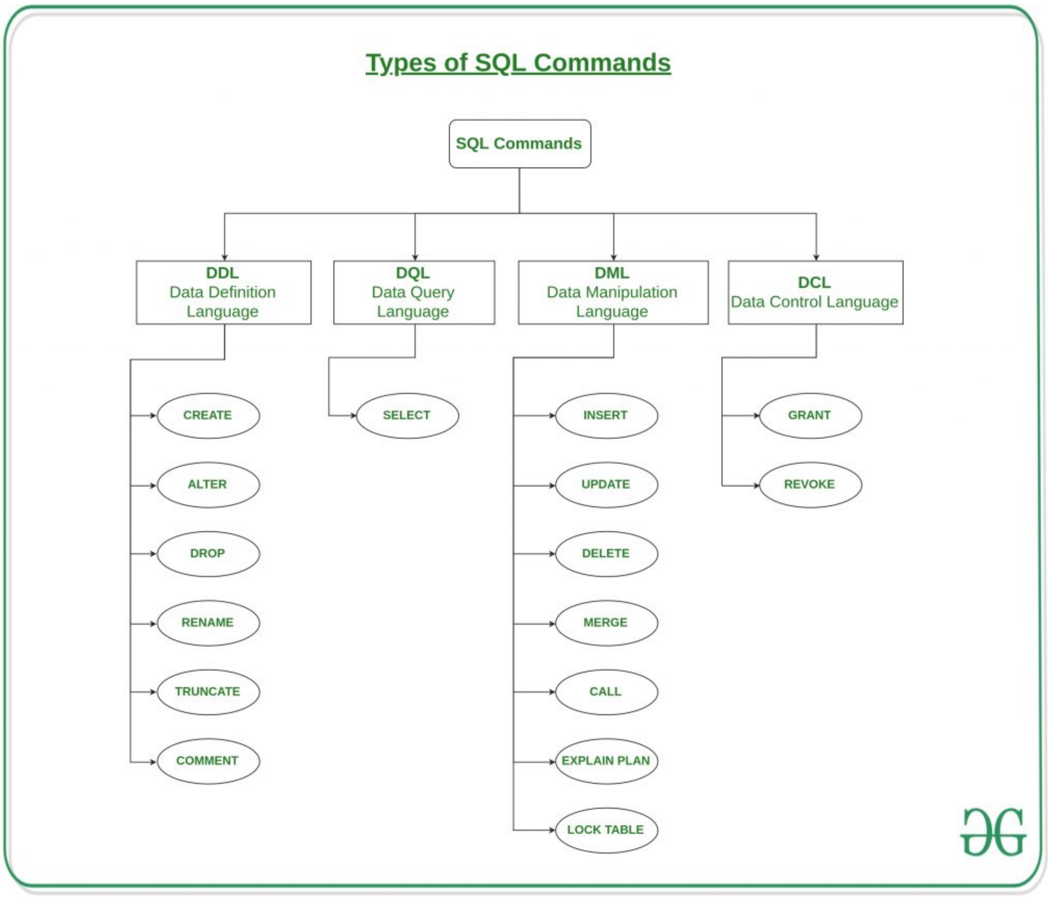
We have combined Classroom and Theory tab and created a new Learn tab for easy access. You can access Classroom and Theory from the left panel.

– DBMS | Introduction to SQL

Structured Query Language is a standard Database language that is used to create, maintain and retrieve the relational database.
SQL Clauses are mainly divided into 4 Categories -

1. DDL - Data Definition Language
2. DQl - Data Query Language
3. DML - Data Manipulation Language
4. DCL - Data Control Language



1. **DDL (Data Definition Language)** :

Data Definition Language is used to define the database structure or schema. DDL is also used to specify additional properties of the data. The storage structure and access methods used by the database system by a set of statements in a special type of DDL called a data storage and definition language. These statements define the implementation details of the database schema, which are usually hidden from the users. The data values stored in the database must satisfy certain consistency constraints.
For example, suppose the university requires that the account balance of a department must never be negative. The DDL provides facilities to specify such constraints. The database system checks these constraints every time the database is updated. In general, a constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to the test. Thus, the database system implements integrity constraints that can be tested with minimal overhead.

1. **Domain Constraints :** A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as the constraints on the values that it can take.
2. **Referential Integrity :** There are cases where we wish to ensure that a value appears in one relation for a given set of attributes also appear in a certain set of attributes in another relation i.e. Referential Integrity. For example, the department listed for each course must be one that actually exists.
3. **Assertions :** An assertion is any condition that the database must always satisfy. Domain constraints and Integrity constraints are special form of assertions.
4. **Authorization :** We may want to differentiate among the users as far as the type of access they are permitted on various data values in database. These differentiation are expressed in terms of Authorization. The most common being :
   read authorization - which allows reading but not modification of data ;
   insert authorization - which allow insertion of new data but not modification of existing data
   update authorization - which allows modification, but not deletion.

Some Commands:

```
CREATE : to create objects in database
ALTER : alters the structure of database
DROP : delete objects from database
RENAME : rename an objects
```

Following SQL DDL-statement defines the department table :

```
create table department
(dept_name  char(20),
  building   char(15),
  budget     numeric(12,2));
```

Execution of the above DDL statement creates the department table with three columns - dept_name, building, and budget; each of which has a specific datatype associated with it.

2. **DQL (Data Query Language) :**

DML statements are used for performing queries on the data within schema objects. The purpose of DQL Command is to get some schema relation based on the query passed to it.

Some Commands :

```
SELECT: retrieve data from the database
```

Example: Perform a query and get the Names of the instructors from the instructor table :

```
select instructor.name
 from instructor
```

3. **DML (Data Manipulation Language) :**

DML statements are used for managing data with in schema objects.
DML are of two types -

1. **Procedural DMLs** : require a user to specify what data are needed and how to get those data.
2. **Declerative DMLs** (also referred as **Non-procedural DMLs**) : require a user to specify what data are needed without specifying how to get those data.

   Declarative DMLs are usually easier to learn and use than procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data.

Some Commands :

```
INSERT: insert data into a table
UPDATE: update existing data within a table
DELETE: deletes all records from a table, space for the records remain
```

Example of SQL query that finds the names of all instructors in the History department :

```
select instructor.name
 from instructor
 where instructor.dept_name = 'History';
```

The query specifies that those rows from the table instructor where the dept_name is History must be retrieved and the name attributes of these rows must be displayed.

4. **DCL (Data Control Language) :**

A Data Control Language is a syntax similar to a computer programming language used to control access to data stored in a database (Authorization). In particular, it is a component of Structured Query Language (SQL).

Examples of DCL commands :

```
GRANT: allow specified users to perform specified tasks.
REVOKE: cancel previously granted or denied permissions.
```

The operations for which privileges may be granted to or revoked from a user or role apply to both the Data definition language (DDL) and the Data manipulation language (DML), and may include CONNECT, SELECT, INSERT, UPDATE, DELETE, EXECUTE and USAGE.

Though many resources claim there to be another category of SQL clauses **TCL - Transaction Control Language**. So we will see in detail about TCL as well.

5. **TCL (Transaction Control Language)** :

Transaction Control Language commands are used to manage transactions in the database. These are used to manage the changes made by DML-statements. It also allows statements to be grouped together into logical transactions.

Examples of TCL commands -

```
COMMIT: Commit command is used to permanently save any transaction
          into the database.
ROLLBACK: This command restores the database to last committed state.
          It is also used with savepoint command to jump to a savepoint
          in a transaction.
SAVEPOINT: Savepoint command is used to temporarily save a transaction so
          that you can rollback to that point whenever necessary.
```

In the Oracle database, executing a DCL command issues an implicit commit. Hence, you cannot roll back the command.

