

## How to Start MySQL Server

Download latest version of XAMPP From <https://www.apachefriends.org/download.html>

Install XAMPP and open XAMPP Control Panel

From Control panel you need to start two services apache, mysql

Now open browser and enter URL <http://localhost/phpmyadmin/> or press admin button from XAMPP control panel.

Now you are connected with MySQL server home page

# 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

# 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:

```
SELECT * FROM Customers;
```

## 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.

## 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

Before continue with MySQL you need to create sample database and add some data in it.

Server: 127.0.0.1 » Database: 1121\_2324 » Table: students

Browser Structure SQL Search Insert Export Import Privileges Operations Tracking Triggers

Table structure Relation view

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
<input type="checkbox"/>	1	roll			No	None		AUTO_INCREMENT	Change  Drop  More
<input type="checkbox"/>	2	fname	utf8mb4_general_ci		No	None			Change  Drop  More
<input type="checkbox"/>	3	lname	utf8mb4_general_ci		No	None			Change  Drop  More
<input type="checkbox"/>	4	city	utf8mb4_general_ci		No	None			Change  Drop  More
<input type="checkbox"/>	5	phone	utf8mb4_general_ci		No	None			Change  Drop  More
<input type="checkbox"/>	6	email	utf8mb4_general_ci		No	None			Change  Drop  More
<input type="checkbox"/>	7	gender	utf8mb4_general_ci		No	None			Change  Drop  More
<input type="checkbox"/>	8	dateofbirth			No	None			Change  Drop  More
<input type="checkbox"/>	9	admissiondata			No	current_timestamp()			Change  Drop  More

Console

Server: 127.0.0.1 » Database: 1121\_2324 » Table: students

Browser Structure SQL Search Insert Export Import Privileges Operations Tracking Triggers

roll fname lname city phone email gender dateofbirth admissiondata

<input type="checkbox"/>	Edit  Copy  Delete	1	Meet	sinojiya	Morbi	998899889900	meet@gmail.com	male	2004-02-12	2024-02-14 12:04:08
<input type="checkbox"/>	Edit  Copy  Delete	2	Brijesh	Miatra	Ahamdabad	889883455	brijesh@gmail.com	male	2007-02-13	2024-02-14 12:04:08
<input type="checkbox"/>	Edit  Copy  Delete	3	Ansh	Amrutiya	Rajkot	998899889900	Ansh@gmail.com	male	2004-02-12	2024-02-14 12:04:48
<input type="checkbox"/>	Edit  Copy  Delete	4	Alan	Thomas	Surat	889883455	alan@gmail.com	male	2007-02-13	2024-02-14 12:04:48
<input type="checkbox"/>	Edit  Copy  Delete	5	krishil	trivedi	Rajkot	998899889900	krishil@gmail.com	male	2004-02-12	2024-02-14 12:05:28
<input type="checkbox"/>	Edit  Copy  Delete	6	sunny	sata	Surat	889883455	sunny@gmail.com	male	2007-02-13	2024-02-14 12:05:28
<input type="checkbox"/>	Edit  Copy  Delete	7	pooja	mori	Rajkot	998899889900	pooja@gmail.com	female	2004-02-12	2024-02-14 12:06:09
<input type="checkbox"/>	Edit  Copy  Delete	8	sanjana	sonagra	Surendranagar	889883455	sanajan@gmail.com	female	2007-02-13	2024-02-14 12:06:09



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

```
SELECT * FROM table_name;
```

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

```
SELECT * from students;
```

## 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 city from students;`

`SELECT DISTINCT city from students;`

`SELECT count(DISTINCT city) from students;`

`SELECT roll, fname, lname, dateofbirth 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 roll, fname, lname, dateofbirth from students WHERE roll = 1;`

`SELECT roll, fname, lname, dateofbirth from students WHERE roll > 5;`

`SELECT roll, fname, lname, dateofbirth 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.!

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

`SELECT * from students WHERE not city = 'rajkot';`

`SELECT * from students WHERE city <> "rajkot";`

### 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 not city = rajkot;`

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

# 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 ...;
```

```
SELECT * from students WHERE roll = 1
```

```
SELECT * from students WHERE roll = 1 and city = 'Rajkot';
```

### OR Syntax

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

```
SELECT * from students WHERE city = 'surat' or city = 'Rajkot';
```

### NOT Syntax

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

```
SELECT * from students WHERE not (city = 'surat' or 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 = 'surat' or city = 'Rajkot' or city = 'morbi');
```

```
SELECT * from students WHERE roll = 1 or roll = 5 and city = 'Rajkot';
```

# 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
```

### ORDER BY DESC Example

```
SELECT * FROM students ORDER by fname desc
```

### ORDER BY Several Columns Example

```
SELECT * FROM students ORDER by fname, city
```

```
SELECT * FROM students ORDER by fname, city desc
```

```
SELECT * FROM students ORDER by fname asc, city 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, ...);
```

```
INSERT into students (fname, lname, city, phone, email, gender, dateofbirth) values ('demo', 'text',
'example', '9900009900', 'demo@example.com', 'male', '2001-02-01')
```

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 students values ('demo', 'text', 'example', '9900009900', 'demo@example.com',
'male', '2001-02-01')
```

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

```
INSERT into students values (null,'demo', 'text', 'example', '9900009900',
'demo@example.com', 'male', '2001-02-01', null)
```

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

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

### Insert Data Only in Specified Columns

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

```
INSERT into students (fname, lname, city) VALUES ('another', 'example', 'of insert')
```

# 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!

If you allow to NULL values in your table first you need to modify column to accept NULL values.

Go to structure section of our table then select specific column to allow null values click on change and select null check box and click save.

```
INSERT into students (fname, lname, city) VALUES ('another', 'example', 'of insert')
```

## 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;
```

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

```
SELECT * from students WHERE gender = NULL;
```

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

```
SELECT * from students WHERE gender is null;
```

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

# 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 = 'bhuj' WHERE roll = 1
```

```
UPDATE students set phone = '9090908080' WHERE roll = 1
```

```
UPDATE students set city = 'Gandhinagar'
```

### UPDATE Multiple Records

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

```
UPDATE students set city = 'rajkot' WHERE roll >= 1 and roll <= 5
```

```
UPDATE students set city = 'ahamdabad' WHERE roll >= 6 and roll <= 10;
```

### Update Warning!

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

```
UPDATE students set fname = 'KRISHIL', city = 'baroda', phone = '9998889990' WHERE roll = 5
```



# 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
```

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

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

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

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

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 Customers LIMIT 3 OFFSET 3;
```

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

```
SELECT * FROM students LIMIT 10, 5; --10 offset 5 limit
```

# 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(dateofbirth) FROM students
```

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

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

```
SELECT min(roll) 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
```

```
SELECT COUNT(roll) FROM students WHERE city = 'rajkot';
```

```
SELECT COUNT(roll) FROM students WHERE not city = 'rajkot';
```

```
SELECT sum(roll) FROM students
```

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

```
SELECT sum(roll), avg(roll) FROM students;
```

