Table of contents:

# 0) [What is DBMS:](#_What_is_DBMS:)

1[) What are relational databases ?](#_What_are_relational)

[2) Relationships](#_Relationships)

[3) RDBMS Benefits and Limitations](#_RDBMS_Benefits_and)

4[)sql vs no-sql](#_SQL_VS_NO-SQL)

[4.1)benifits of no sql](#_What_are_the)

5)[DATA TYPES](#_DATA_TYPES:)

5.1)[MySQL String Data Types](#_MySQL_String_Data)

**5.2)** [MySQL Numeric Data Types](#_Mysql_numeric_datatypes:)

**5.3)** [MySQL Date and Time Data Types](#_MySQL_Date_and)

[Like:](#_Like)

[Date\_format](#_Date_format:)

[Stringfunctions(chat gpt response while doingsql problems on string functionS)](#_String_fuctions(final))

[Subqueries:](#_Subqueries:)

# What is DBMS:

* **DBMS can be classified into two types:** Relational Database Management System (RDBMS) and Non-Relational Database Management System (NoSQL or Non-SQL)
* **RDBMS:**Data is organized in the form of tables and each table has a set of rows and columns. The data are related to each other through primary and foreign keys.
* **NoSQL:**Data is organized in the form of key-value pairs, documents, graphs, or column-based. These are designed to handle large-scale, high-performance scenarios.

# 

# What are relational databases ?

Relational databases are a type of database that store and manage data in a structured format using tables. Each table consists of rows and columns, where each column represents a different data attribute, and each row represents a record containing values for each attribute.

SQL (Structured Query Language) is used to interact with the data. Key features include data integrity, support for complex queries, and ACID properties (Atomicity, Consistency, Isolation, Durability). Examples include MySQL, PostgreSQL, Oracle, and Microsoft SQL Server. They are widely used in various applications like to store and manage data in financial services, and healthcare.

Ex of relational database:

**Example**: In a library database, one table (Books) might store book details with columns like BookID, Title, and AuthorID, while another table (Authors) stores author details with AuthorID and Name. The AuthorID column links the two tables, establishing a relationship between books and authors.

**Key Concepts**

1. **Tables (Relations)**:
   * A table is a collection of related data entries and consists of columns and rows.
   * Each table is identified by a name (e.g., customers, orders).
2. **Rows (Tuples)**:
   * A row in a table represents a single, implicitly structured data item (record).
   * Each row has a unique identifier, typically called a primary key.
3. **Columns (Attributes)**:
   * Each column in a table represents a data field, and each field in a row contains data of a specific type.
   * Columns have names and data types (e.g., INTEGER, VARCHAR).
4. **Primary Key**:
   * A primary key is a column or a set of columns that uniquely identifies each row in a table.
   * No two rows can have the same primary key value.
5. **Foreign Key**:
   * A foreign key is a column or a set of columns in one table that uniquely identifies rows in another table.
   * It creates a link between the data in the two tables.
6. **Indexes**:
   * Indexes are used to speed up the retrieval of data.
   * They work like an index in a book, allowing quick location of rows in a table.

**Features**

* **Structured Query Language (SQL)**:
  + SQL is the standard language used to interact with relational databases.
  + SQL commands are used to perform various operations such as querying, updating, inserting, and deleting data.
* **ACID Properties**:
  + Relational databases ensure data integrity and reliability through ACID properties: Atomicity, Consistency, Isolation, and Durability.
* **Data Integrity**:)(data integrity in database defines the set of rules that are used to maintain the information quality)
  + Constraints such as primary keys, foreign keys, and unique constraints are used to maintain data integrity.

Examples of Relational Database Management Systems (RDBMS)

 **MySQL**:

* Open-source RDBMS known for its reliability and performance.

 **PostgreSQL**:

* Open-source RDBMS known for its advanced features and standards compliance.

 **Oracle Database**:

* A widely-used commercial RDBMS known for its robustness and scalability.

 **Microsoft SQL Server**:

* A commercial RDBMS from Microsoft, known for its integration with other Microsoft products.

**Use Cases**

* **Enterprise Applications**:
  + Relational databases are commonly used in enterprise applications such as Customer Relationship Management (CRM) systems, Enterprise Resource Planning (ERP) systems, and e-commerce platforms.
* **Financial Systems**:
  + They are used in banking and financial applications due to their strong emphasis on data integrity and transaction management.
* **Healthcare Systems**:
  + Used in managing patient records, billing information, and other healthcare-related data.

# Relationships

Relational databases use **keys** to create links(relations) between tables.

A **primary key** is a unique identifier for a row of data.

A **foreign key** is a column or combination of columns used to establish and enforce a link between the data in two tables.

Consider a simple relational database with two tables: Employees and Orders.

**Employees Table:**

| **EmployeeId** | **FirstName** | **LastName** |
| --- | --- | --- |
| 1 | John | Doe |
| 2 | Jane | Smith |
| 3 | Alice | Johnson |

**Orders Table:**

| **OrderId** | **EmployeeId** | **Product** |
| --- | --- | --- |
| 1 | 3 | Apples |
| 2 | 1 | Bananas |
| 3 | 2 | Cherries |

In the Orders table, EmployeeId serves as the foreign key creating a relationship between Orders and Employees. This allows queries that involve data in both tables, such as finding all orders placed by John Doe.

**SQL Query Example**

SELECT Orders.OrderId, Orders.Product, Employees.FirstName, Employees.LastName

FROM Orders

INNER JOIN Employees ON Orders.EmployeeId = Employees.EmployeeId;

This SQL query retrieves data from the Orders and Employees tables, using a JOIN clause to combine rows from both tables based on the matching EmployeeId.

Overall, relational databases provide a powerful mechanism for defining relationships within data and enabling efficient data retrieval.

# RDBMS Benefits and Limitations

Here are some of the benefits of using an RDBMS:

* **Structured Data**: RDBMS allows data storage in a structured way, using rows and columns in tables. This makes it easy to manipulate the data using SQL (Structured Query Language), ensuring efficient and flexible usage.
* **ACID Properties**: ACID stands for Atomicity, Consistency, Isolation, and Durability. These properties ensure reliable and safe data manipulation in a RDBMS, making it suitable for mission-critical applications.
* **Normalization**: RDBMS supports data normalization, a process that organizes data in a way that reduces data redundancy and improves data integrity.
* **Scalability**: RDBMSs generally provide good scalability options, allowing for the addition of more storage or computational resources as the data and workload grow. RDBMSs are generally better suited for vertical scaling (adding more power to a single server)
* **Data Integrity**: RDBMS provides mechanisms like constraints, primary keys, and foreign keys to enforce data integrity and consistency, ensuring that the data is accurate and reliable.
* **Security**: RDBMSs offer various security features such as user authentication, access control, and data encryption to protect sensitive data.
* **SQL as a Standard Language**: SQL is a widely accepted standard for querying and manipulating data. It provides a powerful and easy-to-use interface for interacting with the database.

### Limitations of RDBMS

1. **Complexity and Cost**:
   * **Setup and Maintenance**: RDBMSs can be complex to set up and maintain, often requiring specialized personnel. The licensing costs for commercial RDBMSs can also be high.
2. **Scalability Issues**:
   * **Vertical Scaling Limitations**: RDBMSs are generally better suited for vertical scaling (adding more power to a single server) rather than horizontal scaling (adding more servers), which can be a limitation for very large datasets or high throughput requirements.
   * **It means** Vertical scaling is possible and commonly used with RDBMSs, but horizontal scaling is also possible, though it comes with additional challenges and complexity.
3. **Performance Overhead**:
   * The overhead of ensuring ACID properties, maintaining indexes, and handling complex joins can result in performance issues, especially for large-scale applications with high transaction volumes.
4. **Flexibility**:(rigid schema)
   * **Schema Rigidity**: RDBMSs require a predefined schema, which makes it less flexible to adapt to changing data requirements. Altering the schema can be cumbersome and time-consuming.
5. **Handling Unstructured Data**:
   * **Limited Support**: RDBMSs are not inherently designed to handle unstructured or semi-structured data like documents, images, or JSON, which can be better managed by NoSQL databases.
6. **Network Latency**:
   * **Distributed Systems**: In a distributed RDBMS setup, network latency can become a bottleneck, affecting the performance of the database operations.
7. **Complex Query Processing**:
   * **Resource-Intensive Joins**: Complex queries involving multiple joins can be resource-intensive and slow, which can degrade performance in large databases.
8. **Limited Big Data Capabilities**:
   * **Scalability Constraints**: RDBMSs can struggle with the storage and processing requirements of big data applications, where NoSQL and distributed databases often provide better solutions.

# SQL VS NO-SQL

| **Feature** | **SQL Databases** | **NoSQL Databases** |
| --- | --- | --- |
| **Schema** | Predefined, rigid schema | Flexible, dynamic schema |
| **Data Storage** | Tables with rows and columns | Various models: document, key-value, column-family, graph |
| **Query Language** | SQL (Structured Query Language) | Varies (e.g., MongoDB Query Language, CQL, custom APIs) |
| **Transaction Model** | ACID (Atomicity, Consistency, Isolation, Durability) | BASE (Basically Available, Soft state, Eventual consistency) |
| **Schema Flexibility** | Rigid, changes require downtime | Flexible, schema changes are easy |
| **Scalability** | Vertical scaling (scale-up) | Horizontal scaling (scale-out) |
| **Performance** | Optimized for complex queries and joins | Optimized for fast read/write operations |
| **Consistency** | Strong consistency | Eventual consistency (can be configured for stronger guarantees) |
| **Handling Hierarchical Data** | Requires multiple tables and JOINs | Better suited with document and graph models |
| **Big Data Handling** | Limited scalability | Designed for big data and real-time applications |
| **Use Cases** | Financial systems, ERP, CRM | Social networks, real-time analytics, content management |
|  |  |  |
| **Complex Queries** | Supports complex queries and aggregations | Limited support (varies by database) |
| **Data Integrity** | High due to ACID properties | Lower due to BASE properties |
| **Popular Examples** | MySQL, PostgreSQL, Microsoft SQL Server, Oracle Database | MongoDB, Cassandra, Redis, Couchbase, Neo4j |

# What are the benefits of NoSQL databases?

NoSQL databases offer many benefits over relational databases. NoSQL databases have flexible data models, scale horizontally, have incredibly fast queries, and are easy for developers to work with.