1. **CREATE Clause**

   There are two CREATE statements available in SQL:
   1. CREATE DATABASE
   2. CREATE TABLE

   CREATE DATABASE

   A **Database** is defined as a structured set of data. So, in SQL the very first step to store the data in a well structured manner is to create a database. The **CREATE DATABASE** statement is used to create a new database in SQL.

   Syntax:

   ```
   CREATE DATABASE database_name;

   database_name: name of the database.
   ```

   Example Query: This query will create a new database in SQL and name the database as*university*.

   ```
   CREATE DATABASE university;
   ```

   CREATE TABLE

   We have learned above about creating databases. Now to store the data we need a table to do that. The CREATE TABLE statement is used to create a table in SQL. We know that a table comprises rows and columns. So while creating tables we have to provide all the information to SQL about the names of the columns, type of data to be stored in columns, size of the data, etc. Let us now dive into details on how to use the CREATE TABLE statement to create tables in SQL.

   Syntax:

   ```
   CREATE TABLE table_name
   (
      column1 data_type(size),
      column2 data_type(size),
      column3 data_type(size),
   ....
   );

   table_name:  name of the table.
   column1 name of the first column.
   data_type: Type of data we want to store in the particular column.
             For example,int for integer data.
   size: Size of the data we can store in a particular column. For example, if for
   a column we specify the data_type as int and size as 10 then this column can store an integer
   a number of maximum 10 digits.
   ```

   Example Query: This query will create a table named Students with four columns, ROLL_NO, NAME, ADDRESS, and AGE.

   ```
   CREATE TABLE Student
   (
      ROLL_NO int,
      AGE int,
      NAME varchar(20),
      ADDRESS varchar(20)
   );
   ```

   This query will create a table named Student. The ROLL_NO and AGE field is of type int. The next two columns NAME and ADDRESS are of type varchar and can store characters and the size 20 specifies that these two fields can hold a maximum of 20 characters.

2. **INSERT INTO Clause**

   The INSERT INTO statement of SQL is used to insert a new row in a table. There are two ways of using INSERT INTO statement for inserting rows:
   1. **Only values:** First method is to specify only the value of data to be inserted without the column names.

Syntax

INSERT INTO table_name VALUES (value1, value2, value3,...);
table_name: name of the table.
value1, value2,.. : value of first column, second column,... for the new record

2. **Values with Column Name:** In the second method we will specify both the columns which we want to fill and their corresponding values as shown below:
   Syntax:

   INSERT INTO table_name (column1, column2, column3,..) VALUES (value1, value2, value3,..);
   table_name: name of the table.
   column1: name of first column, second column ...
   value1, value2, value3 : value of first column, second column,... for the new record

Empty Student table After Creation

```
 roll_no | name | address | age
---------+------+---------+-----
(0 rows)
```

Example - Method 1 (Inserting only values) :

```
INSERT INTO Students VALUES ('1','ALEX','NOIDA','19');
```

Student Table will look like this.

```
 roll_no | name | address | age
---------+------+---------+-----
       1 | ALEX | NOIDA   |  19
(1 row)
```

Example - Method 2 (Inserting Values with Column Name) :

```
INSERT INTO Students (ROLL_NO, NAME, Address,Age) VALUES ('2','ALLEN','DELHI','19');
```

Student Table will look like this.

```
 roll_no | name  | address | age
---------+-------+---------+-----
       1 | ALEX  | NOIDA   |  19
       2 | ALLEN | DELHI   |  19
(2 rows)
```

3. **UPDATE Clause**

The UPDATE statement in SQL is used to update the data of an existing table in database. We can update single columns as well as multiple columns using UPDATE statement as per our requirement.
Basic Syntax

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

table_name: name of the table
column1: name of first , second, third column....
value1: new value for first, second, third column....
condition: condition to select the rows for which the values of columns needs to be updated.

NOTE: In the above query the **SET** statement is used to set new values to the particular column and the**WHERE** clause is used to select the rows for which the columns are needed to be updated. If we have not used the WHERE clause then the columns in **all** the rows will be updated. So the WHERE clause is used to choose the particular rows.
Example Query for Updating Multiple Columns

```
UPDATE Students SET NAME = 'BROOK', ADDRESS = 'GURUGRAM' WHERE ROLL_NO = 1;
```

The Student Table will look like.

```
 roll_no | name  | address  | age
---------+-------+----------+-----
       1 | BROOK | GURUGRAM |  19
       2 | ALLEN | DELHI    |  19
(2 rows)
```

4. **DELETE Clause**

   The DELETE Statement in SQL is used to delete existing records from a table. We can delete a single record or multiple records depending on the condition we specify in the WHERE clause.
   Basic Syntax

   DELETE FROM table_name WHERE some_condition;
   table_name: name of the table
   some_condition: condition to choose particular record.

   Note: We can delete single as well as multiple records depending on the condition we provide in WHERE clause. If we omit the WHERE clause then all of the records will be deleted and the table will be empty.
   Example Query for Deleting Record

```
DELETE FROM Students WHERE NAME = 'ALLEN';
```

   The Student table will look like this after deleting ALLEN's record.

```
 roll_no | name  | address  | age
---------+-------+----------+-----
       1 | BROOK | GURUGRAM |  19
(1 row)
```

– SQL | ALTER (ADD, DROP, MODIFY)

ALTER TABLE is used to add, delete/drop or modify columns in the existing table. It is also used to add and drop various constraints on the existing table.

ALTER TABLE - ADD

ADD is used to add columns into the existing table. Sometimes we may require to add additional information, in that case we do not require to create the whole database again, **ADD** comes to our rescue.

Syntax:

```
ALTER TABLE table_name

        ADD (Columnname_1  datatype,

        Columnname_2  datatype,

        ...

        Columnname_n  datatype);
```

ALTER TABLE - DROP

DROP COLUMN is used to drop column in a table. Deleting the unwanted columns from the table.

Syntax:

```
ALTER TABLE table_name

DROP COLUMN column_name;
```

ALTER TABLE-MODIFY

It is used to modify the existing columns in a table. Multiple columns can also be modified at once.
**Syntax may vary slightly in different databases.*

Syntax(Oracle,MySQL,MariaDB):

```
 ALTER TABLE table_name

MODIFY column_name column_type;
```

Syntax(SQL Server):

```
 ALTER TABLE table_name

ALTER COLUMN column_name column_type;
```

Queries

Sample Table:

Student

ROLL_NO	NAME
1	Ram
2	Abhi
3	Rahul
4	Tanu

QUERY:

- To ADD 2 columns AGE and COURSE to table Student.

```
ALTER TABLE Student ADD (AGE number(3),COURSE varchar(40));
```

OUTPUT:

ROLL_NO	NAME	AGE	COURSE
1	Ram		
2	Abhi		
3	Rahul		
4	Tanu		

- MODIFY column COURSE in table Student

```
ALTER TABLE Student MODIFY COURSE varchar(20);
```

After running the above query maximum size of Course Column is reduced to 20 from 40.

- DROP column COURSE in table Student.

```
ALTER TABLE Student DROP COLUMN COURSE;
```

OUTPUT:

ROLL_NO	NAME	AGE
1	Ram	
2	Abhi	
3	Rahul	
4	Tanu	

– DBMS | SQL - SELECT, ORDER BY, DISTINCT

We will be using below Students table to explain the **Example Queries**.

