**What are SQL Functions?**

SQL provides many built-in functions to perform operations on data. These functions are useful while performing mathematical calculations, string concatenations, sub-strings etc. SQL functions are divided into two categories,

1. Aggregate Functions
2. Scalar Functions

**Aggregate Functions**

These functions **return a single value** after performing calculations on a group of values. Following are some of the frequently used Aggregrate functions.

**AVG() Function**

Average returns average value after calculating it from values in a numeric column.

Its general **syntax** is,

SELECT AVG(column\_name) FROM table\_name

**Using AVG() function**

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query to find average salary will be,

SELECT avg(salary) from Emp;

Result of the above query will be,

|  |
| --- |
| **avg(salary)** |
| 8200 |

**COUNT() Function**

Count returns the number of rows present in the table either based on some condition or without condition.

Its general **syntax** is,

SELECT COUNT(column\_name) FROM table-name

**Using COUNT() function**

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query to count employees, satisfying specified condition is,

SELECT COUNT(name) FROM Emp WHERE salary = 8000;

Result of the above query will be,

|  |
| --- |
| **count(name)** |
| 2 |

**Example of COUNT(distinct)**

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query is,

SELECT COUNT(DISTINCT salary) FROM emp;

Result of the above query will be,

|  |
| --- |
| **count(distinct salary)** |
| 4 |

**FIRST() Function**

First function returns first value of a selected column

**Syntax** for FIRST function is,

SELECT FIRST(column\_name) FROM table-name;

**Using FIRST() function**

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query will be,

SELECT FIRST(salary) FROM Emp;

and the result will be,

|  |
| --- |
| **first(salary)** |
| 9000 |

**LAST() Function**

LAST function returns the return last value of the selected column.

**Syntax** of LAST function is,

SELECT LAST(column\_name) FROM table-name;

**Using LAST() function**

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query will be,

SELECT LAST(salary) FROM emp;

Result of the above query will be,

|  |
| --- |
| **last(salary)** |
| 8000 |

**MAX() Function**

MAX function returns maximum value from selected column of the table.

**Syntax** of MAX function is,

SELECT MAX(column\_name) from table-name;

**Using MAX() function**

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query to find the Maximum salary will be,

SELECT MAX(salary) FROM emp;

Result of the above query will be,

|  |
| --- |
| **MAX(salary)** |
| 10000 |

**MIN() Function**

MIN function returns minimum value from a selected column of the table.

**Syntax** for MIN function is,

SELECT MIN(column\_name) from table-name;

**Using MIN() function**

Consider the following **Emp** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query to find minimum salary is,

SELECT MIN(salary) FROM emp;

Result will be,

|  |
| --- |
| **MIN(salary)** |
| 6000 |

**SUM() Function**

SUM function returns total sum of a selected columns numeric values.

**Syntax** for SUM is,

SELECT SUM(column\_name) from table-name;

**Using SUM() function**

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query to find sum of salaries will be,

SELECT SUM(salary) FROM emp;

Result of above query is,

|  |
| --- |
| **SUM(salary)** |
| 41000 |

**Scalar Functions**

Scalar functions return a single value from an input value. Following are some frequently used Scalar Functions in SQL.

**UCASE() Function**

UCASE function is used to convert value of string column to Uppercase characters.

**Syntax** of UCASE,

SELECT UCASE(column\_name) from table-name;

**Using UCASE() function**

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | anu | 22 | 9000 |
| 402 | shane | 29 | 8000 |
| 403 | rohan | 34 | 6000 |
| 404 | scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query for using UCASE is,

SELECT UCASE(name) FROM emp;

Result is,

|  |
| --- |
| **UCASE(name)** |
| ANU |
| SHANE |
| ROHAN |
| SCOTT |
| TIGER |

**LCASE() Function**

LCASE function is used to convert value of string columns to Lowecase characters.

**Syntax** for LCASE is,

SELECT LCASE(column\_name) FROM table-name;

**Using LCASE() function**

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | SCOTT | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query for converting string value to Lower case is,

SELECT LCASE(name) FROM emp;

Result will be,

|  |
| --- |
| **LCASE(name)** |
| anu |
| shane |
| rohan |
| scott |
| tiger |

**MID() Function**

MID function is used to extract substrings from column values of string type in a table.

**Syntax** for MID function is,

SELECT MID(column\_name, start, length) from table-name;

**Using MID() function**

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | anu | 22 | 9000 |
| 402 | shane | 29 | 8000 |
| 403 | rohan | 34 | 6000 |
| 404 | scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SQL query will be,

SELECT MID(name,2,2) FROM emp;

Result will come out to be,

|  |
| --- |
| **MID(name,2,2)** |
| nu |
| ha |
| oh |
| co |
| ig |

**ROUND() Function**

ROUND function is used to round a numeric field to number of nearest integer. It is used on Decimal point values.

**Syntax** of Round function is,

SELECT ROUND(column\_name, decimals) from table-name;

**Using ROUND() function**

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | anu | 22 | 9000.67 |
| 402 | shane | 29 | 8000.98 |
| 403 | rohan | 34 | 6000.45 |
| 404 | scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000.01 |

SQL query is,

SELECT ROUND(salary) from emp;

Result will be,

|  |
| --- |
| **ROUND(salary)** |
| 9001 |
| 8001 |
| 6000 |
| 10000 |
| 8000 |

**SQL JOIN**

SQL Join is used to fetch data from two or more tables, which is joined to appear as single set of data. It is used for combining column from two or more tables by using values common to both tables.

