salary BETWEEN 30000 AND 40000) in Q14 is equivalent to the condition ((Salary >= 30000) AND (Salary <= 40000))

Ordering of Query Results

- The tuples in the result of a query can be ordered by the values of one or more of the attributes that appear in the query result, using the ORDER BY clause.
- The **ORDER BY** statement in sql is used to sort the fetched data in either ascending or descending according to one or more columns.
  - By default ORDER BY sorts the data in ascending order.
  - We can use the keyword **DESC** to sort the data in **descending order** and the keyword **ASC** to sort in **ascending order**.

Q) Fetch all data from the table Student and sort the result in descending order according to the column ROLL_NO.

```
SELECT * FROM Student ORDER BY ROLL_NO DESC;
```

Output:

Student Table

ROLL_NO	NAME	ADDRESS	PHONE	Age
8	NIRAJ	ALIPUR	XXXXXXXXXX	19
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
2	PRATIK	BIHAR	XXXXXXXXXX	19
1	HARSH	DELHI	XXXXXXXXXX	18

Q) Fetch all data from the table Student and then sort the result in ascending order first according to the column Age. and then in descending order according to the column ROLL_NO.

```
SELECT * FROM Student ORDER BY Age ASC , ROLL_NO DESC;
```

Output:

ROLL_NO	NAME	ADDRESS	PHONE	Age
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
1	HARSH	DELHI	XXXXXXXXXX	18
8	NIRAJ	ALIPUR	XXXXXXXXXX	19
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
2	PRATIK	BIHAR	XXXXXXXXXX	19
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
3	RIYANKA	SILIGURI	XXXXXXXXXX	20

- Q) Retrieve a list of employees and the projects they are working on,
  ordered by department and, within each department, ordered alphabetically by last name, then first name.

```
Q15:  SELECT    D.Dname, E.Lname, E.Fname, P.Pname
      FROM      DEPARTMENT D, EMPLOYEE E, WORKS_ON W,
                PROJECT P
      WHERE     D.Dnumber= E.Dno AND E.Ssn= W.Essn AND
                W.Pno= P.Pnumber
      ORDER BY  D.Dname, E.Lname, E.Fname;
```

NESTED QUERIES

Correlated Nested Queries

- Some queries require that existing values in the database be fetched and then used in a comparison condition.
- Such queries can be 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 nested queries, 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 addition to 'IN' operator, all other comparison operators such as (>,>=,<,<= and < >) can be combined with the keywords ANY or ALL.

- 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.**

Q) Retrieve the name of each employee who has a dependent with the same first name and 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 );
```

- Here the nested query has a different result for each tuple in the outer query.
- 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.

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 IN operator can also compare a tuple of values in parentheses with a set or multiset of union-compatible tuples.

Example:

```
SELECT  DISTINCT ESSN
FROM    WORKS_ON
WHERE   (PNO, HOURS) IN     (SELECT PNO, HOURS FROM
                             WORKS_ON WHERE
                             SSN='123456789');
```

- Here it will select the social security numbers of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose SSN = '123456789') works on.
- In addition to the IN 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 keywords ANY and SOME have the same meaning.
- 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 V.

Q) Retrieve 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);
```

Non-Correlated Nested Queries

Q) Retrieve the name and address of all employees who work for the 'Research' Department

```
SELECT FNAME,LNAME, ADDRESS
FROM EMPLOYEE
WHERE DNO IN (SELECT DNUMBER
                    FROM DEPARTMENT
                    WHERE DNAME='Research')
```

- The nested query selects the number of the 'Research' department
- The outer query select an EMPLOYEE tuple if its DNO value is in the result of either nested query.
- In general, we can have several levels of nested queries
- A reference to an unqualified attribute refers to the relation declared in the innermost nested query
- In this example, the nested query is not correlated with the outer query

Aggregation and Grouping

- Aggregate functions are used to summarize information from multiple tuples into a single tuple summary.
- In SQL, built-in functions of aggregation are:

  1) **AVG 2) MIN 3) MAX 4) COUNT and 5) SUM**

1) **AVG :** It returns an average value of 'n', ignoring null values in column.

Eg) SELECT AVG (mark) FROM student;

2) MIN : SELECT MIN (mark) FROM student;

3) MAX: SELECT MAX (mark) FROM student;

4) COUNT : It returns the number of tuples or values as specified in a query.