* Flexible data models

NoSQL databases typically have very flexible schemas. A flexible schema allows you to easily make changes to your database as requirements change. You can iterate quickly and continuously integrate new application features to provide value to your users faster.

* Horizontal scaling

Most SQL databases require you to scale-up vertically (migrate to a larger, more expensive server) when you exceed the capacity requirements of your current server. Conversely, most NoSQL databases allow you to scale-out horizontally, meaning you can add cheaper commodity servers whenever you need to.

* Fast queries

Queries in NoSQL databases can be faster than SQL databases. Why? Data in SQL databases is typically normalized, so queries for a single object or entity require you to join data from multiple tables. As your tables grow in size, the joins can become expensive. However, data in NoSQL databases is typically stored in a way that is optimized for queries. The rule of thumb when you use MongoDB is data that is accessed together should be stored together. Queries typically do not require joins, so the queries are very fast.

* Easy for developers

Some NoSQL databases like MongoDB map their data structures to those of popular programming languages. This mapping allows developers to store their data in the same way that they use it in their application code. While it may seem like a trivial advantage, this mapping can allow developers to write less code, leading to faster development time and fewer bugs.

Drawbacks:

 Limited query capabilities

 Weaker consistency guarantees

 Complexity in data relationships,Data durability

# DATA TYPES:

# **MySQL String Data Types**

1)CHAR(SIZE)(it is a string data type of fixed length)

### CHAR(Size)

**Explanation:**

* **Description:** CHAR is a fixed-length string data type. It can store numbers, letters, and special characters. The length of the string is fixed, meaning if the input string is shorter than the defined size, it will be padded with spaces to reach the specified length.
* **Size:** The size can range from 0 to 255 characters. The default size is 1.
* **Usage:** CHAR is typically used when you know that the data entries will be of a consistent length. For example, storing two-character state codes or fixed-length IDs.

**Example:**

CREATE TABLE example\_char (

col CHAR(10)

);

INSERT INTO example\_char (col) VALUES ('Hello');

In this example:

* A table example\_char is created with one column col of type CHAR with a length of 10.
* When inserting the value 'Hello', it will be stored as 'Hello ' (with 5 trailing spaces) because the length is fixed at 10 characters.

MY QUE: so now if print value in first row of col it prints hello+5 spaces

es, that's correct. When you insert the value 'Hello' into a CHAR(10) column, it is stored as 'Hello ' (with 5 trailing spaces). If you retrieve and print this value, it will include those trailing spaces. Here’s an example of how it would work:

sql

Copy code

-- Creating the table

CREATE TABLE example\_char (

col CHAR(10)

);

-- Inserting a value

INSERT INTO example\_char (col) VALUES ('Hello');

-- Selecting the value

SELECT col FROM example\_char;

When you run the SELECT query, the output will show 'Hello ' with the trailing spaces.

### Example:

sql

Copy code

mysql> SELECT col FROM example\_char;

+------------+

| col |

+------------+

| Hello |

+------------+

### VARCHAR(Size)

**Explanation:**

* **Description:** VARCHAR is a variable-length string data type. It can store numbers, letters, and special characters. Unlike CHAR, VARCHAR does not pad the string with spaces if it is shorter than the specified length.
* **Size:** The size can range from 0 to 65535 characters, although the actual maximum size depends on the maximum row size (usually 65,535 bytes).
* **Usage:** VARCHAR is typically used when the length of the data entries can vary. For example, storing names, addresses, or any other text that doesn't have a fixed length.

**Example:**

sql

Copy code

CREATE TABLE example\_varchar (

col VARCHAR(255)

);

INSERT INTO example\_varchar (col) VALUES ('Hello, World!');

In this example:

* A table example\_varchar is created with one column col of type VARCHAR with a length of 255.
* When inserting the value 'Hello, World!', it is stored exactly as entered without any additional spaces.

### BINARY(Size)

**Explanation:**

* **Description:** BINARY is a fixed-length string data type similar to CHAR, but it stores binary byte strings. The data is stored as raw bytes.
* **Size:** The size parameter specifies the column length in bytes, and it can range from 0 to 255. The default size is 1.
* **Usage:** BINARY is typically used when you need to store fixed-length binary data, such as encryption keys or other binary data where the length is always the same.

**Example:**

sql

Copy code

CREATE TABLE example\_binary (

col BINARY(4)

);

INSERT INTO example\_binary (col) VALUES ('Hi');

In this example:

* A table example\_binary is created with one column col of type BINARY with a length of 4 bytes.
* When inserting the value 'Hi', it will be stored as Hi\0\0 (with 2 null bytes, represented as \0, to make the length 4 bytes).

My ques:

what will be the output when we retrieve from the bianry data tupe columns

When you retrieve data from a column of type BINARY, you will get the stored binary data. This data may not be human-readable directly in the terminal or client interface because it represents raw bytes.

For example, if you insert the value 'Hi' into a BINARY(4) column, it will be stored as Hi\0\0. When you retrieve this value, you will get exactly what was stored, including any null bytes.

Here's an example of how you might retrieve data from a BINARY column:

sql

Copy code

SELECT col FROM example\_binary;

The output you will see might look something like this:

diff

Copy code

+------+

| col |

+------+

| Hi |

+------+

### VARBINARY(Size)

**Explanation:**

* **Description:** VARBINARY is a variable-length binary string data type, similar to VARCHAR but for binary data. It stores binary byte strings.
* **Size:** The size parameter specifies the maximum column length in bytes, and it can range from 0 to 65535.
* **Usage:** VARBINARY is used when you need to store variable-length binary data, such as images, files, or any other binary data where the length can vary.

**Example:**

sql

Copy code

CREATE TABLE example\_varbinary (

col VARBINARY(100)

);

INSERT INTO example\_varbinary (col) VALUES (X'48656C6C6F');

Here's an example of how you might retrieve data from a VARBINARY column:

sql

Copy code

SELECT col FROM example\_varbinary;

The output you will see might look something like this:

diff

Copy code

+----------------+

| col |

+----------------+

| Hello |

+----------------+

In this example:

* A table example\_varbinary is created with one column col of type VARBINARY with a maximum length of 100 bytes.
* When inserting the value X'48656C6C6F' (which is the hexadecimal representation of the ASCII string 'Hello'), it will be stored as raw binary bytes.

# the data is stored as raw bytes,"

When I say "the data is stored as raw bytes," it means that the binary data you insert into a VARBINARY column is stored exactly as it is, without any modifications or interpretations by the database.

Here's what happens when you insert binary data into a VARBINARY column:

1. You provide binary data in some format, such as hexadecimal, binary literals, or using some programming language's binary data representation.
2. The database takes this binary data and stores it directly in the VARBINARY column as a sequence of raw bytes.
3. The database does not attempt to interpret or understand the content of the binary data. It simply stores and retrieves it as a sequence of bytes.

### TEXT(Size)

**Explanation:**

 **Description:** TEXT is a data type used for storing large strings of text. It can hold a string with a maximum length of 65,535 characters.

 **Size:** TEXT does not require a size parameter. It has a fixed maximum length of 65,535 characters.

 **Usage:** TEXT is suitable for storing large amounts of textual data, such as paragraphs of text, articles, or lengthy descriptions.

**Example:**

sql

Copy code

CREATE TABLE example\_text (

col TEXT

);

INSERT INTO example\_text (col) VALUES ('This is a sample text.');

In this example:

* A table example\_text is created with one column col of type TEXT.
* When inserting the value 'This is a sample text.', it is stored in the TEXT column.

Next, let's discuss the TINYTEXT data type:

### TINYTEXT

**Explanation:**

* **Description:** TINYTEXT is a data type used for storing small strings of text. It can hold a string with a maximum length of 255 characters.
* **Size:** Unlike some other data types, TINYTEXT does not require a size parameter. It has a fixed maximum length of 255 characters.
* **Usage:** TINYTEXT is suitable for storing small amounts of textual data, such as short descriptions, names, or labels.

**Example:**

sql

Copy code

CREATE TABLE example\_tinytext (

col TINYTEXT

);

INSERT INTO example\_tinytext (col) VALUES ('Short text.');

In this example:

* A table example\_tinytext is created with one column col of type TINYTEXT.
* When inserting the value 'Short text.', it is stored in the TINYTEXT column.

### LONGTEXT

**Explanation:**

* **Description:** LONGTEXT is a data type used for storing very large strings of text. It can hold a string with a maximum length of 4,294,967,295 characters.
* **Size:** LONGTEXT does not require a size parameter. It has a fixed maximum length of 4,294,967,295 characters.
* **Usage:** LONGTEXT is suitable for storing extremely large amounts of textual data, such as full-length books, extensive articles, or any other content that exceeds the maximum length supported by MEDIUMTEXT.

**Example:**

sql

Copy code

CREATE TABLE example\_longtext (

col LONGTEXT

);

INSERT INTO example\_longtext (col) VALUES ('This is a very long text.');

In this example:

* A table example\_longtext is created with one column col of type LONGTEXT.
* When inserting the value 'This is a very long text.', it is stored in the LONGTEXT column.

### ENUM

**Explanation:**

* **Description:** ENUM is a data type used for defining columns that can have only one of a predefined set of values. Each value in the ENUM list is assigned a numeric index starting from 1.
* **Usage:** ENUM is useful when a column's value is expected to be one of a limited, predefined set of options. It provides data integrity by restricting the values that can be inserted into the column.
* **Maximum Values:** An ENUM column can have a maximum of 65,535 distinct elements.
* **Syntax:** The syntax for defining an ENUM column is: ENUM(value1, value2, value3, ...)

