Software requirements

Download Xampp from https://www.apachefriends.org/download.html

Download latest version of php 8.2.12 + (Feb 2024)

How to open mysql server

Go to xampp control panel and start apache server and mysql server

Then go to browser and enter URL http://localhost/phpmyadmin/

From home page of mysql server select databases section and create new database.

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

The columns in the "Customers" table above are: CustomerID, CustomerName, ContactName, Address, City, PostalCode and Country.

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:

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

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

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

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`

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.

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

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 roll, fname, lname, city, email 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 * Example

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

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 DISTINCT fname FROM students;

SELECT Example Without DISTINCT

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

SELECT city from students

Select count(distinct city) from students

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.!

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 city = 'rajkot'

Select * from students where roll > 5

Select * from students where not roll > 5;

Select * from students where roll != 5;
```

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 roll = 1

SELECT * FROM `students` WHERE roll = 1 and fname = 'udit';

SELECT * FROM `students` WHERE roll = 1 or roll = 2;

SELECT * FROM `students` WHERE roll = 1 or city = 'rajkot';

SELECT * FROM `students` WHERE roll = 1 and  city = 'rajkot' or city = 'surat';

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

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 ORDER by fname

SELECT * from students ORDER by fname desc;

SELECT * FROM students ORDER by fname, city

SELECT * FROM students ORDER by fname, city desc

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, email, gender, dateofbirth, phone) values ("Rachit", "Chauhan", "Baroda", "rachit@gmail.com", "male", "2002-02-02", "9900009900")

INSERT into students values ("Rachit", "Chauhan", "Baroda", "rachit@gmail.com", "male", "2002-02-02", "9900009900")

Error : `Column count doesn't match value count at row 1`

Insert Data Only in Specified Columns

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

INSERT into students (fname, lname, city, email, gender) values ("Rachit", "Chauhan", "Baroda", "rachit@gmail.com", "male");

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 phone = '';

SELECT * FROM `students` WHERE dateofbirth = 'NULL'; // no data

SELECT * FROM `students` WHERE dateofbirth is NULL;

SELECT * FROM `students` WHERE dateofbirth is not NULL;
```

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;
```

The IS NULL Operator

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

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

The IS NOT NULL Operator

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

The MySQL UPDATE Statement

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

UPDATE Syntax

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

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

UPDATE students set city = 'Gandhinagar' //always use where while update data

update students set fname = 'Yagnik', lname = 'Yadav', city = 'Bhavanagar' WHERE roll = 3

update students set fname = 'Yagnik', lname = 'Yadav', city = 'Bhavanagar' WHERE roll > 3 and roll < 6;

UPDATE students set city = 'Rajkot' WHERE roll >= 5

UPDATE Multiple Records

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

Update Warning!

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

UPDATE students set fname = 'tushar', lname = 'kadam' WHERE roll = 4

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
```

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

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

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

ADD a WHERE CLAUSE

```
SELECT * FROM `students` where city = ‘rajkot’ Limit 5
```

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(roll) FROM students;
```

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

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

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

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;
```

```
SELECT COUNT(fees) FROM students
```

```
SELECT COUNT(fees) FROM students WHERE fees > 10000;
```

```
SELECT sum(fees) FROM students WHERE fees > 10000;
```

```
SELECT count(fees), sum(fees) FROM students WHERE fees > 10000;
```

```
SELECT count(fees), sum(fees), avg(fees) FROM students
```