JOIN Keyword is used in SQL queries for joining two or more tables. Minimum required condition for joining table, is **(n-1)** where **n**, is number of tables. A table can also join to itself, which is known as, **Self Join**.

**Types of JOIN**

Following are the types of JOIN that we can use in SQL:

* Inner
* Outer
* Left
* Right

**Cross JOIN or Cartesian Product**

This type of JOIN returns the cartesian product of rows from the tables in Join. It will return a table which consists of records which combines each row from the first table with each row of the second table.

Cross JOIN Syntax is,

SELECT column-name-list

FROM

table-name1 CROSS JOIN table-name2;

**Example of Cross JOIN**

Following is the **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 4 | alex |

and the **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

Cross JOIN query will be,

SELECT \* FROM

class CROSS JOIN class\_info;

The resultset table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 1 | DELHI |
| 4 | alex | 1 | DELHI |
| 1 | abhi | 2 | MUMBAI |
| 2 | adam | 2 | MUMBAI |
| 4 | alex | 2 | MUMBAI |
| 1 | abhi | 3 | CHENNAI |
| 2 | adam | 3 | CHENNAI |
| 4 | alex | 3 | CHENNAI |

As you can see, this join returns the cross product of all the records present in both the tables.

**INNER Join or EQUI Join**

This is a simple JOIN in which the result is based on matched data as per the equality condition specified in the SQL query.

Inner Join Syntax is,

SELECT column-name-list FROM

table-name1 INNER JOIN table-name2

WHERE table-name1.column-name = table-name2.column-name;

**Example of INNER JOIN**

Consider a **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |

and the **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

**Inner** JOIN query will be,

SELECT \* from class INNER JOIN class\_info where class.id = class\_info.id;

The resultset table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |

**Natural JOIN**

Natural Join is a type of Inner join which is based on column having same name and same datatype present in both the tables to be joined.

The syntax for Natural Join is,

SELECT \* FROM

table-name1 NATURAL JOIN table-name2;

**Example of Natural JOIN**

Here is the **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |

and the **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |

**Natural join query will be,**

SELECT \* from class NATURAL JOIN class\_info;

The resultset table will look like,

|  |  |  |
| --- | --- | --- |
| **ID** | **NAME** | **Address** |
| 1 | abhi | DELHI |
| 2 | adam | MUMBAI |
| 3 | alex | CHENNAI |

In the above example, both the tables being joined have **ID** column(same name and same datatype), hence the records for which value of **ID** matches in both the tables will be the result of Natural Join of these two tables.

**OUTER JOIN**

Outer Join is based on both matched and unmatched data. Outer Joins subdivide further into,

1. Left Outer Join
2. Right Outer Join
3. Full Outer Join

**LEFT Outer Join**

The left outer join returns a resultset table with the **matched data** from the two tables and then the remaining rows of the **left** table and null from the **right** table's columns.

Syntax for Left Outer Join is,

SELECT column-name-list FROM

table-name1 LEFT OUTER JOIN table-name2

ON table-name1.column-name = table-name2.column-name;

To specify a condition, we use the ON keyword with Outer Join.

Left outer Join Syntax for **Oracle** is,

SELECT column-name-list FROM

table-name1, table-name2 on table-name1.column-name = table-name2.column-name(+);

**Example of Left Outer Join**

Here is the **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

and the **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Left Outer Join** query will be,

SELECT \* FROM class LEFT OUTER JOIN class\_info ON (class.id = class\_info.id);

The resultset table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| 4 | anu | null | null |
| 5 | ashish | null | null |

**RIGHT Outer Join**

The right outer join returns a resultset table with the **matched data** from the two tables being joined, then the remaining rows of the **right** table and null for the remaining **left** table's columns.

Syntax for Right Outer Join is,

SELECT column-name-list FROM

table-name1 RIGHT OUTER JOIN table-name2

ON table-name1.column-name = table-name2.column-name;

Right outer Join Syntax for **Oracle** is,

SELECT column-name-list FROM

table-name1, table-name2

ON table-name1.column-name(+) = table-name2.column-name;

**Example of Right Outer Join**

Once again the **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

and the **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Right Outer Join** query will be,

SELECT \* FROM class RIGHT OUTER JOIN class\_info ON (class.id = class\_info.id);

The resultant table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| null | null | 7 | NOIDA |
| null | null | 8 | PANIPAT |

**Full Outer Join**

The full outer join returns a resultset table with the **matched data** of two table then remaining rows of both **left** table and then the **right** table.

Syntax of Full Outer Join is,

SELECT column-name-list FROM

table-name1 FULL OUTER JOIN table-name2

ON table-name1.column-name = table-name2.column-name;

**Example of Full outer join is,**

The **class** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | abhi |
| 2 | adam |
| 3 | alex |
| 4 | anu |
| 5 | ashish |

and the **class\_info** table,

|  |  |
| --- | --- |
| **ID** | **Address** |
| 1 | DELHI |
| 2 | MUMBAI |
| 3 | CHENNAI |
| 7 | NOIDA |
| 8 | PANIPAT |

**Full Outer Join** query will be like,

SELECT \* FROM class FULL OUTER JOIN class\_info ON (class.id = class\_info.id);

The resultset table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| 4 | anu | null | null |
| 5 | ashish | null | null |
| null | null | 7 | NOIDA |
| null | null | 8 | PANIPAT |

**SQL VIEW**

A VIEW in SQL is a logical subset of data from one or more tables. View is used to restrict data access.

