MySQL :-

Some Terminologies :-

1) **Data** :- Every information that we use in our application in terms of fetching and processing/manipulation, is called Data.

2) **Data Persistence** :-

Data Persistence simply means Data Existence. It means that the data must survive after the process with which it was created has ended. For a data to be persistent, it must be written to a non-volatile (secondary) storage.

3) **Database** :-

A database (DB) is an organized collection of structured information, or data, typically stored electronically on a non-volatile memory. Using DB we achieve Data Persistence.

Databases are specific to an application or a grup of app. That means Database req. for a single application will be different than DB for other application.

A DB is basically a collection of files.

⇒ In DBMS, there are 3 types of models :-

① Relational, ② Network, ③ Hierarchical

⇒ In all the models, data is stored as collection of files. In Relational Model, data is stored in the form of table. Actual data are written normally as records, as in file handling, but in Relational model, a perception is made superficially so that it seems for the users that data is stored & updated in tables, which becomes easy.

Most Programs like MySQL are based on Relational Model.

⇒ So, whenever we want to add new data, update data, fetch data, or delete data, we tell the database server software, which is MySQL here, and database server does the managing & handling of data in database. It becomes the link b/w App Server (like PHP) or anyone needing data, & the database.

Thus, MySQL is a set of program (software) that manages database according to the requests & needs of other programs.

Now, to communicate with the MySQL (database server software), we need to know a language called SQL (Structured Query Language). MySQL only understands SQL language, and any conversation b/w, say, PHP & MySQL must be in SQL language.

Thus, MySQL is an RDBMS software, and, SQL is a language.

SQL language is common for any DBMS software.

- Thus, DBMS is a generalised way of handling databases. One model of DBMS is relational, and programs which follow relational model of DBMS are called RDBMS software.

- DB2, Oracle, MySQL, MS-Access, SQL Server, are example of Database softwares.

How to use MySQL :-

We need to know how ~~the~~ ~~an~~ MySQL works in management of databases.
There are two versions/parts of MySQL program:-

- MySQL client , - MySQL Server.

Both of these are important to be understood.
We know that management / storage of Database files are done on the server computer (in the 2nd any storage).
The program that actually runs and manages the DB is the MySQL Server. A single MySQL Server program is able to manage all the DB of different applications, and for same DB, it is able to differentiate users.

Now, multiple users send request for multiple operations in same or different DB stored in a single server. Single MySQL program in a server can manage all these requests, seperating/differentiating users. ~~Client~~
Now, the clients cannot directly ask for data to Database server. The program through which client can communicate to MySQL server is MySQL client, because the actual protocol for database communication is not known by user, but the

MySQL client. [Just like communication of Web Servers requires special program called Web Clients, i.e, the browsers].

In the "mysql" folder in WAMP or XAMPP, there is a "bin" folder. Inside that, we have a program ~~called th~~ name "mysql.exe" which is the MySQL Client, & a file named "mysqld.exe" which is the MySQL Server

14:45

How to open MySQL :-

- root is the default user in mysql database. Initially, it does not has any password.

- We are intending now to create, delete, and perform various operations on DB. Obviously, we are the user, so, we will use the user-side program MySQL client named mysql.exe. Thus, we need to start it.

- But before that, we need to start the MySQL server program mysqld.exe, because, for communication to be performed with client, we will require the one with whom we are communicating with. Thus, MySQL server should be up & running first, before we can pass the commands & queries to it for performing DB operations.

- Now, we have to start mysql.exe client program. For that, we need to set our present working directory to the one where the file exists.

```
cd ..... XAMPP/mysql/bin
```

Now, execute that exe file above.

It should be noted that the mysql.exe is a program written in C language. We know in the main() function of any program in C, we can pass command line arguments from cmd, at the time of execution. Thus, for this program too, we need to pass the

3 parameters using command line arguments as follows :-

mysql -u root -p ↵ (hit enter).

First is the program name that we will run.
(-u) means after that we have to write the username as whom we are logging in & authenticating ourselve.
"root" is the default username that is only present initially. If we have any other users in the database, we can use his/her username in place of "root".
"root" user has admin level privileges.

"-p" parameter is for password, It denotes that we will be entering password for the user.

After writing the ~~password~~ command, we we be asked to enter password. If we have any password, we can enter that, but if we do not have, such as for the initial case, we have to just press. (↵).

If everything is OK, then, ~~we~~ ~~will~~ the command prompt will get changed to :-

- Mysql prompt (shown as mysql>) if we use WAMP.
- MariaDB prompt (shown as MariaDB [(none)]>) if we are using XAMPP.

Coming / changing of this prompt indicates we have successfully logged into, and can pass the ~~operations~~ commands now.

Where the user's details / credentials are stored :-

"root" is the default user having no password initially. MySQL has its own database from before, (not a DB of any App, but its own default DB), which is also named as "mysql" (DB name), stored in server computer. This database is prebuilt. In this DB, out of many tables, there is a table named user, which contains usernames, & passwords,

and other details, of the users who can login & pass commands to the DB server. Using these details only, server authenticates the user, and after authentication, only they can start passing queries.

At the start, only "root" user exists. We can make more users, and set passwords for them, as we want, so that more & more people can securely be identified, & can work with a DB.

Now, we are all set to learn MySQL commands :-

- Type [quit] to exit MySQL / MariaDB prompt.

Basic commands to access MySQL :-

In SQL language, each query must end with semicolon (;)

Status

Just write the command status and hit enter.
This command is not of SQL, but is specific to mysql, thus we may or may not use semicolon after it

It shows various details of the MySQL, such as:-

a) Connection id :- To a single database server, many clients are connected. To diff. all of them, database server assigns a specific connection id to each mysql client.

b) Current Database :- The currently selected database.

c) Current user :- It will show the username with which the mysql client is currently logged in, and then @ symbol, and then the ip address of the MySQL server comp.

d) Server version :- Version of MySQL server.

e) TCP port :- port in use for connection.

and many more.

Show databases ; command :-

Command:-

[show databases ;

- This command will show the list of all the databases that is with the MySQL server (to which we are connected).
- In our case, there will be some prebuilt databases, and others (if we have made some).
- There we can find the mysql database which we learnt that it stores user details.
- test database is provided for practice & testing, and is empty initially.

use "database name" ; command :-

Syntax:-

[use database_name ; → name of any database from the list.

- This command is use to select the database.

for example:- [use test ;

- Any operation performed will now be applicable to the database that is selected. Thus for performing any operation, we must select database first.
- After selecting the database, we can run the status command again, and see the name of the selected database in the Current database : field.

Show tables command :-

Syntax:- [show tables ;

This command shows the list of all the tables that is present in the selected database currently. If no tables are present (i.e, DB is empty), it shows Empty set

⇒ Before using this command, we must select a DB using use command, else it will give an error.

⇒ At a time, we can work upon a single DB only, and that is by selecting it.

