

# TABLE OF CONTENTS:

---

## TABLE OF CONTENTS:

New topics will be added from time to time.

### DATABASE CONCEPTS:

INTRODUCTION:

DATABASE

RELATION aka Table:

DOMAIN:

TUPLE aka ROW:

ATTRIBUTE AKA Column:

Degree:

Cardinality:

KEY:

**Primary Key:**

Candidate Key:

Alternate Key:

Foreign Key:

Advantages of SQL:

SOME MYSQL SQL ELEMENTS:

(i) Literals(have fixed value):

(ii) Data Types:

(I) Numeric :

(I) Number:

(II) Decimal/Float:

(ii) CHARACTER OR STRING:

(III) DATE AND TIME:

TYPES OF SQL STATEMENTS

**(i)Data Definition Language** (DDL) statement:

**(ii) Data Manipulation Language** (DML) statement:

### SQL

#### TABLE CREATION COMMANDS:

SELECT

SHOW AND DESCRIBE:

ALIASING:

TABLE ALIASES( PREREQUISITE : JOINS)

COMMENTS

WHERE

RELATIONAL OPERATORS

BETWEEN

LIKE

IN

LOGICAL OPERATOR

ORDER BY

OPERATOR PRECEDENCE:

AGGREGATE FUNCTIONS:

MY SQL FUNCTIONS

STRING FUNTIONS:

NUMERIC FUNTIONS

DATE AND TIME FUNTIONS

Null Handling

MISSED NUANCES

TABLE CREATION COMMANDS (CONTINUED)  
DROP  
CONSTRAINT  
    TABLE CONSTRAINTS  
    NAMED CONSTRAINTS  
    TABLE CREATION FROM EXISTING TABLE  
UPDATE  
DELETE VS TRUNCATE (ROW OPERATION)  
ROLLBACK  
ALTER  
    ADD  
    CHANGE  
    MODIFY  
    ADD  
    DROP  
GROUP BY (COMING SOON) (IMPORTANT)  
JOINS  
    CARTESIAN PRODUCT  
    TABLE ALIASES  
    **EQUI - JOIN**  
    **NATURAL JOIN**  
    EQUI JOIN VS NATURAL JOIN  
PIP  
MYSQL CONNECTOR  
    CONNECTING TO MySQL DATABASE  
    CREATING A CURSOR INSTANCE  
    RECORDS TABLE:  
    EXECUTING QUERIES  
    ACCESSING STORED RESULTSET(OUTPUT) FROM THE CURSOR\_OBJECT  
    rowcount()  
    connection\_name.close()  
PYMYSQL  
    PARAMETERISED QUERIES  
    STRING FORMATTING  
    cursor.commit()  
    INSERTING RECORDS USING MYSQL.CONNECTOR()  
    UPDATING RECORDS USING MYSQL.CONNECTOR()  
    DELETING RECORDS USING MYSQL.CONNECTOR()  
    SIMILARITY BETWEEN mysql.connector() and python  
TABLES USED  
DATABASE PORTION FOR TERM - II(2021-2)

**New topics will be added from time to time.**

visit this link ( <https://github.com/t-sibiraj/sql> ) to get the latest version of this pdf.

# DATABASE CONCEPTS:

# SKIP TO SQL IF YOU ALREADY KNOW DATABASE CONCEPTS #

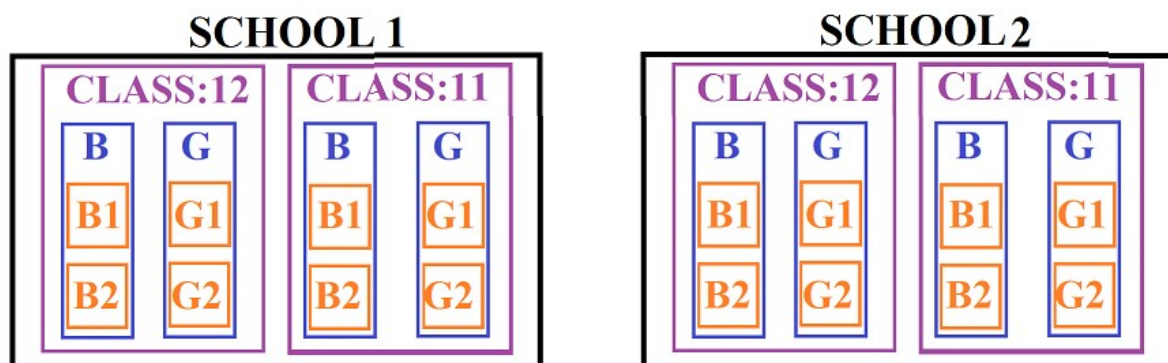
[CLICK ME TO SKIP](#)

EXTENDED READING FOR DATABASE CONCEPTS:

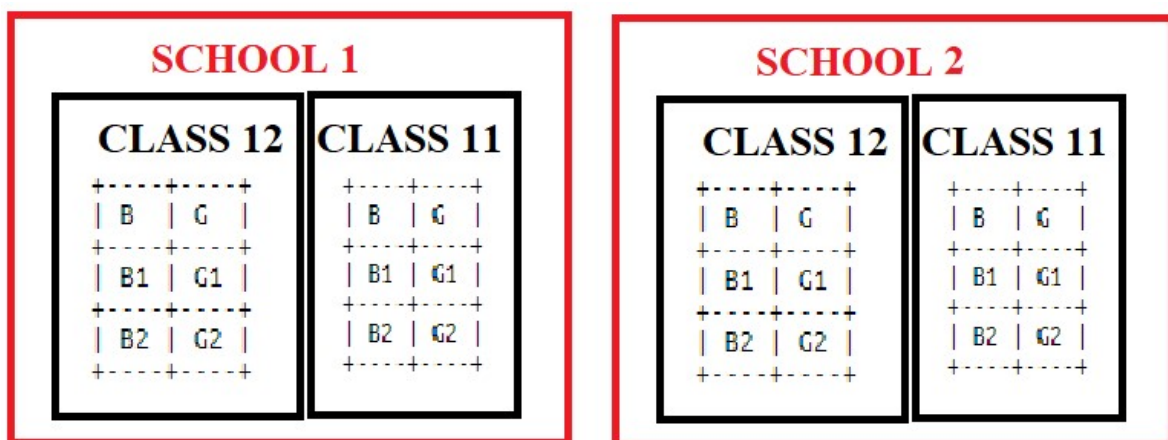
[https://cbseacademic.nic.in/web\\_material/doc/cs/2\\_Computer\\_Science\\_Python\\_ClassXII.pdf](https://cbseacademic.nic.in/web_material/doc/cs/2_Computer_Science_Python_ClassXII.pdf)

## INTRODUCTION:

Let us consider school1 has a database. In which classes class 11 and class 12 are tables. In each class(table) there are two columns(boys and girls). Each column has two rows(B has B1 , B2 as well as G has G1, G2)



SOURCE: t-sibiraj.github.io/learn



SOURCE: t-sibiraj.github.io/learn

**Database:** School1 and School2 (Collection of tables and databases)

**Tables:** Class 12 and Class 11 (Collection of rows and columns)

In Class 12(Table)

There are two columns B and G. There are two records in B(B1 , B2)

There are two rows

We can have a database named \*city\* which could have the databases \*school1\* and \*school2\* in it.

## DATABASE

**Database** is a collection of related information that is organized in such a way that it supports for easy access, modification and maintenance of data

Examples of **database**: MS-Access, MySQL, PostgreSQL, SQLite, Microsoft SQL Server, Oracle, SAP, dBASE, FoxPro, etc..

## RELATION aka Table:

Relation is nothing but a table which is made up of rows and columns

## DOMAIN:

A domain is a **set of acceptable values of a particular column**, which is based on various properties and data types.

Ad No	Name	Gender	Marital Status	SUBJECT
101	A	MALE	UNMARRIED	MATH
105	B	FEMALE	MARRIED	PHYSICS
203	C	MALE	DIVORCED	CHEMSITRY
205	D	FEMALE	WIDOW	COMPUTER SCIENCE

For example:

- (i) The domain of gender column has a set of two possible values i.e, **Male or Female**.
- (ii) The domain of marital status has a set of four possible values i.e, **Married, Unmarried, Widows and Divorced**
- \*\* (iii) The domain of subject has a **set of five possible** values i.e., **Math's,physics,chemistry,computer science and English**

## TUPLE aka ROW:

Horizontal subset/information in a table is called tuple.  
The tuple is also known as a 'record', which gives particular information of the relation (table).

For example:

- (i) In customer table, one row gives information about one customer only.
- (ii) In student table, one row gives information about one student only.

## ATTRIBUTE AKA Column:

Attribute is also known as Columns or column

## Degree:

The number of attributes(fields)(column) in a table

Degree ----> no of columns

## Cardinality:

The number of tuple(record)(rows) in a table

Cardinality ----> no of rows

## KEY:

Key is of **four types**:

- (i) Primary Key
- (ii) Candidate Key
- (iii) Foreign Key
- (iv) Alternate Key

## Primary Key:

A column or **\*\*set** of columns that uniquely identifies a **row\*\*** within a table is called **primary key**.

**PRIMARY KEY** ----> THIS IS SERVES AS AN **UNIQUE** IDENTIFIER

----> TWO PERSON CAN HAVE SAME NAME BUT THEY CAN'T HAVE SAME FINGERPRINT

----> HERE FINGERPRINT SERVES THE PURPOSE OF **PRIMARY KEY**

----> **IN TABLE** WE MUST HAVE A **PRIMARY KEY** TO UNIQUELY IDENTIFY A RECORDS **IN A TABLE**

id	name	gender
2	sam	male
1	ram	female
3	ram	male

**TABLE** NAME: GENDER

**IN THE TABLE** GENDER WE CAN **SELECT** id HAS **PRIMARY KEY** AS IT ONLY HAS **UNIQUE** RECORDS. WE CAN'T **USE** NAME **AND** GENDER **AS PRIMARY KEY** AS TWO PERSON CAN HAVE SAME NAME **AND** TWO PERSON CAN HAVE SAME GENDER

## Candidate Key:

Candidate keys are set of fields (columns with unique values) in the relation that are eligible to act as a primary key.

Candidate key = Collection of Primary key

## Alternate Key:

Out of the candidate keys, after selecting a key as primary key, the remaining keys are called alternate key.

Alternate Key = Candidate key - Primary key

## Foreign Key:

A foreign key is a field (or collection of fields) in one table that **uniquely identifies a row of another table**. In other words, a foreign key is a column or a combination of columns that is used to **establish a link between two tables**.

#FOREIGN KEY CONSTARINT

#FOREIGN KEY IS LIKE PRIMARY KEY. IT IS USED IN RDBMS.

#SO FAR WE HAVE ONLY SEE DBMS.

DBMS ----> Database Management System

RDBMS ----> Relational Database Management System

IN RDBMS TABLES ARE IN RELATION WITH EACH OTHER BUT IN DBMS TABLES ARE NOT IN RELATION WITH EACH OTHER.

DBMS

DATABSSE NAME: RDBMS

id	name	age
1	ram	10
2	sam	20
3	ram	30

TABLE NAME: AGE

id	name	gender
2	sam	male
1	ram	female
3	ram	male

TABLE NAME: GENDER

AS YOU CAN SEE BOTH THE TABLES ARE RELATED TO EACH OTHER BY THE ID COLUMN

id column in table AGE is called the primary key and id in table GENDER is called primary key

id column is called as the foreign key as it is used to relate the two tables AGE AND GENDER.

WE can even choose the gender column as the primary key but we can't choose it as foreign key as it is not present in the age table.

DATABASE DBMS:

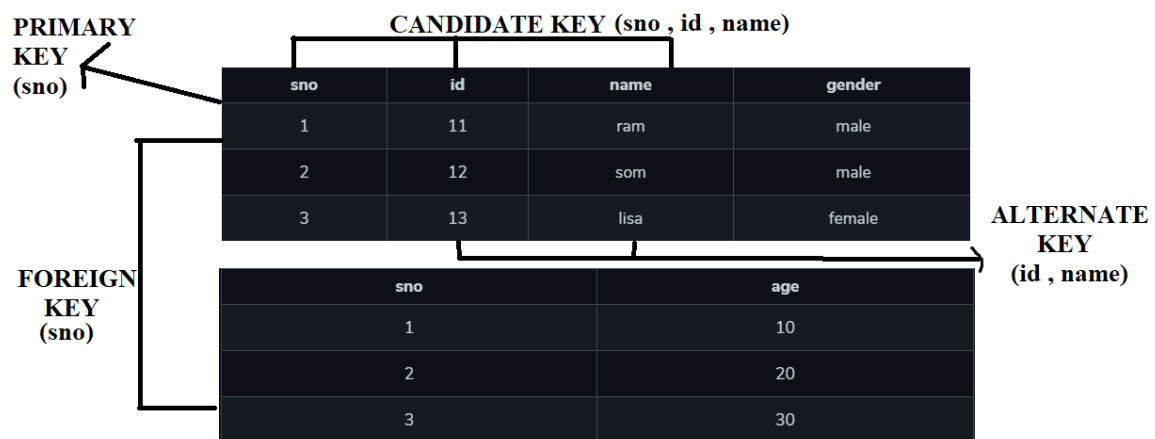
weight	name	age
40	ram	10
50	sam	male
60	ram	male

TABLE NAME: AGE

id	name	gender
2	sam	male
1	ram	female
3	ram	male

TABLE NAME: GENDER

TWO TABLES ARE NOT RELATED TO EACH OTHER SO IT IS CALLED AS DBMS



SOURCE: t-sibiraj.github.io/learn

degree ---> 4(in first table)  
cardinality ---> 3(in first table)

Resources:

<https://powerbidocs.com/2019/12/25/sql-keys/>

#### SUMMARY:

row <-> cardinality <-> tuple <-> record  
column <-> degree <-> field <-> attribute

Primary key = Can use used to uniquely identify the record  
Candidate key = Collection of Primary key  
Alternate key = Candidate key - Primary key

## Advantages of SQL:

---

- (i) SQL is portable
- (ii) High Speed
- (iii) Easy to learn
- (iv) SQL is used with any DBMS system with any vendor: SQL is used for relational databases: SQL is widely used for relational databases.
- (v) SQL acts as both programming language and interactive language:
- (vi) Client/Server language:
- (vii) Supports object based programming

## SOME MYSQL SQL ELEMENTS:

---

### (i) Literals(have fixed value):

---

numeric literal --> 53 , 64  
string literal ----> "abc"  
real literals ----> 17.0 , 17.5

### (ii) Data Types:

---

#### (I) Numeric :

#### (I) Number:



Number:

Positive numbers in the range  $1 \times 10^{-130}$  to  $9.99...9 \times 10^{125}$  with up to 38 significant digits.

