**What is a Database:**

A database is a collection of tables where the data is stored in a way that’s meant to be accessed in a certain way.

A database contains a method for accessing and manipulating the data stored within it.

**What is a DBMS (Database Management System):**

A DBMS is a software that accesses, retrieves and updates the data stored within a database.

We give commands to the DBMS to do all the actions it is capable of doing via queries.

EX- PostgreSQL, Oracle Database, MySQL, SQLite.

**Difference Between SQL and MySQL:**

SQL – Stands for Structured Query Language. It is the language we use to talk to our database.

MySQL – It’s a DBMS. We use SQL to write commands in MySQL, which then runs those commands to manipulate the database. Working with MySQL is primarily writing SQL.

Takeaways: –

1. Once we learn SQL, it’s pretty easy to switch to another DB that uses SQL.

2. What makes databases (DBMS) unique are the features they offer, not the language.

**SQL Statements or Keywords:**

These SQL keywords can be written in lower case but it is recommended to write them in upper case so as to differentiate them from other elements in the query.

\* - the star signs mean we want all the columns contained within a table

* SELECT – Used to select whatever we need to retrieve from the tables from the database –

Ex: -

SELECT \*

FROM table\_name;

* FROM – Used to specify which table we are retrieving information from.
* DROP – Used for deleting entire tables in a database or even a database itself,

Ex: -

DROP TABLE table\_name;

DROP DATABASE database\_name;

* WHERE – Used to specify conditions on the whole table to get the expected output.

EX: -

SELECT user\_name

FROM table\_name

WHERE score > 80;

* CREATE – Used for creating tables and databases from scratch.

EX: -

CREATE DATABASE database\_name;

CREATE TABLE table\_name(column\_name data\_type constraint)

* INSERT – Used for inserting value inside a table.

Ex: -

INSERT INTO table\_name(column1\_name, column2\_name)

VALUE (v1, v2);

Or

VALUES (v1, v2), (v3, v4), (v5, v6);

* UPDATE – Used for updating or changing the existing values

UPDATE table\_name

SET column\_name = new\_value

WHERE some\_condition\_to\_specify\_the\_row

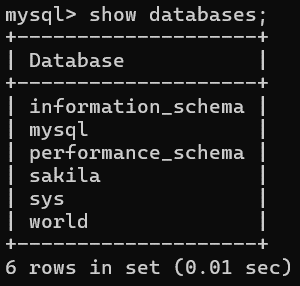
**Facts about MySQL:**

The root user is the highest-level user in MySQL.

Use the lightning mark to run the selected queries or all the queries. Use the second lightning mark to execute the query under the cursor.

The “;” tells the MySQL program where the end of the query is situated at in the program.

**Section – 3 -> Creating Databases & Tables**



We already have our Database server ready. Within a database server we can have multiple

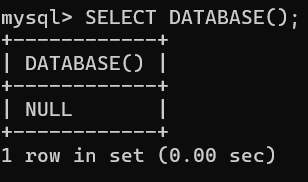
databases stored.

The query “SHOW DATABASES;” can be used to find all the databases stored within the

database server.

Creating a new database can done via this statement – CREATE DATABASE database\_name;

The name assigned to the newly empty database can have empty spaces between words but it is not recommended since that can cause complications later on.



Deleting a database can be achieved via this statement – DROP DATABASE database\_name;

To use a database, we need this statement – USE database\_name;

To check what database we are currently working on – SELECT DATABASE();

Unless we first specify on which database we wish to work on, we will get NULL in return.

We can also just double click on a database in the workbench to start using it. Note that we need to double click on the name of the database directly.

*Tables:*

Tables are what actually hold the data and in turn databases hold the tables. Without tables in them, databases are just empty husks so a database is just a bunch of tables.

All in all, Tables are a collection of related data held in a structured format within a database.

We define the structure of a table by specifying the columns and then insert the data within the table in a format specified by the structure.

We must also specify the type of data to expect in a particular column.

**Data Types:**

When we define the structure of a table, we are not only specifying the names of the columns but also the type of data allowed to be stored inside a column i.e., the type of information permitted in each of the column.

The link to the documentation of data types allowed by MySQL DBMS: -

<https://dev.mysql.com/doc/refman/8.4/en/data-types.html>

Important Data Types: -

* INT (a whole number) – Has a size of 4 bytes. Allows for integers with a max (signed) value of 2147483647 to be stored. Same for the negative side of the numbers.
* VARCHAR(number) (a variable length string) – number here specifies the maximum length of the string allowed.
* CHAR(number) – Has to be declared with a number that indicates the maximum number of characters we want to store in a row. The length of CHAR column is fixed to the length that we declare when we create a table. The length can be a value from 0 to 255.

When CHAR values are stored, they are right-padded with spaces to the specified length. When CHAR values are retrieved, trailing spaces are removed unless the PAD\_CHAR\_TO\_FULL\_LENGTH SQL mode is enabled.

**Syntax for Creating Tables: -**

|  |  |
| --- | --- |
| Syntax | Example |
| CREATE DATABASE database\_name  USE database\_name  CREATE TABLE table\_name  (  column1\_name data\_type,  column2\_name data\_type,  …  ); | CREATE DATABASE Animals  USE Animals  CREATE TABLE Cats  (  Name VARCHAR(20),  Age INT,  …  ); |

**How Do We Know That Our Queries Worked and A Table Was Successfully Created?**

We can use the following query:

USE database\_name

SHOW TABLES;

To look whether the column names were assigned correctly we can use the following query:

USE database\_name

SHOW COLUMNS

FROM table\_name;

OR

USE database\_name

DESCRIBE/DESC table\_name;

Commenting in SQL is done via “-- “ before the line of code. We can comment multiple line of code by selecting them and then using Ctrl + /.

**Deleting Tables:**

Deleting a table can be achieved by this line – DROP TABLE table\_name;

**Section – 4 -> Inserting Data**

We can insert data inside the tables via the INSERT statement.

Syntax:

INSERT INTO table\_name(column1\_name, column2\_name)

VALUE (value1, value2)

Here one thing to keep in mind is the fact that value1 must belong to column1\_name, value2 must belong to column2\_name and the data types of the values must match with columns specified data types otherwise SQL will raise errors.

All in all, the order matters, as in if we have written column1\_name first in the parenthesis then value1 must belong to column1\_name.

Also, while creating a table, we specify the structure of the table via columns but that doesn’t mean while inserting the values we have to adhere to that structure, we can move around the column names as long as the values also get moved around to accommodate the change. And, this won’t change the original structure of the table.

**How To Know If the Values That We Inserted Were Saved Properly in The Table?**

To get the answer of above question we can use the SELECT statement to retrieve the values if they were indeed inserted into the table.

Syntax:

SELECT column1\_name, column2\_name

FROM table\_name;

We can also use \* in place of listing all the column names to retrieve all the columns.

Multiple Inserts syntax:

INSERT INTO table\_name(column1\_name, column2\_name)

VALUES (v10, v20),

(v11, v21),

(v12, v22),

…;

**Working with NOT NULL: -**

NULL means no value is permitted in a database unless we specify it not to be the case. If we permit NULL, then we can skip inserting values for a column while inserting new values. The column for whom we didn’t insert any value will have NULL in place of the expected value. If we don’t permit that by specifying it while creating a new table, MySQL will raise an error whenever we try to insert new values and try skipping inserting value for such columns.

NULL doesn’t mean 0 in fact 0 is itself a value.

We can introduce NULL value via the following queries:

INSERT INTO table\_name(columns1\_name, column2\_name)

VALUES (v1, v2), (v3, v4), (v5, v6), ();

OR

INSERT INTO table\_name

VALUES (v1, v2), (v3, v4), (v5, v6), ();

A new row will be added to the table that contains only NULL values.

We can introduce a constraint of NOT NULL while creating a new table or alter a table.

Ex: -

CREATE TABLE table\_name

(

column1\_name data\_type NOT NULL,

column2\_name data\_type NOT NULL,

…

);

**Quotes in SQL:**

Some versions of DBMS allow us to wrap our text in double quotes but it’s recommended to use single quotes.

To insert quote that are part of text we can use backslash like this \’, using backslash allows us to insert quotes inside a text.

We can insert double quotes inside a text that is being encased by a single quote without a problem.

**Adding Default Values:**

We can assign default values to our columns when we are creating a new table or altering an existing one just like when we specified NOT NULL constraint. Here if we have assigned a default value to a column along with a NOT NULL constraint, we can skip inserting a value in that column and still not get an error cause the default value will then be getting assigned instead of a NULL value unless we have also assigned the constraint UNIQUE to that column.

Syntax:

CREATE TABLE table\_name

(

column1\_name data\_type DEFAULT ‘default value’ NOT NULL,

column2\_name data\_type DEFAULT 21 NOT NULL,

…

);

Note – One thing I observed was that we don’t necessarily need to list the column names while inserting value as long as we are inserting value for every column. We need only do that when we want to insert values for some columns and exclude some columns, otherwise we can go ahead and insert value directly in the following way:

INSERT INTO table\_name

VALUES (v1, v2),

(v3, v4),

…;

Only possible if the Table table\_name has two columns and the values inserted are following the structured defined by the columns.

**Introducing Primary Key:**

Primary key in a table is a column that can be used to uniquely identify each row of the data. In simpler terms a primary key is a column that doesn’t have duplicate values or all the values stored are unique.

Again, we can appoint a column to be the primary key either while creating the table or while altering the table.

Also, there can only be one column within a table that can be assigned this role.

Syntax:

CREATE TABLE table\_name

(

id\_column data\_type NOT NULL PRIMARY KEY,

column1\_name data\_type,

…

);

Note – As seen above, the column id\_column is now a primary key in the table table\_name and so it is incapable of accepting duplicate value and MySQL will throw an error if an attempt to do so was made.

Another way of creating a primary key:

CREATE TABLE table\_name

(

id\_column data\_type NOT NULL,

column1\_name data\_type,

…

PIMARY KEY(id\_column) – parenthesis is mandatory

);

Another juicy detail to know is that the column that we are using as a PRIMARY KEY cannot have null values inside them in addition to no duplicate values so we do not have to explicitly specify the NOT NULL condition for that column.

**Working with AUTO\_INCREMENT:**

We can apply AUTO\_INCREMENT to a primary key if it has the INT data type along with other numeric types. Also, the number starts from 1 and goes on without us doing anything.

Syntax:

CREATE TABLE table\_name

(

id\_column INT AUTO\_INCREMENT,

column1\_name data\_type NOT NULL DEFAULT ‘value’,

column2\_name data\_type NOT NULL DEFAULT 1,

…,

PRIMARY KEY(id\_column)

);

**SECTION CHALLENGE:**

Define an Employees table in the Company database, with the following fields:

* id – number (automatically increments) and primary key
* last\_name – text, mandatory
* first\_name – text, mandatory
* middle\_name – text, not mandatory
* age – number, mandatory
* current\_status – text, mandatory, defaults to ‘employed’

**Solution Query:**

CREATE DATABASE Company;

USE Company;

CREATE TABLE Employees

(

id INT AUTO\_INCREMENT,

first\_name VARCHAR(25) NOT NULL,

middle\_name VARCHAR(25) DEFAULT ‘not any’,

last\_name VARCHAR(25) NOT NULL,

age INT NOT NULL,

current\_status VARCHAR(20) NOT NULL DEFAULT ‘employed’,

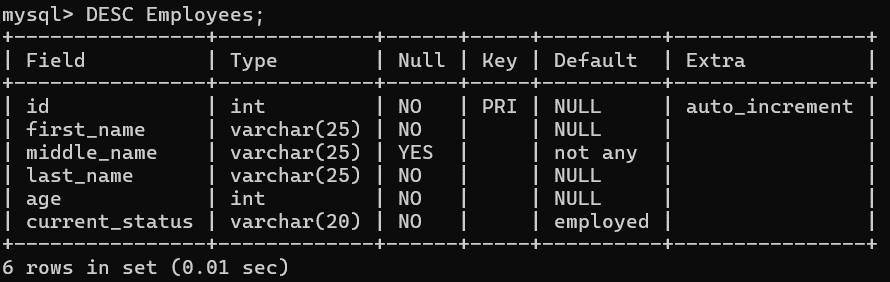
PRIMARY KEY(id)

);

DESC Employees;

DROP DATABASE Company;

Result of above query:



**Section – 5 -> CRUD Basic**

CRUD are the basic four actions that we want to be able to do to the data stored inside a table.

