

Download Xampp

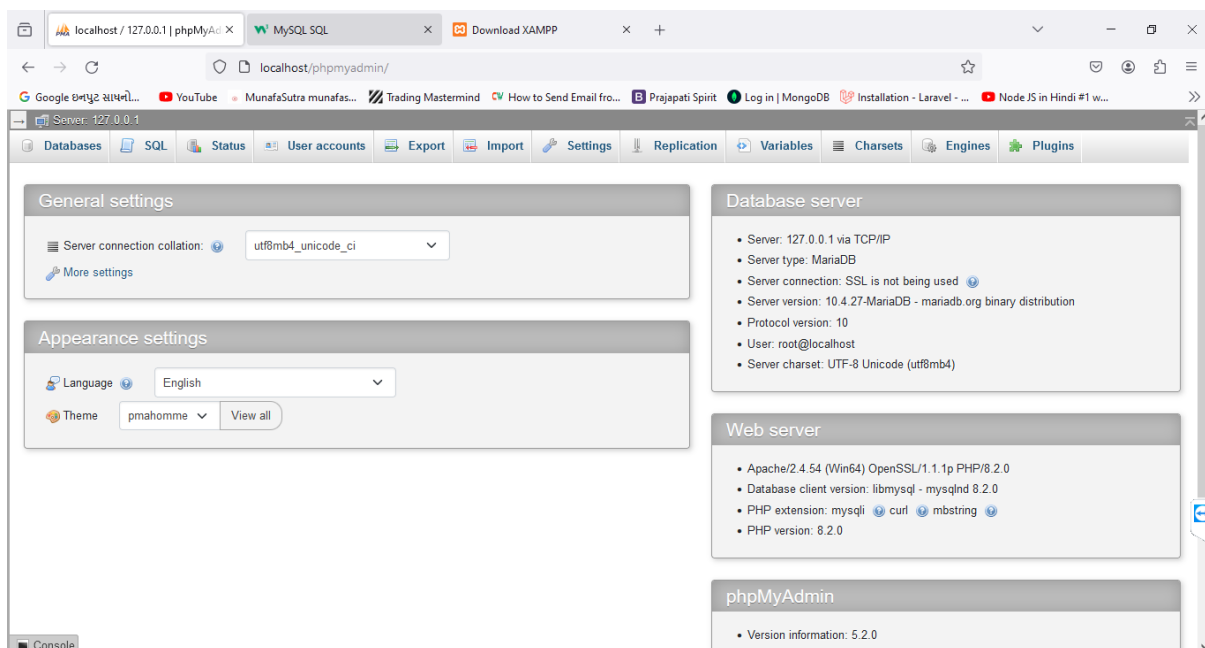
Open <https://www.apachefriends.org/download.html> and download latest version for your operating system.

Install Downloaded software

Open Xampp Admin panel and start apache and mysql service

Close the Xampp Admin panel and open browser

In browser addressbar enter url : localhost/phpMyAdmin



MySQL Tutorial

MySQL is a widely used relational database management system (RDBMS).

MySQL is free and open-source.

MySQL is ideal for both small and large applications.

Introduction to MySQL

MySQL is a very popular open-source relational database management system (RDBMS).

What is MySQL?

- MySQL is a relational database management system
 - MySQL is open-source
 - MySQL is free
 - MySQL is ideal for both small and large applications
 - MySQL is very fast, reliable, scalable, and easy to use
 - MySQL is cross-platform
 - MySQL is compliant with the ANSI SQL standard
 - MySQL was first released in 1995
 - MySQL is developed, distributed, and supported by Oracle Corporation
 - MySQL is named after co-founder Monty Widenius's daughter: My
-

Who Uses MySQL?

- Huge websites like Facebook, Twitter, Airbnb, Booking.com, Uber, GitHub, YouTube, etc.
 - Content Management Systems like WordPress, Drupal, Joomla!, Contao, etc.
 - A very large number of web developers around the world
-

Show Data On Your Web Site

To build a web site that shows data from a database, you will need:

- An RDBMS database program (like MySQL)
- A server-side scripting language, like PHP
- To use SQL to get the data you want
- To use HTML / CSS to style the page

MySQL RDBMS

What is RDBMS?

RDBMS stands for Relational Database Management System.

RDBMS is a program used to maintain a relational database.

RDBMS is the basis for all modern database systems such as MySQL, Microsoft SQL Server, Oracle, and Microsoft Access.

RDBMS uses [SQL queries](#) to access the data in the database.

What is a Database Table?

A table is a collection of related data entries, and it consists of columns and rows.

A column holds specific information about every record in the table.

A record (or row) is each individual entry that exists in a table.

Look at a selection from the Northwind "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

The columns in the "Customers" table above are: CustomerID, CustomerName, ContactName, Address, City, PostalCode and Country. The table has 5 records (rows).

What is a Relational Database?

A relational database defines database relationships in the form of tables. The tables are related to each other - based on data common to each.

Look at the following three tables "Customers", "Orders", and "Shippers" from the Northwind database:

Customers Table

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

The relationship between the "Customers" table and the "Orders" table is the CustomerID column:

Orders Table

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10278	5	8	1996-08-12	2
10280	5	2	1996-08-14	1
10308	2	7	1996-09-18	3
10355	4	6	1996-11-15	1
10365	3	3	1996-11-27	2
10383	4	8	1996-12-16	3
10384	5	3	1996-12-16	3

The relationship between the "Orders" table and the "Shippers" table is the ShipperID column:

Shippers Table

ShipperID	ShipperName	Phone
1	Speedy Express	(503) 555-9831
2	United Package	(503) 555-3199
3	Federal Shipping	(503) 555-9931

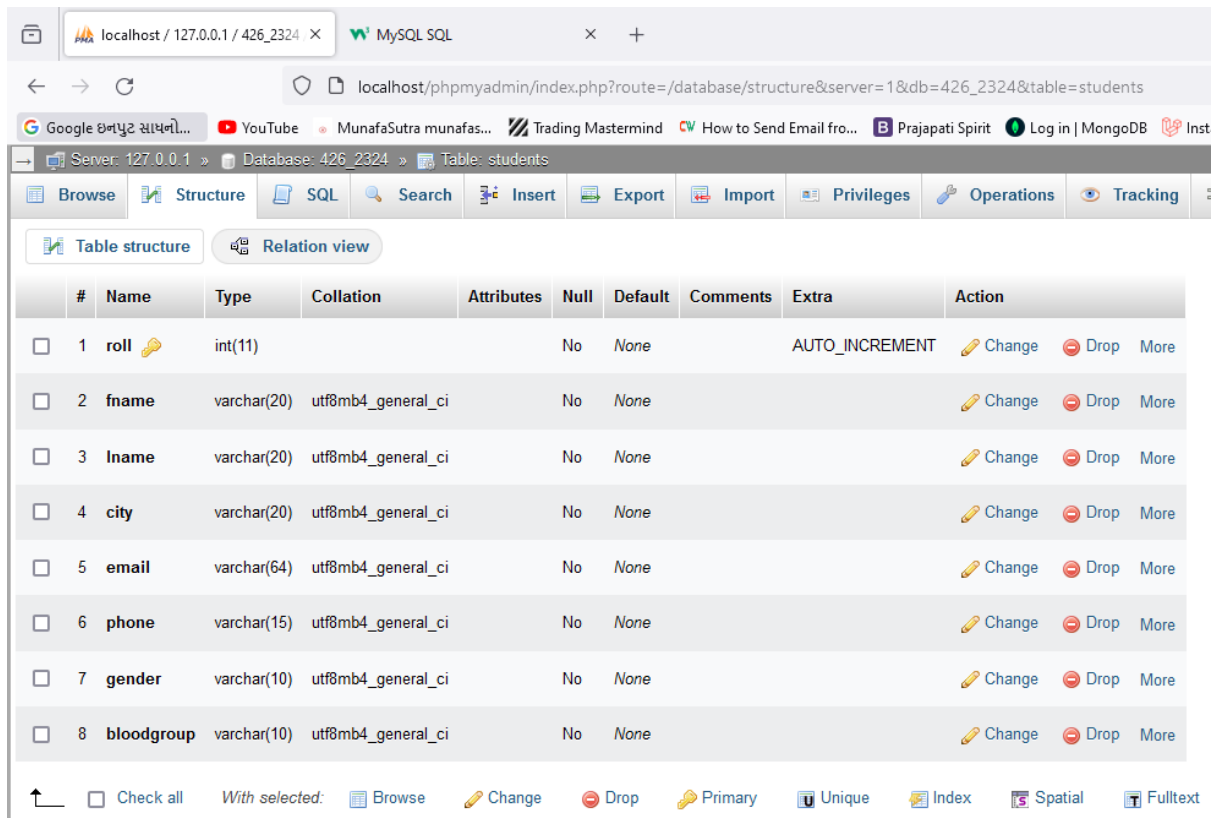
Before begin with MYsql Tutorial you need to create new Database and table with some of the data.

Open database section from localhost/phpMyAdmin -> then select create database section and enter your required database name then press create.

After successfully creation of new database, you will be redirected to newly created database page.

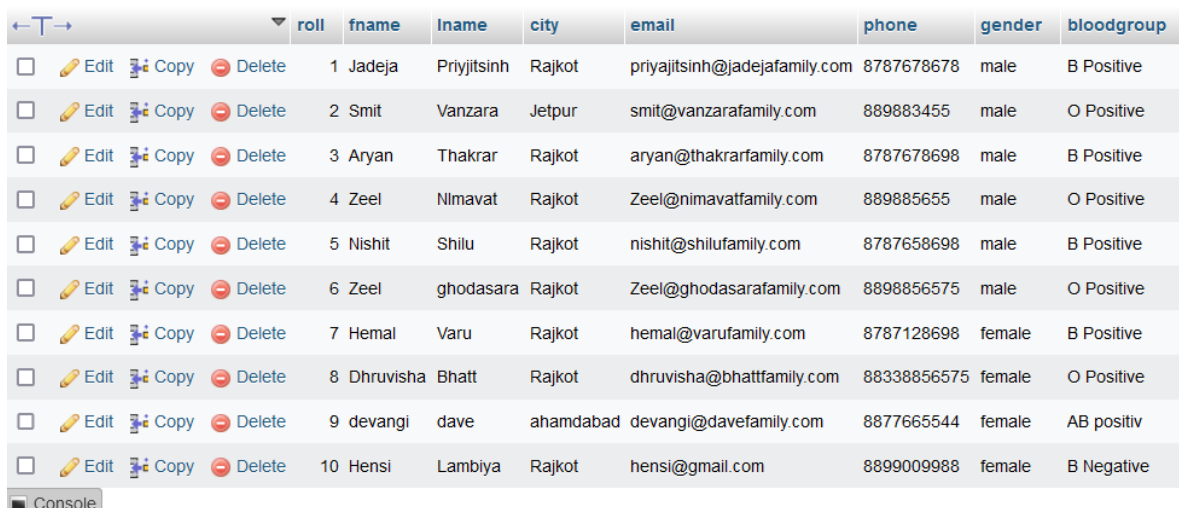
Create new table as following

Enter table name -> students -> create 8 columns



#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	roll	int(11)			No	None		AUTO_INCREMENT	Change Drop More
2	fname	varchar(20)	utf8mb4_general_ci		No	None			Change Drop More
3	lname	varchar(20)	utf8mb4_general_ci		No	None			Change Drop More
4	city	varchar(20)	utf8mb4_general_ci		No	None			Change Drop More
5	email	varchar(64)	utf8mb4_general_ci		No	None			Change Drop More
6	phone	varchar(15)	utf8mb4_general_ci		No	None			Change Drop More
7	gender	varchar(10)	utf8mb4_general_ci		No	None			Change Drop More
8	bloodgroup	varchar(10)	utf8mb4_general_ci		No	None			Change Drop More

Create above table and add some of the data as below



	roll	fname	lname	city	email	phone	gender	bloodgroup
Edit Copy Delete	1	Jadeja	Priyjisinh	Rajkot	priyajitsinh@jadejafamily.com	8787678678	male	B Positive
Edit Copy Delete	2	Smit	Vanzara	Jetpur	smit@vanzarafamily.com	889883455	male	O Positive
Edit Copy Delete	3	Aryan	Thakrar	Rajkot	aryan@thakrarfamily.com	8787678698	male	B Positive
Edit Copy Delete	4	Zeel	Nimavat	Rajkot	Zeel@nimavatfamily.com	889885655	male	O Positive
Edit Copy Delete	5	Nishit	Shilu	Rajkot	nishit@shilufamily.com	8787658698	male	B Positive
Edit Copy Delete	6	Zeel	ghodasara	Rajkot	Zeel@ghodasarafamily.com	8898856575	male	O Positive
Edit Copy Delete	7	Hemal	Varu	Rajkot	hemal@varufamily.com	8787128698	female	B Positive
Edit Copy Delete	8	Dhruvisha	Bhatt	Rajkot	dhruvisha@bhattfamily.com	88338856575	female	O Positive
Edit Copy Delete	9	devangi	dave	ahamdabad	devangi@davefamily.com	8877665544	female	AB positiv
Edit Copy Delete	10	Hensi	Lambiya	Rajkot	hensi@gmail.com	8899009988	female	B Negative

MySQL SQL

What is SQL (Structured Query Language)?

SQL is the standard language for dealing with Relational Databases. SQL is used to insert, search, update, and delete database records.

How to Use SQL

The following SQL statement selects all the records in the "Customers" table:

Example

```
SELECT * FROM Customers;
```

```
SELECT * from students;
```

Keep in Mind That...

- SQL keywords are NOT case sensitive: `select` is the same as `SELECT`

In this tutorial we will write all SQL keywords in upper-case.

```
select * from students;
```

```
SELECT * from students;
```

Semicolon after SQL Statements?

Some database systems require a semicolon at the end of each SQL statement.

Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server. In this tutorial, we will use semicolon at the end of each SQL statement.

Some of The Most Important SQL Commands

- `SELECT` - extracts data from a database
- `UPDATE` - updates data in a database
- `DELETE` - deletes data from a database
- `INSERT INTO` - inserts new data into a database
- `CREATE DATABASE` - creates a new database
- `ALTER DATABASE` - modifies a database
- `CREATE TABLE` - creates a new table
- `ALTER TABLE` - modifies a table
- `DROP TABLE` - deletes a table
- `CREATE INDEX` - creates an index (search key)
- `DROP INDEX` - deletes an index

MySQL SELECT Statement

The MySQL SELECT Statement

The `SELECT` statement is used to select data from a database.

