# Sql queries 1

08 January 2025 11:39

```
-- 1. Select the database
USE zoo;

-- 2. Show tables in the 'zoo' database
SHOW TABLES;

-- 3. Describe the 'pet' table
DESC pet;

-- 4. Modify the 'name' column in the 'pet' table
ALTER TABLE pet MODIFY COLUMN name VARCHAR(30);

-- 5. Drop the 'kis' column from the 'pet' table
ALTER TABLE pet DROP COLUMN kis;

-- 6. Describe the 'habitat' table
DESC habitat;

-- 7. Insert values into the 'habitat' table
INSERT INTO habitat (name) VALUES ('River'), ('forest');

-- 8. Select all records from the 'habitat' table
SELECT * FROM habitat;

-- 9. Select a specific record from the 'habitat' table
SELECT * FROM habitat WHERE id = 1;

-- 10. Select from 'habitat' where 'name' matches 'river'
SELECT * FROM habitat WHERE name = "river";

-- 11. Insert a new value into 'habitat'
INSERT INTO habitat (name) VALUES ('savanna');

-- 12. Modify 'id' column in 'habitat' table to be AUTO_INCREMENT
ALTER TABLE habitat MODIFY COLUMN id INT PRIMARY KEY
AUTO_INCREMENT;

-- 13. Insert new values into 'habitat'
INSERT INTO habitat (name) VALUES ('seet'), ('kalli'), ('shakti');
```

```
-- 14. Update the 'habitat' table
UPDATE habitat SET name = 'ravina' WHERE id = 3;

-- 15. Update 'habitat' based on multiple conditions
UPDATE habitat SET name = 'mohit' WHERE id = 2 OR id = 4;

-- 16. Add new columns 'region' and 'location' to the 'habitat' table
ALTER TABLE habitat ADD region VARCHAR(30), ADD location VARCHAR(20);

-- 17. Update 'habitat' table with values for 'region' and 'location'
UPDATE habitat SET region = 'barmer', location = 'rajasthan' WHERE id = 2;

-- 18. Delete a specific record from the 'habitat' table
DELETE FROM habitat WHERE id = 3;

-- 19. Delete all records from the 'habitat' table
DELETE FROM habitat;

-- 20. Select all records from 'habitat' after updates
SELECT * FROM habitat;

-- 21. Create an alias for the 'habitat' table to simplify queries
SELECT h.id, h.name, h.region, h.location
FROM habitat AS h;

-- 22. Use the alias in a query to filter records
SELECT h.id, h.name, h.region
FROM habitat AS h
WHERE h.region = 'barmer';

-- 23. Select records using alias and specific conditions
SELECT h.id, h.name
FROM habitat AS h
WHERE h.location = 'rajasthan';

-- 24. Select and count the number of habitats in the 'habitat' table
SELECT COUNT(*) AS total_habitats FROM habitat;
```

## 1. Numeric

- **Exact**: INT, BIGINT, SMALLINT, TINYINT, DECIMAL(p, s), NUMERIC(p, s)
- **Approximate**: FLOAT, REAL, DOUBLE

## 2. Character/String

- CHAR(n): Fixed-length
- VARCHAR(n): Variable-length
- TEXT: Large text

### 3. Date/Time

- DATE: YYYY-MM-DD
- TIME: HH:MM:SS
- DATETIME: Combines DATE and TIME
- TIMESTAMP: With timezone info
- YEAR: Year only

### 4. Binary

- BLOB: Binary Large Object
- VARBINARY(n), BINARY(n): Fixed/variable-length binary data

### 5. Boolean

- BOOLEAN: True/False (often stored as 0/1)

### 6. Other

- JSON: JSON data
- ENUM: Predefined set of values
- SET: Multiple predefined values

# Sql cheatsheet

09 January 2025 15:59

182839750
964638

# MySQL Cheat Sheet

LearnSQL.com

MySQL is a popular open-source relational database management system known for its ease of use and scalability. Sometimes, you will need a little help while working on a project. That's why we created this MySQL Cheat Sheet.