C – Create new rows via INSERT INTO table\_name(column names) VALUES(values in order);

R – Read the existing rows via SELECT column\_names FROM table\_name;

U – Update the existing rows via UPDATE table\_name SET column\_name = value WHERE some\_condition;

D – Delete the existing rows via DELETE FROM table\_name WHERE some\_condition;

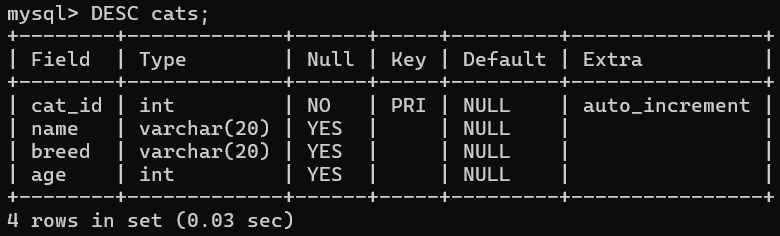
CRUD are basically us making changes to rows in a table.

**Creating a New Dataset to Work With:**

CREATE DATABASE Animals;

USE Animals;

SELECT DATABASE();



CREATE TABLE cats

(

cat\_id INT AUTO\_INCREMENT,

name VARCHAR(20),

breed VARCHAR(20),

age INT,

PRIMARY KEY (cat\_id)

);

DESC cats;

-- C

INSERT INTO cats(name, breed, age)

VALUES (‘Ringo’, ‘Tabby’, 4),

(‘Cindy’, ‘Maine Coon’, 10),

(‘Dumbledore’, ‘Maine Coon’, 11),

(‘Egg’, ‘Persian’, 4),

(‘Misty’, ‘Tabby’, 13),

(‘George Michael’, ‘Ragdoll’, 9),



(‘Jackson’, ‘Sphynx’, 7);



-- R

-- To get all the columns

SELECT \* FROM cats;

-- To get only the columns that we want

SELECT name, age FROM cats;



**The WHERE clause:**

WHERE is a keyword in SQL that allows the user to narrow down the number of rows according to some condition. Also WHERE is not just limited to working with SELECT, we can also use WHERE with UPDATE and DELETE.



Ex:

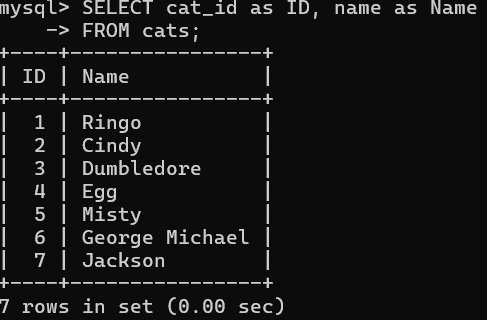
SELECT \* FROM cats

WHERE breed = ‘Tabby’

Interesting thing about WHERE is that we don’t necessarily need to add the column that we are using to check for condition into the SELECT statement.

**Aliases:**

When we are selecting columns to be displayed, we can change them into what we want to see when the table get printed out. Do note that the column’s name will not change in the database itself. To create an alias for a column we use AS keyword in the following way:



SELECT cat\_id as ID, name as Name

FROM cats;

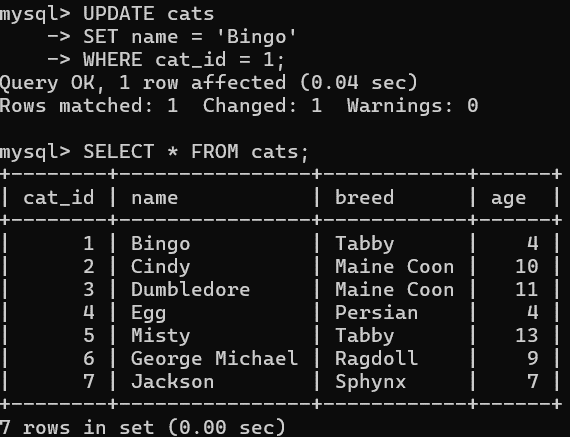
The use Aliases truly shine when we are doing complex operations

to create new fields on the fly and then we can use Aliases to

name the newly created field then and there.

-- U in CRUD is used for altering the existing data or rows in a table

Syntax:



UPDATE table\_name

SET column\_name = value

WHERE some\_condition;

So, as seen in the syntax, we update a table by setting a column with a

value and we specify which row will change using the WHERE statement.

EX: -

UPDATE cats

SET name = ‘Bingo’

WHERE cat\_id = 1

Note – If we don’t use the WHERE clause, all the rows in that column will have their values updated to the new value.

Updating multiple columns at the same time, Syntax: -

UPDATE table\_name

SET column1\_name = value1, column2\_name = value2

WHERE column\_name = some\_value;

**A Good Rule of 👍 (window +.)**

Before doing an UPDATE make sure to SELECT first. Same for the DELETE.

We can use SELECT to check whether our condition that we are using to update is actually reading the right rows or not.

-- D in CRUD is used to DELETE rows from the table

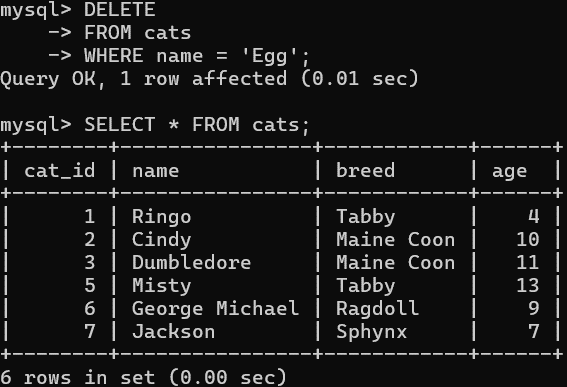
Syntax:

DELETE

FROM table\_name

WHERE some\_condition;

If the WHERE clause is not present, all the rows in the table will be deleted. It is not deleting the tables themselves but it will leave the table like it was when it was newly created using the CREATE statement.



Ex:

DELETE

FROM cats

WHERE name = ‘Egg’

SELECT \* FROM cats;

**Section – 6 -> CRUD Challenges**

**CRUD Challenge Creating:**

Instructions:

* Create a new database called shirts\_db
* Create a new table called shirts
* The new table shirts should have the look like this: -

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| shirt\_id | article | color | shirt\_size | last\_worn |
| 1 | t-shirt | white | S | 10 |
| 2 | t-shirt | green | S | 200 |
| 3 | polo shirt | black | M | 10 |
| 4 | tank top | blue | S | 50 |
| 5 | t-shirt | pink | S | 0 |
| 6 | polo shirt | red | M | 5 |
| 7 | tank top | white | S | 200 |
| 8 | tank top | blue | M | 15 |

Here shirt\_id is a PRIMARY KEY for the table and should AUTO\_INCREMENT.

* Insert the above rows in the newly created table shirts.
* Next add a new shirt that is a purple polo shirt, size M last worn 50 days ago.
* SELECT all shirts and print out the whole table
* SELECT all shirts (i.e., rows) but only show two columns article and color.
* SELECT all medium shirts (i.e., some rows) and print out all the columns except shirt\_id.
* UPDATE all polo shirts, change their size to L.
* UPDATE the shirt that was last worn 15 days ago, change last\_worn to 0.
* UPDATE all white shirts, change their size to ‘XS’ and color to ‘off white’.
* DELETE all old shirts i.e., last\_worn >= 200.
* DELETE all the tank tops.
* DELETE all shirts.
* DROP the entire shirts table.
* DROP the database.

QUERY:

CREATE DATABASE shirts\_db;

USE shirts\_db;

SELECT DATABASE();

CREATE TABLE shirts

(

shirt\_id INT AUTO\_INCREMENT,

article VARCHAR(20) NOT NULL DEFAULT 'Unidentified',

color VARCHAR(20) NOT NULL DEFAULT 'colorless',

shirt\_size VARCHAR(5) NOT NULL DEFAULT 'NA',

last\_worn INT,

PRIMARY KEY (shirt\_id)

);

DESC shirts;

-- C Operations

INSERT INTO shirts(article, color, shirt\_size, last\_worn)

VALUES

('t-shirt', 'white', 'S', 10),

('t-shirt', 'green', 'S', 200),

('polo shirt', 'black', 'M', 10),

('tank top', 'blue', 'S', 50),

('t-shirt', 'pink', 'S', 0),

('polo shirt', 'red', 'M', 5),

('tank top', 'white', 'S', 200),

('tank top', 'blue', 'M', 15);

INSERT INTO shirts(article, color, shirt\_size, last\_worn)

VALUE ('polo shirt', 'purple', 'M', 50);

-- R Operations

-- SELECT all shirts and print out the whole table

SELECT \*

FROM shirts;

-- SELECT all shirts (i.e., rows) but only show two columns article and color

SELECT article, color

FROM shirts;

-- SELECT all medium shirts (i.e., some rows) and print out all the columns except shirt\_id

SELECT article, color, shirt\_size, last\_worn

FROM shirts

WHERE shirt\_size = 'M';

-- U Operations

-- UPDATE all polo shirts, change their size to L

UPDATE shirts

SET shirt\_size = 'L'

WHERE article = 'polo shirt';

-- UPDATE the shirt that was last worn 15 days ago, change last\_worn to 0

UPDATE shirts

SET last\_worn = 0

WHERE last\_worn = 15;

-- UPDATE all white shirts, change their size to ‘XS’ and color to ‘off white’

UPDATE shirts

SET color = 'off white', shirt\_size = 'XS'

WHERE color = 'white';

-- D Operations

-- DELETE all old shirts i.e., last\_worn = 200

DELETE

FROM shirts

WHERE last\_worn = 200;

-- DELETE all the tank tops

DELETE

FROM shirts

WHERE article = 'tank top';

-- DELETE all shirts

DELETE

FROM shirts;

-- DROP the entire shirts table

DROP TABLE shirts;

-- DROP the database

DROP DATABASE shirts\_db;

**Section – 7 -> String Functions**

Now that we can do the basic operations via the CRUD commands, we can move on to levelling these commands up. Now for C in CRUD, there is not much else to do other than what we have already done, the C allows us to insert new rows in the table and we can choose to either insert one row or multiple rows for some columns or all columns but the basic syntax remains all the same.

For the other three commands in CRUD though we can be quite creative and write complex query depending on what we wish to achieve especially when we are selecting data while working with multiple tables.

String Functions – Built in operations that can be used or performed on columns that have textual data stored inside them.

We can run an SQL file while in the command line prompt. Just navigate to where the file is stored using cd commands and then open SQL when you are in the same folder where the file is being saved. After opening the SQL write the following command:

source file\_name.sql

Now, if the file is creating a table but we haven’t chosen any database to use, SQL will raise an error otherwise it will run and accomplish whatever its objective is.

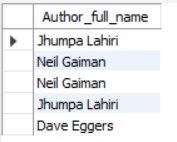
**STRING FUNCTIONS: -**

* CONCAT(str1, str2, …) OR CONCAT(column1\_name, column2\_name, …);

Returns the string that results from concatenating the arguments.

Returns NULL is any of the arguments is NULL.

Note – We always have parenthesis after the name of a function unlike commands.



|  |  |
| --- | --- |
| Example 1 | Example 2 |
| SELECT CONCAT('My', 'S', 'QL')    AS Output; | SELECT CONCAT(author\_fname, ' ', author\_lname)  AS Author\_full\_name  FROM books  LIMIT 5; |

If the alias command AS was not used, newly created column name would have been what follows the SELECT command i.e., CONCAT(author\_fname, ' ', author\_lname).