**Example:**

sql

Copy code

CREATE TABLE example\_enum (

col ENUM('Option 1', 'Option 2', 'Option 3')

);

INSERT INTO example\_enum (col) VALUES ('Option 2');

In this example:

* A table example\_enum is created with one column col of type ENUM with three possible values: 'Option 1', 'Option 2', and 'Option 3'.
* When inserting the value 'Option 2', it is stored in the ENUM column.

Certainly, here are some common use cases for the ENUM data type:

1. **Gender:** You might use ENUM to store gender information where the possible values are 'Male', 'Female', and 'Other'.
2. **Status:** ENUM is often used to represent status values, such as 'Active', 'Inactive', and 'Pending'.
3. **Priority Levels:** For tasks or tickets, you could use ENUM to represent priority levels like 'Low', 'Medium', and 'High'.

### SET

**Explanation:**

* **Description:** SET is a data type used for defining columns that can have zero or more values from a predefined set of options. Each value in the SET list is assigned a numeric index starting from 1.
* **Usage:** SET is useful when a column's value can be a combination of multiple options from a predefined set. It provides data integrity by restricting the values that can be inserted into the column to only those specified in the set.
* **Maximum Values:** A SET column can have a maximum of 64 distinct elements.
* **Syntax:** The syntax for defining a SET column is: SET(value1, value2, value3, ...)

**Example:**

sql

Copy code

CREATE TABLE example\_set (

col SET('Option 1', 'Option 2', 'Option 3')

);

INSERT INTO example\_set (col) VALUES ('Option 1, Option 3');

In this example:

* A table example\_set is created with one column col of type SET with three possible values: 'Option 1', 'Option 2', and 'Option 3'.
* When inserting the value 'Option 1, Option 3', it is stored in the SET column, indicating that both 'Option 1' and 'Option 3' are selected.

### Your Turn

Do you have any questions or doubts about the SET data type? Or would you like to move on to the next topic?

3.5

mext

ChatGPT

Next, let's discuss the BLOB data type:

### BLOB

**Explanation:**

* **Description:** BLOB (Binary Large Object) is a data type used for storing binary data, such as images, audio/video files, or any other type of binary data.
* **Size:** The size parameter specifies the maximum column length in bytes. It can hold up to 65,535 bytes.
* **Usage:** BLOB is suitable for storing large binary objects where the content is not interpreted by the database, such as images or multimedia files.
* **Variants:** There are different variants of BLOB depending on the size requirement:
  + TINYBLOB: Can hold up to 255 bytes.
  + MEDIUMBLOB: Can hold up to 16,777,215 bytes.
  + LONGBLOB: Can hold up to 4,294,967,295 bytes.
  + For MySQL, if you use just "BLOB" without specifying a size, it defaults to the TINYBLOB datatype, which can hold up to 255 bytes.

**Example:**

sql

Copy code

CREATE TABLE example\_blob (

col BLOB

);

INSERT INTO example\_blob (col) VALUES (LOAD\_FILE('/path/to/image.jpg'));

In this example:

* A table example\_blob is created with one column col of type BLOB.
* When inserting binary data (such as an image file) into the BLOB column, it is stored in its binary format.

# Mysql numeric datatypes:

1. **BIT(Size):**
   * **Explanation:** BIT is used for bit-value types, representing binary data. The size parameter specifies the number of bits per value.
   * **Example:**

sql

Copy code

CREATE TABLE example\_bit (

col BIT(8)

);

INSERT INTO example\_bit (col) VALUES (B'10101010');

* + **Output:** When retrieving, you will see the binary representation of the stored bit sequence.

1. **INT(Size):**
   * **Explanation:** INT is used for storing signed integers. The size parameter specifies the display width in digits.
   * **Example:**

sql

Copy code

CREATE TABLE example\_int (

col INT(10)

);

INSERT INTO example\_int (col) VALUES (1234567890);

* + **Output:** When retrieving, you will see the stored integer value.

1. **FLOAT(Size, d):**
   * **Explanation:** FLOAT is used for storing floating-point numbers with a specified precision. The size parameter specifies the total number of digits, and d specifies the number of digits after the decimal point.
   * **Example:**

sql

Copy code

CREATE TABLE example\_float (

col FLOAT(7, 3)

);

INSERT INTO example\_float (col) VALUES (123.456);

* + **Output:** When retrieving, you will see the stored floating-point number with the specified precision.

1. **DOUBLE(Size, d):**
   * **Explanation:** DOUBLE is used for storing double-precision floating-point numbers. The size parameter specifies the total number of digits, and d specifies the number of digits after the decimal point.
   * **Example:**

sql

Copy code

CREATE TABLE example\_double (

col DOUBLE(15, 6)

);

INSERT INTO example\_double (col) VALUES (1234567890.123456);

* + **Output:** When retrieving, you will see the stored double-precision floating-point number with the specified precision.

1. **DECIMAL(Size, d):**
   * **Explanation:** DECIMAL is used for storing fixed-point numbers with exact precision. The size parameter specifies the total number of digits, and d specifies the number of digits after the decimal point.
   * **Example:**

sql

Copy code

CREATE TABLE example\_decimal (

col DECIMAL(10, 2)

);

INSERT INTO example\_decimal (col) VALUES (12345.67);

* + **Output:** When retrieving, you will see the stored fixed-point number with the specified precision.

**6.** **DEC(size, d)**

DEC is an alias for the DECIMAL data type in MySQL. It allows you to specify a fixed-point number with a total number of digits (size) and a number of digits after the decimal point (d).

Here's an example:

sql

Copy code

CREATE TABLE example\_decimal (

col DEC(8, 2)

);

INSERT INTO example\_decimal (col) VALUES (1234.56);

In this example, col is defined as a DECIMAL (or DEC) data type with a total of 8 digits, with 2 digits after the decimal point. The value 1234.56 is then inserted into the col column.

Executing SELECT queries on these tables will allow you to retrieve the data and see the expected output for each numeric data type. Let me know if you need further clarification!

1. **BOOL**:
   * **Explanation**: BOOL is used to represent Boolean values, where zero is considered as false and nonzero values are considered as true.
   * **Example**:

sql

Copy code

CREATE TABLE example\_bool (

col BOOL

);

INSERT INTO example\_bool (col) VALUES (TRUE); -- Inserts true into the col column

# **MySQL Date and Time Data Types**

1. **DATE**:
   * **Explanation**: DATE is used to store dates in the format YYYY-MM-DD, representing years, months, and days. It supports a range from '1000-01-01' to '9999-12-31'.
   * **Example**:

sql

Copy code

CREATE TABLE example\_date (

col DATE

);

INSERT INTO example\_date (col) VALUES ('2024-06-12'); -- Inserts the date '2024-06-12' into the col column

1. **DATETIME(fsp)**:
   * **Explanation**: DATETIME is used to store both date and time values in the format YYYY-MM-DD hh:mm

. It supports a range from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'.

* + **Example**:

sql

Copy code

CREATE TABLE example\_datetime (

col DATETIME

);

INSERT INTO example\_datetime (col) VALUES ('2024-06-12 12:34:56'); -- Inserts the datetime '2024-06-12 12:34:56' into the col column

1. **TIMESTAMP(fsp)**:
   * **Explanation**: TIMESTAMP is used to store the timestamp, which represents the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). It supports a range from '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC.
   * **Example**:

sql

Copy code

CREATE TABLE example\_timestamp (

col TIMESTAMP

);

INSERT INTO example\_timestamp (col) VALUES ('2024-06-12 12:34:56'); -- Inserts the timestamp '2024-06-12 12:34:56' into the col column

1. **TIME(fsp)**:
   * **Explanation**: TIME is used to store time values in the format hh:mm

. It supports a range from '-838:59:59' to '838:59:59'.

* + **Example**:

sql

Copy code

CREATE TABLE example\_time (

col TIME

);

INSERT INTO example\_time (col) VALUES ('12:34:56'); -- Inserts the time '12:34:56' into the col column

1. **YEAR**:
   * **Explanation**: YEAR is used to store years in four-digit format. It supports values from 1901 to 2155, and 0000.
   * **Example**:

sql

Copy code

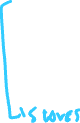
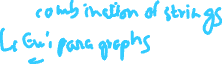
CREATE TABLE example\_year (

col YEAR

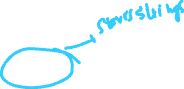
);

INSERT INTO example\_year (col) VALUES (2024); -- Inserts the year '2024' into the col column

Overview:



(X'48656C6C6F')



Enum vs set:



# Create database in sql(chatgpt link: [link](https://chatgpt.com/share/2e6bccf5-f669-4891-bce0-5e8fd3abc032))

 **Creating a Database using CREATE DATABASE**:

* Example:

CREATE DATABASE GeeksForGeeks;

* This command creates a new database named "GeeksForGeeks".

 **Listing Databases using SHOW DATABASES**:

SHOW DATABASES;

* This command lists all databases present in the system, including the newly created "GeeksForGeeks" database.

 **Using a Database with USE statement**:

* Example:

USE GeeksForGeeks;

* This command selects the "GeeksForGeeks" database for use in subsequent SQL queries.

 **Important Points about SQL CREATE DATABASE**:

* Example:
  + A CREATE DATABASE statement is used to create a database.
  + A database consists of tables where data is stored.
  + Database names are case-insensitive.
  + Database names must be unique.
  + Database names have a limit of 128 characters.

# Deleting(drop) database:

 **Using DROP DATABASE to delete a database**:

* Example:

DROP DATABASE GeeksForGeeks;

* This command deletes the database named "GeeksForGeeks" along with all its associated data, such as tables, views, indexes, etc.

 **Verifying the deletion with SHOW DATABASES**:

* Example:

SHOW DATABASES;

* After executing the DROP DATABASE command, this command lists all databases in the system. If "GeeksForGeeks" is no longer listed, it confirms the successful deletion.

 **Using DROP DATABASE IF EXISTS to avoid errors**:

* Example:

DROP DATABASE IF EXISTS GeeksForGeeks;

* This command ensures that the database "GeeksForGeeks" is deleted only if it exists, preventing errors if the database does not exist.

