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?

SQL is the standard language for dealing with Relational Databases.

SQL is used to insert, search, update, and delete database records.

How to Use SQL

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

Example

```
SELECT * FROM Customers;
```

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.

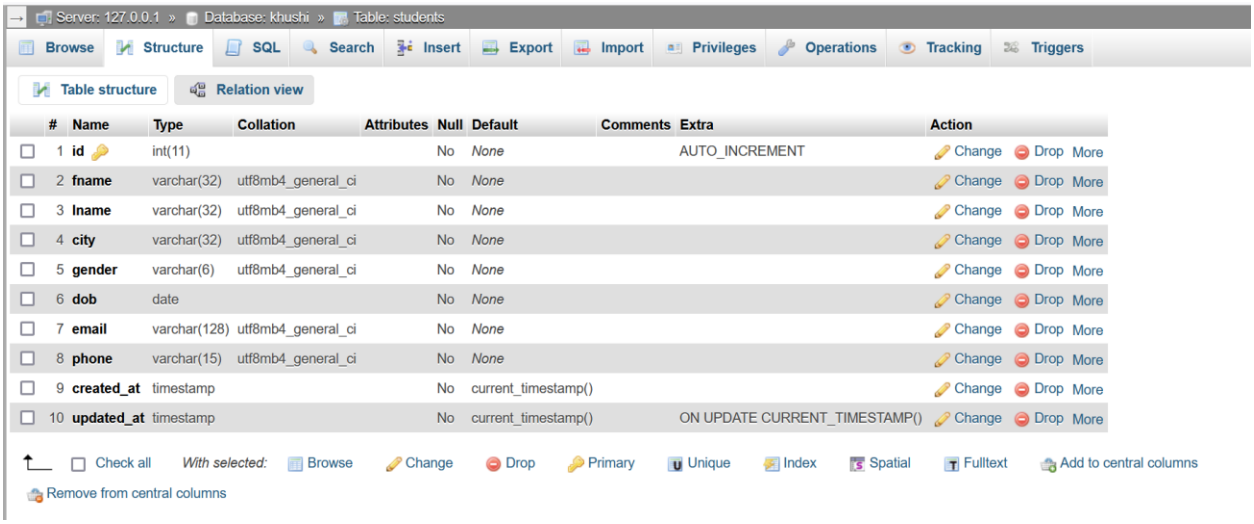
In this tutorial, we will use semicolon at the end of each SQL statement.

Some of The Most Important SQL Commands

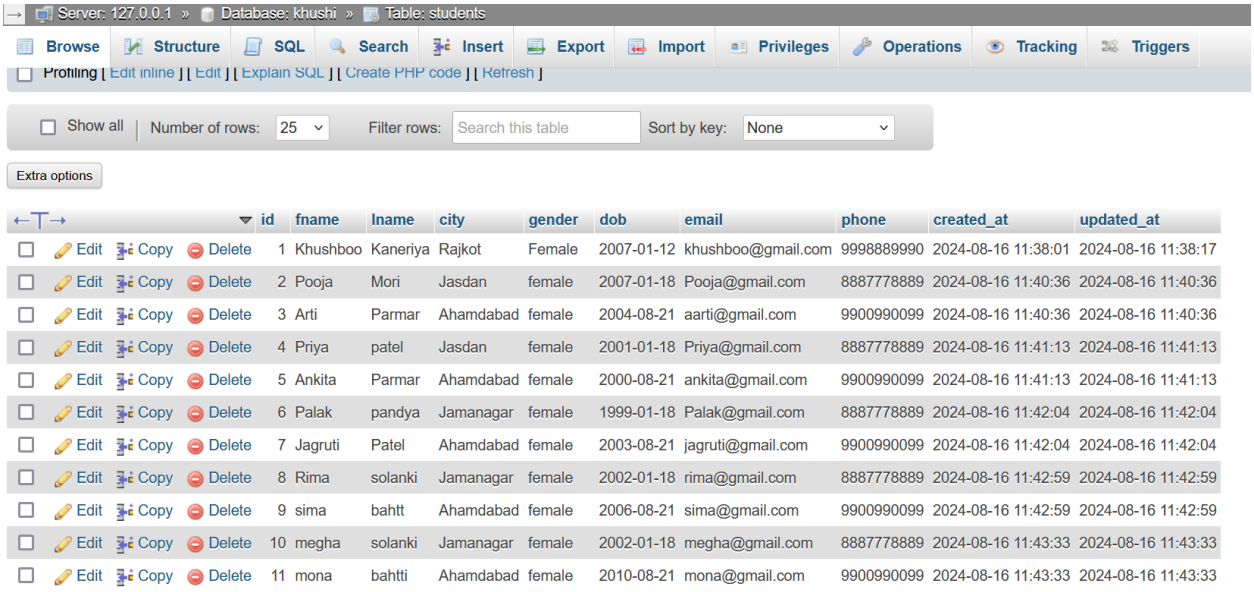
- `SELECT` - extracts data from a database
- `UPDATE` - updates data in a database
- `DELETE` - deletes data from a database
- `INSERT INTO` - inserts new data into a database
- `CREATE DATABASE` - creates a new database
- `ALTER DATABASE` - modifies a database
- `CREATE TABLE` - creates a new table
- `ALTER TABLE` - modifies a table