Syntax for creating a View,

CREATE or REPLACE VIEW view\_name

AS

SELECT column\_name(s)

FROM table\_name

WHERE condition

As you may have understood by seeing the above SQL query, a view is created using data fetched from some other table(s). It's more like a temporary table created with data.

**Creating a VIEW**

Consider following **Sale** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **oid** | **order\_name** | **previous\_balance** | **customer** |
| 11 | ord1 | 2000 | Alex |
| 12 | ord2 | 1000 | Adam |
| 13 | ord3 | 2000 | Abhi |
| 14 | ord4 | 1000 | Adam |
| 15 | ord5 | 2000 | Alex |

SQL Query to Create a View from the above table will be,

CREATE or REPLACE VIEW sale\_view

AS

SELECT \* FROM Sale WHERE customer = 'Alex';

The data fetched from SELECT statement will be stored in another object called **sale\_view**. We can use CREATE and REPLACE seperately too, but using both together works better, as if any view with the specified name exists, this query will replace it with fresh data.

**Displaying a VIEW**

The syntax for displaying the data in a view is similar to fetching data from a table using a SELECTstatement.

SELECT \* FROM sale\_view;

**Force VIEW Creation**

FORCE keyword is used while creating a view, forcefully. This keyword is used to create a View even if the table does not exist. After creating a force View if we create the base table and enter values in it, the view will be automatically updated.

Syntax for forced View is,

CREATE or REPLACE FORCE VIEW view\_name AS

SELECT column\_name(s)

FROM table\_name

WHERE condition;

**Update a VIEW**

UPDATE command for view is same as for tables.

Syntax to Update a View is,

UPDATE view-name SET VALUE

WHERE condition;

**NOTE:** If we update a view it also updates base table data automatically.

**Read-Only VIEW**

We can create a view with read-only option to restrict access to the view.

Syntax to create a view with Read-Only Access

CREATE or REPLACE FORCE VIEW view\_name AS

SELECT column\_name(s)

FROM table\_name

WHERE condition WITH read-only;

The above syntax will create view for **read-only** purpose, we cannot Update or Insert data into read-only view. It will throw an **error**.

**Types of View**

There are two types of view,

* Simple View
* Complex View

|  |  |
| --- | --- |
| **Simple View** | **Complex View** |
| Created from one table | Created from one or more table |
| Does not contain functions | Contain functions |
| Does not contain groups of data | Contains groups of data |

# SQL: create command

**create** is a DDL SQL command used to create a table or a database in relational database management system.

## Creating a Database

To create a database in RDBMS, **create** command is used. Following is the syntax,

CREATE DATABASE <DB\_NAME>;

### Example for creating Database

CREATE DATABASE Test;

The above command will create a database named **Test**, which will be an empty schema without any table.

To create tables in this newly created database, we can again use the create command.

## Creating a Table

create command can also be used to create tables. Now when we create a table, we have to specify the details of the columns of the tables too. We can specify the **names** and **datatypes** of various columns in the create command itself.

Following is the syntax,

CREATE TABLE <TABLE\_NAME>

(

column\_name1 datatype1,

column\_name2 datatype2,

column\_name3 datatype3,

column\_name4 datatype4

);

create table command will tell the database system to create a new table with the given table name and column information.

### Example for creating Table

CREATE TABLE Student(

student\_id INT,

name VARCHAR(100),

age INT);

The above command will create a new table with name **Student** in the current database with 3 columns, namely student\_id, name and age. Where the column student\_id will only store integer, name will hold upto 100 characters and age will again store only integer value.

If you are currently not logged into your database in which you want to create the table then you can also add the database name along with table name, using a dot operator .

For example, if we have a database with name **Test** and we want to create a table **Student** in it, then we can do so using the following query:

CREATE TABLE Test.Student(

student\_id INT,

name VARCHAR(100),

age INT);

### Most commonly used datatypes for Table columns

Here we have listed some of the most commonly used datatypes used for columns in tables.

|  |  |
| --- | --- |
| **Datatype** | **Use** |
| INT | used for columns which will store integer values. |
| FLOAT | used for columns which will store float values. |
| DOUBLE | used for columns which will store float values. |
| VARCHAR | used for columns which will be used to store characters and integers, basically a string. |
| CHAR | used for columns which will store char values(single character). |
| DATE | used for columns which will store date values. |
| TEXT | used for columns which will store text which is generally long in length. For example, if you create a table for storing profile information of a social networking website, then for **about me** section you can have a column of type TEXT. |

**SQL: ALTER command**

alter command is used for altering the table structure, such as,

* to add a column to existing table
* to rename any existing column
* to change datatype of any column or to modify its size.
* to drop a column from the table.

**ALTER Command: Add a new Column**

Using ALTER command we can add a column to any existing table. Following is the syntax,

ALTER TABLE table\_name ADD(

column\_name datatype);

Here is an Example for this,

ALTER TABLE student ADD(

address VARCHAR(200)

);

The above command will add a new column address to the table **student**, which will hold data of type varchar which is nothing but string, of length 200.