# 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

The percent sign and the underscore can also be used in combinations!

```
SELECT * from students
```

```
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 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.

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

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

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"

# 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	bl% finds bl, black, blue, and blob
_	Represents a single character	h_t finds hot, hat, and hit

The wildcards can also be used in combinations!

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

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that starts with "a"
WHERE CustomerName LIKE '%a'	Finds any values that ends 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 starts with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that starts with "a" and ends with "o"

# 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 = 'ahamdabad' or city = 'bhuj'
```

```
SELECT * from students WHERE city in ('rajkot', 'ahamdabad', 'baroda')
```

## IN Operator Examples

The following SQL statement selects all customers that are located in "Germany", "France" or "UK":

```
SELECT * FROM Customers WHERE Country IN ('Germany', 'France', 'UK');
```

```
SELECT * from students WHERE city in ('rajkot', 'ahamdabad', 'baroda')
```

```
SELECT * from students WHERE city not in('rajkot', 'ahamdabad', 'baroda');
```

# 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 BETWEEN 1 and 5`

`SELECT * FROM students WHERE roll not BETWEEN 1 and 5;`

### BETWEEN Text Values Example

`SELECT * FROM students WHERE fname BETWEEN 'ansh' and 'krishil'`

`SELECT * FROM students WHERE dateofbirth BETWEEN '2000-01-01' and '2005-12-31'`

`SELECT * FROM students WHERE dateofbirth not BETWEEN '2000-01-01' and '2005-12-31';`

# 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!

If you try delete query on your table and you need to data back must create shadow copy of your current table to prevent any data loss

`CREATE TABLE studentsBackup as SELECT * FROM students`

### SQL DELETE Example

`DELETE from students WHERE Roll = 1`

`DELETE from students WHERE city = 'gandhinagar'`

`DELETE from students WHERE roll > 5`

`DELETE from students`

Restore data from backup table

`INSERT into students SELECT * from studentsbackup`

Delete from students (empty students table)

`INSERT into students (fname, lname, city, phone, email, gender, dateofbirth) values ('demo', 'text', 'example', '9900009900', 'demo@example.com', 'male', '2001-02-01')`

Notice the roll number started from where we leave before delete

`Select * from students`

If you need to reset whole table with data and data structure then you need to run truncate table

`TRUNCATE TABLE students`

`INSERT into students (fname, lname, city, phone, email, gender, dateofbirth) values ('demo', 'text', 'example', '9900009900', 'demo@example.com', 'male', '2001-02-01')`

`Select * from 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;
```

### Alias Table Syntax

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

---

```
SELECT * from students
```

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

```
SELECT roll as "Roll Number", fname as "First Name", lname as "Last Name", city as "Home Town",  
phone as "Phone Number", email as "Email Address", gender as "Student Gender", dateofbirth as  
"Date of Birth" from students;
```

```
SELECT roll "Roll Number", fname "First Name", lname "Last Name", city "Home Town", phone  
"Phone Number", email "Email Address", gender "Student Gender", dateofbirth "Date of Birth" from  
students;
```

```
SELECT roll RollNumber, fname FirstName, lname LastName, city HomeTown, phone PhoneNumber,  
email EmailAddress, gender StudentGender, dateofbirth DateofBirth from students;
```

The following SQL statement creates two aliases, one for the CustomerName column and one for the ContactName column. **Note:** Single or double quotation marks are required if the alias name contains spaces:

```
SELECT concat_ws("_", roll, fname, lname, city, email, phone, gender, dateofbirth, admissiondata)  
FROM students
```

```
SELECT concat_ws("_", roll, fname, lname, city, email, phone, gender, dateofbirth, admissiondata) as  
"Student Information" FROM students;
```

```
SELECT concat_ws(" * ", roll, fname, lname, city, email, phone, gender, dateofbirth, admissiondata)  
as "Student Information" FROM students;
```

## Alias for Tables Example

### Without alias

```
SELECT students.roll, students.fname, students.lname, students.city, students.phone, students.email,
students.gender, students.dateofbirth, students.admissiondata, attendance.absents,
attendance.presents from students, attendance WHERE students.roll = 1 and students.roll =
attendance.roll
```

### With alias

```
SELECT s.roll, s.fname, s.lname, s.city, s.phone, s.email, s.gender, s.dateofbirth, s.admissiondata,
a.absents, a.presents from students as s, attendance as a WHERE s.roll = 1 and s.roll = a.roll;
```

```
SELECT s.roll, s.fname, s.lname, s.city, s.phone, s.email, s.gender, s.dateofbirth, s.admissiondata,
a.absents, a.presents from students s, attendance a WHERE s.roll = 1 and s.roll = a.roll;
```

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

```
SELECT s.roll, s.fname, s.lname, s.city, s.phone, s.email, s.gender, s.dateofbirth, s.admissiondata,
a.absents, a.presents, sum(a.absents+a.presents) from students s, attendance a WHERE s.roll = 1 and
s.roll = a.roll;
```

```
SELECT s.roll, s.fname, s.lname, s.city, s.phone, s.email, s.gender, s.dateofbirth, s.admissiondata,
a.absents, a.presents, sum(a.absents+a.presents) "Total Days" from students s, attendance a WHERE
s.roll = 1 and s.roll = a.roll;
```

# MySQL Joins

## MySQL Joining Tables

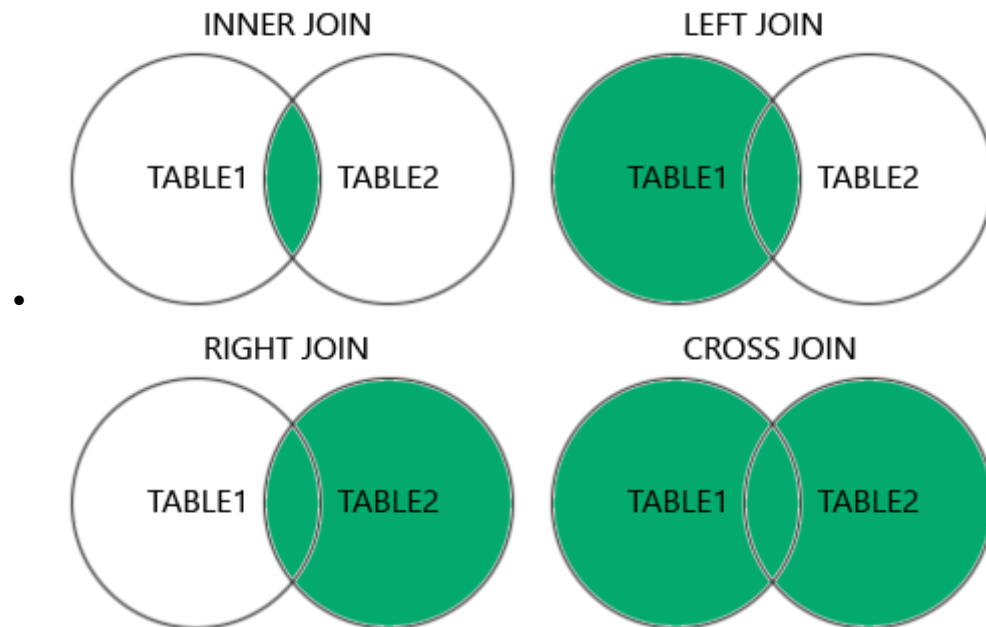
A `JOIN` clause is used to combine rows from two or more tables, based on a related column between them.