Negative numbers from  $-1 \times 10^{-130}$  to  $9.99...99 \times 10^{125}$  with up to 38 significant digits.

Zero.

NUMBER(x,y) ---> x -> total number of digits  
y -> digits after decimal point

NUMBER(x) or NUMBER(x,0) ---> Acts like integer

#NOT SUPPORTED IN MYSQL 8.0 --> NUMBER(x) , NUMBER(x,y) , NUMBER

INTEGER(x) ---> x here presents the number the total number of digits

INTEGER ---> whole numbers between -2,147,483,648 and 2,147,483,647.

SMALLINT --> 5 DIGIT INTEGER

INTEGER --> 10 DIGIT INTEGER

BIGINT --> 19 DIGIT INTEGER

## (II) Decimal/Float:

DECIMAL[ (p [, s])]

- 'p' the total number of significant decimal digits
- 's' the number of digits from the decimal point to the least significant digit.

NOTE: IF YOU PASS AN INTEGER VALUE TO DECIMAL(x) OR DECIMAL(x,0), IT WILL BE STORED AS INTEGER. IF YOU PASS THE SAME INTEGER TO DECIMAL(x,y) THEN THERE WILL BE y ZEROES AFTER THE DECIMAL POINT.

12345 --> VALUE TO BE INSERTED

CREATE TABLE D1(id DECIMAL(5));	---> 12345
CREATE TABLE D1(id DECIMAL(5,0));	---> 12345
CREATE TABLE D1(id DECIMAL(5,3));	----> 12345.000

FLOAT , FLOAT(x,y) , FLOAT(x) --> SIMILAR TO DECIMAL() , DECIMAL(x,y) , DECIMAL(x)

## (III) INT/INTERGER

## (IV) FLOAT

## (ii) CHARACTER OR STRING:

`CHAR(10)` has fixed length, right padded with spaces.  
`VARCHAR(10)` has fixed length, right padded with NULL  
`VARCHAR2(10)` has variable length.

the difference between `VARCHAR` and `VARCHAR2` is that `VARCHAR` is an ANSI standard and it takes up space for variables, whereas the `VARCHAR2` is used only in Oracle but makes more efficient use of space.

#NOT SUPPORTED IN MYSQL 8.0 --> `VARCHAR2`

## (III) DATE AND TIME:

`DATE: 'YYYY-MM-DD' -> '2021-01-01'`

DON'T MISS THE QUOTES

`DATETIME: 'YYYY-MM-DD HH:MM:SS' ----> '2021-01-01 10:10:10'`

DON'T MISS THE QUOTES

`TIME: HH:MM:SS ----> '11:59:10'`

DON'T MISS THE QUOTES

`YEAR:`

-> `YEAR` -> 4-digit format  
-> `YEAR(2)` -> 2-digit format(21) #NOT SUPPORTED IN MYSQL 8.0  
-> `YEAR(4)` -> 4-digit format(2021) #NOT SUPPORTED IN MYSQL 8.0

`TIMESTAMP: (YYYYMMDDHHMMSS) --> 20210101060510`

NO NEED TO USE QUOTES. `KEY IN(INPUT) AS NUMBER(INTEGER)`.

## TYPES OF SQL STATEMENTS

### (i) Data Definition Language (DDL) statement:

DDL statements are used to **create structure of a table, modify the existing structure of the table and remove the existing table**. Some of the DDL statements are CREATE TABLE, ALTER TABLE and DROP TABLE.

**Grant and revoke privileges and roles and maintenance commands**

## (ii) Data Manipulation Language (DML) statement:

Data Manipulation Language (DML) statements are used to **access and manipulate data in existing tables**. The manipulation includes **inserting data into tables, deleting data from the tables, retrieving data and modifying the existing data**. The common DML statements are SELECT, UPDATE, DELETE and INSERT.

### (iii) Transaction Control Language (TCL) Commands:

**COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION**

### (iv) Session Control Commands

### (v) System Control Commands

---

# SQL

(Structured Query Language is a standard language used for accessing databases)

(ALL THE SQL COMMANDS WHICH ARE LISTED BELOW ARE COMPATIBLE WITH MySQL)

MySQL : <https://dev.mysql.com/doc/>

SQL Server: <https://docs.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver15>)

- **SQL IS CASE INSENSITIVE**

Consider

**Name of the table --> records**

sno	student_name	email	year	column_name
1	ram	<a href="mailto:ram@gmail.com">ram@gmail.com</a>	2004	10
2	sam	<a href="mailto:sam@yahoo.com">sam@yahoo.com</a>	2003	20
3	hari	<a href="mailto:hari@outlook.com">hari@outlook.com</a>	2002	30

## TABLE CREATION COMMANDS:

DATATYPES:

char() varchar() integer() decimal() '2001-12-30'  
char varchar integer decimal NUMBER()

#### #TO CREATE A DATABASE

SYNTAX: **CREATE DATABASE** database\_name;

EXAMPLE: **CREATE** db;

# IF WE DON'T KNOW WHETHER A DATABASE EXISTS OR NOT --> WE CAN USE ----> IF NOT EXISTS

CODE: **CREATE DATABASE IF NOT EXISTS** db;

/\* db database will be created if it not exists \*/

#### #TO CREATE A TABLE

SYNTAX: **CREATE TABLE** table\_name  
(column\_name property, ....);

EXAMPLE: **CREATE TABLE** records  
(sno **integer**, student\_name **varchar**(10) , email **varchar**(20), year  
**integer**,column\_name **integer**) ;

**\*\*NOTE: WE MUST USE A DATABASE BEFORE WE CREATE TABLES AND A TABLE MUST HAVE AT LEAST ONE VISIBLE COLUMN\*\***

#### #TO ADD A SINGLE RECORD

SYNTAX: **INSERT INTO** table\_name **VALUES** (item\_name , ....)

NOTE: WE CAN ALSO USE **VALUE** INSTEAD OF **VALUES** AS WE **ADD** A SINGLE RECORDS

EXAMPLE: **INSERT INTO** records **VALUES**  
(1, 'ram', 'ram@gmail.com', 2004, 10)

#### #TO ADD MULTIPLE RECORDS

SYNTAX: **INSERT INTO** table\_name **VALUES**  
(row\_1\_item\_no\_1 , row\_1\_item\_no\_2 , row\_1\_item\_no\_3)  
(row\_2\_item\_no\_1 , row\_2\_item\_no\_2 , row\_2\_item\_no\_2)  
....  
....  
(row\_n\_item\_no\_1 , row\_n\_item\_no\_2 , row\_n\_item\_no\_3)

CODE: **INSERT INTO** records **VALUES**  
(2, 'sam', 'sam@yahoo.com', 2003, 20),  
(3, 'hari', 'hari@outlook.com', 2002, 30),  
(4, 'ramu', 'ramu@gmail.com', 2004, 20);

ANOTHER WAY:

SYNTAX: **INSERT INTO** table\_name(columns) **VALUES**  
(row\_1\_item\_no\_1 , row\_1\_item\_no\_2 , row\_1\_item\_no\_3)  
(row\_2\_item\_no\_1 , row\_2\_item\_no\_2 , row\_2\_item\_no\_2)  
....  
....

```
(row_n_item_no_1 , row_n_item_no_2 , row_n_item_no_3)
```

```
CODE: INSERT INTO records(sno,student_name,email,year,column_name) VALUES
(2, 'sam', 'sam@yahoo.com', 2003, 20),
(3, 'hari', 'hari@outlook.com', 2002, 30),
(4, 'ramu', 'ramu@gmail.com', 2004, 20);
```

## SELECT

### #TO SELECT A SINGLE COLUMN

SYNTAX: **SELECT** column\_name  
**FROM** table\_name;

EXAMPLE: **SELECT** student\_name  
**FROM** records;

OUTPUT:

```
+-----+
| student_name |
+-----+
| ram          |
| sam          |
| hari         |
| ramu         |
+-----+
```

### #TO SELECT MULTIPLE COLUMN

SYNTAX: **SELECT** column\_name\_1,column\_name\_2,column\_name\_3  
**FROM** table\_name;

EXAMPLE: **SELECT** student\_name , year;  
**FROM** records;

OUTPUT:

```
+-----+-----+
| student_name | year |
+-----+-----+
| ram          | 2004 |
| sam          | 2003 |
| hari         | 2002 |
| ramu         | 2004 |
+-----+-----+
```

### #TO SELECT ALL THE COLUMNS FROM A TABLE:

SYNTAX: **SELECT** \*  
**FROM** table\_name;

EXAMPLE: **SELECT** \*  
**FROM** records;

OUTPUT:

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
2	sam	sam@yahoo.com	2003	20
3	hari	hari@outlook.com	2002	30
4	ramu	ramu@gmail.com	2004	20

#TO SELECT ALL THE VALUES FROM A COLUMN WITHOUT ANY DUPLICATES RECORDS

SYNTAX: `SELECT DISTINCT column_name`  
`FROM table_name;`

EXAMPLE: `SELECT DISTINCT year`  
`FROM records;`

OUTPUT:

year
2004
2003
2002

#USING ARITHMETIC OPERATORS

`+` , `-` , `*` , `/` , `%`

#SELF - EXPLANATORY

CODE: `SELECT year , year * 100`  
`FROM records;`

OUTPUT:

year	year * 100
2004	200400
2003	200300
2002	200200
2004	200400

=====

CODE: `SELECT year , year * 10 , "TEST"`  
`FROM records;`

OUTPUT:

year	year * 10	TEST
2004	20040	TEST
2003	20030	TEST

2002	20020	TEST
2004	20040	TEST

#SELF-EXPLANATORY

CODE: `SELECT 7+1;`

OUTPUT:

7+1
8

CODE: `SELECT 7 % 2;`

OUTPUT:

7 % 2
1

## SHOW AND DESCRIBE:

SYNTAX: `SHOW TABLES`

Shows the list of tables inside the current database

OUTPUT:

Tables_in_db
records

SYNTAX: `DESCRIBE table_name` or `DES table_name;`

EXAMPLE: `DESCRIBE records`

NOTE: WE CAN ALSO `USE DES` INSTEAD OF `DESCRIBE`

OUTPUT:

Field	Type	Null	Key	Default	Extra
sno	int	YES		NULL	
student_name	varchar(10)	YES		NULL	
email	varchar(20)	YES		NULL	

year	int	YES		NULL		
column_name	int	YES		NULL		
+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+

## ALIASING:

### #USING ALIAS

SYNTAX: `SELECT column_name as alias_name /*USE OF AS IS OPTIONAL */  
FROM table_name;`

EXAMPLE: `SELECT year as this_will_display_instead_of_year  
FROM records;`

**\*\*NOTE:** Alias name does **not change** the actual **column** name. Original **column** name remains the same**\*\***

OUTPUT:

+-----+
this_will_display_instead_of_year
+-----+
2004
2003
2002
2004
+-----+

**\*\*NOTE:** **USE OF AS IS OPTIONAL**. THE BELOW **CODE** WILL YIELD TTHE SAME RESULT **AS** THE ABOVE ONEE.

ALSO THERE SHOULD BE **NO SPACE IN BETWEEN IF WE USE ALIAS NAME WITHOUT QUOTES**.