Instructions for installing MySQL are available at: https://dev.mysql.com

## CONNECTING TO A MYSQL SERVER

Connect to a MySQL server with a username and a password using the mysql command-line client.
MySQL will prompt for the password:

```
mysql -u [username] -p
```

To connect to a specific database on a MySQL server using a username and a password:

```
mysql -u [username] -p [database]
```

To export data using the `mysqldump` tool:

```
mysqldump -u [username] -p \
  [database] > data_backup.sql
```

To exit the client:

```
quit or exit
```

For a full list of commands:

```
help
```

## CREATING AND DISPLAYING DATABASES

To create a database:

```
CREATE DATABASE zoo;
```

To list all the databases on the server:

```
SHOW DATABASES;
```

To use a specified database:

```
USE zoo;
```

To delete a specified database:

```
DROP DATABASE zoo;
```

To list all tables in the database:

```
SHOW TABLES;
```

To get information about a specified table:

```
DESCRIBE animal;
```

It outputs column names, data types, default values, and more about the table.

## CREATING TABLES

To create a table:

```
CREATE TABLE habitat (
  id INT,
  name VARCHAR(64)
);
```

Use AUTO_INCREMENT to increment the ID automatically with each new record. An AUTO_INCREMENT column must be defined as a primary or unique key:

```
CREATE TABLE habitat (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(64)
);
```

To create a table with a foreign key:

```
CREATE TABLE animal (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(64),
  species VARCHAR(64),
  age INT,
  habitat_id INT,
  FOREIGN KEY (habitat_id)
    REFERENCES habitat(id)
);
```

## MODIFYING TABLES

Use the `ALTER TABLE` statement to modify the table structure.

To change a table name:

```
ALTER TABLE animal RENAME pet;
```

To add a column to the table:

```
ALTER TABLE animal
ADD COLUMN name VARCHAR(64);
```

To change a column name:

```
ALTER TABLE animal
RENAME COLUMN id TO identifier;
```

To change a column data type:

```
ALTER TABLE animal
MODIFY COLUMN name VARCHAR(128);
```

To delete a column:

```
ALTER TABLE animal
DROP COLUMN name;
```

To delete a table:

```
DROP TABLE animal;
```

## QUERYING DATA

To select data from a table, use the `SELECT` command.
An example of a single-table query:

```
SELECT species, AVG(age) AS average_age
FROM animal
WHERE id != 3
GROUP BY species
HAVING AVG(age) > 3
ORDER BY AVG(age) DESC;
```

An example of a multiple-table query:

```
SELECT city.name, country.name
FROM city
[INNER | LEFT | RIGHT] JOIN country
  ON city.country_id = country.id;
```

Use `+`, `-`, `*`, `/` to do some basic math.
To get the number of seconds in a week:

```
SELECT 60 * 60 * 24 * 7; -- result: 604800
```

## AGGREGATION AND GROUPING

- **`AVG(expr)`** – average value of `expr` for the group.
- **`COUNT(expr)`** – count of `expr` values within the group.
- **`MAX(expr)`** – maximum value of `expr` values within the group.
- **`MIN(expr)`** – minimum value of `expr` values within the group.
- **`SUM(expr)`** – sum of `expr` values within the group.

To count the rows in the table:

```
SELECT COUNT(*)
FROM animal;
```

To count the non-NULL values in a column:

```
SELECT COUNT(name)
FROM animal;
```

To count unique values in a column:

```
SELECT COUNT(DISTINCT name)
FROM animal;
```

## GROUP BY

To count the animals by species:

```
SELECT species, COUNT(id)
FROM animal
GROUP BY species;
```

To get the average, minimum, and maximum ages by habitat:

```
SELECT habitat_id, AVG(age),
       MIN(age), MAX(age)
FROM animal
GROUP BY habitat_id;
```

## INSERTING DATA

To insert data into a table, use the `INSERT` command:

```
INSERT INTO habitat VALUES
(1, 'River'),
(2, 'Forest');
```

You may specify the columns in which the data is added. The remaining columns are filled with default values or `NULL`s.

```
INSERT INTO habitat (name) VALUES
('Savanna');
```