⊛ Creating & Removing Databases :-

⇒ DBMS is all about creating, managing and accessing data in efficient manner. Thus, actual important thing is the data.

⇒ We can keep data only in tables, and we can do so only if table exists.
Thus, for to keep data in table, first, we need to create the table(s). Then only data insertion can be done.

⇒ Table(s) cannot exist independently, and tables can only be kept in Database(s). To do so, the Database must exist, and only then we can create & keep tables in them. If DB does not exist, we cannot store/create tables

Thus, the seq is:-
(must be followed)

Create DB ⇒ Create Tables ⇒ keep data

⇒ Databases are application specific.

Create database command :-

Syntax:-

[create database "name" ;] it is must

- By executing this command, a new database with the name provided, will be created.
- For MySQL, the created databases will be present inside mysql/data/ folder.

Example:-

[create database db1 ;

- The commands are not case sensitive.

[CREATE DATABASE db1 ; will also work.

drop database command :-

Syntax:-

[drop database "name" ;

Example:-

[drop database db1 ;

This command will delete the database named db1; from the server computer.

Creating & Removing Tables :-

- A DB consists of multiple tables.
- Each created tables are initially empty.
- To create or remove tables from a DB, the DB must be selected first. In the selected DB only, tables will be created into / removed from.

⇒ Each table has rows and columns, Each rows contains details of individual units. With increasing data, dynamically ~~row~~ number of rows increase or decrease with addition/removal of data, but basic structure of table decided by columns remains fixed (we can alter the columns too, in a table).

⇒ During creation of table, we need to mention the columns in that table, along with the datatype of each column, i.e, the type of data that can be stored in that table. Thus, MySQL also has some predefined data-types

⇒ Also, for each table columns, we need to specify the constraints, that is the specific rule that the data in that column must maintain.

Before that, below is just the commands of how we can create tables and define columns in them, & remove them.

Create table command :-

⇒ This command is case insensitive, It is used to create table.

Syntax :-

```
create table tablename (
→ columnname1   datatype,
→ columnname2   datatype,
. . . . . .
→ );
```

⇒ After the create table keywords, we need to write the name of the table that we want to make, and then place parenthesis ().

In the parenthesis () we have mention the names of columns ①, and their constraints ②, and their datatypes ③,

which constitutes the complete description of each column. Each column description must be seperated by commas(,)

⇒ All column description may be in same line as the create table command, or each in different line, After all the columns being written, then only the parenthesis should be closed, and ending semicolon(;) should be placed, indicating the end of command.

Example:-

```
Create table employee(
    empid int(6),
    ename varchar(50),
    salary float(8,2)
);
```

→ means in this column, we can input integers of max 6 digit length

→ max no. of characters allowed in this column

→ towards left of comma, max 8 digit is allowed, and towards right, max 2.

→ command ends here

These symbols comes by default, & we do not put it.

⇒ In the above table creation, we have not added the constraints, but we can add it later.

⇒ For each datatype, in the paren() after it, we can (have to) define the max. no. of digits/chars allowed for i/p in that field.

describe command :-

Must be run after selecting a database.

Syntax:- [describe "tablename";

Example:- [describe employee;

~~Describe~~ This command shows the structure of the table whose name is passed, such as the fields (columns), data-types of the columns and their constraints, and the default values.

Renaming a table :-

[ALTER TABLE old_table_name RENAME TO new_table_name;

drop table command :-

Syntax :- [drop table "tablename" ;

Ex [drop table employee ;

→ This command deletes the table whose name is specified, if it exists in the DB, else shows error.

In Database terminology :-

1) Columns are called Fields.
2) Rows are called Records

Data types in MySQL :-

NUMERIC TYPES	STRING TYPES	DATE TYPES
INT	CHAR	DATE
SMALLINT	VARCHAR	DATETIME
TINYINT	BINARY	TIMESTAMP
MEDIUMINT	VARBINARY	TIME
BIGINT	BLOB	YEAR.
DECIMAL	TINYBLOB	
NUMERIC	MEDIUMBLOB	
FLOAT	LONGBLOB	
DOUBLE	TEXT	
BIT	TINYTEXT	
	MEDIUMTEXT	
	LONGTEXT	
	ENUM	

⇒ **INT** or **INT (size)**

(We can still limit no. of digits by writing no. in () say, INT (7))

A normal-sized integer that can be signed or unsigned.

Range is from 0 to 4294967295. $(2^{32}-1)$. ~~Upto 11 digits~~

unsigned range

Ex: -1, -45, 0, 1, 5595, -9999.

Signed range is (-2^{31}) to $(2^{31}-1)$

⇒ **VARCHAR** or **VARCHAR (size)**

A variable-length string b/w 0 & 65535 characters in length. We have to define the max length of field in character while using this datatype. For example, if we write:-

VARCHAR (25)

in datatype of a field, the field can contain string of max length of 25 characters.

Ex:- 'T', 'Debanjan', '-5557', 'Hello Students!'.

⇒ **BIT** or **BIT (size)**

It can store only 1 bit, i.e, 0 or 1.

⇒ **FLOAT** or FLOAT (~~size before decimal~~ all the digits after comma, size after decimal)

It can be used to store floating point numbers. Its size can be defined as:-

FLOAT (8,3)

This means, to the ~~left of decimal~~ comma in total (including left & right side), we can write a maximum of 8 digits, and to its right we can write a maximum of 3 digits.

To store 999.99, we write float (5,2);

(5)

999.99

⇒ **BIGINT (size)**

Can store a large integer. Unsigned range is (-2^{63}) to $(2^{63}-1)$ and unsigned range is 0 to (2^{64}).

DATE :-

For the data-type of dates & time, we need not mention any parameter in the bracket, as they have single fixed format.

For DATE, format is : YYYY-MM-DD

Column of this datatype can be defined as :- DOB DATE, (column name)

and its value assigned should be like '1985-09-21'

Note that we must use single quotes & the eiphens in between

TIME :-

For TIME, format is : HH:MI:SS

Column of TIME is defined as :- column name TIME, & in this data-type column value can be assigned in format → '05:12:56'

The time must be enclosed within single quotes (' ') in the value place, and in b/w the colons (:) must be placed.

DATETIME :-

- DATETIME general format : YYYY-MM-DD HH:MI:SS
- column definition format : column-name DATETIME,
- Value assignment format : '2021:12:25 13:26:45'

It is the combination of DATE (first) and TIME (2nd) thus, assignment format is followed, ie eiphen b/w years, month & date, & colons b/w hour, min, second whole with in (' ') and space b/w DATE & TIME.

Predefined functions - can be used in place of values entry :-

1. CURDATE() → Returns current date, of DATE datatype format.
2. CURTIME() → Returns current time, of TIME datatype format,
3. NOW() → Returns current date & time in DATETIME datatype.

Use these functions, in place of values while data record insertion.

ENUM Datatype :-

- The ENUM datatype in MySQL is a string object. It allows us to limit the values chosen from a list of permitted values in the column of its type. It uses mnemonic indexes to represent string values.