The data returned is stored in a **result table**, called the **result-set**.

SELECT Syntax

```
SELECT column1, column2, ...  
FROM table_name;
```

Here, column1, column2, ... are the field names of the table you want to select data from.

```
SELECT roll, fname, lname from students
```

```
SELECT roll, fname, lname, city from students;
```

If you want to select all the fields available in the table, use the following syntax:

```
SELECT * FROM table_name;
```

```
SELECT * FROM students;
```

SELECT * Example

The following SQL statement selects ALL the columns from the "Customers" table:

Example

```
SELECT * FROM Customers;
```

The MySQL SELECT DISTINCT Statement

The `SELECT DISTINCT` statement is used to return only distinct (different) values.

Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.

SELECT DISTINCT Syntax

```
SELECT DISTINCT column1, column2, ...  
FROM table_name;
```

```
SELECT DISTINCT city from students
```

SELECT Example Without DISTINCT

The following SQL statement selects all (including the duplicates) values from the "Country" column in the "Customers" table:

```
SELECT city from students;
```

```
SELECT COUNT(DISTINCT city) FROM students
```

MySQL WHERE Clause

The MySQL WHERE Clause

The `WHERE` clause is used to filter records.

It is used to extract only those records that fulfill a specified condition.

WHERE Syntax

`SELECT column1, column2, ...`

`FROM table_name`

`WHERE condition;`

`SELECT * FROM students WHERE city = 'rajkot'`

`SELECT * FROM students WHERE roll > 5`

`SELECT * FROM students WHERE roll = 5;`

`SELECT * FROM students WHERE roll <> 5;`

`SELECT * FROM students WHERE not roll = 5;`

Note: The `WHERE` clause is not only used in `SELECT` statements, it is also used in `UPDATE`, `DELETE`, etc.!

Text Fields vs. Numeric Fields

SQL requires single quotes around text values (most database systems will also allow double quotes).

However, numeric fields should not be enclosed in quotes:

`SELECT * from students WHERE gender = 'male'`

`SELECT * from students WHERE gender = male`

`SELECT * from students WHERE roll = 8`

`SELECT * from students WHERE roll = '8';`

MySQL AND, OR and NOT Operators

The MySQL AND, OR and NOT Operators

The `WHERE` clause can be combined with `AND`, `OR`, and `NOT` operators.

The `AND` and `OR` operators are used to filter records based on more than one condition:

- The `AND` operator displays a record if all the conditions separated by `AND` are `TRUE`.
- The `OR` operator displays a record if any of the conditions separated by `OR` is `TRUE`.
- The `NOT` operator displays a record if the condition(s) is `NOT TRUE`.

AND Syntax

```
SELECT column1, column2, ...  
FROM table_name  
WHERE condition1 AND condition2 AND condition3 ...;
```

OR Syntax

```
SELECT column1, column2, ...  
FROM table_name  
WHERE condition1 OR condition2 OR condition3 ...;
```

NOT Syntax

```
SELECT column1, column2, ...  
FROM table_name  
WHERE NOT condition;
```

```
SELECT roll, fname, lname, city from students WHERE roll = 1 and city = 'rajkot'
```

```
SELECT roll, fname, lname, city from students WHERE roll = 1 and city = 'surat';
```

```
SELECT roll, fname, lname, city from students WHERE city = 'rajkot' or city = 'surat';
```

```
SELECT roll, fname, lname, city from students WHERE not (city = 'rajkot' or city = 'surat');
```

```
SELECT roll, fname, lname, city from students WHERE not city = 'rajkot' ;
```

Combining AND, OR and NOT

You can also combine the `AND`, `OR` and `NOT` operators.

```
SELECT * from students WHERE roll = 1 and (city = 'rajkot' or city = 'surat')
```

MySQL ORDER BY Keyword

The MySQL ORDER BY Keyword

The `ORDER BY` keyword is used to sort the result-set in ascending or descending order.

The `ORDER BY` keyword sorts the records in ascending order by default. To sort the records in descending order, use the `DESC` keyword.

ORDER BY Syntax

```
SELECT column1, column2, ...  
FROM table_name  
ORDER BY column1, column2, ... ASC|DESC;
```

```
SELECT * from students
```

```
SELECT * from students ORDER by fname;
```

```
SELECT * from students ORDER by fname desc;
```

ORDER BY Several Columns Example

```
SELECT * from students ORDER by fname desc, lname desc;
```

MySQL INSERT INTO Statement

The MySQL INSERT INTO Statement

The `INSERT INTO` statement is used to insert new records in a table.

INSERT INTO Syntax

It is possible to write the `INSERT INTO` statement in two ways:

1. Specify both the column names and the values to be inserted:

```
INSERT INTO table_name (column1, column2, column3, ...)  
VALUES (value1, value2, value3, ...);
```

2. If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. Here, the `INSERT INTO` syntax would be as follows:

```
INSERT INTO table_name  
VALUES (value1, value2, value3, ...);
```

INSERT INTO Example

The following SQL statement inserts a new record in the "Students" table:

```
INSERT into students (fname, lname, city, email, phone, gender, bloodgroup) VALUES  
( 'patel', 'riya', 'ahamdabad', 'riya@gmail.com', '9988999900', 'female', 'o positive');
```

1 row inserted.

Inserted row id: **11** (Query took 0.0032 seconds.)

Did you notice that we did not insert any number into the Roll field?

The **roll** column is an [auto-increment](#) field and will be generated automatically when a new record is inserted into the table.

```
INSERT into students (fname, lname, city, email, phone, gender, bloodgroup) VALUES ('patel', 'siya',  
'ahamdabad', 'siya@gmail.com', '9988999900', 'female', 'o positive');
```

Insert Data Only in Specified Columns

It is also possible to only insert data in specific columns.

```
INSERT into students (fname, lname, city, gender, bloodgroup) VALUES ('patel', 'siya', 'ahamdabad',  
'female', 'o positive');
```

```
INSERT into students VALUES ('patel', 'jiya', 'jiya@gmail.com', '9988998899', 'ahamdabad', 'female', 'o positive');
```

```
#1136 - Column count doesn't match value count at row 1
```

```
INSERT into students VALUES (null, 'patel', 'jiya', 'jiya@gmail.com', '9988998899', 'ahamdabad', 'female', 'o positive');
```

MySQL NULL Values

What is a NULL Value?

A field with a NULL value is a field with no value.

If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

Note: A NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation!

```
SELECT * from students WHERE email is null
```

```
SELECT * from students WHERE email = '';
```

How to Test for NULL Values?

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the `IS NULL` and `IS NOT NULL` operators instead.

IS NULL Syntax

```
SELECT column_names  
FROM table_name  
WHERE column_name IS NULL;
```

IS NOT NULL Syntax

```
SELECT column_names  
FROM table_name  
WHERE column_name IS NOT NULL;
```

```
INSERT into students (fname, lname, city, gender, bloodgroup) VALUES ('patel', 'siya', 'ahamdabad',  
'female', 'o positive');
```

```
INSERT into students (fname, lname, city, gender, bloodgroup) VALUES ('pandya', 'priya',  
'ahamdabad', 'female', 'o positive');
```

```
SELECT * from students WHERE email is null
```

```
SELECT * from students WHERE email is not null;
```

Tip: Always use `IS NULL` to look for NULL values.

The IS NOT NULL Operator

The `IS NOT NULL` operator is used to test for non-empty values (NOT NULL values).

MySQL UPDATE Statement

The MySQL UPDATE Statement

The `UPDATE` statement is used to modify the existing records in a table.

UPDATE Syntax

`UPDATE table_name`

`SET column1 = value1, column2 = value2, ...`

`WHERE condition;`

Note: Be careful when updating records in a table! Notice the `WHERE` clause in the `UPDATE` statement. The `WHERE` clause specifies which record(s) that should be updated. If you omit the `WHERE` clause, all records in the table will be updated!

```
UPDATE students set city = 'Gandhinagar' WHERE roll = 13
```

```
UPDATE students set city = 'Bhuj' WHERE roll > 13
```

UPDATE Multiple Records

It is the `WHERE` clause that determines how many records will be updated.

```
UPDATE students set email = 'demo@gmail.com', phone = '0000000000' where email is null
```

Update Warning!

Be careful when updating records. If you omit the `WHERE` clause, **ALL** records will be updated!

```
UPDATE students set bloodgroup = 'O Negative'
```

```
UPDATE students SET bloodgroup = 'B +ve' WHERE roll >= 12
```

MySQL LIMIT Clause

The MySQL LIMIT Clause

The `LIMIT` clause is used to specify the number of records to return.

The `LIMIT` clause is useful on large tables with thousands of records. Returning a large number of records can impact performance.

LIMIT Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

```
SELECT * FROM students LIMIT 5
```

```
SELECT * FROM students WHERE city = 'rajkot' LIMIT 5;
```

What if we want to select records 4 - 6 (inclusive)?

MySQL provides a way to handle this: by using `OFFSET`.

The SQL query below says "return only 3 records, start on record 4 (`OFFSET 3`)":

```
SELECT * FROM students LIMIT 5 OFFSET 5;
```

```
SELECT * FROM students LIMIT 5 OFFSET 10;
```

```
SELECT * FROM students LIMIT 10, 5;
```

```
SELECT * FROM students LIMIT 0, 5;
```

```
SELECT * FROM students LIMIT 20, 5;
```

MySQL MIN() and MAX() Functions

MySQL MIN() and MAX() Functions

The `MIN()` function returns the smallest value of the selected column.

The `MAX()` function returns the largest value of the selected column.

MIN() Syntax

```
SELECT MIN(column_name)  
FROM table_name  
WHERE condition;
```

MAX() Syntax

```
SELECT MAX(column_name)  
FROM table_name  
WHERE condition;
```

```
SELECT max(roll) FROM students
```

```
SELECT min(roll) FROM students;
```

```
SELECT max(fees) FROM students
```

```
SELECT min(fees) FROM students;
```

MySQL COUNT(), AVG() and SUM() Functions

MySQL COUNT(), AVG() and SUM() Functions

The `COUNT()` function returns the number of rows that matches a specified criterion.

`COUNT()` Syntax

```
SELECT COUNT(column_name)  
FROM table_name  
WHERE condition;
```

The `AVG()` function returns the average value of a numeric column.

`AVG()` Syntax

```
SELECT AVG(column_name)  
FROM table_name  
WHERE condition;
```

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

`SUM()` Syntax

```
SELECT SUM(column_name)  
FROM table_name  
WHERE condition;
```

```
SELECT COUNT(roll) from students where fees > 10000;
```

```
SELECT COUNT(roll), sum(fees) from students where fees > 10000;
```

```
SELECT COUNT(roll), sum(fees) from students where city = 'rajkot';
```

```
SELECT COUNT(roll), sum(fees), avg(fees) from students where city = 'rajkot';
```

MySQL LIKE Operator

The MySQL LIKE Operator

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

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

- The percent sign (%) represents **zero, one, or multiple** characters
- The underscore sign (_) represents **one, single** character

```
SELECT * from students WHERE fname like '%a%'
```

```
SELECT * from students WHERE fname like '%a';
```

```
SELECT * from students WHERE fname like '_a%';
```

```
SELECT * from students WHERE fname not like '_a%';
```

LIKE Syntax

```
SELECT column1, column2, ... FROM table_name WHERE columnN LIKE pattern;
```

Tip: You can also combine any number of conditions using `AND` or `OR` operators.

Here are some examples showing different `LIKE` operators with '%' and '_' wildcards:

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that start with "a"
WHERE CustomerName LIKE '%a'	Finds any values that end with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%'	Finds any values that start with "a" and are at least 2 characters in length
WHERE CustomerName LIKE 'a__%'	Finds any values that start with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that start with "a" and ends with "o"

```
SELECT * from students WHERE fname like '_r%';
```

```
SELECT * from students WHERE fname like 'a%n';
```

MySQL Wildcards

MySQL Wildcard Characters

A wildcard character is used to substitute one or more characters in a string.

Wildcard characters are used with the [LIKE](#) operator. The `LIKE` operator is used in a `WHERE` clause to search for a specified pattern in a column.

Wildcard Characters in MySQL

Symbol	Description	Example
%	Represents zero or more characters	<code>bl%</code> finds <code>bl</code> , <code>black</code> , <code>blue</code> , and <code>blob</code>
_	Represents a single character	<code>h_t</code> finds <code>hot</code> , <code>hat</code> , and <code>hit</code>

The wildcards can also be used in combinations!

```
SELECT * from students WHERE city like 'raj%'
```

MySQL IN Operator

The MySQL IN Operator

The `IN` operator allows you to specify multiple values in a `WHERE` clause.

The `IN` operator is a shorthand for multiple `OR` conditions.