```
SELECT students.roll, students.fname, students.lname, students.city, students.phone, students.email,
students.gender, students.dateofbirth, students.admissiondata, marks.total, marks.result from
students inner JOIN marks on students.roll = marks.roll
```

```
SELECT s.roll, s.fname, s.lname, s.city, s.phone, s.email, s.gender, s.dateofbirth, s.admissiondata,
m.total, m.result from students s inner JOIN marks m on s.roll = m.roll;
```

## Supported Types of Joins in MySQL

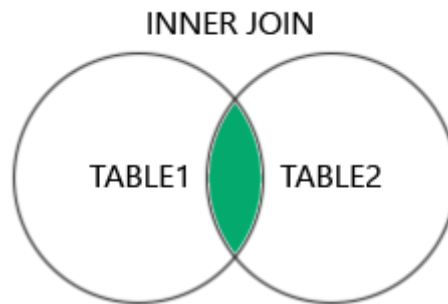
- `INNER JOIN`: Returns records that have matching values in both tables
- `LEFT JOIN`: Returns all records from the left table, and the matched records from the right table
- `RIGHT JOIN`: Returns all records from the right table, and the matched records from the left table
- `CROSS JOIN`: Returns all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



# MySQL INNER JOIN Keyword

## 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 s.roll, s.fname, s.lname, s.city, s.phone, s.email, s.gender, s.dateofbirth, s.admissiondata,  
m.total, m.result from students s inner JOIN marks m on s.roll = m.roll;
```

```
SELECT students.*, attendance.absents, attendance.presents FROM students INNER join attendance  
on students.roll = attendance.roll;
```

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

### JOIN Three Tables

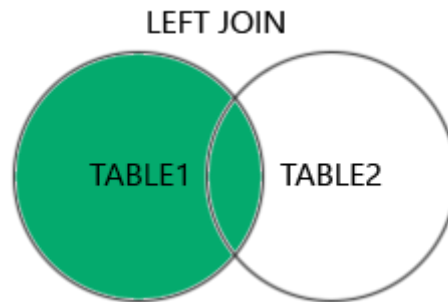
```
SELECT students.roll, students.fname, students.lname, students.city, students.phone, students.email,  
students.gender, students.dateofbirth, students.admissiondata, attendance.absents,  
attendance.presents, marks.total, marks.result from students INNER join attendance on students.roll  
= attendance.roll INNER join marks on students.roll = marks.roll;
```

```
SELECT s.roll, s.fname, s.lname, s.city, s.phone, s.email, s.gender, s.dateofbirth, s.admissiondata,  
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;
```

// inner join

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

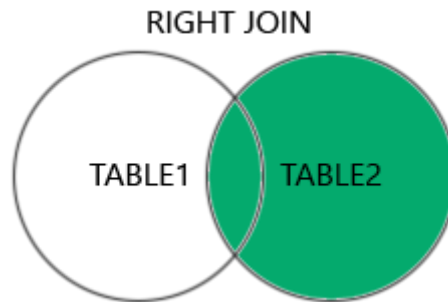
// left join

```
SELECT students.roll, students.fname, students.lname, students.city, students.phone, students.email,  
students.gender, students.dateofbirth, students.admissiondata, 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

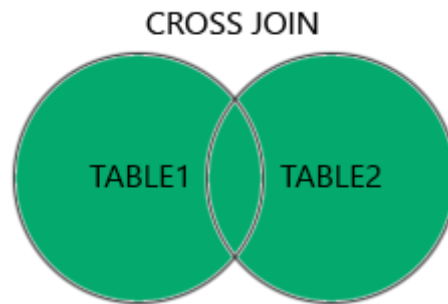
```
SELECT column_name(s) FROM table1 RIGHT JOIN table2 ON table1.column_name =  
table2.column_name;
```

```
SELECT students.roll, students.fname, students.lname, students.city, students.phone, students.email,  
students.gender, students.dateofbirth, students.admissiondata, 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!

## MySQL CROSS JOIN Example

```
SELECT students.* from students CROSS join attendance
```

Return large resultset (total records of students \* total records of attendance)

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

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 students.*, attendance.absents, attendance.presents from students CROSS join attendance  
WHERE students.roll = attendance.roll;
```

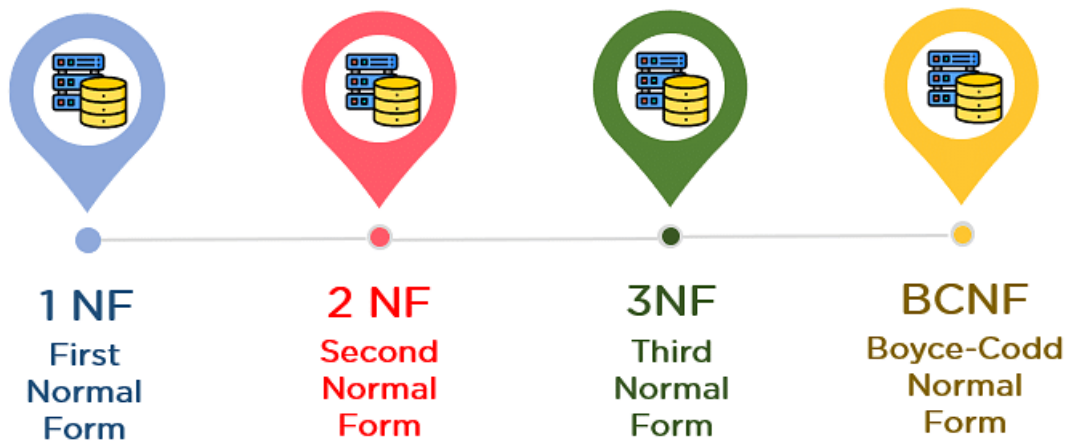
## What Is Normalization in SQL?

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 students record 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

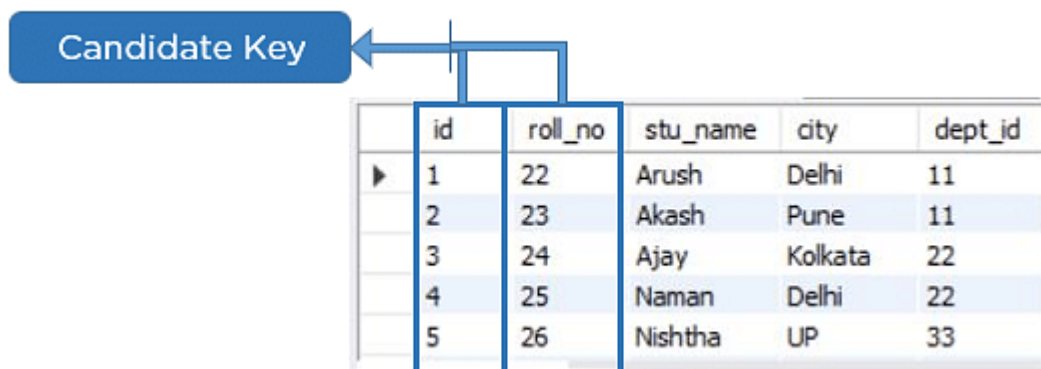
By applying the First Normal Form, you achieve atomicity, and also every column has unique values.

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.



	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

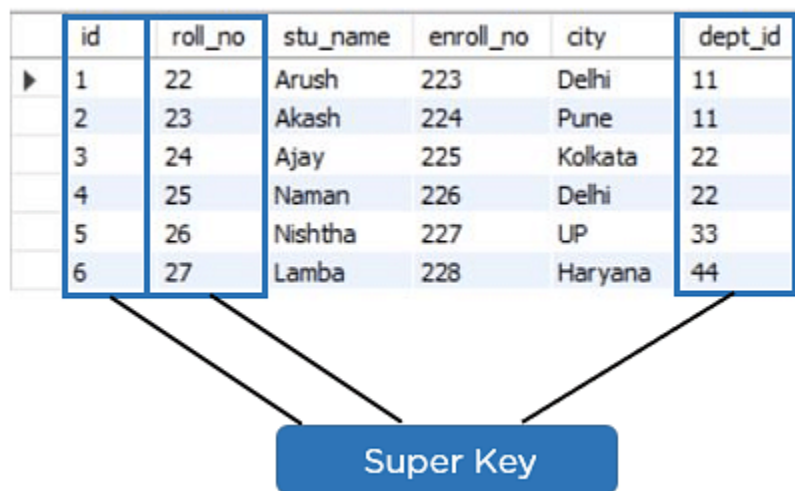
Super key is a set of over one key that can identify a record uniquely in a table, and the Primary Key is a subset of Super Key.

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

## 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.



## 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 → 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

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

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

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

# 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.

MySQL Self Join Example