* CONCAT\_WS(separator, str1, str2, …)

Prety much the same as the CONCAT other than the fact that the first argument passed gets used to combine the text.

Ex. SELECT CONCAT\_WS(‘-’, ‘Deepak’, ‘Sheoran’) -> Deepak-Sheoran

* SUBSTRING(str, pos), SUBSTRING(str FROM pos), SUBSTRING(str, pos, len), SUBSTRING(str FROM pos FOR len)

SUBSTR(str, pos) is the same as SUBSTRING(str, pos)

Takes a single larger string and returns a smaller portion of that string. The returned value then gets stored inside the row of the newly created column.

str is the original string,

pos is the number that tells us how many letters to skip at the start, a negative pos is just telling us to pick up the last pos letters of the substring

len (optional) tells the length of the substring, if the newly cut substring exceeds that length, the letters trailing at the end gets cut.

Ex. SELECT SUBSTRING(‘substring’, -6, 3) -> str

* REVERSE(str)

This function takes whatever string we provide and return it after reversing it.

* REPLACE(str, from\_str, to\_str)

Allows us to replace portions of a string with some other replacement string and return the new value that then gets stored as row in a new column. The original data doesn’t get updated through the use of this function.

Returns NULL if any of the arguments are NULL.

If a single letter is passed as a from\_str, all the matching letters in the str will be replaced by to\_str.

It is case sensitive.

Ex. SELECT REPLACE(‘Deepak Sheoran’, ‘e’, ‘E’) -> DEEpak ShEoran

* CHAR\_LENGTH(str)

This function returns the length of the str passed as an argument. Returns the number of characters.

* LENGTH(str)

This function returns the size in bytes taken by the string that’s being passed as an argument.

* UPPER(str) or UCASE(str) and LOWER(str) or LCASE(str) function

These two functions change the casings of the string that’s being passed as an argument.

* INSERT(str, pos, len, newstr)

This function is very similar to SUBSTRING but instead of taking out the string from the original string it replaces the removed substring with newstr.

EX – SELECT INSERT(‘Original String’, 10, 6, ‘Bring’) -> Original Bring

* LEFT(str, len) and RIGHT(str, len) functions

These two functions can be used to get the leftmost or rightmost substrings from the original string and the length of that returned string can be specified using len argument.

* REPEAT(str, count) function

Returns a string consisting of the string str repeated count times. If count is less than 1, returns an empty string instead. Returns NULL is any of the argument is NULL.

* TRIM([BOTH | LEADING | TRAILING] [remstr] FROM str) function

Returns the string str with all remstr prefixes or suffixes removed. If the specifier is not given BOTH is taken as a default. remstr is optional, if not given spaces are removed. remstr stands for remove string.

Ex – SELECT TRIM(LEADING ‘xx’ FROM ‘xxxyz’); -> xyz

Combined String Functions: -

Yes, we can indeed combine two or more functions to get an output in return by passing in a function as an argument inside another function.

Ex: -

SELECT CONCAT(SUBSTRING(title, 1, 10), ‘…’)

AS Short\_Title

FROM books;

**String Functions Exercise: -**

1. Reverse and Uppercase this sentence – “Why does my cat look at me with such hatred?”

SELECT REVERSE(UPPER(‘Why does my cat look at me with such hatred?’));

1. What does the following query print out: -

SELECT REPLACE(CONCAT(‘I’, ‘ ’, ‘like’, ‘ ’, ‘cats’)), ‘ ’, ‘\_’);

Solution – I\_like\_cats

1. Replace the spaces in the title column of the table books with ‘->’

SELECT REPLACE(title, ‘ ’, ‘->’);

1. Print out a table that has two columns called forwards and backwards, the data in columns forwards contain

author's last name and the data in column backwards contains the reversed author last name

SELECT author\_lname as forwards, REVERSE(author\_lname) as backwards

FROM books;

1. Create a column names 'full name in caps' whose rows contain the full name of the author in upper case

SELECT UPPER(CONCAT(author\_fname, ' ', author\_lname))

AS 'full name in caps'

FROM books;

1. Create a column called blurb that joins two existing columns and some new text in between, the first row

prints 'The Namesake was released in 2003' and so on

SELECT CONCAT(title, ' was released in ', released\_year)

AS blurb

FROM books;

1. Print the book title and length of each title in a separate column called character count

SELECT title, CHAR\_LENGTH(title) AS 'character count'

FROM books;

Last Query: -

SELECT

CONCAT(LEFT(title, 10), '...') AS 'short title',

CONCAT(author\_lname, ',', author\_fname) AS author,

CONCAT(stock\_quantity, ' in stock') AS quantity

FROM books;

**Section – 8 -> Refining Selections**

This section’s primary focus is on refining selection or in simpler terms using some important commands to further print out the data that meet our demand as close as possible.

We can do the following refinement in addition to the one we have already been doing: -

* Sort the selected data according to some order
* Limit the number of rows that can be printed out from the selected data
* Print out only distinct or unique data
* Group the data together according to some categorical column and then apply some aggregation function on it.

**Selecting Distinct Results**

Using the DISTINCT command, we can make sure to only get distinct values out of a column in a table. If any row of data has a duplicate in the same column, we can use DISTINCT to make sure we only get one copy of that value in the printed table.

Syntax: -



SELECT DISTINCT column\_name

FROM table\_name;

We know many people have the same last name and if we wanted to

only see unique last names in the output we can write the following query: -

SELECT DISTINCT author\_lname AS ‘Unique Last Names’

FROM books;

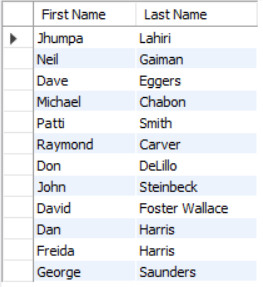
Applying DISTINCT on multiple columns

Syntax: -

SELECT DISTINCT column1\_name, column2\_name

FROM table\_name;

In response to the above query, only distinct rows for the two columns will be printed. If two rows for the two columns have duplicate value, only one value will be printed.



Ex: -

SELECT DISTINCT author\_fname AS ‘First Name’, author\_lname AS ‘Last Name’

FROM books;

All in all, it looks like using DISTINCT for even column will make it apply on all the columns if they were to be selected.

**Sorting The Results Using ORDER BY Command**

Often, we want our printed-out data to be shown in a sorted way either by alphabetically or by ascending or descending. The way we can sort the data that’s being printed is by using the command ORDER BY.

The unique thing about this command is that the command comes after the SELECT and that’s because the sorting will be done after the data has been selected since we are not sorting the original data but the data that’s being selected after going through commands like SELECT, DISTINCT, WHERE etc. We also specify which columns we want to use to sort the data.

ORDER BY does ASCENDING sort by default – For textual column it’s a to z and for numerical columns its lower number to higher number.

To change this, we can just specify the other option by writing it in front of the column name that’s being used for sorting.

Just like WHERE command, we don’t need to SELECT the column that’s being used for sorting the selected data.

Syntax: -

SELECT column1\_name, column2\_name

FROM table\_name

WHERE some\_condition

ORDER BY column\_name DESC/ASC;

Other than specifying the column name in front of the ORDER BY statement we can also use a number to tell which column to use among the selected column for sorting the selected data however it’s not recommended to use this process since it makes the code less readable for others.

Syntax: -

SELECT column1\_name, column2\_name

FROM table\_name

WHERE some\_condition

ORDER BY 2;

Here, it tells MySQL to use column2\_name for sorting the selected data.

Ordering By multiple columns- Useful when we are dealing with duplicate values in that column that was being used for ORDER BY.

Syntax

SELECT column1\_name, column2\_name

FROM table\_name

WHERE some\_condition

ORDER BY column1\_name ASC, column2\_name DESC;

Here what happens is that column1\_name will be used for primarily sorting the data but when we encounter duplicate values within the column1\_name, column2\_name will be used for sorting. All in all, duplicated rows of data become mini group of data where a secondary sorting happens according to the second column.

Another crucial point to remember is that we can even ORDER BY the selected data by using columns that are not even part of the original table but were created using functions or other operations.

Syntax: -

SELECT SOME\_FUNCTION(column1\_name, column2\_name) AS column\_name

FROM table\_name

ORDER BY column\_name;

**Limiting The Number of Rows Using LIMIT Command**

The command LIMIT allows us to limit the number of rows that gets printed.

The command LIMIT is written right at the end of query. On its own we write LIMIT followed by the number of rows we want to see on the screen.

Syntax:

SELECT \*

FROM table\_name

WHERE some\_condition

ORDER BY column

LIMIT number;

The above query gives the first number of rows according to the order and condition.

The use of LIMIT alongside ORDER BY let us print out data that meet a certain criterion like first 5 top scorer in an exam.

We can use two numbers like this LIMIT num1, num2 to show rows starting at a certain index and the second number lets us choose how many rows to pick starting at that index. Indexing starts at 0.

If we write down a number that exceeds the available rows, SQL will not raise an error it will just print out the available rows.

**LIKE Commands For Better Searching**

Using WHERE command we can check for conditions like equality (exact match) or logical conditions but using the LIKE command, we can check for conditions like looking for values that follow a certain pattern like most email will have the extension “@gmail.com”.

In simpler terms LIKE allow us to check for fuzzier conditions.

Syntax: -

SELECT column1\_name, column2\_name

FROM table\_name

WHERE column\_name LIKE a\_certain\_pattern

We can define the certain pattern using wildcards and the values that we remember to be part of the data that we are searching for.

Ex: -

WHERE author\_fname LIKE ‘%da%’

Wildcard characters: -

% -> any number of characters.

\_ -> exactly one character.

Escaping Wildcard

There might be times when we need to insert some wildcard character in the value since the data that we are searching for might have them, to do that we need to make sure SQL read the wildcard as part of the value not as a wildcard character. We can use backslash \ to escape a wildcard character i.e., ‘%da\%%’

Ex: -

SELECT title

FROM books

WHERE title LIKE '\_\_\%%';

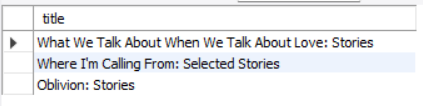
**Refining Selections Exercise: -**

1. Select all story collections- title(s) that contain the word 'stories' in them

SELECT title

FROM books

WHERE title LIKE '%Stories%';



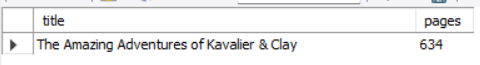
1. Find the longest book according to the page count- Print out the title and page count

SELECT title, pages

FROM books

ORDER BY pages DESC

LIMIT 1;



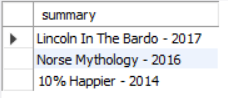
1. Print a summary containing the title and year, for the 3 most recent books

SELECT CONCAT(title, ' - ', released\_year) AS summary

FROM books

ORDER BY released\_year DESC

LIMIT 3;

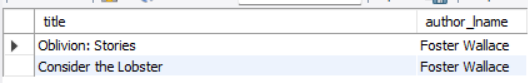


1. Find all the books with an author\_lname that contains a space (' ') - print out title and author\_lname for those that meet the condition

SELECT title, author\_lname

FROM books

WHERE author\_lname LIKE '% %';



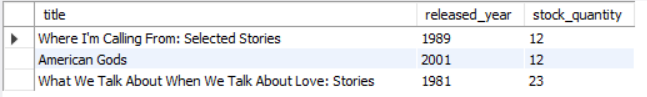
1. Find the 3 books with the lowest stocks - Print title, year and stock for the rows that meet the criterion

SELECT title, released\_year, stock\_quantity

FROM books

ORDER BY stock\_quantity

LIMIT 3;



1. Print title and author\_lname, sorted first by author\_lname and then by title

SELECT title, author\_lname

FROM books

ORDER BY author\_lname, title;

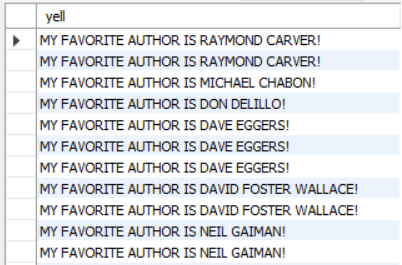


1. Create a new column with an alias 'yell' that is sorted alphabetically by last name and each rows says 'MY FAVORITE AUTHOR IS ' + UPPER(author\_fname) + ' ' + UPPER(author\_lname) + '!'

