VIT | Vidyalankar Institute of Technology
Accredited A+ by NAAC

Structured Query Language (SQL)

Prof. Suja Jayachandran

Assistant Professor
Computer Engineering Department
Vidyalankar Institute of Technology

Overview of SQL

- SQL is a standard language for accessing and manipulating databases.
- **What is SQL?**

➢SQL stands for Structured Query Language

➢SQL lets you access and manipulate databases

➢SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987

- **What Can SQL do?**

➢SQL can execute queries against a database

➢SQL can retrieve data from a database

➢SQL can insert records in a database

➢SQL can update records in a database

➢SQL can delete records from a database

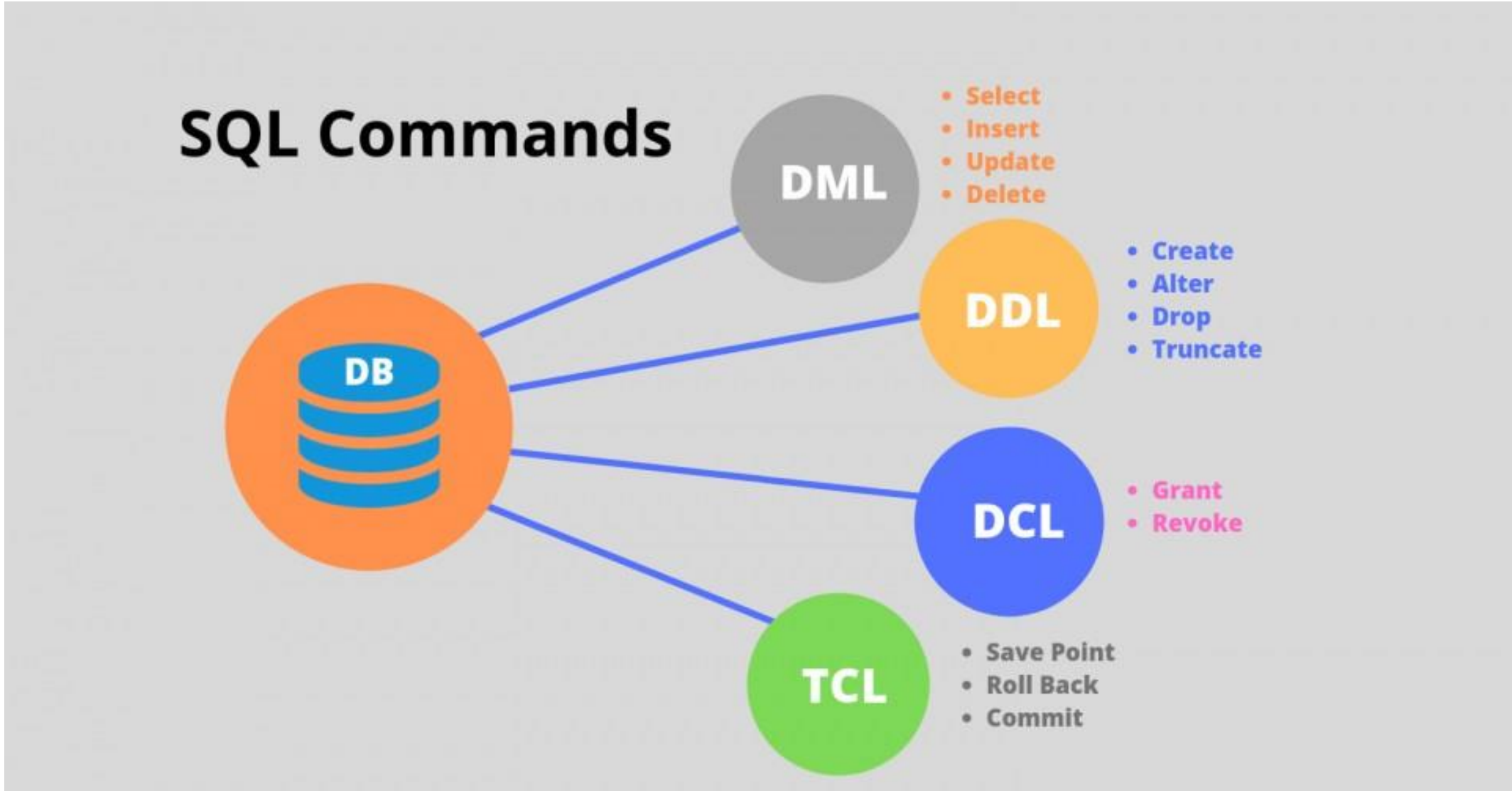
➢SQL can create new databases

➢SQL can create new tables in a database

➢SQL can create stored procedures in a database

➢SQL can create views in a database

➢SQL can set permissions on tables, procedures, and views


SQL Commands
DB
DML
Select
Insert
Update
Delete
DDL
Create
Alter
Drop
Truncate
DCL
Grant
Revoke
TCL
Save Point
Roll Back
Commit

DDL Commands

- DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema.
- It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.
- Examples of DDL commands:

1. CREATE – is used to create the database or its objects (like table, index, function, views, store procedure and triggers).
2. DROP – is used to delete objects from the database.
3. ALTER-is used to alter the structure of the database.
4. TRUNCATE–is used to remove all records from a table, including all spaces allocated for the records are removed.