```
SELECT s1.fname as "Student s1", s2.fname as "Student s2", s1.city as "City s1" from students s1,  
students s2 WHERE s1.roll <> s2.roll and s1.city = s2.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
```

```
SELECT * from students UNION SELECT * from students1
```

### UNION ALL Syntax

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

```
SELECT * from students UNION ALL SELECT * from students1
```

```
SELECT * from students WHERE city = 'Rajkot' UNION ALL SELECT * from students1 WHERE city = 'Rajkot';
```

---

```
SELECT roll, fname, lname, city from students WHERE city = 'Rajkot'
```

```
UNION ALL
```

```
SELECT roll, fname, lname, city 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.

### GROUP BY Syntax

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

Select \* from students

SELECT DISTINCT(city) from students

SELECT city, COUNT(city) from students GROUP by(city)

SELECT city, COUNT(city) from students GROUP by(city) ORDER by COUNT(city);

SELECT city, COUNT(city) from students GROUP by(city) ORDER by COUNT(city) desc;

### GROUP BY With JOIN Example

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

---

```
SELECT students.roll, students.fname, students.lname, students.city, students.gender,
students.phone, students.email, marks.total, marks.result, count(marks.roll) as "Attempts" from
students INNER join marks on students.roll = marks.roll GROUP by (marks.roll);
```



# MySQL HAVING Clause

## The MySQL HAVING Clause

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

### HAVING Syntax

`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, students.city, students.gender,
students.phone, students.email, marks.total, marks.result, count(marks.roll) as "Attempts" from
students INNER join marks on students.roll = marks.roll GROUP by (marks.roll) having
count(marks.roll) > 1;
```

```
SELECT students.roll, students.fname, students.lname, students.city, students.gender,
students.phone, students.email, marks.total, marks.result, count(marks.roll) as "Attempts" from
students INNER join marks on students.roll = marks.roll GROUP by (marks.roll) having
count(marks.roll) >= 3;
```

# MySQL EXISTS Operator

## 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.

```
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 students.roll = marks.roll and  
marks.result = 'pass')
```

```
SELECT * from students WHERE not EXISTS (SELECT roll from marks WHERE students.roll = marks.roll  
and marks.result = 'pass');
```

# 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 WHERE roll = any (SELECT roll FROM marks WHERE result = 'pass');
```

**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;
```

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

#### 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 <=).

```
SELECT * from students WHERE roll = all (SELECT roll from marks WHERE result =  
'pass')
```

# 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 SELECT * from students WHERE city = 'rajkot';\
```

```
INSERT into students1 (fname, lname, city) SELECT fname, lname, city 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`.

### CASE Syntax

#### CASE

```
WHEN condition1 THEN result1
WHEN condition2 THEN result2
WHEN conditionN THEN resultN
ELSE result
```

```
END;
```

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

```
SELECT roll, fname, lname, city, gender, case
```

```
    WHEN gender = 'female' then 'Pink'
```

```
    WHEN gender = 'male' then 'Blue'
```

```
    else 'Black'
```

```
end as "Bag Color", phone, email FROM students;
```

---

```
SELECT roll, fname, lname, city, case
```

```
    WHEN city = 'Rajkot' THEN 'Home Town'
```

```
    WHEN city = 'Ahamdabad' THEN 'Far From Home'
```

```
    WHEN city = 'Gandhinagar' THEN 'Far From Home'
```

```
    WHEN city = 'Surat' THEN 'Too Much Far From Home'
```

```
    ELSE 'Unknown Distance'
```

```
end as "Distance From Home",gender, phone, email FROM students;
```

# MySQL NULL Functions

## MySQL IFNULL() and COALESCE() Functions

```
SELECT roll, absents, presents, (absents + presents) as "Total Days" FROM attendance
```

Add some null values in table and above query

```
SELECT roll, absents, presents, (absents + presents) as "Total Days" FROM attendance
```

## MySQL IFNULL() Function

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

The example below returns 0 if the value is NULL:

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

## MySQL COALESCE() Function

```
SELECT roll, absents, presents, (COALESCE(absents, 0) + COALESCE(presents, 0)) as "Total 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:

```
-- following query replace absent / present to 0 if value of absent / present is null
```

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

---

```
SELECT roll, absents, presents, (COALESCE(absents, 0) + COALESCE(presents, 0)) as "Total Days"  
FROM attendance; -- following query replace absent / present to 0 if value of absent / present is null
```

### 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:

```
/* following query replace absent / present to 0 if value of absent / present is null */
```

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

---

```
SELECT roll, absents, presents, (COALESCE(absents, 0) + COALESCE(presents, 0)) as "Total Days"  
FROM attendance; /* following query replace absent / present to 0 if value of absent / present is null  
*/
```

---

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



# MySQL Operators

## MySQL Arithmetic Operators

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide
%	Module

## MySQL Bitwise Operators

Operator	Description
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR

## MySQL Comparison Operators

Operator	Description	Example
=	Equal to	
>	Greater than	
<	Less than	
>=	Greater than or equal to	
<=	Less than or equal to	
<>	Not equal to	

## MySQL Compound Operators

Operator	Description
+=	Add equals
-=	Subtract equals
*=	Multiply equals

/=	Divide equals
%=	Modulo equals
&=	Bitwise AND equals
^-=	Bitwise exclusive equals
*=	Bitwise OR equals