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;
```

The MySQL DELETE Statement

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

DELETE Syntax

DELETE FROM *table_name* WHERE *condition*;

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

SQL DELETE Example

DELETE FROM students WHERE roll = 5

DELETE FROM students WHERE city = 'rajkot'

Delete All Records

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

DELETE FROM students

INSERT into students (fname, lname, city, email, gender, dateofbirth, phone, fees) values ("Rachit", "Chauhan", "Baroda", "rachit@gmail.com", "male", "2002-02-02", "9900009900", 15000);

- Add multiple rows and check for roll number

It is possible to delete all the data from table and reset complete structure of the table with truncate clause

TRUNCATE TABLE students

INSERT into students (fname, lname, city, email, gender, dateofbirth, phone, fees) values ("Rachit", "Chauhan", "Baroda", "rachit@gmail.com", "male", "2002-02-02", "9900009900", 15000);

- Add multiple rows and check for roll number // started from 1

Insert some data in table

INSERT INTO `students` (`roll`, `fname`, `lname`, `city`, `email`, `gender`, `dateofbirth`, `phone`, `fees`, `admissiondate`) VALUES (NULL, 'udit', 'ghetiya', 'Rajula', 'udit@gmail.com', 'male', '2004-02-11', '998899889900', '18000', current_timestamp()),(NULL, 'Gaurang', 'Pandya', 'Baroda', 'gaurang@gmail.com', 'male', '2004-02-11', '998899889900', '15000', current_timestamp()),(NULL, 'yograjsinh', 'Rana', 'Junagadh', 'yorajsinh@gmail.com', 'male', '2004-02-11', '998899889900', '22000', current_timestamp()),(NULL, 'yadav', 'yagnik', 'Bhavanagar', 'yagnik@gmail.com', 'male', '2004-02-11', '998899889900', '19000', current_timestamp()),(NULL, 'Rachit', 'Chauhan', 'Jamnagar', 'rachit@gmail.com', 'male', '2004-02-11', '998899889900', '12000', current_timestamp()),(NULL, 'adarsh', 'chavda', 'amreli', 'adarsh@gmail.com', 'male', '2004-02-11', '998899889900', '12000', current_timestamp()),(NULL, 'rohan', 'dasadiya', 'Bhuj', 'rohan@gmail.com', 'male', '2004-02-11', '998899889900', '13000', current_timestamp()),(NULL, 'Tushar', 'Kadam', 'Morbi', 'Tushar@gmail.com', 'male', '2004-02-11', '998899889900', '19000', current_timestamp()),(NULL, 'Yash', 'Vaghela', 'Vadodara', 'yash@gmail.com', 'male', '2004-02-11', '998899889900', '22000', current_timestamp()),(NULL, 'Jayrajsinh', 'Parmar', 'Limdi', 'jayrajsinh@gmail.com', 'male', '2004-02-11', '998899889900', '24000', current_timestamp());

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!

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 '%t';

SELECT * FROM students WHERE fname like '%i%';

SELECT * FROM students WHERE fname like '_a%';

SELECT * FROM students WHERE fname like 'a_%';

SELECT * FROM students WHERE fname like 'r__%';

SELECT * FROM students WHERE fname like 'r%t';

SELECT * FROM students WHERE fname not like 'r%t';
```

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 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 = 'baroda' or city = 'morbi';
SELECT * FROM students WHERE city in ('rajkot', 'baroda', 'amreli', 'bhuj', 'junagadh')

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

SELECT * FROM students WHERE roll in (SELECT roll from students WHERE not city = 'rajkot');

Outer query (sub query),

First Execute sub query and get result from database, place this result data between in () and execute outer query,

SELECT * from students WHERE **roll** in (SELECT **roll** FROM marks WHERE result = 'pass');

SELECT * from students WHERE roll not in (SELECT roll FROM marks WHERE result = 'pass');

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 fees BETWEEN 5000 and 15000

SELECT * FROM students WHERE fees not BETWEEN 5000 and 15000;

SELECT * FROM students WHERE fname BETWEEN 'adarsh' and 'tushar'

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

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 FirstName FROM students

SELECT fname as "First Name" FROM students;

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

SELECT concat_ws(" - ", roll, fname, lname, city) as "Student Data" FROM students