IN Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

or:

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (SELECT STATEMENT);
```

```
SELECT * from students WHERE city = 'Rajkot' or city = 'baroda' or city = 'jetpur'
```

IN Operator Examples

```
SELECT * from students WHERE city in ('Rajkot', 'baroda', 'jetpur');
```

```
SELECT * from students WHERE city not in ('Rajkot', 'baroda', 'jetpur');
```


MySQL BETWEEN Operator

The MySQL BETWEEN Operator

The `BETWEEN` operator selects values within a given range. The values can be numbers, text, or dates.

The `BETWEEN` operator is inclusive: begin and end values are included.

BETWEEN Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

```
SELECT * from students WHERE roll >= 5 and roll <= 10
```

```
SELECT * from students WHERE roll between 1 and 5
```

```
SELECT * from students WHERE roll between 5 and 10;
```

```
SELECT * from students WHERE fname between 'aryan' and 'nishit';
```

```
SELECT * from students WHERE fees between 10000 and 15000;
```

```
SELECT * from students WHERE fees not between 10000 and 15000;
```

BETWEEN with IN Example

```
SELECT * from students WHERE fees not between 10000 and 15000 and roll in (1,2,3,4,5);
```

```
SELECT * from students WHERE fees not between 10000 and 15000 and roll <= 5;
```

BETWEEN Text Values Example

```
SELECT * from students WHERE fname between 'aryan' and 'nishit';
```

BETWEEN Dates Example

```
SELECT * from students WHERE admissiondate BETWEEN '2020-01-01' and '2022-12-31'
```

```
SELECT * from students WHERE admissiondate not BETWEEN '2020-01-01' and '2022-12-31';
```

MySQL DELETE Statement

The MySQL DELETE Statement

The `DELETE` statement is used to delete existing records in a table.

DELETE Syntax

DELETE FROM *table_name* **WHERE** *condition*;

Note: Be careful when deleting records in a table! Notice the `WHERE` clause in the `DELETE` statement. The `WHERE` clause specifies which record(s) should be deleted. If you omit the `WHERE` clause, all records in the table will be deleted!

SQL DELETE Example

DELETE from students WHERE city = 'Bhuj'

DELETE from students WHERE bloodgroup = 'B +ve'

Delete All Records

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

DELETE from students

Truncate table students

Delete all the data from table and reset all the auto increment from table

TRUNCATE TABLE students

MySQL Aliases

MySQL Aliases

Aliases are used to give a table, or a column in a table, a temporary name.

Aliases are often used to make column names more readable.

An alias only exists for the duration of that query.

An alias is created with the `AS` keyword.

Alias Column Syntax

```
SELECT column_name AS alias_name  
FROM table_name;
```

```
SELECT roll, fname, lname FROM students;
```

```
SELECT roll as "Roll Number", fname as "First Name", lname as "Last Name" FROM students;
```

Alias Table Syntax

```
SELECT column_name(s)  
FROM table_name AS alias_name;
```

Note: Single or double quotation marks are required if the alias name contains spaces:

```
SELECT roll, fname, lname, city, email, gender, phone from students
```

```
SELECT concat_ws(" - ",roll, fname, lname, city, email, gender, phone ) from students;
```

```
SELECT concat_ws(" - ",roll, fname, lname, city, email, gender, phone ) as "Student Information" from students;
```

Aliases can be useful when:

- There are more than one table involved in a query
- Functions are used in the query
- Column names are big or not very readable
- Two or more columns are combined together

MySQL Joins

MySQL Joining Tables

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

```
SELECT students.roll, students.fname, students.lname, students.city, students.email, students.phone, marks.total, marks.result from students inner join marks on students.roll = marks.roll
```

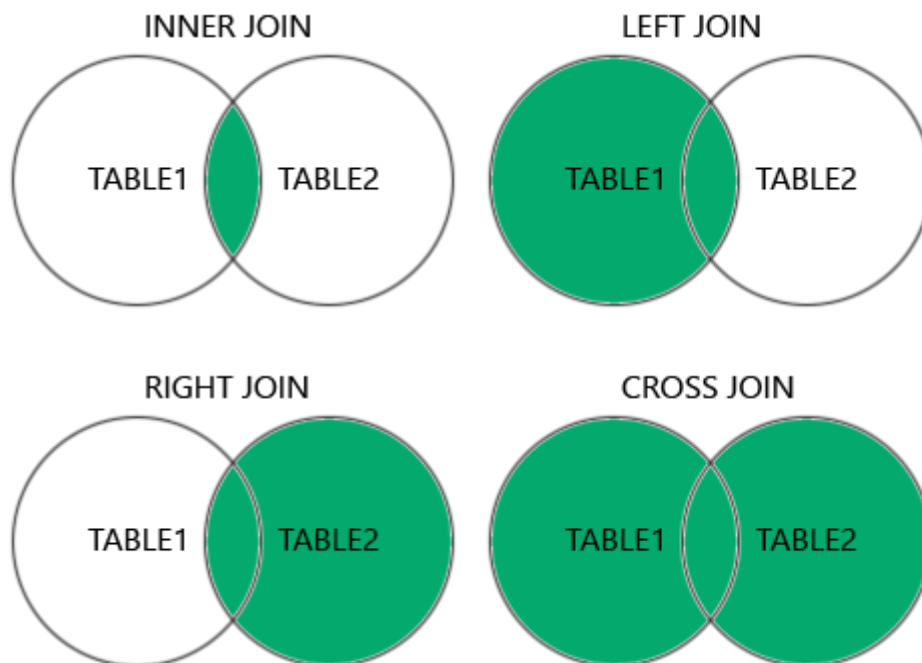
```
SELECT students.roll, students.fname, students.lname, attendance.absents, attendance.presents from students inner join attendance on students.roll = attendance.roll
```

```
SELECT s.roll, s.fname, s.lname, a.absents, a.presents from students as s inner join attendance as a on s.roll = a.roll;
```

```
SELECT s.roll, s.fname, s.lname, a.absents, a.presents from students s inner join attendance a on s.roll = a.roll;
```

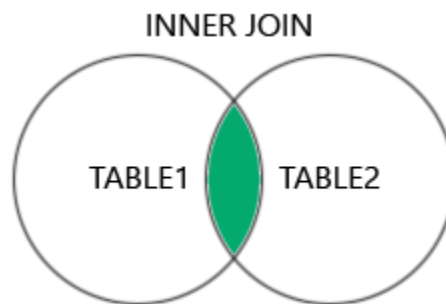
Supported Types of Joins in MySQL

- `INNER JOIN`: Returns records that have matching values in both tables
- `LEFT JOIN`: Returns all records from the left table, and the matched records from the right table
- `RIGHT JOIN`: Returns all records from the right table, and the matched records from the left table
- `CROSS JOIN`: Returns all records from both tables



MySQL INNER JOIN Keyword

The `INNER JOIN` keyword selects records that have matching values in both tables.



INNER JOIN Syntax

`SELECT column_name(s)`

`FROM table1`

`INNER JOIN table2`

`ON table1.column_name = table2.column_name;`

`SELECT students.roll, students.fname, students.lname, students.city, students.email, students.phone, students.gender, attendance.absents, attendance.presents from students INNER join attendance on students.roll = attendance.roll`

`SELECT s.roll, s.fname, s.lname, s.city, s.email, s.phone, s.gender, s.admissiondate, a.absents, a.presents from students s INNER join attendance a on s.roll = a.roll`

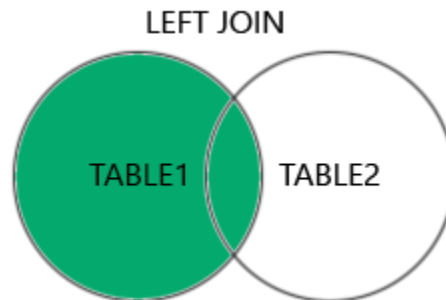
JOIN Three Tables

`SELECT s.roll, s.fname, s.lname, s.city, s.email, s.phone, s.gender, s.admissiondate, a.absents, a.presents, m.total, m.result from students s INNER JOIN attendance a on s.roll = a.roll INNER JOIN marks m on s.roll = m.roll;`

MySQL LEFT JOIN Keyword

MySQL LEFT JOIN Keyword

The `LEFT JOIN` keyword returns all records from the left table (table1), and the matching records (if any) from the right table (table2).



LEFT JOIN Syntax

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

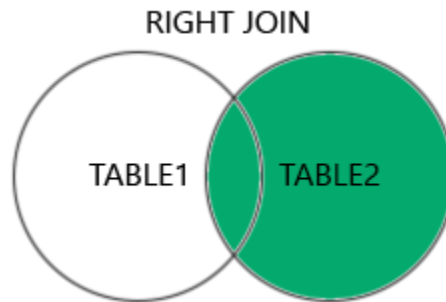
Note: The `LEFT JOIN` keyword returns all records from the left table (Customers), even if there are no matches in the right table (Orders). If not data match in right table empty fields filled with NULL value.

```
SELECT students.roll, students.fname, students.lname, students.city, students.phone, students.email,
attendance.absents, attendance.presents from students LEFT JOIN attendance on students.roll =
attendance.roll
```

MySQL RIGHT JOIN Keyword

MySQL RIGHT JOIN Keyword

The `RIGHT JOIN` keyword returns all records from the right table (table2), and the matching records (if any) from the left table (table1).



RIGHT JOIN Syntax

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

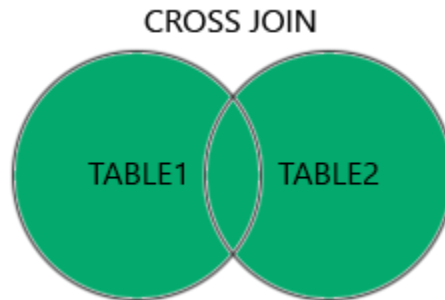
Note: The `RIGHT JOIN` keyword returns all records from the right table (Employees), even if there are no matches in the left table (Orders). If not data match in left table empty fields filled with NULL value.

```
SELECT students.roll, students.fname, students.lname, students.city, students.phone, students.email,
attendance.absents, attendance.presents from students RIGHT JOIN attendance on students.roll =
attendance.roll;
```

MySQL CROSS JOIN Keyword

SQL CROSS JOIN Keyword

The `CROSS JOIN` keyword returns all records from both tables (table1 and table2).



CROSS JOIN Syntax

```
SELECT column_name(s)
FROM table1
CROSS JOIN table2;
```

Note: `CROSS JOIN` can potentially return very large result-sets!

Note: The `CROSS JOIN` keyword returns all matching records from both tables whether the other table matches or not.

```
SELECT * from students CROSS JOIN attendance;
```

If you add a `WHERE` clause (if table1 and table2 has a relationship), the `CROSS JOIN` will produce the same result as the `INNER JOIN` clause:

```
SELECT * from students CROSS JOIN attendance WHERE students.roll = attendance.roll;
```


MySQL Self Join

MySQL Self Join

A self join is a regular join, but the table is joined with itself.

Self Join Syntax

```
SELECT column_name(s)
FROM table1 T1, table1 T2
WHERE condition;
```

T1 and *T2* are different table aliases for the same table.

```
SELECT s1.roll, concat_ws("-", s1.fname, s1.lname) as "Student From Table 1", s2.roll, concat_ws("-",
s2.fname, s2.lname) as "Student From Table 2" from students s1, students s2 WHERE s1.roll <> s2.roll
and s1.city = s2.city ORDER by s1.city;
```

MySQL UNION Operator

The MySQL UNION Operator

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

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

UNION Syntax

```
SELECT column_name(s) FROM table1
```

```
UNION
```

```
SELECT column_name(s) FROM table2
```

```
CREATE table students1 as SELECT * from students WHERE city = 'Rajkot'
```

UNION ALL Syntax

The `UNION` operator selects only distinct values by default. To allow duplicate values, use

`UNION ALL`:

```
SELECT column_name(s) FROM table1
```

```
UNION ALL
```

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

```
SELECT * from students
```

```
UNION
```

```
SELECT * from students1
```

```
SELECT * from students
```

```
UNION ALL
```

```
SELECT * from students1;
```

SQL UNION With WHERE

```
SELECT * from students WHERE city = 'Rajkot'
```

```
UNION ALL
```

```
SELECT * from students1 WHERE city = 'Rajkot';
```

MySQL GROUP BY Statement

The MySQL GROUP BY Statement

The `GROUP BY` statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The `GROUP BY` statement is often used with aggregate functions (`COUNT()`, `MAX()`, `MIN()`, `SUM()`, `AVG()`) to group the result-set by one or more columns.

```
SELECT gender, COUNT(gender) as "Total Students" from students GROUP by gender
```

```
SELECT city, COUNT(city) as "Total Students From " from students GROUP by(city)
```

```
SELECT city, COUNT(city) as "Total Students From " from students GROUP by(city) ORDER by  
(COUNT(city));
```

```
SELECT city, COUNT(city) as "Total Students From " from students GROUP by(city) ORDER by  
(COUNT(city)) desc;
```

GROUP BY Syntax

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

GROUP BY With JOIN Example

```
SELECT students.roll, students.fname, students.lname, COUNT(fees.roll) as "Times",  
sum(fees.amount) as "Total Payment" from students INNER join fees on students.roll = fees.roll  
GROUP by (fees.roll);
```

```
SELECT students.roll, students.fname, students.lname, COUNT(fees.roll) as "Times",  
sum(fees.amount) as "Total Payment" from students INNER join fees on students.roll = fees.roll  
GROUP by (fees.roll) ORDER by sum(fees.amount) desc;
```

MySQL HAVING Clause

The MySQL HAVING Clause

The `HAVING` clause was added to SQL because the `WHERE` keyword cannot be used with aggregate functions.

```
SELECT city, count(roll) from students GROUP BY(city) HAVING COUNT(roll) >= 5;
```

```
SELECT city, count(roll) from students GROUP BY(city) HAVING COUNT(roll) >= 2;
```

```
SELECT students.roll, students.fname, students.lname, COUNT(fees.roll) as "Total Times",  
sum(fees.amount) as "Total Amount" FROM students INNER JOIN fees on students.roll = fees.roll  
GROUP by (fees.roll) HAVING sum(fees.amount) >= 20000 ORDER by (sum(fees.amount)) desc;
```

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

```
SELECT students.roll, students.fname, students.lname, COUNT(fees.roll) as "Total Times",  
sum(fees.amount) as "Total Amount" FROM students INNER JOIN fees on students.roll = fees.roll  
WHERE students.roll = 1 or students.roll = 2 GROUP by (fees.roll) HAVING sum(fees.amount) >=  
20000 ORDER by (sum(fees.amount)) desc;
```

The MySQL EXISTS Operator

The `EXISTS` operator is used to test for the existence of any record in a subquery.

The `EXISTS` operator returns `TRUE` if the subquery returns one or more records.

EXISTS Syntax

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS
(SELECT column_name FROM table_name WHERE condition);
```

```
SELECT * from students WHERE EXISTS (SELECT roll FROM marks WHERE result = 'pass' and
students.roll = marks.roll);
```

MySQL ANY and ALL Operators

The MySQL ANY and ALL Operators

The **ANY** and **ALL** operators allow you to perform a comparison between a single column value and a range of other values.

The ANY Operator

The **ANY** operator:

- returns a boolean value as a result
- returns TRUE if ANY of the subquery values meet the condition

ANY means that the condition will be true if the operation is true for any of the values in the range.