`SELECT year as y FROM records (OR) SELECT year y FROM records (OR) SELECT year 'y' FROM records`

```

-----
---
=====
=====

```

### #ALIASING MULTIPLE COLOUMN NAMES

SYNTAX: `SELECT column_name AS new_name, another_column_name as another_new_name  
FROM table_name;`

CODE: `SELECT student_name as name , year as ' birth year'  
FROM records;`

## TABLE ALIASES( PREREQUISITE : JOINS)

### #TABLE ALIASES

LIKE COLUMN ALIASES WE CAN HAVE ALIAS NAME FOR TABLES TOO

SYNTAX: `SELECT table_alias_1.coloumn_name , table_alias_2  
FROM tabel_name_1 table_alias_1 , table_name_2 table_alias_2;`



CONSIDER THE TABLES BELOW

TABLE NAME: records

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
2	sam	sam@yahoo.com	2003	20
3	hari	hari@outlook.com	2002	30
4	ramu	ramu@gmail.com	2004	20

TABLE NAME: test\_table

name	year	present
ram	2004	NULL
NULL	NULL	NULL
sam	2001	absent

TABLE NAME: test\_table\_3

column_name	sno	student_name	email	year_of_birth
10	1	ram	ram@gmail.com	2004
20	2	sam	sam@yahoo.com	2003
30	3	hari	hari@outlook.com	2002
20	4	ramu	ramu@gmail.com	2004

CODE: SELECT a1.student\_name , a2.year\_of\_birth  
FROM records a1 , test\_table\_3 a2;

OUTPUT:

student_name	year_of_birth
ramu	2004
hari	2004
sam	2004
ram	2004
ramu	2003
hari	2003
sam	2003
ram	2003
ramu	2002
hari	2002

sam		2002	
ram		2002	
ramu		2004	
hari		2004	
sam		2004	
ram		2004	
+-----+			

#WE CAN AVOID THE ABOVE SITUATION USING WHERE CLAUSE

```
CODE: SELECT a1.student_name , a2.year_of_birth
      FROM records a1 , test_table_3 a2
      WHERE a1.sno = a2.sno;
```

OUTPUT:

student_name		year_of_birth	
+-----+			
ram		2004	
sam		2003	
hari		2002	
ramu		2004	
+-----+			

WE CAN ALSO OTHER CONDITION WITH WHERE CLAUSE

## COMMENTS

#LIKE PYTHON WE CAN USE COMMENTS

```
/* THIS IS A COMMENT */
```

## WHERE

#GENERAL SYNTAX

```
SELECT column_name,
FROM table_name,
WHERE condition;
```

NOTE: THE FOLLOWING OPERATORS CAN BE USED IN PLACE OF CONDITION

## RELATIONAL OPERATORS

#USING RELATIONAL OPERATORS

# = (EQUALITY OPERATOR LIKE == IN PYTHON)

```
/* Q: DISPLAY THE RECORDS OF THE STUDENTS WHOSE NAME IS RAM */
```

```
CODE: SELECT *
      FROM records
```

```
WHERE student_name = 'ram';
```

OUTPUT:

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10

# >(greater than) , <(lesser than) , >=(greater than or equal to) , <=(less than or equal to)

```
/* Q: DISPLAY THE RECORDS OF THE STUDENT WHOSE BIRTH YEAR IS LESS THAN OR EQUAL TO 2003 */
```

```
CODE: SELECT *  
FROM records  
WHERE year <= 2003;
```

OUTPUT:

sno	student_name	email	year	column_name
2	sam	sam@yahoo.com	2003	20
3	hari	hari@outlook.com	2002	30

# <> or (!= --> NOT VALID IN MYSQL 8.0)

```
/* Q: DISPLAY THE RECORDS OF THE STUDENT WHOSE BIRTH YEAR IS NOT EQUAL TO 2003 */
```

```
CODE: SELECT *  
FROM records  
WHERE year <> 2003;
```

OUTPUT:

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
3	hari	hari@outlook.com	2002	30
4	ramu	ramu@gmail.com	2004	20

## BETWEEN

#BETWEEN

SYNTAX: `SELECT *`  
`FROM table_name`  
`WHERE column_name BETWEEN lower_limit AND upper_limit; #inclusive of`  
`upper_limit and lower_limit`

`/* Q: DISPLAY THE RECORDS OF THE STUDENT WHOSE BIRTH YEAR IS IN THE RANGE OF`  
`2002 TO 2004 */`

SYNTAX: `SELECT *`  
`FROM records`  
`WHERE year BETWEEN 2002 AND 2004; #includes both 18 and 22`

OUTPUT:

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
2	sam	sam@yahoo.com	2003	20
3	hari	hari@outlook.com	2002	30
4	ramu	ramu@gmail.com	2004	20

## LIKE

#LIKE(only text[string] values) ----> USING WILDCARDS(REGEX) -->  
PATTERNS(REGEX)  
# 'a%' ----> starting with a  
# '%a' ----> ending with a  
# '%a%' ----> contains a  
# % --> wildcard  
# a% --> pattern

`/* SELECT THE NAME OF THE STUDENT [STARTING] WITH THE LETTER R */`

CODE: `SELECT *`  
`FROM records`  
`WHERE student_name LIKE 'r%';`

OUTPUT:

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
4	ramu	ramu@gmail.com	2004	20

\*\*\* Note r% --> will select both r and R \*\*

'%r%' ----> WILL SELECT ALL THE RECORDS CONTAINING THE LETTER r or R -> (CAN BE START OR IN END AND CAN BE IN BETWEEN)

'%r' ----> WILL SELECT ALL THE RECORDS WHICH ENDS WITH LETTER r or R

Empty set (0.07 sec) ----> DISPLAYED WHEN IT FINDS NO MATCHING RECORDS(  
SELECT \* FROM records WHERE  
student\_name LIKE 'r%');

=====

/\* Q: DISPLAY THE RECORDS THE STUDENTS IF THE LENGTH OF THE NAME OF STUDENT IS EXACTLY THREE CHARACTERS \*/

CODE: SELECT \*  
FROM records  
WHERE student\_name LIKE '\_\_\_';

"\_\_\_" ----> There are three underscore(\_) inside the quotes(" ")  
----> Three underscores are used to match any string with exactly three characters  
----> Underscore here represents characters( four underscore matches any string with exactly 4 characters )

OUTPUT:

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
2	sam	sam@yahoo.com	2003	20

=====

#DIFFERENCE BETWEEN '\_\_\_' AND '\_\_\_%'

'\_\_\_' ----> MATCHES ANY STRING WHICH HAS EXACTLY THREE CHARACTERS

'\_\_\_%' ----> MATCHES ANY STRING WHICH HAS AT LEAST THREE CHARACTERS

CODE: SELECT \*  
FROM records  
WHERE student\_name LIKE '\_\_\_%';

OUTPUT:

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
2	sam	sam@yahoo.com	2003	20
3	hari	hari@outlook.com	2002	30
4	ramu	ramu@gmail.com	2004	20

# IN

#IN (MEMBERSHIP)

/\*Q: DISPLAY THE RECORDS OF THE STUDENT IF A STUDENT IS BORN IN THE YEAR 2002 AND 2004 \*/

```
CODE: SELECT *
      FROM records
      WHERE year IN (2002,2004);
```

OUTPUT:

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
3	hari	hari@outlook.com	2002	30
4	ramu	ramu@gmail.com	2004	20

## LOGICAL OPERATOR

#LOGICAL OPERATOR (AND , OR)

#AND (BOTH THE STATEMENTNS MUST BE TRUE)

/\* Treat AND as \* (LOGICAL MULTIPLICATION)

True as 1

False as 0

True AND True ---> 1 \* 1 ---> 1 ---> True

True AND False ---> 1 \* 0 ---> 0 ---> False

False AND True ---> 0 \* 1 ---> 0 ---> False

False AND False ---> 0 \* 0 ---> 0 ---> False

Anything multiplied by zero is zero so if there

Treat OR as + (LOGICAL ADDITION)

True OR True ---> 1 + 1 ---> 1 ---> True /\* here 1+1 --> 1 (still True) \*/

True OR False ---> 1 + 0 ---> 1 ---> True

False OR True ---> 0 + 1 ---> 1 ---> True

False OR False ---> 0 + 0 ---> 0 ---> False

\*/

#AND (BOTH THE STATEMENTNS MUST BE TRUE)

/\* Q: DISPLAY THE REOCORDS OF THE STUDENTS IF THEIR NAME STARTS WITH R AND THEIR BIRTH YEAR IS GREATER THAN OR EQUAL TO2003 \*/

```
CODE: SELECT *
      FROM records
```

```
WHERE student_name LIKE 'r%'
AND year >= 2003;
```

OUTPUT:

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
4	ramu	ramu@gmail.com	2004	20

#OR (AT LEAST ONE OF THE STATEMENT SHOULD BE TRUE)

/\* Q: DISPLAY THE REOCORDS OF THE STUDENTS IF THEIR NAME STARTS WITH R OR IF THEIR BIRTH YEAR IS GREATER THAN OR EQUAL TO 2003 \*/

```
CODE: SELECT *
FROM records
WHERE student_name LIKE 'r%'
OR year >= 2003;
```

OUTPUT:

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
2	sam	sam@yahoo.com	2003	20
4	ramu	ramu@gmail.com	2004	20

#SIMILARITY BETWEEN OR AND IN

```
CODE 1: SELECT *
FROM records
WHERE year = 2002 OR year = 2003 OR year 2004;
```

/\* THE ABOVE QUERY CAN ASLO BE WRITTEN USING THE \*IN\* OPERATOR \*/

```
CODE 2: SELECT *
FROM records
WHERE year IN (2002 , 2003 , 2004);
```

#NOT

```
CODE: SELECT *
FROM records
WHERE student_name NOT LIKE 'a'; /* SELECTS ALL THE RECORDS WHOSE NAME
DOESN'T START WITH A */
```

OUTPUT:

sno	student_name	email	year	column_name
-----	--------------	-------	------	-------------

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
2	sam	sam@yahoo.com	2003	20
3	hari	hari@outlook.com	2002	30
4	ramu	ramu@gmail.com	2004	20

```
CODE: SELECT *
      FROM records
      WHERE year NOT IN (2004,2002) ; /* SELECTS ALL THE RECORDS EXCEPT 2004
AND 2002 */
```

OUTPUT:

sno	student_name	email	year	column_name
2	sam	sam@yahoo.com	2003	20

```
CODE: SELECT *
      FROM records
      WHERE year NOT BETWEEN 2002 AND 2004; /* SELECTS ALL THE RECORD EXCEPT
2002 , 2003 AND 2004 */
```

OUTPUT:

Empty set

# NOT LIKE . IS NOT and so on

## ORDER BY

# ORDER BY

#TO ORDER THE VALUES OF A COLUMN BASED ON ANOTHER COLUMN

#NOTE : ORDER BY orders the value in ascending(ASC) order by default.

```
CODE: SELECT *
      FROM records
      ORDER BY year; #ASC ----> ORDER BY year ASC;
```

OUTPUT:

sno	student_name	email	year	column_name
3	hari	hari@outlook.com	2002	30
2	sam	sam@yahoo.com	2003	20
1	ram	ram@gmail.com	2004	10
4	ramu	ramu@gmail.com	2004	20



=====

=====

#To order an COLUMN which has only words(strings to be specific)

CODE: SELECT \*  
FROM records  
ORDER BY student\_name; #orders in alphabctial order in ASC

OUTPUT:

sno	student_name	email	year	column_name
3	hari	hari@outlook.com	2002	30
1	ram	ram@gmail.com	2004	10
4	ramu	ramu@gmail.com	2004	20
2	sam	sam@yahoo.com	2003	20

=====

=====

#To order in descending order

CODE: SELECT \*  
FROM records  
ORDER BY year DESC; #orders DESC

OUTPUT:

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
4	ramu	ramu@gmail.com	2004	20
2	sam	sam@yahoo.com	2003	20
3	hari	hari@outlook.com	2002	30

## OPERATOR PRECEDENCE:

INTERVAL  
BINARY, COLLATE  
!  
- (unary minus), ~ (unary bit inversion)  
^  
\*, /, DIV, %, MOD  
-, +  
<<, >>  
&  
|  
= (comparison), <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN, MEMBER OF  
BETWEEN, CASE, WHEN, THEN, ELSE  
NOT  
AND, &&  
XOR

```
OR, ||  
= (assignment), :=
```

SOURCE: <https://dev.mysql.com/doc/refman/8.0/en/operator-precedence.html>

## AGGREGATE FUNCTIONS:

#AGGREGATE FUNCTIONS aka GROUP FUNCTIONS aka MULTIPLE ROW FUNCTIONS

SYNTAX: `SELECT aggregate_funtion_name(column_name)`  
`FROM table_name;`

So far we have only seen only **one** aggregate **function**(**DISTINCT**) **if** I am correct.  
**From** now **on** we will learn about the other aggregate functions.

MORE INFO **ON** AGGREGATE FUNTION CAN BE FOUND **AT** :  
<https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html>

YOU CAN ALSO **USE** ALIAS **WITH** AGGREGATE FUNTIONS

#MIN ----> RETURNS THE SMALLEST VALUE IN A COLUMN

CODE: `SELECT MIN(year)`  
`FROM records;`

OUTPUT:

```
+-----+  
| MIN(year) |  
+-----+  
|      2002 |  
+-----+
```

#AGGREGATE FUNCTIONS--> THOSE WHICH RETURN SINGLE VALUE(MIN,MAX,AVG,SUM,etc..)

