# Relational Model in DBMS

Relational Model was proposed by E.F. Codd to model data in the form of relations or tables. After designing the conceptual model of Database using ER diagram, we need to convert the conceptual model in the relational model which can be implemented using any RDBMS languages like Oracle SQL, MySQL etc.

Relational Model represents how data is stored in Relational Databases. A relational database stores data in the form of relations (tables). Consider a relation STUDENT with attributes ROLL_NO, NAME, ADDRESS, PHONE and AGE shown in Table 1.

| ROLL_NO | NAME | ADDRESS | PHONE | AGE |
|---|---|---|---|---|
| 1 | RAM | DELHI | 9455123451 | 18 |
| 2 | RAMESH | GURGAON | 9652431543 | 18 |
| 3 | SUJIT | ROHTAK | 9156253131 | 20 |
| 4 | SURESH | DELHI | | 18 |

**IMPORTANT TERMINOLOGIES**

- **Attribute:** Attributes are the properties that define a relation. e.g.; **ROLL_NO**, **NAME**
- **Relation Schema:** A relation schema represents name of the relation with its attributes. e.g.; STUDENT (ROLL_NO, NAME, ADDRESS, PHONE and AGE) is relation schema for STUDENT. If a schema has more than 1 relation, it is called Relational Schema.
- **Tuple:** Each row in the relation is known as tuple. The above relation contains 4 tuples, one of which is shown as:

1 RAM DELHI 9455123451 18

- **Relation Instance:** The set of tuples of a relation at a particular instance of time is called as relation instance. Table 1 shows the relation instance of STUDENT at a particular time. It can change whenever there is insertion, deletion or updation in the database.
- **Degree:** The number of attributes in the relation is known as degree of the relation. The **STUDENT** relation defined above has degree 5.
- **Cardinality:** The number of tuples in a relation is known as cardinality. The **STUDENT** relation defined above has cardinality 4.
- **Column:** Column represents the set of values for a particular attribute. The column **ROLL_NO** is extracted from relation STUDENT.

**ROLL_NO**
1
2

3
4

- **NULL Values:** The value which is not known or unavailable is called NULL value. It is represented by blank space. e.g.; PHONE of STUDENT having ROLL_NO 4 is NULL.

## Constraints in Relational Model

While designing Relational Model, we define some conditions which must hold for data present in database are called Constraints. These constraints are checked before performing any operation (insertion, deletion and updation) in database. If there is a violation in any of constrains, operation will fail.

**Domain Constraints:** These are attribute level constraints. An attribute can only take values which lie inside the domain range. e.g,; If a constrains AGE>0 is applied on STUDENT relation, inserting negative value of AGE will result in failure.

**Key Integrity:** Every relation in the database should have atleast one set of attributes which defines a tuple uniquely. Those set of attributes is called key. e.g.; ROLL_NO in STUDENT is a key. No two students can have same roll number. So a key has two properties:

- It should be unique for all tuples.
- It can't have NULL values.

**Referential Integrity:** When one attribute of a relation can only take values from other attribute of same relation or any other relation, it is called referential integrity. Let us suppose we have 2 relations

**STUDENT**

| ROLL_NO | NAME | ADDRESS | PHONE | AGE | BRANCH_CODE |
|---|---|---|---|---|---|
| 1 | RAM | DELHI | 9455123451 | 18 | CS |
| 2 | RAMESH | GURGAON | 9652431543 | 18 | CS |
| 3 | SUJIT | ROHTAK | 9156253131 | 20 | ECE |
| 4 | SURESH | DELHI | | 18 | IT |

**BRANCH**

| BRANCH_CODE | BRANCH_NAME |
|---|---|
| CS | COMPUTER SCIENCE |
| IT | INFORMATION TECHNOLOGY |
| ECE | ELECTRONICS AND COMMUNICATION ENGINEERING |
| CV | CIVIL ENGINEERING |

BRANCH_CODE of STUDENT can only take the values which are present in BRANCH_CODE of BRANCH which is called referential integrity constraint. The relation which is referencing to other relation is called REFERENCING RELATION (STUDENT in this case) and the relation to which other relations refer is called REFERENCED RELATION (BRANCH in this case).

# Relational Algebra in DBMS

Relational Algebra is procedural query language, which takes Relation as input and generate relation as output. Relational algebra mainly provides theoretical foundation for relational databases and SQL.

**Operators in Relational Algebra**

**Projection (π)**

Projection is used to project required column data from a relation.
Example :

```
     R

  (A  B  C)

  ----------

   1  2  4

   2  2  3

   3  2  3

   4  3  4
```

```
π (BC)

B  C

-----

2  4

2  3

3  4
```

**Note:** By Default projection removes duplicate data.

**Selection (σ)**

Selection is used to select required tuples of the relations.
for the above relation
σ (c>3)R
will select the tuples which have c more than 3.

**Note:** selection operator only selects the required tuples but does not display them. For displaying, data projection operator is used.
For the above selected tuples, to display we need to use projection also.

π (σ (c>3)R ) will show following tuples.

```
A  B  C
-------
1  2  4
4  3  4
```

**Union (U)**
Union operation in relational algebra is same as union operation in set theory, only constraint is for union of two relation both relation must have same set of Attributes.

**Set Difference (-)**
Set Difference in relational algebra is same set difference operation as in set theory with the constraint that both relation should have same set of attributes.

**Rename (ρ)**
Rename is a unary operation used for renaming attributes of a relation.
ρ (a/b)R will rename the attribute ‘b’ of relation by ‘a’.

**Cross Product (X)**
Cross product between two relations let say A and B, so cross product between A X B will results all the attributes of A followed by each attribute of B. Each record of A will pairs with every record of B.

below is the example

```
A                                   B
 (Name   Age  Sex )                 (Id   Course)
 ------------------                 -------------
 Ram    14   M                       1     DS
 Sona   15   F                       2     DBMS
 kim    20   M
```

A X B

| Name | Age | Sex | Id | Course |
|---|---|---|---|---|
| Ram | 14 | M | 1 | DS |
| Ram | 14 | M | 2 | DBMS |
| Sona | 15 | F | 1 | DS |
| Sona | 15 | F | 2 | DBMS |
| Kim | 20 | M | 1 | DS |
| Kim | 20 | M | 2 | DBMS |

**Note:** if A has ‘n’ tuples and B has ‘m’ tuples then A X B will have ‘n*m’ tuples.

**Natural Join (⋈)**

Natural join is a binary operator. Natural join between two or more relations will result set of all combination of tuples where they have equal common attribute.