Eg) SELECT COUNT (*) FROM student;

5)SUM : SELECT SUM (mark) FROM student;

Q) 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;

Q) Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, and the average salary in this department.

SELECT SUM (SALARY), MAX (SALARY), MIN (SALARY), AVG (SALARY) FROM EMPLOYEE, DEPARTMENT WHERE DNO=DNUMBER AND DNAME='Research';

Q) Retrieve i) the total number of employees in the company and ii) the number of employees in the 'Research' department.

i)SELECT COUNT (*) FROM EMPLOYEE;

ii) SELECT COUNT (*) FROM EMPLOYEE, DEPARTMENT WHERE DNO=DNUMBER AND DNAME='Research';

Q) Count the number of distinct salary values in the database.

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

Q) Retrieve the names of all employees who have two or more dependents

```
SELECT LNAME, FNAME FROM EMPLOYEE WHERE (SELECT
    COUNT (*) FROM DEPENDENT  WHERE SSN=ESSN) >= 2;
```

- Here the correlated nested query counts the number of dependents that each employee has; if this is greater than or equal to 2, the employee tuple is selected.

Grouping : GROUP BY and HAVING Clauses

- In order to partition the relation into non overlapping subsets(or groups) of tuples, each group will consist of the tuples that have the same value of attributes called the grouping attributes.
- We can 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 attributes.

Q) 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;

Q) 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;

HAVING Clause

- It is useful to state a condition that applies to groups rather than to tuples.
- It can be used in conjunction with the GROUP BY clause
- HAVING provides a condition on the group of tuples associated with each value of the grouping attributes and only the groups that satisfy the condition are retrieved in the result of the query.

Q) 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;**

Q)For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

SELECT PNUMBER, PNAME, **COUNT (*) FROM** PROJECT, WORKS_ON, EMPLOYEE **WHERE** PNUMBER=PNO **AND SSN=ESSN AND DNO=5 GROUP BY** PNUMBER, PNAME;

Q)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.

SELECT Dnumber, COUNT (*) FROM DEPARTMENT, EMPLOYEE WHERE Dnumber=Dno AND Salary>40000 AND (SELECT Dno FROM EMPLOYEE GROUP BY Dno HAVING COUNT (*) > 5)

VIEWS

- A view in SQL terminology is a single table that is derived from other tables
- These other tables could be base tables or previously defined views. A view does not necessarily exist in physical form; it is considered a **virtual table**
- view is a way of specifying a table that we need to reference frequently, even though it may not exist physically.

Specification of Views 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**.**
- **Eg) VIEW V1**

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 View 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 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 'ProjectX', we can utilize the WORKS_ON1 view and specify the query as in QV1:

```
SELECT    Fname, Lname
FROM      WORKS_ON1
WHERE     Pname='ProjectX';
```

- 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 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 any more, we can use the **DROP VIEW** command to dispose of it .

Eg) **DROP VIEW** WORKS_ON1;

View Implementation, View Update and Inline Views

- The problem of efficiently implementing a view for querying is complex. For that, two main approaches have been suggested.

  1) **query modification** **2) view materialization**

1) query modification involves modifying the view query into a query on the underlying base tables.

- The previous query V1 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';
```

- The disadvantage of this approach is that the views defined via complex queries are time-consuming to execute, especially if multiple queries are going to be applied to the same view within a short period of time.

2) View Materialization involves physically creating a temporary view table when the view is first queried and keeping that table on the assumption that other queries on the view will follow.

- Here the view tables are updated automatically in accordance with the updations in base tables. Thus keep the view up-to-date.
- An incremental update technique is used for this purpose where the DBMS can determine what new tuples must be inserted, deleted or modified in a materialized view table, when a database update is applied to one of the defining base tables.
- The view is kept as a materialized(physically stored)table as long as it is being queried.

View Update

- Updating of views is complicated and can be ambiguous.
- Consider the WORKS_ON1 view, and suppose that we issue the command to update the PNAME attribute of 'John Smith' from 'ProductX' to 'ProductY'.
- This view update is shown in UV1