#MAX ----> RETURNS THE LARGEST VALUE IN A COLUMN

CODE: `SELECT MAX(year)`  
`FROM records;`

OUTPUT:

```
+-----+  
| MAX(year) |  
+-----+  
|      2004 |  
+-----+
```

#AVG ----> TO FIND THE AVERAGE VALUE

average = 
$$\frac{\text{value}_1 + \text{value}_2 + \dots + \text{value}_n}{\text{no\_of\_values}(n)}$$

```
CODE: SELECT AVG(year)
      FROM records;
```

OUTPUT:

```
+-----+
| AVG(year) |
+-----+
| 2003.2500 |
+-----+
```

$$\text{average} = \frac{2004 + 2003 + 2002 + 2004}{4} = \frac{8013}{4} = 2003.2500$$

#COUNT --> TO COUNT THE NO ITEMS IN A COLUMN

#IT COUNTS ONLY NON-EMPTY( NULL TO BE SPECIFIC) VALUES ---> IT DOESN'T TAKE ACCOUNT OF ANY NULL VALUES

```
CODE: SELECT COUNT(year)
      FROM records;
```

OUTPUT:

```
+-----+
| COUNT(year) |
+-----+
|          4  |
+-----+
```

```
=====
=====
```

#COUNT(\*)

#COUNT(\*) --> TO COUNT THE NUMBER OF ROWS IN A TABLE

```
CODE: SELECT COUNT(*)
      FROM records;
```

OUTPUT:

```
+-----+
| COUNT(*) |
+-----+
|          4  |
+-----+
```

```
=====
=====
```

COUNT(column\_name) VS COUNT(\*)

COUNT(column\_name) ---> ONLY TAKES ACCOUNT OF NON NULL VALUES

COUNT(\*) ----> TAKES ACCOUNT OF NULL AND NON NULL VALUES i.e ALL VALUES

CONSIDER THE TABLE BELOW:

NAME OF THE TABLE: test\_table

name	year	present
ram	2004	NULL
NULL	NULL	NULL
sam	2001	absent

CODE: SELECT COUNT(present) FROM test\_table;

OUTPUT:

COUNT(present)
1

CODE: SELECT COUNT(\*) FROM test\_table;

OUTPUT:

COUNT(*)
3

#SUM ----> RETURNS THE SUM OF ALL THE VALUES IN A COLUMN

CODE: SELECT SUM(year)  
FROM records;

OUTPUT:

SUM(year)
8013

# MORE ON DISTINCT AND ALL

DISTINCT CAN BE USED IN COMBINATION WITH OTHER AGGREGATE FUNCTIONS

MAX( DISTINCT column\_name) , MIN( DISTINCT column\_name) , AVG( DISTINCT  
column\_name) ,  
SUM( DISTINCT column\_name)

WHEN DISTINCT IS USED WITH SUM ONLY DUPLICATE ENTRIES ARE NOT TAKEN INTO ACCOUNT

CODE: SELECT SUM(DISTINCT year)  
FROM records;

OUTPUT:

SUM(DISTINCT year)

```
|          6009 |
+-----+
```

#ALL VS DISTINCT VS COUNT---> COSIDER ALL THE VALUES

CONSIDER THE TABLE BELOW

NAME OF THE TABLE: test\_table\_2

```
+-----+-----+-----+
| name | year | present |
+-----+-----+-----+
| ram  | 2001 | NULL    |
| sam  | 2002 | present |
| ramu | 2003 | present |
| som  | 2004 | absent  |
+-----+-----+-----+
```

COUNT(column\_name) ---> counts only non NULL values  
COUNT(DISTINCT , column\_name) ----> counts only distinct non NULL values  
COUNT(ALL , column\_name) ---> counts only non NULL values

CODE: SELECT COUNT(present) FROM test\_table\_2;

OUTPUT:

```
+-----+
| COUNT(present) |
+-----+
|          3 |    ----> present , present , absent
+-----+
```

CODE: SELECT COUNT(DISTINCT present) FROM test\_table\_2;

OUTPUT:

```
+-----+
| COUNT(DISTINCT present) |
+-----+
|          2 | ----> present , absent
+-----+
```

( present ,absent(duplicate present is not taken into account while counting)

CODE: SELECT COUNT(ALL present) FROM test\_table\_2;

OUTPUT:

```
+-----+
| COUNT(ALL present) |
+-----+
|          3 | --> present , present , absent
+-----+
```

## MY SQL FUNCTIONS

### STRING FUNTIONS:

```
#CHAR()
```

```
#NOTE: NEWER VERSIONS OF MYSQL INTERPRETS THE BINARY RESULT AS HEXADECIMAL
```

```
/* TYPE status IN MYSQL COMMAND LINE CLIENT */
```

```
Connection id:      11
Current database:    db
Current user:        root@localhost
SSL:                 Cipher in use is TLS_AES_256_GCM_SHA384
Using delimiter:     ;
Server version:      8.0.27 MySQL Community Server - GPL
Protocol version:    10
Connection:          localhost via TCP/IP
Server characterset: utf8mb4
Db characterset:     utf8mb4
Client characterset: utf8mb4
Conn. characterset:  utf8mb4
TCP port:            3306
Binary data as:      Hexadecimal #THIS IS THE REASON WHY WE GET HEXADECIAML
VALUE
Uptime:              23 hours 35 min 0 sec
```

```
#TO SOLVE THIS PROBLEM FOLLOW THE STEPS BELOW THERE ARE TWO WAYS:
```

```
WAY 1:
```

```
ONE QUICK FIX TO SOLVE THIS PROBLEM IS TO USE USING ASCII
i.e.. SELECT CHAR(65 USING ASCII);
SYNTAX: CHAR(N,... [USING charset_name])
```

```
MORE INFO CAN BE FOUND AT: https://dev.mysql.com/doc/refman/8.0/en/string-
functions.html#function\_char
```

```
WAY 2:
```

```
STEP 1: GOTO TO THE FOLDER WHERE THE BIN FOLDER OF MYSQL SEREVER WHICH WILL BE
LOCATED INSIDE MYSQL
```

```
IF YOU ARE USING WINDOWS 10 --> C:\windows\System32\cmd.exe --> THIS WOULD BE
PATH OF THE BIN FOLDER(IN MOST CASES IF NOT LOCATE THE FOLDER BY YOURSELF)
```

```
STEP 2: OPEN COMMAND PROMPT AND KEY IN THE BELOW COMMAND
mysql -u root -p --skip-binary-as-hex
```

```
STEP 3: ENTER THE PASSWORD
```

```
STEP 4: RUN THE FOLLOWING
SELECT CHAR(65)
```

```
/* NOTE YOU NEED TO DO THIS STEP EVERY TIME BEFORE YOU USE MYSQL AND MYSQL CLI
CAN BE USED DIRECTLY FROM CMD(COMMAND PROMPT) */
```

```
MORE INFO CAN BE FOUND AT: https://bugs.mysql.com/bug.php?id=99480
```

```
CODE: SELECT CHAR(65 USING ASCII) AS "Alphabet";
```

```
OUTPUT:
```

Alphabet
A

```
=====
=====
```

### #CONCAT()

-> IF COLOUMN NAMES ARE USED THE RECORDS GETS CONCATENATED

CODE: `SELECT CONCAT(student_name , year) as "NAME AND YEAR"`  
`FROM records;`

OUTPUT:

NAME AND YEAR
ram2004
sam2003
hari2002
ramu2004

-> CAN ALSO BE USED TO CONCAT TWO STRINGS

```
=====
=====
```

CODE: `SELECT CONCAT("SIBI" , "RAJ") AS "NAME";`

OUTPUT:

NAME
SIBIRAJ

```
=====
=====
```

CODE: `SELECT "SI" "BI" "RAJ";`

OUTPUT:

SI
SIBIRAJ

```
=====
=====
```

#LOWER()/LCASE() ---> [https://dev.mysql.com/doc/refman/8.0/en/string-functions.html#function\\_lcase](https://dev.mysql.com/doc/refman/8.0/en/string-functions.html#function_lcase)

CODE: `SELECT LOWER("PYTHON");`

OUTPUT:

```

+-----+
| LOWER("PYTHON") |
+-----+
| python          |
+-----+

```

```

=====
=====

```

#UPPER()/LCASE()

CODE: SELECT UPPER("python");

OUTPUT:

```

+-----+
| UPPER("python") |
+-----+
| PYTHON          |
+-----+

```

```

=====
=====

```

# TRIM ---> REMOVES TRAILING AND LEADING SPACES --> LIKE STRIP()

# LTRIM ---> LIKE LSTRIP()

# RTIRM ---> LIKE RSTIP()

CODE: SELECT TRIM(" AKA STRIP ");

OUTPUT:

```

+-----+
| TRIM(" AKA STRIP ") |
+-----+
| AKA STRIP           |
+-----+

```

```

=====
=====

```

#SUBSTRING()/SUBSTR()

SUBSTR(given\_string,3,4) ---> SELECT 4 CHARACTERS STARING FROM THE INDEX 3

LEFT SIDE INDEX STARTS FROM 0 AND RIGHT SIDE INDEX STARTS FROM -1

CODE: SELECT SUBSTR("0123456789" , 3 , 4) AS SLICING;

OUTPUT:

```

+-----+
| SLICING |
+-----+
| 2345    |
+-----+

```

## NUMERIC FUNTIONS

#MOD() ---> RETURNS THE REMAINDER ---> LIKE % IN PYTHON ---> \*\* REMAINDER(INT PART) \*\*



CODE: `SELECT MOD(7,2);`

OUTPUT:

```
+-----+
| MOD(7,2) |
+-----+
|         1 |
+-----+
```

=====

#POWER()

CODE: `SELECT POWER(2,3);` --->  $2^3$  OR  $2^{**}3$

OUTPUT:

```
+-----+
| POWER(2,3) |
+-----+
|          8 |
+-----+
```

=====

#ROUND() ---> TO ROUND THE EXPRESSION TO THE NUMBER OF DECIMAL POINT

MORE INFO ON ROUNDING CAN BE FOUND AT:

<https://tutorax.com/blogue/en/how-to-round-decimals-rounding-numbers-guide/#:~:text=There%20are%20certain%20rules%20to,9%20round%20the%20number%20up.>

CODE: `SELECT ROUND(1.26,1);`

OUTPUT:

```
+-----+
| ROUND(1.26,1) |
+-----+
|          1.3 |
+-----+
```

CODE: `SELECT ROUND(1.25,1);`

OUTPUT:

```
+-----+
| ROUND(1.25,1) |
+-----+
|          1.3 |
+-----+
```

CODE: `SELECT ROUND(1.26,1);`

OUTPUT:

```
+-----+
| ROUND(1.26,1) |
+-----+
|          1.3 |
+-----+
```

=====

#SIGN() ---> RETURNS THE SIGN OF THE NUMBER

CODE: `SELECT SIGN(-10);`

OUTPUT:

```
+-----+
| SIGN(-10) |
+-----+
|          -1 |
+-----+
```

CODE: `SELECT SIGN(10);`

OUTPUT:

```
+-----+
| SIGN(10) |
+-----+
|          1 |
+-----+
```

=====

#SQRT()

CODE: `SELECT SQRT(4);`

OUTPUT:

```
+-----+
| SQRT(4) |
+-----+
|          2 |
+-----+
```

=====

#TRUNCATE() ---> REMOVES(TURNCATES) THE CHARACTERS UPTO TO THE GIVEN DECIMAL PLACES

CODE: `SELECT TRUNCATE(123456,3) AS "I WON'T GET TRUNCATED";` ---> ONLY TRUBCATES DECIMAL PLACES

OUTPUT:

```
+-----+
| I WON'T GET TRUNCATED |
+-----+
|          123456 |
+-----+
```

CODE: `SELECT TRUNCATE(123.456 , 0) AS "MISSING:456";`

OUTPUT:

```
+-----+
| MISSING:456 |
+-----+
|          123 |
+-----+
```

## DATE AND TIME FUNTIONS

# CURDATE() / CURRENT\_DATE() / CURRENT\_DATE ---> RETURNS THE CURRENT DATE

CODE: `SELECT CURDATE();`

OUTPUT:

```
+-----+
| CURDATE() |
+-----+
| 2022-01-08 |
+-----+
```

CODE: `SELECT CURDATE() + 1;`

OUTPUT:

```
+-----+
| CURDATE() + 1 |
+-----+
| 20220109 | ---> 2022-01-08 + 1 ---> 20220109
+-----+
-----
```

#DATE() ---> USED TO EXTRACT YYYY-MM-DD PART

CODE: `SELECT DATE('2001-01-01');`

OUTPUT:

```
+-----+
| DATE('2001-01-01') |
+-----+
| 2001-01-01 |
+-----+
```

CODE: `SELECT DATE('2001-01-01 01:01:01');`

OUTPUT:

```
+-----+
| DATE('2001-01-01 01:01:01') |
+-----+
| 2001-01-01 |
+-----+
```

#MONTH() --> USED TO EXTRACT MM PART

CODE: `SELECT MONTH('2001-01-01');`

OUTPUT:

```
+-----+
| MONTH('2001-01-01') |
+-----+
| 1 |
+-----+
```

#YEAR() ---> SELF EXPLANATORY

CODE: `SELECT YEAR('2001-01-01');`

OUTPUT:

```
+-----+
```

```

| YEAR('2001-01-01') |
+-----+
|                2001 |
+-----+

```

#NOW() ----> RETURNS US THE TIME WHEN THE FUNTION STARTED TO GET EXCECUTED

CODE: SELECT NOW();

OUTPUT:

```

+-----+
| NOW() |
+-----+
| 2000-01-01 01:01:01 |
+-----+

```

#SYSDATE() ----> RETURN US THE CURRENT DATE AND TIME

CODE: SELECT SYSDATE();

OUTPUT:

```

+-----+
| SYSDATE() |
+-----+
| 2000-01-01 01:01:01 |
+-----+

```

CODE: SELECT NOW() , SLEEP(5) , NOW();

OUTPUT:

```

+-----+-----+-----+
| NOW() | SLEEP(5) | NOW() |
+-----+-----+-----+
| 2000-01-01 01:01:01 | 0 | 2000-01-01 01:01:01 | ----> SAME TIME
+-----+-----+-----+

```

CODE: SELECT SYSDATE() , SLEEP(5) , SYSDATE();

OUTPUT:

```

+-----+-----+-----+
| SYSDATE() | SLEEP(5) | SYSDATE() |
+-----+-----+-----+
| 2000-01-01 01:01:01 | 0 | 2000-01-01 01:01:06 | ----> INITIAL
TIME + 5 SECONDS
+-----+-----+-----+

```

CODE: SELECT NOW() , SLEEP(5) , SYSDATE();

OUTPUT:

```

+-----+-----+-----+
| NOW() | SLEEP(5) | SYSDATE() |
+-----+-----+-----+
| 2000-01-01 01:01:01 | 0 | 2000-01-01 01:01:06 | ----> INITIAL
TIME + 5 SECONDS
+-----+-----+-----+

```

CODE: SELECT SYSDATE() , SLEEP(5) , NOW();

OUTPUT:

```
+-----+-----+-----+
| SYSDATE() | SLEEP(5) | NOW() |
+-----+-----+-----+
| 2000-01-01 01:01:01 | 0 | 2000-01-01 01:01:01 | ---> SAME TIME
+-----+-----+-----+
```

## Null Handling

Let us consider the table given below:

NAME OF THE TABLE: records3

```
+-----+-----+-----+
| name | year | present |
+-----+-----+-----+
| ram  | 2001 | present |
| sam  | 2002 | present |
| ramu | 2003 | NULL    |
+-----+-----+-----+
```

To create the table above use the following commands:

```
USE db; ---> (use database_name)
CREATE TABLE records3(name varchar(10) , year integer , present varchar(10));
INSERT INTO records3 VALUES('ram' , 2001 , 'present') , ('sam' , 2002 ,
'present'), ('ramu' , 2003 , NULL);
```

NULL here in the present column means that the person is absent on that particular day(2001-01-01).