create

- Syntax:

```
CREATE TABLE TABLE_NAME (COLUMN_NAME DATATYPES[,....]);
```

- Example:

1.CREATE TABLE EMPLOYEE(Name VARCHAR2(20), Email VARCHAR2(100), DOB DATE);

2. CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
);

Data type	Description
CHARACTER(n)	Character string. Fixed-length n
VARCHAR(n) or CHARACTER VARYING(n)	Character string. Variable length. Maximum length n
BINARY(n)	Binary string. Fixed-length n
BOOLEAN	Stores TRUE or FALSE values
VARBINARY(n) or BINARY VARYING(n)	Binary string. Variable length. Maximum length n
INTEGER(p)	Integer numerical (no decimal). Precision p
SMALLINT	Integer numerical (no decimal). Precision 5
INTEGER	Integer numerical (no decimal). Precision 10
BIGINT	Integer numerical (no decimal). Precision 19
DECIMAL(p,s)	Exact numerical, precision p, scale s. Example: decimal(5,2) is a number that has 3 digits before the decimal and 2 digits after the decimal
NUMERIC(p,s)	Exact numerical, precision p, scale s. (Same as DECIMAL)
FLOAT(p)	Approximate numerical, mantissa precision p. A floating number in base 10 exponential notation.
REAL	Approximate numerical, mantissa precision 7
FLOAT	Approximate numerical, mantissa precision 16
DATE	Stores year, month, and day values
TIME	Stores hour, minute, and second values
TIMESTAMP	Stores year, month, day, hour, minute, and second values
INTERVAL	Composed of a number of integer fields, representing a period of time, depending on the type of interval

A blob is a data type that can store binary data. This is different than most other data types used in databases, such as integers, floating point numbers, characters, and strings, which store letters and numbers. Since blobs can store binary data, they can be used to store images or other multimedia files. blobs are used to store objects such as images, audio files, and video clips, they often require significantly more space than other data types. The amount of data a blob can store varies depending on the database type, but some databases allow blob sizes of several gigabytes.

drop

- DROP TABLE removes tables from the database. Only its owner may destroy a table. To empty a table of rows without destroying the table, use DELETE or TRUNCATE.
- DROP TABLE always removes any indexes, rules, triggers, and constraints that exist for the target table. However, to drop a table that is referenced by a view or a foreign-key constraint of another table, CASCADE must be specified. (CASCADE will remove a dependent view entirely, but in the foreign-key case it will only remove the foreign-key constraint, not the other table entirely.)
- Parameters

Syntax: DROP TABLE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]

1.IF EXISTS

Do not throw an error if the table does not exist. A notice is issued in this case.

2.name

Example: DROP TABLE Employee,Department;

The name (optionally schema-qualified) of the table to drop.

3.CASCADE

Automatically drop objects that depend on the table (such as views).

4.RESTRICT

Refuse to drop the table if any objects depend on it. This is the default

alter

- ALTER TABLE -- change the definition of a table

Syntax: **ALTER TABLE** table_name **action**;

ALTER TABLE changes the definition of an existing table. There are several subforms:

1. **ALTER TABLE** table_name **ADD COLUMN** new_column_name **TYPE**;
2. **ALTER TABLE** table_name **DROP COLUMN** column_name;
3. **ALTER TABLE** table_name **RENAME COLUMN** column_name **TO** new_column_name;
4. **ALTER TABLE** table_name **ALTER COLUMN** column_name [**SET DEFAULT value** | **DROP DEFAULT**];
5. **ALTER TABLE** table_name **ALTER COLUMN** column_name [**SET NOT** NULL| **DROP NOT** NULL];
6. **ALTER TABLE** table_name **ADD CHECK** expression;
7. **ALTER TABLE** table_name **ADD CONSTRAINT** constraint_name constraint_definition;
8. **ALTER TABLE** table_name **RENAME TO** new_table_name;

Example:
ALTER TABLE STU_DETAILS ADD(ADDRESS VARCHAR2(20));

ALTER TABLE customer_groups
RENAME COLUMN name TO group_name;

truncate

- **TRUNCATE TABLE** statement to remove all data from large tables.
- To remove all data from a table, you use the DELETE statement. However, for a large table, it is more efficient to use the TRUNCATE TABLE statement.
- The TRUNCATE TABLE statement removes all rows from a table without scanning it. This is the reason why it is faster than the DELETE statement.

Syntax: **TRUNCATE TABLE** table_name;

- Example:

TRUNCATE TABLE invoices, customers;

TRUNCATE TABLE invoices **CASCADE**;

DML Commands

- The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements.
- Examples of DML Commands are:

1. INSERT – is used to insert data into a table.
2. UPDATE – is used to update existing data within a table.
3. DELETE – is used to delete records from a database table.
4. SELECT- is used to retrieve data from the a database.

insert

- Syntax:

```
insert into tablename values (value1,value2..);
insert into tablename(attribute1,attribute2) values(value1,value2);
```

- Example:

CREATE TABLE products (product_no integer, name text , price numeric);

INSERT INTO products VALUES (1, 'Cheese', 9.99);

The above syntax has the drawback that you need to know the order of the columns in the table. To avoid that you can also list the columns explicitly.