SELECT CONCAT('MY FAVORITE AUTHOR IS ', UPPER(author\_fname), ' ', UPPER(author\_lname), '!') AS yell

FROM books

ORDER BY author\_lname;



**Section – 9 -> Aggregate Functions**

Aggregate functions allow the user to run basic analysis on their data by letting them calculate measures like total sum, average, count, grouping up the data according to some common factor and finding measures for those common groups.

Aggregate functions are functions that can operate on multiple rows or multiple pieces of data to give us in return measures like max or min, average, sum, count etc.

**COUNT() Built-in Aggregate Function: -**

COUNT() function allow us to count number of rows that either follow a specific condition or just all the rows available in a table or column if there is no condition placed on the selected data.

Syntax: -

SELECT COUNT(column\_name)

FROM table\_name

WHERE condition;

All in all, what COUNT does is it counts the number of rows after the data has been selected after going through any of the conditions written in the query.

To get the count of all rows available in the table -> SELECT COUNT(\*) FROM table\_name;

Note – The COUNT Function doesn’t play nice with other columns because the COUNT function aggregates all the rows to a single value unlike other columns where each row is printed out separately depending on the conditions.

Difference between using \* vs a specific column name in the COUNT() function

When we are putting the \* in the COUNT function, we are asking it to return the number of rows available in the table but we put a column instead of \*, we are asking how many rows of data is present in that column and the biggest difference comes from the fact that NULL rows are still considered as rows while null values are not considered as data so the two will return different values if there are null rows available in the table.

We can also use COUNT function with DISTINCT to get only the unique values inside a column.

SELECT COUNT(DISTINCT column\_name)

FROM table\_name;

And of course, we can use count to find the number of rows that meet certain condition since count gets to work only after the data has been selected after going through any of the conditions applied on it.

**GROUP BY**

GROUP BY summarizes or aggregates identical data into single rows. The identical data is mostly recognized by using columns that have qualitative or categorical data.

What GROUP BY does is that it creates many smaller groups, where each group is created using the categorical data. Once these group have been made, we can use aggregate function on these individual groups to get measures.

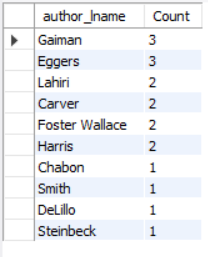
Syntax: -

SELECT column\_name, COUNT(column\_name)

FROM table\_name

GROUP BY column\_name;

Ex: -



SELECT author\_lname, COUNT(\*) AS Count

FROM books

GROUP BY author\_lname

ORDER BY Count DESC

LIMIT 10;

All in all, what GROUP BY does is, it breaks the table into many smaller tables according the column that we use to GROUP BY and then the aggregate function work on each smaller table separately instead of the whole table.

If we are using GROUP BY on a column then we can’t SELECT another column to get printed out because maybe two rows belong to one group and so for the column that’s being used for grouping the data, a single value will be enough to represent those two rows but for the other column that we are trying to select, those two rows might have two different values and so there will be an error.

Most of the time we won’t need to print another column because if use GROUP BY in the first place to apply aggregate function on the groups and boil down the multiple rows in a group to a single value. So, unless we are using aggregate function on another column and selecting that, it isn’t recommended to use column by themselves when we are group the rows.

**MIN and MAX aggregate functions**

The MIN and MAX allow us to find the minimum and maximum values inside a column or a group of rows in a column.

Syntax without GROUP BY: -

SELECT MIN/MAX(column\_name)

FROM table\_name;

Syntax with GROUP BY: -

SELECT column\_name, MIN/MAX(column1\_name)

FROM table\_name

GROUP BY column\_name;

MIN and MAX return a single value just like COUNT function.

We can use MIN/MAX function with the WHERE command to test for conditions and answer questions like name the title of the book with the highest number of pages. We can do that using subquery.

Trying to do that without the subquery:

SELECT title

FROM books

ORDER BY pages DESC

LIMIT 1;

**Subqueries**

Subquery is a query within a query and we surround the subquery within parenthesis. It’s important to surround the subquery within parenthesis so that MySQL will be able to identify it.

Syntax: -

SELECT column1\_name, column2\_name

FROM table\_name

WHERE column\_name = (SELECT MAX(column\_name) FROM table\_name);

The above query could return more than a single row if there are multiple rows with same max column\_name value.

Subquery run first and returns the value that then gets used to check for the condition.

**Grouping By multiple columns**

The only times when we might want to group by multiple columns and create subgroups of data is when within the first group there are still many duplicates for many values and we want the analysis on these values so we can write the name of this column after the first column.

Syntax: -

SELECT column1\_name, column2\_name, AGG\_FUN(column\_name) -- AGG\_FUN is any of the aggregate function

FROM table\_name

GROUP BY column1\_name, column2\_name;

Here the first group will be created using column1\_name and then within the groups created, subgroups will be created using column2\_name.

**MIN() and MAX() functions with GROUP BY**

We can use MIN and MAX in conjuncture with GROUP BY to find the MAX and MIN values with groups or even subgroups depending on how many columns are being used for grouping.

Syntax: -

SELECT column1\_name, column2\_name, MAX/MIN(column\_name)

FROM table\_name

GROUP BY column2\_name, column1\_name;

Q. Find the year each author published their first book

SELECT author\_fname, author\_lname, COUNT(title) AS 'Number of Books Published', MIN(released\_year) AS 'Earliest Publication', MAX(released\_year) AS 'Latest Publication'

FROM books

GROUP BY author\_lname, author\_fname;

In the above query, if an author has published multiple books in many different years, there will be multiple rows representing those publications and due to us grouping by rows according to author’s names, all these rows will now have become part of a subgroup and so next the aggregate function MIN will find the smallest year according to its values and return that year.

**SUM() function**

The SUM aggregate function sums all the values together that are present in a group or subgroup if we are grouping rows of data and if not then it will sum up all the values and return a single value just all the aggregate functions that came before it.

Syntax for SUM without GROUP BY: -

SELECT SUM(column\_name)

FROM table\_name;

Syntax for SUM with GROUP BY: -

SELECT column1\_name, SUM(column\_name)

FROM table\_name

GROUP BY column1\_name

Ex: -

SELECT CONCAT\_WS(' ', author\_fname, author\_lname) AS author, SUM(pages) AS 'Total Pages Written'

FROM books

GROUP BY author;

**AVG() function**

Same as all the previous aggregate functions.

Q. Calculate the average pages written and average book written per author

SELECT



CONCAT\_WS(' ', author\_fname, author\_lname) AS author,

COUNT(title) AS 'Total Books Written',

SUM(pages) AS 'Total Pages Written',

AVG(pages) AS 'Average Pages Written'

FROM books

GROUP BY author;

**Aggregate Functions Exercises**

USE book\_shop;

SELECT DATABASE();

SHOW TABLES;

DESC books;

-- OR

SHOW COLUMNS

FROM books;

1. Print the number of books in the database

SELECT COUNT(title) AS 'Total Number of Books'

FROM books;

2. Print out how many books were released in each year

SELECT released\_year AS Year, COUNT(title) AS 'Number of Books Released'

FROM books

GROUP BY Year

ORDER BY Year;

3. Print out the total number of books in stock

SELECT SUM(stock\_quantity) AS 'Total Number of Book in Stock'

FROM books;

4. Find the average released year for each author

SELECT CONCAT\_WS(' ', author\_fname, author\_lname) as author, AVG(released\_year)

FROM books

GROUP BY author;

5. Find the full name of the author who wrote the longest book

SELECT CONCAT\_WS(' ', author\_fname, author\_lname) as author, pages

FROM books

WHERE pages = (SELECT MAX(pages) FROM books);

6. Find the number of books published per year and the average number of pages per year

SELECT released\_year AS year, COUNT(title) AS '# books', AVG(pages) AS 'avg pages'

FROM books

GROUP BY year

ORDER BY year;

**Deleting Rows that have NULL values in them**

Syntax: -

DELETE

FROM table\_name

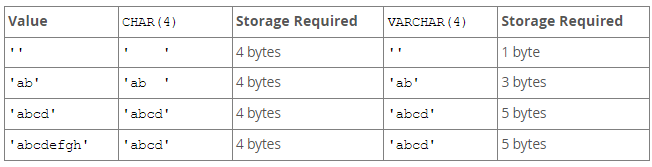
WHERE column\_name IS NULL;

**Section – 10 -> Revisiting Data Types**

**Data types for Storing Text:**

CHAR and VARCHAR, what’s the difference between the two?

Both CHAR and VARCHAR allow us to specify the maximum CHAR\_LENGTH but VARCHAR is optimized to be able to store text of different sizes or length in the best possible way while CHAR will have a fixed size even if we insert text that needs less size to be stored in the memory. When an entry that has a smaller number of expected characters, the right side of that entry will be filled with spaces until its size matches the length specified by CHAR and then the string will be stored in memory.



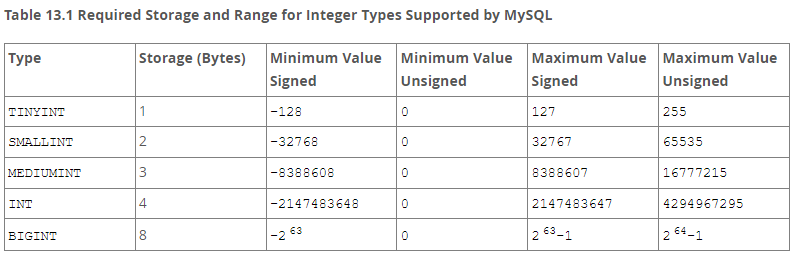
When to use which

Use VARCHAR to store text that varies in length for each entry like names of people, city etc.

USE CHAR when we are confident that each entry will have the same length or size. CHAR is faster for fixed length text. Also, if we are explicitly adding trailing spaces in our entry that’s being stored in a column with CHAR, even those spaces will be removed upon retrieval.

**Data Types for Storing Integers:**

Available data types for storing integers- TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT.



If we wish to only insert positive values inside the column then we can use UNSIGNED like this: -

CREATE TABLE table\_name(column\_name TINYINT UNSIGNED);

Now, the column will not accept negative values but that in turn allows us to extend the range of allowed value for the different data types.

**Data Types for Storing Decimals:**

If we insert decimals into columns that have the data type integers then the number will be stored as an integer and the fractional part will be rounded off.

Available data types for storing decimals: -

* Fixed-Point Types (Exact Value) - DECIMAL(precision, scale) Or NUMERIC

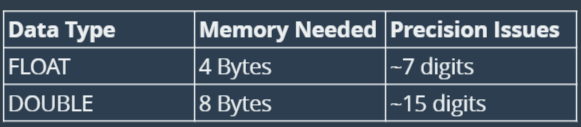
Precision – Represents the number of significant digits that are stored for values i.e., total number of digits.

Scale – Represents the number of digits that can be stored following the decimal point. If 0, DECIMAL values contain no decimal point or fractional part i.e., digits after decimal.

DECIMAL allows us to store precise decimals.

Number of digits allowed before the decimal point = Precision – Scale. Also, if we enter digits that exceed the scale, the excess part will be truncated and the value that will be stored will be less precise.

* Floating-Point Types (Approximate Value) –



The table indicate that we can store up to 7 digits after the decimal point using FLOAT without losing precision while using DOUBLE, we can store almost double that.