```
UV1:  UPDATE WORKS_ON1
      SET    Pname = 'ProductY'
      WHERE  Lname='Smith' AND Fname='John'
             AND Pname='ProductX';
```

UPDATING VIEWS

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is **not** met, then we will not be allowed to update the view.

- The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
- The view must include the PRIMARY KEY of the table based upon which the view has been created.
- The SELECT statement should not have the DISTINCT keyword.
- The View should have all NOT NULL values.
- The view should not be created using nested queries or complex queries.
- The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

Database system uses one of the three ways to keep the materialized view updated:

- Update the materialized view as soon as the relation on which it is defined is updated.
- Update the materialized view every time the view is accessed.
- Update the materialized view periodically.

in-line view

- It is also possible to define a view table in the FROM clause of an SQL query. This is known as an in-line view. In this case, the view is defined within the query itself.

Uses of a View :

A good database should contain views due to the given reasons:

- **Restricting data access –**
  Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table.
- **Hiding data complexity –**
  A view can hide the complexity that exists in a multiple table join.
- **Simplify commands for the user –**
  Views allows the user to select information from multiple tables without requiring the users to actually know how to perform a join.
- **Store complex queries –**
  Views can be used to store complex queries.
- **Rename Columns –**
  Views can also be used to rename the columns without affecting the base tables provided the number of columns in view must match the number of columns specified in select statement. Thus, renaming helps to to hide the names of the columns of the base tables.
- **Multiple view facility –**
  Different views can be created on the same table for different users.

ASSERTIONS

- Assertions can be used to specify additional types of constraints that are outside the scope of the built-in relational model constraints (primary key & unique keys, entity integrity and referential integrity).

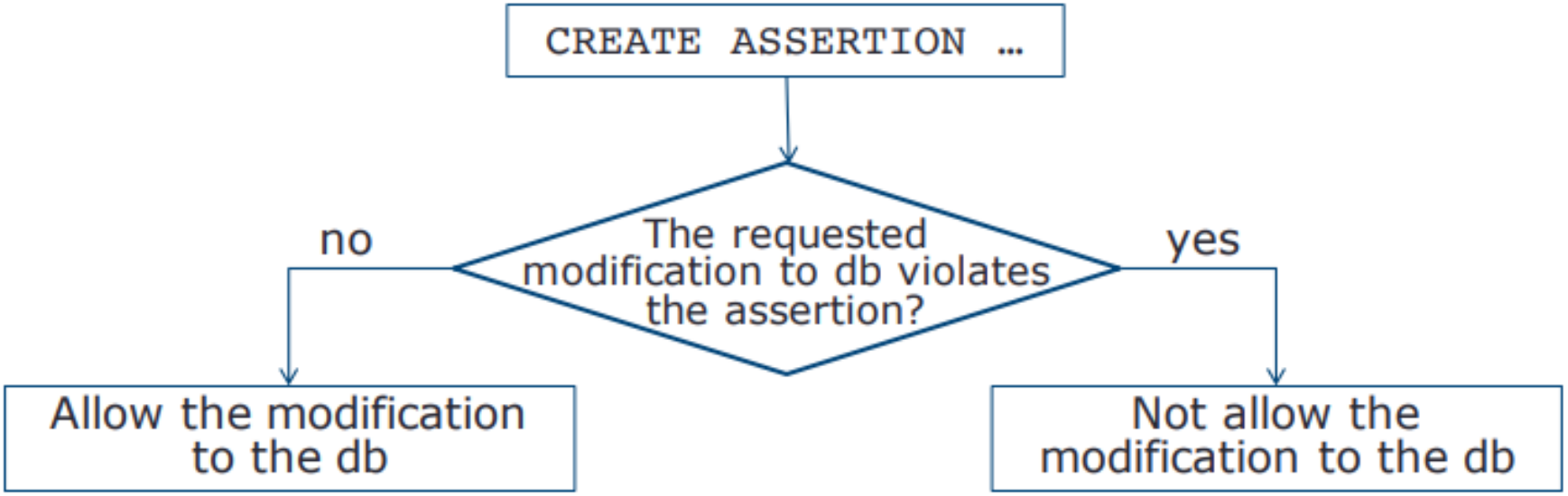
Specifying General Constraints as Assertions in SQL

- In SQL, we can specify general constraints through declarative assertions, using CREATE ASSERTION statement.
- Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.
- The basic technique for writing assertions is to specify a query that selects any tuples that violate the desired condition.

General form of Assertion

```
CREATE ASSERTION <assertion-name>
CHECK <predicate>;
```

```
DROP ASSERTION <assertion-name>
```



Q)The total length of all movies by a given studio shall not exceed 10,000 minutes.