## MySQL Logical Operators

Operator	Description	Example
ALL	TRUE if all of the subquery values meet the condition	
AND	TRUE if all the conditions separated by AND is TRUE	
ANY	TRUE if any of the subquery values meet the condition	
BETWEEN	TRUE if the operand is within the range of comparisons	
EXISTS	TRUE if the subquery returns one or more records	
IN	TRUE if the operand is equal to one of a list of expressions	
LIKE	TRUE if the operand matches a pattern	
NOT	Displays a record if the condition(s) is NOT TRUE	
OR	TRUE if any of the conditions separated by OR is TRUE	
SOME	TRUE if any of the subquery values meet the condition	

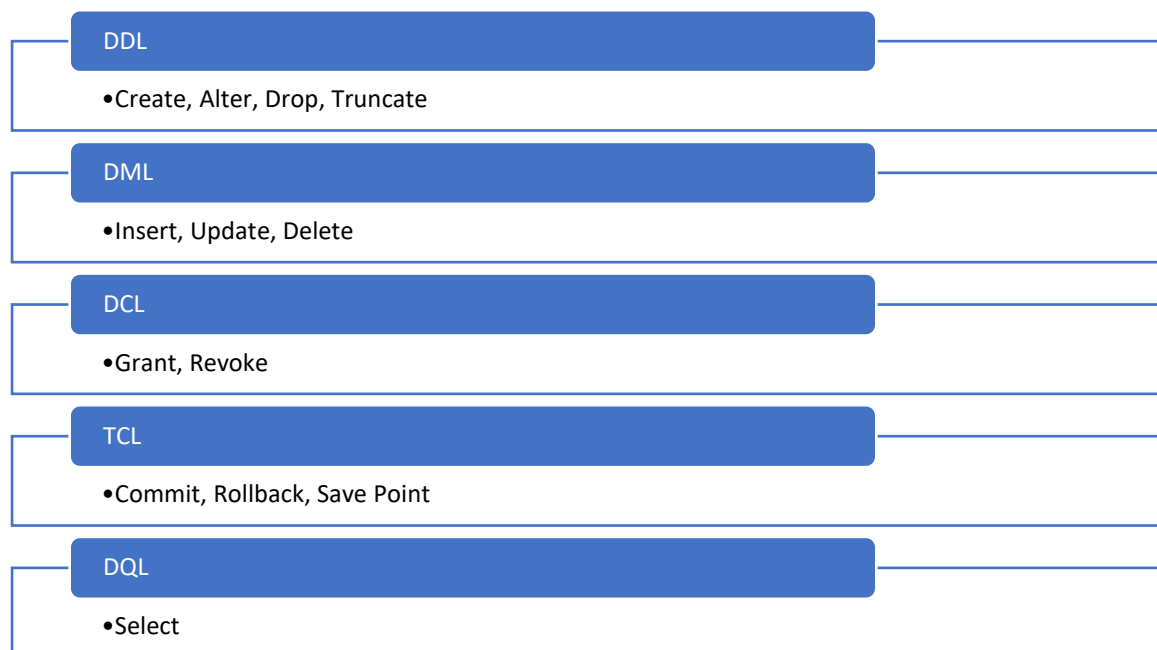
# Type of SQL Commands

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



### 1. 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

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

**Syntax:**

```
CREATE TABLE TABLE_NAME (COLUMN_NAME DATATYPES[,....]);
```

## 2. 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

## 3. 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

## 4. 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

## 5. Data Query Language

DQL is used to fetch the data from the database.

It uses only one command:

- SELECT

# 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":

**ip:** 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
```

```
create DATABASE example1121
```

```
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":

**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;`

```
show DATABASES;
```

```
drop DATABASE example1121
```

```
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.).

## MySQL CREATE TABLE Example

The following example creates a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City:

```
create TABLE persons(personid int AUTO_INCREMENT PRIMARY key, fname varchar(20), lname  
varchar(20), email varchar(128), city varchar(20), dateofbirth date, createdat timestamp DEFAULT  
CURRENT_TIMESTAMP)
```

---

## 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 personsBackup as SELECT * from persons
```

```
create TABLE personsBackup1 as SELECT personid, fname, lname, city, email 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 gender varchar(10)
```

### 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 gender
```

---

```
ALTER TABLE persons add COLUMN gender varchar(10) AFTER lname
```

### 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 gender varchar(6)
```

```
ALTER TABLE persons MODIFY COLUMN gender tinyint
```

# 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!

```
drop table personsbackup
```

## MySQL TRUNCATE TABLE

The `TRUNCATE TABLE` statement is used to delete the data inside a table, but not the table itself.

Syntax

**TRUNCATE TABLE** *table\_name*;

```
TRUNCATE TABLE students
```

When you truncate any table all the data from deleted and all the setting reset for table.

# 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.

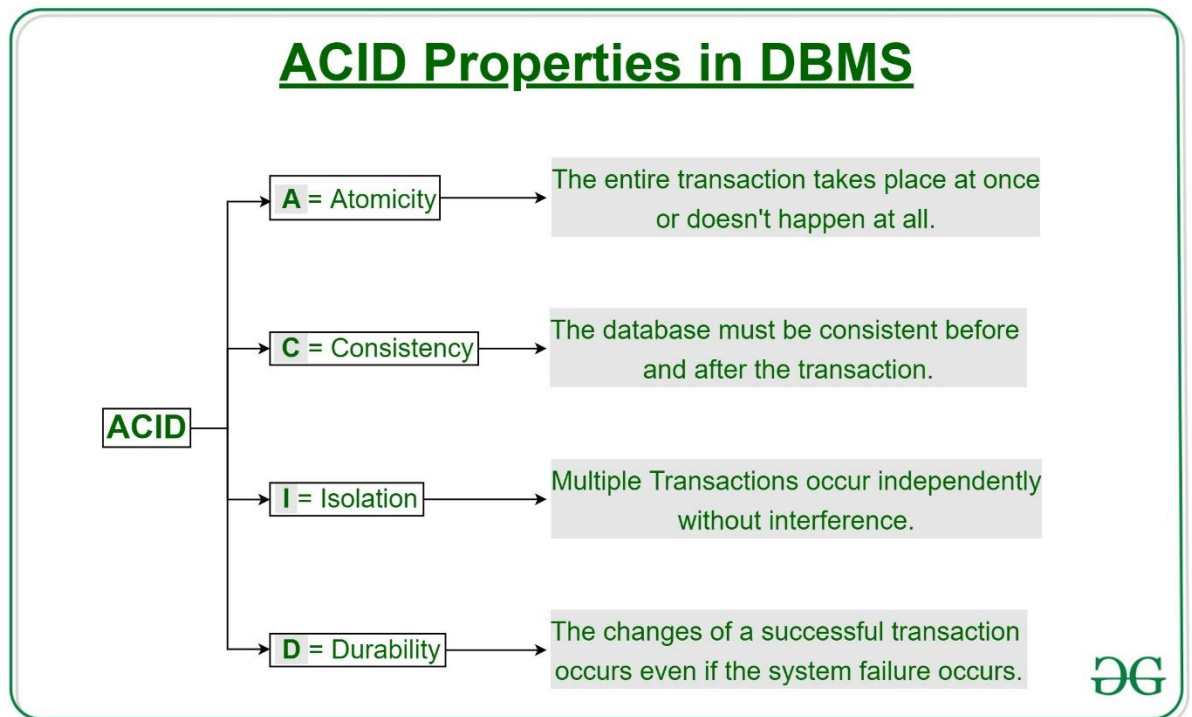
Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