For example

INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);

INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);

If you don't have values for all the columns, you can omit some of them. In that case, the columns will be filled with their default values. For example

INSERT INTO products (product_no, name) VALUES (1, 'Cheese');

INSERT INTO products VALUES (1, 'Cheese');

update

- Syntax

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Example:

```
UPDATE Customers
SET ContactName = ’John', City= ’Mumbai'
WHERE CustomerID = 1;
```

delete

- DELETE :deletes rows that satisfy the WHERE clause from the specified table. If the WHERE clause is absent, the effect is to delete all rows in the table. The result is a valid, but empty table.
- Syntax:

```
DELETE FROM table_name WHERE condition;
DELETE FROM table_name;
```

Example: DELETE FROM Customers WHERE CustomerName='John';

DROP	DELETE	TRUNCATE
Drop Command is a DDL Command	DELETE Command is a DML Command	TRUNCATE Command is a DDL Command
It removes the whole table from the database	It removes all/single/multiple rows from the table	It removes all the rows exist in the table
WHERE clause cannot be used with DROP command	WHERE clause can be used with DELETE command	WHERE clause cannot be used with TRUNCATE command
Rollback cannot be possible	Rollback can be possible	Rollback cannot be possible
Commit cannot be possible	Commit can be possible	Commit cannot be possible
No Trigger will be fired	Trigger will be fired	No Trigger will be fired
Faster and Time saving	Slowest and Time consuming	Faster and Time saving
Syntax DROP TABLE table_name;	**Syntax** DELETE TABLE table_name; COMMIT;	**Syntax** TRUNCATE TABLE table_name;

select

- Select is the most commonly used statement in SQL. The SELECT Statement in SQL is used to retrieve or fetch data from a database. We can fetch either the entire table or according to some specified rules. The data returned is stored in a result table. This result table is also called result-set.

Syntax:SELECT column1,column2 FROM table_name
column1 , column2: names of the fields of the table
table_name: from where we want to fetch
Example:SELECT * FROM table_name;

- **The SQL SELECT DISTINCT Statement**
- The SELECT DISTINCT statement is used to return only distinct (different) values.
- Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.
- **SELECT DISTINCT Syntax**
- SELECT DISTINCT *column1*, *column2, ...*
  FROM *table_name*;
- **The SQL AND, OR and NOT Operators**
- The WHERE clause can be combined with AND, OR, and NOT operators.
- The AND and OR operators are used to filter records based on more than one condition:
- The AND operator displays a record if all the conditions separated by AND are TRUE.
- The OR operator displays a record if any of the conditions separated by OR is TRUE.
- The NOT operator displays a record if the condition(s) is NOT TRUE.
- **AND Syntax**
- SELECT *column1*, *column2, ...*
  FROM *table_name*
  WHERE *condition1* AND *condition2* AND *condition3 ...*;

- **OR Syntax**
- SELECT *column1*, *column2, ...*
  FROM *table_name*
  WHERE *condition1* OR *condition2* OR *condition3 ...*;
- **NOT Syntax**
- SELECT *column1*, *column2, ...*
  FROM *table_name*
  WHERE NOT *condition*;
- **The SQL SELECT TOP Clause**
- The SELECT TOP clause is used to specify the number of records to return.

The SELECT TOP clause is useful on large tables with thousands of records.Returning a large number of records can impact performance.

- SELECT *column_name(s)*
  FROM *table_name*
  WHERE *condition*
  LIMIT *number*;
- **SQL Dates**
- SELECT * FROM Orders WHERE OrderDate='2008-11-11'

Set operations in SQL

1. UNION
2. UNION ALL
3. INTERSECT
4. MINUS/EXCEPT

1.UNION

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

First

ID	Name
1	Ajay
2	Vijay

Second

ID	Name
2	Vijay
3	Ramesh

ID	Name
1	Ajay
2	Vijay
3	Ramesh

2.UNION ALL

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

ID	Name
1	Ajay
2	Vijay
2	Vijay
3	Ramesh

3.INTERSECT

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

ID	Name
2	Vijay

4.MINUS/EXCEPT

```
SELECT * FROM First
EXCEPT
SELECT * FROM Second;
```

ID	Name
1	Ajay

String Operation

- The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.
- There are two wildcards often used in conjunction with the LIKE operator:
- % - The percent sign represents zero, one, or multiple characters
- _ - The underscore represents a single character
- SELECT *column1, column2, ...*
  FROM *table_name*
  WHERE *columnN* LIKE *pattern*;

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that start with "a"
WHERE CustomerName LIKE '%a'	Finds any values that end with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%'	Finds any values that start with "a" and are at least 2 characters in length
WHERE CustomerName LIKE 'a__%'	Finds any values that start with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that start with "a" and ends with "o"

- The **BETWEEN operator** selects values within a given range. The values can be numbers, text, or dates.The BETWEEN operator is inclusive: begin and end values are included.

Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

- select F_NAME , L_NAME
  from EMPLOYEES
  where ADDRESS LIKE '%Elgin,IL%’ ;

- select F_NAME , L_NAME
from EMPLOYEES
where B_DATE LIKE '197%’ ;

- select *
from EMPLOYEES
where (SALARY BETWEEN 60000 and 70000) and DEP_ID = 5 ;

Aggregate Function

- Min():The MIN() function returns the smallest value of the selected column.
- Max():The MAX() function returns the largest value of the selected column.
- Count():The COUNT() function returns the number of rows that matches a specified criterion.
- Sum():The SUM() function returns the total sum of a numeric column.
- Avg() :The AVG() function returns the average value of a numeric column.