Syntax to list the values :- ENUM('value1', 'value2', ...)

Creating table with a column of ENUM datatype :-

```
CREATE Student(
    roll_no   int(10) PRIMARY KEY,
    name      varchar(50),
    DOB       date,
    gender    ENUM('MALE', 'FEMALE', 'Others') NOT NULL);
```

Indexes: 1, 2, 3

- In the column gender we can now only input these 3 strings as values, and nothing else, else it will give ERROR
- As the no. of values for ENUM datatype, we can provide infinite values, during column definition.

```
INSERT INTO Student (roll_no, name, DOB, gender)
VALUES (1, 'Debanjan', '2001-03-01', 'MALE');
```

- In place of values in the ENUM column, we can also put integers, and this will be considered as the value in the ENUM list, in the place of the provided integer index.

Example :-

```
INSERT INTO Student (roll_no, name, DOB, gender)
VALUES (2, 'Priti', '1996-12-21', 1);
```

This will insert the value at index 1 in the ENUM values, which is 'FEMALE'.

Indexing in ENUM of MySQL starts from 1.

Inserting Data into Tables :-

⇒ String & time when written, should always be written inside Single quates (' ') [Remember Hint: In C, char uses single quotes].

⇒ insert into command :-

- Using this command, we can add a single data record, into a table, at once (which means whole row.)

Syntax:-

```
insert into table_name (column 1, column 2, column 3, ..) <-|
-> values (value 1, value 2, value 3, ..);
```

After the command keywords insert into, write the name of table, in the parenthesis, we have to write the exact names of the columns that exists, and after pressing <-|, we have to use values keyword, and in its parenthesis (), we have to write the data of the rows, into the cells, in exactly the sequence that we have mentioned in the parenthesis () of insert into ()

For ex if we do the sequence change in first line we have to do it in 2nd line too.

```
insert into table_name (column 2, column 3, column 1)
-> values (value 2, value 3, value 1);
```

For example:-

If a table named employee has 3 fields as:- emp_id int(4), ename varchar(20), salary float(8.2) Then, we can add a record into the table as follow:-

```
insert into employee (emp_id, ename, salary)
-> values (6, 'Aman', 3500.00);
```

⇒ We can add a record with data fields in it, lesser than the totall fields (columns) present in the table. But for that, number of fields mentioned in the parenthesis of insert into must match number of columns in values, else, will generate an error.

Example:-

```
insert into employee (emp_id, ename)
-> values (8, 'Rahul');
```

Here, the remaining columns in which value is not added in the record, (Salary here), will have NULL as value [if not NULL constraint is not set].

⇒ If no. of columns specified in keyword command line (1st) does not match the values, then it will give error

```
insert into employee (emp_id, ename, salary)
-> values (45, 'Savita');
```

3 columns mentioned but only 2 passed

error

⇒ If we want to add all the data field values in a record, then in the insert into we may not write the column names in the parenthesis.

⇒ But, the data values then should be placed sequentially in which the columns exist in the table, i.e, the sequence in which they are written during creation. This sequence can also be known using the describe command.

Example:-

```
insert into employee
-> values (8, 'Rahul', 56000.75);
```

Updating Data in tables :-

- Updating data means changing the data in records which is already present in the table.
  It means, for any existing record, we can change the data in its any specific field(s)/column(s).

UPDATE command & SET keyword :-

Syntax:-

```
update table_name
SET column_name = new_value ;
```

- Here, in table named table_name, all the values of all records in the column_name field will get changed to new_value.

- Instead of one column, if we we want to change value in multiple columns simultaneously, we can do it as :-

```
update table_name
SET column_name1 = new_value1 ,
SET column_name2 = new_value2 ,
SET column_name3 = new_value3 ;
....
```

put semicolon only at the end of whole command

But, these changes will takes place in these specified columns in all the records in the entire table.

- To make changes happen in the columns/fields for specified records only, we have to use the while condition.

Syntax:-

```
update table_name
SET column_name = new value,
.....
where condition ;
```

Examples:-

Suppose, we have an employee table as follow:-

emp_id	ename	salary
1	Rahul	5100.50
2	Sameer	6050.00
3	Arjun	7050.00
4	Naveen	4150.00
5	Ashish	4650.00
6	Aman	3650.00
7	Sabita	2000.00

⇒ We want to increase salary of all the records (employees) by 50 rupees, The command will be:-
[since we will change all the records, by same value, we will not use where condition].

Code:-
```
update employee
-> set salary = salary+50;
```

Now, the database will become:-

emp_id	ename	salary
1	Rahul	5150.50
2	Sameer	6100.00
3	Arjun	7100.00
4	Naveen	4200.00
5	Ashish	4700.00
6	Aman	3700.00
7	Sabita	2050.00

⇒ Now, in the updated table, we want to increase the salary by 100 rupees for the employees whose salary are less than 5000. Then, we need to implement where condition for updation, The query will be :-

Code:-

```
update employee
-> set salary = salary + 100
-> where salary < 5000;
```

The rows which will satisfy this condition, in these rows only, this change will be done.

The table now is:-

emp_id	ename	salary
1	Rahul	5150.50
2	Sameer	6100.00
3	Arjun	7100.00
4	Naveen	4300.00
5	Ashish	4500.00
6	Aman	3800.00
7	Sobita	2150.00

⇒ Now, we will see a code to change data in multiple columns simultaneously. Here, in the 7th row, we will change the ename as well as salary as follows:-

Code:-

```
update employee
-> set ename = 'Savita', salary = 2500
-> where emp_id = 7;
```

⇒ And, the 7th row name & salary will get changed.

⇒ To delete a column in a table, command is :-

```
ALTER TABLE table_name
DROP column_name;
```

with ALTER command :-

- ADD
- MODIFY
- DROP
- CHANGE → rename columns.

: change old new type; can be used

Delete data from table :-

⇒ By deleting, we means clearing a single record (row), a set of records, or all the records in a table. A single column of any record(s) cannot be deleted (i.e, a cell cannot be deleted).

DELETE FROM command :-

Syntax:-

- [DELETE FROM table_name;

⇒ This command deletes all the rows/records from the table named table-name.

- [DELETE FROM table_name where condition;

⇒ This command will delete selected records from table named table-name, which satisfies the condition.

⇒ But, in both cases, in 1st case too, the table will exist.

Example:- Let us suppose there is a table named person, with fields:- person_id int(5), name varchar(10), age int(3).