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

# 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 i.e. 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	
<b>T1</b>	<b>T2</b>
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

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

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

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

This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

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
4. **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 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.

```
CREATE TABLE persons (person_id int AUTO_INCREMENT PRIMARY key, fname
varchar(20), lname varchar(20), city varchar(20))
```

```
INSERT into persons (fname, lname, city) VALUES ('Ansh', 'Amrutiya', 'Rajkot')
```

```
INSERT into persons (fname, lname, city) VALUES ('Ansh', 'Amrutiya', null);
```

```
INSERT into persons (fname, lname, city) VALUES ('Ansh', null, null);
```

```
INSERT into persons (fname, lname, city) VALUES (null, null, null);
```

## NOT NULL on CREATE TABLE

```
drop table persons
```

```
CREATE TABLE persons (person_id int AUTO_INCREMENT PRIMARY key, fname
varchar(20) not null, lname varchar(20) not null, city varchar(20) not null)
```

```
INSERT into persons (fname, lname, city) VALUES ('Ansh', 'Amrutiya', 'Rajkot')
```

```
INSERT into persons (fname, lname, city) VALUES ('Ansh', 'Amrutiya', NULL)
```

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

## NOT NULL on ALTER TABLE

```
ALTER table persons add COLUMN age int
```

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

```
INSERT into persons (fname, lname, city, age) VALUES ('Brijesh', 'Miyatra', '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 on CREATE TABLE

```
drop table persons
```

```
CREATE table persons (person_id int AUTO_INCREMENT PRIMARY key, fname  
varchar(20) not null, lname varchar(20) not null, city varchar(20), email varchar(50))
```

```
INSERT into persons (fname, lname, city, email) values ('Ansh', 'Amrutiya', 'Rajkot',  
'demo@gmail.com'), ('krishil', 'trivedi', 'Surat', 'demo@gmail.com')
```

```
drop TABLE persons
```

```
CREATE table persons (person_id int AUTO_INCREMENT PRIMARY key, fname  
varchar(20) not null, lname varchar(20) not null, city varchar(20), email varchar(50),  
UNIQUE(email))
```

```
INSERT into persons (fname, lname, city, email) values ('Ansh', 'Amrutiya', 'Rajkot',  
'demo@gmail.com'), ('krishil', 'trivedi', 'Surat', 'demo@gmail.com')
```

```
#1062 - Duplicate entry 'demo@gmail.com' for key 'email'
```

```
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 (person_id int AUTO_INCREMENT PRIMARY key, fname  
varchar(20) not null, lname varchar(20) not null, city varchar(20), email varchar(50),  
CONSTRAINT unq_email UNIQUE(email))
```

```
INSERT into persons (fname, lname, city, email) values ('Ansh', 'Amrutiya', 'Rajkot',  
'demo@gmail.com'), ('krishil', 'trivedi', 'Surat', 'demo@gmail.com')
```

```
#1062 - Duplicate entry 'demo@gmail.com' for key 'unq_email'
```

drop table persons

## UNIQUE Constraint on ALTER TABLE

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

`ALTER TABLE Persons ADD UNIQUE (ID);`

```
CREATE table persons (person_id int AUTO_INCREMENT PRIMARY key, fname varchar(20), lname
varchar(20), city varchar(20), email varchar(50))
```

```
ALTER table persons ADD CONSTRAINT unq_email UNIQUE(email)
```

## DROP a UNIQUE Constraint

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

```
ALTER TABLE persons drop CONSTRAINT unq_email
```

---

The major drawback of unique constraint is unique filed allows null to insert in column

```
ALTER table persons ADD CONSTRAINT unq_email UNIQUE(email)
```

```
INSERT into persons (fname, lname, city, email) values ('Alan', 'Thomas', 'Morbi', NULL)
```

```
INSERT into persons (fname, lname, city, email) values ('Alan', 'Thomas', 'Morbi', NULL)
```

```
INSERT into persons (fname, lname, city, email) values ('Alan', 'Thomas', 'Morbi', NULL)
```

```
INSERT into persons (fname, lname, city, email) values ('Alan', 'Thomas', 'Morbi', NULL)
```

```
INSERT into persons (fname, lname, city, email) values ('Alan', 'Thomas', 'Morbi', NULL)
```

---

Create unique constraint for multiple columns

```
create TABLE villageList (villageID int AUTO_INCREMENT PRIMARY key, villageName varchar(20) not
null, cityName varchar(20) not null, taluka varchar(20) not null, district varchar(20) not null, state
varchar(20) not null, country varchar(20) not null, pincode int not null)
```

```
INSERT INTO `villagelist` (`villageID`, `villageName`, `cityName`, `taluka`, `district`, `state`, `country`,
`pincode`) VALUES (NULL, 'Nagarpur', 'Rajkot', 'Rajkot', 'Rajkot', 'Gujarat', 'India', '112233');
```

```
INSERT INTO `villagelist` (`villageID`, `villageName`, `cityName`, `taluka`, `district`, `state`, `country`,
`pincode`) VALUES (NULL, 'Nagarpur', 'Rajkot', 'Rajkot', 'Rajkot', 'Gujarat', 'India', '112233');
```

```
Select * from villagelist
```

---

truncate table villagelist

ALTER table villagelist add CONSTRAINT unq\_villagename UNIQUE(villageName)

INSERT INTO `villagelist` (`villageID`, `villageName`, `cityName`, `taluka`, `district`, `state`, `country`, `pincode`) VALUES (NULL, '**Nagalpar**', 'Rajkot', 'Rajkot', 'Rajkot', 'Gujarat', 'India', '112233');

INSERT INTO `villagelist` (`villageID`, `villageName`, `cityName`, `taluka`, `district`, `state`, `country`, `pincode`) VALUES (NULL, '**Nagalpar**', 'Rajkot', 'Rajkot', 'Rajkot', 'Gujarat', 'India', '112233');

#1062 - Duplicate entry 'Nagalpar' for key 'unq\_villagename'

INSERT INTO `villagelist` (`villageID`, `villageName`, `cityName`, `taluka`, `district`, `state`, `country`, `pincode`) VALUES (NULL, '**Nagalpar**', 'Anjar', 'Bhuj', 'Rajkot', 'Gujarat', 'India', '112233');

ALTER table villagelist drop CONSTRAINT unq\_villagename

ALTER table villagelist add CONSTRAINT unq\_villagename UNIQUE(villageName, cityName, taluka, district)

---

INSERT INTO `villagelist` (`villageID`, `villageName`, `cityName`, `taluka`, `district`, `state`, `country`, `pincode`) VALUES (NULL, '**Nagalpar**', 'Rajkot', 'Rajkot', 'Rajkot', 'Gujarat', 'India', '112233');

#1062 - Duplicate entry '**Nagalpar-Rajkot-Rajkot-Rajkot**' for key 'unq\_villagename'

INSERT INTO `villagelist` (`villageID`, `villageName`, `cityName`, `taluka`, `district`, `state`, `country`, `pincode`) VALUES (NULL, '**Nagalpar**', 'Anjar', 'Bhuj', 'Rajkot', 'Gujarat', 'India', '112233');

# 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).

---

Example without primary key

drop table persons