- SELECT MIN(*column_name*)
  FROM *table_name*
  WHERE *condition*;
- SELECT MAX(*column_name*)
  FROM *table_name*
  WHERE *condition*;
- SELECT COUNT(*column_name*)
  FROM *table_name*
  WHERE *condition*;
- SELECT AVG(*column_name*)
  FROM *table_name*
  WHERE *condition*;
- SELECT SUM(*column_name*)
  FROM *table_name*
  WHERE *condition*;

Group by and Having clause

- The GROUP BY statement groups rows that have the same values into summary rows.
- The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.
- **GROUP BY Syntax**
- SELECT *column_name(s)*
  FROM *table_name*
  WHERE *condition*
  GROUP BY *column_name(s)*
  ORDER BY *column_name(s);*
- SELECT COUNT(CustomerID), Country
  FROM Customers
  GROUP BY Country
  ORDER BY COUNT(CustomerID) DESC;

- The **HAVING clause** was added to SQL because the WHERE keyword could not be used with aggregate functions.
- SELECT *column_name(s)*
  FROM *table_name*
  WHERE *condition*
  GROUP BY *column_name(s)*
  HAVING *condition*
  ORDER BY *column_name(s);*
- The following SQL statement lists the number of customers in each country, sorted high to low (Only include countries with more than 5 customers):
- SELECT COUNT(CustomerID), Country
  FROM Customers
  GROUP BY Country
  HAVING COUNT(CustomerID) > 5
  ORDER BY COUNT(CustomerID) DESC;

- Examples

➢select DEP_ID, COUNT(*), AVG(SALARY)
from EMPLOYEES
group by DEP_ID;

➢select DEP_ID, COUNT(*) AS "NUM_EMPLOYEES", AVG(SALARY) AS "AVG_SALARY"
from EMPLOYEES
group by DEP_ID;

➢select DEP_ID, COUNT(*) AS "NUM_EMPLOYEES", AVG(SALARY) AS "AVG_SALARY"
from EMPLOYEES
group by DEP_ID
having count(*) < 4
order by AVG_SALARY;

Views in SQL

- In SQL, a view is a virtual table based on the result-set of an SQL statement. 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.
- **CREATE VIEW Syntax**
- CREATE VIEW *view_name* AS
  SELECT *column1*, *column2*, ...
  FROM *table_name*
  WHERE *condition*;
- **Example**
- The following SQL creates a view that selects every product in the "Products" table with a price higher than the average price:

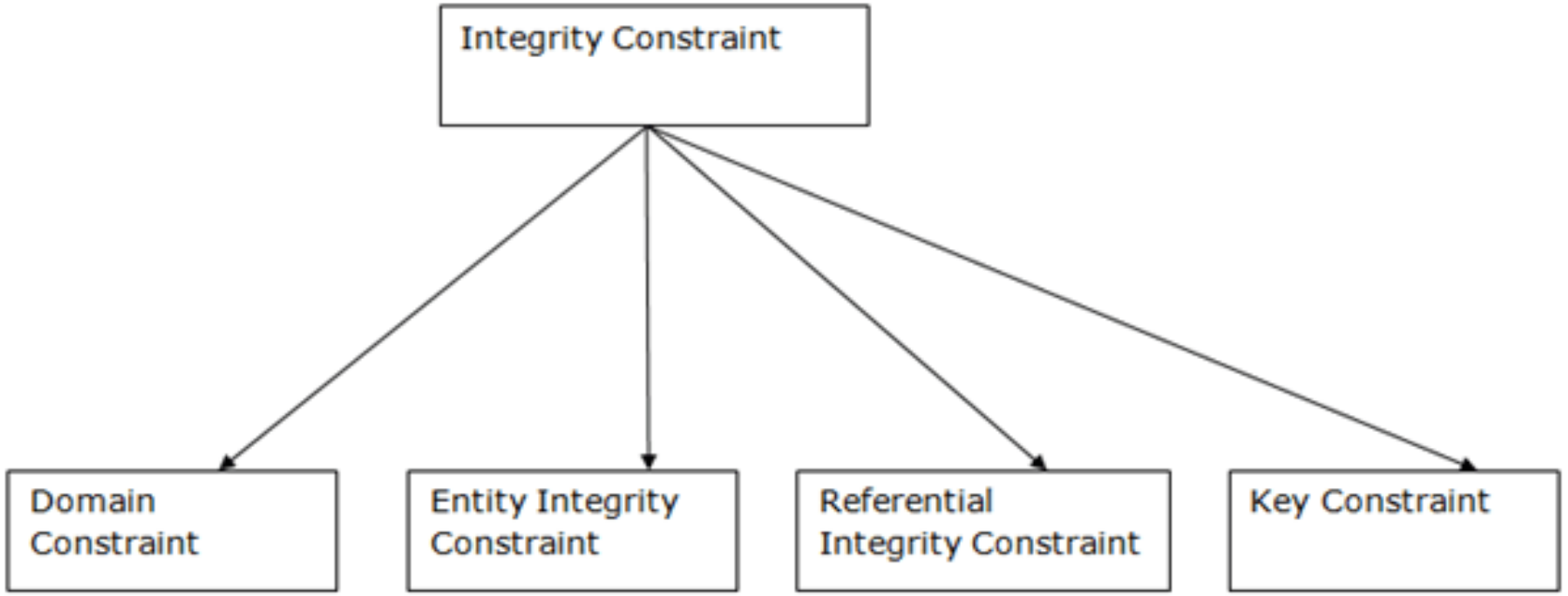
```
CREATE VIEW [Products Above Average Price] AS
SELECT ProductName, Price
FROM Products
WHERE Price > (SELECT AVG(Price) FROM Products);
```