The day we are here referring to is 2001-01-01.

#IFNULL()

Syntax: IFNULL(column\_name , value\_to\_be\_substitued)

IFNULL() ---> Used to change all the NULL value from the give column to the given value

```
CODE: SELECT name , year , IFNULL(present , 'absent')
      FROM records3;
```

OUTPUT:

```
+-----+-----+-----+
| name | year | IFNULL(present , 'absent') |
+-----+-----+-----+
| ram  | 2001 | present                    |
| sam  | 2002 | present                    | ---> NULL values are
changed into absent
| ramu | 2003 | absent                     |
+-----+-----+-----+
```

```
=====
=====
```

#USING \*\*AS\*\* WITH IFNULL()

```
CODE: SELECT name , year , IFNULL(present , 'absent') AS 'attendance'
```

```
FROM records3;
```

OUTPUT:

name	year	attendance
ram	2001	present
sam	2002	present
ramu	2003	absent

#FINDING NULL USING THE WHERE OPERATOR:

```
CODE: SELECT *
      FROM records3
      WHERE present IS NULL; ---> SELECTS ALL THE NULL VALUE
```

OUTPUT:

name	year	present
ramu	2003	NULL

```
CODE: SELECT *
      FROM records3
      WHERE present IS NOT NULL; ---> SELF - EXPLANATORY
```

OUTPUT:

name	year	present
ram	2001	present
sam	2002	present

## MISSSED NUANCES

```
/* In this section we will cover a few nuances that we have missed earlier */
```

```
SELECT year (or) SeLeCt year (or) sELECT yEar
FROM records; (or) fRoM records; (or) fRoM rECROds;
```

ALL THE ABOVE QUERIES YIELD THE SAME RESULT ---> SQL IS CASE INSENSTIVE

NUMBER()

NUMBER(5,3) ---> NUMBER WITH A MAXIMUM OF 5 DIGIT WITH 3 DECIMAL PLACES

STRING VS NUMERIC FUNTIONS

THE OUTPUT OF ALL THE STRING FUNCTIONS STARTS FROM THE LEFT

THE OUTPUT OF ALL THE NUMERIC FUNCTIONS STARTS FROM THE RIGHT

NUMERIC FUNCTION

```
+-----+
| MISSING:456 |
+-----+
|           123 |
+-----+
```

STRING FUNCTION

```
+-----+
| LOWER("PYTHON") |
+-----+
| python          |
+-----+
```

```
=====
=====
```

SUBSTR(given\_string , start\_index , no\_of\_characters)

start\_index ---> can be negative or positive (POSITIVE OR NEGATIVE NUMBERS)

no\_of\_character ---> must be a positive integer(NATURAL NUMBERS)

CODE: SELECT SUBSTR('0123456789' , 3 , -4);

OUTPUT:

```
+-----+
| SUBSTR('0123456789' , 3 , -4) |
+-----+
|                               |
+-----+
```

```
=====
=====
```

USING ARITHMETIC AND RELATIONAL OPERATOR WITH DATE AND TIME FUNCTIONS

CODE: SELECT YEAR('2001-01-01') + 10;

OUTPUT:

```
+-----+
| YEAR('2001-01-01') + 10 |
+-----+
|                2011 |
+-----+
```

```
=====
=====
```

CODE: SELECT YEAR('2001-01-01') > 10;

OUTPUT:

```
+-----+
| YEAR('2001-01-01') > 10 |
+-----+
|                1 | --> 1 --> True
+-----+
```

## TABLE CREATION COMMANDS (CONTINUED)

### DROP

#TO DELETE A DATABASE

SYNTAX: **DROP DATABASE** database\_name;  
EXAMPLE: **DROP DATABASE** db; #---> Database db will be deleted if it exists  
OUTPUT: Query OK, 0 rows affected (0.31 sec)

#### #TO DELETE A TABLE

SYNTAX: **DROP table\_name**;  
EXAMPLE: **DROP** records; #---> Table records will be deleted if it exists  
OUTPUT: Query OK, 0 rows affected (1.15 sec)

#### #TO DELETE A COLUMN

SYNTAX: **ALTER TABLE table\_name**  
          **DROP COLUMN** column\_name;  
EXAMPLE: **ALTER TABLE** records  
          **DROP COLUMN** year; # column year will be deleted

#### #IF EXISTS

CAN BE USED TO DELETE A TABLE IF IT EXISTS

SYNTAX: **DROP TABLE IF EXISTS table\_name**;  
SYNTAX: **DROP TABLE IF EXISTS** records; #---> Table records will be deleted if it exists

## CONSTRAINT

#### #CONSTRAINT

DATABASE INTEGRITY CONSTRAINTS:

- (i) Unique constraint
  - (ii) Primary Key constraint
  - (iii) Foreign Key constraint
  - (iv) Check constraint
  - (v) Default Key constraint
  - (vi) NOT NULL
  - (vii) ENUM
  - (viii) SET
- and so on...

=====

#NOT NULL ---> SHOULD BE USED WHEN YOU DON'T WANT AN COLUMN TO ACCEPT NULL DATA

CODE: **CREATE TABLE** records4  
      (sno **integer**,  
      student\_name **varchar**(10)  
      ,email **varchar**(20) **NOT NULL**) ;  
/\* email column now can't accept NOT NULL values\*/

CODE: **INSERT INTO** records4 **VALUE**(1,'ram',NULL);  
OUTPUT: ERROR 1048 (23000): Column 'email' cannot be null

=====



#UNIQUE ---> SHOULD BE USED WHEN YOU DON'T WANT A COLOUMN TO HAVE UNIQUE RECORDS(NO DUPLICATE RECORDS)

```
CODE: CREATE TABLE records6
      ( sno          integer      NOT NULL UNIQUE,
        student_name varchar(10) ,
        email         varchar(20) NOT NULL) ;
```

```
CODE: INSERT INTO records6 VALUE
      (1,'ram','ram@gmail.com');
```

OUTPUT: Query OK, 1 row affected

```
CODE: INSERT INTO records6 VALUE
      (1,'ram','ram@yahoo.com');
```

OUTPUT: ERROR 1062 (23000): Duplicate entry '1' for key 'records6.sno'

=====

#PRIMARY KEY ---> CAN BE APPLIED TO ONLY ONE COLOUMN AND IT DOESN'T ALLOW NULL VALUES

# ERROR(ERROR 1068 (42000): Multiple primary key defined) --->  
PRODUCED WHEN APLIED TO MULTIPLE COLUMNS

PRIMARY KEY ---> THIS IS SERVES AS AN UNIQUE IDENTIFIER

---> TWO PERSON CAN HAVE SAME NAME BUT THEY CAN'T HAVE SAME FINGERPRINT

---> HERE FINGERPRINT SERVES THE PURPOSE OF PRIMARY KEY

---> IN TABLE WE MUST HAVE A PRIMARY KEY TO UNIQUELY IDENTIFY A RECORDS IN A TABLE

AS WE KNOW PRIMARY KEY DOES NOT ALLOW NULL VALUES THE PRIMARY KEY ALSO ACTS LIKE NOT NULL CONSTARINT

```
CODE: CREATE TABLE records7
      (name NOT NULL PRIMARY KEY);    ---> NOT NULL MAY OR MAY NOT BE USED WITH
PRIMARY KEY
```

(OR)

```
CREATE TABLE records7
      (name PRIMARY KEY); #PRIMARY KEY ALSO ACTS LIKE NOT NULL
```

=====

#DEFAULT CONSTRAINT

```
CODE: CREATE TABLE records8(name DEFAULT 'I AM')
```

```
      INSERT INTO records8 VALUE();
```

/\* IN THIS CASE THE DEFAULT VALUE 'I AM' WILL BE ADDED \*/

```
      SELECT * FROM records8;
```

OUTPUT:

```
      +-----+
      | name |
```

```
+-----+
| I AM |
+-----+
```

NOTE: THE MAX SIZE OF DEFAULT VALUE IS 10;

# AS WE DID NOT INCLUDE ANY VALUE WHILE ADDING RECORDS THE DEFAULT VALUE 'I AM' WAS INSERTED

```
=====
=====
```

#CHECK CONSTRAINT---> CAN BE USED WHEN YOU WANT TO ALLOW CONSTRAINTS BASED ON CERTAIN LIMIT

CODE: CREATE TABLE records9 VALUE  
 (name varchar(10) ,  
 age integer CHECK(age > 18) ) #IT ONLY ALLOWS VALUES GREATER THAN  
 18 IN THE age COLUMN

CHECK(column\_1 < column\_2) ---> CAN BE USED TO COMPARE TWO COLUMNS

name varchar(10) CHECK(name in ('ram' , 'som' , 'ramu'))

BETWEEN , LOGICAL OPERATOR AND OTHER OPERATORS CAN BE USED.

```
=====
=====
```

#FOREIGN KEY CONSTARINT

#FOREIGN KEY IS LIKE PRIMARY KEY. IT IS USED IN RDBMS.

#SO FAR WE HAVE ONLY SEE DBMS.

DBMS ---> Database Management System

RDBMS ---> Relational Database Management System

IN RDBMS TABLES ARE IN RELATION WITH EACH OTHER BUT IN DBMS TABLES ARE NOT IN RELATION WITH EACH OTHER.

DBMS

DATABSSE NAME: RDBMS

id	name	age
1	ram	10
2	sam	20
3	ram	30

TABLE NAME: AGE

id	name	gender
2	sam	male
1	ram	female
3	ram	male

TABLE NAME: GENDER

AS YOU CAN SEE BOTH THE TABELS ARE RELATED TO EACH OTHER BY THE ID COLOUMN

id column in table AGE is called the primary key and id in table GENDER is called primary key

id column is called as the foreign key as it is used to relate the two tables AGE AND GENDER.

WE can even choose the gender column as the primary key but we can't choose it as foreign key as it is not present in the age table.

DATABASE DBMS:

weight	name	age
40	ram	10
50	sam	male
60	ram	male

TABLE NAME: AGE

id	name	gender
2	sam	male
1	ram	female
3	ram	male

TABLE NAME: GENDER

TWO TABLES ARE NOT RELATED TO EACH OTHER SO IT IS CALLED AS DBMS

=====

#FIRST LET US CREATE A PARENT TABLE WITH A PRIMARY KEY sno

CODE: CREATE TABLE parent(sno integer NOT NULL PRIMARY KEY)

#NOW LET US CREATE A CHILD TABLE WITH A FOREIGN KEY sno AND id AS PRIMARY KEY

CODE: CREATE TABLE child  
(sno integer NOT NULL PRIMARY KEY,

sno integer REFERENCES parent (sno))

#If we skip sno MySQL will the reference the primary key of the the table parent by default

=====

#ON DELETE CASCADE

#FIRST LET US CREATE A PARENT TABLE WITH A PRIMARY KEY sno

CODE: CREATE TABLE parent(sno integer NOT NULL PRIMARY KEY)

#NOW LET US CREATE A CHILD TABLE WITH A FOREIGN KEY sno AND id AS PRIMARY KEY

CODE: CREATE TABLE child  
(sno integer NOT NULL PRIMARY KEY,

sno integer REFERENCES parent (sno)) ON DELETE CASCADE

-----

ON DELETE CASCADE:

--> To be used when you want the related rows in child table to get deleted when the row gets deleted in the parent table

--> For example let's say a row you deleted a row in the parent table all the related row which are present in the child will get deleted automatically if you use ON DELETE CASCADE

=====

#ON UPDATE CASCADE

#FIRST LET US CREATE A PARENT TABLE WITH A PRIMARY KEY sno  
CODE: CREATE TABLE parent(sno integer NOT NULL PRIMARY KEY)

#NOW LET US CREATE A CHILD TABLE WITH A FOREIGN KEY sno AND id AS PRIMARY KEY  
CODE: CREATE TABLE child

(sno integer NOT NULL PRIMARY KEY,  
  
sno integer REFERENCES parent (sno)) ON UPDATE CASCADE

-----  
ON UPDATE CASCADE:

--> To be used when you want the changes in the parent table to reflect back in the child table(only related rows get updated with the new changes)

--> For example let's say you update a row in the parent table all the related row which are present in the child table will get updated with the new changes automatically if you use ON UPDATE CASCADE.