- `DROP TABLE` - deletes a table
- `CREATE INDEX` - creates an index (search key)
- `DROP INDEX` - deletes an index

Create Database and table as following



Add some data in students table



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*;

```
SELECT id, fname, lname FROM students
```

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 * 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 DISTINCT(city) FROM students
```

SELECT Example Without DISTINCT

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

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

MySQL WHERE Clause

The MySQL WHERE Clause

The `WHERE` clause is used to filter records.

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

WHERE Syntax

SELECT *column1*, *column2, ...* FROM *table_name* WHERE *condition*;

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 id, fname, lname, city 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 id, fname, lname, city FROM students WHERE id > 5
```

```
SELECT id, fname, lname, city FROM students WHERE id = 5;
```

```
SELECT id, fname, lname, city FROM students WHERE id >= 5;
```

```
SELECT id, fname, lname, city FROM students WHERE id < 5;
```

```
SELECT id, fname, lname, city FROM students WHERE id <= 5;
```

```
SELECT id, fname, lname, city FROM students WHERE id <> 5;
```

```
SELECT id, fname, lname, city FROM students WHERE not id = 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 ...;
```

OR Syntax

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

NOT Syntax

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

```
SELECT * FROM students WHERE id = 1

SELECT * FROM students WHERE id = 1 and city = 'Surat';

SELECT * FROM students WHERE id = 1 or city = 'Surat';

SELECT * FROM students WHERE not city = 'Surat';
```

Combining AND, OR and NOT

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

```
SELECT * FROM students WHERE id = 1 and city = 'Rajkot' or city = 'Surat';

SELECT * FROM students WHERE id = 1 and (city = 'Rajkot' or city = 'Surat');

SELECT * FROM students WHERE id = 1 and (not city = 'Rajkot' and not city = 'Surat');
```

MySQL ORDER BY Keyword

The MySQL ORDER BY Keyword

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

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

ORDER BY Syntax

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

SELECT * FROM students

SELECT * FROM students ORDER by (fname);

SELECT * FROM students ORDER by (fname) DESC;

ORDER BY Several Columns Example

SELECT * FROM students ORDER by fname, city;

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*, ...);

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 (fname, lname, city, gender, dob, email, phone) VALUES ('Dhruvisha', 'Bhatt', 'Junagadh', 'female', '2008-09-09', 'dhruvisha@gmail.com', '9900999999')

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) VALUES ('Devangi', 'Dave')

Warning: #1364 Field 'city' doesn't have a default value
Warning: #1364 Field 'gender' doesn't have a default value
Warning: #1364 Field 'dob' doesn't have a default value
Warning: #1364 Field 'email' doesn't have a default value
Warning: #1364 Field 'phone' doesn't have a default value

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!

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.

SELECT * FROM `students` WHERE city is null

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

```
ALTER TABLE `students` CHANGE `city` `city` VARCHAR(32) NULL;
```

SELECT * FROM `students` WHERE city is null

SELECT * FROM `students` WHERE city is not null;

The IS NULL Operator

The `IS NULL` operator is used to test for empty values (NULL values).

The IS NOT NULL Operator

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

MySQL UPDATE Statement

The MySQL UPDATE Statement

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

UPDATE Syntax

UPDATE *table_name* SET *column1* = *value1*, *column2* = *value2*, ... WHERE *condition*;

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

```
UPDATE students set gender = 'Female'
```

```
UPDATE students set city = 'Rajkot' WHERE id = 10
```

UPDATE Multiple Records

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

```
UPDATE students set city = 'Rajkot', gender = 'female', phone = '9998887770' WHERE id = 10;
```

Update Warning!

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

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 * FROM students

SELECT * FROM students LIMIT 5;

SELECT * FROM students LIMIT 5 OFFSET 5;

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

MySQL LIMIT Examples

The following SQL statement selects the first three records from the "Customers" table:

Example

```
SELECT * FROM Customers LIMIT 3;
```

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

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

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

Example

```
SELECT * FROM Customers LIMIT 3 OFFSET 3;

SELECT * from students WHERE city = 'Ahamdabad';

SELECT * from students WHERE city = 'Ahamdabad' LIMIT 2;
```

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 min(dob) FROM students;

SELECT max(dob) FROM students;

SELECT MAX(id) FROM students

SELECT min(id) FROM students;

SELECT MIN(dob) FROM students;

SELECT MIN(dob) as "Oldest Student" FROM students;

SELECT MAX(dob) as "Youngest Student" 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(id) FROM students
```

```
SELECT COUNT(id) FROM students WHERE city = 'Ahamdabad';
```

```
SELECT COUNT(id) FROM students WHERE city <> 'Ahamdabad';
```

```
SELECT sum(id) FROM students
```

```
SELECT avg(id) FROM students
```

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!

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 'k%'

SELECT * from students WHERE fname like '%k';

SELECT * from students WHERE fname like '%k%';

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

SELECT * from students WHERE fname like '_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"

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

```
SELECT * from students WHERE fname like 'a___';

SELECT * from students WHERE fname like 'a___%';

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

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:

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, ...);
```

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