Let us see below example

Emp (Name, Id, Dept_name)

| Name | Id | Dept_name |
|---|---|---|
| A | 120 | IT |
| B | 125 | HR |
| C | 110 | Sale |
| D | 111 | IT |

Dep (Dept_name, Manager)

| Dept_name | Manager |
|---|---|
| Sale | Y |
| Prod | Z |
| IT | A |

Emp ⋈ Dep

| Name | Id | Dept_name | Manager |
|---|---|---|---|

```
------------------------------
A     120   IT          A
C     110   Sale        Y
D     111   IT          A
```

**Conditional Join**

Conditional join works similar to natural join. In natural join, by default condition is equal between common attribute while in conditional join we can specify the any condition such as greater than, less than, not equal

Let us see below example

```
         R                          S
  (ID   Sex   Marks)          (ID   Sex   Marks)
  -----------------          --------------------
   1   F   45                   10   M   20
   2   F   55                   11   M   22
   3   F   60                   12   M   59
```

Join between R And S with condition **R.marks >= S.marks**

```
R.ID   R.Sex   R.Marks   S.ID   S.Sex   S.Marks
-----------------------------------------------
1       F       45        10     M        20
1       F       45        11     M        22
2       F       55        10     M        20
2       F       55        11     M        22
3       F       60        10     M        20
3       F       60        11     M        22
3       F       60        12     M        59
```

# Extended Operators in Relational Algebra

- **Join**
- **Intersection**
- **Divide**

The relations used to understand extended operators are STUDENT,

STUDENT_SPORTS, ALL_SPORTS and EMPLOYEE which are shown in Table 1, Table 2, Table 3 and Table 4 respectively.

**STUDENT**

| ROLL_NO | NAME | ADDRESS | PHONE | AGE |
|---|---|---|---|---|
| 1 | RAM | DELHI | 9455123451 | 18 |
| 2 | RAMESH | GURGAON | 9652431543 | 18 |
| 3 | SUJIT | ROHTAK | 9156253131 | 20 |
| 4 | SURESH | DELHI | 9156768971 | 18 |

**Table 1**

**STUDENT_SPORTS**

| ROLL_NO | SPORTS |
|---|---|
| 1 | Badminton |
| 2 | Cricket |
| 2 | Badminton |
| 4 | Badminton |

**Table 2**

**ALL_SPORTS**

| SPORTS |
|---|
| Badminton |
| Cricket |

**Table 3**

**EMPLOYEE**

| EMP_NO | NAME | ADDRESS | PHONE | AGE |
|---|---|---|---|---|
| 1 | RAM | DELHI | 9455123451 | 18 |
| 5 | NARESH | HISAR | 9782918192 | 22 |
| 6 | SWETA | RANCHI | 9852617621 | 21 |
| 4 | SURESH | DELHI | 9156768971 | 18 |

**Table 4**

***Intersection (∩):*** Intersection on two relations R1 and R2 can only be computed if R1 and R2 are **union compatible** (These two relation should have same number of attributes and corresponding attributes in two relations have same domain). Intersection operator when applied on two relations as R1∩R2 will give a relation with tuples which are in R1 as well as R2. Syntax:

**Relation1 ∩ Relation2**

Example: Find a person who is student as well as employee- **STUDENT ∩ EMPLOYEE**

In terms of basic operators (union and minus) :

**STUDENT ∩ EMPLOYEE = STUDENT + EMPLOYEE - (STUDENT U EMPLOYEE)**

**RESULT:**

| ROLL_NO | NAME | ADDRESS | PHONE | AGE |
|---|---|---|---|---|
| 1 | RAM | DELHI | 9455123451 | 18 |

| | | | | |
|---|---|---|---|---|
| 4 | SURESH | DELHI | 9156768971 | 18 |

***Conditional Join($\bowtie_c$):*** Conditional Join is used when you want to join two or more relation based on some conditions. Example: Select students whose ROLL_NO is greater than EMP_NO of employees

$$\mathbf{STUDENT} \bowtie_{c\ \mathbf{STUDENT.ROLL\_NO > EMPLOYEE.EMP\_NO}} \mathbf{EMPLOYEE}$$

In terms of basic operators (cross product and selection) :

$$\sigma_{\mathbf{(STUDENT.ROLL\_NO > EMPLOYEE.EMP\_NO)}}\mathbf{(STUDENT \times EMPLOYEE)}$$

**RESULT:**

| ROLL_NO | NAME | ADDRESS | PHONE | AGE | EMP_NO | NAME | ADDRESS | PHONE | AGE |
|---|---|---|---|---|---|---|---|---|---|
| | RAM | GURG | 96524 | | | RA | DELH | 94551 | |
| 2 | ESH | AON | 31543 | 18 | 1 | M | I | 23451 | 18 |
| | SUJI | ROHT | 91562 | | | RA | DELH | 94551 | |
| 3 | T | AK | 53131 | 20 | 1 | M | I | 23451 | 18 |
| | SUR | | 91567 | | | RA | DELH | 94551 | |
| 4 | ESH | DELHI | 68971 | 18 | 1 | M | I | 23451 | 18 |

***Equijoin($\bowtie$):*** Equijoin is a **special case of conditional join** where only equality condition holds between a pair of attributes. As values of two attributes will be equal in result of equijoin, only one attribute will be appeared in result.

Example:Select students whose ROLL_NO is equal to EMP_NO of employees

$$\mathbf{STUDENT} \bowtie_{\mathbf{STUDENT.ROLL\_NO = EMPLOYEE.EMP\_NO}} \mathbf{EMPLOYEE}$$

In terms of basic operators (cross product, selection and projection) :

$$\prod_{\mathbf{(STUDENT.ROLL\_NO,\ STUDENT.NAME,\ STUDENT.ADDRESS,\ STUDENT.PHONE,\ STUDENT.AGE\ EMPLOYEE.NAME,\ EMPLOYEE.ADDRESS,\ EMPLOYEE.PHONE,\ EMPLOYEE > AGE)}}\mathbf{(}\sigma_{\mathbf{(STUDENT.ROLL\_NO = EMPLOYEE.EMP\_NO)}}\ \mathbf{(STUDENT \times EMPLOYEE))}$$

RESULT:

| ROLL_NO | NAME | ADDRESS | PHONE | AGE | NAME | ADDRESS | PHONE | AGE |
|---|---|---|---|---|---|---|---|---|
| | | | 945512 | | | | 945512 | |
| 1 | RAM | DELHI | 3451 | 18 | RAM | DELHI | 3451 | 18 |
| | SURE | | 915676 | | SURE | | 915676 | |
| 4 | SH | DELHI | 8971 | 18 | SH | DELHI | 8971 | 18 |

***Natural Join($\bowtie$):*** It is a special case of equijoin in which equality condition hold on all attributes which have same name in relations R and S (relations on which join operation is applied). While applying natural join on two relations, there is no need to write equality condition explicitly. Natural Join will also return the similar attributes only once as their value will be same in resulting relation.