 **Important Points about DROP DATABASE**:

* Example:
  + Deleting a Database: Deleting a database is a permanent action and results in the loss of all data stored in it.
  + Backup: It's essential to take a proper backup of the database before deletion to prevent data loss.
  + Privileges: Ensure necessary privileges are granted to delete the database, typically requiring administrative rights.
  + Database State: A database can be dropped regardless of its state, whether it's offline, read-only, suspect, etc.
  + Replication: For databases involved in replication, replication needs to be removed before dropping the database.
  + System Databases: System databases cannot be dropped as they are essential for the functioning of the database management system.

RENAMING DATABASE:

 **Syntax**: To rename a database in SQL, you can use the following syntax:

ALTER DATABASE [current\_database\_name]

MODIFY NAME = [new\_database\_name];

 **Example**: Suppose we have a database named Test, and we want to rename it to Example.

ALTER DATABASE Test MODIFY NAME = Example;

Here's the correct syntax for renaming a database in MySQL:

RENAME DATABASE alter1 TO alter33;

This statement renames the alter1 database to alter33.

Here's a step-by-step guide to renaming a database in MySQL:

1. Dump the existing database alter1 to a SQL dump file using the mysqldump command-line tool:

mysqldump -u username -p --databases alter1 > alter1\_dump.sql

1. Create a new database named alter33:

CREATE DATABASE alter33;

1. Import the SQL dump file into the newly created database:

mysql -u username -p alter33 < alter1\_dump.sql

This series of steps effectively renames the database alter1 to alter33.

# TABLE(CREATE,DROP,….):LINK

## CREATING A TABLE:

The SQL CREATE TABLE statement is a powerful tool used to define the structure of a table within a database. It specifies the table name, column names, data types, and constraints. Below, we will cover the syntax, examples, and key points about the CREATE TABLE statement in SQL.

### Syntax

To create a table in SQL, you can use the following syntax:

CREATE TABLE table\_name (

column1 datatype (size),

column2 datatype (size),

...

columnN datatype (size)

);

* **table\_name**: The name of the table.
* **column**: The name of the column.
* **datatype**: The type of data the column will hold (e.g., INT, VARCHAR).
* **size**: The size of the datatype (for VARCHAR, it defines the maximum number of characters).

### Example 1: Creating a Table with a Primary Key

Here, we create a table named Employee with several columns, including a primary key.

sql

Copy code

CREATE TABLE Employee (

EmployeeID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

Department VARCHAR(50),

Salary DECIMAL(10, 2)

);

### Example 2: Creating a Table with Constraints and Inserting Data

In this example, we create a Customer table and insert data into it. This table includes a CHECK constraint to ensure the age is between 0 and 99.

CREATE TABLE Customer (

CustomerID INT PRIMARY KEY,

CustomerName VARCHAR(50),

LastName VARCHAR(50),

Country VARCHAR(50),

Age INT CHECK (Age >= 0 AND Age <= 99),

Phone INT

);

-- Inserting data into the Customer table

INSERT INTO Customer (CustomerID, CustomerName, LastName, Country, Age, Phone)

VALUES

(1, 'Shubham', 'Thakur', 'India', 23, 'xxxxxxxxxx'),

(2, 'Aman', 'Chopra', 'Australia', 21, 'xxxxxxxxxx'),

(3, 'Naveen', 'Tulasi', 'Sri Lanka', 24, 'xxxxxxxxxx'),

(4, 'Aditya', 'Arpan', 'Austria', 21, 'xxxxxxxxxx'),

(5, 'Nishant', 'Jain', 'Spain', 22, 'xxxxxxxxxx');

### Example 3: Creating a Table from Another Table

You can create a new table by copying the structure and data from an existing table.

CREATE TABLE SubTable AS

SELECT CustomerID, CustomerName

FROM Customer;

This statement creates a new table named SubTable with the same CustomerID and CustomerName columns and their data from the Customer table.

### Important Points

* **Structure Definition**: The CREATE TABLE statement defines the table structure, including column names and data types.
* **Constraints**: You can add constraints such as PRIMARY KEY, NOT NULL, UNIQUE, CHECK, and DEFAULT to enforce rules on the table data.
* **DESC Command**: Use DESC table\_name; to display the structure of the created table.
* **Avoiding Errors**: If the table already exists, MySQL will throw an error. To avoid this, you can use CREATE TABLE IF NOT EXISTS syntax.

### Example: Using CREATE TABLE IF NOT EXISTS

CREATE TABLE IF NOT EXISTS Employee (

EmployeeID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

Department VARCHAR(50),

Salary DECIMAL(10, 2)

);

This ensures that the Employee table is created only if it does not already exist, avoiding errors if the table is already present.

The CREATE TABLE statement is fundamental in SQL for defining and creating new tables in a database. Properly understanding and utilizing this statement is essential for effective database management and design.

## DROP TABLE:

The DROP TABLE statement in SQL is used to delete a table from a database. This operation removes the table's data, structure, constraints, and permissions permanently. It is important to use this command with caution because it is irreversible.

### SQL DROP TABLE Syntax

The syntax to drop a table is:

sql

Copy code

DROP TABLE table\_name;

This syntax is consistent across different SQL databases like Oracle, SQL Server, and MySQL.

### Examples

#### Example 1: Creating and Dropping a Table

First, we will create a database and a table, insert some data, and then drop the table.

sql

Copy code

-- Create a new database

CREATE DATABASE Newcafe;

USE Newcafe;

-- Create a table named 'categories'

CREATE TABLE categories (

CategoryID INT NOT NULL PRIMARY KEY,

CategoryName NVARCHAR(50) NOT NULL,

ItemDescription NVARCHAR(50) NOT NULL

);

-- Insert data into the 'categories' table

INSERT INTO categories (CategoryID, CategoryName, ItemDescription)

VALUES

(1, 'Beverages', 'Soft Drink'),

(2, 'Condiments', 'Sweet and Savoury sauces'),

(3, 'Confections', 'Sweet bread');

-- Select all data from the 'categories' table

SELECT \* FROM categories;

The resulting table categories will have the following data:

| **CategoryID** | **CategoryName** | **ItemDescription** |
| --- | --- | --- |
| 1 | Beverages | Soft Drink |
| 2 | Condiments | Sweet and Savoury sauces |
| 3 | Confections | Sweet bread |

#### Example 2: Dropping the Table

To drop the categories table, use the following query:

sql

Copy code

DROP TABLE categories;

Executing this statement will remove the categories table along with all its data and structure.

### Important Points About SQL DROP TABLE

* **Permanent Deletion**: The DROP TABLE statement deletes the table and all associated data, indexes, triggers, constraints, and permissions permanently.
* **Irreversible**: Once executed, the action cannot be undone.
* **Error Prevention**: To avoid errors when trying to drop a table that does not exist, use DROP TABLE IF EXISTS.

#### Example: Using DROP TABLE IF EXISTS

sql

Copy code

DROP TABLE IF EXISTS categories;

* **Partitioned Tables**: When dropping a partitioned table, the DROP TABLE statement removes the table definition, all partitions, data stored in those partitions, and all partition definitions.
* **Temporary Tables**: The DROP TABLE statement can also be used to drop temporary tables by including the TEMPORARY keyword.

#### Example: Dropping a Temporary Table

sql

Copy code

DROP TEMPORARY TABLE temp\_table\_name;

* **Verification**: To verify if a table has been dropped, you can use the DESC command. If the table is dropped, the DESC command will return an error indicating that the table does not exist.

### Summary

The DROP TABLE statement is an essential SQL command for database management, allowing you to permanently remove tables and all associated data and structure. Always ensure that the table is no longer needed before executing this command, as it is irreversible and will permanently delete the table and its contents.

SQL DELETE STATEMENT:

The SQL DELETE statement is a crucial operation used to remove data from a database table. It can selectively delete records based on specific conditions, helping manage database size, maintain data accuracy, and ensure integrity. Here's a detailed look at the DELETE statement, including its syntax, examples, and important points.

### SQL DELETE Syntax

The basic syntax for the DELETE statement is:

DELETE FROM table\_name

WHERE some\_condition;

* **table\_name**: The name of the table from which you want to delete records.
* **some\_condition**: The condition that specifies which records to delete. If the WHERE clause is omitted, all records in the table will be deleted.

### Examples

Let's assume we have a table named GFG\_Employees containing personal details of employees.

#### Creating the Sample Table and Inserting Data

CREATE TABLE GFG\_Employees (

id INT PRIMARY KEY,

name VARCHAR(20),

email VARCHAR(25),

department VARCHAR(20)

);

INSERT INTO GFG\_Employees (id, name, email, department) VALUES

(1, 'Jessie', 'jessie23@gmail.com', 'Development'),

(2, 'Praveen', 'praveen\_dagger@yahoo.com', 'HR'),

(3, 'Bisa', 'dragonBall@gmail.com', 'Sales'),

(4, 'Rithvik', 'msvv@hotmail.com', 'IT'),

(5, 'Suraj', 'srjsunny@gmail.com', 'Quality Assurance'),

(6, 'Om', 'OmShukla@yahoo.com', 'IT'),

(7, 'Naruto', 'uzumaki@konoha.com', 'Development');

SELECT \* FROM GFG\_Employees;

The resulting table GFG\_Employees will have the following data:

| **id** | **name** | **email** | **department** |
| --- | --- | --- | --- |
| 1 | Jessie | jessie23@gmail.com | Development |
| 2 | Praveen | [praveen\_dagger@yahoo.com](mailto:praveen_dagger@yahoo.com) | HR |
| 3 | Bisa | dragonBall@gmail.com | Sales |
| 4 | Rithvik | msvv@hotmail.com | IT |
| 5 | Suraj | srjsunny@gmail.com | Quality Assurance |
| 6 | Om | [OmShukla@yahoo.com](mailto:OmShukla@yahoo.com) | IT |
| 7 | Naruto | uzumaki@konoha.com | Development |

#### Deleting a Single Record

To delete the record where the name is 'Rithvik':

sql

Copy code

DELETE FROM GFG\_Employees WHERE name = 'Rithvik';

This will remove the record for 'Rithvik' from the GFG\_Employees table.