```
CREATE ASSERTION sumLength
CHECK (10000 >= ALL
            (SELECT     SUM(length)
             FROM       Movies
             GROUP BY   studioName ) )
```

- Since this constraint involves only the relation Movies, it can be expressed as a tuple-based **CHECK** constraint.

```
CHECK (10000 >= ALL
            (SELECT     SUM(length)
             FROM       Movies
             GROUP BY   studioName ) )
```

Eg) The salaries of an employee must not be greater than the salary of the manager of the corresponding department

```
CREATE ASSERTION Salary_check CHECK( NOT EXISTS
(SELECT * FROM Employee e, Employee m, Department d
WHERE e.salary>m.salary AND e.Dno= d.Dunmber AND
d.mgr_ssn=m.ssn));
```

Note: The **EXISTS** operator is used to test for the existence of any record in a subquery. The **NOT EXISTS in SQL** Server will check the Subquery for rows existence, and if there are **no** rows then it will return TRUE, otherwise FALSE.

TRIGGERS

- A trigger is a statement that the system executes automatically as a side effect of a modification to the database.
- A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs.
- Action to be taken when certain events occur and when certain conditions are satisfied.
- For example,
  - a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.
  - it may be useful to specify a condition that, if violated, causes some user to be informed of the violation

Benefits of Triggers

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Types of Triggers

- **Row Trigger :** Trigger is fired each time a row in the table is affected
- **Statement Trigger :** Trigger is fired even if no rows are affected
- **Before / After Trigger :** These triggers apply to both row and statement trigger which is to be specify trigger timing.

Syntax: Trigger

```
create trigger [trigger_name]
[before | after]
{insert | update | delete}
on [table_name]
[for each row]
[trigger_body]
```

[trigger_name]: Creates or replaces an existing trigger with the trigger_name.

[before | after]: This specifies when the trigger will be executed.

{insert | update | delete}: This specifies the DML operation.

on [table_name]: This specifies the name of the table associated with the trigger.

[for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.

[trigger_body]: This provides the operation to be performed as trigger is fired

Creating Triggers (complete syntax)

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
   Declaration-statements
BEGIN
   Executable-statements
EXCEPTION
   Exception-handling-statements
END;
```

- **CREATE [OR REPLACE] TRIGGER trigger_name**
  - Creates or replaces an existing trigger with the trigger_name.
- **{BEFORE | AFTER }**
  - This specifies when the trigger will be executed. (before or after the triggering statement)
- **{INSERT [OR] | UPDATE [OR] | DELETE}**
  - This specifies the DML operation.
- **[OF col_name]**
  - This specifies the column name that will be updated.
- **[ON table_name]**
  - This specifies the name of the table associated with the trigger.

- **[REFERENCING OLD AS o NEW AS n]**
  - This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- **[FOR EACH ROW]**
  - This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
  - Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- **WHEN (condition)**
  - This provides a condition for rows for which the trigger would fire.
  - This clause is valid only for row-level triggers.

Example: Given Student Report Database, in which student marks assessment is recorded. In such schema, create a trigger so that the total and average of specified marks is automatically inserted whenever a record is insert.

Note: Here, as trigger will invoke before record is inserted so, BEFORE tag can be used.

create trigger stud_marks **before INSERT on** Student **for each row set** Student.total = Student.subj1 + Student.subj2 + Student.subj3, Student.per = Student.total * 60 / 100;

- Above SQL statement will create a trigger in the student database in which whenever subjects marks are entered, before inserting this data into the database, trigger will compute those two values and insert with the entered values.

Student

```
+-----+-------+-------+-------+-------+-------+------+
| tid | name  | subj1 | subj2 | subj3 | total | per  |
+-----+-------+-------+-------+-------+-------+------+
```

mysql> insert into Student values(100, "ABCDE", 20, 20, 20, 0, 0);

```
mysql> select * from Student;
+-----+-------+-------+-------+-------+-------+------+
| tid | name  | subj1 | subj2 | subj3 | total | per  |
+-----+-------+-------+-------+-------+-------+------+
| 100 | ABCDE |    20 |    20 |    20 |    60 |   36 |
+-----+-------+-------+-------+-------+-------+------+
```

Example

- Create a row-level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table.
- This trigger will display the salary difference between the old values and new values.

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff := :NEW.salary  - :OLD.salary;
   dbms_output.put_line('Old salary: ' || :OLD.salary);
   dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```