## TABLE CONSTRAINTS

#TABLE CONSTRAINTS ---> CONSTRAINT APPLIED TO MULTIPLE COLUMNS

CODE: CREATE TABLE t1  
(age integer,  
name VARCHAR(10) NOT NULL,  
email VARCHAR(20) NOT NULL  
UNIQUE(name , email)); #UNIQUE CONSTRAINT WILL BE APPLIED TO name and  
email column  
FOREIGN KEY(sno) REFERENCES records(sno)

## NAMED CONSTRAINTS

#ASSIGNING NAME TO CONSTRAINTS

MySQL my default assigns name to constraints in the format SYS\_Cn , where n is an integer

For eg: SYS\_C123456 , SYS\_C654321

But we can force change the name of the constraint

SYNTAX: CONSTRAINT the\_name\_you\_want constraint\_name;

CODE: CREATE TABLE students (  
id INTEGER CONSTRAINT new\_name PRIMARY KEY,  
NAME varchar(15)  
);

```
#DEFAULT NAME OF PRIMARY KEY CONSTRAINT HAS BEEN CHANGED TO new_name
#
```

## TABLE CREATION FROM EXISTING TABLE

```
#CREATING TABLE FROM EXISTING TABLE
```

```
#METHOD 1:(INSERT AND SELECT)
```

```
#table_2 should be created earlier
```

```
#column datatype and count in records should match column datatype in table_2
```

```
CODE: INSERT INTO table_2
```

```
    SELECT student_name FROM records; #RECORDS IN records ---> COPIED TO
TABLE_2
```

```
#METHOD 2:(AS SELECT)
```

```
CODE: CREATE TABLE T2 AS
```

```
    SELECT student_name , year FROM records; #USE OF AS IS OPTIONAL
```

```
+-----+-----+
| student_name | year |
+-----+-----+
| ram          | 2004 |
| sam          | 2003 |
| hari         | 2002 |
| ramu         | 2004 |
+-----+-----+
```

```
THE ABOVE RECORDS WILL BE STORED IN BOTH THE CASE
```

## UPDATE

```
SYNTAX: UPDATE table_name
        SET column_name = value;
```

```
#TO UPDATE ALL THE RECORDS IN A COLUMN
```

```
CODE: UPDATE records
```

```
    SET year = 2000;
```

```
TABLE:
```

```
+-----+-----+-----+-----+-----+
| sno | student_name | email          | year | column_name |
+-----+-----+-----+-----+-----+
| 1   | ram          | ram@gmail.com  | 2000 |              |
| 2   | sam          | sam@yahoo.com  | 2000 |              |
| 3   | hari         | hari@outlook.com | 2000 |              |
| 4   | ramu         | ramu@gmail.com | 2000 |              |
+-----+-----+-----+-----+-----+
```

#TO UPDATE A PARTICULAR RECORD

WE CAN USE WHERE TO ACHIEVE THIS TASK

CODE: UPDATE records  
SET year = 2000  
WHERE student\_name = "ram";

TABLE:

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2000	10
2	sam	sam@yahoo.com	2003	20
3	hari	hari@outlook.com	2002	30
4	ramu	ramu@gmail.com	2004	20

SET column\_name = 2000 + value #USING EXPRESSION

## DELETE VS TRUNCATE (ROW OPERATION)

# BOTH DELETE AND TRUNCATE ARE USED TO DELETE ROWS AND COLUMNS , COLOUMNS RESPECTIVELY BUT THE TABLE NAME STILL EXIST  
#BUT DROP IS USED TO DELETE THE TABLE COMPLETELY [TABLE\_NAME + ALL THE ROWS AND COLUMNS] OR A PARTICULAR COLUMN

#TO DELETE ALL THE ROWS FROM A TABLE USING DELETE

SYNTAX: DELETE FROM table\_name;  
CODE: DELETE FROM records;  
OUTPUT: Query OK, 0 rows affected (1.57 sec)  
TABLE: Empty set (0.00 sec)

#THE ABOVE CODE DELETES ALL THE ROWS AND COLUMNS BUT THE TABLE IS STILL THERE

#TO DELETE ALL THE ROWS FROM A TABLE USING TRUNCATE

SYNTAX: TRUNCATE table\_name;  
CODE: TRUNCATE records;  
OUTPUT: Query OK, 0 rows affected (1.12 sec)  
TABLE: Empty set (0.00 sec)

#THE ABOVE CODE DELETES ALL THE ROWS AND COLUMNS BUT THE TABLE IS STILL THERE

#TO DELETE A PARTICULAR ROW FROM A TABLE USING DELETE

SYNTAX: DELETE FROM table\_name  
WHERE condition

CODE: DELETE FROM records  
WHERE name = 'ram';

TABLE:

sno	student_name	email	year	column_name
2	sam	sam@yahoo.com	2003	20
3	hari	hari@outlook.com	2002	30
4	ramu	ramu@gmail.com	2004	20

=====

#TO DELETE A PARTICULAR ROW FROM A TABLE USING TRUNCATE

WE CAN '' DO THAT USING TRUNCATE

DELETE VS TRUNCATE

DELETE	TRUNCATE
Can be used to remove a single or multiple rows	Can be <b>used</b> to remove all the rows
WHERE clause can be used	WHERE clause can't be used
Slower in comparison	Faster in comparison
DML Command	DDL Command
It logs each deleted row in transaction log (OUT OF SYLLABUS)	It doesn't log each deleted row in transaction log (OUT OF SYLLABUS)

SOURCE: t-sibiraj.github.io/sql

## ROLLBACK

THE FOLLOWING CONCEPT IS (/\*OUT OF SYLLABUS \*/)

OUT OF SYLLABUS ----- OUT OF SYLLABUS ----- OUT OF SYLLABUS -----  
OUT OF SYLLABUS -----

#WHEN WE DO SOME CHANGES ,WE CAN UNDO THE CHANGES IF WE USE DELETE, WHICH WE CAN'T DO WHEN WE USE TRUNCATE

commit ---> to be used to save changes

rollback --> like the undo button which we can use to rollback the changes

CODE: mysql>DELETE FROM records  
WHERE year = 2004;

mysql> COMMIT

mysql> ROLLBACK

#now we can undo the last transaction. The commit acts like an checkpoint to which we can revert back using ROLLBACK

OUT OF SYLLABUS ----- OUT OF SYLLABUS ----- OUT OF SYLLABUS -----  
OUT OF SYLLABUS -----

# ALTER

SYNTAX: `ALTER TABLE table_name`  
`clause_name name`

## ADD

#TO ADD A COLUMN

SYNTAX: `ALTER TABLE table_name`  
`ADD column_name datatype(size) CONSTRAINT constraint_name; size and`  
`constraint --> optional`

CODE: `ALTER TABLE records`  
`ADD (result integer); #USE OF () IS OPTIONAL WHILE ADDING SINGLE COLUMN`

TABLE:

sno	student_name	email	year	column_name	result
1	ram	ram@gmail.com	2004	10	NULL
2	sam	sam@yahoo.com	2003	20	NULL
3	hari	hari@outlook.com	2002	30	NULL
4	ramu	ramu@gmail.com	2004	20	NULL

#TO ADD MULTIPLE COLUMN

CODE: `ALTER TABLE records`  
`ADD (maths integer , computer_science integer);`

TABLE:

sno	student_name	email	year	column_name	result	maths	computer_science
1	ram	ram@gmail.com	2004	10	NULL	NULL	NULL
2	sam	sam@yahoo.com	2003	20	NULL	NULL	NULL
3	hari	hari@outlook.com	2002	30	NULL	NULL	NULL
4	ramu	ramu@gmail.com	2004	20	NULL	NULL	NULL

#TO ADD A COLUMN WITH A CONSTRAINT



```
CODE: ALTER TABLE records
      ADD (physics integer NOT NULL);
```

WE HAVE DELETED ALL THE COLUMNS WHICH HAS NULL VALUE BEFORE ADDING PHYSICS COLUMN

TABLE:

	sno	student_name	email	year	column_name	physics
	1	ram	ram@gmail.com	2004	10	0
	2	sam	sam@yahoo.com	2003	20	0
	3	hari	hari@outlook.com	2002	30	0
	4	ramu	ramu@gmail.com	2004	20	0

#0 IS ADDED BY DEFAULT AS WE HAVE ADDED NOT NULL CONSTARINT

## CHANGE

```
SYNTAX: ALTER TABLE table_name
        CHANGE name
```

CHANGE:

---> Can rename a column and change its definition, or both.

#TO CHANGE THE COLUMN NAME

```
SYNTAX: ALTER TABLE table_name
        CHANGE old_column_name new_column_name datatype(size) CONSTRAINT
constraint_name;
```

```
CODE: ALTER TABLE records
      CHANGE physics description varchar(250);
```

TABLE:

	sno	student_name	email	year	column_name	physics
	1	ram	ram@gmail.com	2004	10	0
	2	sam	sam@yahoo.com	2003	20	0
	3	hari	hari@outlook.com	2002	30	0



description	varchar(50)	YES		NULL	
sno	int	NO	PRI	NULL	
student_name	varchar(10)	YES		NULL	
email	varchar(20)	YES		NULL	
year	int	YES		NULL	
column_name	int	YES		NULL	

## DROP

### #TO REMOVE A COLUMN

SYNTAX: ALTER TABLE table\_name  
DROP column\_name;

CODE: ALTER TABLE records  
DROP description; #description column ----> deleted

TABLE:

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
2	sam	sam@yahoo.com	2003	20
3	hari	hari@outlook.com	2002	30
4	ramu	ramu@gmail.com	2004	20

### #TO REMOVE THE PRIMARY KEY

SYNTAX: ALTER TABLE table\_name  
DROP PRIMARY KEY;

CODE: ALTER TABLE records  
DROP PRIMARY KEY; #sno will no longer be primary key

TABLE:

Field	Type	Null	Key	Default	Extra
sno	int	NO		NULL	
student_name	varchar(10)	YES		NULL	
email	varchar(20)	YES		NULL	
year	int	YES		NULL	
column_name	int	YES		NULL	

### #TO REMOVE THE FOREIGN KEY

SYNTAX: ALTER TABLE table\_name

```
DROP constraint_name column_name;
```

```
CODE: ALTER TABLE records
      DROP FOREIGN KEY email; #email will no longer be foreign key
```

```
=====
=====
```

#CASCADE

```
CODE: ALTER TABLE table_name
      DROP PRIMARY KEY CASCADE;
```

#When we use CASCADE it removes(drops) any foreign key which references the primary key

## GROUP BY(COMING SOON)(IMPORTANT)

#GROUP BY IS AN MULTIPLE ROW FUNTION LIKE AGGREGATE FUNCTION

## JOINS

WE CAN ALSO ACCESS COLUMNS BY USING THE BELOW FORMAT

```
table_name.column_name ----> .(dot)
alias_table_name.column_name ----> .(dot)
```

```
column_name
alias_column_name
```

JOIN ----> JOIN is nothing but a query which can be used to combine rows from two or more tables

## CARTESIAN PRODUCT

LET US CONSIDER THE TABLES BELOW

TABLE NAME: records

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
2	sam	sam@yahoo.com	2003	20
3	hari	hari@outlook.com	2002	30
4	ramu	ramu@gmail.com	2004	20

TABLE NAME: test\_table

name	year	present
------	------	---------

ram	2004	NULL	
NULL	NULL	NULL	
sam	2001	absent	

CARTESIAN PRODUCT IS NOTHING BUT UNSRESTRICTED JOIN

TABLE STRUCTURE OF records:  
test\_table

TABLE STRUCTURE OF

records

_____ sno
_____ student_name
_____ email
_____ year
_____ column_name

test\_table

_____ name
_____ year
_____ present

In SQL we can write the Cartesian product of records and test\_table as follows

CODE: SELECT \*  
FROM records, test\_table

#Now the a total 15 records will be shown (5 \* 3)

#Each row in records table will be multiplied with all the row from test\_table

sno	student_name	email	year	column_name	name	year	present
1	ram	ram@gmail.com	2004	10	sam	2001	absent
1	ram	ram@gmail.com	2004	10	NULL	NULL	NULL
1	ram	ram@gmail.com	2004	10	ram	2004	NULL
2	sam	sam@yahoo.com	2003	20	sam	2001	absent
2	sam	sam@yahoo.com	2003	20	NULL	NULL	NULL
2	sam	sam@yahoo.com	2003	20	ram	2004	NULL
3	hari	hari@outlook.com	2002	30	sam	2001	absent
3	hari	hari@outlook.com	2002	30	NULL	NULL	NULL
3	hari	hari@outlook.com	2002	30	ram	2004	NULL
4	ramu	ramu@gmail.com	2004	20	sam	2001	absent
4	ramu	ramu@gmail.com	2004	20	NULL	NULL	NULL
4	ramu	ramu@gmail.com	2004	20	ram	2004	NULL

SOURCE: [t-sibiraj.github.io/sql](https://t-sibiraj.github.io/sql)

AS you can see the row 1,ram,ram@gmail.com,2004 from table records is multiplied with all the rows of the table test\_table which is colour coded in blue

A MORE SIMPLE VERSION IS GIVEN BELOW

CODE: SELECT student\_name , present  
FROM records , test\_table;

----> student\_name [cartesian product] present

```
mysql> SELECT student_name , present
-> FROM records , test_table;
```

student_name	present
ram	absent
ram	NULL
ram	NULL
sam	absent
sam	NULL
sam	NULL
hari	absent
hari	NULL
hari	NULL
ramu	absent
ramu	NULL
ramu	NULL

12 rows in set (0.00 sec)

SOURCE: [t-sibiraj.github.io/sql](https://t-sibiraj.github.io/sql)

As you can see above row ram is multiplied with all the three rows present in the **present** column. And the same is repeated with sam , hari and ramu row.

```
ram * ( absent + NULL + NULL)
sam * (absent + NULL + NULL)
hari * (absent + NULL + NULL)
ramu * (absent + NULL + NULL)
```

THE ORDER WOULD HAVE BEEN CHANGED IF THE COLUMN present WAS WRITTEN BEFORE THE name column.