ANY Syntax

```
SELECT column_name(s) FROM table_name WHERE column_name operator ANY (SELECT column_name FROM table_name WHERE condition);
```

```
SELECT * from students
```

```
SELECT * from marks
```

```
SELECT * from marks WHERE result <> 'pass';
```

```
SELECT roll from marks WHERE result <> 'pass';
```

```
SELECT * from students WHERE roll = any (SELECT roll from marks WHERE result <> 'pass');
```

```
SELECT * from students WHERE roll = any (SELECT roll from attendance WHERE presents > 150);
```

Note: The *operator* must be a standard comparison operator (=, <>, !=, >, >=, <, or <=).

The ALL Operator

The **ALL** operator:

- returns a boolean value as a result
- returns TRUE if ALL of the subquery values meet the condition
- is used with **SELECT**, **WHERE** and **HAVING** statements

ALL means that the condition will be true only if the operation is true for all values in the range.

ALL Syntax With SELECT

```
SELECT ALL column_name(s)
FROM table_name
WHERE condition;
```

ALL Syntax With WHERE or HAVING

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ALL
(SELECT column_name
FROM table_name
WHERE condition);
```

Note: The *operator* must be a standard comparison operator (=, <>, !=, >, >=, <, or <=).

The following SQL statement lists the ProductName if ALL the records in the OrderDetails table has Quantity equal to 10. This will of course return FALSE because the Quantity column has many different values (not only the value of 10):

```
SELECT roll, fname, lname from students where roll = all (SELECT roll from attendance WHERE
presents > 100 AND students.roll = attendance.roll);
```

```
SELECT roll,fname,lname from students where roll = all (SELECT roll from attendance WHERE
presents > 150 AND students.roll = attendance.roll);
```

MySQL INSERT INTO SELECT Statement

The MySQL INSERT INTO SELECT Statement

The `INSERT INTO SELECT` statement copies data from one table and inserts it into another table.

The `INSERT INTO SELECT` statement requires that the data types in source and target tables matches.

Note: The existing records in the target table are unaffected.

INSERT INTO SELECT Syntax

Copy all columns from one table to another table:

```
INSERT INTO table2
SELECT * FROM table1
WHERE condition;
```

```
INSERT into students1 SELECT * from students
```

Copy only some columns from one table into another table:

```
INSERT INTO table2 (column1, column2, column3, ...)
SELECT column1, column2, column3, ...
FROM table1
WHERE condition;
```

```
INSERT into students1 (fname, lname, city, gender) SELECT fname, lname, city, gender from students
WHERE city = 'rajkot'
```


MySQL CASE Statement

The MySQL CASE Statement

The `CASE` statement goes through conditions and returns a value when the first condition is met (like an if-then-else statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the `ELSE` clause.

If there is no `ELSE` part and no conditions are true, it returns `NULL`.

```
SELECT roll, fname, lname, city, gender, case when gender = 'male' THEN 'Blue Color' when gender = 'female' THEN 'Pink Color' end as "Favorite Color" from students
```

```
SELECT roll, fname, lname, city, gender, case when gender = 'male' THEN 'Blue Color' when gender = 'female' THEN 'Pink Color' else 'Brown Color' end as "Favorite Color" from students;
```

CASE Syntax

CASE

`WHEN condition1 THEN result1`

`WHEN condition2 THEN result2`

`WHEN conditionN THEN resultN`

`ELSE result`

`END;`

```
SELECT roll, fname, lname, city, gender, case when city = 'rajkot' THEN 'Home Town' when city = 'surat' THEN 'Far From Home' when city = 'ahamdabad' THEN 'Far Home' when city = 'bhuj' THEN 'Kutchh Area' when city = 'jetpur' THEN 'Near by Home' else 'Unknown Distance' end as "Distance From Home" from students;
```

MySQL NULL Functions

MySQL IFNULL() and COALESCE() Functions

```
SELECT roll, absents, presents, (absents+presents) as "Total Working Days" from attendance;
```

MySQL IFNULL() Function

The MySQL [IFNULL\(\)](#) function lets you return an alternative value if an expression is NULL.

```
SELECT roll, absents, presents, (ifnull(absents, 0)+ifnull(presents,0)) as "Total Working Days" from attendance;
```

MySQL COALESCE() Function

Or we can use the [COALESCE\(\)](#) function, like this:

```
SELECT roll, absents, presents, (COALESCE(absents, 0) + COALESCE(presents,0)) as "Total Working Days" from attendance;
```

MySQL Comments

MySQL Comments

Comments are used to explain sections of SQL statements, or to prevent execution of SQL statements.

Single Line Comments

Single line comments start with --.

Any text between -- and the end of the line will be ignored (will not be executed).

The following example uses a single-line comment as an explanation:

```
-- Select all:  
SELECT * FROM Customers;
```

Multi-line Comments

Multi-line comments start with /* and end with */.

Any text between /* and */ will be ignored.

The following example uses a multi-line comment as an explanation:

Example

```
/*Select all the columns  
of all the records  
in the Customers table:*/  
SELECT * FROM Customers;
```

```
SELECT * FROM Customers -- WHERE City='Berlin';
```

```
-- SELECT * FROM Customers;  
SELECT * FROM Products;
```

```
/*SELECT * FROM Customers;  
SELECT * FROM Products;  
SELECT * FROM Orders;  
SELECT * FROM Categories;*/  
SELECT * FROM Suppliers;
```

To ignore just a part of a statement, also use the `/* */` comment.

The following example uses a comment to ignore part of a line:

```
SELECT roll, /*absents, presents,*/ (COALESCE(absents, 0) + COALESCE(presents,0)) as  
"Total Working Days" from attendance;
```

```
SELECT * FROM Customers WHERE (CustomerName LIKE 'L%'  
OR CustomerName LIKE 'R%' /*OR CustomerName LIKE 'S%'  
OR CustomerName LIKE 'T%'*/ OR CustomerName LIKE 'W%')  
AND Country='USA' ORDER BY CustomerName;
```

MySQL Operators

```
SELECT * FROM students WHERE roll = some (SELECT roll FROM marks WHERE result = 'pass');
```

```
SELECT * FROM students WHERE roll = some (SELECT roll FROM marks WHERE result = 'fail');
```

MySQL CREATE DATABASE Statement

The MySQL CREATE DATABASE Statement

The `CREATE DATABASE` statement is used to create a new SQL database.

Syntax

`CREATE DATABASE databasename;`

CREATE DATABASE Example

The following SQL statement creates a database called "testDB":

```
CREATE DATABASE testDB
```

Tip: Make sure you have admin privilege before creating any database. Once a database is created, you can check it in the list of databases with the following SQL command: `SHOW DATABASES;`

```
show DATABASES
```

MySQL DROP DATABASE Statement

The MySQL DROP DATABASE Statement

The `DROP DATABASE` statement is used to drop an existing SQL database.

Syntax

`DROP DATABASE databasename;`

Note: Be careful before dropping a database. Deleting a database will result in loss of complete information stored in the database!

DROP DATABASE Example

The following SQL statement drops the existing database "testDB":

```
show DATABASES;
```

```
drop DATABASE testDB
```

```
show DATABASES;
```

Tip: Make sure you have admin privilege before dropping any database. Once a database is dropped, you can check it in the list of databases with the following SQL command: `SHOW DATABASES;`

MySQL CREATE TABLE Statement

The MySQL CREATE TABLE Statement

The `CREATE TABLE` statement is used to create a new table in a database.

Syntax

```
CREATE TABLE table_name (  
    column1 datatype,  
    column2 datatype,  
    column3 datatype,  
    ....  
);
```

The column parameters specify the names of the columns of the table.

The datatype parameter specifies the type of data the column can hold (e.g. varchar, integer, date, etc.).

Tip: For an overview of the available data types, go to our complete [Data Types Reference](#).

```
create TABLE persons(personid int AUTO_INCREMENT PRIMARY key, fname varchar(20), lname  
varchar(20), city varchar(20), email varchar(20), phone varchar(20), gender varchar(10))
```

```
#1046 - No database selected
```

```
Use 426_2324
```

```
Use your database name
```

```
create TABLE persons(personid int AUTO_INCREMENT PRIMARY key, fname varchar(20), lname  
varchar(20), city varchar(20), email varchar(20), phone varchar(20), gender varchar(10))
```

Create Table Using Another Table

A copy of an existing table can also be created using `CREATE TABLE`.

The new table gets the same column definitions. All columns or specific columns can be selected. If you create a new table using an existing table, the new table will be filled with the existing values from the old table.

Syntax

```
CREATE TABLE new_table_name AS SELECT column1, column2, FROM existing_table_name WHERE  
....;
```

```
CREATE table persons_backup as SELECT * from persons
```


MySQL ALTER TABLE Statement

MySQL ALTER TABLE Statement

The `ALTER TABLE` statement is used to add, delete, or modify columns in an existing table.

The `ALTER TABLE` statement is also used to add and drop various constraints on an existing table.

ALTER TABLE- ADD Column

To add a column in a table, use the following syntax:

```
ALTER TABLE table_name ADD column_name datatype;
```

```
ALTER TABLE persons add COLUMN dateofbirth date
```

ALTER TABLE- DROP COLUMN

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

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

```
ALTER TABLE persons DROP COLUMN dateofbirth
```

ALTER TABLE- MODIFY COLUMN

To change the data type of a column in a table, use the following syntax:

```
ALTER TABLE table_name MODIFY COLUMN column_name datatype;
```

```
ALTER table persons MODIFY COLUMN email varchar(128)
```

MySQL DROP TABLE Statement

The MySQL DROP TABLE Statement

The `DROP TABLE` statement is used to drop an existing table in a database.

Syntax

DROP TABLE *table_name*;

Note: Be careful before dropping a table. Deleting a table will result in loss of complete information stored in the table!

MySQL DROP TABLE Example

drop TABLE persons

MySQL TRUNCATE TABLE

The `TRUNCATE TABLE` statement is used to delete the data inside a table, but not the table itself.(reset auto increment to 1)

Syntax

TRUNCATE TABLE *table_name*;

MySQL Constraints

SQL constraints are used to specify rules for data in a table.

Create 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.

Syntax

```
CREATE TABLE table_name (  
    column1 datatype constraint,  
    column2 datatype constraint,  
    column3 datatype constraint,  
    ....  
);
```

MySQL Constraints

SQL constraints are used to specify rules for the data in a table.

Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

[NOT NULL](#) - Ensures that a column cannot have a NULL value

[UNIQUE](#) - Ensures that all values in a column are different

[PRIMARY KEY](#) - A combination of a `NOT NULL` and `UNIQUE`. Uniquely identifies each row in a table

[FOREIGN KEY](#) - Prevents actions that would destroy links between tables

[CHECK](#) - Ensures that the values in a column satisfies a specific condition

[DEFAULT](#) - Sets a default value for a column if no value is specified

[CREATE INDEX](#) - Used to create and retrieve data from the database very quickly

MySQL NOT NULL Constraint

MySQL NOT NULL Constraint

By default, a column can hold NULL values.

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

This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

NOT NULL on CREATE TABLE

```
CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255) NOT NULL,  
    Age int  
);
```

NOT NULL on ALTER TABLE

```
ALTER TABLE Persons  
MODIFY Age int NOT NULL;
```

```
CREATE TABLE persons (personid int, fname varchar(20), lname varchar(20), city varchar(20))
```

```
INSERT into persons (personid, fname, lname, city) values (1, 'Jeel', 'Ghodasara', 'Rajkot')
```

```
INSERT into persons (personid, fname, lname, city) values (2, 'Jeel', null, 'Rajkot')
```

```
INSERT into persons (personid, fname, lname, city) values (3, null, null, 'Rajkot')
```

```
select * from persons
```

```
drop table persons
```

```
CREATE TABLE persons (personid int not null, fname varchar(20) not null, lname varchar(20) not null,  
city varchar(20) not null)
```

```
INSERT into persons (personid, fname, lname, city) values (1, 'Jeel', 'Ghodasara', 'Rajkot')
```

```
select * from persons
```

```
INSERT into persons (personid, fname, lname, city) values (2, 'Jeel', null, 'Rajkot')
```

```
#1048 - Column 'lname' cannot be null
```

```
alter TABLE persons add COLUMN age int
```

```
INSERT into persons (personid, fname, lname, city, age) values (1, 'Jeel', 'Ghodasara', 'Rajkot', null)
```

```
ALTER TABLE persons MODIFY COLUMN age int NOT null
```

```
INSERT into persons (personid, fname, lname, city, age) values (1, 'Jeel', 'Ghodasara', 'Rajkot', null)
```

```
#1048 - Column 'age' cannot be null
```

MySQL UNIQUE Constraint

MySQL UNIQUE Constraint

The `UNIQUE` constraint ensures that all values in a column are different.

Both the `UNIQUE` and `PRIMARY KEY` constraints provide a guarantee for uniqueness for a column or set of columns.

A `PRIMARY KEY` constraint automatically has a `UNIQUE` constraint.

However, you can have many `UNIQUE` constraints per table, but only one `PRIMARY KEY` constraint per table.

Unique constraint allows null values in column

UNIQUE Constraint on CREATE TABLE

The following SQL creates a `UNIQUE` constraint on the "ID" column when the "Persons" table is created:

```
CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    UNIQUE (ID)  
);  
  
drop table persons;  
  
create table persons (personid int, fname varchar(20), lname varchar(20), city varchar(20),  
UNIQUE(personid))  
  
INSERT into persons (personid, fname, lname, city) values (1, 'Jeel', 'Ghodasara', 'Rajkot')  
  
Select * from persons  
  
INSERT into persons (personid, fname, lname, city) values (1, 'Jeel', 'Ghodasara', 'Rajkot')  
  
#1062 - Duplicate entry '1' for key 'personid'  
  
INSERT into persons (personid, fname, lname, city) values (2, 'Jeel', 'Ghodasara', 'Rajkot');  
  
INSERT into persons (personid, fname, lname, city) values (null, 'Jeel', 'Ghodasara', 'Rajkot');  
  
INSERT into persons (personid, fname, lname, city) values (null, 'Jeel', 'Ghodasara', 'Rajkot');  
  