Example: Select students whose ROLL_NO is equal to ROLL_NO of STUDENT_SPORTS as:

$$STUDENT \bowtie STUDENT\_SPORTS$$

In terms of basic operators (cross product, selection and projection) :

$$\prod_{(STUDENT.ROLL\_NO,\ STUDENT.NAME,\ STUDENT.ADDRESS,\ STUDENT.PHONE,\ STUDENT.AGE\ STUDENT\_SPORTS.SPORTS)}(\sigma_{(STUDENT.ROLL\_NO=STUDENT\_SPORTS.ROLL\_NO)}(STUDENT \times STUDENT\_SPORTS))$$

**RESULT:**

| ROLL_NO | NAME | ADDRESS | PHONE | AGE | SPORTS |
|---|---|---|---|---|---|
| 1 | RAM | DELHI | 9455123451 | 18 | Badminton |
| 2 | RAMESH | GURGAON | 9652431543 | 18 | Cricket |
| 2 | RAMESH | GURGAON | 9652431543 | 18 | Badminton |
| 4 | SURESH | DELHI | 9156768971 | 18 | Badminton |

Natural Join is by default inner join because the tuples which does not satisfy the conditions of join does not appear in result set. e.g.; The tuple having ROLL_NO 3 in STUDENT does not match with any tuple in STUDENT_SPORTS, so it has not been a part of result set.

***Left Outer Join(⟕):*** When applying join on two relations R and S, some tuples of R or S does not appear in result set which does not satisfy the join conditions. But Left Outer Joins gives all tuples of R in the result set. The tuples of R which do not satisfy join condition will have values as NULL for attributes of S.

Example:Select students whose ROLL_NO is greater than EMP_NO of employees and details of other students as well

$$STUDENT \&\#10197_{STUDENT.ROLL\_NO>EMPLOYEE.EMP\_NO} EMPLOYEE$$

**RESULT**

| ROLL_NO | NAME | ADDRESS | PHONE | AGE | EMP_NO | NAME | ADDRESS | PHONE | AGE |
|---|---|---|---|---|---|---|---|---|---|
| | RAM | GUR | 96524 | | | RA | DELH | 94551 | |
| 2 | ESH | GAON | 31543 | 18 | 1 | M | I | 23451 | 18 |
| | SUJI | ROHT | 91562 | | | RA | DELH | 94551 | |
| 3 | T | AK | 53131 | 20 | 1 | M | I | 23451 | 18 |
| | SUR | | 91567 | | | RA | DELH | 94551 | |
| 4 | ESH | DELHI | 68971 | 18 | 1 | M | I | 23451 | 18 |
| | | | 94551 | | NUL | NU | | | NU |
| 1 | RAM | DELHI | 23451 | 18 | L | LL | NULL | NULL | LL |

***Right Outer Join(⟖):*** When applying join on two relations R and S, some tuples of R or S does not appear in result set which does not satisfy the join conditions. But Right Outer Joins gives all tuples of S in the result set. The tuples of S which do not satisfy join condition will have values as NULL for attributes of R.

Example: Select students whose ROLL_NO is greater than EMP_NO of employees and details of other Employees as well

$$STUDENT \bowtie_{STUDENT.ROLL\_NO > EMPLOYEE.EMP\_NO} EMPLOYEE$$

**RESULT:**

| ROLL_NO | NAME | ADDRESS | PHONE | AGE | EMP_NO | NAME | ADDRESS | PHONE | AGE |
|---|---|---|---|---|---|---|---|---|---|
| | RAM | GAO GUR | 96524 | | | | DELH I | 94551 | |
| 2 | ESH | N | 31543 | 18 | 1 | RAM | | 23451 | 18 |
| | SUJI | ROHT | 91562 | | | | DELH I | 94551 | |
| 3 | T | AK | 53131 | 20 | 1 | RAM | | 23451 | 18 |
| | SUR | DELH | 91567 | | | | DELH | 94551 | |
| 4 | ESH | I | 68971 | 18 | 1 | RAM | I | 23451 | 18 |
| | NUL | | | NU | | NAR | HISA | 97829 | |
| NULL | L | NULL | NULL | LL | 5 | ESH | R | 18192 | 22 |
| | NUL | | | NU | | SWE | RAN | 98526 | |
| NULL | L | NULL | NULL | LL | 6 | TA | CHI | 17621 | 21 |
| | NUL | | | NU | | SUR | DELH | 91567 | |
| NULL | L | NULL | NULL | LL | 4 | ESH | I | 68971 | 18 |

***Full Outer Join(⟗):*** When applying join on two relations R and S, some tuples of R or S does not appear in result set which does not satisfy the join conditions. But Full Outer Joins gives all tuples of S and all tuples of R in the result set. The tuples of S which do not satisfy join condition will have values as NULL for attributes of R and vice versa.
Example:Select students whose ROLL_NO is greater than EMP_NO of employees and details of other Employees as well and other Students as well

$$STUDENT \bowtie_{STUDENT.ROLL\_NO > EMPLOYEE.EMP\_NO} EMPLOYEE$$

**RESULT:**

| ROLL_NO | NAME | ADDRESS | PHONE | AGE | EMP_NO | NAME | ADDRESS | PHONE | AGE |
|---|---|---|---|---|---|---|---|---|---|
| | RAM | GAO GUR | 96524 | | | | DELH I | 94551 | |
| 2 | ESH | N | 31543 | 18 | 1 | RAM | | 23451 | 18 |
| | SUJI | ROHT | 91562 | | | | DELH I | 94551 | |
| 3 | T | AK | 53131 | 20 | 1 | RAM | | 23451 | 18 |
| | SUR | DELH | 91567 | | | | DELH | 94551 | |
| 4 | ESH | I | 68971 | 18 | 1 | RAM | I | 23451 | 18 |
| | NUL | | | NU | | NAR | HISA | 97829 | |
| NULL | L | NULL | NULL | LL | 5 | ESH | R | 18192 | 22 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | NUL | | | NU | | SWE | RAN | 98526 | |
| NULL | L | NULL | NULL | LL | 6 | TA | CHI | 17621 | 21 |
| | NUL | | | NU | | SUR | DELH | 91567 | |
| NULL | L | NULL | NULL | LL | 4 | ESH | I | 68971 | 18 |
| | | DELH | 94551 | | NUL | NUL | | | NU |
| 1 | RAM | I | 23451 | 18 | L | L | NULL | NULL | LL |

***Division Operator (÷):*** Division operator A÷B can be applied if and only if:

- Attributes of B is proper subset of Attributes of A.
- The relation returned by division operator will have attributes = (All attributes of A – All Attributes of B)
- The relation returned by division operator will return those tuples from relation A which are associated to every B's tuple.