```
SELECT * from students WHERE city in('surat',  'Rajkot', 'Ahamdabad');
```

```
SELECT * from students WHERE city not in('surat',  'Rajkot', 'Ahamdabad');
```

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 id BETWEEN 1 and 5;
```

```
SELECT * from students WHERE dob BETWEEN '2000-01-01' and '2007-12-31';
```

```
SELECT * from students WHERE fname BETWEEN 'arti' and 'palak'
```

```
SELECT * from students WHERE fname  not BETWEEN 'arti' and 'palak';
```

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 fname as "First Name", lname as "Last Name" 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 id as ID, fname as "First Name", lname as "Last Name" from students;

SELECT id as "Student ID", fname as "First Name", lname as "Last Name" from students;

SELECT concat_ws(" ", id, fname, lname, city, email, phone, dob) as "Student Details" FROM students;

SELECT concat_ws(" _ ", id, fname, lname, city, email, phone, dob) as "Student Details" FROM students;

Alias for Tables Example

SELECT students.id, students.fname, students.lname, students.city, students.gender, students.dob, students.email, students.phone, attendance.absents, attendance.presents from students, attendance

SELECT students.id, students.fname, students.lname, students.city, students.gender, students.dob, students.email, students.phone, attendance.absents, attendance.presents from students, attendance WHERE students.id = attendance.stduent_id;

SELECT s.id, s.fname, s.lname, s.city, s.gender, s.dob, s.email, s.phone, a.absents, a.presents from students s, attendance a WHERE s.id = a.stduent_id;

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 COUNT(id) FROM students WHERE city = 'Rajkot';
```

```
SELECT COUNT(id) as "Students From Rajkot" FROM students WHERE city = 'Rajkot';
```

MySQL Joins

MySQL Joining Tables

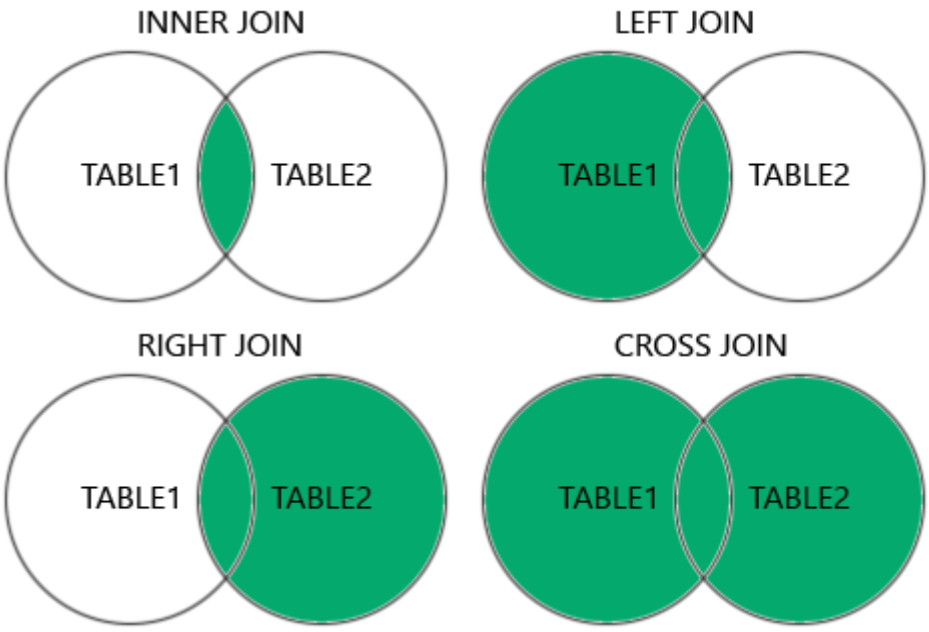
A `JOIN` clause is used to combine rows from two or more tables, based on a related column between them.

SELECT students.*, attendance.absents, attendance.presents from students INNER join attendance

SELECT students.*, attendance.absents, attendance.presents from students INNER join attendance on students.id = attendance.stduent_id;

Supported Types of Joins in MySQL

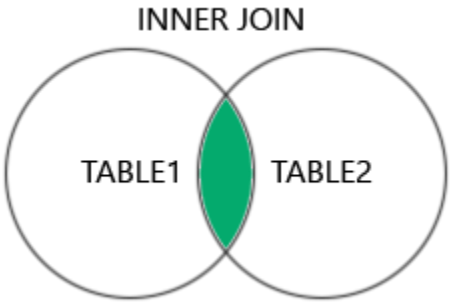
- `INNER JOIN`: Returns records that have matching values in both tables
- `LEFT JOIN`: Returns all records from the left table, and the matched records from the right table
- `RIGHT JOIN`: Returns all records from the right table, and the matched records from the left table
- `CROSS JOIN`: Returns all 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 students.id, students.fname, students.lname, students.city, students.gender, students.dob, students.email, students.phone, attendance.absents, attendance.presents from students INNER JOIN attendance on students.id = attendance.stduent_id

Note: The `INNER JOIN` keyword selects all rows from both tables as long as there is a match between the columns. If there are records in the "Orders" table that do not have matches in "Customers", these orders will not be shown!

JOIN Three Tables

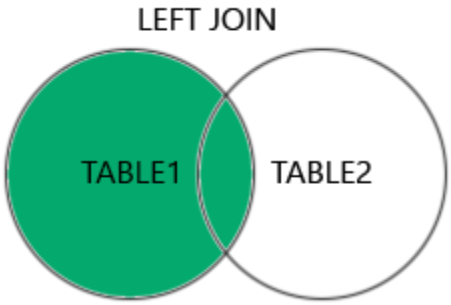
SELECT students.id, students.fname, students.lname, students.city, students.gender, students.dob, students.email, students.phone, attendance.absents, attendance.presents, marks.total, marks.result from students INNER JOIN attendance on students.id = attendance.stduent_id INNER JOIN marks on students.id = marks.stduent_id;

SELECT s.id, s.fname, s.lname, s.city, s.gender, s.dob, s.email, s.phone, a.absents, a.presents, m.total, m.result from students s INNER join attendance a on s.id = a.stduent_id INNER join marks m on s.id = m.stduent_id

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*;

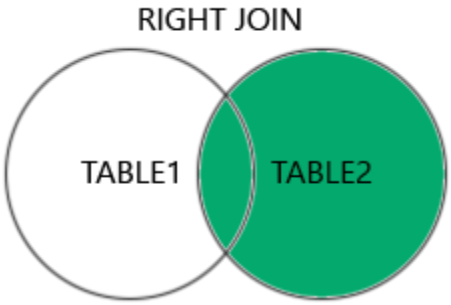
SELECT students.id, students.fname, students.lname, students.city, students.gender, students.dob, students.email, students.phone, attendance.absents, attendance.presents FROM students LEFT join attendance on students.id = attendance.stduent_id

Note: The `LEFT JOIN` keyword returns all records from the left table (Customers), even if there are no matches in the right table (Orders).

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*;

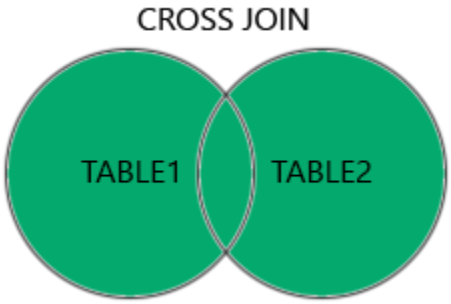
SELECT students.id, students.fname, students.lname, students.city, students.gender, students.dob, students.email, students.phone, attendance.absents, attendance.presents FROM students RIGHT join attendance on students.id = attendance.stduent_id;

Note: The `RIGHT JOIN` keyword returns all records from the right table (Employees), even if there are no matches in the left table (Orders).

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!