**ALTER Command: Add multiple new Columns**

Using ALTER command we can even add multiple new columns to any existing table. Following is the syntax,

ALTER TABLE table\_name ADD(

column\_name1 datatype1,

column-name2 datatype2,

column-name3 datatype3);

Here is an Example for this,

ALTER TABLE student ADD(

father\_name VARCHAR(60),

mother\_name VARCHAR(60),

dob DATE);

The above command will add three new columns to the **student** table

**ALTER Command: Add Column with default value**

ALTER command can add a new column to an existing table with a default value too. The default value is used when no value is inserted in the column. Following is the syntax,

ALTER TABLE table\_name ADD(

column-name1 datatype1 DEFAULT some\_value

);

Here is an Example for this,

ALTER TABLE student ADD(

dob DATE DEFAULT '01-Jan-99'

);

The above command will add a new column with a preset default value to the table **student**.

**ALTER Command: Modify an existing Column**

ALTER command can also be used to modify data type of any existing column. Following is the syntax,

ALTER TABLE table\_name modify(

column\_name datatype

);

Here is an Example for this,

ALTER TABLE student MODIFY(

address varchar(300));

Remember we added a new column address in the beginning? The above command will modify the address column of the **student** table, to now hold upto 300 characters.

**ALTER Command: Rename a Column**

Using ALTER command you can rename an existing column. Following is the syntax,

ALTER TABLE table\_name RENAME

old\_column\_name TO new\_column\_name;

Here is an example for this,

ALTER TABLE student RENAME

address TO location;

The above command will rename address column to location.

**ALTER Command: Drop a Column**

ALTER command can also be used to drop or remove columns. Following is the syntax,

ALTER TABLE table\_name DROP(

column\_name);

Here is an example for this,

ALTER TABLE student DROP(

address);

The above command will drop the address column from the table **student**.

# Truncate, Drop or Rename a Table

In this tutorial we will learn about the various DDL commands which are used to re-define the tables.

## TRUNCATE command

TRUNCATE command removes all the records from a table. But this command will not destroy the table's structure. When we use TRUNCATE command on a table its (auto-increment) primary key is also initialized. Following is its syntax,

TRUNCATE TABLE table\_name

Here is an example explaining it,

TRUNCATE TABLE student;

The above query will delete all the records from the table **student**.

In DML commands, we will study about the DELETE command which is also more or less same as the TRUNCATE command. We will also learn about the difference between the two in that tutorial.

## DROP command

DROP command completely removes a table from the database. This command will also destroy the table structure and the data stored in it. Following is its syntax,

DROP TABLE table\_name

Here is an example explaining it,

DROP TABLE student;

The above query will delete the **Student** table completely. It can also be used on Databases, to delete the complete database. For example, to drop a database,

DROP DATABASE Test;

The above query will drop the database with name **Test** from the system.

## RENAME query

RENAME command is used to set a new name for any existing table. Following is the syntax,

RENAME TABLE old\_table\_name to new\_table\_name

Here is an example explaining it.

RENAME TABLE student to students\_info;

The above query will rename the table **student** to **students\_info**.

# Using INSERT SQL command

Data Manipulation Language (DML) statements are used for managing data in database. DML commands are not auto-committed. It means changes made by DML command are not permanent to database, it can be rolled back.

Talking about the Insert command, whenever we post a Tweet on Twitter, the text is stored in some table, and as we post a new tweet, a new record gets inserted in that table.

## INSERT command

Insert command is used to insert data into a table. Following is its general syntax,

INSERT INTO table\_name VALUES(data1, data2, ...)

Lets see an example,

Consider a table **student** with the following fields.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **name** | **age** |

INSERT INTO student VALUES(101, 'Adam', 15);

The above command will insert a new record into **student** table.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **name** | **age** |
| 101 | Adam | 15 |

### Insert value into only specific columns

We can use the INSERT command to insert values for only some specific columns of a row. We can specify the column names along with the values to be inserted like this,

INSERT INTO student(id, name) values(102, 'Alex');

The above SQL query will only insert id and name values in the newly inserted record.

### Insert NULL value to a column

Both the statements below will insert NULL value into **age** column of the **student** table.

INSERT INTO student(id, name) values(102, 'Alex');

Or,

INSERT INTO Student VALUES(102,'Alex', null);

The above command will insert only two column values and the other column is set to null.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex |  |

### Insert Default value to a column

INSERT INTO Student VALUES(103,'Chris', default)

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex |  |
| 103 | chris | 14 |

Suppose the column age in our tabel has a default value of 14.

Also, if you run the below query, it will insert default value into the age column, whatever the default value may be.

INSERT INTO Student VALUES(103,'Chris')

# Using UPDATE SQL command

Let's take an example of a real-world problem. These days, Facebook provides an option for **Editing**your status update, how do you think it works? Yes, using the **Update** SQL command.

Let's learn about the syntax and usage of the UPDATE command.

## UPDATE command

UPDATE command is used to update any record of data in a table. Following is its general syntax,

UPDATE table\_name SET column\_name = new\_value WHERE some\_condition;

WHERE is used to add a condition to any SQL query, we will soon study about it in detail.

Lets take a sample table **student**,

|  |  |  |
| --- | --- | --- |
| **student\_id** | **name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex |  |
| 103 | chris | 14 |

UPDATE student SET age=18 WHERE student\_id=102;

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | chris | 14 |

In the above statement, if we do not use the WHERE clause, then our update query will update age for all the columns of the table to **18**.

### Updating Multiple Columns

We can also update values of multiple columns using a single UPDATE statement.

UPDATE student SET name='Abhi', age=17 where s\_id=103;

The above command will update two columns of the record which has s\_id 103.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

### UPDATE Command: Incrementing Integer Value

When we have to update any integer value in a table, then we can fetch and update the value in the table in a single statement.