#### Deleting Multiple Records

To delete records where the department is 'Development':

DELETE FROM GFG\_Employees WHERE department = 'Development';

This will delete the records for 'Jessie' and 'Naruto'.

#### Deleting All Records

To remove all records from the table, you can use one of the following queries:

DELETE FROM GFG\_Employees;

-- Or

DELETE \* FROM GFG\_Employees;

After executing either of these queries, the GFG\_Employees table will be empty.

### Important Points About SQL DELETE

* **Conditional Deletion**: The DELETE statement can be used to delete single or multiple records based on the condition specified in the WHERE clause.
* **Deleting All Records**: If the WHERE clause is omitted, all records in the table will be deleted, but the table structure will remain intact.
* **Reversible Operation**: DELETE is a DML (Data Manipulation Language) command, meaning the operation can be rolled back if not yet committed.
* **Careful Use**: Use the DELETE statement with caution, especially without a WHERE clause, to avoid unintentionally removing all records.
* **Data Integrity**: Using the DELETE statement helps maintain data integrity by removing outdated or unnecessary records.

### Conclusion

The SQL DELETE statement is an essential tool for database management, allowing selective or complete removal of records from a table. Proper use of conditions in the WHERE clause ensures precise control over which records are deleted, aiding in data maintenance and inte

RENAMING TABLE,COLUMN AND ADDING NEW COL:

The ALTER TABLE statement in SQL is used to modify the structure of an existing table. This can include renaming the table, renaming columns, adding new columns, or changing column data types. The syntax can vary slightly between different SQL databases like MySQL, MariaDB, and Oracle.

# Renaming a Table

To rename a table, you use the following syntax:

sql

Copy code

ALTER TABLE table\_name

RENAME TO new\_table\_name;

### Renaming a Column

#### MySQL and Oracle

sql

Copy code

ALTER TABLE table\_name

RENAME COLUMN old\_name TO new\_name;

#### MariaDB

sql

Copy code

ALTER TABLE table\_name

CHANGE COLUMN old\_name new\_name datatype;

### Example: Creating and Modifying a Table

Let's create a table named Student, insert some data, and perform various ALTER TABLE operations.

#### Creating the Student Table

sql

Copy code

CREATE TABLE Student (

id INT PRIMARY KEY,

name VARCHAR(50),

age INT,

email VARCHAR(50),

phone VARCHAR(20)

);

-- Inserting data into the Student table

INSERT INTO Student (id, name, age, email, phone)

VALUES

(1, 'Amit', 20, 'amit@gmail.com', '9999999999'),

(2, 'Rahul', 22, 'rahul@yahoo.com', '8888888888'),

(3, 'Priya', 21, 'priya@hotmail.com', '7777777777'),

(4, 'Sonia', 23, 'sonia@gmail.com', '6666666666'),

(5, 'Kiran', 19, 'kiran@yahoo.com', '5555555555');

-- Select all data from the Student table

SELECT \* FROM Student;

#### Example 1: Renaming a Column

To change the column name name to FIRST\_NAME in the Student table:

##### MySQL and Oracle

sql

Copy code

ALTER TABLE Student

RENAME COLUMN name TO FIRST\_NAME;

##### MariaDB

ALTER TABLE Student

CHANGE COLUMN name FIRST\_NAME VARCHAR(50);

#### Example 2: Renaming the Table

To change the table name from Student to Student\_Details:

ALTER TABLE Student

RENAME TO Student\_Details;

#### Example 3: Adding a New Column

To add a new column marks to the Student table:

ALTER TABLE Student

ADD marks INT;

After adding the new column, the table structure will be:

| **id** | **FIRST\_NAME** | **age** | **email** | **phone** | **marks** |
| --- | --- | --- | --- | --- | --- |
| 1 | Amit | 20 | amit@gmail.com | 9999999999 | NULL |
| 2 | Rahul | 22 | [rahul@yahoo.com](mailto:rahul@yahoo.com) | 8888888888 | NULL |
| 3 | Priya | 21 | priya@hotmail.com | 7777777777 | NULL |
| 4 | Sonia | 23 | sonia@gmail.com | 6666666666 | NULL |
| 5 | Kiran | 19 | [kiran@yahoo.com](mailto:kiran@yahoo.com) | 5555555555 | NULL |

### Important Notes

* **Syntax Variations**: Always check the specific syntax for the SQL database you are using as it might vary slightly between different systems.
* **Changes to Table Structure**: The ALTER TABLE command allows for various modifications like adding, dropping, and renaming columns, and changing data types.
* **Impact on Data**: Be cautious when modifying table structures as these changes can affect existing data and applications relying on that data.

### Conclusion

The ALTER TABLE statement is a powerful tool for modifying the structure of an existing table in a database. Whether you need to rename a table, rename a column, or add new columns, understanding the appropriate syntax and implications of these changes is crucial for effective database management.

# DROP AND TRUNCATE:

The SQL DROP and TRUNCATE commands are both used to remove data from a table, but they operate differently and have different implications for the database schema. Here's a detailed explanation of each command, their syntax, and key differences:

## DROP Command

### Overview

The DROP command is used to delete an entire database or table, including its structure and data. This operation is irreversible, meaning that once a table or database is dropped, it cannot be recovered.

### Syntax

sql

Copy code

DROP object object\_name;

* object: The type of object to be dropped (e.g., TABLE, DATABASE).
* object\_name: The name of the object to be dropped.

### Examples

#### Drop a Table

sql

Copy code

DROP TABLE table\_name;

* table\_name: The name of the table to be deleted.

#### Drop a Database

sql

Copy code

DROP DATABASE database\_name;

* database\_name: The name of the database to be deleted.

### Example Queries

sql

Copy code

-- Drop a table named student\_details

DROP TABLE student\_details;

-- Drop a database named student\_data

DROP DATABASE student\_data;

## TRUNCATE Command

### Overview

The TRUNCATE command is used to remove all rows from a table, but it preserves the table structure for future use. This operation is also irreversible, but it does not affect the table schema, indexes, or constraints.

### Syntax

sql

Copy code

TRUNCATE TABLE table\_name;

* table\_name: The name of the table to be truncated.

### Example Query

sql

Copy code

-- Truncate a table named student\_details

TRUNCATE TABLE student\_details;

## Differences Between DROP and TRUNCATE

| **Feature** | **DROP** | **TRUNCATE** |
| --- | --- | --- |
| **Operation Type** | Removes the entire table or database | Removes all rows but preserves the table |
| **Data and Structure** | Deletes both data and structure | Deletes only data, keeps structure intact |
| **Integrity Constraints** | Removes integrity constraints and indexes | Does not remove integrity constraints |
| **Views** | Removes associated views | Does not affect views |
| **Space Reclamation** | Frees up table space | Does not free table space |
| **Speed** | Slower due to more operations | Faster as it bypasses many checks |
| **Rollback** | Permanent and cannot be rolled back | Also permanent and cannot be rolled back |

## Example Use Cases

### DROP Table Example

sql

Copy code

-- Drop a table named student\_details

DROP TABLE student\_details;

**Output**: The student\_details table will be permanently removed from the database.

### DROP Database Example

sql

Copy code

-- Drop a database named student\_data

DROP DATABASE student\_data;

**Output**: The student\_data database will be permanently removed along with all its tables and data.

### TRUNCATE Table Example

sql

Copy code

-- Truncate a table named student\_details

TRUNCATE TABLE student\_details;

**Output**: All rows in the student\_details table will be removed, but the table structure will remain for future use.

## Important Points

### SQL DROP Statement

* Completely removes a table or database from the database.
* Is a permanent operation and cannot be rolled back.
* Removes integrity constraints and indexes associated with the table.
* Slower compared to the TRUNCATE statement due to more operations.

### SQL TRUNCATE Statement

* Removes all rows from a table but preserves the table structure and columns.
* Is faster compared to the DROP statement.
* Resets the identity column (if any) back to its seed value.
* Does not remove integrity constraints associated with the table.

By understanding the differences between DROP and TRUNCATE, you can choose the appropriate command for your database management needs.

# Sql copy table:

Copying data from one table to another in SQL can be done using the SELECT INTO statement. This statement allows you to create a new table and insert the data from an existing table into it. Below are various examples demonstrating how to use the SELECT INTO statement to copy tables in SQL.

## Syntax of SELECT INTO Statement

SELECT \* INTO New\_table\_name FROM old\_table\_name;

### Example 1: Copying All Columns

Consider a table Cars with the following structure:

| **Car Name** | **Car Color** | **Car Cost** |
| --- | --- | --- |
| Hyundai Creta | White | 1085000 |
| Hyundai Venue | White | 950000 |
| Hyundai i20 | Red | 900000 |
| Kia Sonet | White | 1000000 |
| Kia Seltos | Black | 800000 |
| Swift Dezire | Red | 795000 |

To copy all columns and data from Cars to a new table Car\_Details:

SELECT \* INTO Car\_Details FROM Cars;

To verify:

SELECT \* FROM Car\_Details;

### Example 2: Copying Specific Columns

Consider the same Cars table. To copy only the Car\_Name and Car\_Color columns to a new table Car\_Color:

sql

Copy code

SELECT Car\_Name, Car\_Color INTO Car\_Color FROM Cars;

To verify:

SELECT \* FROM Car\_Color;

### Example 3: Copying with a Condition

If you want to copy only specific rows based on a condition, such as cars with Car\_Color as 'Black':

SELECT \* INTO Black\_Car\_Details FROM Cars WHERE Car\_Color = 'Black';

To verify:

SELECT \* FROM Black\_Car\_Details;

### Example 4: Copying Data from Employee Table

Consider an Employee table:

| **Emp\_Id** | **Emp\_Name** | **Emp\_Salary** | **Emp\_City** |
| --- | --- | --- | --- |
| 201 | Abhay | 45000 | Goa |
| 202 | Ankit | 45000 | Delhi |
| 203 | Bheem | 38000 | Goa |
| 204 | Ram | 49000 | Goa |
| 205 | Sumit | 40000 | Delhi |

To copy all data to a new table Coding\_Employees:

SELECT \* INTO Coding\_Employees FROM Employee;

To verify:

SELECT \* FROM Coding\_Employees;

To copy only employees with a salary greater than 40,000 to a new table Emp\_Salary\_40000:

SELECT \* INTO Emp\_Salary\_40000 FROM Employee WHERE Emp\_Salary > 40000;

To verify:

SELECT \* FROM Emp\_Salary\_40000;

### Example 5: Creating a Backup Without Data

To create a table with the same structure but without any data, use a WHERE clause that always evaluates to false, such as WHERE 1=2:

SELECT \* INTO Empty\_Car\_Details FROM Cars WHERE 1=2;

To verify:

SELECT \* FROM Empty\_Car\_Details;

## Key Points

* **Structure Copy:** The new table will have the same structure as the source table.
* **Data Copy:** All or specific data can be copied based on the query.
* **Conditions:** Use WHERE clause to filter which data to copy.
* **Automatic Creation:** The SELECT INTO statement creates the new table automatically.
* **Database Scope:** The new table is created in the same database unless specified otherwise.