---

Without join and alias

SELECT students.roll, students.fname, students.lname, students.city, students.email, students.gender, students.dateofbirth, students.phone, students.fees, marks.total, marks.result FROM students, marks WHERE students.fname = 'bhavdeep' and students.roll = marks.roll;

With alias

SELECT s.roll, s.fname, s.lname, s.city, s.email, s.gender, s.dateofbirth, s.phone, s.fees, m.total, m.result FROM students as s, marks as m WHERE s.fname = 'bhavdeep' and s.roll = m.roll;

SELECT CURRENT_TIMESTAMP

SELECT CURRENT_TIMESTAMP as "Today is :";

Aliases can be useful when:

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

MySQL Joins

MySQL Joining Tables

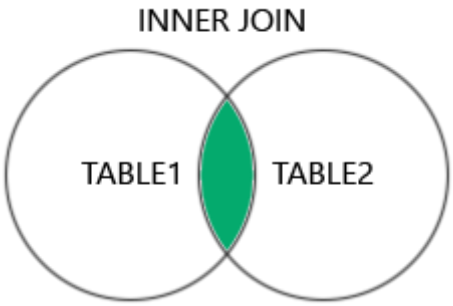
A `JOIN` clause is used to combine rows from two or more tables, based on a related column between them.

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

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

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.email, s.gender, s.dateofbirth, s.phone, s.course, s.course, s.admissiondate, m.total, m.result, a.absents, a.presents from students s inner JOIN marks m on s.roll = m.roll INNER join attendance a on s.roll = a.roll;

SELECT s.*, m.total, m.result, a.absents, a.presents from students s inner JOIN marks m on s.roll = m.roll INNER join attendance a on s.roll = a.roll;

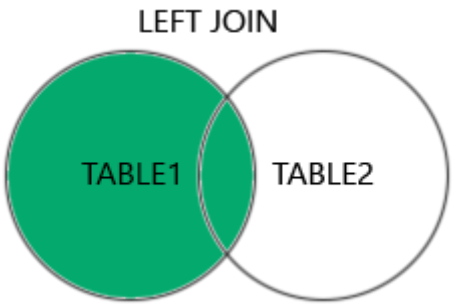
SELECT s.*, m.*, a.absents, a.presents from students s inner JOIN marks m on s.roll = m.roll INNER join attendance a on s.roll = a.roll;

/// never do this

SELECT s.*, m.*, a.* from students s inner JOIN marks m on s.roll = m.roll INNER join attendance a on s.roll = a.roll;

MySQL LEFT JOIN Keyword

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



LEFT JOIN Syntax

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

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

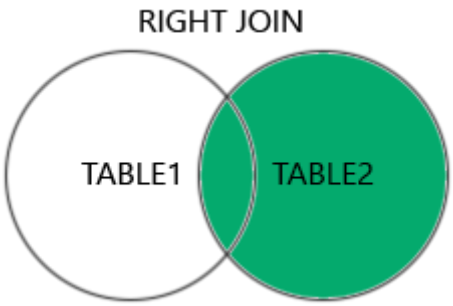
select s.roll, s.fname, s.lname, s.city, s.email, s.gender, s.dateofbirth, s.phone, s.fees, s.course, s.admissiondate, m.total, m.result from students s left join marks m on s.roll = m.roll;

grab all the data from left table (students) if there is no data in right table (marks) then sql show NULL for empty values

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

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

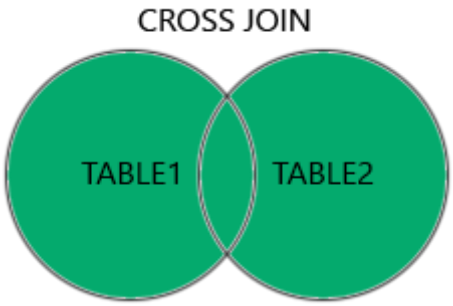
SELECT *column_name(s)* FROM *table1* RIGHT JOIN *table2* ON *table1.column_name* = *table2.column_name*;

select s.roll, s.fname, s.lname, s.city, s.email, s.gender, s.dateofbirth, s.phone, s.fees, s.course, s.admissiondate, m.total, m.result from students s right join marks m on s.roll = m.roll;