INSERT into persons (personid, fname, lname, city) values (null, 'Jeel', 'Ghodasara', 'Rajkot');  
  
drop TABLE persons
```

To name a `UNIQUE` constraint, and to define a `UNIQUE` constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE persons (personid int, fname varchar(20), lname varchar(20), city varchar(20),  
CONSTRAINT unq_pid UNIQUE (personid))
```

```
INSERT into persons (personid, fname, lname, city) values (1, 'Jeel', 'Ghodasara', 'Rajkot');
```

```
INSERT into persons (personid, fname, lname, city) values (1, 'Jeel', 'Ghodasara', 'Rajkot');
```

```
#1062 - Duplicate entry '1' for key 'unq_pid'
```

DROP a `UNIQUE` Constraint

To drop a `UNIQUE` constraint, use the following SQL:

```
ALTER TABLE persons drop CONSTRAINT unq_pid
```

```
INSERT into persons (personid, fname, lname, city) values (1, 'Jeel', 'Ghodasara', 'Rajkot');
```

```
INSERT into persons (personid, fname, lname, city) values (1, 'Jeel', 'Ghodasara', 'Rajkot');
```

```
INSERT into persons (personid, fname, lname, city) values (1, 'Jeel', 'Ghodasara', 'Rajkot');
```

`UNIQUE` Constraint on ALTER TABLE

```
TRUNCATE TABLE persons
```

```
ALTER TABLE persons add CONSTRAINT unq_pid UNIQUE(personid)
```

```
INSERT into persons (personid, fname, lname, city) values (1, 'Jeel', 'Ghodasara', 'Rajkot');
```

```
INSERT into persons (personid, fname, lname, city) values (1, 'Jeel', 'Ghodasara', 'Rajkot');
```

`UNIQUE` Constraint on Multiple Columns TABLE

```
create table listofvillages (villageid int AUTO_INCREMENT PRIMARY key, villagename varchar(20),  
taluka varchar(20), distrcit varchar(20), CONSTRAINT unq_village_taluka UNIQUE(villagename,  
taluka))
```

```
INSERT into listofvillages (villagename, taluka, distrcit) VALUES('Navagam', 'Rajkot', 'Rajkot')
```

```
INSERT into listofvillages (villagename, taluka, distrcit) VALUES('Navagam', 'Gondal', 'Rajkot');
```

```
INSERT into listofvillages (villagename, taluka, distrcit) VALUES('Navagam', 'Gondal', 'Rajkot');
```

```
#1062 - Duplicate entry 'Navagam-Gondal' for key 'unq_village_taluka'
```

```
INSERT into listofvillages (villagename, taluka, distrcit) VALUES('Madhapar', 'Rajkot', 'Rajkot');
```

```
INSERT into listofvillages (villagename, taluka, distrcit) VALUES('Madhapar', 'Bhuj', 'Kutchh');
```

```
SELECT * from listofvillages
```

MySQL PRIMARY KEY Constraint

MySQL PRIMARY KEY Constraint

The `PRIMARY KEY` constraint uniquely identifies each record in a table.

Primary keys must contain `UNIQUE` values, and cannot contain `NULL` values.

A table can have only **ONE** primary key; and in the table, this primary key can consist of single or multiple columns (fields).

PRIMARY KEY on CREATE TABLE

The following SQL creates a `PRIMARY KEY` on the "ID" column when the "Persons" table is created:

```
drop table persons
```

```
create TABLE persons (personid int PRIMARY KEY, fname varchar(20), lname varchar(20), city varchar(20))
```

```
INSERT into persons (personid, fname, lname, city) values (1, 'Jeel', 'Ghodasara', 'Rajkot');
```

```
INSERT into persons (personid, fname, lname, city) values (1, 'Jeel', 'Ghodasara', 'Rajkot');
```

```
#1062 - Duplicate entry '1' for key 'PRIMARY'
```

To allow naming of a `PRIMARY KEY` constraint, and for defining a `PRIMARY KEY` constraint on multiple columns, use the following SQL syntax

```
drop table persons
```

```
create TABLE persons (personid int, fname varchar(20), lname varchar(20), city varchar(20),  
CONSTRAINT pri_key_pid PRIMARY KEY (personid))
```

```
INSERT into persons (personid, fname, lname, city) values (1, 'Jeel', 'Ghodasara', 'Rajkot');
```

```
INSERT into persons (personid, fname, lname, city) values (1, 'Jeel', 'Ghodasara', 'Rajkot');
```

PRIMARY KEY on ALTER TABLE

To create a `PRIMARY KEY` constraint on the "ID" column when the table is already created, use the following SQL:

```
drop table persons
```

```
create TABLE persons (personid int, fname varchar(20), lname varchar(20), city varchar(20))
```

```
ALTER TABLE persons add PRIMARY key (personid)
```


DROP a PRIMARY KEY Constraint

To drop a `PRIMARY KEY` constraint, use the following SQL:

```
ALTER TABLE Persons  
DROP PRIMARY KEY;
```

```
ALTER TABLE persons drop PRIMARY key;
```

To allow naming of a `PRIMARY KEY` constraint, and for defining a `PRIMARY KEY` constraint on multiple columns, use the following SQL syntax:

```
ALTER TABLE Persons  
ADD CONSTRAINT PK_Person PRIMARY KEY (ID,LastName);
```

Note: If you use `ALTER TABLE` to add a primary key, the primary key column(s) must have been declared to not contain `NULL` values (when the table was first created).

```
ALTER TABLE persons add CONSTRAINT pri_key_pid_fname PRIMARY key (personid,  
fname)
```

```
CREATE TABLE students (roll int AUTO_INCREMENT PRIMARY KEY, fname varchar(20), lname  
varchar(20), city varchar(20), phone varchar(20), email varchar(20), gender varchar(10))
```

```
INSERT into students (fname, lname, city, phone, email, gender) values ('priyajitsinh', 'jadeja', 'rajkot',  
'9988990000', 'jadeja@gmail.com', 'male'),('smit', 'vanzara', 'rajkot', '9988991100',  
'smit@gmail.com', 'male'),('aryan', 'thakrar', 'rajkot', '9988911000', 'aryan@gmail.com',  
'male'),('hemal', 'varu', 'rajkot', '9228990000', 'hemal@gmail.com', 'female'),('dhruvisha', 'bhatt',  
'rajkot', '9228990000', 'bhatt@gmail.com', 'female'),('devangi', 'dave', 'rajkot', '3388990000',  
'devangi@gmail.com', 'female'),('hensi', 'lambiya', 'rajkot', '7788990000', 'hensi@gmail.com',  
'female')
```

```
select * from students
```

MySQL FOREIGN KEY Constraint

MySQL FOREIGN KEY Constraint

The `FOREIGN KEY` constraint is used to prevent actions that would destroy links between tables.

A `FOREIGN KEY` is a field (or collection of fields) in one table, that refers to the [PRIMARY KEY](#) in another table.

The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

Look at the following two tables:

Persons Table

PersonID (Primary key)	LastName	FirstName	Age
1	Hansen	Ola	30
2	Svendson	Tove	23
3	Pettersen	Kari	20

Orders Table

OrderID	OrderNumber	PersonID (foreign key)
1	77895	3
2	44678	3
3	22456	2
4	24562	1

Notice that the "PersonID" column in the "Orders" table points to the "PersonID" column in the "Persons" table.

The "PersonID" column in the "Persons" table is the `PRIMARY KEY` in the "Persons" table.

The "PersonID" column in the "Orders" table is a `FOREIGN KEY` in the "Orders" table.

The `FOREIGN KEY` constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

FOREIGN KEY on CREATE TABLE

create table attendance (attendance_id int AUTO_INCREMENT PRIMARY key, roll int, absents int, present int, FOREIGN key(roll) REFERENCES students(roll))

```
INSERT INTO `attendance` (`attendance_id`, `roll`, `absents`, `present`) VALUES (NULL, '5', '112', '112');
```

```
INSERT INTO `attendance` (`attendance_id`, `roll`, `absents`, `present`) VALUES (NULL, '15', '112', '112');
```

```
#1452 - Cannot add or update a child row: a foreign key constraint fails
(`426_2324`.`attendance`, CONSTRAINT `attendance_ibfk_1` FOREIGN KEY
(`roll`) REFERENCES `students` (`roll`))
```

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE exams (examid int AUTO_INCREMENT PRIMARY KEY, roll int, examdate date, subject
varchar(20), marks int, result varchar(20), CONSTRAINT fk_student_exam FOREIGN KEY (roll)
REFERENCES students(roll))
```

```
INSERT INTO `exams` (`examid`, `roll`, `examdate`, `subject`, `marks`, `result`) VALUES (NULL, '5',
'2024-04-01', 'MySql', '123', 'fail');
```

```
INSERT INTO `exams` (`examid`, `roll`, `examdate`, `subject`, `marks`, `result`) VALUES (NULL, '15',
'2024-04-01', 'MySql', '123', 'fail');
```

```
#1452 - Cannot add or update a child row: a foreign key constraint fails
(`426_2324`.`exams`, CONSTRAINT `fk_student_exam` FOREIGN KEY (`roll`)
REFERENCES `students` (`roll`))
```

FOREIGN KEY on ALTER TABLE

create TABLE fees (feesid int AUTO_INCREMENT PRIMARY KEY, roll int, paymentdate date, amount decimal(10,2), paymentmode varchar(20), referenceid varchar(20))

```
ALTER TABLE fees add CONSTRAINT fk_student_fees_roll FOREIGN KEY(roll) REFERENCES
students(roll)
```

DROP a FOREIGN KEY Constraint

To drop a FOREIGN KEY constraint, use the following SQL:

```
ALTER TABLE fees drop CONSTRAINT fk_student_fees_roll
```

```
FROM students WHERE `students`.`roll` = 5"
```

You cannot delete data from primary table until single row present with same reference in child table

MySQL CHECK Constraint

MySQL CHECK Constraint

The `CHECK` constraint is used to limit the value range that can be placed in a column.

If you define a `CHECK` constraint on a column it will allow only certain values for this column.

If you define a `CHECK` constraint on a table it can limit the values in certain columns based on values in other columns in the row.

CHECK on CREATE TABLE

`CREATE TABLE` Persons (

 ID int `NOT NULL`,

 LastName varchar(255) `NOT NULL`,

 FirstName varchar(255),

 Age int,

`CHECK` (Age>=18)

);

`INSERT` into persons (id, lastname, firstname, age) values (1,'smit', 'vanzara', 17)

#4025 - CONSTRAINT `CONSTRAINT_1` failed for `426_2324`.`persons`

`INSERT` into persons (id, lastname, firstname, age) values (1,'smit', 'vanzara', 27)

CHECK on ALTER TABLE

`ALTER TABLE` students add CONSTRAINT check_gender CHECK (gender = 'male' or gender = 'female')

`INSERT INTO` students (fname, lname, city, email, phone, gender) VALUES ('Demo', 'text', 'morbi', 'demo@gmail.com', '9900887766', 'fenal')

#4025 - CONSTRAINT `check_gender` failed for `426_2324`.`students`

DROP a CHECK Constraint

To drop a `CHECK` constraint, use the following SQL:

`ALTER TABLE` students drop CONSTRAINT check_gender

MySQL DEFAULT Constraint

MySQL DEFAULT Constraint

The `DEFAULT` constraint is used to set a default value for a column.

The default value will be added to all new records, if no other value is specified.

DEFAULT on CREATE TABLE

```
CREATE TABLE students_address (addressid int AUTO_INCREMENT PRIMARY KEY, roll int, line1  
varchar(40), line2 varchar(40), landmark varchar(40), city varchar(20), state varchar(20) DEFAULT  
'Gujarat', country varchar(20) DEFAULT 'india')
```

DEFAULT on ALTER TABLE

```
ALTER TABLE students add COLUMN admission_date timestamp DEFAULT CURRENT_TIMESTAMP
```

```
INSERT INTO `students` (`roll`, `fname`, `lname`, `city`, `phone`, `email`, `gender`, `admission_date`)  
VALUES (NULL, 'Another ', 'Example', 'of', '9988990088', 'demo@msil.com', 'male',  
current_timestamp());
```

DROP a DEFAULT Constraint

To drop a `DEFAULT` constraint, use the following SQL:

```
ALTER TABLE students_address ALTER STATE drop DEFAULT
```

MySQL CREATE INDEX Statement

MySQL CREATE INDEX Statement

The `CREATE INDEX` statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

Note: Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

CREATE INDEX Syntax

Creates an index on a table. Duplicate values are allowed:

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);

CREATE INDEX students_fname on students(fname)
```

CREATE UNIQUE INDEX Syntax

Creates a unique index on a table. Duplicate values are not allowed:

```
CREATE UNIQUE INDEX index_name
ON table_name (column1, column2, ...);

CREATE unique INDEX students_fname1 on students(fname);
```

DROP INDEX Statement

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

```
ALTER TABLE table_name
DROP INDEX index_name;

drop INDEX students_fname1 on students;

ALTER TABLE students DROP index students_fname;
```

MySQL AUTO INCREMENT Field

What is an AUTO INCREMENT Field?

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.

Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

MySQL AUTO_INCREMENT Keyword

MySQL uses the `AUTO_INCREMENT` keyword to perform an auto-increment feature.

By default, the starting value for `AUTO_INCREMENT` is 1, and it will increment by 1 for each new record.

To let the `AUTO_INCREMENT` sequence start with another value, use the following SQL statement:

```
ALTER TABLE Persons AUTO_INCREMENT=100;
```

```
ALTER TABLE students AUTO_INCREMENT = 200
```


MySQL Working with Dates

MySQL Dates

The most difficult part when working with dates is to be sure that the format of the date you are trying to insert, matches the format of the date column in the database.

As long as your data contains only the date portion, your queries will work as expected. However, if a time portion is involved, it gets more complicated.

MySQL Date Data Types

MySQL comes with the following data types for storing a date or a date/time value in the database:

DATE - format YYYY-MM-DD
DATETIME - format: YYYY-MM-DD HH:MI:SS
TIMESTAMP - format: YYYY-MM-DD HH:MI:SS
YEAR - format YYYY or YY

Note: The date data type are set for a column when you create a new table in your database!

```
SELECT * from students WHERE dateofbirth >= '2002-12-15'
```

```
SELECT * from students WHERE dateofbirth >= '2006-01-01';
```

```
SELECT * from students WHERE dateofbirth <= '2006-01-01';
```

Note: Two dates can easily be compared if there is no time component involved!

```
SELECT * from students WHERE admission_date = '2024-04-05'
```

```
SELECT * from students WHERE admission_date like '2024-04-05%'
```

```
SELECT * from students WHERE date(admission_date) = '2024-04-05';
```

```
SELECT * from students WHERE not date(admission_date) = '2024-04-05';
```

Tip: To keep your queries simple and easy to maintain, do not use time-components in your dates, unless you have to!