Another way to copy sql table:

Creating backups or duplicates of tables in SQL is a crucial operation for maintaining data integrity and security, especially when performing operations that might modify or delete existing data. Here, we'll discuss different ways to copy, duplicate, or create backups of tables in SQL using common SQL commands.

## Syntax Overview

To create a copy or backup of a table, the basic syntax is:

CREATE TABLE new\_table AS SELECT \* FROM existing\_table;

## Examples

### Example 1: Backup Table with All Columns and Data

In this example, we'll create a backup table called stud\_1 that is a duplicate of the student\_information table, including all its data.

**Query:**

CREATE TABLE stud\_1 AS SELECT \* FROM student\_information;

SELECT \* FROM stud\_1;

**Output:**

| **ID** | **Age** | **Student Name** | **Sex** |
| --- | --- | --- | --- |
| 1 | 22 | Harry | Male |
| 2 | 23 | Vishal | Male |
| 3 | 20 | Snehal | Female |
| 4 | 25 | Ram | Male |
| 5 | 24 | Hina | Female |

### Example 2: Backup Table with Specific Columns and Data

In this example, we'll create a backup table called stud\_2 that includes only specific columns from the student\_information table.

**Query:**

sql

Copy code

CREATE TABLE stud\_2 AS

SELECT id, student\_name FROM student\_information;

SELECT \* FROM stud\_2;

**Output:**

| **ID** | **Student Name** |
| --- | --- |
| 1 | Harry |
| 2 | Vishal |
| 3 | Snehal |
| 4 | Ram |
| 5 | Hina |

### Example 3: Backup Table Without Data

To create a backup table without copying the data, we can use a condition in the WHERE clause that always returns false (e.g., WHERE 1=2).

**Query:**

sql

Copy code

CREATE TABLE geeks\_student AS

SELECT \* FROM student\_information WHERE 1=2;

SELECT \* FROM geeks\_student;

**Output:**

| **ID** | **Age** | **Student Name** | **Sex** |
| --- | --- | --- | --- |
|  |  |  |  |

### Example 4: Backup Table with Specific Columns and No Data

In this example, we'll create a backup table called geek\_student that includes only specific columns from the student\_information table but without copying any data.

**Query:**

CREATE TABLE geek\_student AS

SELECT id, student\_name FROM student\_information WHERE 1=2;

SELECT \* FROM geek\_student;

**Output:**

| **ID** | **Student Name** |
| --- | --- |
|  |  |

## Important Notes

* **Structure Preservation:** When creating a backup table without data, the structure (schema) of the original table is preserved.
* **Column Selection:** You can choose specific columns to be included in the backup table by specifying them in the SELECT statement.
* **Efficiency:** Creating a backup table with no data using WHERE 1=2 is efficient because it does not involve copying any rows, only the structure.
* **DDL Operations:** The CREATE TABLE AS SELECT (CTAS) statement is a Data Definition Language (DDL) operation and can be used to create new tables based on the results of a SELECT query.

By using these techniques, you can ensure that you have a backup of your tables, which can be helpful for recovery, testing, and tracking changes.

# SQL TEMP TABLES:

Temporary tables in SQL are used to store and process intermediate results. They are similar to permanent tables but are automatically deleted when the session that created them is closed. They can be extremely useful when you need to handle temporary data during the execution of complex queries or procedures. There are two types of temporary tables: Local Temporary Tables and Global Temporary Tables.

## Local Temporary Tables

### Characteristics:

* **Scope:** Available only within the session that created them.
* **Lifetime:** Automatically dropped when the session ends.
* **Naming:** Prefixed with a single #.

### Syntax:

**Creating a Local Temporary Table:**

CREATE TABLE #EmpDetails (id INT, name VARCHAR(25));

**Inserting Data into a Local Temporary Table:**

INSERT INTO #EmpDetails VALUES (01, 'Lalit'), (02, 'Atharva');

**Selecting Data from a Local Temporary Table:**

SELECT \* FROM #EmpDetails;

**Dropping a Local Temporary Table:**

DROP TABLE #EmpDetails;

### Example within a Stored Procedure:

CREATE PROCEDURE ProcTemp

AS

BEGIN

CREATE TABLE #EmpDetails (id INT, name VARCHAR(25));

INSERT INTO #EmpDetails VALUES (01, 'Lalit'), (02, 'Atharva');

SELECT \* FROM #EmpDetails;

END;

**Executing the Procedure:**

EXECUTE ProcTemp;

## Global Temporary Tables

### Characteristics:

* **Scope:** Visible to all sessions.
* **Lifetime:** Dropped when the last session referencing the table is closed.
* **Naming:** Prefixed with a double ##.

### Syntax:

**Creating a Global Temporary Table:**

CREATE TABLE ##EmpDetails (id INT, name VARCHAR(25));

**Inserting Data into a Global Temporary Table:**

INSERT INTO ##EmpDetails VALUES (01, 'Lalit'), (02, 'Atharva');

**Selecting Data from a Global Temporary Table:**

SELECT \* FROM ##EmpDetails;

**Dropping a Global Temporary Table:**

DROP TABLE ##EmpDetails;

### Example:

CREATE TABLE ##EmpDetails (id INT, name VARCHAR(25));

INSERT INTO ##EmpDetails VALUES (01, 'Lalit'), (02, 'Atharva');

SELECT \* FROM ##EmpDetails;

## Use Cases

* **Intermediate Storage:** Useful for holding intermediate results during complex data manipulation operations.
* **Session-Specific Data:** Ideal for storing session-specific data that does not need to persist beyond the session's life.
* **Batch Processing:** Helpful in batch processing where temporary results are used and discarded after use.

## Important Points

* **Automatic Deletion:** Temporary tables are automatically deleted when the session ends (for local temp tables) or when the last session using them ends (for global temp tables).
* **Indexing:** You can create indexes on temporary tables.
* **Performance:** Temporary tables can improve performance by breaking down complex queries into simpler steps.

By using temporary tables effectively, you can manage and manipulate temporary data in your SQL operations, ensuring efficient and effective data processing.

# ALTER:

The ALTER TABLE statement in SQL is used to modify the structure of an existing table. This can include adding, modifying, or deleting columns, as well as changing table constraints and renaming the table itself.

### ALTER TABLE ADD Column

**Adding a Single Column:**

sql

Copy code

ALTER TABLE table\_name

ADD column\_name column\_definition;

Example:

sql

Copy code

ALTER TABLE Cars

ADD Car\_Model VARCHAR(20);

**Adding Multiple Columns:**

sql

Copy code

ALTER TABLE table\_name

ADD (column\_name1 column\_definition, column\_name2 column\_definition, ...);

Example:

sql

Copy code

ALTER TABLE Employee

ADD (Emp\_ContactNo NUMBER(13), Emp\_EmailID VARCHAR(50));

### ALTER TABLE MODIFY Column

**Modifying a Single Column:**

sql

Copy code

ALTER TABLE table\_name

MODIFY column\_name column\_definition;

Example:

sql

Copy code

ALTER TABLE Cars

MODIFY Car\_Color VARCHAR(50);

**Modifying Multiple Columns:**

sql

Copy code

ALTER TABLE table\_name

MODIFY (column\_name1 column\_definition, column\_name2 column\_definition, ...);

Example:

sql

Copy code

ALTER TABLE Employee

MODIFY (Emp\_ContactNo INT, Emp\_EmailID VARCHAR(80));

### ALTER TABLE DROP Column

**Dropping a Column:**

sql

Copy code

ALTER TABLE table\_name

DROP COLUMN column\_name;

Example:

sql

Copy code

ALTER TABLE Cars

DROP COLUMN Car\_Color;

**Dropping Multiple Columns:**

sql

Copy code

-- You must run separate ALTER TABLE statements for each column.

ALTER TABLE Employee

DROP COLUMN Emp\_Salary;

ALTER TABLE Employee

DROP COLUMN Emp\_City;

### ALTER TABLE RENAME Column

**Renaming a Column:**

sql

Copy code

ALTER TABLE table\_name

RENAME COLUMN old\_name TO new\_name;

Example:

sql

Copy code

ALTER TABLE Cars

RENAME COLUMN Car\_Color TO Colors;

sql

Copy code

ALTER TABLE Employee

RENAME COLUMN Emp\_City TO Emp\_Address;

### Real-World Examples:

#### 1. Adding a Column for Product Information

A company wants to add a new column Release\_Date to the Products table to keep track of when each product was released.

ALTER TABLE Products

ADD Release\_Date DATE;

#### 2. Modifying a Column Data Type for Customer Emails

A retail company needs to increase the length of the Email column in the Customers table to store longer email addresses.

ALTER TABLE Customers

MODIFY Email VARCHAR(100);

#### 3. Dropping an Unused Column from an Orders Table

A company decides to remove the Discount\_Code column from the Orders table as it is no longer used.