Consider the relation STUDENT_SPORTS and ALL_SPORTS given in Table 2 and Table 3 above.

To apply division operator as

**STUDENT_SPORTS÷ ALL_SPORTS**

- The operation is valid as attributes in ALL_SPORTS is a proper subset of attributes in STUDENT_SPORTS.
- The attributes in resulting relation will have attributes {ROLL_NO,SPORTS}-{SPORTS}=ROLL_NO
- The tuples in resulting relation will have those ROLL_NO which are associated with all B's tuple {Badminton, Cricket}. ROLL_NO 1 and 4 are associated to Badminton only. ROLL_NO 2 is associated to all tuples of B. So the resulting relation will be:

**ROLL_NO**
2

# Relational Calculus

- **Procedural Language** - Those Languages which clearly define how to get the required results from the Database are called Procedural Language. **Relational algebra** is a Procedural Language.
- **Declarative Language** - Those Language that only cares about What to get from the database without getting into how to get the results are called Declarative Language. **Relational Calculus** is a Declarative Language.

So Relational Calculus is a Declarative Language that uses Predicate Logic or First-Order Logic to determine the results from Database.

## Types of Relational Calculus in DBMS

### Relational Calculus is of Two Types:

- Tuple Relational Calculus (TRC)
- Domain Relational Calculus (DRC)

### Tuple Relational Calculus (TRC)

Tuple Relational Calculus in DBMS uses a tuple variable (t) that goes to each row of the table and checks if the predicate is true or false for the given row. Depending on the given predicate condition, it returns the row or part of the row.

### The Tuple Relational Calculus expression Syntax

```
{t | P(t)}
```

Where t is the tuple variable that runs over every Row, and P(t) is the predicate logic expression or condition.

Let's take an example of a Customer Database and try to see how TRC expressions work.

*Customer Table*

| Customer_id | Name | Zip code |
|---|---|---|
| 1 | Rohit | 12345 |
| 2 | Rahul | 13245 |
| 3 | Rohit | 56789 |
| 4 | Amit | 12345. |

**Example 1**: Write a TRC query to get all the data of customers whose zip code is

12345.

**TRC Query: {t \| t ∈ Customer ∧ t.Zipcode = 12345} or TRC Query: {t \| Customer(t) ∧ t[Zipcode] = 12345 }**

**Workflow of query** - The tuple variable "t" will go through every tuple of the Customer table. Each row will check whether the Cust_Zipcode is 12345 or not and only return those rows that satisfies the Predicate expression condition.

The TRC expression above can be read as **"Return all the tuple which belongs to the Customer Table and whose Zipcode is equal to 12345."**

**Result of the TRC expression above:**

**Result of the TRC expression above:**

| Customer_id | Name | Zip code |
|---|---|---|
| 1 | Rohit | 12345 |
| 4. | Amit | 12345 |

**Example 2**: Write a TRC query to get the customer id of all the Customers.

**TRC query: { t \| ∃s (s ∈ Customer ∧ s.Customer_id = t.customer_id) }**

**Result of the TRC Query:**

| Customer_id |
|---|
| 1 |
| 2 |
| 3 |
| 4 |

## Domain Relational Calculus (DRC)

Domain Relational Calculus uses domain Variables to get the column values required from the database based on the predicate expression or condition.

**The Domain realtional calculus expression syntax:**

```
{<x1,x2,x3,x4...> \| P(x1,x2,x3,x4...)}
```

**where,**

**<x1,x2,x3,x4...>** are domain variables used to get the column values required, and **P(x1,x2,x3...)** is predicate expression or condition.

Let's take the example of Customer Database and try to understand DRC queries with some examples.

*Customer Table*

| Customer_id | Name | Zip code |
|---|---|---|
| 1 | Rohit | 12345 |
| 2 | Rahul | 13245 |
| 3 | Rohit | 56789 |
| 4 | Amit | 12345 |

**Example 1**: Write a DRC query to get the data of all customers with Zip code 12345.

**DRC query: {<x1,x2,x3> \| <x1,x2> ∈ Customer ∧ x3 = 12345 }**

# SQL Commands

- SQL commands are instructions. It is used to communicate with the database. It is also used to perform specific tasks, functions, and queries of data.
- SQL can perform various tasks like create a table, add data to tables, drop the table, modify the table, set permission for users.

## Types of SQL Commands

There are five types of SQL commands: DDL, DML, DCL, TCL, and DQL.

### 1. Data Definition Language (DDL)

- DDL changes the structure of the table like creating a table, deleting a table, altering a table, etc.
- All the command of DDL are auto-committed that means it permanently save all the changes in the database.

Here are some commands that come under DDL:

- CREATE
- ALTER
- DROP
- TRUNCATE

**a. CREATE** It is used to create a new table in the database.

**Syntax**

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

**Example:**

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

**b. DROP:** It is used to delete both the structure and record stored in the table.

**Syntax**

- DROP TABLE table_name;

**Example**

- DROP TABLE EMPLOYEE;

**c. ALTER:** It is used to alter the structure of the database. This change could be either to modify the characteristics of an existing attribute or probably to add a new attribute.

**Syntax:**

To add a new column in the table

- ALTER TABLE table_name ADD column_name COLUMN-definition;

To modify existing column in the table:

- ALTER TABLE table_name MODIFY(column_definitions....);

**EXAMPLE**

- ALTER TABLE STU_DETAILS ADD(ADDRESS VARCHAR2(20));
- ALTER TABLE STU_DETAILS MODIFY (NAME VARCHAR2(20));

**d. TRUNCATE:** It is used to delete all the rows from the table and free the space

containing the table.

**Syntax:**

- TRUNCATE TABLE table_name;

**Example:**

TRUNCATE TABLE EMPLOYEE;

## 2. Data Manipulation Language

- DML commands are used to modify the database. It is responsible for all form of changes in the database.
- The command of DML is not auto-committed that means it can't permanently save all the changes in the database. They can be rollback.

Here are some commands that come under DML:

- INSERT
- UPDATE
- DELETE

**a. INSERT:** The INSERT statement is a SQL query. It is used to insert data into the row of a table.

**Syntax:**

- INSERT INTO TABLE_NAME
- (col1, col2, col3,.... col N)
- VALUES (value1, value2, value3, .... valueN);

Or

- INSERT INTO TABLE_NAME
- VALUES (value1, value2, value3, .... valueN);

**For example:**

- INSERT INTO javatpoint (Author, Subject) VALUES ("Sonoo", "DBMS");

**b. UPDATE:** This command is used to update or modify the value of a column in the table.

**Syntax:**

- UPDATE table_name SET [column_name1

= value1,...column_nameN = valueN] [WHERE CONDITION]

**For example:**

- UPDATE students
- SET User_Name = 'Sonoo'
- WHERE Student_Id = '3'

**c. DELETE:** It is used to remove one or more row from a table.

**Syntax:**

- DELETE FROM table_name [WHERE condition];

**For example:**

- DELETE FROM javatpoint
- WHERE Author="Sonoo";

## 3. Data Control Language

DCL commands are used to grant and take back authority from any database user.