## UPDATING DATA

To update the data in a table, use the `UPDATE` command:

```
UPDATE animal
SET
  species = 'Duck',
  name = 'Quack'
WHERE id = 2;
```

## DELETING DATA

To delete data from a table, use the `DELETE` command:

```
DELETE FROM animal
WHERE id = 1;
```

This deletes all rows satisfying the `WHERE` condition.
To delete all data from a table, use the `TRUNCATE TABLE` statement:

```
TRUNCATE TABLE animal;
```

## CASTING

From time to time, you need to change the type of a value. Use the `CAST()` function to do this.
In MySQL, you can cast to these data types:

```
CHAR      NCHAR  BINARY DATE   DATETIME
DECIMAL   DOUBLE FLOAT  REAL   SIGNED
UNSIGNED TIME    YEAR   JSON   spatial_type
```

To get a number as a signed integer:

```
SELECT CAST(1234.567 AS signed);
-- result: 1235
```

To change a column type to double:

```
SELECT CAST(column AS double);
```

Try out the interactive **SQL from A to Z in MySQL** course at **LearnSQL.com**, and check out our other **SQL courses**.

# MySQL Cheat Sheet

LearnSQL.com

## TEXT FUNCTIONS

### FILTERING THE OUTPUT

To fetch the city names that are not Berlin:

```
SELECT name
FROM city
WHERE name != 'Berlin';
```

### TEXT OPERATORS

To fetch the city names that start with a 'P' or end with an 's':

```
SELECT name
FROM city
WHERE name LIKE 'P%' OR name LIKE '%s';
```

To fetch the city names that start with any letter followed by 'ublin' (like Dublin in Ireland or Lublin in Poland):

```
SELECT name
FROM city
WHERE name LIKE '_ublin';
```

### CONCATENATION

Use the CONCAT() function to concatenate two strings:

```
SELECT CONCAT('Hi ', 'there!');
-- result: Hi there!
```

If any of the string is NULL, the result is NULL:

```
SELECT CONCAT(Great ', 'day', NULL);
-- result: NULL
```

MySQL allows specifying a separating character (separator) using the CONCAT_WS() function. The separator is placed between the concatenated values:

```
SELECT CONCAT_WS(' ', 1, 'Olivier',
'Norris'); -- result: 1 Olivier Norris
```

### OTHER USEFUL TEXT FUNCTIONS

To get the count of characters in a string:

```
SELECT LENGTH('LearnSQL.com');
-- result: 12
```

To convert all letters to lowercase:

```
SELECT LOWER('LEARNSQL.COM');
-- result: learnsql.com
```

To convert all letters to uppercase:

```
SELECT UPPER('LearnSQL.com');
-- result: LEARNSQL.COM
```

To get just a part of a string:

```
SELECT SUBSTRING('LearnSQL.com', 9);
-- result: .com
SELECT SUBSTRING('LearnSQL.com', 1, 5);
-- result: Learn
```

To replace a part of a string:

```
SELECT REPLACE('LearnSQL.com', 'SQL',
'Python');
-- result: LearnPython.com
```

## NUMERIC FUNCTIONS

To get the remainder of a division:

```
SELECT MOD(13, 2); -- result: 1
```

To round a number to its nearest integer:

```
SELECT ROUND(1234.56789); -- result: 1235
```

To round a number to three decimal places:

```
SELECT ROUND(1234.56789, 3);
-- result: 1234.568
```

To round a number up:

```
SELECT CEIL(13.1); -- result: 14
SELECT CEIL(-13.9); -- result: -13
```

The CEIL(x) function returns the smallest integer not less than x. To round the number down:

```
SELECT FLOOR(13.8); -- result: 13
SELECT FLOOR(-13.2); -- result: -14
```

The FLOOR(x) function returns the greatest integer not greater than x. To round towards 0 irrespective of the sign of a number:

```
SELECT TRUNCATE(13.56, 0); -- result: 13
SELECT TRUNCATE(-13.56, 1); -- result: -13.5
```

To get the absolute value of a number:

```
SELECT ABS(-12); -- result: 12
```

To get the square root of a number:

```
SELECT SQRT(9); -- result: 3
```

## USEFUL NULL FUNCTIONS

To fetch the names of the cities whose rating values are not missing:

```
SELECT name
FROM city
WHERE rating IS NOT NULL;
```

### COALESCE(x, y, ...)

To replace NULL in a query with something meaningful:

```
SELECT domain,
       COALESCE(domain, 'domain missing')
FROM contacts;
```

The COALESCE() function takes any number of arguments and returns the value of the first argument that is not NULL.