```
SELECT students.*, attendance.absents, attendance.presents FROM students CROSS JOIN attendance;
```

Note: The `CROSS JOIN` keyword returns all matching records from both tables whether the other table matches or not. So, if there are rows in "Customers" that do not have matches in "Orders", or if there are rows in "Orders" that do not have matches in "Customers", those rows will be listed as well.

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.id = attendance.stduent_id;
```

MySQL Self Join

MySQL Self Join

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

Self Join Syntax

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

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

SELECT s1.id, s1.fname, s1.lname, s1.city FROM students s1, students s2 WHERE s1.city = s2.city and s1.id <> s2.id;

SELECT s1.id, s1.fname, s1.lname, s1.city FROM students s1, students s2 WHERE s1.city = s2.city and s1.id <> s2.id ORDER by s1.city;

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

```
SELECT * FROM students WHERE City = 'Rajkot'

SELECT * FROM students_1 WHERE City = 'surat';
```

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

UNION

SELECT * FROM students_1 WHERE City = 'surat';
```

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

UNION

SELECT * FROM students_1 WHERE City = 'rajkot';
```

UNION ALL Syntax

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

```
SELECT column_name(s) FROM table1 UNION ALL SELECT column_name(s) FROM table2;
```

Note: The column names in the result-set are usually equal to the column names in the first `SELECT` statement.

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

UNION all

SELECT * FROM students_1 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 city, COUNT(city) FROM student

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;

SELECT students.*, attendance.absents, attendance.presents, sum(attendance.absents+attendance.presents) as "Total Days" FROM students INNER join attendance on students.id = attendance.stduent_id GROUP by (students.id);

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 city, COUNT(id) FROM students GROUP by (city)

SELECT city, COUNT(id) FROM students GROUP by (city) WHERE count(id) >= 3;

SELECT city, COUNT(id) FROM students GROUP by (city) HAVING count(id) >= 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.

EXISTS Syntax

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

SELECT students.* FROM students where EXISTS (SELECT marks.stduent_id FROM marks WHERE marks.stduent_id = students.id 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);
```

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 students.id, students.fname, students.lname FROM students WHERE id = any (SELECT
marks.stduent_id from marks WHERE marks.result = 'fail')
```

SQL ALL Examples

The following SQL statement lists ALL the product names:

SELECT * from students WHERE true;

SELECT all fname from students WHERE true;

SELECT fname, lname FROM students WHERE id = all (SELECT marks.stduent_id from marks WHERE marks.result = 'fail');

```
SELECT students.* from students WHERE id = all (SELECT attendance.stduent_id
FROM attendance WHERE attendance.absents = 150);
```