Students Table

roll_no	name	address	age
1	BROOK	GURUGRAM	19
2	ALEX	NOIDA	19
3	ALLEN	NOIDA	21
4	ROBIN	DELHI	20
5	CALVIN	DELHI	18

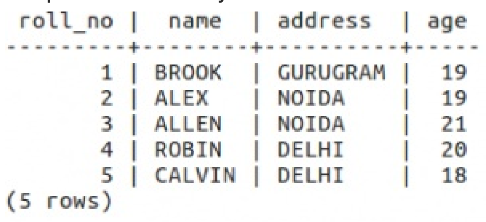
1. **SELECT Clause**

   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.
   Basic Syntax

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

   This query will return all the rows in the table with fields column1 , column2.
   - To fetch the entire table or all the fields in the table:

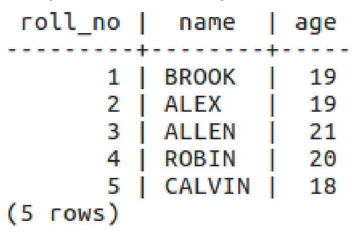
     ```
     SELECT * FROM Students;
     ```

     Output of the Query will look like this.
     

     ```
      roll_no |  name  | address  | age
     ---------+--------+----------+-----
            1 | BROOK  | GURUGRAM |  19
            2 | ALEX   | NOIDA    |  19
            3 | ALLEN  | NOIDA    |  21
            4 | ROBIN  | DELHI    |  20
            5 | CALVIN | DELHI    |  18
     (5 rows)
     ```

   - Query to fetch the fields roll_no, name, age from the table Students:

     ```
     SELECT roll_no, name, age FROM Students;
     ```

     Output of the Query will look like this.
     

     ```
      roll_no |  name  | age
     ---------+--------+-----
            1 | BROOK  |  19
            2 | ALEX   |  19
            3 | ALLEN  |  21
            4 | ROBIN  |  20
            5 | CALVIN |  18
     (5 rows)
     ```

2. **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.

   Syntax of all ways of using ORDER BY is shown below:
   - **Sort according to one column:** To sort in ascending or descending order we can use the keywords ASC or DESC respectively.

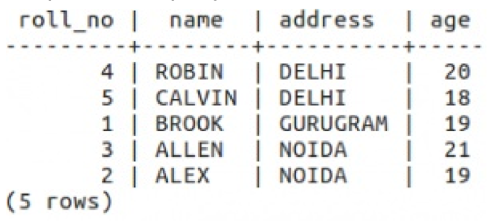
Syntax:

> **SELECT * FROM table_name ORDER BY column_name ASC|DESC**
> **table_name**: name of the table.
> **column_name**: name of the column according to which the data is needed to be arranged.
> **ASC**: to sort the data in ascending order.
> **DESC**: to sort the data in descending order.
> | : use either ASC or DESC to sort in ascending or descending order

Example : Query to get Student Records in Descending Order by name.

```
SELECT * FROM Students ORDER BY name DESC;
```

Output of the query will look like this.



```
 roll_no |  name  | address  | age
---------+--------+----------+-----
       4 | ROBIN  | DELHI    |  20
       5 | CALVIN | DELHI    |  18
       1 | BROOK  | GURUGRAM |  19
       3 | ALLEN  | NOIDA    |  21
       2 | ALEX   | NOIDA    |  19
(5 rows)
```

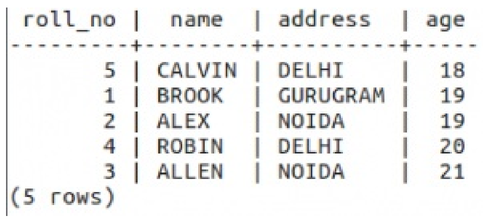
- **Sort according to multiple columns:** To sort in ascending or descending order we can use the keywords ASC or DESC respectively. To sort according to multiple columns, separate the names of columns by (,) operator.
  Syntax:

> **SELECT * FROM table_name ORDER BY column1 ASC|DESC, column2 ASC|DESC**

Example : Query to get Student Records in Ascending order by Age and Descending Order by name.

```
SELECT * FROM Students ORDER BY age ASC, name DESC;
```

Output of the query will look like this.



```
 roll_no |  name  | address  | age
---------+--------+----------+-----
       5 | CALVIN | DELHI    |  18
       1 | BROOK  | GURUGRAM |  19
       2 | ALEX   | NOIDA    |  19
       4 | ROBIN  | DELHI    |  20
       3 | ALLEN  | NOIDA    |  21
(5 rows)
```

Note : In Order By Clause the order of first column defined is given preference, later column order is used if the former column attributes are same. In the Above Example **age** is given preference.

3. **DISTINCT Clause**

The distinct keyword is used in conjunction with select keyword. It is helpful when there is need of avoiding the duplicate values present in any specific columns/table. When we use distinct keyword only the **unique values** are fetched.

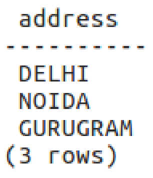
Basic Syntax:

> **SELECT DISTINCT column1,column2 FROM table_name**
> **column1 , column2:** names of the fields of the table
> **table_name:** from where we want to fetch
> This query will return all the unique combination of rows in the table with fields column1 , column2.

NOTE: If distinct keyword is used with multiple columns, the distinct combination is displayed in the result set.
Example : Query to get distinct address from Student Records.

```
SELECT DISTINCT address FROM Sudents;
```

Output of the query will look like this.



```
 address
----------
 DELHI
 NOIDA
 GURUGRAM
(3 rows)
```

– DBMS | SQL - LIKE, BETWEEN, IN

We will be using below Students table to explain the **Example Queries**.

Students Table

roll_no	name	address	age
1	BROOK	GURUGRAM	19
2	ALEX	NOIDA	19
3	ALLEN	NOIDA	21
4	ROBIN	DELHI	20
5	CALVIN	DELHI	18

1. **LIKE Clause**

   Sometimes we may require tuples from the database which match certain patterns. For example, we may wish to retrieve all columns where the tuples start with the letter 'y', or start with 'b' and end with 'l', or even more complicated and restrictive string patterns. This is where the LIKE Clause comes to rescue, often coupled with the WHERE Clause in SQL.

   There are two kinds of wildcards used to filter out the results:

   - % : Used to match zero or more characters. (Variable Length)
   - _ : Used to match exactly one character. (Fixed Length)

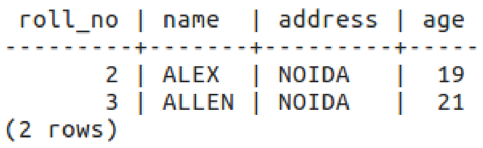
   The following are the rules for pattern matching with the LIKE Clause:

Pattern	Meaning
'a%'	Match strings which start with 'a'
'%a'	Match strings with end with 'a'
'a%t'	Match strings which contain the start with 'a' and end with 't'.
'%wow%'	Match strings which contain the substring 'wow' in them at any position.
'_wow%'	Match strings which contain the substring 'wow' in them at the second position.
'_a%'	Match strings which contain 'a' at the second position.
'a_%_%'	Match strings which start with 'a' and contain at least 2 more characters.

   Example : Query to get Student Records, Where Student's name start with**'A'**.

   ```
   SELECT * FROM Students WHERE name LIKE 'A%';
   ```

   Output of the query will look like this.

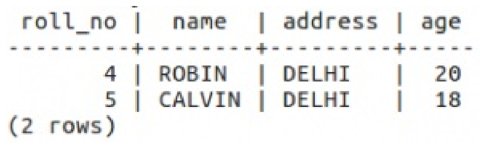


```
 roll_no | name  | address | age
---------+-------+---------+-----
       2 | ALEX  | NOIDA   |  19
       3 | ALLEN | NOIDA   |  21
(2 rows)
```

   Example : Query to get Student Records, Where Student's address second character is**'E'**.