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

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 * from students CROSS join marks;
```

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 * from students CROSS join marks WHERE students.roll = marks.roll;
```

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*;

SELECT t1.fname, t1.lname from students t1, students t2 WHERE t1.roll != t2.roll and t1.city = t2.city;

SELECT t1.roll, t1.fname, t1.lname, t1.city from students t1, students t2 WHERE t1.roll != t2.roll and t1.city = t2.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

UNION Syntax

SELECT *column_name(s)* FROM *table1* UNION SELECT *column_name(s)* FROM *table2*;

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

SELECT * from students

UNION ALL

SELECT * FROM students1;

SELECT * from students

UNION ALL

SELECT * FROM students1 ORDER by city;

SELECT * from students where city = 'rajkot'

UNION ALL

SELECT * FROM students1 where city = 'rajkot';

MySQL GROUP BY Statement

The MySQL GROUP BY Statement

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

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

```
SELECT city, count(city) FROM students GROUP by (city)
```

GROUP BY With JOIN Example

```
SELECT students.roll, COUNT(marks.marksid), sum(marks.total) from students INNER join marks on students.roll = marks.roll GROUP by (roll);
```

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(city) from students GROUP by (city) having count(city) = 1;
```

```
SELECT city, count(city) from students GROUP by (city) having count(city) > 1;
```

```
SELECT students.roll, COUNT(marks.marksid), sum(marks.total) from students INNER join marks on students.roll = marks.roll GROUP by (roll) HAVING sum(marks.total) >= 400;
```

The MySQL EXISTS Operator

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

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

EXISTS Syntax

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

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

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

```
SELECT students.roll, students.fname, students.lname from students where EXISTS (SELECT roll FROM marks WHERE students.roll = marks.roll and marks.result = 'pass')
```

```
SELECT students.roll, students.fname, students.lname 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.

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')
```

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.

```
SELECT all fname FROM students WHERE true
```

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 (fname, lname, city, email, gender, dateofbirth, phone, fees, course) SELECT fname, lname, city, email, gender, dateofbirth, phone, fees, course from students

INSERT into students1 (fname, lname, city, email, gender, dateofbirth, phone, fees, course) SELECT fname, lname, city, email, gender, dateofbirth, phone, fees, course from students where city = 'rajkot';

MySQL INSERT INTO SELECT Examples

The following SQL statement copies "Suppliers" into "Customers" (the columns that are not filled with data, will contain NULL):

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, gender, city, email FROM students

SELECT roll, fname, lname, gender, city,

case

        WHEN city = 'Rajkot' THEN "Home Town"

        WHEN city = 'Surat' THEN "Too much Far from Home Town"

        WHEN city = 'morbi' THEN "Near by Home Town"

        WHEN city = 'limdi' THEN "Far from Home Town"

        WHEN city = 'Bhavanagar' THEN "Far from Home Town"

        WHEN city = 'Amreli' THEN "Near by Home Town"

        WHEN city = 'Jamnagar' THEN "Near by Home Town"

        WHEN city = 'Bhuj' THEN "Far from Home Town"

        WHEN city = 'Rajula' THEN "Far from Home Town"

        WHEN city = 'vadodara' or city = 'baroda' THEN "Far from Home Town"

        else "Unkonown istance"

end as "Distance From Rajkot", email FROM students;
```

MySQL NULL Functions

MySQL IFNULL() and COALESCE() Functions

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

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, sum(ifnull(absents, 0) + ifnull(presents, 0)) as "Total Working Days" from attendance GROUP by (roll);

MySQL COALESCE() Function

SELECT roll, sum(coalesce(absents, 0) + coalesce(presents, 0)) as "Total Working Days" from attendance GROUP by (roll);