ALTER TABLE Orders

DROP COLUMN Discount\_Code;

#### 4. Renaming a Column for Better Clarity in Employee Records

To make the column names more descriptive, an organization wants to rename the DOB column to Date\_Of\_Birth in the Employees table.

ALTER TABLE Employees

RENAME COLUMN DOB TO Date\_Of\_Birth;

These examples demonstrate how the ALTER TABLE statement can be used to manage and update the structure of database tables in SQL, allowing for flexibility and maintenance of the database schema.

# CLAUSES

SQL WHERE \

The SQL WHERE clause is used to filter records in various SQL statements such as SELECT, UPDATE, and DELETE. It specifies the conditions that the data must meet to be included in the results or affected by the statement. Here’s a detailed explanation with examples:

### Basic Syntax

#### SELECT Statement

sql

Copy code

SELECT column1, column2, ..., columnN

FROM table\_name

WHERE conditions;

#### UPDATE Statement

sql

Copy code

UPDATE table\_name

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

WHERE conditions;

#### DELETE Statement

sql

Copy code

DELETE FROM table\_name

WHERE conditions;

### Conditional Operators

* =: Equal to
* >: Greater than
* <: Less than
* >=: Greater than or equal to
* <=: Less than or equal to
* <>: Not equal to (can also use != in some SQL dialects)

### Examples

#### SELECT with WHERE

Fetches records from the employees table where the age is greater than 30.

sql

Copy code

SELECT first\_name, last\_name, age

FROM employees

WHERE age > 30;

#### UPDATE with WHERE

Updates the salary of employees in the employees table where the department is 'HR'.

sql

Copy code

UPDATE employees

SET salary = salary \* 1.10

WHERE department = 'HR';

#### DELETE with WHERE

Deletes records from the employees table where the status is 'inactive'.

sql

Copy code

DELETE FROM employees

WHERE status = 'inactive';

### Using Multiple Conditions

You can combine multiple conditions using logical operators like AND and OR.

#### AND Operator

sql

Copy code

SELECT first\_name, last\_name

FROM employees

WHERE age > 30 AND department = 'Sales';

This query selects employees who are older than 30 and work in the Sales department.

#### OR Operator

sql

Copy code

SELECT first\_name, last\_name

FROM employees

WHERE age > 30 OR department = 'Sales';

This query selects employees who are either older than 30 or work in the Sales department.

### Using Wildcards with LIKE

The LIKE operator is used for pattern matching.

sql

Copy code

SELECT first\_name, last\_name

FROM employees

WHERE first\_name LIKE 'J%';

This query selects employees whose first names start with 'J'.

### Using IN

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

sql

Copy code

SELECT first\_name, last\_name

FROM employees

WHERE department IN ('HR', 'Sales', 'IT');

This query selects employees who work in HR, Sales, or IT departments.

### Using BETWEEN

The BETWEEN operator selects values within a given range.

sql

Copy code

SELECT first\_name, last\_name

FROM employees

WHERE age BETWEEN 30 AND 40;

This query selects employees whose age is between 30 and 40 inclusive.

By utilizing the WHERE clause effectively, you can precisely control which records are retrieved or affected by your SQL statements, making your data manipulation more efficient and targeted.

SQL AND

### SQL AND Condition

The SQL AND condition is used to combine multiple conditions in an SQL query. It ensures that all the specified conditions are met for a row to be included in the results or affected by the statement. The AND condition can be used with SELECT, INSERT, UPDATE, and DELETE statements.

### Basic Syntax

#### SELECT Statement

sql

Copy code

SELECT columns

FROM table\_name

WHERE condition1 AND condition2;

### Examples

#### Example Data

Consider the following employees table:

| **ID** | **First\_Name** | **Last\_Name** | **Department** | **Location** |
| --- | --- | --- | --- | --- |
| 1 | Harshad | Kuwar | Marketing | Pune |
| 2 | Anurag | Rajput | IT | Mumbai |
| 3 | Chaitali | Tarle | IT | Chennai |
| 4 | Pranjal | Patil | IT | Chennai |
| 5 | Suraj | Tripathi | Marketing | Pune |
| 6 | Roshni | Jadhav | Finance | Bangalore |
| 7 | Sandhya | Jain | Finance | Bangalore |

### SQL AND Condition with SELECT

#### Example 1: Retrieve employees in the IT department located in Chennai

sql

Copy code

SELECT \* FROM employees

WHERE Department = 'IT' AND Location = 'Chennai';

| **ID** | **First\_Name** | **Last\_Name** | **Department** | **Location** |
| --- | --- | --- | --- | --- |
| 3 | Chaitali | Tarle | IT | Chennai |
| 4 | Pranjal | Patil | IT | Chennai |

#### Example 2: Retrieve employees in the IT department located in Mumbai

sql

Copy code

SELECT \* FROM employees

WHERE Department = 'IT' AND Location = 'Mumbai';

| **ID** | **First\_Name** | **Last\_Name** | **Department** | **Location** |
| --- | --- | --- | --- | --- |
| 2 | Anurag | Rajput | IT | Mumbai |

### SQL AND Condition with UPDATE

#### Example 1: Update the location of an employee in Marketing whose first name is Suraj to Delhi

sql

Copy code

UPDATE employees

SET Location = 'Delhi'

WHERE Department = 'Marketing' AND First\_Name = 'Suraj';

Verify the update:

sql

Copy code

SELECT \* FROM employees;

| **ID** | **First\_Name** | **Last\_Name** | **Department** | **Location** |
| --- | --- | --- | --- | --- |
| 1 | Harshad | Kuwar | Marketing | Pune |
| 2 | Anurag | Rajput | IT | Mumbai |
| 3 | Chaitali | Tarle | IT | Chennai |
| 4 | Pranjal | Patil | IT | Chennai |
| 5 | Suraj | Tripathi | Marketing | Delhi |
| 6 | Roshni | Jadhav | Finance | Bangalore |
| 7 | Sandhya | Jain | Finance | Bangalore |

#### Example 2: Update the department of an employee with ID 7 in Finance to HR

sql

Copy code

UPDATE employees

SET Department = 'HR'

WHERE Department = 'Finance' AND ID = 7;

Verify the update:

sql

Copy code

SELECT \* FROM employees;

| **ID** | **First\_Name** | **Last\_Name** | **Department** | **Location** |
| --- | --- | --- | --- | --- |
| 1 | Harshad | Kuwar | Marketing | Pune |
| 2 | Anurag | Rajput | IT | Mumbai |
| 3 | Chaitali | Tarle | IT | Chennai |
| 4 | Pranjal | Patil | IT | Chennai |
| 5 | Suraj | Tripathi | Marketing | Delhi |
| 6 | Roshni | Jadhav | Finance | Bangalore |
| 7 | Sandhya | Jain | HR | Bangalore |

### SQL AND Condition with DELETE

#### Example 1: Delete the record where the last name is Jain and the location is Bangalore

sql

Copy code

DELETE FROM employees

WHERE Last\_Name = 'Jain' AND Location = 'Bangalore';

Verify the deletion:

sql

Copy code

SELECT \* FROM employees;

| **ID** | **First\_Name** | **Last\_Name** | **Department** | **Location** |
| --- | --- | --- | --- | --- |
| 1 | Harshad | Kuwar | Marketing | Pune |
| 2 | Anurag | Rajput | IT | Mumbai |
| 3 | Chaitali | Tarle | IT | Chennai |
| 4 | Pranjal | Patil | IT | Chennai |
| 5 | Suraj | Tripathi | Marketing | Delhi |
| 6 | Roshni | Jadhav | Finance | Bangalore |

#### Example 2: Delete the record where the department is IT and the location is Mumbai

sql

Copy code

DELETE FROM employees

WHERE Department = 'IT' AND Location = 'Mumbai';

Verify the deletion:

sql

Copy code

SELECT \* FROM employees;

| **ID** | **First\_Name** | **Last\_Name** | **Department** | **Location** |
| --- | --- | --- | --- | --- |
| 1 | Harshad | Kuwar | Marketing | Pune |
| 3 | Chaitali | Tarle | IT | Chennai |
| 4 | Pranjal | Patil | IT | Chennai |
| 5 | Suraj | Tripathi | Marketing | Delhi |
| 6 | Roshni | Jadhav | Finance | Bangalore |

By using the AND condition in your SQL queries, you can filter data more precisely based on multiple criteria.

SQL OR

### SQL OR Condition

The SQL OR condition is used to create a SQL statement where records are returned when any one of the specified conditions is met. It can be used in a SELECT, INSERT, UPDATE, or DELETE statement.