   ```
   SELECT * FROM Students WHERE address LIKE '_E%';
   ```

   Output of the query will look like this.



```
 roll_no |  name  | address | age
---------+--------+---------+-----
       4 | ROBIN  | DELHI   |  20
       5 | CALVIN | DELHI   |  18
(2 rows)
```

2. **BETWEEN Clause**

The SQL BETWEEN condition allows you to easily test if an expression is within a range of values (inclusive). The values can be text, date, or numbers. It can be used in a SELECT, INSERT, UPDATE, or DELETE statement. The SQL BETWEEN Condition will return the records where expression is within the range of value1 and value2.

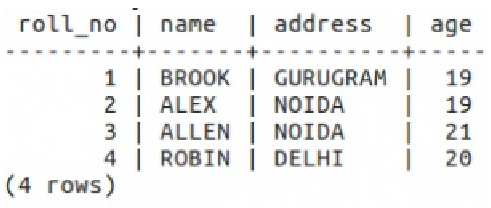
Syntax:

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

Example : Query to get Student Records, Where Student's age is between 19 and 21.

```
SELECT * FROM Students WHERE age BETWEEN 19 AND 21;
```

Output of the query will look like this.

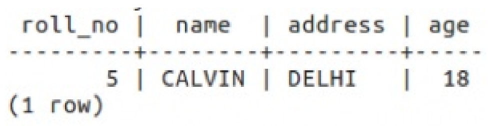


```
 roll_no | name  | address  | age
---------+-------+----------+-----
       1 | BROOK | GURUGRAM |  19
       2 | ALEX  | NOIDA    |  19
       3 | ALLEN | NOIDA    |  21
       4 | ROBIN | DELHI    |  20
(4 rows)
```

Example : Query to get Student Records, Where Student's age is**not** between 19 and 21.

```
SELECT * FROM Students WHERE age NOT BETWEEN 19 AND 21;
```

Output of the query will look like this.



```
 roll_no |  name  | address | age
---------+--------+---------+-----
       5 | CALVIN | DELHI   |  18
(1 row)
```

3. **IN Clause**

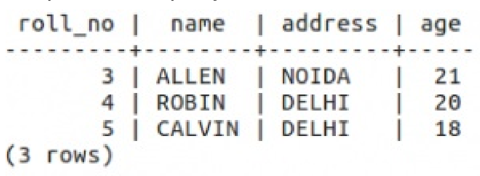
IN operator allows you to easily test if the expression matches any value in the list of values. It is used to remove the need of multiple OR condition in SELECT, INSERT, UPDATE or DELETE. You can also use NOT IN to exclude the rows in your list.
Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (list_of_values);
```

Example : Query to get Student Records, Where Student's age is in the set {18,20,21}.

```
SELECT * FROM Students WHERE age IN (18,20,21);
```

Output of the query will look like this.

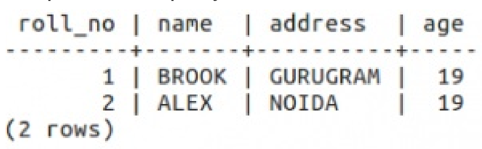


```
 roll_no |  name  | address | age
---------+--------+---------+-----
       3 | ALLEN  | NOIDA   |  21
       4 | ROBIN  | DELHI   |  20
       5 | CALVIN | DELHI   |  18
(3 rows)
```

Example : Query to get Student Records, Where Student's age is**not** in the set {18,20,21}.

```
SELECT * FROM Students WHERE age NOT IN (18,20,21);
```

Output of the query will look like this.



```
 roll_no | name  | address  | age
---------+-------+----------+-----
       1 | BROOK | GURUGRAM |  19
       2 | ALEX  | NOIDA    |  19
(2 rows)
```

A SQL Join statement is used to combine data or rows from two or more tables based on a common field between them. Different types of Joins are:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN

Consider the following Students and Marks tables below for example queries:

Students Table

roll_no	name	address	age
1	BROOK	GURUGRAM	19
2	ALEX	NOIDA	19
3	ALLEN	NOIDA	21
4	ROBIN	DELHI	20
5	CALVIN	DELHI	18

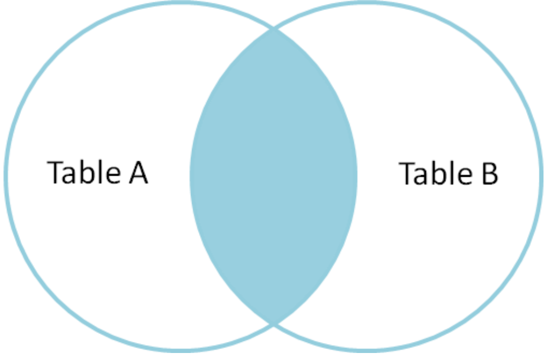
Marks Table

roll_no	course	score
1	OS	95
1	CN	90
2	CN	85
3	OS	98
4	DS	87
6	DS	95

1. **INNER JOIN:** The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be same.
   Syntax:

   ```
   SELECT table1.column1,table1.column2,table2.column1,....
   FROM table1
   INNER JOIN table2
   ON table1.matching_column = table2.matching_column;

   table1: First table.
   table2: Second table
   matching_column: Column common to both the tables.
   ```

   Note: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.



Example : Query to get names, course and score of students enrolled in different courses.

```
SELECT Students.name, Marks.course, Marks.score
FROM Students INNER JOIN Marks
ON Students.roll_no = Marks.roll_no;
```

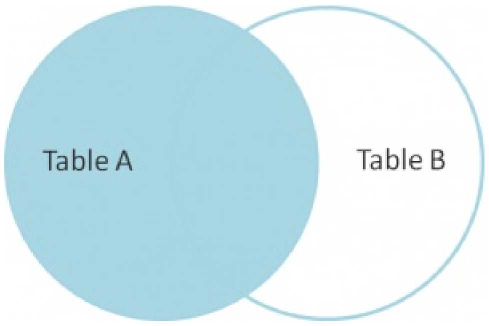
Output of the Query will look like this.

```
 name  | course | score
-------+--------+-------
 BROOK | OS     |    95
 BROOK | CN     |    90
 ALEX  | CN     |    85
 ALLEN | OS     |    98
 ROBIN | DS     |    87
(5 rows)
```

2. **LEFT JOIN**: This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of join. The rows for which there is no matching row on right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT OUTER JOIN.
Syntax:

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
LEFT JOIN table2
ON table1.matching_column = table2.matching_column;

table1: First table.
table2: Second table
matching_column: Column common to both the tables.
```

Note: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are same.



Example : Query for LEFT or LEFT OUTER JOIN.

```
SELECT Students.name, Marks.course, Marks.score
FROM Students LEFT JOIN Marks
ON Students.roll_no = Marks.roll_no;
```

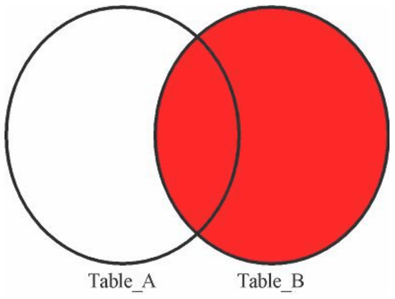
Output of the Query will look like this.

```
  name   | course | score
---------+--------+-------
 BROOK   | OS     |    95
 BROOK   | CN     |    90
 ALEX    | CN     |    85
 ALLEN   | OS     |    98
 ROBIN   | DS     |    87
 CALVIN  | NULL   |   NULL
(6 rows)
```

3. **RIGHT JOIN**: RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of join. The rows for which there is no matching row on left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.
   Syntax:

   ```
   SELECT table1.column1,table1.column2,table2.column1,....
   FROM table1
   RIGHT JOIN table2
   ON table1.matching_column = table2.matching_column;

   table1: First table.
   table2: Second table
   matching_column: Column common to both the tables.
   ```

   Note: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are same.