Here are some commands that come under DCL:

- Grant
- Revoke

**a. Grant:** It is used to give user access privileges to a database.

**Example**

GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USER;

**b. Revoke:** It is used to take back permissions from the user.

**Example**

- REVOKE SELECT, UPDATE ON MY_TABLE FROM USER1, USER2;

## 4. Transaction Control Language

TCL commands can only use with DML commands like INSERT, DELETE and UPDATE only.

These operations are automatically committed in the database that's why they cannot be used while creating tables or dropping them.

Here are some commands that come under TCL:

- COMMIT
- ROLLBACK
- SAVEPOINT

**a. Commit:** Commit command is used to save all the transactions to the database.

**Syntax:**

- COMMIT;

**Example:**

- DELETE FROM CUSTOMERS

- WHERE AGE = 25;
- COMMIT;

**b. Rollback:** Rollback command is used to undo transactions that have not already been saved to the database.

**Syntax:**

ROLLBACK;

**Example:**

- DELETE FROM CUSTOMERS
- WHERE AGE = 25;
- ROLLBACK;

**c. SAVEPOINT:** It is used to roll the transaction back to a certain point without rolling back the entire transaction.

**Syntax:**

- SAVEPOINT SAVEPOINT_NAME;

## 5. Data Query Language

DQL is used to fetch the data from the database.

It uses only one command:

- SELECT

**a. SELECT:** This is the same as the projection operation of relational algebra. It is used to select the attribute based on the condition described by WHERE clause.

**Syntax:**

- SELECT expressions
- FROM TABLES

- WHERE conditions;

**For example:**

- SELECT emp_name
- FROM employee
- WHERE age > 20;

# Characteristics of SQL

- SQL is easy to learn.
- SQL is used to access data from relational database management systems.
- SQL can execute queries against the database.
- SQL is used to describe the data.
- SQL is used to define the data in the database and manipulate it when needed.
- SQL is used to create and drop the database and table.
- SQL is used to create a view, stored procedure, function in a database.
- SQL allows users to set permissions on tables, procedures, and views.

**Advantages of SQL :**

SQL has many advantages which makes it popular and highly demanded. It is a reliable and efficient language used for communicating with the database. Some advantages of SQL are as follows:

- **Faster Query Processing –**
  Large amount of data is retrieved quickly and efficiently. Operations like Insertion, deletion, manipulation of data is also done in almost no time.

- **No Coding Skills –**
  For data retrieval, large number of lines of code is not required. All basic keywords such as SELECT, INSERT INTO, UPDATE, etc are used and also the syntactical rules are not complex in SQL, which makes it a user-friendly language.

- **Standardized Language –**

Due to documentation and long establishment over years, it provides a uniform platform worldwide to all its users.

- **Portable –**
  It can be used in programs in PCs, server, laptops independent of any platform (Operating System, etc). Also, it can be embedded with other applications as per need/requirement/use.

- **Interactive Language –**
  Easy to learn and understand, answers to complex queries can be received in seconds.

- **Multiple data views –**

# SQL Data Types

Data types are used to represent the nature of the data that can be stored in the database table. For example, in a particular column of a table, if we want to store a string type of data then we will have to declare a string data type of this column.

Data types mainly classified into three categories for every database.

- String Data types
- Numeric Data types
- Date and time Data types

## Data Types in MySQL, SQL Server and Oracle Databases

### MySQL Data Types

| | |
|---|---|
| **CHAR(Size)** | It is used to specify a fixed length string that can contain numbers, letters, and special characters. Its size can be 0 to 255 characters. Default is 1. |
| **VARCHAR(Size)** | It is used to specify a variable length string that can contain numbers, letters, and special characters. Its size can be from 0 to 65535 characters. |
| **BINARY(Size)** | It is equal to CHAR() but stores binary byte strings. Its size parameter specifies the column length in the bytes. Default is 1. |

| | |
|---|---|
| **VARBINARY(Size)** | It is equal to VARCHAR() but stores binary byte strings. Its size parameter specifies the maximum column length in bytes. |
| **TEXT(Size)** | It holds a string that can contain a maximum length of 255 characters. |
| **TINYTEXT** | It holds a string with a maximum length of 255 characters. |
| **MEDIUMTEXT** | It holds a string with a maximum length of 16,777,215. |
| **LONGTEXT** | It holds a string with a maximum length of 4,294,967,295 characters. |
| **ENUM(val1, val2, val3,...)** | It is used when a string object having only one value, chosen from a list of possible values. It contains 65535 values in an ENUM list. If you insert a value that is not in the list, a blank value will be inserted. |
| **SET( val1,val2,val 3,....)** | It is used to specify a string that can have 0 or more values, chosen from a list of possible values. You can list up to 64 values at one time in a SET list. |
| **BLOB(size)** | It is used for BLOBs (Binary Large Objects). It can hold up to 65,535 bytes. |

A list of data types used in MySQL database.

#### MySQL Numeric Data Types

### MySQL Numeric Data Types

| | |
|---|---|
| **BIT(Size)** | It is used for a bit-value type. The number of bits per value is specified in<br>size. Its size can be 1 to 64. The default value is 1. |
| **INT(size)** | It is used for the integer value. Its signed range varies<br>from -2147483648<br>to 2147483647 and unsigned range varies from 0 to 4294967295.<br>The size parameter specifies the max display width that is 255. |
| **INTEGER(size)** | It is equal to INT(size). |
| **FLOAT(size, d)** | It is used to specify a floating point number. Its size parameter specifies the total number of digits. The number of digits after the decimal point specified by **d** parameter. |
| **FLOAT(p)** | It is used to specify a floating point number. MySQL used p parameter to<br>determine whether to use FLOAT or DOUBLE. If p is between 0 to24, the<br>data type becomes FLOAT (). If p is from 25 to 53, the data type becomes<br>DOUBLE(). |

| | |
|---|---|
| **DOUBLE(size, d)** | It is a normal size floating point number. Its size parameter specifies the total number of digits. The number of digits after the decimal is specified by d parameter. |
| **DECIMAL(size, d)** | It is used to specify a fixed point number. Its size parameter specifies the total number of digits. The number of digits after the decimal parameter is specified by **d** parameter. The maximum value for the size is 65, and the default value is 10. The maximum value for **d** is 30, and the default value is 0. |
| **DEC(size, d)** | It is equal to DECIMAL(size, d). |
| **BOOL** | It is used to specify Boolean values true and false. Zero is considered as false, and nonzero values are considered as true. |
| **DATE** | It is used to specify date format YYYY-MM-DD. Its supported range is from '1000-01-01' to '9999-12-31'. |
| **DATETIME(fsp)** | It is used to specify date and time combination. Its format is YYYY-MM-DD Its supported range is from '1000-01-01 00:00:00' to 9999-12-31 23:59:59 |
| **TIMESTAMP(fsp)** | It is used to specify the timestamp. Its value is stored as the number of sec since the Unix epoch('1970-01-01 00:00:00' UTC). Its format is YYYY-MM-D Its supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03: |
| **TIME(fsp)** | It is used to specify the time format. Its format is hh:mm:ss. Its supported r from '-838:59:59' to '838:59:59' |
| **YEAR** | It is used to specify a year in four-digit format. Values allowed in four digit from 1901 to 2155, and 0000. |