CREATE TABLE test

(

Value\_Entered\_Float VARCHAR(30),

Value\_Stored\_Float FLOAT,

Value\_Entered\_Double VARCHAR(30),

Value\_Stored\_Double DOUBLE

);

INSERT INTO test

VALUES

('1.123', 1.123, '1.123', 1.123),

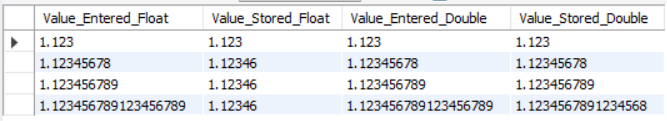
('1.12345678', 1.12345678, '1.12345678', 1.12345678),

('1.123456789', 1.123456789, '1.123456789', 1.123456789),

('1.123456789123456789', 1.123456789123456789, '1.123456789123456789', 1.123456789123456789);

SELECT \*

FROM test;



DECIMAL can be more exact but it takes up more space or it can take up more space while if we use FLOAT or DOUBLE, we can store larger number of decimal points using less space but that comes at the cost of precision.

**Data Types for Storing Dates and Times:**

* DATE – For storing values with Date but no time. Format – ‘YYYY-MM-DD’.
* TIME – For storing values with a Time but no date. Format – ’HH:MM:SS’.
* DATETIME – For storing values with a Date and Time. Format – ‘YYYY-MM-DD HH:MM:SS’.

We also get some functions and commands that are specifically designed for working with Dates and Times.

The functions and commands that can be used to fetch the current date and time dynamically: -

* CURDATE() [OR CURRENT\_DATE(), CURRENT\_DATE] – Returns the current date.
* CURTIME() [OR CURRENT\_TIME(), CURRENT\_TIME] – Returns the current time.
* NOW() [OR CURRENT\_TIMESTAMP(), CURRENT\_TIMESTAMP] – Returns the current date and time.

CREATE TABLE test

(

Id TINYINT AUTO\_INCREMENT,

Present\_Date DATE,

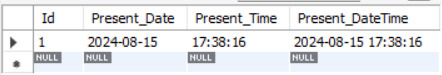
Present\_Time TIME,

Present\_DateTime DATETIME,

PRIMARY KEY(Id)

);

INSERT INTO test(Present\_Date, Present\_Time, Present\_DateTime)



VALUE (CURDATE(), CURTIME(), NOW());

**Functions for Formatting Dates and Extracting Portions of a Date or Time into a Different Format**

**Date Functions:**

* DATE(datetime) – Returns the date.
* DAY(date) or DAYOFMONTH(date) – Returns the day of the month for date, in the range 1 to 31, or 0 for dates such as ‘0000-00-00’ and NULL for NULL.
* DAYOFWEEK(date) – Returns the weekday index for date (1 =Sunday, 2 = Monday, …, 7 = Saturday)
* DAYOFYEAR(date) – Returns the day of the year for date, in the range 1 to 366.
* DAYNAME(date) – Returns the name of the weekday for date. Returns NULL is date is NULL.
* MONTHNAME(date) – Returns the name of the month.
* WEEKOFYEAR(date) - Returns the calendar week of the date as a number in the range from 1 to 53. Returns NULL if ***date*** is NULL.
* MONTH(date) – Returns the month for the date.
* YEAR(date) – Returns the year for the date.

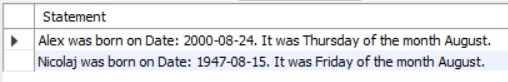
**Time Functions:**

* TIME(datetime) – Returns the time.
* SECOND(time) – Returns the second (0:59).
* MINUTE(time) – Returns the minute from the time.
* HOUR(time) – Returns the hour.

Ex: -

SELECT CONCAT(Name, ' was born on the Date: ', BirthDate, '. It was ', DAYNAME(BirthDate), ' of the month ', MONTHNAME(BirthDate), '.') AS Statement

FROM people;



**Formatting Dates and Times Using Function DATE\_FORMAT(date, format) and TIME\_FORMAT(time, format)**

Formats the ***date*** value according to the ***format*** string. If either argument is NULL, the function returns NULL.

The format in the arguments lets us format the date in any specific way by using specifiers.

Note – We can use DATE\_FORMAT function to format time also but we can’t format dates using TIME\_FORMAT.

Specifiers for Dates: -

1. %a – For abbreviated weekday name (Sun, Mon, …, Sat).
2. %W – For weekday name (Sunday, Monday, …, Saturday).
3. %w – For weekday in numeric (0 = Sunday, 1 = Monday, …, 6 = Saturday).
4. %b - For abbreviated month name (Jan, Feb, …, Dec).
5. %M – For month names (January, February, …, December).
6. %m – For month in numeric (00, 01, 02, …, 12).
7. %Y – Year, numeric, four digits.
8. %y – Year, numeric, two digits.
9. %% - %.
10. %c – Month numeric (0, …, 12).
11. %D – Day of the month with English suffix (0th, 1st, 2nd, …).
12. %d – Day of the month, numeric (00, 01, 02, …, 31).
13. %e – Day of the month, numeric (0, 1, 2, …, 31).

Specifiers for Times: -

1. %p – AM or PM.
2. %r – Time, 12 hours (hh:mm:ss followed by AM or PM).
3. %S or %s – Seconds (00, …, 59).
4. %T – Time, 24-hour (hh:mm:ss).
5. %k – Hour (0, …, 23).
6. %l – Hour (1, …, 12).
7. %i – Minutes (00, …, 59).

**Built in Date Math Functions:**

Using Date Math, we can calculate the time difference between two dates or times, add some specific time to a date and time, check how many days/months/weeks/years are between two dates.

Date Math Functions:

* DATEDIFF(expr1, expr2) – Returns expr1 – expr2 expressed as a value in days from one date to another. expr1 and expr2 are date or datetime expressions. Only the date part of the expressions is used in calculation.
* DATE\_ADD(date, INTERVAL expr unit), DATE\_SUB(date, INTERVAL expr unit) – These two functions allow us to take some date and do math with some interval (anything can be the interval like year, day, seconds, minutes etc.)

Ex – DATE\_ADD(NOW(), INTERVAL 1 YEAR)

* TIMEDIFF(expr1, expr2) – Returns expr1 – expr2 expressed as a time value.

We can also use operators like +, - to work with dates and time like -> SELECT NOW() – INTERVAL 18 YEAR;

TIMESTAMP – Date type for storing date and time just like DATETIME but in less storage and within a smaller range as compared to DATETIME.

The NOW() function returns current DATETIME whose datatype is TIMESTAMP.

**DEFAULT & ON UPDATE TIMESTAMPS**

Using DEFAULT constraint while creating a table we can make it so that each time a new row is added, the time at which the row was created gets recorded as well.

We can also use the ON UPDATE to record the date/time/datetime whenever someone updates any column in an existing row.

Syntax

CREATE TABLE table\_name

(

column\_id datatype AUTO\_INCREMENT,

column\_datetime TIMESTAMP/DATETIME/DATE/TIME DEFAULT NOW()/CURRENT\_TIMESTAMP/CURDATE()/…,

column\_datetime\_update TIMESTAMP ON UPDATE NOW() DEFAULT NOW()

);

Exercise: -

CREATE DATABASE Section\_10\_Exercises;

USE Section\_10\_Exercises;

-- 1. What is a good use case for CHAR?

-- -> CHAR datatype can be useful in situations when we wish to store textual data whose length will remain the same for every entry.

-- 2. Write the appropriate data types for the columns

CREATE TABLE inventory

(

item\_id SMALLINT UNSIGNED AUTO\_INCREMENT,

item\_name VARCHAR(30),

price DECIMAL(9,2),

quantity TINYINT UNSIGNED,

PRIMARY KEY(item\_id)

);

-- 3. What's the difference between DATETIME and TIMESTAMP

-- -> Both of the datatypes store information related to dates and times in the format -> 'YYYY-MM-DD HH:MM:SS',

but TIMESTAMP takes less storage space to store the same data but can store date and times within a much shorter range as compared to DATETIME.

-- 4. Write a SELECT statement to print out the current time

SELECT CURTIME() AS 'Current Time';

-- 5. Write a SELECT statement to print out the current date

SELECT CURDATE() AS 'Current Date';

-- 6. Write a SELECT statement to print out the current Day of the Week (number)

SELECT DAYOFWEEK(CURDATE()) AS 'Current Day of the Week in Number';

-- 7. Write a SELECT statement to print out the current Day of the Week (name)

SELECT DAYNAME(CURDATE()) AS 'Current Day of the Week';

-- 8. Write a SELECT statement to print out the current Date in this format -> mm/dd/yyyy

SELECT DATE\_FORMAT(CURDATE(), '%m/%e/%Y') AS 'Today\'s Date';

-- 9. Write a SELECT statement to print out the current Date and Time in this format -> January 2nd at 3:15 AM

SELECT DATE\_FORMAT(NOW(), '%M %D at %l:%i %p') AS 'Information About Current Date and Time';

-- 10. Create a table names tweets that stores 1. The tweet content 2. A Username 3. Time it was created 4. Time it was updated

CREATE TABLE tweets

(

tweet\_content VARCHAR(200),

username VARCHAR(20),

created\_at TIMESTAMP DEFAULT NOW(),

updated\_at TIMESTAMP ON UPDATE NOW() DEFAULT NOW()

);

DESC tweets;

SHOW TABLES;

DROP DATABASE Section\_10\_Exercises;

**Section – 11 -> Comparison and Logical Operators**

Section Intro – This section will contain information about comparison and logical operator. The comparison operator, mainly >, <, >=, <=, =, != are used to check for condition while the logical operators, mainly AND, OR are used to combine two or more conditions and that allows us to check for even more complex conditions and make more complex selections.

The usage of these conditional operators allows us to filter data like select all the rows where before the cell in a column has a value before (left side of the spectrum) a certain value in comparison or has a value after (right side of the spectrum) a certain value in comparison or even in between two values.

Notes: –

* In SQL, conditional calculation like 20 > 10 doesn’t yield Boolean value, instead they either print 0 (for false) or 1 (True). Whenever NULL is a part of conditional calculations, we get NULL in return instead of 0 or 1.
* Column aliases can be used with GROUP BY and ORDER BY clauses. We cannot use a column alias with WHERE and HAVING clauses.

1. Not Equal Conditional Operator -> ‘!=’

For filtering and selecting data that doesn’t match some value.

Syntax: -

SELECT some\_columns

FROM table\_name

WHERE column != some\_value;

1. NOT LIKE Conditional Operator -> ‘NOT LIKE’

For filtering and selecting data that is not like or doesn’t contain a specific sequence of value.

Syntax: -

SELECT some-columns

FROM table\_name

WHERE some\_column NOT LIKE ‘%\_%’;

1. LOGICAL AND operator. Used to combine two or more conditional operator and returns 1 in case all the individual condition return 1 else returns 0.

Syntax: -

SELECT some\_columns

FROM table\_name

WHERE some\_condition AND some\_other\_condition AND …;

1. LOGICAL OR operator. Used to combine two or more conditional operator and returns 1 in case any of the individual condition return 1 else returns 0 when all the condition return 0.

Syntax: -

SELECT some\_columns

FROM table\_name

WHERE some\_condition OR some\_other\_condition OR …;

1. BETWEEN Conditional Operator

For filtering or selecting data between a lower threshold and upper threshold. The upper and lower thresholds are inclusive.

Syntax: -

SELECT some\_columns

FROM table\_name

WHERE column BETWEEN lower\_value AND higher\_value;

1. NOT BETWEEN Conditional Operator

Instead of getting the values between the lower and upper threshold we get the value on the left side of the lower threshold and right side of the upper threshold. The upper and lower thresholds are exclusive.