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 students_1 (fname, lname, city, email, phone, gender, dob) SELECT fname, lname, city, email, phone, gender, dob FROM students WHERE City = 'Rajkot'
```

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

Copy only some columns from one table into another table:

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

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 id, fname, lname, city, email, phone, gender, dob from students
```

```
SELECT id, fname, lname, city, case
        WHEN city = 'Rajkot' THEN 'Home Town'
        WHEN city = 'Ahamdabad' THEN 'Far From Home'
        WHEN city = 'Surat' THEN 'Too much Far From Home'
end as "Distance From Home", email, phone, gender, dob from students;
```

```
SELECT id, fname, lname, city, case
        WHEN city = 'Rajkot' THEN 'Home Town'
        WHEN city = 'Ahamdabad' THEN 'Far From Home'
        WHEN city = 'Surat' THEN 'Too much Far From Home'
        else 'Unkonwn Distance'
end as "Distance From Home", email, phone, gender, dob from students;
```

MySQL NULL Functions

MySQL IFNULL() and COALESCE() Functions

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 stduent_id, absents, presents from attendance

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

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

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

MySQL COALESCE() Function

Or we can use the `COALESCE()` function, like this:

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

MySQL Comments

MySQL Comments

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

Single Line Comments

Single line comments start with `--`.

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

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

Example

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

The following example uses a single-line comment to ignore the end of a line:

Example

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

-- COALESCE function replace null values with specified value

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

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:

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

-- Example of Multiline / block comment

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

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 testdb

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

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*;

DROP database testdb;

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

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

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.).

```
CREATE TABLE users (id int AUTO_INCREMENT PRIMARY key, fname varchar(20), lname varchar(20), email varchar(128), phone varchar(15), gender varchar(10), city varchar(20), state varchar(20), country varchar(20))
```

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 students_backup_1 as SELECT * from students
```

```
CREATE TABLE students_backup_2 as SELECT id, fname, lname, city from students
```

MySQL DROP TABLE Statement

The MySQL DROP TABLE Statement

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

Syntax

```
DROP TABLE table_name;
```

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

MySQL DROP TABLE Example

The following SQL statement drops the existing table "Shippers":

Example

```
DROP TABLE Shippers;
```

drop table students_backup_2

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!

```
DELETE from students_1 WHERE city = 'Rajkot';

DELETE FROM students_1 WHERE id = 1

DELETE FROM students_1
```

```
INSERT INTO `students_1` (`id`, `fname`, `lname`, `city`, `gender`, `dob`, `email`, `phone`, `created_at`, `updated_at`) VALUES (NULL, 'Devarshi', 'Mer', 'rajkot', 'male', '2007-01-12', 'demo@gmail.com', '9998889990', current_timestamp(), current_timestamp());
```

Auto increment id will be continue from last record before delete all the data from table

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_1

INSERT INTO `students_1` (`id`, `fname`, `lname`, `city`, `gender`, `dob`, `email`, `phone`, `created_at`, `updated_at`) VALUES (NULL, 'Devarshi', 'Mer', 'rajkot', 'male', '2007-01-12', 'demo@gmail.com', '9998889990', current_timestamp(), current_timestamp());
```

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 users add COLUMN userpassword varchar(64)

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 users drop column userpassword

ALTER TABLE users add column userpassword varchar(64) after email;

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 users MODIFY COLUMN fname varchar(32)

ALTER TABLE users add COLUMN dob date

ALTER TABLE users CHANGE dob date_of_Birth date

DROP COLUMN Example

Next, we want to delete the column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

Example

```
ALTER TABLE Persons
DROP COLUMN DateOfBirth;
```

```
ALTER TABLE users DROP COLUMN date_of_Birth
```

MySQL Constraints

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

Create Constraints

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

Syntax

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

MySQL Constraints

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

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

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

The following constraints are commonly used in SQL:

- `NOT NULL` - Ensures that a column cannot have a NULL value
- `UNIQUE` - Ensures that all values in a column are different
- `PRIMARY KEY` - A combination of a `NOT NULL` and `UNIQUE`. Uniquely identifies each row in a table
- `FOREIGN KEY` - Prevents actions that would destroy links between tables
- `CHECK` - Ensures that the values in a column satisfies a specific condition
- `DEFAULT` - Sets a default value for a column if no value is specified
- `CREATE INDEX` - Used to create and retrieve data from the database very quickly

MySQL NOT NULL Constraint

MySQL NOT NULL Constraint

By default, a column can hold NULL values.

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

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

NOT NULL on CREATE TABLE

CREATE TABLE students (id int AUTO_INCREMENT PRIMARY key, fname varchar(20) **not null**, lname varchar(20) **not null**, city varchar(20))

INSERT INTO students (fname, lname, city) VALUES ('Demo', 'Text', 'Rajkot')

INSERT INTO students (fname, lname, city) VALUES ('Demo', 'Text', null);

INSERT INTO students (fname, lname, city) VALUES ('Demo', null, null);

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

NOT NULL on ALTER TABLE

ALTER TABLE students MODIFY COLUMN city varchar(20) NOT null

INSERT INTO students (fname, lname, city) VALUES ('Demo', 'Text', null);

```
#1048 - Column 'city' 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.