With following data:-

person_id	name	age
1	Rahul	18
2	Ravi	19
3	Ajay	18
4	Vijay	17
5	Ramesh	16

⇒ We want to delete the 2nd record of Ravi, the command is :-

- [DELETE FROM person where person_id = 2;

OR

- [delete from person where name = 'Ravi';

Thus, 2nd row gets deleted & the table now is :-

person_id	name	age
1	Rahul	18
3	Ajay	18
4	Vijay	17
5	Ramesh	16

⇒ If we want to delete a set of rows, like delete records with age lesser than 18, we can do it as :-

· [delete from person where age<18;

The table becomes :-

person_id	name	age
1	Rahul	18
3	Ajay	18

⇒ If we want to clear the entire table, command is :-

[delete from person;

After this command, no record exists in the table. But, the table person still exists, with all its field names. Just the records get deleted, Structure remains preserved.

Select Data from table :-

- Selecting data is used for viewing data records in a table, and also, selecting is done for using the data in the tables. For it, SELECT command is used:-

SELECT command :-

This is one of the most used command for a DB, as after a DB is made, we need to view and use the data mainly for various reasons, and to view or use the data, we need to select the data first.

This command has many variations, but, we will start with the basic ones :-

1. All columns, All rows,
2. Selected columns, All rows
3. All columns, selected rows,
4. Selected columns, selected rows.

---

1. All columns, all rows :-

Syntax:-

```
[ SELECT * FROM table_name;
```

→ asterisk.

(3:50) The seq. of columns in the view list will follow the sequence in which they are created

> * (Asterisk) symbol means All. This command will show all records & columns, i.e, view the whole table table_name

If we want the sequence of columns to be different in viewing list, we can write the names of columns in place of * in the above code.

For example, viewing salary before ename in the prev. employee table, command will be :-

```
[ Select emp_id, salary, ename from Employee;
```

② Selected columns, All rows :-

⇒ In this case, we want view of all the rows in the table, but, ~~the~~ we do want to view only the selected field's data for them (and not data of every column).

Command:-

[Select column 1, column 2, ... FROM table_name;

↓

write the column names here (seperated by commas) which you want to view.

Ex:- [Select ename from employee

---

③ All columns, Selected rows :-

⇒ We want to view data of all the fields for each record, but, ~~as~~ in the list, we want specific records (rows) only, instead of all the rows.

⇒ Here, as we want to filter specific records in the view list, we need to use WHERE clause.

[Whenever, in any operation, the term "Specific rows" or "some records" words appear, use where clause.]

Syntax:

[SELECT * FROM tablename where condition;

⇒ In place of condition, we have to write condition/expression, and the records satisfying the condition will get viewed.

⇒ The conditions that we can write here, has several (many) variations, in terms of operators, etc. They will be discussed in detail.

Example:-

[select * from employee where salary > 4000;

We can also combine conditions using logical operations.

Example:-

[select * from employee where salary > 4000 and ename = 'Rahul';

We can use and and or as logical operators.

-> Suppose, we want to view the records, whose name starts with letter A, the command will be :-

-> [select * from employee where ename like 'A%';

like is a keyword, which compares the fields for similarity. 'A%' means the names in which after 'A' there are any character, or any number of characters.

---

④ Selected columns, selected rows :-

Syntax:-

[select column1, column2, ... from table-name where condition

-> This sequence will be followed in view list.

-> Similar to previous syntax, just replace the *, with the fields that we want to view, and only those fields' data for selected rows will be shown, in the sequence of the columns mentioned.

Example:-

[select ename, salary FROM employee where ename like 'A%';

⇒ Certain other variation of select command:-

-> In real life, tables contains vast amt of fields & records. Thus, we need to learn to filter them as per our needs so that we can view & use them efficiently :-

Display a field with an alias name set :-

[select column_name as "alias" from table_name;

must be in quotes

This will show all the data in column_name named field in table_name table, with the alias as the name of column.

Show all the unique/distinct values in a field :-

Com

[SELECT DISTINCT column_name FROM table_name;

This command will show all the unique values in the column_name field of table_name table.

Concatenate 2 (or more) columns & show in 1 column :-

Com

[select concat(column1, 'any symbol', column2) from table_name;

For example, if we want to concatenate & show the full name in one column from columns First_name & Last_name, seperate by space, the command will be :-

[select concat(First_name, ' ', Last_name) from Employee;

Sorting Records in a table :-

We can sort & view records according to ascending or descending order of values in specific field, using the ORDER BY clause, and using ASC or DESC keyword.

To sort records in ascending order of values in column1

[SELECT * FROM table_name ORDER BY column1_name ASC;

To sort all records in descending order of column1.

[SELECT * FROM table_name ORDER BY column1 DESC;

CONSTRAINTS IN MySQL :-

- Constraints are used to limit the type of data that can go into a table.
- Constraints can be specified when a table is created (with the CREATE TABLE statement), or after the table is created (with the ALTER TABLE statement).

- For a field for whom a constraint is set, during insertion, the records ~~that~~ whose that respective field data satisfies the constraint, will only be added into the table, else, ~~it~~ the record will not be accepted (it is discarded)

- Constraints are actually set for the columns / fields [as entire field follows same datatype, so, under a single constraint]

- The constraints that can be set are :-

  - NOT NULL,
  - UNIQUE,
  - PRIMARY KEY,
  - FOREIGN KEY,
  - DEFAULT,
  - CHECK.

NOT NULL Constraint :-

- The NOT NULL constraint enforces a column to NOT accept NULL values.

- The NOT NULL constraint enforces a field to always contain a value. This means that you cannot insert a new record, or update a record without adding a value to / for the field in which NOT NULL constraint is set

- NULL is a special value in SQL, which means nothing, ie, the cell / field is empty / blank.

- NULL is not equal to an empty string in SQL.

⇒ We have seen that, using the INSERT INTO command, when we insert data into some of the fields only for any record, the rest fields gets assigned NULL for that record. That happened to those fields due to NOT NULL constraint not set.

During creation (or alteration) if we set NOT NULL for that field, it can neither take NULL value explicitly, nor implicitly.

Syntax:-

```
CREATE TABLE table_name
(
 column1    data_type   NOT NULL,
 column2    data_type   NOT NULL,
 column3    datatype,
 ...
);
```

Example:-

```
create table employee
->(empid     int(4)      NOT NULL,
-> ename     varchar(15),
-> salary    float(8,2),
-> nick_name varchar(6));
```

⇒ Here, the empid field has NOT NULL constraint set. If we describe this table using describe command we can see that for this field, the Null field in description table is set to NO.

⇒ If to this field, NULL value is tried to be set ~~implicitly~~ explicitly, it will give ERROR 1048 (23000).

⇒ when explicitly, we want to assign NULL to that field like this :-

Ex

```
insert into employee (empid, ename, salary, nick_name)
-> values( NULL, 'sumit', 4000.00, 'chinku');
```

It shows ERROR 1048 (23000), and not record gets inserted.

⇒ But, when NULL value is tried to be passed implicitly, (by the system itself), like this :-

Code

```
insert into employee (ename, salary, nick_name)
-> values ('Suman', 6000.0, 'chinu');
```

mode and empid is not included

It shows 1 warning, but the records gets inserted. We can see what data is in that field in that case, using [select * from employee;

empid	ename	salary	nick_name
0	Suman	6000.0	chinu

Thus, in place of NULL, 0 is added to the field for the record implicitly, due to the constraint.