To see the output:

```
SELECT * FROM [Products Above Average Price];
```

- **SQL CREATE OR REPLACE VIEW Syntax**
- CREATE OR REPLACE VIEW *view_name* AS
  SELECT *column1*, *column2*, ...
  FROM *table_name*
  WHERE *condition*;
- **SQL DROP VIEW Syntax**
- DROP VIEW *view_name*;

- Constraints in DBMS



Integrity Constraints: used to apply business rules for the database tables.

- NOT NULL Constraint − Ensures that a column cannot have NULL value.
- DEFAULT Constraint − Provides a default value for a column when none is specified.
- UNIQUE Constraint − Ensures that all values in a column are different.
- PRIMARY Key − Uniquely identifies each row/record in a database table.
- FOREIGN Key − Uniquely identifies a row/record in any of the given database table.
- CHECK Constraint − The CHECK constraint ensures that all the values in a column satisfies certain conditions.

ALTER TABLE EMPLOYEES DROP CONSTRAINT EMPLOYEES_PK;

- CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int
  );
- ALTER TABLE Persons
  MODIFY Age int NOT NULL;
- CREATE TABLE Persons (
    ID int NOT NULL UNIQUE,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
  );
- CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    PRIMARY KEY (ID)
  );

- CREATE TABLE Orders (
  OrderID int NOT NULL,
  OrderNumber int NOT NULL,
  PersonID int,
  PRIMARY KEY (OrderID),
  FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
  );
- CREATE TABLE Persons (
  ID int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Age int,
  CHECK (Age>=18)
  );
- CREATE TABLE Persons (
  ID int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Age int,
  City varchar(255) DEFAULT ‘Mumbai'
  );
- CREATE INDEX *index_name*
  ON *table_name* (*column1*, *column2*, ...);

- **Domain constraints** can be defined as the definition of a valid set of values for an attribute. The data type of **domain** includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding **domain**.

ID	NAME	SEMENSTER	AGE
1000	Tom	1st	17
1001	Johnson	2nd	24
1002	Leonardo	5th	21
1003	Kate	3rd	19
1004	Morgan	8th	A

Not allowed. Because AGE is an integer attribute

- **Entity integrity constraints**

The entity integrity constraint states that primary key value can't be null.

This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.

A table can contain a null value other than the primary key field.

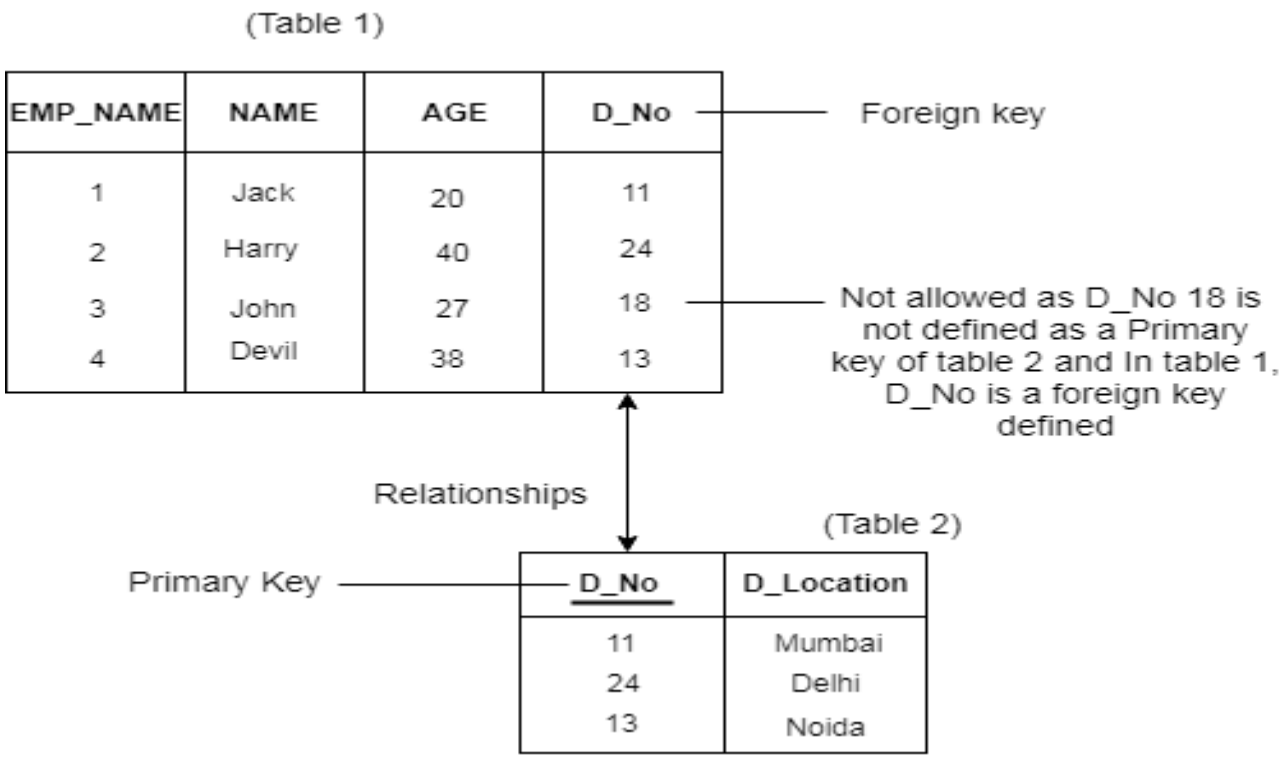
Example:

EMPLOYEE

EMP_ID	EMP_NAME	SALARY
123	Jack	30000
142	Harry	60000
164	John	20000
	Jackson	27000

Not allowed as primary key can’t contain a NULL value

- **Referential Integrity Constraints**
- A referential integrity constraint is specified between two tables.
- In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.



- Create Foreign key with cascade delete - Using CREATE TABLE statement
- Syntax

- The syntax for creating a foreign key with cascade delete using a CREATE TABLE statement:

- CREATE TABLE child_table

```
(
  column1 datatype [ NULL | NOT NULL ],
  column2 datatype [ NULL | NOT NULL ],
  ...

  CONSTRAINT fk_name
    FOREIGN KEY (child_col1, child_col2, ... child_col_n)
    REFERENCES parent_table (parent_col1, parent_col2, ... parent_col_n)
    ON DELETE CASCADE
    [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
);
```

- The syntax for creating a foreign key with cascade delete using an ALTER TABLE statement is:

```
ALTER TABLE child_table
ADD CONSTRAINT fk_name
    FOREIGN KEY (child_col1, child_col2, ... child_col_n)
    REFERENCES parent_table (parent_col1, parent_col2, ... parent_col_n)
    ON DELETE CASCADE;
```

- **Key constraints**
- Keys are the entity set that is used to identify an entity within its entity set uniquely.
- An entity set can have multiple keys, but out of which one key will be the primary key. A primary key can contain a unique and null value in the relational table.

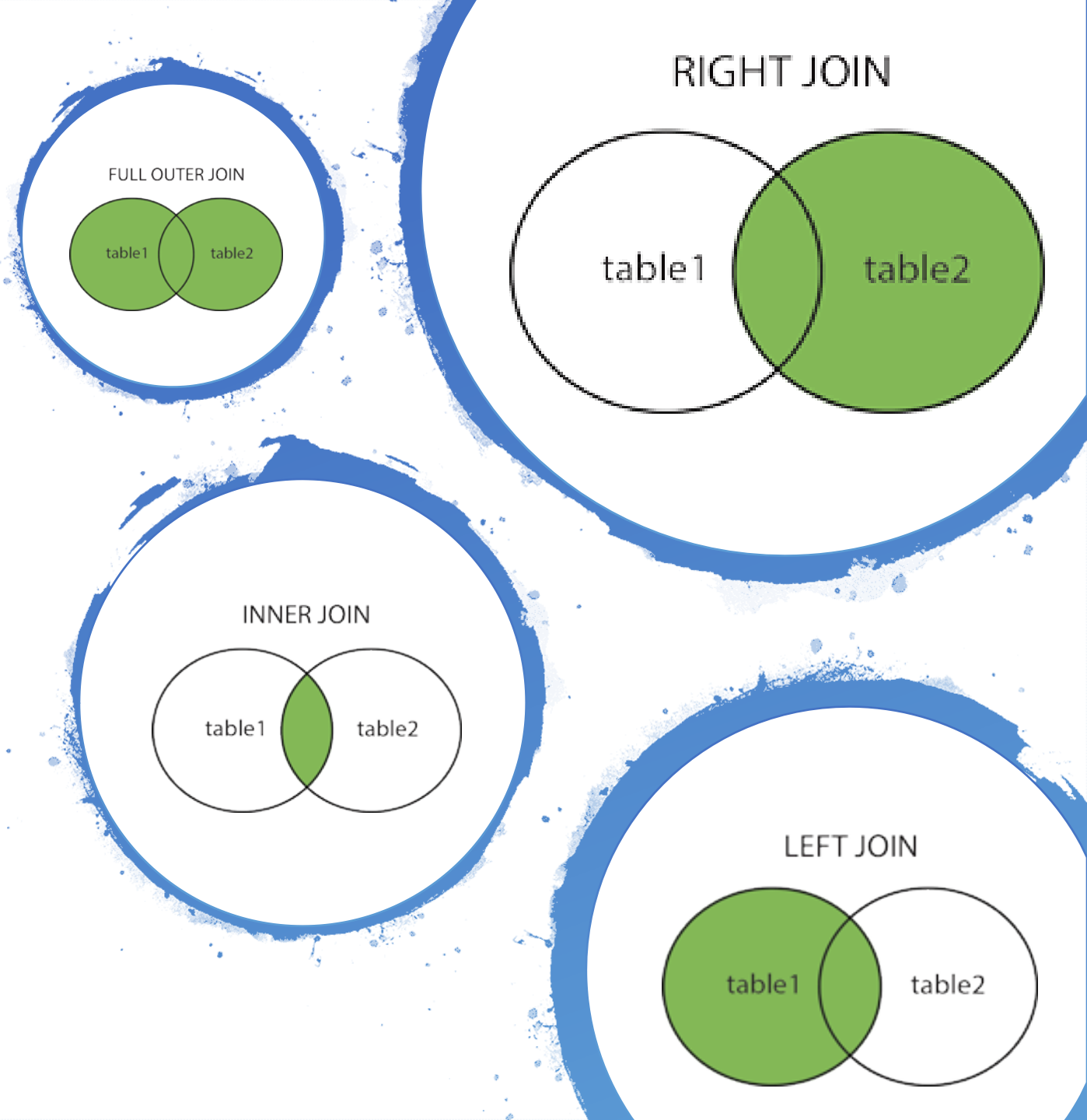
ID	NAME	SEMENSTER	AGE
1000	Tom	1st	17
1001	Johnson	2nd	24
1002	Leonardo	5th	21
1003	Kate	3rd	19
1002	Morgan	8th	22