```
CREATE TABLE persons (person_id int, fname varchar(20) not null, lname varchar(20) not null, city varchar(20) not null, email varchar(50) not null)
```

```
INSERT into persons (person_id, fname, lname, city, email) VALUES (1, 'Krishil', 'trivedi', 'Rajkot', 'krishil@gmail.com')
```

```
INSERT into persons (person_id, fname, lname, city, email) VALUES (1, 'Kaushik', 'trivedi', 'Rajkot', 'kaushik@gmail.com');
```

```
Select * from persons
```

Drop table persons

---

```
CREATE TABLE persons (person_id int, fname varchar(20) not null, lname varchar(20) not null, city varchar(20) not null, email varchar(50) not null, PRIMARY key(person_id))
```

```
INSERT into persons (person_id, fname, lname, city, email) VALUES (1, 'Krishil', 'trivedi', 'Rajkot', 'krishil@gmail.com')
```

```
INSERT into persons (person_id, fname, lname, city, email) VALUES (1, 'Kaushik', 'trivedi', 'Rajkot', 'kaushik@gmail.com');
```

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

drop TABLE persons

```
CREATE TABLE persons (person_id int, fname varchar(20) not null, lname varchar(20) not null, city varchar(20) not null, email varchar(50) not null, CONSTRAINT pk_pid PRIMARY key(person_id))
```

```
INSERT into persons (person_id, fname, lname, city, email) VALUES (1, 'Krishil', 'trivedi', 'Rajkot', 'krishil@gmail.com');
```

```
INSERT into persons (person_id, fname, lname, city, email) VALUES (1, 'Kaushik', 'trivedi', 'Rajkot', 'kaushik@gmail.com');
```

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

ALTER TABLE persons drop PRIMARY key

---

#### PRIMARY KEY on ALTER TABLE

ALTER TABLE persons add CONSTRAINT pk\_pid PRIMARY key (person\_id)

ALTER TABLE persons drop PRIMARY key

---

alter table persons add CONSTRAINT pk\_pid\_fname PRIMARY key(person\_id, fname)

INSERT into persons (person\_id, fname, lname, city, email) VALUES (1, '**Krishil**', 'trivedi', 'Rajkot', 'krishil@gmail.com');

INSERT into persons (person\_id, fname, lname, city, email) VALUES (1, '**Kaushik**', 'trivedi', 'Rajkot', 'krishil@gmail.com');

INSERT into persons (person\_id, fname, lname, city, email) VALUES (1, 'Kaushik', 'trivedi', 'Rajkot', 'krishil@gmail.com');

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

INSERT into persons (person\_id, fname, lname, city, email) VALUES (1, '**Kritesh**', 'trivedi', 'Rajkot', 'krishil@gmail.com');

# 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
5		23456	1
6		23457	3
7		11223	2
8		19876	4

- **Error Because PersonID 4 is not exist in person table**

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 feesInformation (feesid int AUTO_INCREMENT PRIMARY key, roll int, paymentdate date, amount int, paymentmode varchar(20))
```

```
INSERT INTO `feesinformation` (`feesid`, `roll`, `paymentdate`, `amount`, `paymentmode`) VALUES (NULL, '29', '2024-02-27', '1000', 'Cash');
```

---

**Above query break the links between students table and feesinformation table because student table does not contains any record with roll no. 29.**

---

```
drop table feesinformation
```

```
create table feesInformation (feesid int AUTO_INCREMENT PRIMARY key, roll int, paymentdate date, amount int, paymentmode varchar(20), FOREIGN key (roll) REFERENCES students(roll))
```

```
INSERT INTO `feesinformation` (`feesid`, `roll`, `paymentdate`, `amount`, `paymentmode`) VALUES (NULL, '29', '2024-02-27', '1000', 'Cash');
```

---

```
#1452 - Cannot add or update a child row: a foreign key constraint fails
(`1121_2324`.`feesinformation`, CONSTRAINT `feesinformation_ibfk_1` FOREIGN
KEY (`roll`) REFERENCES `students` (`roll`))
```

---

```
INSERT INTO `feesinformation` (`feesid`, `roll`, `paymentdate`, `amount`, `paymentmode`) VALUES (NULL, '9', '2024-02-27', '1000', 'Cash');
```

---

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

```
drop TABLE feesinformation
```

```
create table feesInformation (feesid int AUTO_INCREMENT PRIMARY key, roll int, paymentdate date, amount int, paymentmode varchar(20), CONSTRAINT fk_roll FOREIGN key (roll) REFERENCES students(roll))
```

```
INSERT INTO `feesinformation` (`feesid`, `roll`, `paymentdate`, `amount`, `paymentmode`) VALUES (NULL, '29', '2024-02-27', '1000', 'Cash');
```

---

```
#1452 - Cannot add or update a child row: a foreign key constraint fails
(`1121_2324`.`feesinformation`, CONSTRAINT `fk_roll` FOREIGN KEY (`roll`)
REFERENCES `students` (`roll`))
```

## FOREIGN KEY on ALTER TABLE

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

```
drop TABLE feesinformation
```

```
create table feesInformation (feesid int AUTO_INCREMENT PRIMARY key, roll int, paymentdate date, amount int, paymentmode varchar(20))
```

```
ALTER TABLE feesinformation add CONSTRAINT fk_roll FOREIGN key (roll) REFERENCES students(roll)
```

---

```
INSERT INTO `feesinformation` (`feesid`, `roll`, `paymentdate`, `amount`, `paymentmode`) VALUES (NULL, '9', '2024-02-27', '1000', 'Cash');
```

```
INSERT INTO `feesinformation` (`feesid`, `roll`, `paymentdate`, `amount`, `paymentmode`) VALUES (NULL, '49', '2024-02-27', '1000', 'Cash')
```

---

## DROP a FOREIGN KEY Constraint

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

```
ALTER table feesinformation drop CONSTRAINT fk_roll
```



# 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

The following SQL creates a `CHECK` constraint on the "Age" column when the "Persons" table is created. The `CHECK` constraint ensures that the age of a person must be 18, or older:

```
CREATE TABLE Persons ( ID int NOT NULL, LastName varchar(255) NOT NULL,
FirstName varchar(255), Age int, CHECK (Age>=18));
```