[-♾️, lower threshold) U (upper threshold, ♾️]

SELECT some\_columns

FROM table\_name

WHERE column NOT BETWEEN lower\_value AND higher\_value;

**Cast Function for changing the data types of a value:**

CAST(expr AS type)

SELECT CAST(‘9:00:00’ AS DATE)

**Comparing Dates and Times:**

The usual comparison conditional operators can be used for comparing dates with each other without encountering any major issues but we should cast the values into dates and times before doing the comparison so to avoid any possible error.

All in all, the usual comparison and logical operator can be used to check for condition with dates and times as long as we are using the values of the same type to compare them with or the dates and times are written in the right format even if they are in string form.

We can also use dates and times function to extract numbers like DAYOFMONTH and compare this with a number to check for a condition related to only to days and so on.

1. IN Comparison Operator

The IN operator is quite helpful in cases when we want to check for multiple conditions like checking to see if our columns have a value that match among many possible choices. Without the IN operator, we would have to use multiple OR conditions but using IN we can

This operator will mostly be helpful for data types that have textual information.

Syntax:

SELECT some\_columns

FROM table\_name

WHERE column\_name IN (value1, value2, …);

1. NOT IN Comparison Operator

The opposite of the IN comparison operator, it will select all the values that don’t match with values listed in the parenthesis after the IN operator.

Syntax:

SELECT some\_columns

FROM table\_name

WHERE column\_name NOT IN (value1, value2, …);

1. MODULO (%) Operator or The Remainder Operator

Returns the remainder after the operation.

Syntax:

SELECT some\_columns

FROM table\_name

WHERE column\_name % some\_value = some\_value;

1. CASE statements

CASE statements are used within the SELECT statements to create a new column and assign values inside that column depending on conditions that we can specify not using the WHERE clause since that’s outside the SELECT statement but the WHEN clause that’s within the CASE statement.

To use CASE, we need to always use the END statement to let the DBMS know its end.

Syntax

SELECT some\_columns,

CASE

WHEN some\_condition THEN some\_value

WHEN some\_other\_condition THEN some\_other\_value

ELSE some\_other\_value

END AS alias

FROM table\_name;

1. IS NULL operator

To check for NULL values, we should use IS NULL statement since checking for NULL values using the equality operator doesn’t work.

Syntax:

SELECT some\_columns

FROM table\_name

WHERE column IS NULL; [or column IS NOT NULL;]

**Exercises:**

-- Evaluate the following

SELECT 10 != 10; -- 0

SELECT 15 > 14 AND 99 - 5 <= 94; -- 1

SELECT 1 IN (5, 3) OR 9 BETWEEN 8 AND 10; -- 1

-- Select all books written before 1980 (non inclusive)

USE book\_shop;

SELECT title, released\_year

FROM books

WHERE released\_year < 1980

ORDER BY released\_year DESC;

-- Select all books written by Eggers or Chabon

SELECT CONCAT\_WS(' ', author\_fname, author\_lname) AS Author, title, released\_year

FROM books

WHERE author\_lname = 'Eggers' OR author\_lname = 'Chabon';

-- Select all the books written by Lahiri, published after 2000

SELECT CONCAT\_WS(' ', author\_fname, author\_lname) AS Author, title, released\_year

FROM books

WHERE author\_lname = 'Lahiri' AND released\_year > 2000;

-- Select all books with page count between 100 and 200

SELECT title, pages

FROM books

WHERE pages BETWEEN 100 AND 200

ORDER BY pages;

-- Select all books where author\_lname starts with a C or an S

SELECT CONCAT\_WS(' ', author\_fname, author\_lname) AS Author, title, released\_year

FROM books

WHERE author\_lname LIKE 'C%' OR author\_lname LIKE 'S%';

-- Match the output

SELECT title, author\_lname,

CASE

WHEN title LIKE '%stories%' THEN 'Short Stories'

WHEN title = 'Just Kids' OR title = 'A Heartbreaking Work of Staggering Genius' THEN 'Memoir'

ELSE 'Novel'

END AS TYPE

FROM books;

-- Make this happen

SELECT author\_fname, author\_lname,

CASE

WHEN COUNT(title) = 1 THEN '1 book'

ELSE CONCAT(COUNT(title), ' books')

END AS COUNT

FROM books

GROUP BY author\_fname, author\_lname

ORDER BY count(title) DESC;

**Section – 12 -> Constraints & ALTER TABLE**

**Constraints-** Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

Constraints in general are used to validate the entry of a row in a table, if any of the conditions put forth by the constraints are not met, the entry of that row is not accepted.

**UNIQUE Constraint:**

In addition to constraint that have so far been introduced like DEFAULT, NOT NULL, PRIMARY KEY we now have the UNIQUE constraint that can used to make sure every new entry in a column is unique or in other words isn’t a duplicate.

We can apply this constraint on columns while creating a table or by altering an existing table.

Syntax:

CREATE TABLE table\_name

(

Id\_column datatype AUTO\_INCREMENT,

column1 datatype NOT NULL UNIQUE,

column2 datatype DEFAULT value,

PRIMARY KEY(Id\_column)

);

Ex:

CREATE TABLE companies

(

supplier\_id INT UNSIGNED AUTO\_INCREMENT,

name VARCHAR(255) NOT NULL,

phone VARCHAR(15) NOT NULL UNIQUE,

address VARCHAR(255) NOT NULL,

PRIMARY KEY (supplier\_id)

);

**CHECK Constraint:**

CHECK constraint is like an extra fancy constraint that we can define, basically custom constraint on a given column, or columns. Using CHECK, we can do something like applying a condition on a column where it can only accept values within a certain range or values greater than a certain value or smaller than a certain value and so on, or even use LIKE operator to even apply accepting conditions on textual values.

Syntax:

CREATE TABLE table\_name

(

column datatype CHECK (column condition)

);

Ex:

CREATE TABLE partiers

(

name VARCHAR(50),

age INT CHECK (age > 18),

palindrome VARHCAR(100) CHECK (REVERSE(palindrome) = palindrome)

);

**Naming the Constraints:**

Whenever we are entering rows in a table, we might be shown error that tells us that a certain cell in the rows has violated a constraint and it would be very helpful if we knew a way to recognize the constraint that stopped the insertion of the data and so to do that, we can name the constraints that will later be helpful in managing the database.

It is best practice to name a constraint in a way that it easily describes what kind of constraint stopped the insertion of the row.

To name a constraint we follow the following Syntax:

…,

column\_name datatype,

CONSTRAINT name\_of\_the\_constraint type\_of\_constraint,

…

And that’s it.

Ex:

…,

age INT,

CONSTRAINT age\_over\_18 CHECK (age > 18)

)

**Multi Column Constraints:**

We can set up constraints that use multiple columns like normally UNIQUE constraint when applied on a column doesn’t allow for duplicate values to occur in a column but we use multiple columns in a UNIQUE constraint we can allow for duplicate within a column as long as the multiple columns that we used in the UNIQUE constraint doesn’t have a duplicate row.

Ex:

CONSTRAINT name\_address UNIQUE (name, address),

CONSTRAINT sprice\_gt\_pprice CHECK (sale\_price >= purchase\_price),

**ALTER TABLE: Adding Columns**

The ALTER TABLE table\_name allows us to fundamentally mess around with columns that is to say to do everything that we can do with columns like adding new columns, dropping the existing columns, modifying the existing columns (adding/removing/modifying constraints, datatypes etc, renaming the columns) etc.

Adding a new column -> Syntax:

ALTER TABLE table\_name

ADD COLUMN column\_name datatype constraints; -- COLUMN is optional

If our table already have some rows when we are adding a new column and we put the constraint of NOT NULL, for string-based data types the column will be filled with empty string ‘’ and for numerical based data types column will be filled with 0 unless we use the DEFAULT statement to specify a default value.

**ALTER TABLE: Dropping Columns**

Syntax:

ALTER TABLE table\_name

DROP COLUMN column\_name; -- COLUMN is optional

**ALTER TABLE: Renaming**

Renaming Tables

One way

RENAME TABLE table\_name TO new\_table\_name;

Another way

ALTER TABLE table\_name

RENAME TO|AS new\_table\_name;

Renaming Columns

Syntax:

ALTER TABLE table\_name

RENAME COLUMN column\_name to new\_column\_name; -- COLUMN is required

**ALTER TABLE: Modifying Columns**

Using ALTER TABLE command, we can modify a column as in change its data type and the constraints placed on it.

Syntax:

ALTER TABLE table\_name

MODIFY COLUMN column\_name new\_datatype new\_constraints;

Ex:

ALTER TABLE companies

MODIFY COLUMN name VARCHAR(100) DEFAULT ‘Unknown’;

When we wish to change the column name and change its definition:

Syntax

ALTER TABLE table\_name

CHANGE column\_name new\_column\_name new\_datatype new\_constraints;

**ALTER TABLE: Constraints**

Adding constraints to columns

Syntax:

ALTER TABLE table\_name

ADD CONSTRAINT constraint\_name constraint\_type (column\_name condition\_if\_neccessary);

Example:

ALTER TABLE houses

ADD CONSTRAINT positive\_purchase\_price CHECK (purchase\_price >= 0);

Dropping constraints from columns

Syntax:

ALTER TABLE table\_name

DROP {CHECK|CONSTRAINT} symbol

Example:

ALTER TABLE houses

DROP CONSTRAINT positive\_purchase\_price

**Section – 13 -> One to Many Joins**

This section will make us work with tables that are related to each and provide the knowledge that will be crucial for working with multiple tables. Some of the important topics that this section introduces us with are: -

* Primary Keys
* Foreign Keys
* Joins
* One to many
* Many to many

So far, we have only worked with one table but in real world scenarios we have work with work with multiple tables that are related to each other through columns called keys. We can join different tables using these keys to get the data that we need to solve a particular problem and in the end that’s all matters, depending on the question we are trying to answer we have to join different tables stored inside a database and then further use functions and commands to get the desired output.

**Relationships Basics:**

The different ways a data can be related. A table can have different types of relationship with other tables and they are: -

* One to One Relationship (not that common),
* One to Many Relationships,
* Many to One Relationships,
* Many to Many Relationships and
* Self-Referencing Relationships.

One to One relationship (1: 1) - When we know for sure that one row in one table is related to only one corresponding row in the other table. So, in this relationship, each row in one table is linked to one and only one row in another table, and vice versa.

Ex – Customer (contains only the basic information about a customer) and Customer\_Details (contains detailed information about the user).

One to Many Relationship (1: N) - In this relationship, a row in one table can be associated with multiple rows in another table, but each row in the second table can be linked back to only one row in the first table.

Ex - A customers table and an orders table where each customer can have multiple orders, but each order is linked to only one customer.

Many-to-One (N: 1) Relationship - This is essentially the reverse of a one-to-many relationship, where many rows in one table correspond to one row in another table.

Ex - Multiple employees (in an employee’s table) reporting to one department (in a departments table).

**Many-to-Many (N: N) Relationship -** In this relationship, rows in one table can be associated with multiple rows in another table, and vice versa. This usually requires a third, "junction" table to break the relationship into two one-to-many relationships.

Ex- A students table and a courses table where each student can enroll in multiple courses, and each course can have multiple students. A junction table like student\_courses would hold the student IDs and course IDs to link them.

**Self-Referencing Relationship -** A table that has a relationship with itself. This is used for hierarchical data.

Ex - An employees table where each employee might report to another employee, creating a manager-employee hierarchy within the same table.

**Keys Used in Relationships**

* **Primary Key (PK):** A unique identifier for each record in a table. It ensures that each row can be uniquely identified.
* **Foreign Key (FK):** A field in one table that uniquely identifies a row of another table or the same table. It creates the link between two tables. It also checks that the row values that’s being inserted into it already exists in the table that it’s connecting its own table with.

**Why It’s a Good Idea to Work with Related Data:**

When the data is spread across multiple tables we can avoid duplicate rows for some columns, NULL values for some columns, make it so that a table contains only unique records according to some key and so on.

If we want the whole data, we can always join the tables accordingly, all in all related databases give us a lot of freedom to manipulate the data.