### NULLIF(x, y)

To save yourself from *division by 0* errors:

```
SELECT last_month, this_month,
  this_month * 100.0
    / NULLIF(last_month, 0)
  AS better_by_percent
FROM video_views;
```

The NULLIF(x, y) function returns NULL if x equals y, else it returns the value of x value.

## DATE AND TIME

There are 5 main time-related types in MySQL:

```
DATE    TIME    DATETIME    TIMESTAMP    YEAR
```

**DATE** – stores the year, month, and day in the YYYY-MM-DD format.

**TIME** – stores the hours, minutes, and seconds in the HH:MM:SS format.

**DATETIME** – stores the date and time in the YYYY-MM-DD HH:MM:SS format. The supported range is '1000-01-01 00:00:00' to '9999-12-31 23:59:59'.

**TIMESTAMP** – stores the date and time. The range is '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC. MySQL converts TIMESTAMP values from the current time zone to UTC for storage, and back from UTC to the current time zone for retrieval.

**YEAR** – stores the year in the YYYY format.

### INTERVALS

An interval is the duration between two points in time.

To define an interval: `INTERVAL 1 DAY`

This syntax consists of the INTERVAL keyword, a value, and a time part keyword (YEAR, QUARTER, MONTH, WEEK, DAY, HOUR, MINUTE, SECOND, MICROSECOND).

You may combine different INTERVALs using the + or - operator:

```
INTERVAL 1 YEAR + INTERVAL 3 MONTH
```

You may also use the standard SQL syntax:

```
INTERVAL '1-3' YEAR_MONTH
-- 1 year and 3 months
INTERVAL '3-12' HOUR_MINUTE
-- 3 hours 12 minutes
```

### WHAT TIME IS IT?

To answer this question, use:

- CURRENT_TIME or CURTIME – to get the current time.
- CURRENT_DATE or CURDATE – to get the current date.
- NOW() or CURRENT_TIMESTAMP – to get the current timestamp with both of the above.

### CREATING VALUES

To create a date, time, or datetime, write the value as a string and cast it to the proper type.

```
SELECT CAST('2021-12-31' AS date),
  CAST('15:31' AS time),
  CAST('2021-12-31 23:59:29' AS datetime);
```

You may skip casting in simple conditions; the database knows what you mean.

```
SELECT airline, flight_no, departure_time
FROM airport_schedule
WHERE departure_time < '12:00';
```

## EXTRACTING PARTS OF DATES

To extract a part of a date, use the functions YEAR, MONTH, WEEK, DAY, HOUR, and so on.

```
SELECT YEAR(CAST('2021-12-31' AS date));
-- result: 2021
SELECT MONTH(CAST('2021-12-31' AS date));
-- result: 12
SELECT DAY(CAST('2021-12-31' AS date));
-- result: 31
```

## DATE ARITHMETICS

To add or subtract an interval from a DATE, use the ADDDATE() function:

```
ADDDATE('2021-10-31', INTERVAL 2 MONTH);
-- result: '2021-12-31'
ADDDATE('2014-04-05', INTERVAL -3 DAY);
-- result: '2014-04-02'
```

To add or subtract an interval from a TIMESTAMP or DATETIME, use the TIMESTAMPADD() function:

```
TIMESTAMPADD(MONTH, 2,
  '2014-06-10 07:55:00');
-- result: '2014-08-10 07:55:00'
TIMESTAMPADD(MONTH, -2,
  '2014-06-10 07:55:00');
-- result: '2014-04-10 07:55:00'
```