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).

```
-- find null values and replace with 0

SELECT roll, sum(coalesce(absents, 0) + coalesce(presents, 0)) as "Total Working Days" from attendance GROUP by (roll);
```

Multi-line Comments

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

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

The following example uses

a multi-line comment as an explanation:

Example

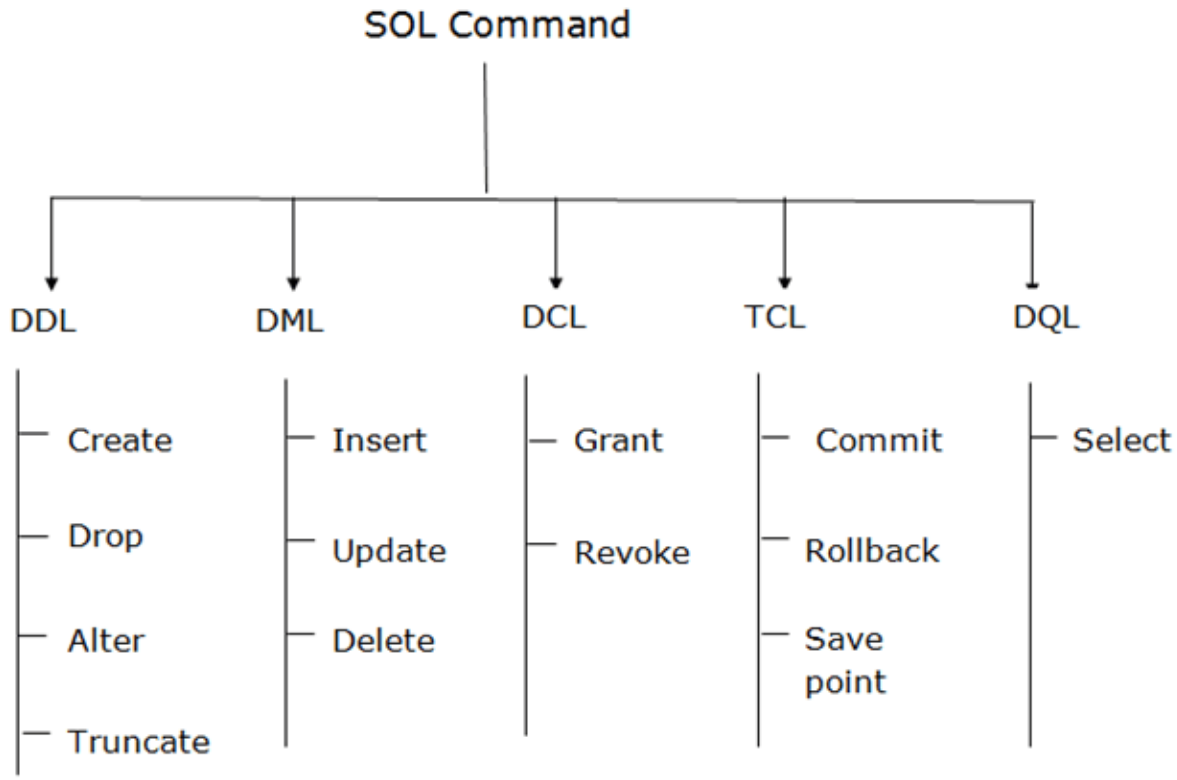
```
/*Select all the columns
of all the records
in the Customers table:*/
SELECT * FROM Customers;


/* find null values and replace with 0 (this is multiline comment in MySql)*/

SELECT roll, sum(coalesce(absents, 0) + coalesce(presents, 0)) as "Total Working Days" from attendance GROUP by (roll);
```

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

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.