## TABLE ALIASES

### #TABLE ALIASES

LIKE COLUMN ALIASES WE CAN HAVE ALIAS NAME FOR TABLES TOO

SYNTAX: `SELECT table_alias_1.column_name , table_alias_2`  
`FROM table_name_1 table_alias_1 , table_name_2 table_alias_2;`

CONSIDER THE TABLES BELOW

TABLE NAME: records

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
2	sam	sam@yahoo.com	2003	20
3	hari	hari@outlook.com	2002	30
4	ramu	ramu@gmail.com	2004	20

TABLE NAME: test\_table

```
+-----+-----+-----+
```

name	year	present
ram	2004	NULL
NULL	NULL	NULL
sam	2001	absent

TABLE NAME: test\_table\_3

sno	student_name	email	year_of_birth	column_name
1	ram	ram@gmail.com	2004	10
2	sam	sam@yahoo.com	2003	20
3	hari	hari@outlook.com	2002	30
4	ramu	ramu@gmail.com	2004	20

CODE: `SELECT a1.student_name , a2.year_of_birth  
FROM records a1 , test_table_3 a2;`

OUTPUT:

student_name	year_of_birth
ramu	2004
hari	2004
sam	2004
ram	2004
ramu	2003
hari	2003
sam	2003
ram	2003
ramu	2002
hari	2002
sam	2002
ram	2002
ramu	2004
hari	2004
sam	2004
ram	2004

#WE CAN AVOID THE ABOVE SITUATION USING WHERE CLAUSE

CODE: `SELECT a1.student_name , a2.year_of_birth  
FROM records a1 , test_table_3 a2  
WHERE a1.sno = a2.sno;`

OUTPUT:

student_name	year_of_birth
ram	2004
sam	2003
hari	2002
ramu	2004

## EQUI - JOIN

### #EQUI - JOIN

- > Can be used to combine tables based on matching column values
- > Column names may or may be same
- > resultant table contains repeated columns

We can perform equi join in two ways:

WAY ONE:

SYNTAX: `SELECT *`  
`FROM table_name_1, tabel_name_2`  
`WHERE table_name_1.column_name = tabel_name_2.column_name;`

WAY TWO:

SYNTAX: `SELECT *`  
`FROM table_name_1`  
`JOIN tabel_name_2`  
`ON table_name_1.column_name = tabel_name_2.column_name;`

EXAMPLE:

TABLE NAME: e1

id	name
1	ram
2	sam
3	hari
4	som
8	tom
5	mike

TABLE NAME: e2

id	age
1	10
2	20
3	30
4	40
7	70
5	50

CODE:

```
SELECT *
FROM e1 , e2
WHERE e1.id = e2.id;
```

(OR)

```
SELECT *
FROM e1
JOIN e2
ON e1.id = e2.id;
```

The above code selects all the records from the two tables whihc have same id

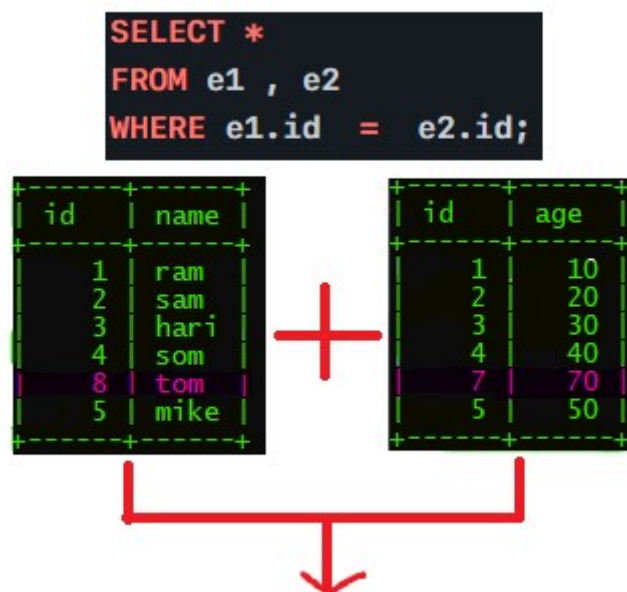




source: [t-sibiraj.github.io/learn](https://t-sibiraj.github.io/learn)

Now only **columns** colour coded (highlighted) in green will be selected as we have used the **condition** `e1.id = e2.id` in the **where** clause. Those in pink won't be selected.

the records (8 , 'tom') and (7,70) won't be selected



source: [t-sibiraj.github.io/learn](https://t-sibiraj.github.io/learn)

The resulting table will contain duplicate columns

```
mysql> SELECT * FROM e1 , e2 WHERE e1.id = e2.id;
```

id	name	id	age
1	ram	1	10
2	sam	2	20
3	hari	3	30
4	som	4	40
5	mike	5	50

source: [t-sibiraj.github.io/learn](https://t-sibiraj.github.io/learn)

# NATURAL JOIN

## #EQUI - JOIN

- > Can be used to used to combine tables which have column columns
- > No duplicate columns are returned
- > Column name and data type should be same

SYNTAX: `SELECT *`  
`FROM table_name_1 NATURAL JOIN table_name_2;`

EXAMPLE:

```
SELECT *  
FROM e1 NATURAL JOIN e2;
```

TABLE NAME: e1

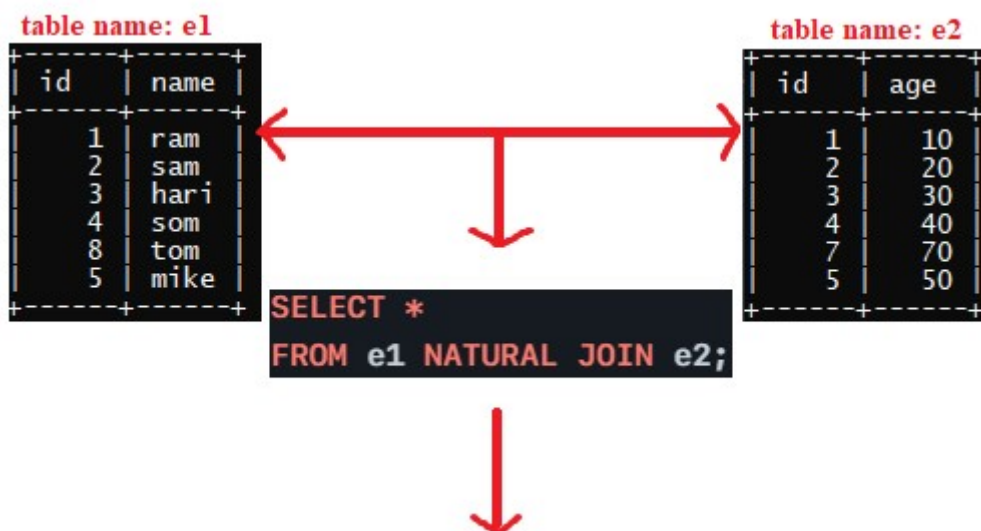
id	name
1	ram
2	sam
3	hari
4	som
8	tom
5	mike

TABLE NAME: e2

id	age
1	10
2	20
3	30
4	40
7	70
5	50

As we can see both the table has identical columns with same name and data.

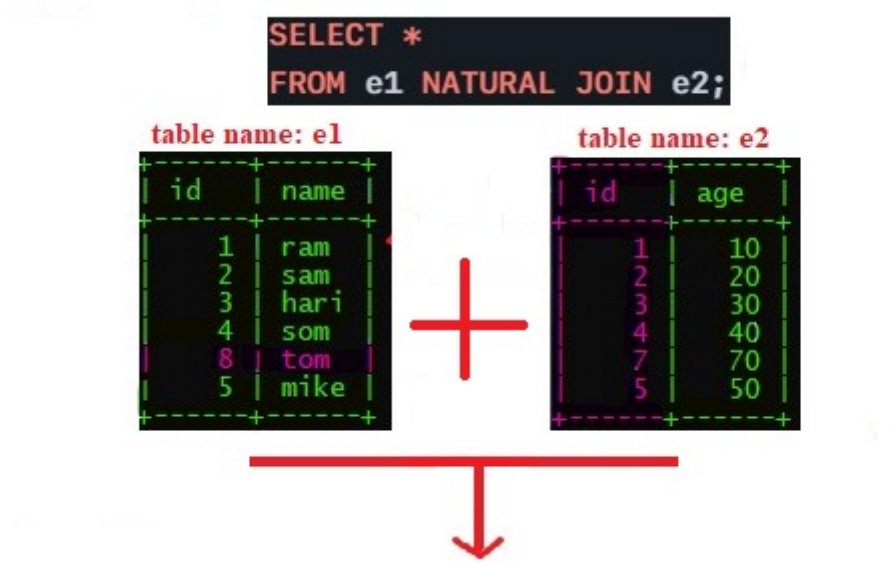
We can perform NATURAL JOIN. The output we get when we perform NATURAL JOIN will be similar to that of EQUI JOIN but no duplicates columns will be repeated.



source: [t-sibiraj.github.io/sql](https://t-sibiraj.github.io/sql)

As the id column is identical in both the tables we can perform natural join  
Now this time id column won't be displayed two times as we perform natural join

Those which are colour coded in green will be selected and those in red won't get selected.



source: [t-sibiraj.github.io/learn](https://t-sibiraj.github.io/learn)

The resulting table now contains only unique columns

```
mysql> SELECT * FROM e1 NATURAL JOIN e2;
```

id	name	age
1	ram	10
2	sam	20
3	hari	30
4	som	40
5	mike	50

source: [t-sibiraj.github.io/learn](https://t-sibiraj.github.io/learn)

## EQUI JOIN VS NATURAL JOIN

EQUI JOIN:

- > DUPLICATE COLUMNS
- > COLUMN NAME AND DATA TYPE MAY OR MAY NOT BE SAME
- > CAN BE APPLIED ON MULTIPLE COLUMNS
- > WE MUST MENTION COLUMN NAME

NATURAL JOIN:

- > UNIQUE COLUMNS
- > COLUMN NAME AND DATA TYPE MUST BE SAME
- > CAN BE APPLIED ON MULTIPLE COLUMNS
- > THERE IS NO NEED TO MENTION COLUMN NAME

TABLE NAME: e1

id	name
----	------

TABLE NAME: e2

id	age
----	-----

1	ram
2	sam
3	hari
4	som
8	tom
5	mike

1	10
2	20
3	30
4	40
7	70
5	50

```
mysql> SELECT * FROM e1 , e2 WHERE e1.id = e2.id;
```

id	name	id	age
1	ram	1	10
2	sam	2	20
3	hari	3	30
4	som	4	40
5	mike	5	50

```
mysql> SELECT * FROM e1 NATURAL JOIN e2;
```

id	name	age
1	ram	10
2	sam	20
3	hari	30
4	som	40
5	mike	50

source: t-sibiraj.github.io/learn

## PIP

### SIMPLE DEFINITION:

Think of **PyPI** as a place where people upload their python libraries and modules

Like a website where people upload education material

We can use the **pip** to install the libraries uploaded by the people on PyPI in our computer

We can use our browser to download the education material uploaded by others on the website in our computer

### FORMAL DEFINITION:

The Python Package Index (PyPI) is a repository of software for the Python programming language.

(source: <https://pypi.org/>)

repository: storage location for software packages

PIP is nothing but a package management system. It is used to download libraries, modules created by other people which they have uploaded to PyPI.

1. **OPEN CMD WITH ADMINISTER PRIVILEGE**
2. **TO CHECK IF PIP IS INSTALLED TYPE EITHER pip or pip3** #either should work
3. TYPE pip **install** name\_of\_the\_package or pip3 **install** name\_of\_the\_package

```
4.pip install mysql-connector-python
5.pip install pymysql
```

```
>>>from math import *
>>>floor(1.2)
1
>>>ceil(1.2)
2
```

```
#import the module
import mysql.connector as connector
```

[illegible]

```
'''
host ---> It is the host name or the IP address of the database server. As our
database is a local database we can use localhost

user ---> the username you have on MySQL

password ---> the password which you have set

database ---> this is optional. You should key in the name of the database
'''

#mysql.connector as connector
CODE: connection = connector.connect(host="localhost", user="root", passwd =
"root",
                                     database = "db")

=====

=====

#TO CHECK IF THE CONNECTION TO THE DATABASE IS SUCCESSFUL WE CAN USE
.is_connected() METHOD

SYNTAX: connection_object.is_connected() ---> True -----> Successfully Connected
        ---> False -----> Unsuccessful Connection

CODE:
>>> connector.is_connected()
True
```

## CREATING A CURSOR INSTANCE

```
#WE MUST USE CURSOR IF WE WANT TO PERFORM ROW BY ROW PROCESSING

#The output for our query get stored in the cursor we can access single or
multiple rows at a time from it .(This will get clear when we study about
fetchall(), fetchone()).

#The output for our query is called the resultset

=====

=====

#import mysql.connector as connector
#connection = connector.connect(details)

SYNTAX: cursor_object = connection_object.cursor()

CODE:   cursor = connection.cursor()
```

## RECORDS TABLE:

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
2	sam	sam@yahoo.com	2003	20
3	hari	hari@outlook.com	2002	30
4	ramu	ramu@gmail.com	2004	20

## EXCECUTING QUERIES

SYNTAX: `cursor_object.execute(your_sql_query)`

CODE: `cursor.execute("SELECT * FROM records")`

Note: -> The output that is the retrieved ,to be precise the resultset is now stored in the cursor

-> result set ---> output for your query

#We need the include the query within quotes and pass that string to `cursor.execute()`

## ACCESSING STORED RESULTSET(OUTPUT) FROM THE CURSOR\_OBJECT

`#fetchall()` ---> RETURNS ALL THE ROW FROM THE RESULT SET(OUTPUT FOR YOUR QUERY)

CODE:

```
print(cursor.fetchall)
```

OUTPUT:

```
[(1, 'ram', 'ram@gmail.com', 2004, 10), (2, 'sam', 'sam@yahoo.com', 2003, 20), (3, 'hari', 'hari@outlook.com', 2002, 30), (4, 'ramu', 'ramu@gmail.com', 2004, 20)]
```