   Example : Query for RIGHT or RIGHT OUTER JOIN.

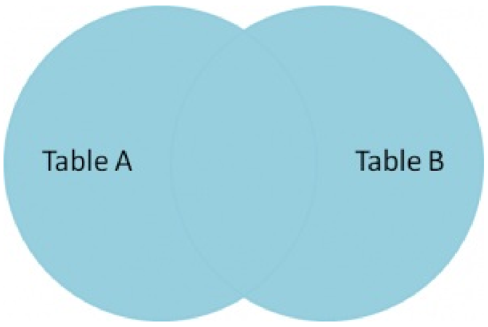
   ```
   SELECT Students.name, Marks.course, Marks.score
   FROM Students RIGHT JOIN Marks
   ON Students.roll_no = Marks.roll_no;
   ```

   Output of the Query will look like this.

   ```
    name  | course | score
   -------+--------+-------
    BROOK | OS     |    95
    BROOK | CN     |    90
    ALEX  | CN     |    85
    ALLEN | OS     |    98
    ROBIN | DS     |    87
     NULL | DS     |    95
   (6 rows)
   ```

4. **FULL JOIN:** FULL JOIN creates the result-set by combining result of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both the tables. The rows for which there is no matching, the result-set will contain *NULL* values.
   Syntax:

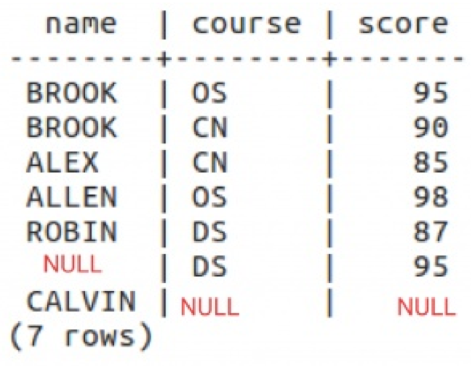
```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
FULL JOIN table2
ON table1.matching_column = table2.matching_column;

table1: First table.
table2: Second table
matching_column: Column common to both the tables.
```



Example : Query for FULL JOIN.

```
SELECT Students.name, Marks.course, Marks.score
FROM Students FULL JOIN Marks
ON Students.roll_no = Marks.roll_no;
```

Output of the Query will look like this.



```
  name   | course | score
---------+--------+-------
 BROOK   | OS     |    95
 BROOK   | CN     |    90
 ALEX    | CN     |    85
 ALLEN   | OS     |    98
 ROBIN   | DS     |    87
  NULL   | DS     |    95
 CALVIN  | NULL   |   NULL
(7 rows)
```

– SQL | Join (Cartesian Join & Self Join)

In this article, we will discuss the remaining two JOINS:

- **CARTESIAN JOIN**
- **SELF JOIN**

Consider the two tables below:

Student

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	Ram	Delhi	XXXXXXXXXX	18
2	RAMESH	GURGAON	XXXXXXXXXX	18
3	SUJIT	ROHTAK	XXXXXXXXXX	20
4	SURESH	Delhi	XXXXXXXXXX	18

StudentCourse

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4

1. **CARTESIAN JOIN**: The CARTESIAN JOIN is also known as CROSS JOIN. In a CARTESIAN JOIN there is a join for each row of one table to every row of another table. This usually happens when the matching column or WHERE condition is not specified.
   - In the absence of a WHERE condition the CARTESIAN JOIN will behave like a CARTESIAN PRODUCT . i.e., the number of rows in the result-set is the product of the number of rows of the two tables.
   - In the presence of WHERE condition this JOIN will function like a INNER JOIN.
   - Generally speaking, Cross join is similar to an inner join where the join-condition will always evaluate to True

   Syntax:

   ```
   SELECT table1.column1 , table1.column2, table2.column1...

   FROM table1

   CROSS JOIN table2;




   table1: First table.

   table2: Second table
   ```

Example Queries(CARTESIAN JOIN):

- In the below query we will select NAME and Age from Student table and COURSE_ID from StudentCourse table. In the output you can see that each row of the table Student is joined with every row of the table StudentCourse. The total rows in the result-set = 4 * 4 = 16.

  ```
  SELECT Student.NAME, Student.AGE, StudentCourse.COURSE_ID

  FROM Student

  CROSS JOIN StudentCourse;
  ```

  Output:

NAME	AGE	COURSE_ID
Ram	18	1
Ram	18	2
Ram	18	2
Ram	18	3
RAMESH	18	1
RAMESH	18	2
RAMESH	18	2
RAMESH	18	3
SUJIT	20	1
SUJIT	20	2
SUJIT	20	2
SUJIT	20	3
SURESH	18	1
SURESH	18	2
SURESH	18	2
SURESH	18	3

2. **SELF JOIN**: As the name signifies, in SELF JOIN a table is joined to itself. That is, each row of the table is joined with itself and all other rows depending on some conditions. In other words, we can say that it is a join between two copies of the same table.**Syntax:**

```
SELECT a.coulmn1 , b.column2

FROM table_name a, table_name b

WHERE some_condition;
```

table_name: Name of the table.

some_condition: Condition for selecting the rows.

Example Queries(SELF JOIN):

```
SELECT a.ROLL_NO , b.NAME

FROM Student a, Student b

WHERE a.ROLL_NO < b.ROLL_NO;
```

Output:

ROLL_NO	NAME
1	RAMESH
1	SUJIT
2	SUJIT
1	SURESH
2	SURESH
3	SURESH

– DBMS | SQL - Aggregate Functions, GROUP BY

We will be using below Students table to explain the **Example Queries**.

Students Table

roll_no	name	address	age
1	BROOK	GURUGRAM	19
2	ALEX	NOIDA	19
3	ALLEN	NOIDA	21
4	ROBIN	DELHI	20
5	CALVIN	DELHI	18

In database management an aggregate function is a function where the values of multiple rows are grouped together as input on certain criteria to form a single value of more significant meaning.

Various Aggregate Functions

```
1) Count()
2) Sum()
3) Avg()
4) Min()
5) Max()
```

1. **Count( ) Example** queries to demonstrate above function.

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

   ```
   Output : 5
   ```

   Returns total number of records .i.e 5.

   ```
   SELECT COUNT(age) FROM Students;
   ```

   ```
   Output : 5
   ```

   Return number of Non Null values over the column age. i.e 5.

   ```
   SELECT COUNT(DISTINCT age) FROM Students;
   ```

   ```
   Output : 4
   ```

   Return number of distinct Non Null values over the column age .i.e 4

2. **Sum( ) Example** queries to demonstrate above function.

```
SELECT SUM(age) FROM Students;
```

```
Output : 97
```

Sum all Non Null values of Column salary i.e.,97

```
SELECT SUM(DISTINCT age) FROM Students;
```

```
Output : 78
```

Sum of all distinct Non-Null values i.e., 78.

3. **Avg( ) Example** queries to demonstrate above function.

```
SELECT AVG(age) FROM Students;
```

```
Output : 19.40
```

Avg(age) = sum(age) / Count(age) = 97/5 = 19.40

```
SELECT AVG(DISTINCT age) FROM Students;
```

```
Output : 19.50
```

Avg(Distinct age) = sum(Distinct age) / Count(Distinct age) = 78/4 = 19.50

4. **Min( ) Example** query to demonstrate above function.

```
SELECT MIN(age) FROM Students;
```

```
Output : 18
```

Return minimum non null value over the column age. i.e 18.

5. **Max( ) Example** query to demonstrate above function.

```
SELECT MAX(age) FROM Students;
```

```
Output : 21
```

Return maximum non null value over the column age. i.e 21.