For example, if we have to update the age column of **student** table every year for every student, then we can simply run the following UPDATE statement to perform the following operation:

UPDATE student SET age = age+1;

As you can see, we have used age = age + 1 to increment the value of age by 1.

**NOTE:** This style only works for integer values.

# Using DELETE SQL command

When you ask any question in [Studytonight's Forum](https://www.studytonight.com/studyroom/) it gets saved into a table. And using the **Delete**option, you can even delete a question asked by you. How do you think that works? Yes, using the Delete DML command.

Let's study about the syntax and the usage of the Delete command.

## DELETE command

DELETE command is used to delete data from a table.

Following is its general syntax,

DELETE FROM table\_name;

Let's take a sample table **student**:

|  |  |  |
| --- | --- | --- |
| **s\_id** | **name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

### Delete all Records from a Table

DELETE FROM student;

The above command will delete all the records from the table **student**.

### Delete a particular Record from a Table

In our **student** table if we want to delete a single record, we can use the WHERE clause to provide a condition in our DELETE statement.

DELETE FROM student WHERE s\_id=103;

The above command will delete the record where s\_id is 103 from the table **student**.

|  |  |  |
| --- | --- | --- |
| **S\_id** | **S\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |

### Isn't DELETE same as TRUNCATE

TRUNCATE command is different from DELETE command. The delete command will delete all the rows from a table whereas truncate command not only deletes all the records stored in the table, but it also re-initializes the table(like a newly created table).

**For eg:** If you have a table with 10 rows and an **auto\_increment** primary key, and if you use DELETEcommand to delete all the rows, it will delete all the rows, but will not re-initialize the primary key, hence if you will insert any row after using the DELETE command, the auto\_increment primary key will start from 11. But in case of TRUNCATE command, primary key is re-initialized, and it will again start from 1.

# Commit, Rollback and Savepoint SQL commands

Transaction Control Language(TCL) commands are used to manage transactions in the database. These are used to manage the changes made to the data in a table by DML statements. It also allows statements to be grouped together into logical transactions.

## COMMIT command

COMMIT command is used to permanently save any transaction into the database.

When we use any DML command like INSERT, UPDATE or DELETE, the changes made by these commands are not permanent, until the current session is closed, the changes made by these commands can be rolled back.

To avoid that, we use the COMMIT command to mark the changes as permanent.

Following is commit command's syntax,

COMMIT;

## ROLLBACK command

This command restores the database to last commited state. It is also used with SAVEPOINT command to jump to a savepoint in an ongoing transaction.

If we have used the UPDATE command to make some changes into the database, and realise that those changes were not required, then we can use the ROLLBACK command to rollback those changes, if they were not commited using the COMMIT command.

Following is rollback command's syntax,

ROLLBACK TO savepoint\_name;

## SAVEPOINT command

SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.

Following is savepoint command's syntax,

SAVEPOINT savepoint\_name;

In short, using this command we can **name** the different states of our data in any table and then rollback to that state using the ROLLBACK command whenever required.

### Using Savepoint and Rollback

Following is the table **class**,

|  |  |
| --- | --- |
| **id** | **name** |
| 1 | Abhi |
| 2 | Adam |
| 4 | Alex |

Lets use some SQL queries on the above table and see the results.

INSERT INTO class VALUES(5, 'Rahul');

COMMIT;

UPDATE class SET name = 'Abhijit' WHERE id = '5';

SAVEPOINT A;

INSERT INTO class VALUES(6, 'Chris');

SAVEPOINT B;

INSERT INTO class VALUES(7, 'Bravo');

SAVEPOINT C;

SELECT \* FROM class;

**NOTE:** SELECT statement is used to show the data stored in the table.

The resultant table will look like,

|  |  |
| --- | --- |
| **id** | **name** |
| 1 | Abhi |
| 2 | Adam |
| 4 | Alex |
| 5 | Abhijit |
| 6 | Chris |
| 7 | Bravo |

Now let's use the ROLLBACK command to roll back the state of data to the **savepoint B**.

ROLLBACK TO B;

SELECT \* FROM class;

Now our **class** table will look like,

|  |  |
| --- | --- |
| **id** | **name** |
| 1 | Abhi |
| 2 | Adam |
| 4 | Alex |
| 5 | Abhijit |
| 6 | Chris |

Now let's again use the ROLLBACK command to roll back the state of data to the **savepoint A**

ROLLBACK TO A;

SELECT \* FROM class;

Now the table will look like,

|  |  |
| --- | --- |
| **id** | **name** |
| 1 | Abhi |
| 2 | Adam |
| 4 | Alex |
| 5 | Abhijit |

So now you know how the commands COMMIT, ROLLBACK and SAVEPOINT works.

# SELECT SQL Query

SELECT query is used to retrieve data from a table. It is the most used SQL query. We can retrieve complete table data, or partial by specifying conditions using the WHERE clause.

### Syntax of SELECT query

SELECT query is used to retieve records from a table. We can specify the names of the columns which we want in the resultset.

SELECT

column\_name1,

column\_name2,

column\_name3,

...

column\_nameN

FROM table\_name;

### Time for an Example

Consider the following **student** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **name** | **age** | **address** |
| 101 | Adam | 15 | Chennai |
| 102 | Alex | 18 | Delhi |
| 103 | Abhi | 17 | Banglore |
| 104 | Ankit | 22 | Mumbai |

SELECT s\_id, name, age FROM student;

The above query will fetch information of s\_id, name and age columns of the **student** table and display them,

|  |  |  |
| --- | --- | --- |
| **s\_id** | **name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |
| 104 | Ankit | 22 |

As you can see the data from address column is absent, because we did not specif it in our SELECTquery.