#resultset is in the form of a list of rows(these rows are in the form of tuples)

\*\*\*NOTE: ONCE WE USE FETCH ALL WE NEED TO AGAIN EXECUTE THE QUERY USING `cursor.execute(query)**`

```
=====
```

#STORING THE RESULT SET IN A VARIABLE

CODE:

```
cursor.execute("SELECT * FROM records")
resultset = cursor.fetchall() #resultset is in the form of a list
print(resultset)
```

OUTPUT:

```
[(1, 'ram', 'ram@gmail.com', 2004, 10), (2, 'sam', 'sam@yahoo.com', 2003, 20), (3, 'hari', 'hari@outlook.com', 2002, 30), (4, 'ramu', 'ramu@gmail.com', 2004, 20)]
```

```
=====
=====
```

#### #ACCESSING INDIVIDUAL ROWS

CODE:

```
cursor.execute("SELECT * FROM records")
row1 = cursor.fetchall()[0]
print(row1)
```

OUTPUT:

```
(1, 'ram', 'ram@gmail.com', 2004, 10)
```

```
=====
=====
```

#### #TRAVERSING AND PRINTING ALL THE ROWS

CODE:

```
cursor.execute("SELECT * FROM records")
rows = cursor.fetchall()
for row in rows:
    print(row)
```

OUTPUT:

```
(1, 'ram', 'ram@gmail.com', 2004, 10)
(2, 'sam', 'sam@yahoo.com', 2003, 20)
(3, 'hari', 'hari@outlook.com', 2002, 30)
(4, 'ramu', 'ramu@gmail.com', 2004, 20)
```

#fetchmany() ---> Can be use to retrieve a particular number of rows

#### #RETRIEVE 2 RECORD FROM record TABLE

CODE:

```
cursor.execute("SELECT * FROM records")
two_record = cursor.fetchmany(2)
print(two_record)
```

OUTPUT:

```
[(1, 'ram', 'ram@gmail.com', 2004, 10), (2, 'sam', 'sam@yahoo.com', 2003, 20)]
```

#NOTE: The rows are in the form of a tuple inside a list

#WE CAN'T AGAIN FETCH THE FIRST TWO RECORDS. TO DO THAT WE SHOULD AGAIN EXECUTE QUERY FROM FIRST

```
=====
=====
```

CODE:

```
cursor.execute("SELECT * FROM records")
two_record = cursor.fetchmany(2)      #first two rows
print(two_record)
next_two_record = cursor.fetchmany(2)  #last two rows
print(next_two_record)
no_more_rows = cursor.fetchmany(2)    #As there is no more row to fetch ,
empty list is stored
print(no_more_rows)
```

OUTPUT:



```
[(1, 'ram', 'ram@gmail.com', 2004, 10), (2, 'sam', 'sam@yahoo.com', 2003, 20)]
[(3, 'hari', 'hari@outlook.com', 2002, 30), (4, 'ramu', 'ramu@gmail.com', 2004, 20)]
[]
```

#fetchone() ---> Can be used when we want to fetch one single row

#FETCH A ROW FROM records TABLE

CODE:

```
cursor.execute("SELECT * FROM records")
only_one_row = cursor.fetchone()
print(only_one_row)
```

OUTPUT:

```
(1, 'ram', 'ram@gmail.com', 2004, 10)
```

## rowcount()

#rowcount() ---> Can be used to know how many records(rows) have been retrieved so far

----> It takes account of the previous retrievals

CODE:

```
cursor.execute("SELECT * FROM records")

row1 = cursor.fetchone()
print("Rows(records) retrieved so far", cursor.rowcount()) #1

row2 = cursor.fetchone()
print("Rows(records) retrieved so far", cursor.rowcount()) #2

row3 = cursor.fetchmany(2)
print("Rows(records) retrieved so far", cursor.rowcount()) #4
```

OUTPUT:

```
Rows(records) retrieved so far 1
Rows(records) retrieved so far 2
Rows(records) retrieved so far 4
```

```
=====
=====
```

#IF YOU RUN INTO ERROR:

DOCS:

1. <https://dev.mysql.com/doc/connector-python/en/connector-python-tutorial-cursorbuffered.html>
2. <https://dev.mysql.com/doc/connector-python/en/connector-python-api-mysqldb-cursor-rowcount.html>

FORUMS:

1. <https://stackoverflow.com/questions/29772337/python-mysql-connector-unread-result-found-when-using-fetchone>

2. <https://arrayoverflow.com/question/python-mysql-connector-errors-internalerror-unread-result-found/3196>

## connection\_name.close()

#After retrieving the records and using the database we must close the connection

#To do that use the following command

```
connection.close()
```

## PYMYSQL

WE CAN DO SAME WITH ANOTHER LIBRARY CALLED PYMYSQL

pymysql vs mysql.connector:

---> pymysql purely written in python and made by python

---> mysql.connector made by oracle

```
#import pymysql
```

```
import pymysql as pym
```

#TO CREATE A CONNECTION(i.e TO CONNECT TO A DATABASE)

SYNTAX: connection\_name = pymysql.connect("host\_name" , "user\_name"  
,"password" , "database")

CODE: connection = pym.connect("localhost" , "root" , "root"  
,"db")

#TO TEST IF THE CONNECTION TO THE DATABASE WAS SUCCESSFULL

```
connection.is_connected()
```

NOTE: THE THE FOLLOWING STEPS ARE SAME LIKE mysql.connector library

#TO CREATE A CURSOR OBJECT

```
cursor = connection.cursor()
```

#TO EXECUTE QUERIES

```
cursor.execute("SELECT * FROM records")
```

#TO FETCH ROWS FROM RESULT SET(OUTPUT)

```
rows = cursor.fetchall()
```

#TO DISPLAY THE ROWS(RECORDS)

```
for row in rows:  
    print(row)
```

#ROWCOUNT()

```
count = cursor.rowcount()
```

# PARAMETERISED QUERIES

We provide some parameters or values from outside (by using function like `input()`) to run few queries

These queries are called as parameterised queries

## STRING FORMATTING

Before we learn about parameterised queries we must know what string formatting is.

### #NEW WAY TO FORMAT STRINGS

`"{}".format()` --> value inside the bracket gets substituted in the set bracket  
--> The set brackets are called as placeholders

```
>>>details = "My name is {} and I am {} years old".format("ram" , 20)
>>>print(details)
My name is ram and I am 20 years old
#ram goes into the first set bracket and 20 into the second
```

```
>>>details = "My name is {} and I am {} years old".format(20,"ram")
>>>print(details)
My name is 20 and I am ram years old
#20 goes into the first set bracket and ram into the second
```

```
=====
=====
```

```
>>>details = "My name is {0} and I am {1} years old".format("ram" , 20)
>>>print(details)
My name is ram and I am 20 years old
#ram is in zeroth index and 20 is in 1st index
```

```
>>>details = "My name is {1} and I am {0} years old".format(20,"ram")
>>>print(details)
My name is 20 and I am ram years old
'''ram is in zeroth index and 20 is in 1st index. As we have used 1st index first
, the value in the 1st index ("ram") gets substituted in the set bracket'''
```

```
=====
=====
```

```
>>>details = "My name is {name} and I am {age} years old".format(age = 20,name =
"ram")
>>>print(details)
My name is ram and I am 20 years old
#now we have named the placeholder values as name and age.
```

```
=====
=====
```

Example:

#Write a program in python where you should get year from the user and display the details of the that student whose year is greater than the given year. The details of the students are stored in a database

CODE:

```
import mysql.connector as connector
connection = connector.connect(host="localhost", user="root", passwd =
"root", database = "db")
cursor = connection.cursor()
year = input("Enter the year:")
cursor.execute("SELECT * FROM records WHERE year > {}".format(year))
print(cursor.fetchall())
connector.close()
```

OUTPUT:

```
Enter the year:2003
[(1, 'ram', 'ram@gmail.com', 2004, 10), (4, 'ramu', 'ramu@gmail.com', 2004,
20)]
```

#OLD WAY TO FORMAT STRINGS

#This type of formating is used in C language

```
%s ---> To be used with string (can also be used with numbers)
%d ---> To be used with integers
%f ---> To use used with float
```

%char acts like {}

CODE:

```
name = "ram"
age = 20
print("My name is %s and I am %d years old." % (name, age))
```

OUTPUT:

```
My name is ram and I am 20 years old
```

## cursor.commit()

#We should use cursor.commit() whenever we do some changes in the database

#So far we have only been retrieving the records(rows) from the database

#But when we execute queries which modify the database we must use the cursor.commit() to save changes in the database()

```
cursor.commit()
```

## INSERTING RECORDS USING MYSQL.CONNECTOR()

#Creating a cursor

```
import mysql.connector as connector
connection = connector.connect(host="localhost", user="root", passwd = "root",
database = "db")
cursor = connection.cursor()
```

```

#we should use cursor.commit() ---> As we are changing the database(i.e
inserting records)

#inserting records

cursor.execute("INSERT INTO records values(1, 'som', 'som@gmail.com' ,2005
,40)")
cursor.commit()
connector.close()
=====
=====

(OR)

=====
=====

query = "INSERT INTO records values(1, 'som', 'som@gmail.com' ,2005 ,40)"
cursor.execute(query)
cursor.commit()
connector.close()
=====
=====

(OR)

=====
=====

#Using parameterised queries

#getting the input
sno = int(input("Enter the sno:"))
name = input("Enter the student name:")
email = input("Enter the email:")
year = int(input("Enter the year of birth:"))
column_name = int(input("Eneter the column_name value:"))

#executing query
query = "INSERT INTO records values({}, {} , {} , {} ,
{} )".format(sno,name,email,year,column_name)
cursor.execute(query)
cursor.commit()
connector.close()
#We need the include the query within quotes and pass that string to
cursor.execute()

```

## UPDATING RECORDS USING MYSQL.CONNECTOR()

```

#Creaing a curosor
import mysql.connector as connector
connection = connector.connect(host="localhost", user="root", passwd = "root",
database = "db")
cursor = connection.cursor()

#UPDATING RECORDS

query = '''UPDATE records SET year = 2004
WHERE year = 2002 '''
#year 2002 gets updated with a value
of 2004

```

```

cursor.execute(query)
cursor.commit()                                #use commit() to save changes
connector.close()

=====

=====

(OR)

=====

=====

#Using parameterised queries
year_old = int(input("Enter the year which needs to be updated:"))
year_new = int(input("Enter the new value for the year:"))

query = "UPDATE record SET year = {} WHERE year = {}".format(year_old ,
year_new)
cursor.execute(query)
cursor.commit()
connector.close()

```

## DELETING RECORDS USING MYSQL.CONNECTOR()

```

#Creaiing a curosor
import mysql.connector as connector
connection = connector.connect(host="localhost", user="root", passwd = "root",
database = "db")
cursor = connection.cursor()

#DELETING RECORDS

query = "DELETE FROM records WHERE name = 'ram' "
cursor.execute(query)
cursor.commit()
connector.close()
#We should use commit() as we are modifying the database. To save changes we
should use cursor.commit()

=====

=====

(OR)

=====

=====

#using parameterised queries

name = input("Enter the name of student whose record you wish to be deleted:")
query = "DELETE FROM records WHERE name = {}".format("name")
cursor.execute(query)
cursor.commit()
connector.close()

```

## SIMILARITY BETWEEN mysql.connector() and python

```
fetchall()      ---> readallines()
fetchone()      ---> readline()
fetchmany(n)    ---> read(n).split()
```

all the fetch method works in linear fashion

once we access the first two rows we have only access to the next rows not the previous rows

when you open a text file and add or delete some data it's the same like adding or deleting records using execute

But we must hit the save button before closing the text file to save the changes. If we don't do that our changes won't get updated in the text files. To do the same in mysql.connector() we have the commit() method. It acts like a save button.

connector\_name.close() ---> It is the same like closing the text file which we have opened.

## TABLES USED

records:

sno	student_name	email	year	column_name
1	ram	ram@gmail.com	2004	10
2	sam	sam@yahoo.com	2003	20
3	hari	hari@outlook.com	2002	30
4	ramu	ramu@gmail.com	2004	20

records3:

name	year	present
ram	2001	present
sam	2002	present
ramu	2003	NULL

test\_table:

name	year	present
ram	2004	NULL
NULL	NULL	NULL
sam	2001	absent

test\_table\_2:

name	year	present
------	------	---------

ram	2001	NULL	
sam	2002	present	
ramu	2003	present	
som	2004	absent	
+-----+	+-----+	+-----+	

## DATABASE PORTION FOR TERM - II(2021-2)

(I) Database concepts: introduction to database concepts and its need

(II) Relational data model: relation, attribute, tuple, domain, degree, cardinality, keys (candidate key, primary key, alternate key, foreign key)

(III) Structured Query Language: introduction, Data Definition Language and Data Manipulation

Language, data type (char(n), varchar(n), int, float, date), constraints (not null, unique, primary

key), create database, use database, show databases, drop database, show tables, create table,

describe table, alter table (add and remove an attribute, add and remove primary key), drop table,

insert, delete, select, operators (mathematical, relational and logical), aliasing, distinct clause,

where clause, in, between, order by, meaning of null, is null, is not null, like, update command,

delete command

(IV) Aggregate functions (max, min, avg, sum, count), group by, having clause, joins : Cartesian product on two tables, equi-join and natural join

(V) Interface of python with an SQL database: connecting SQL with Python, performing insert, update,

delete queries using cursor, display data by using fetchone(), fetchall(), rowcount, creating

database connectivity applications

Views 23 / 1147



This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

<https://creativecommons.org/licenses/by-sa/4.0/legalcode>