To add or subtract TIME from a DATETIME, use the ADDTIME() function:

```
ADDTIME('2018-02-12 10:20:24', '12:43:02');
-- result: '2018-02-12 23:03:26'
ADDTIME('2018-02-12 10:20:24', '-12:43:02');
-- result: '2018-02-11 21:37:22'
```

To find the difference between two dates, use the DATEDIFF() function:

```
DATEDIFF('2015-01-01', '2014-01-02');
-- result: 364
```

To find the difference between two times, use the TIMEDIFF() function:

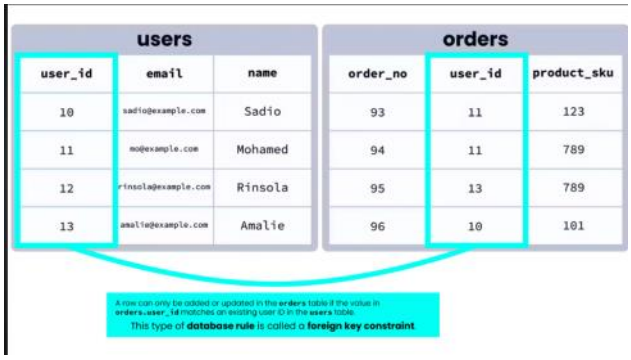
```
SELECT TIMEDIFF('09:30:00', '07:55:00');
-- result: '01:35:00'
```

To find the difference between two datetimes (in a given unit of time), use the TIMESTAMPDIFF() function. Here's an example with the difference given in weeks:

```
SELECT TIMESTAMPDIFF(
  WEEK, '2018-02-26', '2018-03-21'
); -- result: 3
```

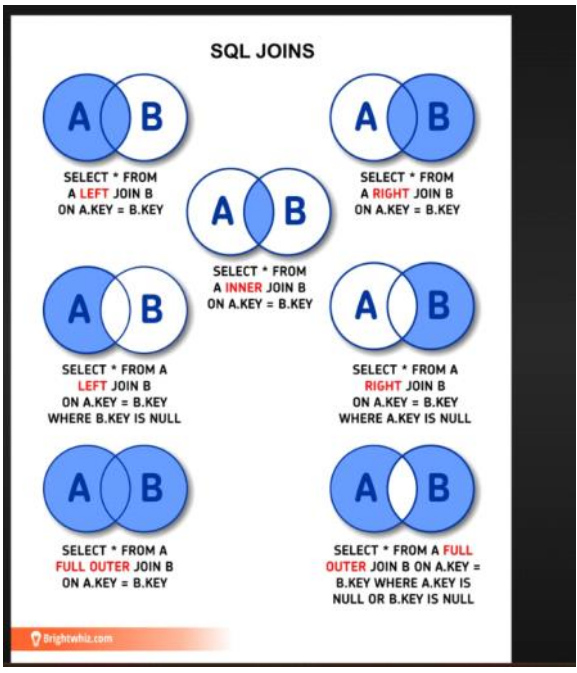
Try out the interactive **SQL from A to Z in MySQL** course at **LearnSQL.com**, and check out our other **SQL courses**.





# Joins

19 January 2025 19:18




```
CREATE TABLE employees (
   emp_id INT PRIMARY KEY,
   emp_name VARCHAR(20),
   emp_salary DOUBLE
);

CREATE TABLE departments (
   dept_id INT PRIMARY KEY,
   emp_id INT,
   dept_name VARCHAR(20),
   FOREIGN KEY (emp_id) REFERENCES employees(emp_id)
);


INSERT INTO employees (emp_id, emp_name, emp_salary)
VALUES
   (1, 'Alice', 50000.50),
   (2, 'Bob', 60000.75),
   (3, 'Charlie', 55000.00),
   (4, 'David', 45000.00);

INSERT INTO departments (dept_id, emp_id, dept_name)
VALUES
   (1, 1, 'HR'),
   (2, 2, 'Finance'),
   (3, 3, 'IT'),
   (4, 5, 'Marketing'); -- emp_id 5 does not exist in employees
```

Step 3: Apply Joins

```
SELECT e.emp_name, d.dept_name
FROM employees e
INNER JOIN departments d
ON e.emp_id = d.emp_id;
```