```
DROP TABLE students

CREATE TABLE students (id int AUTO_INCREMENT PRIMARY KEY, fname varchar(20) not null, lname varchar(20) not null, email varchar(64)  UNIQUE)


INSERT INTO students (fname, lname, email) VALUES ('Priya', 'Patel',  'priya@gmail.com')

INSERT INTO students (fname, lname, email) VALUES ('Priya', 'Patel',  'priya@gmail.com')

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


INSERT INTO students (fname, lname, email) VALUES ('Priya', 'Patel',  null);

INSERT INTO students (fname, lname, email) VALUES ('Priya', 'Patel',  null);

INSERT INTO students (fname, lname, email) VALUES ('Priya', 'Patel',  null);

INSERT INTO students (fname, lname, email) VALUES ('Priya', 'Patel',  null);

INSERT INTO students (fname, lname, email) VALUES ('Priya', 'Patel',  null);
```

UNIQUE Constraint on CREATE TABLE

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

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

drop TABLE students

CREATE TABLE students (id int AUTO_INCREMENT PRIMARY KEY, fname varchar(20) not null, lname varchar(20) not null, email varchar(64), phone varchar(15), UNIQUE (email))

drop TABLE students

CREATE TABLE students (id int AUTO_INCREMENT PRIMARY KEY, fname varchar(20) not null, lname varchar(20) not null, email varchar(64), phone varchar(15), CONSTRAINT **uniqueEmail** UNIQUE(email))

INSERT into students (fname, lname, email, phone) VALUES ('Ridhhi', 'Patel', 'riddhi@gmail.com', '9988990099')

INSERT into students (fname, lname, email, phone) VALUES ('Ridhhi', 'Patel', 'riddhi@gmail.com', '9988990099')

```
#1062 - Duplicate entry 'riddhi@gmail.com' for key 'uniqueEmail'
```

To name a `UNIQUE` constraint, and to define a `UNIQUE` 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,
   CONSTRAINT UC_Person UNIQUE (ID,LastName)
);
```

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 students **drop** CONSTRAINT uniqueEmail;

ALTER TABLE students **add** CONSTRAINT unqieEmail UNIQUE (email)

DROP a UNIQUE Constraint

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

create TABLE villagelist (villageid int AUTO_INCREMENT PRIMARY key, villagename varchar(20), taluka varchar(20), district varchar(20), state varchar(20), constraint unqiqeVillageName UNIQUE(villagename))

INSERT into villagelist (villagename, taluka, district, state) VALUES('Navagam', 'Rajkot', 'Rajkot', 'gujarat')

INSERT into villagelist (villagename, taluka, district, state) VALUES('Navagam', 'gondal', 'Rajkot', 'gujarat');

```
#1062 - Duplicate entry 'Navagam' for key 'unqiqeVillageName'
```

ALTER TABLE villagelist drop CONSTRAINT unqiqeVillageName

ALTER TABLE villagelist add CONSTRAINT unqiqeVillageName UNIQUE(villagename, taluka, district);

INSERT into villagelist (villagename, taluka, district, state) VALUES('Navagam', 'gondal', 'Rajkot', 'gujarat');

INSERT into villagelist (villagename, taluka, district, state) VALUES('Navagam', 'jamnagar', 'jamnagar', 'gujarat');

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).

CREATE TABLE students (id int AUTO_INCREMENT PRIMARY key, fname varchar(20), lname varchar(20), email varchar(64))

CREATE TABLE students (id int AUTO_INCREMENT, fname varchar(20), lname varchar(20), email varchar(64), PRIMARY key (id))

CREATE TABLE students (id int AUTO_INCREMENT, fname varchar(20), lname varchar(20), email varchar(64), CONSTRAINT pk_id PRIMARY key (id))

insert into students (id, fname, lname, email) values (1, 'khushboo', 'kaneriya', 'khushboo@kaneriya.com')

insert into students (id, fname, lname, email) values (1, 'khushboo', 'kaneriya', 'khushboo@kaneriya.com')

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

PRIMARY KEY on CREATE TABLE

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

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

To allow naming of a `PRIMARY KEY` constraint, and for defining a `PRIMARY KEY` 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,
    CONSTRAINT PK_Person PRIMARY KEY (ID,LastName)
);
```

Note: In the example above there is only ONE `PRIMARY KEY` (PK_Person). However, the VALUE of the primary key is made up of TWO COLUMNS (ID + LastName).

DROP a PRIMARY KEY Constraint

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

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

```
ALTER TABLE students drop PRIMARY key
```

PRIMARY KEY on ALTER TABLE

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

```
ALTER TABLE Persons
ADD PRIMARY KEY (ID);
```

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

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

```
ALTER TABLE students add CONSTRAINT pk_id PRIMARY key (id)
```

```
insert into students (id, fname, lname, email) values (1, 'khushboo', 'kaneriya',
'khushboo@kaneriya.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	LastName	FirstName	Age
1	Hansen	Ola	30
2	Svendson	Tove	23
3	Pettersen	Kari	20

Orders Table

OrderID	OrderNumber	PersonID
1	77895	3
2	44678	3
3	22456	2
4	24562	1

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

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

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

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

FOREIGN KEY on CREATE TABLE

The following SQL creates a `FOREIGN KEY` on the "PersonID" column when the "Orders" table is created:

```
CREATE TABLE Orders (
   OrderID int NOT NULL,
   OrderNumber int NOT NULL,
   PersonID int,
   PRIMARY KEY (OrderID),
   FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

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