GROUP BY Clause

The GROUP BY Statement in SQL is used to arrange identical data into groups with the help of some functions. i.e if a particular column has same values in different rows then it will arrange these rows in a group.

Important Points:

- GROUP BY clause is used with the SELECT statement.
- In the query, GROUP BY clause is placed after the WHERE clause.
- In the query, GROUP BY clause is placed before ORDER BY clause if used any.

Syntax:

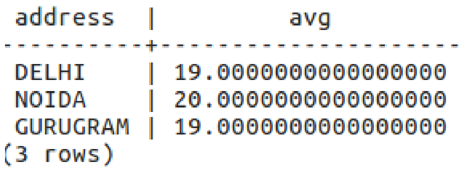
```
SELECT column1, function_name(column2)
FROM table_name
WHERE condition
GROUP BY column1, column2
ORDER BY column1, column2;
```

function_name: Name of the function used for example, SUM() , AVG().
table_name: Name of the table.
condition: Condition used.

Example query to demonstrate GROUP BY Clause.

```
SELECT address, AVG(age) FROM Students GROUP BY address;
```

Output of the Above query will look like this.



```
 address  |          avg
----------+---------------------
 DELHI    | 19.0000000000000000
 NOIDA    | 20.0000000000000000
 GURUGRAM | 19.0000000000000000
(3 rows)
```

This returns the Average Age for each address present in Students Table.

– DBMS | SQL - LIMIT and OFFSET

We will be using below Students table to explain the **Example Queries**.

Students Table

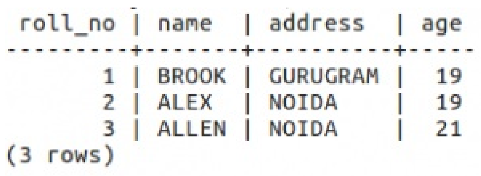
roll_no	name	address	age
1	BROOK	GURUGRAM	19
2	ALEX	NOIDA	19
3	ALLEN	NOIDA	21
4	ROBIN	DELHI	20
5	CALVIN	DELHI	18

If there are a large number of tuples satisfying the query conditions, it might be resourceful to view only a handful of them at a time.

- The **LIMIT** clause is used to set an upper limit on the number of tuples returned by SQL.
- It is important to note that this clause is not supported by all SQL versions.
- The LIMIT clause can also be specfied using the SQL 2008 OFFSET/FETCH FIRST clauses.
- The limit/offset expressions must be a non-negative integer.

Example Queries to demonstrate LIMIT Clause.

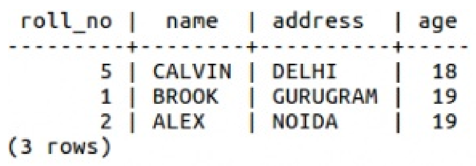
```
SELECT *
FROM Students
LIMIT 3;
```



```
 roll_no | name  | address  | age
---------+-------+----------+-----
       1 | BROOK | GURUGRAM |  19
       2 | ALEX  | NOIDA    |  19
       3 | ALLEN | NOIDA    |  21
(3 rows)
```

Output of the query gives First three Student Records.

```
SELECT *
FROM Students
ORDER BY age
LIMIT 3;
```



```
 roll_no |  name  | address  | age
---------+--------+----------+-----
       5 | CALVIN | DELHI    |  18
       1 | BROOK  | GURUGRAM |  19
       2 | ALEX   | NOIDA    |  19
(3 rows)
```

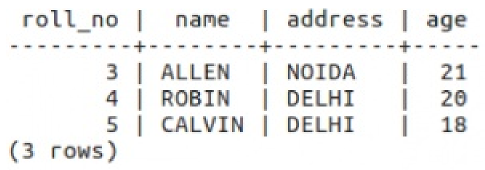
Output of the query gives three Student Records in order of Ascending Ages.

The LIMIT operator can be used in situations such as the above, where we need to find the top **N** students in a class and based on any condition statements.

Using LIMIT along with OFFSET

LIMIT **x** OFFSET **y** simply means skip the first y entries and then return the next x entries.
OFFSET can only be used with the ORDER BY clause. It cannot be used on its own.
OFFSET value must be greater than or equal to zero. It cannot be negative, else returns error.
Example query to demonstrate LIMIT and OFFSET.

```
SELECT *
FROM Students
LIMIT 3 OFFSET 2
ORDER BY roll_no;
```



```
 roll_no |  name  | address | age
---------+--------+---------+-----
       3 | ALLEN  | NOIDA   |  21
       4 | ROBIN  | DELHI   |  20
       5 | CALVIN | DELHI   |  18
(3 rows)
```

Returns 3 Student Records skipping first two records in Table.

– DBMS | SQL - UNION, INTERSECT, EXISTS

Consider the following Students and Marks tables below for example queries:

Students Table

roll_no	name	address	age
1	BROOK	GURUGRAM	19
2	ALEX	NOIDA	19
3	ALLEN	NOIDA	21
4	ROBIN	DELHI	20
5	CALVIN	DELHI	18

Marks Table

roll_no	course	score
1	OS	95
1	CN	90
2	CN	85
3	OS	98
4	DS	87
6	DS	95

1. **UNION Clause**

   The Union Clause is used to combine two separate select statements and produce the result set as a union of both the select statements.
   NOTE:
   1. The fields to be used in both the selet statements must be in same order, same number and same data type.
   2. The Union clause produces distinct values in the result set, to fetch the duplicate values too UNION ALL must be used instead of just UNION.

   Basic Syntax:

   ```
   SELECT column_name(s) FROM table1
   UNION
   SELECT column_name(s) FROM table2;
   Resultant set consists of distinct values.
   ```

   Example query to demonstrate UNION Clause.

   ```
   SELECT roll_no FROM Students UNION SELECT roll_no FROM Marks;
   ```

   ```
    roll_no
   ---------
          2
          3
          5
          4
          6
          1
   (6 rows)
   ```

   Output of the query gives Union Set of Roll Numbers.

2. **INTERSECT Clause**

   As the name suggests, the intersect clause is used to provide the result of the intersection of two select statements. This implies the result contains all the rows which are common to both the SELECT statements.

   Syntax :

   ```
   SELECT column-1, column-2 ……
   FROM table 1
   WHERE…..
   INTERSECT
   SELECT column-1, column-2 ……
   FROM table 2
   WHERE…..
   ```

   Example query to demonstrate INTERSECT Clause.

   ```
   SELECT roll_no FROM Students INTERSECT SELECT roll_no FROM Marks;
   ```

   ```
    roll_no
   ---------
          3
          4
          2
          1
   (4 rows)
   ```

   Output of the query gives Intersection Set of Roll Numbers.

3. **EXISTS Clause**

The EXISTS condition 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 or False. It can be used in a SELECT, UPDATE, INSERT or DELETE statement.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS
  (SELECT column_name(s)
   FROM table_name
   WHERE condition);
```

Example query to demonstrate EXISTS Clause.