**Working with Foreign Keys and Creating Tables that are related to each other:**

Let’s create two tables’ Customers and Order. They have to One-to-Many relationship between them since one customer can have multiple orders but an order can be from only one customer. If we create a constraint called foreign key in the table then it won’t allow row entries whose entry don’t exist in the other table’s column or primary key that it is referencing.

SHOW DATABASES;

CREATE DATABASE RDB;

USE RDB;

SELECT DATABASE();

CREATE TABLE Customers

(

Customer\_Id SMALLINT UNSIGNED AUTO\_INCREMENT PRIMARY KEY,

First\_Name VARCHAR(25) NOT NULL,

Last\_Name VARCHAR(25) NOT NULL,

Email\_Id VARCHAR(40) NOT NULL UNIQUE,

CONSTRAINT email\_format CHECK(Email\_Id LIKE '%@gmail.com') -- only allows email that have the extension

-- @gmail.com

);

CREATE TABLE Orders

(

Order\_Id SMALLINT UNSIGNED AUTO\_INCREMENT PRIMARY KEY,

Order\_Date DATE NOT NULL,

Amount DECIMAL(6,2) NOT NULL,

Customer\_Id SMALLINT UNSIGNED,

FOREIGN KEY (Customer\_Id) REFERENCES Customers(Customer\_Id) -- A new syntax in the mix to take a look at

);

Syntax for creating a FOREIGN KEY reference

FOREIGN KEY (column\_name) REFERENCE other\_table(column name in the other table)

Note – Columns with datatype DATE doesn’t allow default values using CURDATE() function that fetches the current date dynamically instead they only allow for static data value like ‘2020-12-24’.

**Joins:**

Many times, when working with relational database we would be asked question that only one table won’t be able to answer alone so we will need to access multiple rows at the same time to find the answer.

How to get customer and order data together so we can see when the customer order and the order amount.

We can use subqueries here but the better approach would be to use joins.

Types of Joins:

Cross Join (Dumb Join) – Matches every row of one table with every row of another table, we will get every combination possible so if one table had 3 rows and the other table had 2 rows, we will get a table that has 6 rows. Not very useful.

Syntax:

SELECT \*

FROM table1\_name, table2\_name;

Inner Join – Most common type of join used, select the rows from both the table and joins them with other if their primary and foreign key matches. If the two tables have a one-to-one relationship, then duplicate rows will not be created but if the tables have one to many relationships, the rows will be duplicated for the table that has the one in the one-to-many relationship.

It is not mandatory to only use primary and foreign keys for joining.

Syntax:

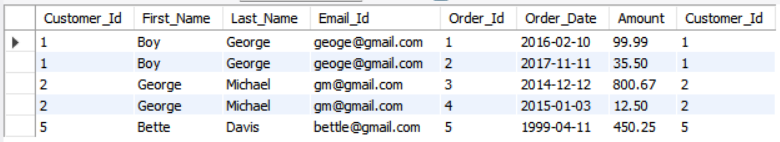
SELECT column\_names

FROM table1\_name AS A

JOIN table2\_name AS B

ON A.primary\_key = B.foreign\_key -- here the keys are columns

Ex:



SELECT \*

FROM Customers

JOIN Orders

ON Customers.Customer\_Id = Orders.Customer\_Id;

**Inner Join with Group By:**

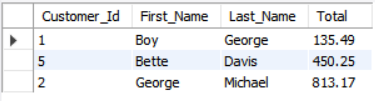
Same rule applies nothing new to consider but must take precaution against selecting columns that we are not using for grouping by.

Ex:

SELECT A.Customer\_Id, First\_Name, Last\_Name, SUM(Amount) AS Total

FROM Customers AS A

JOIN Orders AS B



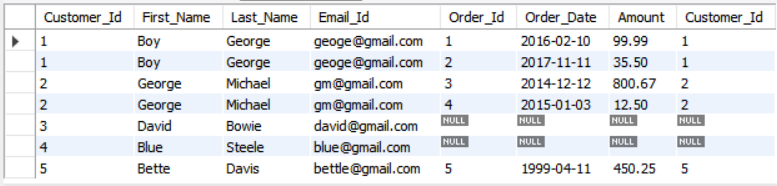
ON A.Customer\_Id = B.Customer\_Id

GROUP BY A.Customer\_Id, First\_Name, Last\_Name

ORDER BY Total;

**Left Join** – Select everything from the first table or the left table, along with any matching records from the second table or the right table.

Ex:



SELECT \*

FROM Customers AS A

LEFT JOIN Orders AS B

ON A.Customer\_Id = B.Customer\_Id;

IFNULL(expr, value) function can be used for replacing the values in a expression where there is NULL to the value.

Syntax:

SELECT column1\_name, IFNULL(SUM(column\_name), 0)

FROM table

GROUP BY column1\_name;

**Right Join** – Select everything from the second table or the right table, along with any matching records from the first table or the left table.

**ON DELETE Cascade:**

What happens when we try to delete a row from a table whose column is being used as a reference in another table. If we apply the constraint of foreign key to a column in a table and reference another’s table’s column, the table won’t allow entry for rows whose value don’t exist in another table so what if in that case we entered rows in that table and they were allowed but later we deleted some rows from the table that was referenced. Well, we face an error just like when we try to enter rows in the second table so we can’t delete rows unless we delete the corresponding rows from both tables at the same time or we can just change the value of the column.

We use ON DELETE CASCADE makes it so that all the rows are deleted in the second table too in response to rows being deleted in the first table.

We can set this behaviour up when referencing: -

FOREIGN KEY column REFERENCES table(column\_id) ON DELETE CASCADE;

Exercise:

-- 1. Write a schema for two tables Students and Papers and they have a one-to-many relationship between them

CREATE DATABASE College;

USE College;

CREATE TABLE Students

(

Student\_Id SMALLINT UNSIGNED AUTO\_INCREMENT,

First\_Name VARCHAR(20) NOT NULL,

PRIMARY KEY(Student\_Id)

);

DESC Students;

CREATE TABLE Papers

(

Title VARCHAR(50) NOT NULL,

Grade TINYINT UNSIGNED NOT NULL,

Student\_Id SMALLINT UNSIGNED NOT NULL,

CONSTRAINT Grade\_Range CHECK(Grade BETWEEN 0 AND 100),

FOREIGN KEY(Student\_Id) REFERENCES Students(Student\_Id) ON DELETE CASCADE

);

DESC Papers;

-- Inserting the data into the tables

INSERT INTO Students(First\_Name)

VALUES ('Caleb'), ('Samantha'), ('Raj'), ('Carlos'), ('Lisa');

INSERT INTO Papers

VALUES

('My First Book Report', 60, 1),

('My Second Book Report', 75, 1),

('Russian Lit Through The Ages', 94, 2),

('De Montaigne and The Art of The Essay', 98, 2),

('Borges and Magical Realism', 89, 4);

SELECT \*

FROM Students;

SELECT \*

FROM Papers;

-- Print the output that's on the screen

SELECT First\_Name, Title, Grade

FROM Students AS A

JOIN Papers AS B

ON A.Student\_Id = B.Student\_Id

ORDER BY Grade DESC;

-- Print the output that's on the screen

SELECT First\_Name, Title, Grade

FROM Students AS A

LEFT JOIN Papers AS B

ON A.Student\_Id = B.Student\_Id;

-- Make some changes to the above query

SELECT First\_Name, IFNULL(Title, 'MISSING'), IFNULL(Grade, 0)

FROM Students AS A

LEFT JOIN Papers AS B

ON A.Student\_Id = B.Student\_Id;

-- Print the output that's on the screen

SELECT First\_Name, IFNULL(AVG(Grade), 0) AS Average

FROM Students AS A

LEFT JOIN Papers AS B

ON A.Student\_Id = B.Student\_Id

GROUP BY First\_Name

ORDER BY Average DESC;

-- Print the output that's on the screen

SELECT First\_Name, IFNULL(AVG(Grade), 0) AS Average,

CASE

WHEN AVG(Grade) >= 75 THEN 'PASSING'

ELSE 'FAILING'

END AS Passing\_Status

FROM Students AS A

LEFT JOIN Papers AS B

ON A.Student\_Id = B.Student\_Id

GROUP BY First\_Name

ORDER BY Average DESC;

-- Students that haven't submitted a report

SELECT First\_Name

FROM Students AS A

LEFT JOIN Papers AS B

ON A.Student\_Id = B.Student\_Id

WHERE Grade IS NULL;

DROP DATABASE College;

**Section – 14 -> Many to Many Joins**

Just as a reminder Many to Many relationships are those when many rows from one table can be related to many rows of the other table and vice versa for ex. the table movie and the table reviewers where one movie can have multiple reviewers but also one reviewer can review multiple movies.

Other many to many relationships example:

1. Books <> Authors
2. Posts <> Tags
3. Students <> Classes

To create a many-to-many relationship between our two tables namely Movies and Reviewers we will need a join table called Reviews where both tables will have a one-to-many relationship with this Reviews table.

In addition to CASE WHEN THEN END statement which can be used for checking multiple condition, we also have the IF() function which can be used for checking a single condition and assigning value depending on the outcome.

Syntax = SELECT IF(column > value, value if true, value if false);

Syntax For Joining three tables:

SELECT columns

FROM table1 AS A

JOIN table2 AS B ON B.Id = A.B\_Id

JOIN table3 AS C ON C.Id = A.C\_Id;

-- Here table 1 is the one that is the join table.

**Section – 15 -> Views, Modes, & More!**

**Views** – Views are stored queries that when invoked produce a result set. A view acts as a virtual table.

Syntax:

CREATE VIEW view\_name AS

SELECT columns

FROM table1 AS A

JOIN table2 AS B ON B.Id = A.B\_Id

JOIN table3 AS C ON C.Id = A.C\_Id;

Print out the data stored in a view:

SELECT \*

FROM view\_name;

Drop the view:

DROP VIEW view\_name;

All in all, views allow us to store a table generated through a query and then later we can use that view to work with the virtual table just like when we work with a real table, however not all the functions and commands can be applied on this virtual table.

**Can Dos and Can’t Dos with Views:**

Only a small portion of views are insertable and updateable i.e., INSERT, UPDATE and DELETE doesn’t work well with views contain any of the following:

* Aggregate Functions or Window Functions [SUM, MIN, MAX, COUNT, AVG, and so forth],
* DISTINCT,
* GROUP BY,
* HAVING,
* UNION or UNION ALL,
* Subquery in the select list

Nondependent Subqueries in the select list fail for INSERT, but are okay for UPDATE, DELETE. For dependent subqueries in the select list, no data change statements are permitted,

* Certain Joins,
* Reference to nonupdatable view in the FROM clause,
* Subquery in the WHERE clause that refers to a table in the FROM clause.

Note- If we update a view that permits to be updated then the table from which we have created the view will also be updated (Inserted, Updated or Deleted).

To make some minor changes to the view that has already been created we can either drop it and make those changes in the original query and create the view again or we can use the following Syntax:

CREATE OR REPLACE VIEW view\_name AS

Query;

We need this syntax since we won’t be able to save the changed query to the same view\_name since that already exists.

We can always use the ALTER command on the view to change the column name, view name etc.

ALTER VIEW view\_name AS

Query;

**Commands/Clauses that Work with GROUP BY:**

* HAVING – HAVING is a command that can be used for filtering the groups returned after going through GROUP BY. The HAVING clause specifies conditions on groups, typically formed by the GROUP BY clause.
* WITH ROLLUP – The GROUP BY clause permits a WITH ROLLUP modifier that causes summary output to include extra rows that represent higher-level (that is, super-aggregate) summary operations. ROLLUP thus enables us to answer questions at multiple levels of analysis with a single query like if we use WITH ROLLUP in a query that is grouping data by some categorical columns and printing the average of each group then WITH ROLLUP will add an additional row that will return the average of the whole column.