MySQL Views

MySQL CREATE VIEW Statement

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.

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

CREATE VIEW Syntax

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

```
create view getStudents as SELECT students.roll, students.fname, students.lname, students.city,  
students.phone, students.email, students.dateofbirth, students.gender, attendance.absents,  
attendance.present from students inner join attendance on students.roll = attendance.roll;
```

```
SELECT * from getstudents;  
SELECT * FROM `getstudents` WHERE roll = 2;
```

Note: A view always shows up-to-date data! The database engine recreates the view, every time a user queries it.

MySQL Updating a View

A view can be updated with the `CREATE OR REPLACE VIEW` statement.

CREATE OR REPLACE VIEW Syntax

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

```
create or replace view getStudents as SELECT students.roll, students.fname, students.lname,  
students.city, students.phone, students.email, students.dateofbirth, students.gender,  
attendance.absents, attendance.present from students inner join attendance on students.roll =  
attendance.roll;
```

```
create or replace view getStudents as SELECT students.roll, students.fname, students.lname,  
students.city, students.phone, students.email, students.dateofbirth, students.gender,  
attendance.absents, attendance.present, fees.paymentdate, fees.amount, fees.paymentmode from
```

students inner join attendance on students.roll = attendance.roll INNER join fees on students.roll = fees.roll;

SELECT * from getstudents

MySQL Dropping a View

A view is deleted with the `DROP VIEW` statement.

DROP VIEW Syntax

`DROP VIEW` *view_name*;

drop view getstudents;

SQL Commands

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

Types of SQL Commands

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

DDL	•Create, Drop, Alter, Truncate
DML	•Insert, Update, Delete
DCL	•Grant, Revoke
TCL	•Commit, Rollback, Save Point
DQL	•Select

Data Definition Language (DDL)

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

Here are some commands that come under DDL:

- CREATE
- ALTER
- DROP
- TRUNCATE

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

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

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

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

Data Manipulation Language

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

Here are some commands that come under DML:

- INSERT
- UPDATE
- DELETE

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

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

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

Data Control Language

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

Here are some commands that come under DCL:

- Grant
- Revoke

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

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

Transaction Control Language

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

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

Here are some commands that come under TCL:

- COMMIT
- ROLLBACK
- SAVEPOINT

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

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

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

Data Query Language

DQL is used to fetch the data from the database.

It uses only one command:

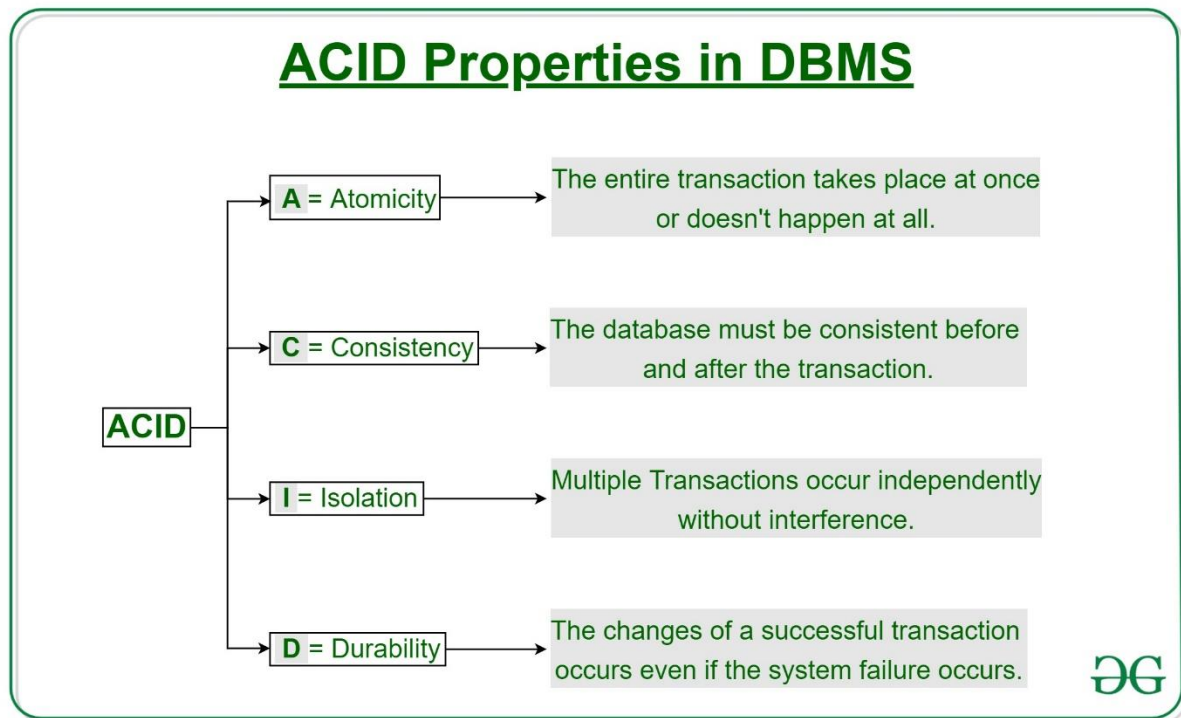
- SELECT

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

ACID Properties in DBMS

A **transaction** is a single logical unit of work that accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.

In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.



Atomicity:

By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway ie. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.

- Abort**: If a transaction aborts, changes made to the database are not visible.
- Commit**: If a transaction commits, changes made are visible.

Atomicity is also known as the 'All or nothing rule'.

Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.

Before: X : 500	Y: 200
Transaction T	
T1	T2
Read (X)	Read (Y)
X: = X - 100	Y: = Y + 100
Write (X)	Write (Y)
After: X : 400	Y : 300

If the transaction fails after completion of **T1** but before completion of **T2**. (say, after **write(X)** but before **write(Y)**), then the amount has been deducted from **X** but not added to **Y**. This results in an **inconsistent database state**. Therefore, the transaction must be executed in its entirety in order to ensure the correctness of the database state.

Consistency:

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above, The total amount before and after the transaction must be maintained.

Total **before T** occurs = **500 + 200 = 700**. Total **after T occurs** = **400 + 300 = 700**.

Therefore, the database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result, T is incomplete.

Isolation:

This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.

Let **X= 500, Y = 500**.

Consider two transactions **T** and **T''**.

T	T''
Read (X)	Read (X)
X: = X*100	Read (Y)
Write (X)	Z: = X + Y
Read (Y)	Write (Z)
Y: = Y - 50	
Write (Y)	

Suppose **T** has been executed till **Read (Y)** and then **T''** starts. As a result, interleaving of operations takes place due to which **T''** reads the correct value of **X** but the incorrect value of **Y** and sum computed

by

T'': (X+Y = 50, 000+500=50, 500)

is thus not consistent with the sum at end of the transaction:

T: (X+Y = 50, 000 + 450 = 50, 450)

by

T'': (X+Y = 50, 000+500=50, 500)

is thus not consistent with the sum at end of the transaction:

T: (X+Y = 50, 000 + 450 = 50, 450)

Durability:

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system

failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

Some important points:

Property	Responsibility for maintaining properties
----------	---

Atomicity	Transaction Manager
-----------	---------------------

Consistency	Application programmer
-------------	------------------------

Isolation	Concurrency Control Manager
-----------	-----------------------------

Durability	Recovery Manager
------------	------------------

The **ACID** properties, in totality, provide a mechanism to ensure the correctness and consistency of a database in a way such that each transaction is a group of operations that acts as a single unit, produces consistent results, acts in isolation from other operations, and updates that it makes are durably stored.

ACID properties are the four key characteristics that define the reliability and consistency of a transaction in a Database Management System (DBMS). The acronym ACID stands for Atomicity, Consistency, Isolation, and Durability. Here is a brief description of each of these properties:

1. **Atomicity:** Atomicity ensures that a transaction is treated as a single, indivisible unit of work. Either all the operations within the transaction are completed successfully, or none of them are. If any part of the transaction fails, the entire transaction is rolled back to its original state, ensuring data consistency and integrity.
2. **Consistency:** Consistency ensures that a transaction takes the database from one consistent state to another consistent state. The database is in a consistent state both before and after the transaction is executed. Constraints, such as unique keys and foreign keys, must be maintained to ensure data consistency.
3. **Isolation:** Isolation ensures that multiple transactions can execute concurrently without interfering with each other. Each transaction must be isolated from other transactions until it is completed. This isolation prevents dirty reads, non-repeatable reads, and phantom reads.
4. **Durability:** Durability ensures that once a transaction is committed, its changes are permanent and will survive any subsequent system failures. The transaction's changes are saved to the database permanently, and even if the system crashes, the changes remain intact and can be recovered.

Overall, ACID properties provide a framework for ensuring data consistency, integrity, and reliability in DBMS. They ensure that transactions are executed in a reliable and consistent manner, even in the presence of system failures, network issues, or other problems. These properties make DBMS a reliable and efficient tool for managing data in modern organizations.

Advantages of ACID Properties in DBMS:

1. **Data Consistency:** ACID properties ensure that the data remains consistent and accurate after any transaction execution.
2. **Data Integrity:** ACID properties maintain the integrity of the data by ensuring that any changes to the database are permanent and cannot be lost.
3. **Concurrency Control:** ACID properties help to manage multiple transactions occurring concurrently by preventing interference between them.
4. **Recovery:** ACID properties ensure that in case of any failure or crash, the system can recover the data up to the point of failure or crash.

Disadvantages of ACID Properties in DBMS:

1. **Performance:** The ACID properties can cause a performance overhead in the system, as they require additional processing to ensure data consistency and integrity.
2. **Scalability:** The ACID properties may cause scalability issues in large distributed systems where multiple transactions occur concurrently.
3. **Complexity:** Implementing the ACID properties can increase the complexity of the system and require significant **expertise** and resources.

Overall, the advantages of ACID properties in DBMS outweigh the disadvantages. They provide a reliable and consistent approach to data management, ensuring data integrity, accuracy, and reliability. However, in some cases, the overhead of implementing ACID properties can cause performance and scalability issues. Therefore, it's important to balance the benefits of ACID properties against the specific needs and requirements of the system.

MySQL Data Types

The data type of a column defines what value the column can hold: integer, character, money, date and time, binary, and so on.

MySQL Data Types (Version 8.0)

Each column in a database table is required to have a name and a data type.

An SQL developer must decide what type of data that will be stored inside each column when creating a table. The data type is a guideline for SQL to understand what type of data is expected inside of each column, and it also identifies how SQL will interact with the stored data.

In MySQL there are three main data types: string, numeric, and date and time.

String Data Types

Data type	Description
CHAR(size)	A FIXED length string (can contain letters, numbers, and special characters). The <i>size</i> parameter specifies the column length in characters - can be from 0 to 255. Default is 1
VARCHAR(size)	A VARIABLE length string (can contain letters, numbers, and special characters). The <i>size</i> parameter specifies the maximum column length in characters - can be from 0 to 65535
BINARY(size)	Equal to CHAR(), but stores binary byte strings. The <i>size</i> parameter specifies the column length in bytes. Default is 1
VARBINARY(size)	Equal to VARCHAR(), but stores binary byte strings. The <i>size</i> parameter specifies the maximum column length in bytes.
TINYBLOB	For BLOBs (Binary Large Objects). Max length: 255 bytes
TINYTEXT	Holds a string with a maximum length of 255 characters
TEXT(size)	Holds a string with a maximum length of 65,535 bytes
BLOB(size)	For BLOBs (Binary Large Objects). Holds up to 65,535 bytes of data
MEDIUMTEXT	Holds a string with a maximum length of 16,777,215 characters
MEDIUMBLOB	For BLOBs (Binary Large Objects). Holds up to 16,777,215 bytes of data
LONGTEXT	Holds a string with a maximum length of 4,294,967,295 characters

LONGBLOB	For BLOBs (Binary Large Objects). Holds up to 4,294,967,295 bytes of data
ENUM(val1, val2, val3, ...)	A string object that can have only one value, chosen from a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted. The values are sorted in the order you enter them
SET(val1, val2, val3, ...)	A string object that can have 0 or more values, chosen from a list of possible values. You can list up to 64 values in a SET list

Numeric Data Types

Data type	Description
BIT(<i>size</i>)	A bit-value type. The number of bits per value is specified in <i>size</i> . The <i>size</i> parameter can hold a value from 1 to 64. The default value for <i>size</i> is 1.
TINYINT(<i>size</i>)	A very small integer. Signed range is from -128 to 127. Unsigned range is from 0 to 255. The <i>size</i> parameter specifies the maximum display width (which is 255)
BOOL	Zero is considered as false, nonzero values are considered as true.
BOOLEAN	Equal to BOOL
SMALLINT(<i>size</i>)	A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The <i>size</i> parameter specifies the maximum display width (which is 255)
MEDIUMINT(<i>size</i>)	A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The <i>size</i> parameter specifies the maximum display width (which is 255)
INT(<i>size</i>)	A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The <i>size</i> parameter specifies the maximum display width (which is 255)
INTEGER(<i>size</i>)	Equal to INT(<i>size</i>)
BIGINT(<i>size</i>)	A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The <i>size</i> parameter specifies the maximum display width (which is 255)
FLOAT(<i>size</i> , <i>d</i>)	A floating point number. The total number of digits is specified in <i>size</i> . The number of digits after the decimal point is specified in the <i>d</i>

parameter. This syntax is deprecated in MySQL 8.0.17, and it will be removed in future MySQL versions

FLOAT(*p*)
A floating point number. MySQL uses the *p* value to determine whether to use FLOAT or DOUBLE for the resulting data type. If *p* is from 0 to 24, the data type becomes FLOAT(). If *p* is from 25 to 53, the data type becomes DOUBLE()

DOUBLE(*size, d*)
A normal-size floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter

DOUBLE PRECISION(*size, d*)

DECIMAL(*size, d*)
An exact fixed-point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. The maximum number for *size* is 65. The maximum number for *d* is 30. The default value for *size* is 10. The default value for *d* is 0.