```
CREATE TABLE Orders (
   OrderID int NOT NULL,
   OrderNumber int NOT NULL,
   PersonID int,
   PRIMARY KEY (OrderID),
   CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID)
   REFERENCES Persons(PersonID)
);
```

CREATE TABLE students (id int AUTO_INCREMENT PRIMARY key, fname varchar(20), lname varchar(20), city varchar(20), email varchar(64), phone varchar(15), gender varchar(10), CONSTRAINT UNIQUEfname UNIQUE(fname), CONSTRAINT UNIQUElname UNIQUE(lname))

CREATE TABLE students (id int AUTO_INCREMENT PRIMARY key, fname varchar(20), lname varchar(20), city varchar(20), email varchar(64), phone varchar(15), gender varchar(10))

INSERT into students (fname, lname, city, email, phone, gender) values ('Riya', 'Dave', 'rajkot', 'Riya@gmail.com', '9988990099', 'female')

INSERT into students (fname, lname, city, email, phone, gender) values ('Priya', 'Dave', 'rajkot', 'Priya@gmail.com', '9988990099', 'female'), ('Siya', 'Dave', 'rajkot', 'Siya@gmail.com', '9988990099', 'female'), ('smita', 'Patel', 'rajkot', 'smita@gmail.com', '9988990099', 'female'), ('dipti', 'Dave', 'rajkot', 'dipti@gmail.com', '9988990099', 'female'), ('sneha', 'Dave', 'rajkot', 'sneha@gmail.com', '9988990099', 'female');

CREATE TABLE attendance (id int PRIMARY key AUTO_INCREMENT, student_id int, absents int, presents int, CONSTRAINT fk_student_id FOREIGN key (student_id) REFERENCES students(id))

INSERT into attendance (student_id, absents, presents) VALUES (1, 111, 112)

INSERT into attendance (student_id, absents, presents) VALUES (2, 111, 122);

INSERT into attendance (student_id, absents, presents) VALUES (12, 111, 122);

```
#1452 - Cannot add or update a child row: a foreign key constraint fails
(`khushi`.`attendance`, CONSTRAINT `fk_student_id` FOREIGN KEY (`student_id`)
REFERENCES `students` (`id`))
```

DROP a FOREIGN KEY Constraint

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

```
ALTER TABLE Orders
DROP FOREIGN KEY FK_PersonOrder;
```

ALTER TABLE attendance drop CONSTRAINT fk_student_id

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:

```
ALTER TABLE Orders
ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

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

```
ALTER TABLE Orders
ADD CONSTRAINT FK_PersonOrder
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

ALTER TABLE attendance add CONSTRAINT fk_student_id FOREIGN KEY (student_id) REFERENCES students(id)

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')
);
```

```
CREATE table results (id int AUTO_INCREMENT PRIMARY KEY, student_id int, marks int, result varchar(4), CONSTRAINT fk_student_result FOREIGN key (student_id) REFERENCES students(id), CONSTRAINT checkMarks CHECK (marks >= 0 and marks <= 100))
```

```
INSERT into results (student_id, marks, result) VALUES (1, 43, 'pass')
```

```
INSERT into results (student_id, marks, result) VALUES (1, 413, 'pass');
```

```
#4025 - CONSTRAINT `checkMarks` failed for `khushi`.`results`
```

DROP a CHECK Constraint

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

```
ALTER TABLE Persons
DROP CHECK CHK_PersonAge;
```

```
ALTER TABLE results drop CHECK checkmarks
```

```
ALTER TABLE results drop CONSTRAINT checkmarks
```

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')
);
```

```
ALTER TABLE results add CONSTRAINT checkmarks CHECK (marks >= 0 and marks <= 100);
```

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

The following SQL sets a `DEFAULT` value for the "City" column when the "Persons" table is created:

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

The `DEFAULT` constraint can also be used to insert system values, by using functions like `CURRENT_DATE()`:

```
CREATE TABLE Orders (
   ID int NOT NULL,
   OrderNumber int NOT NULL,
   OrderDate date DEFAULT CURRENT_DATE()
);
```

```
create TABLE users (id int AUTO_INCREMENT PRIMARY key, fname varchar(20), lname varchar(20), city varchar(20) DEFAULT 'Rajkot')
```

```
INSERT into users (fname, lname) VALUES ('Khushboo', 'Kaneriya')
```

```
INSERT into users (fname, lname, city) VALUES ('Khushboo', 'Kaneriya', 'surat')
```

DROP a DEFAULT Constraint

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

```
ALTER TABLE Persons
ALTER City DROP DEFAULT;
```

```
ALTER TABLE users ALTER city DROP DEFAULT
```

```
INSERT into users (fname, lname) VALUES ('Khushboo', 'Kaneriya')
```

DEFAULT on ALTER TABLE

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

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