Syntax: –

SELECT categorical column, AVG(numerical column) AS Average

FROM table

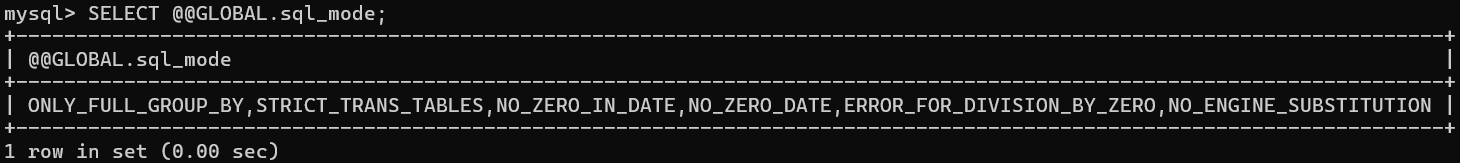
GROUP BY categorical column WITH ROLLUP;

SQL Modes – These modes are basically settings that we can turn on and off to change the behaviour and validation of MySQL.

There are two different scopes for a SQL mode: -

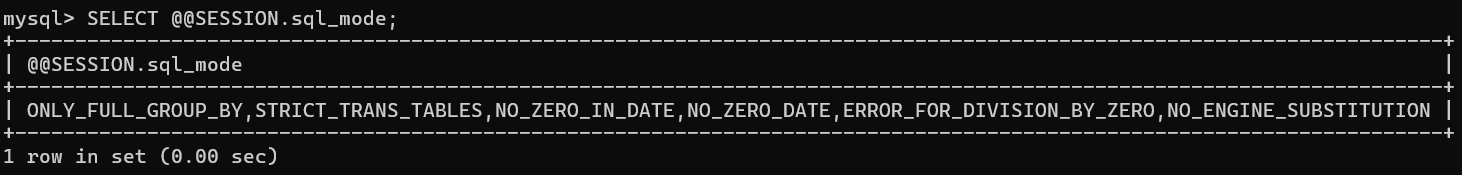
1. Global Mode – Settings that are enabled globally, to view –

SELECT @@GLOBAL.sql\_mode;



1. Session Mode – Settings that are enabled in the current or a particular session only, to view –

SELECT @@SESSION.sql\_mode;



As seen, the session mode is the same as the global because we haven’t changed any modes yet.

To change the modes for each type we have the following commands:

SET GLOBAL sql\_mode = ‘modes’;

SET SESSION sql\_mode = ‘modes’;

If we put ‘’ in place of ‘modes’, all the default modes will be disabled.

**SQL Modes:**

* STRICT\_TRANS\_TABLES – Basically raises error wherever there is a situation that warrants it and blocks the updating of data in the table.
* ONLY\_FULL\_GROUP\_BY – Rejects queries for which the select list, HAVING condition, or ORDER BY list refer to nonaggregated columns that are neither named in the GROUP BY clause nor are functionally dependent on (uniquely dependent by) GROUP BY columns.
* NO\_ZERO\_IN\_DATE – This mode affects whether the server permits dates in which the year part is non zero but month or day part is 0.
  + If this mode is not enabled, dates with zero parts are permitted and inserts produce no warning.
  + If this mode is enabled, dates with zero parts are inserted as ‘0000-00-00’ and produces a warning.
  + If this mode and strict mode are enabled, dates with zero parts are not permitted and inserts produce an error.
* STRICT\_MODE – This mode will be enabled if we have either STRICT\_ALL\_TABLES or STRICT\_TRANS\_TABLES enabled.

**Section – 16 -> Window Functions**

Window Functions – Window functions for each row from a query, perform a calculation using rows related to that row.

A window function performs an aggregate-like operation on a set of query rows. However, whereas an aggregate operation groups rows into a single result row, a window function produces a result for each query row.

So, window functions perform aggregate operations on groups of rows, but they produce a result for each row.

Syntax:

SELECT columns,

AGG\_FUN(numerical\_column) OVER(PARTITION BY categorical\_column) AS Alias

-- AGG\_FUN can be AVG, SUM, COUNT, MIN, MAX and so on

FROM table\_name;

Example:

SELECT Emp\_No, Department, Salary,

AVG(Salary) OVER(PARTITION BY Department) AS Department\_Average

FROM Employees;

All in all, what window functions allow us to do is to look at the aggregate values alongside individual rows.

**Using OVER():**

The OVER() clause constructs a window. When it’s empty, the window will include all records.

To do aggregation by group we must use PARTITION BY inside the parenthesis.

**PARTITION BY:**

Inside of the OVER() we can use PARTITION BY to from rows into groups of rows. Or we can say that using PARTITION BY will create windows that will include records related to different groups.

We can use PARTITION BY to form smaller sub windows to group our rows together.

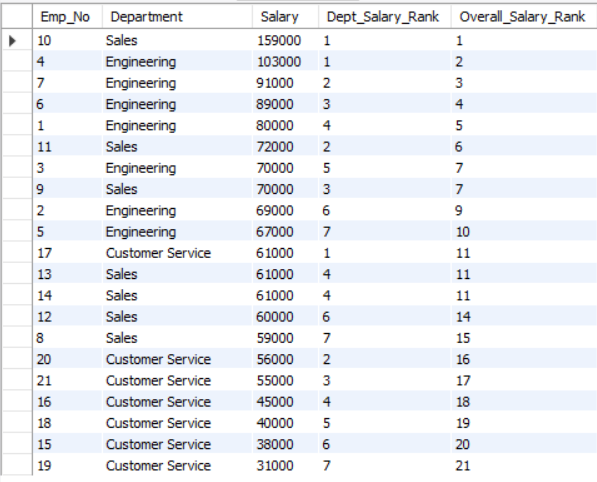
**ORDER BY with Windows:**

We can include the ORDER BY clause inside the OVER() clause to re-order rows within each window.

When we provide ORDER BY to certain aggregate functions like average and sum is that we will get a rolling/moving average or a rolling/moving sum.

**Functions that are solely Window Functions:**

* CUME\_DIST() – Cumulative distribution value,
* DENSE\_RANK() – Rank of current row within its partition, without gaps,
* FIRST\_VALUE() – Value of argument from first row of window frame,
* LAG() – Value of argument from row lagging current row within partition,
* LAST\_VALUE() – Value of argument from last row of window frame,
* LEAD() – Value of argument from row leading current row within partition,
* NTH\_VALUE() – Value of argument from N-th row of window frame,
* NTILE() – Bucket number of current row within its partition,
* PERCENT\_RANK() – Percentage rank value,
* RANK() – Rank of current row within its partition, with gaps
* ROW\_NUMBER() – Index of current row within its partition.



Thorough Examination:

RANK – When we use rank any values that have the same value gets assigned the same

rank and a number get skipped.

Ex:

SELECT Emp\_No, Department, Salary,

RANK() OVER(PARTITION BY Department ORDER BY Salary DESC) AS Dept\_Salary\_Rank,

RANK() OVER(ORDER BY Salary DESC) AS Overall\_Salary\_Rank

FROM Employees;

DENSE\_RANK – Almost the same as RANK but it doesn’t skip a number when assigning rank to values that are same like rank does.

ROW\_NUMBER – Simply numbers the rows and acts like an index column.

Ex:

SELECT Emp\_No, Department, Salary,

ROW\_NUMBER() OVER(PARTITION BY Department ORDER BY Salary DESC) AS Department\_Row\_Number,

RANK() OVER(PARTITION BY Department ORDER BY Salary DESC) AS Dept\_Salary\_Rank,

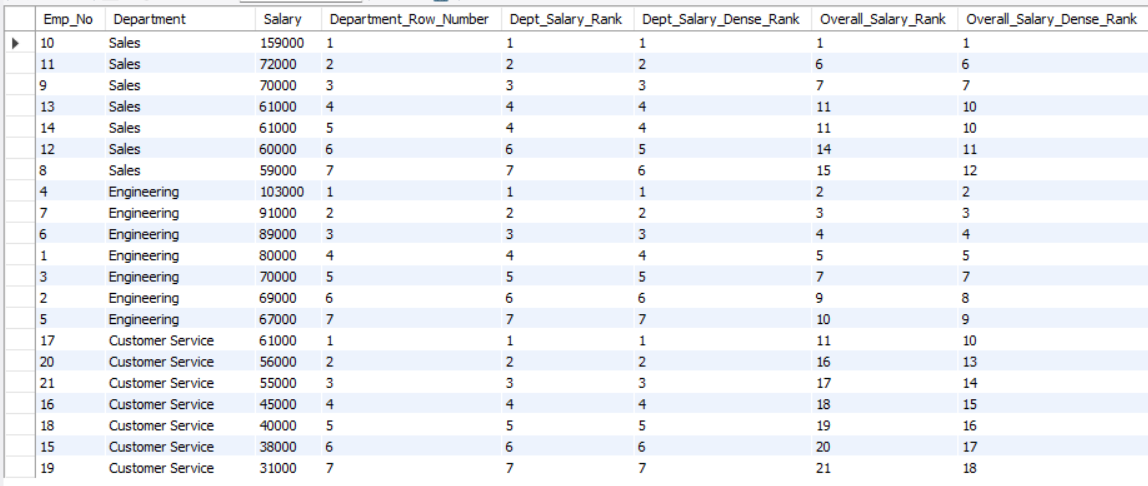
DENSE\_RANK() OVER(PARTITION BY Department ORDER BY Salary DESC) AS Dept\_Salary\_Dense\_Rank,

RANK() OVER(ORDER BY Salary DESC) AS Overall\_Salary\_Rank,

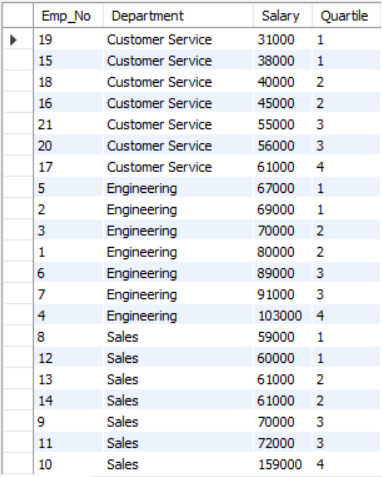
DENSE\_RANK() OVER(ORDER BY Salary DESC) AS Overall\_Salary\_Dense\_Rank

FROM Employees

ORDER BY Department DESC;



NTILE(**N**) – **N** must be a positive integer in the range 1 to 263. Bucket number return values range from 1 to **N**. **N** cannot be NULL.



Ex:

SELECT Emp\_No, Department, Salary,

NTILE(4) OVER(PARTITION BY Department ORDER BY Salary) AS Quartile

FROM Employees

ORDER BY Department;

FIRST\_VALUE(expr) – Returns the value of expr from the first row of the window

frame.

SELECT Emp\_No, Department, Salary,

FIRST\_VALUE(Emp\_No) OVER(PARTITION BY Department ORDER BY Salary DESC) AS Most\_Payed

FROM Employees;

LAST\_VALUE(expr) – Almost the same as FIRST\_VALUE but instead of printing the first value it selects the last value within the window frame.

NTH\_VALUE(expr, N) – Returns the value of expr from the N-th row of the window frame. If there is no such row, the return value is NULL.

LAG(expr, N, default) – Returns the value of expr from the row that lags (precedes) the current row by N rows within its partition. If there is no such row, the return value is default. For ex- if N=3, the return value for the first two rows is default. If N or default are missing, the default is 1 and NULL, respectively.

N must be a literal non negative integer. If N is 0, expr is evaluated for the current row.

**Section – 20 -> Introducing Node**

MySQL and other languages like PHP, Node, Ruby, C#, C++, Java, Python.

We can use NodeJS to play the part of a link between the web app and the MySQL database.

Goal – Use NodeJS to randomly generate and insert 500+ users into a database and therein lies the advantage of using another programming language with MySQL since it allows us to write code that can do tasks otherwise not possible by the SQL language.