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

```
CREATE TABLE Persons ( ID int NOT NULL, LastName varchar(255) NOT NULL,
FirstName varchar(255), Age int, City varchar(255), CONSTRAINT CHK_Person CHECK
(Age>=18 AND City='Sandnes'));
```

```
drop table persons
```

```
create TABLE persons (personid int AUTO_INCREMENT PRIMARY key, fname
varchar(20) not null, lname varchar(20) not null, city varchar(20) not null, age int, CHECK
(age >= 18))
```

```
INSERT INTO `persons` (`personid`, `fname`, `lname`, `city`, `age`) VALUES (NULL,
'brijesh', 'miatra', 'Rajkot', '19');
```

```
INSERT INTO `persons` (`personid`, `fname`, `lname`, `city`, `age`) VALUES (NULL,
'ansh', 'amrutiya', 'Rajkot', '17');
```

```
#4025 - CONSTRAINT `CONSTRAINT_1` failed for `1121_2324`.`persons`
```

---

```
drop table persons
```

```
create TABLE persons (personid int AUTO_INCREMENT PRIMARY key, fname varchar(20) not null,
lname varchar(20) not null, city varchar(20) not null, age int, CONSTRAINT check_age CHECK (age >=
18))
```

```
INSERT INTO `persons` (`personid`, `fname`, `lname`, `city`, `age`) VALUES (NULL, 'brijesh', 'miatra', 'Rajkot', '19');
```

```
INSERT INTO `persons` (`personid`, `fname`, `lname`, `city`, `age`) VALUES (NULL, 'ansh', 'amrutiya', 'Rajkot', '17');
```

```
#4025 - CONSTRAINT `check_age` failed for `1121_2324`.`persons`
```

## DROP a CHECK Constraint

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

```
ALTER TABLE persons drop CONSTRAINT check_age
```

## CHECK on ALTER TABLE

To create a `CHECK` constraint on the "Age" column when the table is already created, use the following SQL:

```
ALTER TABLE persons add CONSTRAINT check_age CHECK (age >= 18)
```

# 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

drop TABLE persons

```
CREATE TABLE persons (personid int AUTO_INCREMENT PRIMARY key, fname
varchar(20) not null, lname varchar(20) not null, email varchar(50) not null, city varchar(20)
DEFAULT 'Rajkot', created_at timestamp DEFAULT CURRENT_TIMESTAMP)
```

```
INSERT INTO `persons` (`personid`, `fname`, `lname`, `email`, `city`, `created_at`)
VALUES (NULL, 'alan', 'thomas', 'alan@gmail.com', 'Rajkot', current_timestamp());
```

```
INSERT INTO `persons` (`personid`, `fname`, `lname`, `email`) VALUES (NULL, 'alan',
'thomas', 'alan@gmail.com');
```

```
INSERT INTO `persons` (`personid`, `fname`, `lname`, `email`, `city`) VALUES (NULL,
'alan', 'thomas', 'alan@gmail.com', 'Morbi');
```

### DEFAULT on ALTER TABLE

```
ALTER TABLE persons ALTER email set DEFAULT 'demo@gmail.com'
```

```
INSERT INTO persons (fname, lname) VALUES ('Krishil', 'Trivedi')
```

### DROP a DEFAULT Constraint

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

```
ALTER TABLE persons ALTER email drop DEFAULT
```

```
INSERT INTO persons (fname, lname) VALUES ('Krishil', 'Trivedi')
```

**Warning: #1364 Field 'email' doesn't have a default value**

# 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 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, ...);
```

## DROP INDEX Statement

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

```
ALTER TABLE table_name  
DROP INDEX index_name;
```

```
create index index_fname on students(fname);
```

```
drop index index_fname on students
```

# 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.

The following SQL statement defines the "Personid" column to be an auto-increment primary key field in the "Persons" table:

```
CREATE TABLE Persons (  
    Personid int NOT NULL AUTO_INCREMENT,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    PRIMARY KEY (Personid)  
);
```

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

```
INSERT into persons (fname, lname, city) values ('Meet', 'Sinojiya', 'Morbi')
```

```
INSERT into persons (fname, lname, city) values ('Brijesh', 'Miatra', 'Rajkot');
```

```
Select * from persons
```

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

```
ALTER TABLE persons AUTO_INCREMENT = 1100
```

```
INSERT into persons (fname, lname, city) VALUES ('Ansh', 'Amrutiya', 'Rajkot')
```

# 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!

## Working with Dates

```
SELECT * from students WHERE dateofbirth = '2004-02-12'
```

```
SELECT * from students WHERE dateofbirth BETWEEN '2001-01-01' and '2005-12-31';
```

```
SELECT * from students WHERE dateofbirth not BETWEEN '2001-01-01' and '2005-12-31';
```

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

```
SELECT * from students WHERE admissiondata = '2024-02-14'
```

```
SELECT * from students WHERE admissiondata = '2024-02-14 12:04:08'
```

```
SELECT * from students WHERE admissiondata like '2024-02-14%';
```

```
SELECT * from students WHERE date(admissiondata) = '2024-02-14';
```

# 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;
```

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

```
create view v1 as SELECT * from students
```

```
SELECT * from v1
```

```
SELECT * from v1 WHERE roll = 1;
```

---

Query without join

```
SELECT students.roll, students.fname, students.lname, students.city, students.phone, students.email,  
students.gender, students.dateofbirth, marks.total, marks.result, attendance.absents,  
attendance.presents from students inner join marks on students.roll = marks.roll inner join  
attendance on students.roll = attendance.roll;
```

---

Query with view

```
Create or repalce view getStudents AS SELECT students.roll, students.fname, students.lname,  
students.city, students.phone, students.email, students.gender, students.dateofbirth, marks.total,  
marks.result, attendance.absents, attendance.presents from students inner join marks on  
students.roll = marks.roll inner join attendance on students.roll = attendance.roll;
```

```
SELECT * from getstudents
```

```
SELECT * from getstudents WHERE roll = 1;
```

## MySQL Dropping a View

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

DROP VIEW Syntax

`DROP VIEW` *view\_name*;

DROP view v1

DROP view getstudents



# 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)	<b>A FIXED length</b> string (can contain letters, numbers, and special characters). The <i>size</i> parameter specifies the column length in characters - can be from 0 to <b>255</b> . Default is 1
VARCHAR(size)	<b>A VARIABLE length</b> 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 <b>65535</b>
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( <i>p</i> )	A floating point number. MySQL uses the <i>p</i> value to determine whether to use FLOAT or DOUBLE for the resulting data type. If <i>p</i> is from 0 to 24, the data type becomes FLOAT(). If <i>p</i> is from 25 to 53, the data type becomes DOUBLE()
DOUBLE( <i>size</i> , <i>d</i> )	A normal-size 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
DOUBLE PRECISION( <i>size</i> , <i>d</i> )	
DECIMAL( <i>size</i> , <i>d</i> )	An exact fixed-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. The maximum number for <i>size</i> is 65. The maximum number for <i>d</i> is 30. The default value for <i>size</i> is 10. The default value for <i>d</i> is 0.
DEC( <i>size</i> , <i>d</i> )	Equal to DECIMAL( <i>size</i> , <i>d</i> )

**Note:** All the numeric data types may have an extra option: UNSIGNED or ZEROFILL. If you add the UNSIGNED option, MySQL disallows negative values for the column. If you add the ZEROFILL option, MySQL automatically also adds the UNSIGNED attribute to the column.

#### 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 Functions