### Select all records from a table

A special character **asterisk** \* is used to address all the data(belonging to all columns) in a query. SELECT statement uses \* character to retrieve all records from a table, for all the columns.

SELECT \* FROM student;

The above query will show all the records of **student** table, that means it will show complete dataset of the table.

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **name** | **age** | **address** |
| 101 | Adam | 15 | Chennai |
| 102 | Alex | 18 | Delhi |
| 103 | Abhi | 17 | Banglore |
| 104 | Ankit | 22 | Mumbai |

### Select a particular record based on a condition

We can use the [WHERE clause](https://www.studytonight.com/dbms/where-clause.php) to set a condition,

SELECT \* FROM student WHERE name = 'Abhi';

The above query will return the following result,

|  |  |  |  |
| --- | --- | --- | --- |
| 103 | Abhi | 17 | Rohtak |

### Performing Simple Calculations using SELECT Query

Consider the following **employee** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 101 | Adam | 26 | 5000 |
| 102 | Ricky | 42 | 8000 |
| 103 | Abhi | 25 | 10000 |
| 104 | Rohan | 22 | 5000 |

Here is our SELECT query,

SELECT eid, name, salary+3000 FROM employee;

The above command will display a new column in the result, with **3000** added into existing salaries of the employees.

|  |  |  |
| --- | --- | --- |
| **eid** | **name** | **salary+3000** |
| 101 | Adam | 8000 |
| 102 | Ricky | 11000 |
| 103 | Abhi | 13000 |
| 104 | Rohan | 8000 |

So you can also perform simple mathematical operations on the data too using the SELECT query to fetch data from table.

# Using the WHERE SQL clause

WHERE clause is used to specify/apply any condition while retrieving, updating or deleting data from a table. This clause is used mostly with SELECT, UPDATE and DELETEquery.

When we specify a condition using the WHERE clause then the query executes only for those records for which the condition specified by the WHERE clause is true.

### Syntax for WHERE clause

Here is how you can use the WHERE clause with a DELETE statement, or any other statement,

DELETE FROM table\_name WHERE [condition];

The WHERE clause is used at the end of any SQL query, to specify a condition for execution.

### Time for an Example

Consider a table **student**,

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **name** | **age** | **address** |
| 101 | Adam | 15 | Chennai |
| 102 | Alex | 18 | Delhi |
| 103 | Abhi | 17 | Banglore |
| 104 | Ankit | 22 | Mumbai |

Now we will use the SELECT statement to display data of the table, based on a condition, which we will add to our SELECT query using WHERE clause.

Let's write a simple SQL query to display the record for student with s\_id as 101.

SELECT s\_id,

name,

age,

address

FROM student WHERE s\_id = 101;

Following will be the result of the above query.

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **name** | **age** | **address** |
| 101 | Adam | 15 | Noida |

#### Applying condition on Text Fields

In the above example we have applied a condition to an integer value field, but what if we want to apply the condition on name field. In that case we must enclose the value in single quote ' '. Some databases even accept double quotes, but single quotes is accepted by all.

SELECT s\_id,

name,

age,

address

FROM student WHERE name = 'Adam';

Following will be the result of the above query.

|  |  |  |  |
| --- | --- | --- | --- |
| **s\_id** | **name** | **age** | **address** |
| 101 | Adam | 15 | Noida |

## Operators for WHERE clause condition

Following is a list of operators that can be used while specifying the WHERE clause condition.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | Equal to |
| != | Not Equal to |
| < | Less than |
| > | Greater than |
| <= | Less than or Equal to |
| >= | Greate than or Equal to |
| BETWEEN | Between a specified range of values |
| LIKE | This is used to search for a pattern in value. |
| IN | In a given set of values |

**SQL LIKE clause**

LIKE clause is used in the condition in SQL query with the WHERE clause. LIKE clause compares data with an expression using wildcard operators to match pattern given in the condition.

**Wildcard operators**

There are two wildcard operators that are used in LIKE clause.

* **Percent sign %**: represents zero, one or more than one character.
* **Underscore sign \_**: represents only a single character.

**Example of LIKE clause**

Consider the following **Student** table.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

SELECT \* FROM Student WHERE s\_name LIKE 'A%';

The above query will return all records where **s\_name** starts with character 'A'.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **age** |
| 101 | Adam | 15 |
| 102 | Alex | 18 |
| 103 | Abhi | 17 |

**Using \_ and %**

SELECT \* FROM Student WHERE s\_name LIKE '\_d%';

The above query will return all records from **Student** table where **s\_name** contain 'd' as second character.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **age** |
| 101 | Adam | 15 |

**Using % only**

SELECT \* FROM Student WHERE s\_name LIKE '%x';

The above query will return all records from **Student** table where **s\_name** contain 'x' as last character.

|  |  |  |
| --- | --- | --- |
| **s\_id** | **s\_Name** | **age** |
| 102 | Alex | 18 |

# ORDER BY Clause

Order by clause is used with SELECT statement for arranging retrieved data in sorted order. The **Order by** clause by default sorts the retrieved data in ascending order. To sort the data in descending order DESC keyword is used with Order by clause.