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

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

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*;

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

show DATABASES;

create DATABASE demo729;

show DATABASES

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!

```
show DATABASES

DROP DATABASE demo729

show DATABASES
```

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

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 persons (personid int AUTO_INCREMENT PRIMARY key, fname varchar(20), lname varchar(20), email varchar(128), phone varchar(15), gender varchar(6), `dateof birth` date)
```

Create Table Using Another Table

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

The new table gets the same column definitions. All columns or specific columns can be selected.

If you create a new table using an existing table, the new table will be filled with the existing values from the old table.

Syntax

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

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

```
CREATE TABLE persons_backup1 as SELECT personid, fname, lname 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 city varchar(30)

ALTER TABLE persons add COLUMN state varchar(30), add COLUMN country varchar(20)

ALTER TABLE persons add COLUMN zipcode int after city

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 zipcode

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(10)

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 persons_backup1
```

MySQL TRUNCATE TABLE

The `TRUNCATE TABLE` statement is used to delete (reset the table structure to new – auto Increment) the data inside a table, but **not** the table itself.

Syntax

```
TRUNCATE TABLE table_name;
```

```
TRUNCATE TABLE students1
```

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 **persons1** (personid int AUTO_INCREMENT PRIMARY key, fname varchar(20) not null, lname varchar(20) not null, city varchar(20) not null)

INSERT INTO persons1 (fname, lname, city) values ('Bhavdeep', 'Sorathiya', 'Rajkot')

INSERT INTO persons1 (fname, lname, city) values ('Bhavdeep', 'Sorathiya', NULL);

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

NOT NULL on ALTER TABLE

ALTER table persons1 add COLUMN age int

ALTER TABLE persons1 MODIFY COLUMN age int not null

To create a `NOT NULL` constraint on the "Age" column when the "Persons" table is already created, use the following SQL:

INSERT INTO persons1 (fname, lname, city, age) values ('Bhavdeep', 'Sorathiya', 'baroda', 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

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

```
create table persons (personid int not null, fname varchar(20) not null, lname varchar(20) not null, city varchar(20) not null, UNIQUE(personid))

INSERT into persons (personid, fname, lname, city) values (1, 'Udit', 'Ghetiya', 'Rajkot')

INSERT into persons (personid, fname, lname, city) values (1, 'Udit', 'patel', 'Rajkot');

#1062 - Duplicate entry '1' for key 'personid'

INSERT into persons (personid, fname, lname, city) values (2, 'Udit', 'patel', 'Rajkot');
```

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

```
drop table persons;

create table persons (personid int not null, fname varchar(20) not null, lname varchar(20) not null, city varchar(20) not null, CONSTRAINT unqid UNIQUE (personid))

INSERT into persons (personid, fname, lname, city) values (1, 'Udit', 'Ghetiya', 'Rajkot')

INSERT into persons (personid, fname, lname, city) values (1, 'Udit', 'Ghetiya', 'Rajkot')

#1062 - Duplicate entry '1' for key 'unqid'

Following query drop named constrained unqid from persons table.

alter TABLE persons drop CONSTRAINT unqid

INSERT into persons (personid, fname, lname, city) values (1, 'Udit', 'Ghetiya', 'Rajkot')

INSERT into persons (personid, fname, lname, city) values (1, 'Udit', 'Ghetiya', 'Rajkot')