DEC(*size, d*)
Equal to DECIMAL(*size, d*)

Date and Time Data Types

Data type	Description
DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'
DATETIME(<i>fsp</i>)	A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time
TIMESTAMP(<i>fsp</i>)	A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT_TIMESTAMP and ON UPDATE CURRENT_TIMESTAMP in the column definition
TIME(<i>fsp</i>)	A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'
YEAR	A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000. MySQL 8.0 does not support year in two-digit format.

MySQL String Functions

Function	Description
<u>ASCII</u>	Returns the ASCII value for the specific character
<u>CHAR_LENGTH</u>	Returns the length of a string (in characters)
<u>CHARACTER_LENGTH</u>	Returns the length of a string (in characters)
<u>CONCAT</u>	Adds two or more expressions together
<u>CONCAT_WS</u>	Adds two or more expressions together with a separator
<u>FIELD</u>	Returns the index position of a value in a list of values
<u>FIND_IN_SET</u>	Returns the position of a string within a list of strings
<u>FORMAT</u>	Formats a number to a format like "#,###,###.##", rounded to a specified number of decimal places
<u>INSERT</u>	Inserts a string within a string at the specified position and for a certain number of characters
<u>INSTR</u>	Returns the position of the first occurrence of a string in another string
<u>LCASE</u>	Converts a string to lower-case
<u>LEFT</u>	Extracts a number of characters from a string (starting from left)
<u>LENGTH</u>	Returns the length of a string (in bytes)
<u>LOCATE</u>	Returns the position of the first occurrence of a substring in a string
<u>LOWER</u>	Converts a string to lower-case
<u>LPAD</u>	Left-pads a string with another string, to a certain length
<u>LTRIM</u>	Removes leading spaces from a string
<u>MID</u>	Extracts a substring from a string (starting at any position)
<u>POSITION</u>	Returns the position of the first occurrence of a substring in a string
<u>REPEAT</u>	Repeats a string as many times as specified
<u>REPLACE</u>	Replaces all occurrences of a substring within a string, with a new substring
<u>REVERSE</u>	Reverses a string and returns the result
<u>RIGHT</u>	Extracts a number of characters from a string (starting from right)

<u>RPAD</u>	Right-pads a string with another string, to a certain length
<u>RTRIM</u>	Removes trailing spaces from a string
<u>SPACE</u>	Returns a string of the specified number of space characters
<u>STRCMP</u>	Compares two strings
<u>SUBSTR</u>	Extracts a substring from a string (starting at any position)
<u>SUBSTRING</u>	Extracts a substring from a string (starting at any position)
<u>SUBSTRING_INDEX</u>	Returns a substring of a string before a specified number of delimiter occurs
<u>TRIM</u>	Removes leading and trailing spaces from a string
<u>UCASE</u>	Converts a string to upper-case
<u>UPPER</u>	Converts a string to upper-case

```
SELECT ascii('A');
```

```
SELECT fname, char_length(fname) FROM students
```

```
SELECT fname, character_length(fname) FROM students;
```

```
SELECT roll, concat(fname, " ", lname ) as "Full Name" FROM students;
```

```
SELECT roll, concat_ws(" _ ", fname, lname, city ) as "Full Name" FROM students;
```

```
SELECT FIELD("q", "s", "q", "l");
```

```
SELECT FIND_IN_SET("q", "s,q,l");
```

```
SELECT FORMAT(250500.5634, 2);
```

```
SELECT INSERT("W3Schools.com", 1, 9, "Example");
```

```
SELECT INSTR("W3Schools.com", "3") AS MatchPosition;
```

```
SELECT LCASE("SQL Tutorial is FUN!");
```

```
SELECT LEFT("SQL Tutorial", 3) AS ExtractString;
```

```
SELECT RIGHT("SQL Tutorial", 3) AS ExtractString;
```

```
SELECT LENGTH("SQL Tutorial") AS LengthOfString;
```

```
SELECT LOCATE("3", "W3Schools.com") AS MatchPosition;

SELECT LOWER("SQL Tutorial is FUN!");

SELECT LPAD("SQL Tutorial", 20, "*");

SELECT RPAD("SQL Tutorial", 20, "*");

SELECT LTRIM("  SQL Tutorial  ") AS LeftTrimmedString;

SELECT RTRIM("  SQL Tutorial  ") AS LeftTrimmedString;

SELECT TRIM("  SQL Tutorial  ") AS LeftTrimmedString;

SELECT MID("SQL Tutorial", 5, 3) AS ExtractString;

SELECT POSITION("3" IN "W3Schools.com") AS MatchPosition;

SELECT REPEAT("SQL Tutorial", 3);

SELECT REPLACE("SQL Tutorial", "SQL", "HTML");

SELECT REVERSE("SQL Tutorial");

SELECT SPACE(50);

SELECT STRCMP("SQL Tutorial", "SQL Tutorial");

SELECT SUBSTR("SQL Tutorial", 5, 3) AS ExtractString;

SELECT SUBSTRING("SQL Tutorial", 5, 3) AS ExtractString;

SELECT SUBSTRING_INDEX("www.w3schools.com", ".", 1);

SELECT SUBSTRING_INDEX("www.w3schools.com", ".", 2);

SELECT ucase("www.w3schools.com");

SELECT upper("www.w3schools.com");
```


MySQL Numeric Functions

Function	Description
<u>ABS</u>	Returns the absolute value of a number
<u>ACOS</u>	Returns the arc cosine of a number
<u>ASIN</u>	Returns the arc sine of a number
<u>ATAN</u>	Returns the arc tangent of one or two numbers
<u>ATAN2</u>	Returns the arc tangent of two numbers
<u>AVG</u>	Returns the average value of an expression
<u>CEIL</u>	Returns the smallest integer value that is \geq to a number
<u>CEILING</u>	Returns the smallest integer value that is \geq to a number
<u>COS</u>	Returns the cosine of a number
<u>COT</u>	Returns the cotangent of a number
<u>COUNT</u>	Returns the number of records returned by a select query
<u>DEGREES</u>	Converts a value in radians to degrees
<u>DIV</u>	Used for integer division
<u>EXP</u>	Returns e raised to the power of a specified number
<u>FLOOR</u>	Returns the largest integer value that is \leq to a number
<u>GREATEST</u>	Returns the greatest value of the list of arguments
<u>LEAST</u>	Returns the smallest value of the list of arguments
<u>LN</u>	Returns the natural logarithm of a number
<u>LOG</u>	Returns the natural logarithm of a number, or the logarithm of a number to a specified base
<u>LOG10</u>	Returns the natural logarithm of a number to base 10
<u>LOG2</u>	Returns the natural logarithm of a number to base 2
<u>MAX</u>	Returns the maximum value in a set of values
<u>MIN</u>	Returns the minimum value in a set of values
<u>MOD</u>	Returns the remainder of a number divided by another number
<u>PI</u>	Returns the value of PI

<u>POW</u>	Returns the value of a number raised to the power of another number
<u>POWER</u>	Returns the value of a number raised to the power of another number
<u>RADIANS</u>	Converts a degree value into radians
<u>RAND</u>	Returns a random number
<u>ROUND</u>	Rounds a number to a specified number of decimal places
<u>SIGN</u>	Returns the sign of a number
<u>SIN</u>	Returns the sine of a number
<u>SQRT</u>	Returns the square root of a number
<u>SUM</u>	Calculates the sum of a set of values
<u>TAN</u>	Returns the tangent of a number
<u>TRUNCATE</u>	Truncates a number to the specified number of decimal places

SELECT abs(-12)

SELECT acos(0.12)

SELECT atan(0.12);

SELECT asin(0.12);

SELECT atan2(0.12);

SELECT avg(roll) from students

SELECT ceil(12.34)

SELECT ceiling(12.34);

SELECT cos(1.4);

SELECT cot(1.4);

SELECT count(roll) from students WHERE city = 'Rajkot';

SELECT degrees(1.56);

SELECT 100 div 3

SELECT 100 mod 3;

SELECT exp(1)

SELECT exp(2);

SELECT floor(12.34)

SELECT GREATEST(3, 12, 34, 8, 25);

SELECT LEAST(3, 12, 34, 8, 25);

SELECT ln(0.15)

SELECT log(1.15);

SELECT log2(1.15);

SELECT log10(1.15);

SELECT max(roll) FROM students

SELECT min(roll) FROM students;

SELECT pi()

SELECT pow(2, 10)

SELECT power(2, 10);

SELECT radians(180);

SELECT rand()

SELECT round(12.34)

SELECT round(12.54);

SELECT sign(10)

SELECT sign(-10);

SELECT sign(0);

SELECT sin(1.20);

SELECT sqrt(144);

SELECT sum(roll) from students;

```
SELECT tan(1.28)
```

```
SELECT truncate(12.34, 0)
```

```
SELECT truncate(12.34, 1);
```

```
SELECT truncate(12.59, 1);
```

MySQL Date Functions

Function	Description
<u>ADDDATE</u>	Adds a time/date interval to a date and then returns the date
<u>ADDTIME</u>	Adds a time interval to a time/datetime and then returns the time/datetime
<u>CURDATE</u>	Returns the current date
<u>CURRENT_DATE</u>	Returns the current date
<u>CURRENT_TIME</u>	Returns the current time
<u>CURRENT_TIMESTAMP</u>	Returns the current date and time
<u>CURTIME</u>	Returns the current time
<u>DATE</u>	Extracts the date part from a datetime expression
<u>DATEDIFF</u>	Returns the number of days between two date values
<u>DATE_ADD</u>	Adds a time/date interval to a date and then returns the date
<u>DATE_FORMAT</u>	Formats a date
<u>DATE_SUB</u>	Subtracts a time/date interval from a date and then returns the date
<u>DAY</u>	Returns the day of the month for a given date
<u>DAYNAME</u>	Returns the weekday name for a given date
<u>DAYOFMONTH</u>	Returns the day of the month for a given date
<u>DAYOFWEEK</u>	Returns the weekday index for a given date
<u>DAYOFYEAR</u>	Returns the day of the year for a given date
<u>EXTRACT</u>	Extracts a part from a given date
<u>FROM_DAYS</u>	Returns a date from a numeric datevalue

<u>HOUR</u>	Returns the hour part for a given date
<u>LAST_DAY</u>	Extracts the last day of the month for a given date
<u>LOCALTIME</u>	Returns the current date and time
<u>LOCALTIMESTAMP</u>	Returns the current date and time
<u>MAKEDATE</u>	Creates and returns a date based on a year and a number of days value
<u>MAKETIME</u>	Creates and returns a time based on an hour, minute, and second value
<u>MICROSECOND</u>	Returns the microsecond part of a time/datetime
<u>MINUTE</u>	Returns the minute part of a time/datetime
<u>MONTH</u>	Returns the month part for a given date
<u>MONTHNAME</u>	Returns the name of the month for a given date
<u>NOW</u>	Returns the current date and time
<u>PERIOD_ADD</u>	Adds a specified number of months to a period
<u>PERIOD_DIFF</u>	Returns the difference between two periods
<u>QUARTER</u>	Returns the quarter of the year for a given date value
<u>SECOND</u>	Returns the seconds part of a time/datetime
<u>SEC_TO_TIME</u>	Returns a time value based on the specified seconds
<u>STR_TO_DATE</u>	Returns a date based on a string and a format
<u>SUBDATE</u>	Subtracts a time/date interval from a date and then returns the date
<u>SUBTIME</u>	Subtracts a time interval from a datetime and then returns the time/datetime
<u>SYSDATE</u>	Returns the current date and time
<u>TIME</u>	Extracts the time part from a given time/datetime
<u>TIME_FORMAT</u>	Formats a time by a specified format
<u>TIME_TO_SEC</u>	Converts a time value into seconds
<u>TIMEDIFF</u>	Returns the difference between two time/datetime expressions
<u>TIMESTAMP</u>	Returns a datetime value based on a date or datetime value
<u>TO_DAYS</u>	Returns the number of days between a date and date "0000-00-00"

<u>WEEK</u>	Returns the week number for a given date
<u>WEEKDAY</u>	Returns the weekday number for a given date
<u>WEEKOFYEAR</u>	Returns the week number for a given date
<u>YEAR</u>	Returns the year part for a given date
<u>YEARWEEK</u>	Returns the year and week number for a given date

```
SELECT adddate(CURRENT_DATE, INTERVAL 100 day)
```

```
SELECT adddate(CURRENT_DATE, INTERVAL 100 week);
```

```
SELECT addtime(CURRENT_TIMESTAMP, "100");
```

```
SELECT CURRENT_DATE
```

```
SELECT CURDATE();
```

```
SELECT CURRENT_TIME
```

```
SELECT CURRENT_TIMESTAMP
```

```
SELECT curtime()
```

```
SELECT DATE(CURRENT_TIMESTAMP)
```

```
SELECT time(CURRENT_TIMESTAMP);
```

```
SELECT year(CURRENT_TIMESTAMP);
```

```
SELECT month(CURRENT_TIMESTAMP);
```

```
SELECT day(CURRENT_TIMESTAMP);
```

```
SELECT hour(CURRENT_TIMESTAMP);
```

```
SELECT minute(CURRENT_TIMESTAMP);
```

```
SELECT second(CURRENT_TIMESTAMP);
```

```
SELECT datediff(CURRENT_DATE, "2025-01-14")
```

```
SELECT date_add(CURRENT_DATE, INTERVAL 100 day)
```

```
SELECT date_format(CURRENT_TIMESTAMP, "%D %m %Y %H %i" );
```

```
SELECT date_format(CURRENT_TIMESTAMP, "%d %M %Y %H %i" );
```

```
SELECT date_sub(CURRENT_DATE, INTERVAL 100 day)

SELECT dayname(CURRENT_TIMESTAMP)

SELECT dayofmonth(CURRENT_TIMESTAMP);

SELECT dayofweek(CURRENT_TIMESTAMP);

SELECT dayofyear(CURRENT_TIMESTAMP);

SELECT extract(month from CURRENT_TIMESTAMP)

SELECT from_days(1008000); // from 0/0/0000 to days

SELECT last_day(CURRENT_TIMESTAMP)

SELECT LOCALTIME

SELECT LOCALTIMESTAMP;

SELECT MAKEDATE(2027, 3);

SELECT MAKETIME(21, 35, 4);

SELECT MICROSECOND("2027-07-27 07:27:00.0002678");

SELECT monthname(CURRENT_DATE)

SELECT now()

SELECT PERIOD_ADD(202401, 15);

SELECT PERIOD_DIFF(202401, 202908);

SELECT QUARTER(CURRENT_DATE);

SELECT SEC_TO_TIME(10);

SELECT STR_TO_DATE("August 10 2017", "%M %d %Y");

SELECT STR_TO_DATE("August 10 2027", "%M %d %Y");

SELECT subdate(CURRENT_DATE, INTERVAL 100 month);

SELECT SUBTIME("2017-06-15 10:24:21.000004", "5.000001");

SELECT sysdate()

SELECT TIME_FORMAT("19:30:10", "%H %i %s %p");
```

```

SELECT TIME_TO_SEC("19:30:10");

SELECT TIMEDIFF("19:10:11", "17:10:10");

SELECT TIMESTAMP("2017-07-23", "13:10:11");

SELECT TIMESTAMP(CURRENT_DATE, CURRENT_TIME);

SELECT TO_DAYS(CURRENT_DATE);

SELECT WEEK(CURRENT_DATE);

SELECT WEEKOFYEAR(CURRENT_DATE);

SELECT YEARWEEK(CURRENT_DATE);