## Syntax of Order By

SELECT column-list|\* FROM table-name ORDER BY ASC | DESC;

### Using default Order by

Consider the following **Emp** table,

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SELECT \* FROM Emp ORDER BY salary;

The above query will return the resultant data in ascending order of the **salary**.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 403 | Rohan | 34 | 6000 |
| 402 | Shane | 29 | 8000 |
| 405 | Tiger | 35 | 8000 |
| 401 | Anu | 22 | 9000 |
| 404 | Scott | 44 | 10000 |

### Using Order by DESC

Consider the **Emp** table described above,

SELECT \* FROM Emp ORDER BY salary DESC;

The above query will return the resultant data in descending order of the **salary**.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 404 | Scott | 44 | 10000 |
| 401 | Anu | 22 | 9000 |
| 405 | Tiger | 35 | 8000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |

# Group By Clause

Group by clause is used to group the results of a SELECT query based on one or more columns. It is also used with SQL functions to group the result from one or more tables.

Syntax for using Group by in a statement.

SELECT column\_name, function(column\_name)

FROM table\_name

WHERE condition

GROUP BY column\_name

### Example of Group by in a Statement

Consider the following **Emp** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 9000 |
| 405 | Tiger | 35 | 8000 |

Here we want to find **name** and **age** of employees grouped by their **salaries** or in other words, we will be grouping employees based on their salaries, hence, as a result, we will get a data set, with unique salaries listed, along side the first employee's name and age to have that salary. Hope you are getting the point here!

group by is used to group different row of data together based on any one column.

SQL query for the above requirement will be,

SELECT name, age

FROM Emp GROUP BY salary

Result will be,

|  |  |
| --- | --- |
| **name** | **age** |
| Rohan | 34 |
| Shane | 29 |
| Anu | 22 |

### Example of Group by in a Statement with WHERE clause

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 9000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 6000 |
| 404 | Scott | 44 | 9000 |
| 405 | Tiger | 35 | 8000 |

SQL query will be,

SELECT name, salary

FROM Emp

WHERE age > 25

GROUP BY salary

Result will be.

|  |  |
| --- | --- |
| **name** | **salary** |
| Rohan | 6000 |
| Shane | 8000 |
| Scott | 9000 |

You must remember that Group By clause will always come at the end of the SQL query, just like the Order by clause.

# HAVING Clause

**Having** clause is used with SQL Queries to give more precise condition for a statement. It is used to mention condition in Group by based SQL queries, just like WHERE clause is used with SELECT query.

**Syntax** for HAVING clause is,

SELECT column\_name, function(column\_name)

FROM table\_name

WHERE column\_name condition

GROUP BY column\_name

HAVING function(column\_name) condition

### Example of SQL Statement using HAVING

Consider the following **Sale** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **oid** | **order\_name** | **previous\_balance** | **customer** |
| 11 | ord1 | 2000 | Alex |
| 12 | ord2 | 1000 | Adam |
| 13 | ord3 | 2000 | Abhi |
| 14 | ord4 | 1000 | Adam |
| 15 | ord5 | 2000 | Alex |

Suppose we want to find the **customer** whose **previous\_balance** sum is more than **3000**.

We will use the below SQL query,

SELECT \*

FROM sale GROUP BY customer

HAVING sum(previous\_balance) > 3000

Result will be,

|  |  |  |  |
| --- | --- | --- | --- |
| **oid** | **order\_name** | **previous\_balance** | **customer** |
| 11 | ord1 | 2000 | Alex |

The main objective of the above SQL query was to find out the name of the customer who has had a **previous\_balance** more than **3000**, based on all the previous sales made to the customer, hence we get the first row in the table for customer Alex.

# DISTINCT keyword

The distinct keyword is used with SELECT statement to retrieve unique values from the table. Distinct removes all the duplicate records while retrieving records from any table in the database.

### Syntax for DISTINCT Keyword

SELECT DISTINCT column-name FROM table-name;

### Example using DISTINCT Keyword

Consider the following **Emp** table. As you can see in the table below, there is employee **name**, along with employee **salary** and **age**.

In the table below, multiple employees have the same salary, so we will be using DISTINCT keyword to list down distinct salary amount, that is currently being paid to the employees.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 5000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 10000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 8000 |

SELECT DISTINCT salary FROM Emp;

The above query will return only the unique salary from **Emp** table.

|  |
| --- |
| **salary** |
| 5000 |
| 8000 |
| 10000 |

# AND & OR operator

The AND and OR operators are used with the WHERE clause to make more precise conditions for fetching data from database by combining more than one condition together.

## AND operator

AND operator is used to set multiple conditions with the WHERE clause, alongside, SELECT, UPDATE or DELETE SQL queries.

### Example of AND operator

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 5000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 12000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 9000 |

SELECT \* FROM Emp WHERE salary < 10000 **AND** age > 25

The above query will return records where **salary** is less than **10000** and **age** greater than **25**. Hope you get the concept here. We have used the AND operator to specify two conditions with WHEREclause.

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 402 | Shane | 29 | 8000 |
| 405 | Tiger | 35 | 9000 |

## OR operator

OR operator is also used to combine multiple conditions with WHERE clause. The only difference between AND and OR is their behaviour.

When we use AND to combine two or more than two conditions, records satisfying all the specified conditions will be there in the result.

But in case of OR operator, atleast one condition from the conditions specified must be satisfied by any record to be in the resultset.