Not allowed. Because all row must be unique

Join

- A JOIN clause is used to combine rows from two or more tables, based on a related column between them.
- Here are the different types of the JOINs in SQL:

- (INNER) JOIN: Returns records that have matching values in both tables
- LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table
- RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table
- FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table



- INNER JOIN Syntax
- ```
  SELECT column_name(s)
  FROM table1
  INNER JOIN table2
  ON table1.column_name = table2.column_name;
  ```

- LEFT JOIN Syntax
- ```
  SELECT column_name(s)
  FROM table1
  LEFT JOIN table2
  ON table1.column_name = table2.column_name;
  ```

- RIGHT JOIN Syntax
- ```
  SELECT column_name(s)
  FROM table1
  RIGHT JOIN table2
  ON table1.column_name = table2.column_name;
  ```

- FULL OUTER JOIN Syntax
- ```
  SELECT column_name(s)
  FROM table1
  FULL OUTER JOIN table2
  ON table1.column_name = table2.column_name
  WHERE condition;
  ```

Views

- Views in kind SQL are of virtual tables.
- A view also has rows and columns as they are in a real table in the database.
- We can create a view by selecting fields from one or more tables present in the database
- A View can either have all the rows of a table or specific rows based on certain condition

Student Details

S_ID	NAME	ADDRESS
1	Harsh	Kolkata
2	Ashish	Durgapur
3	Pratik	Delhi
4	Dhanraj	Bihar
5	Ram	Rajasthan

Student marks

ID	NAME	MARKS	AGE
1	Harsh	90	19
2	Suresh	50	20
3	Pratik	80	19
4	Dhanraj	95	21
5	Ram	85	18

CREATE VIEW

CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE condition;

Example:

CREATE VIEW DetailsView AS
SELECT NAME, ADDRESS
FROM StudentDetails
WHERE S_ID < 5

SELECT * FROM DetailsView;

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

- CREATE VIEW StudentNames AS

  SELECT S_ID, NAME

  FROM StudentDetails

  ORDER BY NAME;

S_ID	NAMES
2	Ashish
4	Dhanraj
1	Harsh
3	Pratik
5	Ram

- **Creating View from multiple tables**: create a View named MarksView from two tables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the select statement

```
CREATE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS,
StudentMarks.MARKS
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

NAME	ADDRESS	MARKS
Harsh	Kolkata	90
Pratik	Delhi	80
Dhanraj	Bihar	95
Ram	Rajasthan	85

DELETING VIEWS

DROP VIEW view_name;

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.

1. The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
2. The SELECT statement should not have the DISTINCT keyword.
3. The View should have all NOT NULL values.
4. The view should not be created using nested queries or complex queries.
5. 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

- **CREATE OR REPLACE VIEW** statement to add or remove fields from a view.
- Syntax:

CREATE OR REPLACE VIEW view_name AS

SELECT column1,coulmn2,..

FROM table_name

WHERE condition;

Example

CREATE OR REPLACE VIEW MarksView AS

SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS, StudentMarks.AGE

FROM StudentDetails, StudentMarks

WHERE StudentDetails.NAME = StudentMarks.NAME;

- Inserting a row in a view:
- We can insert a row in a View in a same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a View.Syntax:
- INSERT INTO view_name(column1, column2 , column3,..)

  VALUES(value1, value2, value3..);

Example

```
INSERT INTO DetailsView(NAME, ADDRESS)
VALUES("Suresh","Gurgaon");
```

- **Deleting a row from a View**
- DELETE FROM view_name

  WHERE condition;

Example:DELETE FROM DetailsView

WHERE NAME="Suresh";

CHECK OPTION

- The WITH CHECK OPTION clause in SQL is a very useful clause for views. It is applicable to a updatable view. If the view is not updatable, then there is no meaning of including this clause in the CREATE VIEW statement.
- The WITH CHECK OPTION clause is used to prevent the insertion of rows in the view where the condition in the WHERE clause in CREATE VIEW statement is not satisfied.
- If we have used the WITH CHECK OPTION clause in the CREATE VIEW statement, and if the UPDATE or INSERT clause does not satisfy the conditions then they will return an error.

```
CREATE VIEW SampleView AS
SELECT S_ID, NAME
FROM  StudentDetails
WHERE NAME IS NOT NULL
WITH CHECK OPTION;
```

// insert a new row with null value in the NAME column then it will give an error because the view is created with the condition for NAME column as NOT NULL.