```

MySQL Advanced Functions

Function	Description
<u>BIN</u>	Returns a binary representation of a number
<u>BINARY</u>	Converts a value to a binary string
<u>CASE</u>	Goes through conditions and return a value when the first condition is met
<u>CAST</u>	Converts a value (of any type) into a specified datatype
<u>COALESCE</u>	Returns the first non-null value in a list
<u>CONNECTION_ID</u>	Returns the unique connection ID for the current connection
<u>CONV</u>	Converts a number from one numeric base system to another
<u>CONVERT</u>	Converts a value into the specified datatype or character set
<u>CURRENT_USER</u>	Returns the user name and host name for the MySQL account that the server used to authenticate the current client
<u>DATABASE</u>	Returns the name of the current database
<u>IF</u>	Returns a value if a condition is TRUE, or another value if a condition is FALSE
<u>IFNULL</u>	Return a specified value if the expression is NULL, otherwise return the expression
<u>ISNULL</u>	Returns 1 or 0 depending on whether an expression is NULL

<u>LAST_INSERT_ID</u>	Returns the AUTO_INCREMENT id of the last row that has been inserted or updated in a table
<u>NULLIF</u>	Compares two expressions and returns NULL if they are equal. Otherwise, the first expression is returned
<u>SESSION_USER</u>	Returns the current MySQL user name and host name
<u>SYSTEM_USER</u>	Returns the current MySQL user name and host name
<u>USER</u>	Returns the current MySQL user name and host name
<u>VERSION</u>	Returns the current version of the MySQL database

SELECT BIN(15);

SELECT BINARY "W3Schools.com";

SELECT CAST("2017-08-29" AS DATE);

SELECT COALESCE(NULL, NULL, NULL, 'W3Schools.com', NULL, 'Example.com');

SELECT CONNECTION_ID()

SELECT CONV(15, 10, 2);

SELECT CONVERT("2017-08-29", DATE);

SELECT CURRENT_USER;

SELECT DATABASE()

SELECT IF(500<1000, "True", "False");

SELECT NULLIF(25, 25);

SELECT NULLIF(25, 125);

SELECT SESSION_USER()

SELECT SYSTEM_USER();

SELECT USER();

SELECT version();

What is Normalization in SQL?

As an SQL Developer, you often work with enormous amounts of data stored in different tables that are present inside multiple [databases](#). It often becomes strenuous to extract the data if it is not organized correctly. Using Normalization, you can solve the problem of data redundancy and organize the data using different forms. This tutorial will help you get to know the concept of Normalization in SQL.

In this tutorial, you will learn the following topics:

- What is Normalization in SQL?
- First Normal Form (1st NF)
- Second Normal Form (2nd NF)
- Third Normal Form (3rd NF)
- Boyce Codd Normal Form

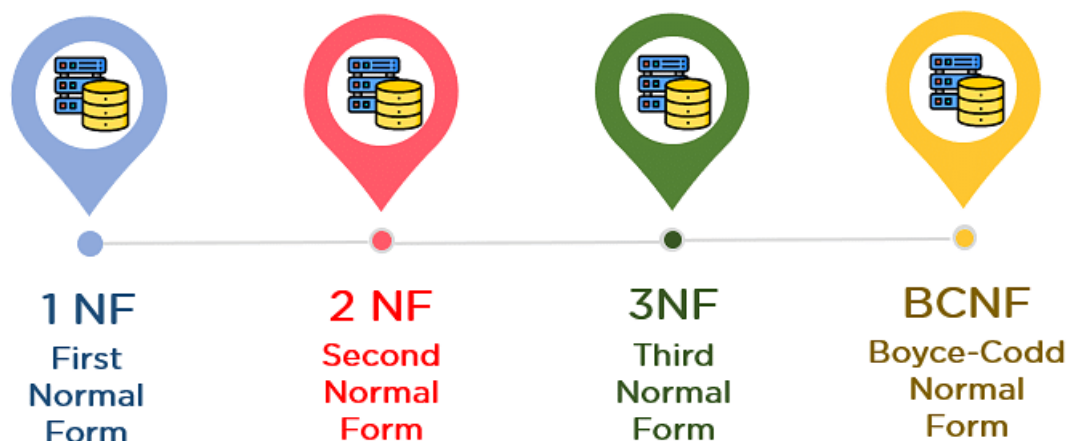
What Is Normalization in SQL?

Normalization is the process to eliminate data redundancy and enhance data integrity in the table. Normalization also helps to organize the data in the database. It is a multi-step process that sets the data into tabular form and removes the duplicated data from the relational tables.

Normalization organizes the columns and tables of a database to ensure that database integrity constraints properly execute their dependencies. It is a systematic technique of decomposing tables to eliminate data redundancy (repetition) and undesirable characteristics like Insertion, Update, and Deletion anomalies.

In 1970 Edgar F. Codd defined the First Normal Form.

Now let's understand the types of Normal forms with the help of examples.



1st Normal Form (1NF)

- A table is referred to as being in its First Normal Form if atomicity of the table is 1.
- Here, atomicity states that a single cell cannot hold multiple values. It must hold only a single-valued attribute.
- The First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

Now you will understand the First Normal Form with the help of an example.

Below is a students' record table that has information about student roll number, student name, student course, and age of the student.

	rollno	name	course	age
▶	1	Rahul	c/c++	22
	2	Harsh	java	18
	3	Sahil	c/c++	23
	4	Adam	c/c++	22
	5	Lisa	java	24
	6	James	c/c++	19
✱	NULL	NULL	NULL	NULL

In the studentsrecord table, you can see that the course column has two values. Thus it does not follow the First Normal Form. Now, if you use the First Normal Form to the above table, you get the below table as a result.

	rollno	name	course	age
▶	1	Rahul	c	22
	1	Rahul	c++	22
	2	Harsh	java	18
	3	Sahil	c	23
	3	Sahil	c++	23
	4	Adam	c	22
	4	Adam	c++	22
	5	Lisa	java	24
	6	James	c	19
	6	James	c++	19

By applying the First Normal Form, you achieve atomicity, and also every column has unique values.

Before proceeding with the Second Normal Form, get familiar with **Candidate Key** and **Super Key**.

Candidate Key

A candidate key is a set of one or more columns that can identify a record uniquely in a table, and YOU can use each candidate key as a **primary key**.

Now, let's use an example to understand this better.

Candidate Key

	id	roll_no	stu_name	city	dept_id
▶	1	22	Arush	Delhi	11
	2	23	Akash	Pune	11
	3	24	Ajay	Kolkata	22
	4	25	Naman	Delhi	22
	5	26	Nishtha	UP	33

Super Key

Super key is a set of over one key that can identify a record uniquely in a table, and the Primary Key is a subset of Super Key.

Let's understand this with the help of an example.

	id	roll_no	stu_name	enroll_no	city	dept_id
▶	1	22	Arush	223	Delhi	11
	2	23	Akash	224	Pune	11
	3	24	Ajay	225	Kolkata	22
	4	25	Naman	226	Delhi	22
	5	26	Nishtha	227	UP	33
	6	27	Lamba	228	Haryana	44

Super Key

Second Normal Form (2NF)

The first condition for the table to be in Second Normal Form is that the table has to be in **First Normal Form**. The table should not possess partial dependency. The partial dependency here means the proper subset of the candidate key should give a non-prime attribute.

Now understand the Second Normal Form with the help of an example.

Consider the table Location:

	cust_id	storeid	store_location
▶	1	D1	Toronto
	2	D3	Miami
	3	T1	California
	4	F2	Florida
	5	H3	Texas

The Location table possesses a composite primary key **cust_id, storeid**. The non-key attribute is store_location. In this case, **store_location** only depends on **storeid**, which is a part of the primary key. Hence, this table does not fulfill the second normal form.

To bring the table to Second Normal Form, you need to split the table into two parts. This will give you the below tables:

	cust_id	storeid
▶	1	D1
	2	D3
	3	T1
	4	F2
	5	H3

	storeid	store_location
▶	D1	Toronto
	D3	Miami
	T1	California
	F2	Florida
	H3	Texas

As you have removed the partial functional dependency from the location table, the column store_location entirely depends on the primary key of that table, storeid.

Now that you understood the 1st and 2nd Normal forms, you will look at the next part of this Normalization in SQL tutorial.

Third Normal Form (3NF)

The first condition for the table to be in Third Normal Form is that the table should be in the **Second Normal Form**.

The second condition is that there should be no transitive dependency for non-prime attributes, which indicates that non-prime attributes (which are not a part of the candidate key) should not depend on other non-prime attributes in a table. Therefore, a transitive dependency is a functional dependency in which $A \rightarrow C$ (A determines C) indirectly, because of $A \rightarrow B$ and $B \rightarrow C$ (where it is not

the case that $B \rightarrow A$). The third Normal Form ensures the reduction of data duplication. It is also used to achieve data integrity.

Below is a student table that has student id, student name, subject id, subject name, and address of the student as its columns.

	stu_id	name	subid	sub	address
▶	1	Arun	11	SQL	Delhi
	2	Varun	12	Java	Bangalore
	3	Harsh	13	C++	Delhi
	4	Keshav	12	Java	Kochi

In the above student table, stu_id determines subid, and subid determines sub. Therefore, stu_id determines sub via subid. This implies that the table possesses a **transitive functional dependency**, and it does not fulfill the third normal form criteria.

Now to change the table to the third normal form, you need to divide the table as shown below:

	stu_id	name	subid	address
▶	1	Arun	11	Delhi
	2	Varun	12	Bangalore
	3	Harsh	13	Delhi
	4	Keshav	12	Kochi

	subid	subject
▶	11	SQL
	12	java
	13	C++
	12	Java

As you can see in both the tables, all the non-key attributes are now fully functional, dependent only on the primary key. In the first table, columns name, subid, and addresses only depend on stu_id. In the second table, the sub only depends on subid.

Boyce Codd Normal Form (BCNF)

Boyce Codd Normal Form is also known as 3.5 NF. It is the superior version of 3NF and was developed by **Raymond F. Boyce and Edgar F. Codd** to tackle certain types of anomalies which were not resolved with 3NF.

The first condition for the table to be in Boyce Codd Normal Form is that the table should be in the **third normal form**. Secondly, every Right-Hand Side (RHS) attribute of the functional dependencies should depend on the super key of that particular table.

For example :

You have a functional dependency $X \rightarrow Y$. In the particular functional dependency, X has to be the part of the super key of the provided table.

Consider the below **subject** table:

	stuid	subject	professor
▶	1	SQL	Prof. Mishra
	2	Java	Prof. Anand
	2	C++	Prof. Kanth
	3	Java	Prof. James
	4	DBMS	Prof. Lokesh

The subject table follows these conditions:

- Each student can enroll in multiple subjects.
- Multiple professors can teach a particular subject.
- For each subject, it assigns a professor to the student.

In the above table, student_id and subject together form the primary key because using student_id and subject; you can determine all the table columns.

Another important point to be noted here is that one professor teaches only one subject, but one subject may have two professors.

Which exhibit there is a dependency between subject and professor, i.e. subject depends on the professor's name.

to

The table is in 1st Normal form as all the column names are unique, all values are atomic, and all the values stored in a particular column are of the same domain.

The table also satisfies the 2nd Normal Form, as there is no Partial Dependency.

And, there is no Transitive Dependency; hence, the table also satisfies the 3rd Normal Form.

This table follows all the Normal forms except the Boyce Codd Normal Form.

As you can see stuid, and subject forms the primary key, which means the subject attribute is a prime attribute.

However, there exists yet another dependency - professor \rightarrow subject.

BCNF does not follow in the table as a subject is a prime attribute, the professor is a non-prime attribute.

To transform the table into the BCNF, you will divide the table into two parts. One table will hold stuid which already exists and the second table will hold a newly created column profid.

	stuid	profid
►	1	101
	2	102
	2	103
	3	102
	4	104

	profid	subject	professor
►	1	SQL	Prof. Mishra
	2	Java	Prof. Anand
	2	C++	Prof. Kanth
	3	Java	Prof. James
	4	DBMS	Prof. Lokesh

And in the second table will have the columns profid, subject, and professor, which satisfies the BCNF.

With this, you have reached the conclusion of the 'Normalization in SQL' tutorial.

Advantages of Normalization

- Normalization helps to minimize data redundancy.
- Greater overall database organization.
- Data consistency within the database.
- Much more flexible database design.
- Enforces the concept of relational integrity.

Disadvantages of Normalization

- You cannot start building the database before knowing what the user needs.
- The performance degrades when normalizing the relations to higher normal forms, i.e., 4NF, 5NF.
- It is very time-consuming and difficult to normalize relations of a higher degree.
- Careless decomposition may lead to a bad database design, leading to serious problems.