# SQL Operators

Operators are the foundation of any programming language. We can define operators as symbols that help us to perform specific mathematical and logical computations on operands. In other words, we can say that an operator operates the operands. SQL operators have three different categories.

- Arithmetic operator

- Comparison operator
- Logical operator

## Arithmetic operators:

We can use various arithmetic operators on the data stored in the tables. Arithmetic Operators are:

| Operator | Description |
|---|---|
| + | The addition is used to perform an addition operation on the data values. |
| – | This operator is used for the subtraction of the data values. |
| / | This operator works with the 'ALL' keyword and it calculates division operations. |
| * | This operator is used for multiply data values. |
| % | Modulus is used to get the remainder when data is divided by another. |

## Comparison operators:

Another important operator in SQL is a comparison operator, which is used to compare one expression's value to other expressions. SQL supports different types of the comparison operator, which is described below:

| Operator | Description |
|---|---|
| = | Equal to. |
| > | Greater than. |
| < | Less than. |
| >= | Greater than equal to. |
| <= | Less than equal to. |
| <> | Not equal to. |

## Logical operators:

The Logical operators are those that are true or false. They return true or false values to combine one or more true or false values.

| Operator | Description |
|---|---|
| AND | Logical AND compares between two Booleans as expressions and returns true when both expressions are true. |

| | |
|---|---|
| OR | Logical OR compares between two Booleans as expressions and returns true when one of the expressions is true. |
| NOT | Not takes a single Boolean as an argument and changes its value from false to true or from true to false. |

Special operators:

| Operator | Description |
|---|---|
| ALL | ALL is used to select all records of a SELECT STATEMENT. It compares a value to every value in a list ofof results from a query. The ALL must be preceded by the comparison operators and evaluates to TRUE if the query returns no rows. |
| ANY | ANY compares a value to each value in a list of results from a query and evaluates to true if the result of an inner query contains at least one row. |
| BETWEEN | The SQL BETWEEN operator tests an expression against a range. The range consists of a beginning, followed by an AND keyword and an end expression. |
| IN | The IN operator checks a value within a set of values separated by commas and retrieves the rows from the table which are matching. |
| EXISTS | The EXISTS checks the existence of a result of a subquery. The EXISTS subquery tests whether a subquery fetches at least one row. When no data is returned then this operator returns 'FALSE'. |
| SOME | SOME operator evaluates the condition between the outer and inner tables and evaluates to true if the final result returns any one row. If not, then it evaluates to false. |
| UNIQUE | UNIQUE operator searches every unique row of a specified table. |

# SQL INDEX Keyword

## CREATE INDEX

The `CREATE INDEX` command is used to create indexes in tables (allows duplicate values).

Indexes are used to retrieve data from the database very fast. The users

cannot see the indexes, they are just used to speed up searches/queries.

The following SQL creates an index named "idx_lastname" on the "LastName" column in the "Persons" table:

```
CREATE INDEX idx_lastname
ON Persons (LastName);
```

If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas:

```
CREATE INDEX idx_pname
ON Persons (LastName, FirstName);
```

# DROP INDEX

The `DROP INDEX` command is used to delete an index in a table.

**MySQL:**

```
ALTER TABLE table_name
DROP INDEX index_name;
```

# SQL Views

## SQL CREATE VIEW Statement

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.

You can add SQL statements and functions to a view and present the

data as if the data were coming from one single table.

A view is created with the `CREATE VIEW` statement.

## CREATE VIEW Syntax

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

# SQL Aggregate Functions

### Aggregate functions in SQL

As the Basic SQL Tutorial points out, SQL is excellent at aggregating data the way you might in a pivot table in Excel. You will use aggregate functions all the time, so it's important to get comfortable with them. The functions themselves are the same ones you will find in Excel or any other analytics program. We'll cover them individually in the next few lessons. Here's a quick preview:

- `COUNT` counts how many rows are in a particular column.
- `SUM` adds together all the values in a particular column.
- `MIN` and `MAX` return the lowest and highest values in a particular column, respectively.
- `AVG` calculates the average of a group of selected values.

### COUNT() Function

The COUNT() function returns the number of rows in a database table.

Syntax:

COUNT(*)

or

COUNT( [ALL|DISTINCT] expression )

Example:

We will use the ‘products’ table from the sample database for our demonstration.

| product_id | name | quantity_in_stock | unit_price |
|---|---|---|---|
| 1 | Foam Dinner Plate | 70 | 1.21 |
| 2 | Pork - Bacon,back Peameal | 49 | 4.65 |
| 3 | Lettuce - Romaine, Heart | 38 | 3.35 |
| 4 | Brocolinni - Gaylan, Chinese | 90 | 4.53 |
| 5 | Sauce - Ranch Dressing | 94 | 1.63 |
| 6 | Petit Baguette | 14 | 2.39 |
| 7 | Sweet Pea Sprouts | 98 | 3.29 |
| 8 | Island Oasis - Raspberry | 26 | 0.74 |
| 9 | Longan | 67 | 2.26 |
| 10 | Broom - Push | 6 | 1.09 |

The following SQL statement fetches the number of products in the table.

```
SELECT COUNT(product_id)
FROM Products;
```

This will produce the following result.

| | COUNT(product_id) |
|---|---|
| ▶ | 10 |

The below-given command will display those product ids where the unit price is greater than 4.

```
SELECT COUNT(product_id)
FROM Products
WHERE unit_price > 4;
```

This will display the following result.

| | COUNT(product_id) |
|---|---|
| ▶ | 2 |

Let's look at how we can use GROUP BY and HAVING functions with the COUNT function.