Thus, for implicit case, in the field set with NOT NULL constraint, 0 is added. (zero)

⇒ Altering a column to add/remove constraint

Syntax:-

```
ALTER TABLE table_name
-> MODIFY column_name datatype constraint;
```

optional

⇒ Using this command, the table named table_name the existing field named column_name gets changed in terms of datatype & constraint. The datatype mentioned becomes new datatype for this field, and

constraint is provided gets set for the field. If any constraint is already set for the field, and during alteration, we do not provide the constraint thing, the set constraint gets removed.

In simple terms, the definition given for the field in the command after MODIFY keyword, becomes totally the new definition of that field.

For example in the employee table created, if we want to remove the NOT NULL constraint from the empid field, our command will be :-

Com
```
ALTER TABLE employee
-> MODIFY empid int(4);
```

If a table is already created, and in any of its field, NULL already exists as data, if we alter that field to set NOT NULL constraint, then, MySQL will show warning, but the constraint will be set successfully.

In that field, all the existing NULL values will get converted to 0 as value(s) / data.

In the empid, field, we can again set NOT NULL constraint as:-

Com
```
ALTER TABLE employee
-> MODIFY empid int(4) NOT NULL;
```

⑥ UNIQUE constraint :-

- The UNIQUE constraint uniquely identifies each record in a database table. This means that, in the entire table, for the field in which UNIQUE constraint is set, no two values (values of no no record) of this field can be the same. For each record, this field have a unique value.

- The UNIQUE & PRIMARY KEY constraints both provide a guarantee for uniqueness of the values in a field (set of fields too for UNIQUE constraint).

- NULL can be assigned as value (implicitly & explicitly) in a field with UNIQUE constraint set. The value can even repeat in such a field, i.e, in a UNIQUE set field in a table, more than one record can have NULL value. This is because, NULL is not a data value, and it means "nothing."

- In a single table, we can set the UNIQUE constraint to one or more columns / fields.

Syntax :-

① Method

```
CREATE TABLE table_name
(
  Column 1   data_type  UNIQUE,
  column 2   data_type  UNIQUE,
  column 3   data_type
);
```

② Method

```
CREATE TABLE table_name
(
  column1  data_type,
  column 2  data_type,
  UNIQUE (column1)
);
```

Thus, during table creation, while defining the column names & datatype, we can just write UNIQUE after data-type & before comma (,) for each column we want to apply this constraint, or after definition of all column names & data-type in the create table command, just before the closing bracket & semicolon [);]

write the UNIQUE() keyword, and in its parenthesis, write the column name, in which you want to add the constraint.

Example:-

① method

```
create table Persons
( P_Id int UNIQUE,
LastName varchar(255),
FirstName varchar(255),
city varchar(255)
);
```

② method

```
create table Persons
( P_Id int,
LastName varchar(255),
FirstName varchar(255),
city varchar(255),
UNIQUE(P_Id)
);
```

If we already have a created table, and in it we want to add constraint to an existing column, the command :-

Syntax:-

```
ALTER TABLE table_name
→ ADD UNIQUE (column_name);
```

```
ALTER TABLE table_name
→ MODIFY column_name data_type UNIQUE;
```

Example:-

①
```
ALTER TABLE Persons
→ ADD UNIQUE(P_id);
```

②
```
ALTER TABLE Persons
→ MODIFY P_id. int UNIQUE;
```

PRIMARY KEY constraint :-

- The PRIMARY KEY constraint uniquely identifies each record in a database table.
- A PRIMARY KEY column cannot contain NULL values.
- PRIMARY KEY set column must contain unique values.
  [No two data corresponding to this field can be same in two or more records in a table].
- Each table should have a PRIMARY KEY column, and each table can have only ONE (1) PRIMARY KEY column.
- We can create a PRIMARY KEY by combining two or more column. In this case, their combination (all the combined column value) will generate the KEY that will uniquely identify each record, but in this case two also, as they are combined, PRIMARY KEY column is single for a table.

  We cannot have more than one seperate PRIMARY KEY column in a single table. 11:20
- It is not mandatory that each table should have a PRIMARY KEY column, and such a table which does not has it set is called a weak Entity.
- We can set PRIMARY KEY in a table while :-
  1) Creating the table, 2) Altering the table.

Syntax

Method 1

```
create table table_name
→ (column1 datatype PRIMARY KEY,
→ column2 data_type,
→ );
```

⇒ While defining the column names & their datatypes, we just have to write PRIMARY KEY keyword after the datatype & before the comma, of the column name, which we want to make the PRIMARY KEY field.

Example:-

```
create table book
→ (bid int(5) PRIMARY KEY,
→ title varchar(50),
→ price float(7,2));
```

space in b/w

If we describe the above book table, we can see that for the bid row, in the Key column, PRI is written.

⇒ Now, if we try to enter records into this table having duplicate ~~bo~~ bid, MySQL will show Error 1062 and record insertion will not happen.

Command

~~create table table_name~~
~~→ (bid int(5),~~
~~→ title varchar(50),~~
~~→ price float(7,2),~~
~~→ PRIMARY KEY (b~~

Method ②

```
create table table_name
→ (column1 datatype,
→ column2 datatype,
→ column3 datatype,
→ PRIMARY KEY (column_name));
```

Here, in 2nd method after defining all the column names & their data types, just before closing brack & semi colon, and the write the key term PRIMARY KEY(), and in its parenthesis, the column name which we want to make the primary key.

Example

```
create table book
→ ( bid int(5),
→ title varchar (50),
→ price float (7,2),
→ PRIMARY KEY (bid));
```

* Hence, bid becomes the PRIMARY KEY.

---

* Suppose we have an already created table with no PRIMARY KEY set. In such table, we can set it, as :-

Syntax:-

```
ALTER TABLE tablename
→ ADD PRIMARY KEY (column_name);
```

Example:-

```
alter table book
→ add PRIMARY KEY (bid);
```

* If the column was existing but was not PRIMARY KEY, after this command, it will have the constraint.

NOTE :-

If in a column of a table, values in two or more records are same, and we try to make such a column a PRIMARY KEY, using ALTER TABLE command, then mysql will give ERROR 1062, and that column cannot be made as the PRIMARY KEY.

6) FOREIGN KEY constraint :-

- A FOREIGN KEY in one table points to a PRIMARY KEY in another table.
- In FOREIGN KEY values may repeat, but the values must exist in the master table where it is the PRIMARY KEY, If the value does not exist in PRIMARY KEY, that value cannot be inserted as FOREIGN KEY in any record.
- If a PRIMARY KEY column value exists as FOREIGN KEY value in another table linked to this table, then we just cannot delete that record in the master table, as it will give an Error saying that a FOREIGN KEY value is referring to this PRIMARY KEY.