**2. Left Join**

Fetch all rows from employees and matching rows from departments.

```
SELECT e.emp_name, d.dept_name
FROM employees e
LEFT JOIN departments d
ON e.emp_id = d.emp_id;
```

**3. Right Join**

Fetch all rows from departments and matching rows from employees.

```
SELECT e.emp_name, d.dept_name
FROM employees e
RIGHT JOIN departments d
ON e.emp_id = d.emp_id;
```

**4. Full Outer Join**

Fetch all rows from both tables, with NULL for non-matching rows.

```
SELECT e.emp_name, d.dept_name
FROM employees e
FULL OUTER JOIN departments d
ON e.emp_id = d.emp_id;
```

**5. Self Join**

Compare rows within the same employees table (e.g., employees with the same salary).

```
SELECT e1.emp_name AS Employee1, e2.emp_name AS Employee2
FROM employees e1
INNER JOIN employees e2
ON e1.emp_salary = e2.emp_salary AND e1.emp_id != e2.emp_id;
```

**6. Union**

Combine results from two queries (eliminates duplicates).

```
SELECT emp_name FROM employees
```

UNION
SELECT dept_name FROM departments;

### 1. INNER JOIN

- **Definition**: Combines rows from two tables where the join condition is true.
- **Condition**: Matches rows from both tables based on a common column.
- **Syntax**:

```sql
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

- **Key Points**:
  - Only matching rows are included in the result.
  - Non-matching rows are excluded.

### 2. LEFT JOIN (LEFT OUTER JOIN)

- **Definition**: Retrieves all rows from the left table and matching rows from the right table. Non-matching rows in the right table are filled with `NULL`.
- **Condition**: Joins rows where the condition matches, but retains all rows from the left table.
- **Syntax**:

```sql
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.column = table2.column;
```

- **Key Points**:
  - Ensures all rows from the left table are included, even if there's no match in the right table.

### 3. RIGHT JOIN (RIGHT OUTER JOIN)

- **Definition**: Retrieves all rows from the right table and matching rows from the left table. Non-matching rows in the left table are filled with `NULL`.
- **Condition**: Joins rows where the condition matches, but retains all rows from the right table.
- **Syntax**:

```sql
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.column = table2.column;
```

- **Key Points**:
  - Ensures all rows from the right table are included, even if there's no match in the left table.

### 4. FULL OUTER JOIN

- **Definition**: Retrieves all rows from both tables. Non-matching rows are filled with `NULL` on either side.
- **Condition**: Joins rows where the condition matches but includes all rows from both tables.
- **Syntax**:

```sql
SELECT columns
FROM table1
FULL OUTER JOIN table2
ON table1.column = table2.column;
```

- **Key Points**:
  - Combines the effects of LEFT and RIGHT joins.

## 5. SELF JOIN

- **Definition**: A table is joined with itself to compare rows within the same table.
- **Condition**: Joins rows where the specified condition matches within the same table.
- **Syntax**:

```sql
SELECT a.column1, b.column2
FROM table a
INNER JOIN table b
ON a.column = b.column;
```

- **Key Points**:
  - Used for hierarchical data, finding duplicates, or comparing rows within the same table.

## 6. UNION

- **Definition**: Combines results from two SELECT statements, removing duplicates.
- **Condition**: Both SELECT statements must have the same number of columns, and the columns must have compatible data types.
- **Syntax**:

```sql
SELECT columns FROM table1
UNION
SELECT columns FROM table2;
```

- **Key Points**:
  - Removes duplicate rows by default.

## 7. UNION ALL

- **Definition**: Combines results from two SELECT statements, including duplicates.
- **Condition**: Same as UNION.
- **Syntax**:

```sql
SELECT columns FROM table1
UNION ALL
SELECT columns FROM table2;
```

- **Key Points**:
  - Does not remove duplicates, making it faster than UNION.

# CSV file import to Sql

21 January 2025 04:37

To import a CSV file into SQL using queries, you can follow these general steps. The exact approach may vary depending on the database you are using (e.g., MySQL, PostgreSQL, SQL Server, etc.). Below is an example for MySQL.

## Steps to import CSV file to MySQL:

1. **Create the table in SQL (if not already created)**:
   Ensure the table exists and its structure matches the columns of the CSV file.

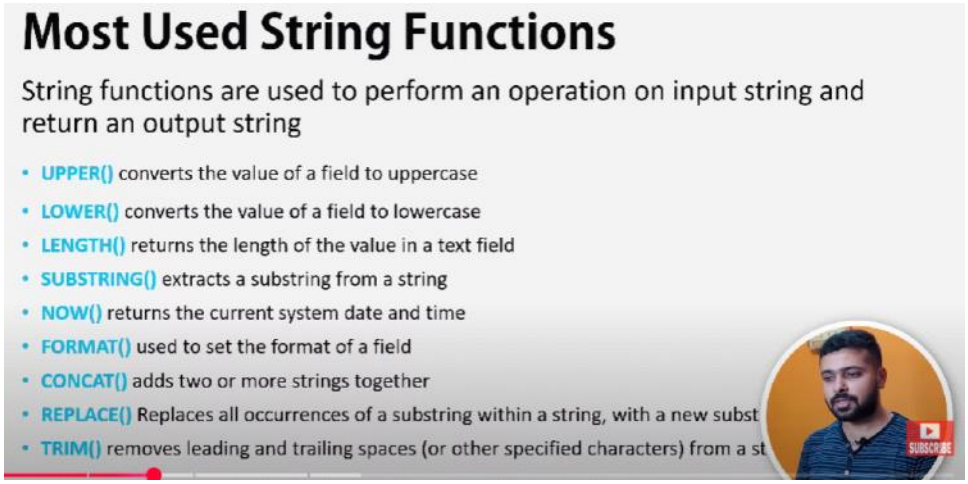
```
CREATE TABLE my_table (
   id INT,
   name VARCHAR(100),
   age INT,
   email VARCHAR(100)
);
```

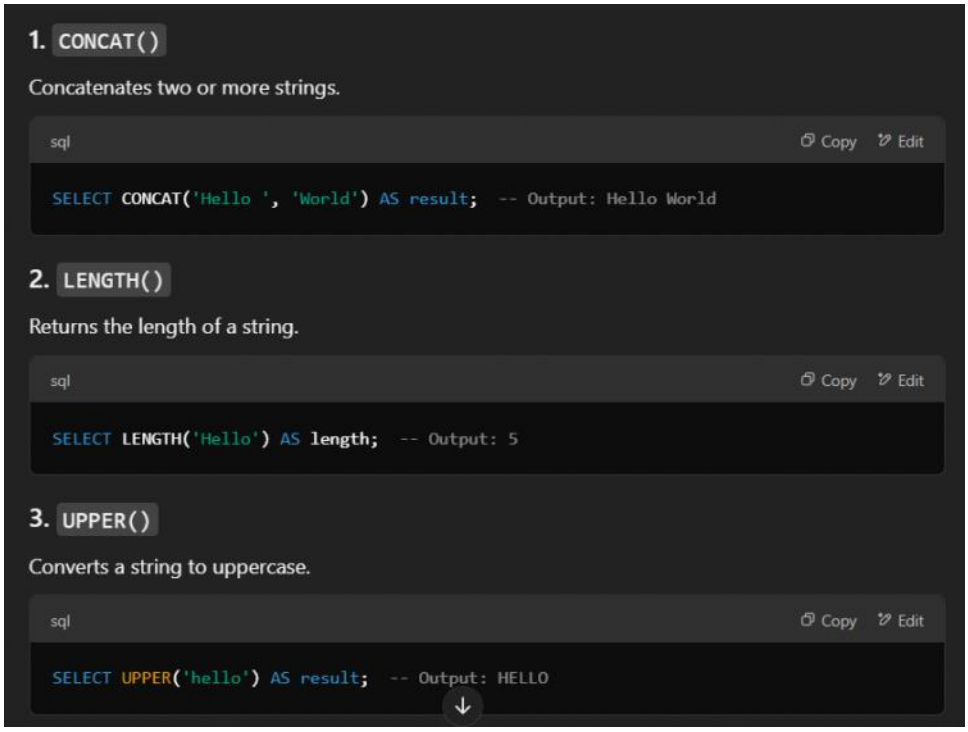
**Use the LOAD DATA INFILE command to import the CSV file**:
You can use the LOAD DATA INFILE SQL statement to load data from a CSV file into your SQL table. Here is the syntax:

```
LOAD DATA INFILE 'C:/path/to/your/file.csv'
INTO TABLE my_table
FIELDS TERMINATED BY ','  -- Specifies the separator (comma in this case)
ENCLOSED BY '"'          -- Optional: Specifies the character enclosing the fields (e.g., double quotes)
LINES TERMINATED BY '\n'  -- Specifies the line terminator (new line in this case)
IGNORE 1 LINES;         -- Optional: Ignores the first line (header) of the CSV
```

# String function in sql

21 January 2025 04:40



### 4. `LOWER()`

Converts a string to lowercase.

```sql
SELECT LOWER('HELLO') AS result;  -- Output: hello
```

### 5. `SUBSTRING()`

Extracts a substring from a string.

```sql
SELECT SUBSTRING('Hello World', 1, 5) AS result;  -- Output: Hello
```

### 6. `TRIM()`

Removes leading and trailing spaces from a string.

```sql
SELECT TRIM('   Hello   ') AS result;  -- Output: Hello
```

### 7. `REPLACE()`

Replaces all occurrences of a substring within a string.

```sql
SELECT REPLACE('Hello World', 'World', 'SQL') AS result;  -- Output: Hello SQL
```

### 8. `INSTR()`

Returns the position of the first occurrence of a substring.

```sql
SELECT INSTR('Hello World', 'World') AS position;  -- Output: 7
```

### 9. `LEFT()`

Extracts the left part of a string (n characters).

```sql
SELECT LEFT('Hello World', 5) AS result;  -- Output: Hello
```

### 10. `RIGHT()`

Extracts the right part of a string (n characters).

```sql
SELECT RIGHT('Hello World', 5) AS result;  -- Output: World
```

### 11. `REVERSE()`

Reverses the characters of a string.

```sql
SELECT REVERSE('Hello') AS result;  -- Output: olleH
```

### 12. `CHARINDEX()` (SQL Server)

Returns the position of the first occurrence of a substring in a string.

```sql
SELECT CHARINDEX('World', 'Hello World') AS position;  -- Output: 7
```

# Group BY

21 January 2025 04:46

- The GROUP BY clause in SQL is used to group rows with the same values in specified columns and perform aggregate functions (like COUNT, SUM, AVG, etc.) on each group.

- Combine it with HAVING for conditional filtering on grouped results

```
create DATABASE sales_db;
USE sales_db;
CREATE TABLE sales(
sale_id INT auto_increment primary KEY,
product_name VARCHAR(50),
category varchar(50),
amount DECIMAL(10,2)
);
insert INTO sales(product_name,category,amount) VALUES
('Laptop', 'Electronics', 1200.00),
('Phone', 'Electronics', 800.00),
('Table', 'Furniture', 150.00),
('Chair', 'Furniture', 100.00),
('Headphones', 'Electronics', 50.00);

select * from sales;
SELECT category,SUM(amount) AS total_sales from sales group by
category;
```

output

Electronics 2050.00

Furniture 250.00

```
CREATE DATABASE students_db;
USE students_db;

CREATE TABLE students (
   student_id INT AUTO_INCREMENT PRIMARY KEY,
   class VARCHAR(10),
   subject VARCHAR(50),
   marks INT
);

INSERT INTO students (class, subject, marks) VALUES
```

```
('10A', 'Math', 85),
('10A', 'Science', 90),
('10B', 'Math', 70),
('10B', 'Science', 75),
('10A', 'Math', 95);
select * from students;

select class, subject,avg(marks) as average_marks from
students group by class,subject;

select class, avg(marks) as average_marks from students
group by class having avg(marks)>80;
```