Consider the following dataset:

| customer_id | first_name | last_name | birth_date | phone | address |
|---|---|---|---|---|---|
| 1 | Alfred | MacCaffrey | 1986-03-28 | 781-932-9754 | 0 Sage Terrace |
| 2 | Ines | Brushfield | 1986-04-13 | 804-427-9456 | 14187 Commerc |
| 3 | Freddi | Boagey | 1985-02-07 | 719-724-7869 | 251 Springs Jun |
| 4 | Ambur | Roseburgh | 1974-04-14 | 407-231-8017 | 30 Arapahoe Te |
| 5 | Clemmie | Betchley | 1973-11-07 | NULL | 5 Spohn Circle |
| 6 | Elka | Twiddell | 1991-09-04 | 312-480-8498 | 7 Manley Drive |
| 7 | Ilene | Dowson | 1964-08-30 | 615-641-4759 | 50 Lillian Crossin |
| 8 | Thacher | Naseby | 1993-07-17 | 941-527-3977 | 538 Mosinee Ce |
| 9 | Romola | Rumgay | 1992-05-23 | 559-181-3744 | 3520 Ohio Trail |
| 10 | Levy | Mynett | 1969-10-13 | 404-246-3370 | 68 Lawn Avenue |
| 11 | Sam | Tully | NULL | NULL | 16 commecial ce |
| 12 | Brian | Ross | NULL | NULL | 43 Bakers Stree |
| NULL | NULL | NULL | NULL | NULL | NULL |

The SQL command given below will list the number of customers in each city.

```
SELECT COUNT(customer_id), city
FROM Customers
GROUP BY city
HAVING COUNT(customer_id) > 0;
```

This will produce the following results:

| COUNT(customer_id) | city |
|---|---|
| 2 | Hampton |
| 1 | Colorado Springs |
| 1 | Orlando |
| 1 | Arlington |
| 1 | Chicago |
| 1 | Nashville |
| 1 | Sarasota |
| 1 | Visalia |
| 1 | Atlanta |
| 2 | New York |

## SUM() Function

The SUM() function returns the total sum of a numeric column.

Syntax:

SUM()

or

SUM( [ALL|DISTINCT] expression )

Example:

The following SQL statement finds the sum of the "unit price" fields in the "products" table:

```
SELECT SUM(unit_price)
FROM products;
```

This will produce the following result.

Let’s look at how we can use GROUP BY and HAVING functions with the SUM function.

Consider the following dataset:

| customer_id | first_name | last_name | birth_date | phone | address |
|---|---|---|---|---|---|
| 1 | Alfred | MacCaffrey | 1986-03-28 | 781-932-9754 | 0 Sage Terrace |
| 2 | Ines | Brushfield | 1986-04-13 | 804-427-9456 | 14187 Commerc |
| 3 | Freddi | Boagey | 1985-02-07 | 719-724-7869 | 251 Springs Jun |
| 4 | Ambur | Roseburgh | 1974-04-14 | 407-231-8017 | 30 Arapahoe Te |
| 5 | Clemmie | Betchley | 1973-11-07 | NULL | 5 Spohn Circle |
| 6 | Elka | Twiddell | 1991-09-04 | 312-480-8498 | 7 Manley Drive |
| 7 | Ilene | Dowson | 1964-08-30 | 615-641-4759 | 50 Lillian Crossin |
| 8 | Thacher | Naseby | 1993-07-17 | 941-527-3977 | 538 Mosinee Ce |
| 9 | Romola | Rumgay | 1992-05-23 | 559-181-3744 | 3520 Ohio Trail |
| 10 | Levy | Mynett | 1969-10-13 | 404-246-3370 | 68 Lawn Avenue |
| 11 | Sam | Tully | NULL | NULL | 16 commecial ce |
| 12 | Brian | Ross | NULL | NULL | 43 Bakers Stree |
| NULL | NULL | NULL | NULL | NULL | NULL |

The SQL command below will list the number of customers in each city, having a sum of points greater than 3000.

```
SELECT COUNT(customer_id), city
FROM Customers
GROUP BY city
HAVING SUM(points) > 3000;
```

This will produce the following result:

| COUNT(customer_id) | city |
|---|---|
| 2 | Hampton |
| 1 | Arlington |
| 1 | Chicago |
| 2 | New York |

## AVG() Function

The AVG() function calculates the average of a set of values.

Syntax:

AVG()

or

AVG( [ALL|DISTINCT] expression )

Example:

The following SQL command calculates the average quantity in stock.

```
SELECT AVG(quantity_in_stock)
FROM Products;
```

This will produce the following result.

| | AVG(quantity_in_stock) |
|---|---|
| ▶ | 55.2000 |

## MIN() Function

The MIN() aggregate function returns the lowest value (minimum) in a set of non-NULL

values.

Syntax:

MIN()

or

MIN( [ALL|DISTINCT] expression )

Example:

```
SELECT MIN(quantity_in_stock)
FROM products;
```

The above code will give us the minimum quantity in stock in the products table.

| | MIN(quantity_in_stock) |
|---|---|
| ▶ | 6 |

Also Read: How to Aggregate Data Using Group By in SQL?

## MAX() Function

The MAX() aggregate function returns the highest value (maximum) in a set of non-NULL values.

Syntax:

AVG()

or

AVG( [ALL|DISTINCT] expression )

Example:

The code depicted below will give us the maximum quantity in stock in the products table.

```
SELECT MAX(quantity_in_stock)
FROM products;
```

This will produce the following result.

| | MAX(quantity_in_stock) |
|---|---|
| ▶ | 98 |

# SQL JOIN

A `JOIN` clause is used to combine rows from two or more tables, based on a related column between them.

| OrderID | CustomerID | OrderDate |
|---|---|---|
| 10308 | 2 | 1996-09-18 |
| 10309 | 37 | 1996-09-19 |
| 10310 | 77 | 1996-09-20 |

Let's look at a selection from the "Orders" table:

Then, look at a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName |
|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo |
| 3 | Antonio Moreno Taquería | Antonio Moreno |

Notice that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.

Then, we can create the following SQL statement (that contains an `INNER JOIN`), that selects records that have matching values in both tables:

# The SQL UNION Operator

The `UNION` operator is used to combine the result-set of two or more `SELECT` statements.

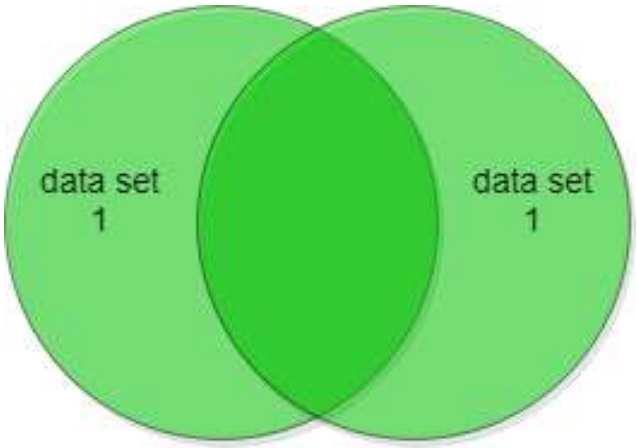
- Every `SELECT` statement within `UNION` must have the same number of columns
- The columns must also have similar data types
- The columns in every `SELECT` statement must also be in the same order

## UNION Syntax

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

# SQL | INTERSECT Clause

The INTERSECT clause in SQL is used to combine two SELECT statements but the dataset returned by the INTERSECT statement will be the intersection of the data-sets of the two SELECT statements. In simple words, the INTERSECT statement will return only those rows which will be common to both of the SELECT statements.
Pictorial Representation:




The INTERSECT statement will return only those rows present in the red shaded region. i.e. common to both of the data-sets.