```
SELECT roll_no, name FROM Students
WHERE
EXISTS ( SELECT * FROM Marks WHERE Students.roll_no = Marks.roll_no AND Marks.score > 90);
```

```
 roll_no | name
---------+-------
       1 | BROOK
       3 | ALLEN
(2 rows)
```

Output of the query gives student Roll No. and Names, whose score is greater than 90 in any Course.

– SQL | Subqueries

In SQL a Subquery can be simply defined as a query within another query. In other words we can say that a Subquery is a query that is embedded in WHERE clause of another SQL query.

Important rules for Subqueries:

- You can place the Subquery in a number of SQL clauses:WHERE clause, HAVING clause, FROM clause. Subqueries can be used with SELECT, UPDATE, INSERT, DELETE statements along with expression operator. It could be equality operator or comparison operator such as =, >, =, <= and Like operator.
- A subquery is a query within another query. The outer query is called as**main query** and inner query is called as **subquery**.
- The subquery generally executes first, and its output is used to complete the query condition for the main or outer query.
- Subquery must be enclosed in parentheses.
- Subqueries are on the right side of the comparison operator.
- ORDER BY command **cannot** be used in a Subquery. GROUPBY command can be used to perform same function as ORDER BY command.
- Use single-row operators with singlerow Subqueries. Use multiple-row operators with multiple-row Subqueries.

Syntax: There is not any general syntax for Subqueries. However, Subqueries are seen to be used most frequently with SELECT statement as shown below:

```
SELECT column_name
FROM table_name
WHERE column_name expression operator
  ( SELECT COLUMN_NAME  from TABLE_NAME   WHERE ... );
```

Sample Table:

DATABASE

NAME	ROLL_NO	LOCATION	PHONE_NUMBER
Ram	101	Chennai	9988775566

Raj	102	Coimbatore	8877665544
Sasi	103	Madurai	7766553344
Ravi	104	Salem	8989898989
Sumathi	105	Kanchipuram	8989856868

STUDENT

NAME	ROLL_NO	SECTION
Ravi	104	A
Sumathi	105	B
Raj	102	A

Sample Queries

:

- To display NAME, LOCATION, PHONE_NUMBER of the students from DATABASE table whose section is A

```
Select NAME, LOCATION, PHONE_NUMBER from DATABASE
WHERE ROLL_NO IN
(SELECT ROLL_NO from STUDENT where SECTION='A');
```

Explanation : First subquery executes " SELECT ROLL_NO from STUDENT where SECTION='A' " returns ROLL_NO from STUDENT table whose SECTION is 'A'.Then outer-query executes it and return the NAME, LOCATION, PHONE_NUMBER from the DATABASE table of the student whose ROLL_NO is returned from inner subquery.

Output:

NAME	ROLL_NO	LOCATION	PHONE_NUMBER
Ravi	104	Salem	8989898989
Raj	102	Coimbatore	8877665544

- Insert Query Example:

Table1: Student1

NAME	ROLL_NO	LOCATION	PHONE_NUMBER
Ram	101	chennai	9988773344
Raju	102	coimbatore	9090909090
Ravi	103	salem	8989898989

Table2: Student2

NAME	ROLL_NO	LOCATION	PHONE_NUMBER
Raj	111	chennai	8787878787
Sai	112	mumbai	6565656565
Sri	113	coimbatore	7878787878

To insert Student2 into Student1 table:

```
INSERT INTO Student1  SELECT * FROM Student2;
```

Output:

NAME	ROLL_NO	LOCATION	PHONE_NUMBER
Ram	101	chennai	9988773344
Raju	102	coimbatore	9090909090
Ravi	103	salem	8989898989
Raj	111	chennai	8787878787
Sai	112	mumbai	6565656565
Sri	113	coimbatore	7878787878

- To delete students from Student2 table whose rollno is same as that in Student1 table and having location as chennai

```
DELETE FROM Student2
WHERE ROLL_NO IN ( SELECT ROLL_NO
              FROM Student1
              WHERE LOCATION = 'chennai');
```

Output:

```
1 row delete successfully.
```

Display Student2 table:

NAME	ROLL_NO	LOCATION	PHONE_NUMBER
Sai	112	mumbai	6565656565
Sri	113	coimbatore	7878787878

- To update name of the students to geeks in Student2 table whose location is same as Raju,Ravi in Student1 table

```
UPDATE Student2
SET NAME='geeks'
WHERE LOCATION IN ( SELECT LOCATION
            FROM Student1
            WHERE NAME IN ('Raju','Ravi'));
```

Output:

```
1 row updated successfully.
```

Display Student2 table:

NAME	ROLL_NO	LOCATION	PHONE_NUMBER
Sai	112	mumbai	6565656565
geeks	113	coimbatore	7878787878

– SQL | Sub queries in From Clause

From clause can be used to specify a sub-query expression in SQL. The relation produced by the sub-query is then used as a new relation on which the outer query is applied.

- Sub queries in the from clause are supported by most of the SQL implementations.
- The correlation variables from the relations in from clause cannot be used in the sub-queries in the from clause.

Syntax:

```
SELECT column1, column2 FROM

(SELECT column_x  as C1, column_y FROM table WHERE PREDICATE_X)

as table2

WHERE PREDICATE;
```

Note: The sub-query in the from clause is evaluated first and then the results of evaluation are stored in a new temporary relation.
Next, the outer query is evaluated, selecting only those tuples from the temporary relation that satisfies the predicate in the where clause of the outer query.

Query

Example 1: Find all professors whose salary is greater than the average budget of all the departments.

Instructor relation:

InstructorID	Name	Department	Salary
44547	Smith	Computer Science	95000
44541	Bill	Electrical	55000
47778	Sam	Humanities	44000
48147	Erik	Mechanical	80000
411547	Melisa	Information Technology	65000
48898	Jena	Civil	50000

Department relation:

Department Name	Budget
Computer Science	100000
Electrical	80000
Humanities	50000
Mechanical	40000
Information Technology	90000
Civil	60000

Query:

```
select I.ID, I.NAME, I.DEPARTMENT, I.SALARY from

(select avg(BUDGET) as averageBudget from DEPARTMENT) as BUDGET, Instructor as I
where I.SALARY > BUDGET.averageBudget;
```

Output

InstructorID	Name	Department	Salary
44547	Smith	Computer Science	95000
48147	Erik	Mechanical	80000

Explanation: The average budget of all departments from the department relation is 70000. Erik and Smith are the only instructors in the instructor relation whose salary is more than 70000 and thereofre are present in the output relation.

– DBMS | SQL Nested Queries

In nested queries, a query is written inside a query. The result of the inner query is used in the execution of the outer query.Consider the following Students and Marks tables below for example queries:

Students Table

roll_no	name	address	age
1	BROOK	GURUGRAM	19
2	ALEX	NOIDA	19
3	ALLEN	NOIDA	21
4	ROBIN	DELHI	20
5	CALVIN	DELHI	18

Marks Table

roll_no	course	score
1	OS	95
1	CN	90
2	CN	85
3	OS	98
4	DS	87
6	DS	95

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 the inner query is independent of the outer query, but the result of the inner query is used in the execution of the outer query. Various operators like IN, NOT IN, ANY, ALL, etc are used in writing independent nested queries.
  For Example
  Suppose we want to find out the Student Details who have scored greater than or equal to 90 in any of the course.

  Step 1: We will be finding out the Roll_No of the Students scored greater than or equal to 90 from Marks Table.

  ```
  SELECT Roll_No FROM Marks WHERE Score >= 90;
  ```

  Step 2: We will be using result of step 1 to find the student details.