### Example of OR operator

Consider the following **Emp** table

|  |  |  |  |
| --- | --- | --- | --- |
| **eid** | **name** | **age** | **salary** |
| 401 | Anu | 22 | 5000 |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 12000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 9000 |

SELECT \* FROM Emp WHERE salary > 10000 OR age > 25

The above query will return records where **either** salary is greater than 10000 **or** age is greater than 25.

|  |  |  |  |
| --- | --- | --- | --- |
| 402 | Shane | 29 | 8000 |
| 403 | Rohan | 34 | 12000 |
| 404 | Scott | 44 | 10000 |
| 405 | Tiger | 35 | 9000 |

**Division Operator in SQL**

The division operator is used when we have to evaluate queries which contain the keyword ALL.

Some instances where division operator is used are:

1. Which person has account in all the banks of a particular city?
2. Which students have taken all the courses required to graduate?

In above specified problem statements, the description after the keyword 'all' defines a set which contains some elements and the final result contains those units which satisfy these requirements.

Another way how you can identify the usage of division operator is by using the logical implication of if...then. In context of the above two examples, we can see that the queries mean that,

1. If there is a bank in that particular city, that person must have an account in that bank.
2. If there is a course in the list of courses required to be graduated, that person must have taken that course.

Do not worry if you are not clear with all this new things right away, we will try to expain as we move on with this tutorial.

We shall see the second example, mentioned above, in detail.

**Table 1: Course\_Taken** → It consists of the names of Students against the courses that they have taken.

|  |  |
| --- | --- |
| **Student\_Name** | **Course** |
| Robert | Databases |
| Robert | Programming Languages |
| David | Databases |
| David | Operating Systems |
| Hannah | Programming Languages |
| Hannah | Machine Learning |
| Tom | Operating Systems |

**Table 2: Course\_Required** → It consists of the courses that one is required to take in order to graduate.

|  |
| --- |
| **Course** |
| Databases |
| Programming Languages |

**Using Division Operator**

So now, let's try to find out the correct SQL query for getting results for the first requirement, which is:

**Query:** Find all the students who can graduate. (i.e. who have taken all the subjects required for one to graduate.)

Unfortunately, there is no direct way by which we can express the division operator. Let's walk through the steps, to write the query for the division operator.

**1. Find all the students**

Create a set of all students that have taken courses. This can be done easily using the following command.

CREATE TABLE AllStudents AS SELECT DISTINCT Student\_Name FROM Course\_Taken

This command will return the table **AllStudents**, as the resultset:

|  |
| --- |
| **Student\_name** |
| Robert |
| David |
| Hannah |
| Tom |

**2. Find all the students and the courses required to graduate**

Next, we will create a set of students and the courses they need to graduate. We can express this in the form of Cartesian Product of **AllStudents** and **Course\_Required** using the following command.

CREATE table StudentsAndRequired AS

SELECT AllStudents.Student\_Name, Course\_Required.Course

FROM AllStudents, Course\_Required

Now the new resultset - table **StudentsAndRequired** will be:

|  |  |
| --- | --- |
| **Student\_Name** | **Course** |
| Robert | Databases |
| Robert | Programming Languages |
| David | Databases |
| David | Programming Languages |
| Hannah | Databases |
| Hannah | Programming Languages |
| Tom | Databases |
| Tom | Programming Languages |

**3. Find all the students and the required courses they have not taken**

Here, we are taking our first step for finding the students who cannot graduate. The idea is to simply find the students who have not taken certain courses that are required for graduation and hence they wont be able to graduate. This is simply all those tuples/rows which are present in **StudentsAndRequired** and not present in **Course\_Taken**.

CREATE table StudentsAndNotTaken AS

SELECT \* FROM StudentsAndRequired WHERE NOT EXISTS

(Select \* FROM Course\_Taken WHERE StudentsAndRequired.Student\_Name = Course\_Taken.Student\_Name

AND StudentsAndRequired.Course = Course\_Taken.Course)

The table **StudentsAndNotTaken** comes out to be:

|  |  |
| --- | --- |
| **Student\_Name** | **Course** |
| David | Programming Languages |
| Hannah | Databases |
| Tom | Databases |
| Tom | Programming Languages |

**4. Find all students who cannot graduate**

All the students who are present in the table **StudentsAndNotTaken** are the ones who cannot graduate. Therefore, we can find the students who cannot graduate as,

CREATE table CannotGraduate AS SELECT DISTINCT Student\_Name FROM StudentsAndNotTaken

|  |
| --- |
| **Student\_name** |
| David |
| Hannah |
| Tom |

**5. Find all students who can graduate**

The students who can graduate are simply those who are present in **AllStudents** but not in **CannotGraduate**. This can be done by the following query:

CREATE Table CanGraduate AS SELECT \* FROM AllStudents

WHERE NOT EXISTS

(SELECT \* FROM CannotGraduate WHERE

CannotGraduate.Student\_name = AllStudents.Student\_name)

The results will be as follows:

|  |
| --- |
| **Student\_name** |
| Robert |

Hence we just learned, how different steps can lead us to the final answer. Now let us see how to write all these 5 steps in one single query so that we do not have to create so many tables.

SELECT DISTINCT x.Student\_Name FROM Course\_Taken AS x WHERE NOT

EXISTS(SELECT \* FROM Course\_Required AS y WHERE NOT

EXISTS(SELECT \* FROM Course\_Taken AS z

WHERE z.Student\_name = x.Student\_name

AND z.Course = y.Course ))

|  |
| --- |
| **Student\_name** |
| Robert |

This gives us the same result just like the 5 steps above.