**Note**: The number and type of fields present in the two data-sets must be same and similar.
**Syntax**:

```
SELECT column1 , column2 ....
FROM table_names
WHERE condition

INTERSECT

SELECT column1 , column2 ....
FROM table_names
WHERE condition
```

**Sample Tables**:

Customers Table:

| ID | Name | Address | Age |
|---|---|---|---|
| 1 | Harsh | Delhi | 20 |
| 2 | Pratik | Mumbai | 21 |
| 3 | Akash | Kolkata | 35 |
| 4 | Varun | Madras | 30 |
| 5 | Souvik | Banaras | 25 |
| 6 | Dhanraj | Siliguri | 22 |
| 7 | Riya | Chennai | 19 |

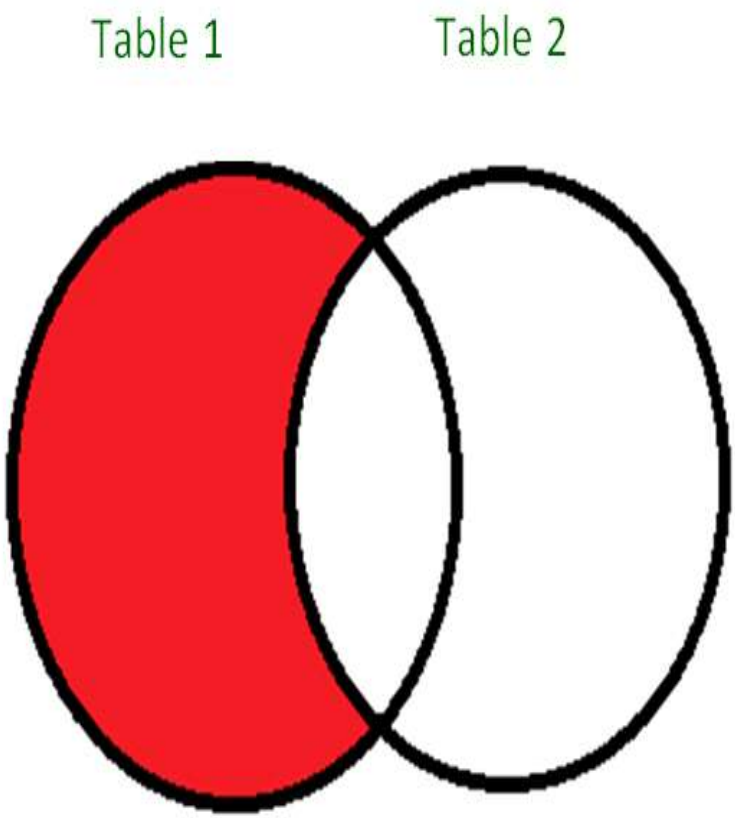
Orders Table:

| Oid | Date | Customer_id |
|---|---|---|
| 102 | 2017-10-08 | 3 |
| 100 | 2017-10-08 | 3 |
| 101 | 2017-11-20 | 2 |
| 103 | 2016-5-20 | 4 |

# SQL | MINUS Operator

The Minus Operator in SQL is used with two SELECT statements. The MINUS operator is used to subtract the result set obtained by first SELECT query from the result set obtained by second SELECT query. In simple words, we can say that MINUS operator will return only those rows which are unique in only first SELECT query and not those rows which are common to both first and second SELECT queries.

**Pictorial Representation**:




As you can see is in the above diagram, the MINUS operator will return only those rows which are present in the result set from Table1 and not present in the result set of Table2.

**Basic Syntax**:

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

FROM table_name

WHERE condition

MINUS

SELECT column1 , column2 , ... columnN

FROM table_name

WHERE condition;
```

**columnN**: column1, column2.. are the name of columns of the table.

**Important Points**:

- The WHERE clause is optional in the above query.

- The number of columns in both SELECT statements must be same.
- The data type of corresponding columns of both SELECT statement must be same.
    - Table1

Table 1

| Name | Address | Age | Gra |
|---|---|---|---|
| Harsh | Delhi | 20 | A |
| Gaurav | Jaipur | 21 | B |
| Pratik | Mumbai | 21 | A |
| Dhanraj | Kolkata | 22 | B |

Table 2

| Name | Age | Phone | Grade |
|---|---|---|---|
| Akash | 20 | xxxxxxxxxx | A |
| Dhiraj | 21 | xxxxxxxxxx | B |
| Vaibhav | 21 | xxxxxxxxxx | A |
| Dhanraj | 22 | xxxxxxxxxx | B |

- **Queries**:
- `SELECT NAME, AGE , GRADE`
- `FROM Table1`
- `MINUS`

- `SELECT NAME, AGE, GRADE`
- `FROM Table2`
- Output:
  The above query will return only those rows which are unique in 'Table1'. We can clearly see that values in the fields NAME, AGE and GRADE for the last row in both tables are same. Therefore, the output will be the first three rows from Table1. The obtained output is shown below:

| Name | Age | Grade |
|---|---|---|
| Harsh | 20 | A |
| Gaurav | 21 | B |
| Pratik | 21 | A |

# What is Cursor in SQL ?

**Cursor** is a Temporary Memory or Temporary Work Station. It is Allocated by Database Server at the Time of Performing DML(Data Manipulation Language) operations on Table by User. Cursors are used to store Database Tables. There are 2 types of Cursors: Implicit Cursors, and Explicit Cursors. These are explained as following below.

- **Implicit Cursors:**
  Implicit Cursors are also known as Default Cursors of SQL SERVER. These Cursors are allocated by SQL SERVER when the user performs DML operations.
- **Explicit Cursors :**
  Explicit Cursors are Created by Users whenever the user requires them. Explicit Cursors are used for Fetching data from Table in Row-By-Row Manner.

**How to create Explicit Cursor:**

- **Declare Cursor Object.**
  **Syntax :** DECLARE cursor_name CURSOR FOR SELECT * FROM table_name
  `DECLARE s1 CURSOR FOR SELECT * FROM studDetails`

**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]
```