In One (x) to Many (y) relation :-

The table containing details of values of (x) will be the master table, and table containing details of values of (y) will be the table having FOREIGN KEY.

In Many to Many relation :-

- In this case, a seperate table is created, called the Junction Table, in which both the PRIMARY KEY columns of the two tables (in which many-to-many relation exist) will exist as FOREIGN KEY.

Thus, as each of 2 column is now Foreign Key, values in both may repeat, and many-to-many relation can be established.

A single table may have one or more FOREIGN KEY columns.

Syntax

Suppose, (t1) is master table, having pkc named column/field set as the primary key. To make it FOREIGN KEY in t2 table:-

```
CREATE TABLE t2
(
  c1 ~~type~~ data_type,
  c2      data_type,
  c3      data_type,
  PRIMARY KEY (c1),
  FOREIGN KEY (c2) REFERENCES t1(pkc)
);
```

writing column name in ( ) is must here

primary key column in table t1

It one

In table t2, c2 named column is the FOREIGN KEY column, that will point to (& contain values existing in) pkc column in table t1, which is the PRIMARY KEY ~~column~~ in that table.

The foreign key column name & the primary key column name in master table may be different, or same.

Example Scenario :-

- Tables : Bus, Driver.
- Condition : more than one driver can be assigned to a single bus, but one single driver is allocated only one bus. It is One (Bus) to Many (Driver) relation.

Queries :-

short of bus Id

```
Create table Bus
(Bid int(5),
Company varchar(25),
Reg No varchar(20),
PRIMARY KEY (Bid)
);
```

short of Driver Id

```
create table Driver
(Did int(6),
name varchar(30),
phone varchar(10),
busid int(5),
PRIMARY KEY (Did),
FOREIGN KEY (busid) REFERENCES Bus(Bid)
);
```

master table name

Primary Key column name in Master table

Creating foreign Key by altering table:-

Syntax:

```
ALTER TABLE t2
ADD FOREIGN KEY (c2) REFERENCES t1(pkc);
```

DEFAULT Constraint :-

- The DEFAUT constraint is used to insert a default value into a column/set a default value for a column.
- The default value set will be automatically (implicitly) added to all new records, ~~in place of~~ instead of NULL, if no value is specified for that field while inserting a record.

Syntax:-

- Setting DEFAULT constraint while creating a table :-

```
CREATE TABLE table_name
(
 column1 datatype DEFAULT value
 column2 int(4) DEFAUT 45,
 column 3 varchar (10) DEFAULT 'abc',
);
```

default value for int

default value for varchar

Examples:-

```
CREATE TABLE Persons
(P_Id int PRIMARY KEY,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar (255),
City varchar (255) DEFAULT 'BHOPAL'
);
```

- Setting DEFAULT constraint by altering an existing table.

```
ALTER TABLE Persons
ALTER City SET DEFAULT 'CHITTARANJAN';
```

the column name in which default value is to be set.

- This default value will go into the field for only that record in which no data is supplied explicitly.
- We can see the set default value in any field by using the describe "table_name" command.

CHECK constraint :-

- The CHECK constraint is used to limit the value range that can be placed in a column.
- If CHECK constraint is defined on any column, it only allows certain values to be accepted in that column/field.
- For example, in any table, if we have a column, say, named age in which age of persons are stored. If we want that for all the records (persons here), if their age is b/w 20 yrs & 30yrs, then only that record will be accepted to input, else, not, we can acheive this using the/ setting the CHECK constraint on age field suitably.

Syntax:-

Setting CHECK constraint while creating a table:-

```
CREATE TABLE table_name
( column1 data-type,
  column2 data-type,
  CHECK (condition)
);
```

Example:-

```
CREATE TABLE Persons
( P_Id int(8) NOT NULL,
  name varchar(30),
  city varchar(20),
  CHECK (P_Id > 0)
);
```

- By this command, when each record will be inserted, value in its P_Id field will be check. If it is greater than 0, the record will be inserted in the table, and if it does not satisfies this condition, that record will not get inserted in the table

Setting CHECK constraint in an already created table :-

Syntax:-

```
ALTER TABLE table_name
ADD CHECK (Condition);
```

Example :-

```
ALTER TABLE Persons
ADD CHECK (P_Id>0);
```

⇒ We can combine more than one conditions in the parameter of CHECK constraint, using logical operator and, or.

AUTO_INCREMENT attribute :-

- The AUTO_INCREMENT attribute can be used to generate a unique identity for new rows.
- When no value is specified for the AUTO_INCREMENT set column in a table, MySQL assigns sequence numbers as values to this field (of records) automatically. Generally we use this for the PRIMARY KEY column.
- We can set an initial value for the AUTO_INCREMENT column, and then, if we keep on inserting records, by ~~that~~ providing value to this field (keeping this empty), then in this field, the value for each record will be = (Value of Previous Record) + 1.

  If we do not set/provide any initial value, then by default the values start from 1, and then gets incremented by 1 for successive records.

Syntax:-

While creating the table:-

```
CREATE TABLE table_name
( column1    data_type    AUTO_INCREMENT,
  column2    data_type,
  column3    data_type,
  PRIMARY KEY (column1)
);
```

This is mandatory, and this is in general convention

- column1 has AUTO_INCREMENT set, thus, while inserting records, we may not provide values for this column.

Example:-

```
Create table Student
(rollno int(4) PRIMARY KEY auto_increment,
 name varchar(15),
 age int(3));
```

→ When we describe the table now, under the Extra column, we can see ~~Result~~ auto_increment written for the column for which we have set it.

Now, let us try adding into the table, two records, in which we will not pass the rollno, as:-

① [insert into student (name, age) values ('Rahul', 15);

② [insert into Student (name, age) values ('Ajay', 14);

The table will be:-

automatically gets assigned with increment

rollno	name	age
1	Rahul	15
2	Ajay	14

Now, if we want to add record, by entering the value for rollno explicitly, we can easily do that :-

[insert into Student (rollno, name, age) values (10, 'Ajay', 14);

The table currently is :-

rollno	name	age
1	Rahul	15
2	Ajay	14
10	Ajay	14

⇒ Now, if we add ~~value~~ record, by not providing rollno explicitly :-

[insert into student (name, age) values ('Ramesh', 13)

Table becomes:-

rollno	name	age
1	Rahul	15
2	Ajay	14
10	Ajay	14
11	Ramesh	13

So, in rollno field, if no data provided, then implicitly the previous value is incremented by 1 & added.

There can be only a single auto_increment set field in a single table, and it must be the PRIMARY KEY, in order to set it for auto_increment.

Personal Notes :-

1) ~~Addi~~ Deleting a column in an existing table :-

```
alter table table_name
drop column_name;
```

2) Adding multiple columns to a table :-

```
alter table table_name
add column1 type,
add column2 type;
```

3) Renaming a column in a table :-

```
alter table table_name
change old column new_column datatype;
```

4) Suppose, this is a table definition :-

```
create table abc(
column1 type,
column2 type,
. . .
);
```

line #1
line #2
line #...