INSERT into persons (personid, fname, lname, city) values (1, 'Udit', 'Ghetiya', 'Rajkot')
```

UNIQUE Constraint on ALTER TABLE

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

Truncate table persons

alter TABLE persons add UNIQUE(personid)

INSERT into persons (personid, fname, lname, city) values (**1**, 'Udit', 'Ghetiya', 'Rajkot')

INSERT into persons (personid, fname, lname, city) values (**1**, 'Udit', 'Ghetiya', 'Rajkot')

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

drop table persons

create table persons (personid int not null, fname varchar(20) not null, lname varchar(20) not null, city varchar(20) not null)

ALTER TABLE persons add CONSTRAINT unqid UNIQUE (personid)

INSERT into persons (personid, fname, lname, city) values (**1**, 'Udit', 'Ghetiya', 'Rajkot')

INSERT into persons (personid, fname, lname, city) values (**1**, 'Udit', 'Ghetiya', 'Rajkot')

INSERT into persons (personid, fname, lname, city) values (null, 'Udit', 'Ghetiya', 'Rajkot')

INSERT into persons (personid, fname, lname, city) values (null, 'Udit', 'Ghetiya', 'Rajkot')

- Major drawback of unique constraint is it allows NULL values in column.

DROP a UNIQUE Constraint

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

ALTER table persons drop CONSTRAINT unqid

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

create table listoftown (townid int AUTO_INCREMENT PRIMARY key, townname varchar(20) UNIQUE, cityname varchar(20), district varchar(20), state varchar(20))

INSERT INTO listoftown (townname, cityname, district, state) values ('Navagam', 'Rajkot', 'Rajkot', 'Gujarat')

INSERT INTO listoftown (townname, cityname, district, state) values ('Navagam', 'Bhuj', 'Kutchh', 'Gujarat');

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

drop table listoftown

create table listoftown (townid int AUTO_INCREMENT PRIMARY key, townname varchar(20), cityname varchar(20), district varchar(20), state varchar(20), CONSTRAINT unqtown UNIQUE(townname, cityname, district))

INSERT INTO listoftown (townname, cityname, district, state) values ('Navagam', 'Rajkot', 'Rajkot', 'Gujarat')

INSERT INTO listoftown (townname, cityname, district, state) values ('Navagam', 'Bhuj', 'Kutchh', '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).

PRIMARY KEY on CREATE TABLE

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

```
CREATE TABLE person (id int, fname varchar(20) not null, lname varchar(20) not null, city varchar(20) not null, PRIMARY key(id))
```

```
INSERT into person (id, fname, lname, city) VALUES(1, 'bhavdeep', 'sorathiya', 'rajkot')
```

```
INSERT into person (id, fname, lname, city) VALUES(1, 'bhavdeep', 'sorathiya', 'rajkot')
```

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

```
drop TABLE person
```

CREATE TABLE person (id int, fname varchar(20) not null, lname varchar(20) not null, city varchar(20) not null, CONSTRAINT prikey_id PRIMARY key (id))

INSERT into person (id, fname, lname, city) VALUES(1, 'bhavdeep', 'sorathiya', 'rajkot')

INSERT into person (id, fname, lname, city) VALUES(1, 'bhavdeep', 'sorathiya', 'rajkot')

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

ALTER TABLE person drop PRIMARY KEY

INSERT into person (id, fname, lname, city) VALUES(1, 'bhavdeep', 'sorathiya', 'rajkot')

INSERT into person (id, fname, lname, city) VALUES(1, 'bhavdeep', 'sorathiya', 'rajkot')

INSERT into person (id, fname, lname, city) VALUES(1, 'bhavdeep', 'sorathiya', 'rajkot')

Drop table person

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);
```

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

CREATE TABLE person (id int, fname varchar(20) not null, lname varchar(20) not null, city varchar(20) not null)

ALTER TABLE person add PRIMARY key (id)

INSERT into person (id, fname, lname, city) VALUES(1, 'bhavdeep', 'sorathiya', 'rajkot')

INSERT into person (id, fname, lname, city) VALUES(1, 'bhavdeep', 'sorathiya', 'rajkot')

ALTER TABLE person drop PRIMARY key

ALTER TABLE person add PRIMARY key (id, fname)

INSERT into person (id, fname, lname, city) VALUES(**1, 'bhavdeep'**, 'sorathiya', 'rajkot');

INSERT into person (id, fname, lname, city) VALUES(**1, 'bhavdip'**, 'sorathiya', 'rajkot')

INSERT into person (id, fname, lname, city) VALUES(**2, 'bhavdip',** 'sorathiya', 'rajkot');