```
ALTER TABLE users ALTER city set DEFAULT 'Ahamdabad'
```

```
INSERT into users (fname, lname) VALUES ('Khushboo', 'Kaneriya')
```

```
ALTER TABLE users add COLUMN created_at timestamp DEFAULT CURRENT_TIMESTAMP
```

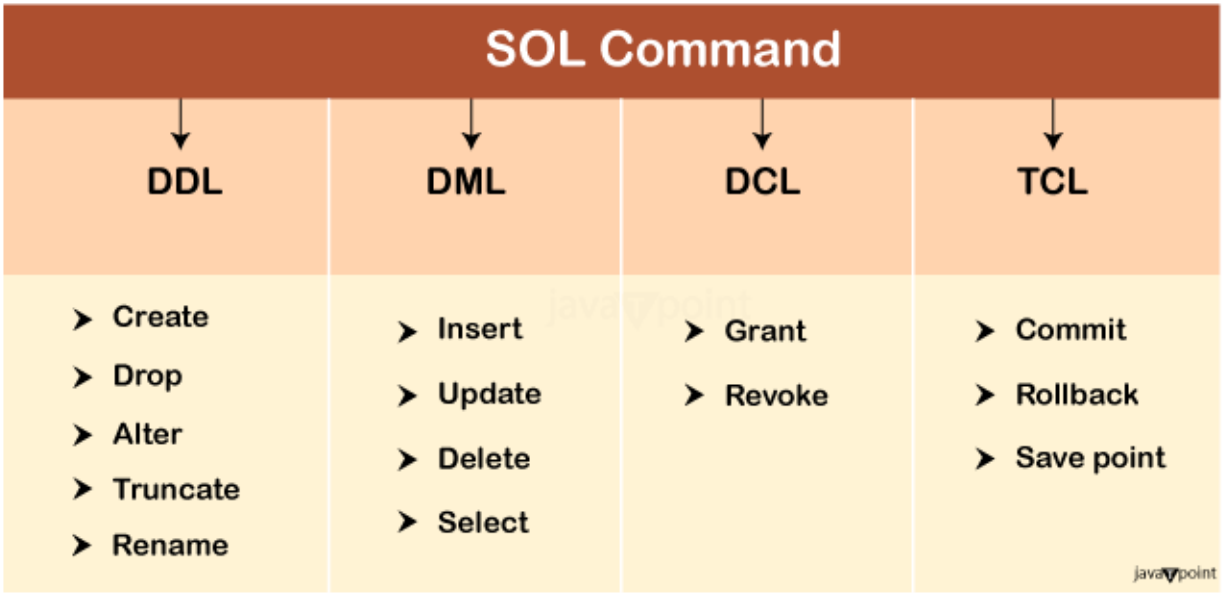
```
INSERT into users (fname, lname) VALUES ('Khushboo', 'Kaneriya')
```

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 four types of SQL commands: DDL, DML, DCL, TCL.



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.

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.

3. Data Control Language

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

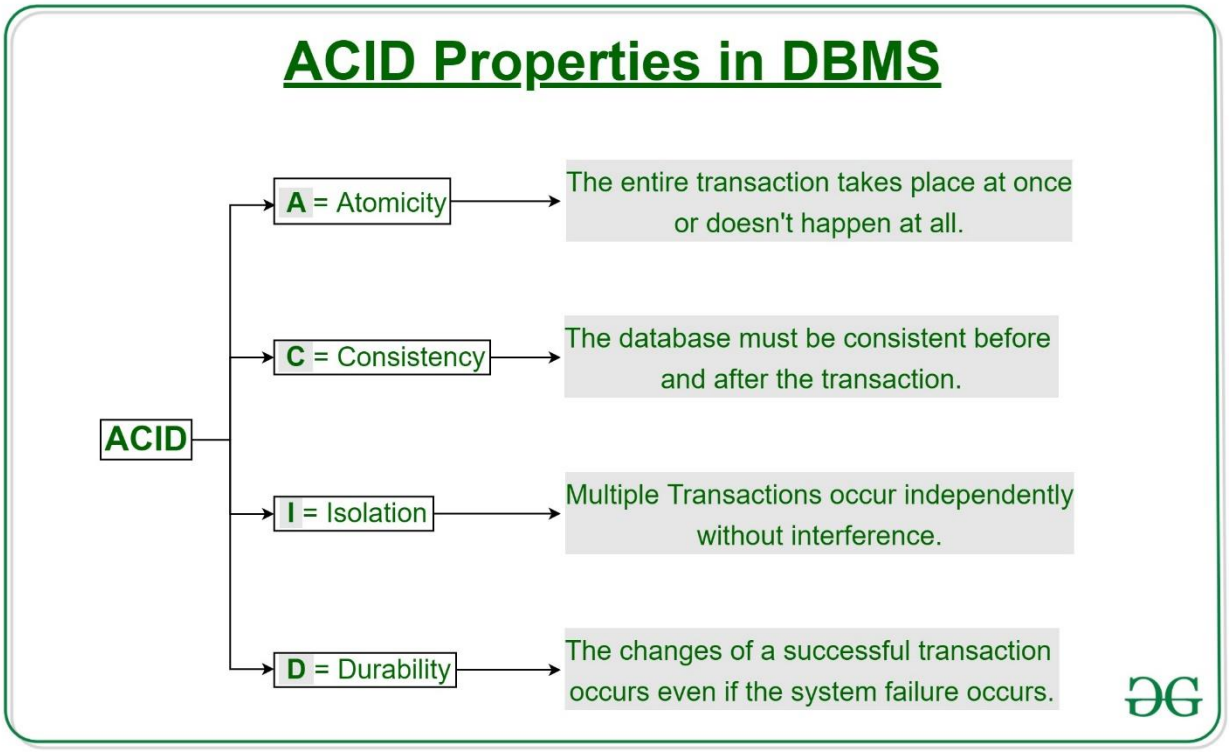
Following are the some commands that come under DCL:

4. Transaction Control Language

Transactions are atomic i.e. either every statement succeeds or none of statement succeeds. There are number of Transaction Control statements available that allow us to control this behavior. These statements ensure data consistency. 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.

ACID Properties in DBMS



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.

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.

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.

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.