Using alter table abc → add & drop, you actually add or drop lines in this definition.
So, apart from using add & drop for column creation/deletion, you can also use them to ~~add/rem~~ set/remove constraints, that appear/are in new line in definition, such as :-

PRIMARY KEY (. .),
Foreign Key (. . .),
etc.

Aggregate Functions :-

COUNT(), MAX(), MIN(), AVG(), SUM()

COUNT() :-

The COUNT() function returns the number of rows/records that matches a specified criteria.

- The AVG() function returns the average value of a numeric column.
- The SUM() function returns the total sum of a numeric data column.

Example:-

```
select max(age) from students;
```

- max() function shows the max value present in a numeric column. It shows it in a list, of single column.

Example:-

max(age)
30

Example

```
select min(age) from students;
```

table name

```
select count(column_name) from table_name;
```

- Shows how many records are there in the table.

```
select sum(column_name) from table_name;
```

- Shows the sum of all values in the column_name field.

We can use alias also :-

```
select avg(column_name) as "average_value" from table_name;
```

- Shows the average value of column_name, in a column named as average_value.

ORDER BY clause :-

The ORDER BY keyword/clause is used to sort the result set in ascending ~~order~~ or descending order.

For ascending order, ASC flag is used :-

```
SELECT * FROM table_name
ORDER BY column_name ASC ;
```

Here, all the columns & records will be shown according to ascending order of the values in column_name field.

Similarly, for descending order, DESC flag is used :-

```
SELECT * from table_name
ORDER BY column_name DESC ;
```

Eg2 :-

```
SELECT * from table ORDER BY column1 DESC, column2 ASC;
```

LIMIT keyword :-

The LIMIT keyword is used to restrict the number of records selected.

It is generally used with SELECT command, ~~also~~, more specifically with/after ORDER BY clause.

For example, if we have a table named student in which students of different classes are there (for some reason). In it, there is a column named age having there ages.

If we want to view ~~most~~ youngest 5 students in the table, we can do it using LIMIT as :-

```
select * from student order by age ASC LIMIT 5;
```

LIKE operator :-

⇒ The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

⇒ There are two wildcards often used in conjunction with the LIKE operator:-

① % - The % (percent) sign represents zero, one or multiple characters.

② _ - The underscore sign represents a single character.

Example:-

⇒ Suppose in the student table previously, we have a column named fname which has the first names of the student. If we want to show the names of students starting with, say, letter "B", the command will be:-

[select * from student where fname LIKE 'B%';

Some patterns:-

LIKE '_r%' → Find values that have "r" in 2nd position.

LIKE 'a__%' → Find any values that start with "a" and are atleast 3 characters in length.

How to get the count of records satisfying a particular condition:-

⇒ Say, in student table we want to find no. of students there with ~~age~~ age 18 in age column. Syntax is:-

Command

[select count(age) from student where age=25;

Another example:-

[select count(fname) from student where fname like "d%";

GROUP BY Clause :-

- The GROUP BY clause groups a set of rows into a set of summary rows by (similar/matching) values of column(s). The GROUP BY clause returns one row for each group, thus it reduces the number of rows, in the output set.

- The GROUP BY clause is an optional clause of the SELECT statement, and is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()).

Syntax:-

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

we basically write aggregate functions here, when using GROUP BY

Here, we write the column name in which values repeat, and the repeated values will be grouped, & shown (each) in one row each in the o/p.

Here, we are taking classicmodels DB & orders table for consideration & the example of commands below:-

The table has columns :- orderNumber, orderDate, requiredDate, shippedDate, status, comments, customerNumber

- We want to view the status column grouped. This can be done by :-

```
SELECT status FROM orders GROUP BY status;
```

This will give the same o/p as :-

```
SELECT DISTINCT status FROM orders;
```

only that the GROUP BY commands shows the o/p automatically in ascending order.

Using it with aggregate functions:

Now, if we want to get the count of the number of orders corresponding to each status our command will be:

Cont'd

```
SELECT status, count(*)
FROM orders
GROUP BY status;
```

means you can count any row.

O/P

STATUS	COUNT(*)
Cancelled	6
Disputed	3
In Process	6
On Hold	4
Resolved	4
Shipped	303

We could have also written

```
SELECT status, count(status)
FROM orders
GROUP BY status;
```

Basically, it will group all the similar status in one column each, and in each row, represents each group, and in the o/p, in the count(*) or count(status) column, show the no. of records belonging to each group.

SQL JOINS :-

- A relational database consists of multiple related tables linked together using ~~data~~ common columns, which are known as Foreign Keys columns. In terms of business perspective, we require ~~data~~ complete data from two or more tables.

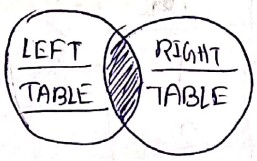
- A JOIN clause is used to combine rows & form a single row, from two or more tables, based on a related column between them.

For 2 tables to be joined together, they must have a common attribute, i.e, one must have a FOREIGN KEY referencing other.

MySQL supports following types of join :-

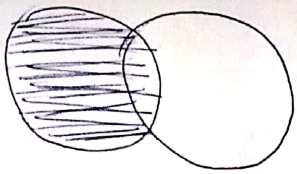
1) INNER JOIN
2) LEFT JOIN (left outer join).
3) RIGHT JOIN (right outer join)
4) CROSS JOIN (full outer join).

1) (INNER) JOIN :- Returns those records only that have matching values in both tables.



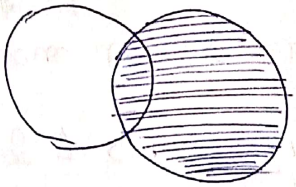
Suppose (L) is the records in Left Table, & (R) is the records in Right Table. Then, when we do inner join, the o/p shows (the selected columns of) the records common in (intersection of) both tables, i.e, ~~records~~ $(L \cap R)$

2) LEFT (OUTER) JOIN :- Returns all records from the left table, and the matched records from the right table. The columns in Right Field in the o/p remains NULL for which no maches are there



Left Outer Join

3) RIGHT (OUTER) JOIN :- Returns all records from the right table, and only the matched records from the left table.