```
INSERT INTO SampleView(S_ID)
VALUES(6);
```

- **Uses of a View** :
  A good database should contain views due to the given reasons:

1. **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.
2. **Hiding data complexity** –
   A view can hide the complexity that exists in a multiple table join.
3. **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.
4. **Store complex queries** –
   Views can be used to store complex queries.
5. **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 hide the names of the columns of the base tables.
6. **Multiple view facility** –
   Different views can be created on the same table for different users.

Nested Queries

- In nested queries, a query is written inside a query. The result of inner query is used in execution of outer query.

STUDENT

S_ID	S_NAME	S_ADDRESS	S_PHONE	S_AGE
S1	RAM	DELHI	9455123451	18
S2	RAMESH	GURGAON	9652431543	18
S3	SUJIT	ROHTAK	9156253131	20
S4	SURESH	DELHI	9156768971	18

COURSE

C_ID	C_NAME
C1	DSA
C2	Programming
C3	DBMS

- Student_Course

S_ID	C_ID
S1	C1
S1	C3
S2	C1
S3	C2
S4	C2
S4	C3

- There are mainly two types of nested queries:
- Independent Nested Queries: In independent nested queries, query execution starts from innermost query to outermost queries. The execution of inner query is independent of outer query, but the result of inner query is used in execution of outer query. Various operators like IN, NOT IN, ANY, ALL etc are used in writing independent nested queries.

- IN: If we want to find out S_ID who are enrolled in C_NAME ‘DSA’ or ‘DBMS’, we can write it with the help of independent nested query and IN operator. From COURSE table, we can find out C_ID for C_NAME ‘DSA’ or DBMS’ and we can use these C_IDs for finding S_IDs from STUDENT_COURSE TABLE.
- STEP 1: Finding C_ID for C_NAME =’DSA’ or ‘DBMS’
- Select C_ID from COURSE where C_NAME = ‘DSA’ or C_NAME = ‘DBMS’
- STEP 2: Using C_ID of step 1 for finding S_ID
- Select S_ID from STUDENT_COURSE where C_ID IN
- (SELECT C_ID from COURSE where C_NAME = ‘DSA’ or C_NAME=’DBMS’);
- The inner query will return a set with members C1 and C3 and outer query will return those S_IDs for which C_ID is equal to any member of set (C1 and C3 in this case). So, it will return S1, S2 and S4.

- If we want to find out names of STUDENTs who have either enrolled in ‘DSA’ or ‘DBMS’, it can be done as:

```
Select S_NAME from STUDENT where S_ID IN
(Select S_ID from STUDENT_COURSE where C_ID IN
(SELECT C_ID from COURSE where C_NAME=’DSA’ or C_NAME=’DBMS’));
```

- NOT IN: If we want to find out S_IDs of STUDENTs who have neither enrolled in ‘DSA’ nor in ‘DBMS’, it can be done as:

```
Select S_ID from STUDENT where S_ID NOT IN
(Select S_ID from STUDENT_COURSE where C_ID IN
(SELECT C_ID from COURSE where C_NAME=’DSA’ or C_NAME=’DBMS’));
```

The innermost query will return a set with members C1 and C3. Second inner query will return those S_IDs for which C_ID is equal to any member of set (C1 and C3 in this case) which are S1, S2 and S4. The outermost query will return those S_IDs where S_ID is not a member of set (S1, S2 and S4). So it will return S3

- Co-related Nested Queries: In co-related nested queries, the output of inner query depends on the row which is being currently executed in outer query. e.g.; If we want to find out S_NAME of STUDENTs who are enrolled in C_ID ‘C1’, it can be done with the help of co-related nested query as:

select S_NAME from STUDENT S where EXISTS

(select * from STUDENT_COURSE SC where S.S_ID=SC.S_ID and SC.C_ID=’C1’);

- For each row of STUDENT S, it will find the rows from STUDENT_COURSE where S.S_ID = SC.S_ID and SC.C_ID=’C1’. If for a S_ID from STUDENT S, atleast a row exists in STUDENT_COURSE SC with C_ID=’C1’, then inner query will return true and corresponding S_ID will be returned as output.

- Trigger: A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.
- Syntax:
- create trigger [trigger_name]

  [before | after]

  {insert | update | delete}

  on [table_name]

  [for each row]

  [trigger_body]

- Explanation of syntax:

1. create trigger [trigger_name]: Creates or replaces an existing trigger with the trigger_name.
2. [before | after]: This specifies when the trigger will be executed.
3. {insert | update | delete}: This specifies the DML operation.
4. on [table_name]: This specifies the name of the table associated with the trigger.
5. [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
6. [trigger_body]: This provides the operation to be performed as trigger is fired

- BEFORE and AFTER of Trigger:
- BEFORE triggers run the trigger action before the triggering statement is run.
- AFTER triggers run the trigger action after the triggering statement is run.

- 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.
- Student(id,name,subj1,subj2,subj3,total,per)

```
create trigger stud_marks
 before INSERT
 on
 Student
 for each row
 set Student.total = Student.subj1 + Student.subj2 + Student.subj3,
Student.per = Student.total * 100 / 300;
```

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.

```
insert into Student values(0, "ABCDE", 70, 60, 60, 0, 0);
```