INSERT into person (id, fname, lname, city) VALUES(**2, 'bhavdip'**, 'sorathiya', 'rajkot');

```
#1062 - Duplicate entry '2-bhavdip' for key 'PRIMARY'
```

DROP a PRIMARY KEY Constraint

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

ALTER TABLE Person DROP PRIMARY KEY;

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

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

create TABLE fees (feesid int AUTO_INCREMENT PRIMARY key, roll int, paymentdate date, amount int, paymentmode varchar(10), paymenttimestamp timestamp DEFAULT CURRENT_TIMESTAMP)

INSERT INTO `fees` (`feesid`, `roll`, `paymentdate`, `amount`, `paymentmode`, `paymenttimestamp`) VALUES (NULL, '123', '2024-02-17', '5000', 'cash', current_timestamp());

Above data is invalid because we don't have student with roll 123

drop TABLE fees

create TABLE fees (feesid int AUTO_INCREMENT PRIMARY key, roll int, paymentdate date, amount int, paymentmode varchar(10), paymenttimestamp timestamp DEFAULT CURRENT_TIMESTAMP, FOREIGN key (roll) REFERENCES students(roll))

INSERT INTO `fees` (`feesid`, `roll`, `paymentdate`, `amount`, `paymentmode`, `paymenttimestamp`) VALUES (NULL, '123', '2024-02-17', '5000', 'cash', current_timestamp());

```
#1452 - Cannot add or update a child row: a foreign key constraint fails
(`729_2324`.`fees`, CONSTRAINT `fees_ibfk_1` FOREIGN KEY (`roll`)
REFERENCES `students` (`roll`))
```

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

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

drop TABLE fees

create TABLE fees (feesid int AUTO_INCREMENT PRIMARY key, roll int, paymentdate date, amount int, paymentmode varchar(10), paymenttimestamp timestamp DEFAULT CURRENT_TIMESTAMP, CONSTRAINT fk_roll FOREIGN key (roll) REFERENCES students(roll))

INSERT INTO `fees` (`feesid`, `roll`, `paymentdate`, `amount`, `paymentmode`, `paymenttimestamp`) VALUES (NULL, '123', '2024-02-17', '5000', 'cash', current_timestamp());

```
#1452 - Cannot add or update a child row: a foreign key constraint fails
(`729_2324`.`fees`, CONSTRAINT `fk_roll` FOREIGN KEY (`roll`) REFERENCES
`students` (`roll`))
```

DROP a FOREIGN KEY Constraint

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

ALTER TABLE fees DROP CONSTRAINT fk_roll

INSERT INTO `fees` (`feesid`, `roll`, `paymentdate`, `amount`, `paymentmode`, `paymenttimestamp`) VALUES (NULL, '123', '2024-02-17', '5000', 'cash', current_timestamp());

INSERT INTO `fees` (`feesid`, `roll`, `paymentdate`, `amount`, `paymentmode`, `paymenttimestamp`) VALUES (NULL, '1234', '2024-02-17', '5000', 'cash', current_timestamp());

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 fees add CONSTRAINT fk_roll FOREIGN key(roll) REFERENCES students(roll)

```
#1452 - Cannot add or update a child row: a foreign key constraint fails
(`729_2324`.`#sql-2668_353`, CONSTRAINT `fk_roll` FOREIGN KEY (`roll`)
REFERENCES `students` (`roll`))
```

TRUNCATE TABLE fees

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

INSERT INTO `fees` (`feesid`, `roll`, `paymentdate`, `amount`, `paymentmode`, `paymenttimestamp`) VALUES (NULL, '1', '2024-02-17', '4000', 'cash', current_timestamp()), (NULL, '2', '2024-02-17', '3500', 'case', current_timestamp());

DELETE from students WHERE roll = 1

```
#1451 - Cannot delete or update a parent row: a foreign key constraint
fails (`729_2324`.`fees`, CONSTRAINT `fk_roll` FOREIGN KEY (`roll`)
REFERENCES `students` (`roll`))
```