  ```
  SELECT * FROM Students WHERE Roll_No
  IN
  (SELECT DISTINCT Roll_No FROM Marks WHERE Score >= 90);
  ```

  ```
   roll_no | name  | address  | age
  ---------+-------+----------+-----
         1 | BROOK | GURUGRAM |  19
         3 | ALLEN | NOIDA    |  21
  (2 rows)
  ```

  The inner query will return a set with Roll No (1, 3, 6) and the outer query will return the Student Details Whose Roll_No is in the set (1, 3, 6). The output will contain Student Details of Roll No 1 and 3.

  For Example
  If we want to find out the Student Details who have neither enrolled in ‘CN’ nor ‘OS’, it can be done as:

  ```
  SELECT * FROM Students WHERE ROll_No
  NOT IN
  (SELECT DISTINCT Roll_No FROM Marks WHERE Course = 'CN' or course = 'OS');
  ```

  ```
   roll_no |  name  | address | age
  ---------+--------+---------+-----
         4 | ROBIN  | DELHI   |  20
         5 | CALVIN | DELHI   |  18
  (2 rows)
  ```

  The inner query will return a set with Roll No (1, 2, 3) and the outer query will return the Student Details Whose Roll_No is not in the set (1, 2, 3). The output will contain Student Details of Roll No 4 and 5.

- **Co-related Nested Queries:** In co-related nested queries, the output of the inner query depends on the row which is being currently executed in the outer query.
  For Example If we want to find out Student Details who are enrolled in the course ‘CN’, it can be done with the help of co-related nested query as:

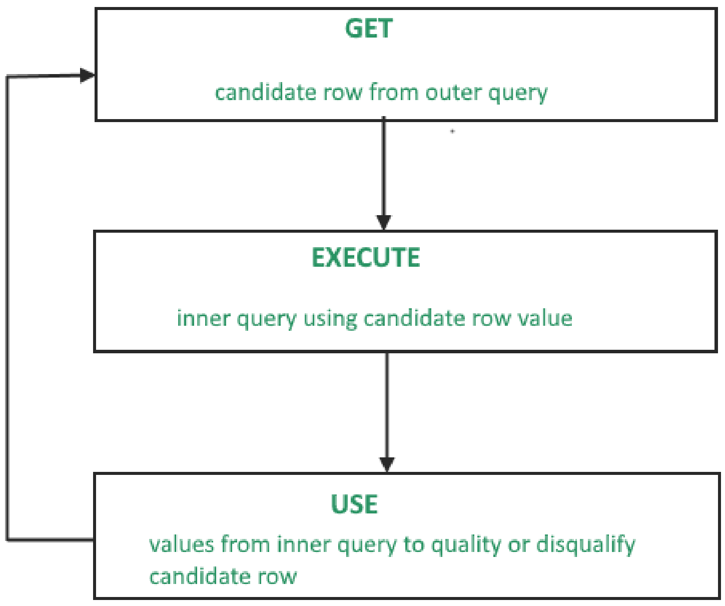
  ```
  SELECT * FROM Students WHERE
  EXISTS
  (SELECT * FROM Students WHERE Student.Roll_No = Marks.Roll_No AND Marks.Course = 'CN');
  ```

  For each row of Student Table, it will find the rows from Marks Table where Student.Roll_No = Marks.Roll_No AND Marks.Course = 'CN'. If for a particular Student from Students Table, at least a row exists in Marks Table with Course = 'CN', then the inner query will return true and corresponding Student Detail will be returned as output.

  ```
   roll_no | name  | address  | age
  ---------+-------+----------+-----
         1 | BROOK | GURUGRAM |  19
         3 | ALLEN | NOIDA    |  21
  (2 rows)
  ```

– SQL Correlated Subqueries

Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.


GET
candidate row from outer query
EXECUTE
inner query using candidate row value
USE
values from inner query to quality or disqualify candidate row

A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a **SELECT**, **UPDATE**, or **DELETE** statement.

```
SELECT column1, column2, ....

FROM table1 outer

WHERE column1 operator

                    (SELECT column1, column2

                     FROM table2

                     WHERE expr1 =

                               outer.expr2);
```

A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. In other words, you can use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.

Nested Subqueries Versus Correlated Subqueries :

With a normal nested subquery, the inner **SELECT** query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each candidate row considered by the outer query. In other words, the inner query is driven by the outer query.

NOTE : You can also use the **ANY** and **ALL** operator in a correlated subquery.

EXAMPLE of Correlated Subqueries : Find all the employees who earn more than the average salary in their department.

```
SELECT last_name, salary, department_id
 FROM employees outer
 WHERE salary >
               (SELECT AVG(salary)
                FROM employees
                WHERE department_id =
                       outer.department_id);
```

Other use of correlation are in **UPDATE** and **DELETE**

CORRELATED UPDATE :

```
UPDATE table1 alias1
 SET column = (SELECT expression
              FROM table2 alias2
              WHERE alias1.column =
                    alias2.column);
```

Use a correlated subquery to update rows in one table based on rows from another table.

CORRELATED DELETE :

```
DELETE FROM table1 alias1
 WHERE column1 operator
              (SELECT expression
               FROM table2 alias2
               WHERE alias1.column = alias2.column);
```

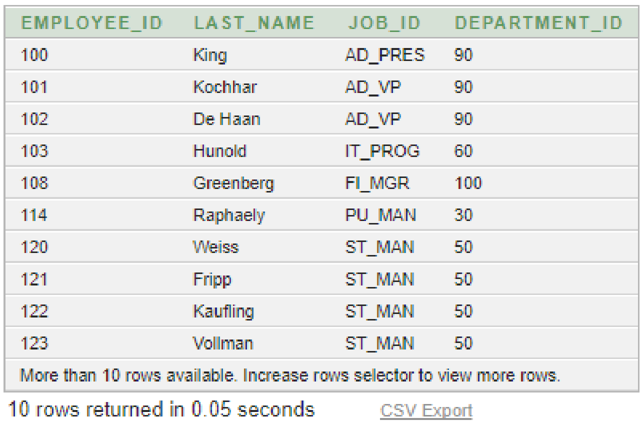
Use a correlated subquery to delete rows in one table based on the rows from another table.

Using the EXISTS Operator :

The EXISTS operator tests for existence of rows in the results set of the subquery. If a subquery row value is found the condition is flagged **TRUE** and the search does not continue in the inner query, and if it is not found then the condition is flagged **FALSE** and the search continues in the inner query.
EXAMPLE of using EXIST operator :
Find employees who have at least one person reporting to them.

```
SELECT employee_id, last_name, job_id, department_id

FROM employees outer

WHERE EXISTS ( SELECT 'X'

FROM employees

WHERE manager_id =

outer.employee_id);
```

OUTPUT :



EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90
103	Hunold	IT_PROG	60
108	Greenberg	FI_MGR	100
114	Raphaely	PU_MAN	30
120	Weiss	ST_MAN	50
121	Fripp	ST_MAN	50
122	Kaufling	ST_MAN	50
123	Vollman	ST_MAN	50

More than 10 rows available. Increase rows selector to view more rows.

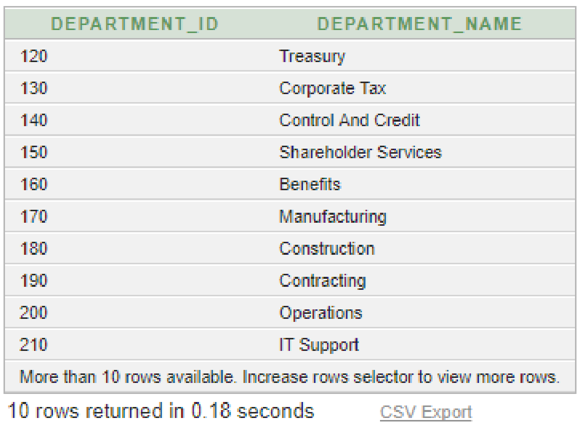
10 rows returned in 0.05 seconds CSV Export

EXAMPLE of using NOT EXIST operator :

Find all departments that do not have any employees.

```
SELECT department_id, department_name

FROM departments d

WHERE NOT EXISTS (SELECT 'X'

FROM employees

WHERE department_id

= d.department_id);
```

OUTPUT :



DEPARTMENT_ID	DEPARTMENT_NAME
120	Treasury
130	Corporate Tax
140	Control And Credit
150	Shareholder Services
160	Benefits
170	Manufacturing
180	Construction
190	Contracting
200	Operations
210	IT Support

More than 10 rows available. Increase rows selector to view more rows.

10 rows returned in 0.18 seconds CSV Export