Right Outer Join

Let us take two sample tables, to apply on them and understand different SQL joins :-

customers

cid	cname	cemail
1	vinod	vinod@thapa.com
2	bahadur	bahadur@thapa.com
3	thapa	thapa@thapa.com
5	thapatechnical	tt@gmail.com

orders

oid	orderdate	oamount	cid
1	2019-05-05	55	1
2	2019-08-06	85	2
3	2019-08-05	155	1
4	2019-05-12	95	3

foreign key

Both the tables are linked using keys. Suppose, we are asked to view whole details of the customers who have ordered something along with date & other details, we will use INNER JOIN.

```
SELECT * FROM customers
INNER JOIN orders
ON customers.cid = orders.cid;
```

o/p:-

cid	cname	cemail	oid	orderdate	oamount	cid
1	vinod	vinod@thapa.com	1	2019-05-05	55	1
2	bahadur	bahadur@thapa.com	2	2019-08-06	85	2
1	vinod	vinod@thapa.com	3	2019-08-05	155	1
3	thapa	thapa@thapa.com	4	2019-05-12	95	3

Suppose, in the o/p, we do not need all the columns of both table. For that, we can use following command to filter:-

⇒ We have to mention column name of cid to view it in the o/p. But, if we want to do that, we want to mention of which tables cid we are meaning, like orders.cid or customers.cid else, MySQL will throw ERROR.

comm

```
select customers.cid, cname, oamount from customers
JOIN orders
ON customers.cid = orders.cid;
```

If we only write JOIN instead of INNER JOIN, MySQL automatically considers it to be innerjoin.

cid	cname	oamount
1	vinod	55
2	bahadur	85
1	vinod	155
3	thapa	95

Now, suppose, we are asked to show all the customers name/details, alongwith who ordered something, and those who did not, we will use LEFT JOIN

Syntax:-

```
SELECT column-names FROM table1 LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

must be foreign key

Examples:-

```
SELECT customers.cid, cname, oamount FROM customers
LEFT JOIN orders
ON customers.cid = orders.cid;
```

cid	cname	oamount
1	vinod	55
2	bahadur	85
1	vinod	155
3	thapa	95
5	thapatechnical	NULL

Similar is the query for RIGHT JOIN. Here, it will be similar to INNER JOIN o/p as all right columns exist in LEFT.

Command

```
SELECT customers.cid, cname, orderdate, oamount
FROM customers RIGHT JOIN orders
ON customers.cid = orders.cid;
```

Dropping constraints from fields :-

If in a table, constraints are already set for fields, and also the ~~data~~ table contains records, then without affecting the data records, the constraints can be removed as :-

PRIMARY KEY :-

Since a table can have 1 primary key column at max, command to remove it :-

```
ALTER TABLE table_name DROP PRIMARY KEY;
```

NOT NULL :-

This constraint can remain set for multiple columns in a table. To remove it, just MODIFY the column with new definition, without using NOT NULL constraint :-

```
ALTER TABLE tablename MODIFY columnname datatype;
```

DEFAULT :-

Though altering table to add DEFAULT value has some diff. command (similar too) the DEFAULT constraint can be removed by setting this value to NULL.

```
ALTER TABLE tablename MODIFY column type DEFAULT NULL;
```

- AUTO_INCREMENT can also be removed similarly, by modifying the column definition using ALTER TABLE & MODIFY commands.

To remove constraints or alter something, if we want to see how exactly we defined the table columns while creating it, we can use the :-
SHOW CREATE TABLE command :-

```
SHOW CREATE TABLE table_name;
```

And it will show the exact definition of table creation.

By entering this command, after the column names definition, we can see the constraints along with their name, such as:-

PRIMARY KEY ('uid'),
UNIQUE KEY 'email' ('email')

constraint name

column in which this constraint is set

From this, we can know the constraint names, which we can use to remove the (some) constraints.

UNIQUE KEY :-

We know the constraint names now, and thus, using it we can drop constraint (applicable for UNIQUE key only)

Command:-

```
ALTER TABLE table_name DROP CONSTRAINT constraint_name
```

without quotes

With this, any constraint can be deleted.

constraint_name
name must be written without quotes

Viewing columns from multiple tables simultaneously having matching column value using SELECT command :-

⇒ Suppose we have 2 all. tables:-

Employee

empid	empname	deptid
1	emp 1	1
2	emp 2	5
3	emp 3	1
7	emp 7	5
8	emp 8	1
9	emp 9	5
11	emp 11	1
13	emp 13	5

department

deptid	deptname
1	IT
5	Admin

⇒
```
SELECT employee.empname, department.deptname
from employee, department
where employee.deptid = department.deptid;
```

o/p

empname	deptname
emp 1	IT
emp 2	Admin
emp 3	IT
emp 7	Admin
emp 8	IT
emp 9	Admin
emp 11	IT
emp 13	Admin

LIMIT keyword :-

:> LIMIT keyword is used to limit a task/query to affect only as many no. of rows, as is specified after LIMIT keyword

:> This can be useful to delete duplicate rows, in many aspect :- such as of the rows having exactly identical values of any (column) (or all columns), it can be used.

Ex:-

fname	lname	phn.	class
ravi	mishra	498	5
ravin	roy	124	3
Sumit	kr.	291	5
ravi	mishra	498	5
ravi	mishra	498	5

→ STUDENTS table

Suppose, we want to delete 2 of 3 ~~dup~~ exactly duplicate rows in above column, we can use following query:-

[delete from STUDENTS where phn=498 LIMIT 2;

Then, 2 records ~~[illegible]~~ matching the condition of WHERE clause, will be deleted, from top.

After normalising, we get a set of related tables, containing individual entities relevant data.

Now, in a web application, users generally wants data that constitutes related data from more than 1 table simultaneously.

At that time, required joint entity record is obtained by combining 2 (or more) tables. That tables are obtained by performing SQL join operations.

Ex:-

Getting a single student record & his company details from students and companies table :-

```
SELECT * FROM students INNER JOIN companies
ON
students.company_id = companies.id WHERE
students.name = "Navjot";
```

MySQL Supports :-

1) INNER JOIN, 2) LEFT JOIN, 3) RIGHT JOIN.

SELF JOIN :-

id ~~name salary~~

Suppose, we have the following table :-

Employee table

id	name	salary	managerId
1	Joe	70,000	3
2	Henry	80000	4
3	Sam	60000	Null
4	Max	90000	Null

We are required to get the names (only) of employees, who earn more than manager.

This can be done by self join as follows:-

```
SELECT a.name FROM Employee a JOIN Employee b
ON a.managerId = b.id WHERE a.salary > b.salary;
```

While doing SELF join, same table appears on both side of JOIN keyword, thus we must assign diff. alias to the tables on both sides, so that their columns can be operated on differently [e.g., here, a.managerId & b.id can be handled separately, only due to diff. alias a, b]

O/P

name
Joe

Can also be performed using multiple tables after FROM keyword :-

```
SELECT a.name FROM Employee a, Employee b ..
```

GROUP BY & HAVING in SQL :-

- Using GROUP BY, we can group repeating values of any column, into distinct groups, in which ~~contain~~ distinct values are shown once, with the details of 1st record of each group as representative.

After GROUP BY and column name, we use HAVING keyword, and then we can provide any condition.
In that condition, we can also use aggregate functions.

Applications :-

- Finding the duplicate values occurring in a table.

Ex:- Write an SQL query which fetches all the duplicate emails :-

Person

id	email
1	a@b.com
2	c@d.com
3	a@b.com

```
SELECT email FROM Person GROUP
BY email HAVING count(email) >1;
```

op

email
a@b.com