### Basic Syntax

#### SELECT Statement

sql

Copy code

SELECT columns

FROM table\_name

WHERE condition1 OR condition2;

### Examples

#### Example Data

Consider the following employees table:

| **ID** | **First\_Name** | **Last\_Name** | **Department** | **Location** |
| --- | --- | --- | --- | --- |
| 1 | Harshad | Kuwar | Marketing | Pune |
| 2 | Anurag | Rajput | IT | Mumbai |
| 3 | Chaitali | Tarle | IT | Chennai |
| 4 | Pranjal | Patil | IT | Chennai |
| 5 | Suraj | Tripathi | Marketing | Pune |
| 6 | Roshni | Jadhav | Finance | Bangalore |
| 7 | Sandhya | Jain | Finance | Bangalore |

### SQL OR Condition with SELECT

#### Example 1: Retrieve employees in the IT department or located in Chennai

sql

Copy code

SELECT \* FROM employees

WHERE Department = 'IT' OR Location = 'Chennai';

| **ID** | **First\_Name** | **Last\_Name** | **Department** | **Location** |
| --- | --- | --- | --- | --- |
| 2 | Anurag | Rajput | IT | Mumbai |
| 3 | Chaitali | Tarle | IT | Chennai |
| 4 | Pranjal | Patil | IT | Chennai |

#### Example 2: Retrieve employees in the Marketing department or located in Noida

sql

Copy code

SELECT \* FROM employees

WHERE Department = 'Marketing' OR Location = 'Noida';

| **ID** | **First\_Name** | **Last\_Name** | **Department** | **Location** |
| --- | --- | --- | --- | --- |
| 1 | Harshad | Kuwar | Marketing | Pune |
| 5 | Suraj | Tripathi | Marketing | Pune |
| 7 | Sandhya | Jain | Finance | Bangalore |

### SQL OR Condition with UPDATE

#### Example 1: Update the location of employees in Marketing or whose last name is Tarle to Delhi

sql

Copy code

UPDATE employees

SET Location = 'Delhi'

WHERE Department = 'Marketing' OR Last\_Name = 'Tarle';

Verify the update:

sql

Copy code

SELECT \* FROM employees;

| **ID** | **First\_Name** | **Last\_Name** | **Department** | **Location** |
| --- | --- | --- | --- | --- |
| 1 | Harshad | Kuwar | Marketing | Delhi |
| 2 | Anurag | Rajput | IT | Mumbai |
| 3 | Chaitali | Tarle | IT | Delhi |
| 4 | Pranjal | Patil | IT | Chennai |
| 5 | Suraj | Tripathi | Marketing | Delhi |
| 6 | Roshni | Jadhav | Finance | Bangalore |
| 7 | Sandhya | Jain | Finance | Bangalore |

#### Example 2: Update the department of employees in Finance or whose first name is Sandhya to HR

sql

Copy code

UPDATE employees

SET Department = 'HR'

WHERE Department = 'Finance' OR First\_Name = 'Sandhya';

Verify the update:

sql

Copy code

SELECT \* FROM employees;

| **ID** | **First\_Name** | **Last\_Name** | **Department** | **Location** |
| --- | --- | --- | --- | --- |
| 1 | Harshad | Kuwar | Marketing | Delhi |
| 2 | Anurag | Rajput | IT | Mumbai |
| 3 | Chaitali | Tarle | IT | Delhi |
| 4 | Pranjal | Patil | IT | Chennai |
| 5 | Suraj | Tripathi | Marketing | Delhi |
| 6 | Roshni | Jadhav | HR | Bangalore |
| 7 | Sandhya | Jain | HR | Bangalore |

### SQL OR Condition with DELETE

#### Example 1: Delete records where the last name is Jain or location is Bangalore

sql

Copy code

DELETE FROM employees

WHERE Last\_Name = 'Jain' OR Location = 'Bangalore';

Verify the deletion:

sql

Copy code

SELECT \* FROM employees;

| **ID** | **First\_Name** | **Last\_Name** | **Department** | **Location** |
| --- | --- | --- | --- | --- |
| 1 | Harshad | Kuwar | Marketing | Delhi |
| 2 | Anurag | Rajput | IT | Mumbai |
| 3 | Chaitali | Tarle | IT | Delhi |
| 4 | Pranjal | Patil | IT | Chennai |
| 5 | Suraj | Tripathi | Marketing | Delhi |

#### Example 2: Delete records where the department is Marketing or location is Delhi

sql

Copy code

DELETE FROM employees

WHERE Department = 'Marketing' OR Location = 'Delhi';

Verify the deletion:

sql

Copy code

SELECT \* FROM employees;

| **ID** | **First\_Name** | **Last\_Name** | **Department** | **Location** |
| --- | --- | --- | --- | --- |
| 2 | Anurag | Rajput | IT | Mumbai |
| 4 | Pranjal | Patil | IT | Chennai |

By using the OR condition in your SQL queries, you can filter data based on multiple criteria where any one of the conditions being true will include the row in the result set or affect the row in the data manipulation operation.

# SQL AS

### SQL SELECT AS Clause

The SQL AS clause is used to assign a new, temporary name (alias) to a table column or even the table itself within a query. This can make query results easier to understand and present.

### Syntax

#### Assigning an Alias to Columns

sql

Copy code

SELECT Column\_Name1 AS New\_Column\_Name1, Column\_Name2 AS New\_Column\_Name2

FROM Table\_Name;

#### Assigning an Alias to Tables

sql

Copy code

SELECT t.Column\_Name1, t.Column\_Name2

FROM Table\_Name AS t

WHERE t.Condition\_Column = some\_value;

### Examples

#### Example 1: Renaming Columns

Consider the orders table:

| **Day\_of\_order** | **Customer** | **Product** | **Quantity** |
| --- | --- | --- | --- |
| 11-09-2001 | Ajeet | Mobile | 2 |
| 13-12-2001 | Mayank | Laptop | 20 |
| 26-12-2004 | Balaswamy | Water cannon | 35 |

##### Query: Rename Day\_of\_order to Date and Customer to Client

sql

Copy code

SELECT Day\_of\_order AS Date, Customer AS Client, Product, Quantity

FROM orders;

##### Result:

| **Date** | **Client** | **Product** | **Quantity** |
| --- | --- | --- | --- |
| 11-09-2001 | Ajeet | Mobile | 2 |
| 13-12-2001 | Mayank | Laptop | 20 |
| 26-12-2004 | Balaswamy | Water cannon | 35 |

#### Example 2: Renaming Columns and Using Functions

Consider the students table:

| **Student\_RollNo** | **Student\_Name** | **Student\_Gender** | **Student\_MobileNumber** | **Student\_HomeTown** | **Student\_Age** | **Student\_Percentage** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | Rohit More | Male | 9890786123 | Lucknow | 23 | 75 |
| 2 | Kunal Shah | Male | 7789056784 | Chandigarh | 20 | 92 |
| 3 | Kartik Goenka | Male | 9908743576 | Ahmedabad | 22 | 89 |
| 4 | Anupama Shah | Female | 8890907656 | Chennai | 24 | 92 |
| 5 | Snehal Jain | Female | 8657983476 | Surat | 21 | 94 |

##### Query: Calculate average percentage and rename columns

sql

Copy code

SELECT Student\_Name AS Student, AVG(Student\_Percentage) AS Average\_Percentage

FROM students;

##### Result:

| **Student** | **Average\_Percentage** |
| --- | --- |
| Rohit More | 88.4000 |

##### Query: Get roll number and mobile number with temporary names

sql

Copy code

SELECT Student\_RollNo AS 'Roll No', Student\_MobileNumber AS 'Mobile Number'

FROM students;

##### Result:

| **Roll No** | **Mobile Number** |
| --- | --- |
| 1 | 9890786123 |
| 2 | 7789056784 |
| 3 | 9908743576 |
| 4 | 8890907656 |
| 5 | 8657983476 |

##### Query: Concatenate phone number and hometown

sql

Copy code

SELECT Student\_RollNo AS 'Roll No', CONCAT(Student\_MobileNumber, ', ', Student\_HomeTown) AS Student\_Info

FROM students;

##### Result:

| **Roll No** | **Student\_Info** |
| --- | --- |
| 1 | 9890786123, Lucknow |
| 2 | 7789056784, Chandigarh |
| 3 | 9908743576, Ahmedabad |
| 4 | 8890907656, Chennai |
| 5 | 8657983476, Surat |

#### Example 3: Assigning an Alias to a Table

##### Query: Create an alias for the students table and select specific rows

sql

Copy code

SELECT s.Student\_RollNo, s.Student\_Name, s.Student\_Gender, s.Student\_MobileNumber, s.Student\_HomeTown

FROM students AS s

WHERE s.Student\_RollNo = 3;

##### Result:

| **Student\_RollNo** | **Student\_Name** | **Student\_Gender** | **Student\_MobileNumber** | **Student\_HomeTown** |
| --- | --- | --- | --- | --- |
| 3 | Kartik Goenka | Male | 9908743576 | Ahmedabad |

### Conclusion

The SQL AS clause is a powerful tool for improving the readability and manageability of query results by allowing temporary renaming of columns and tables. This can make data easier to understand and present, without altering the underlying schema.

HAVING CLAUSE:

### SQL HAVING Clause

The HAVING clause in SQL is used to filter groups of records defined by the GROUP BY clause in a SELECT statement. It is used to place conditions on groups rather than individual rows, which makes it different from the WHERE clause. The HAVING clause is necessary when you want to apply conditions on aggregated data.

### Difference Between HAVING and WHERE Clauses

| **HAVING** | **WHERE** |
| --- | --- |
| Used to filter data after grouping. | Used to filter data before grouping. |
| Can include aggregate functions. | Cannot include aggregate functions. |
| Always used with GROUP BY. | Can be used without GROUP BY. |
| Used only with SELECT statements. | Can be used with SELECT, UPDATE, and DELETE statements. |
| Implements conditions on column operations. | Implements conditions on row operations. |
| It is a post-filter. | It is a pre-filter. |
| Filters groups of records. | Filters individual records. |

### Syntax of HAVING Clause

sql

Copy code

SELECT column1, column2, ..., aggregate\_function(column)

FROM table\_name

GROUP BY column1, column2, ...

HAVING condition;

### Examples of HAVING Clause

#### Example 1: Using SUM Aggregate Function

Consider the Employee table:

| **Emp\_Id** | **Emp\_Name** | **Emp\_Salary** | **Emp\_City** |
| --- | --- | --- | --- |
| 201 | Abhay | 2000 | Goa |
| 202 | Ankit | 4000 | Delhi |
| 203 | Bheem | 8000 | Jaipur |
| 204 | Ram | 2000 | Goa |
| 205 | Sumit | 5000 | Delhi |

**Query: Sum of salaries by city**

sql

Copy code

SELECT SUM(Emp\_Salary) AS Total\_Salary, Emp\_City

FROM Employee

GROUP BY Emp\_City;

**Result:**

| **Total\_Salary** | **Emp\_City** |
| --- | --- |
| 4000 | Goa |
| 9000 | Delhi |
| 8000 | Jaipur |

**Query: Cities with total salary greater than 5000**

sql

Copy code

SELECT SUM(Emp\_Salary) AS Total\_Salary, Emp\_City

FROM Employee

GROUP BY Emp\_City

HAVING SUM(Emp\_Salary) > 5000;

**Result:**

| **Total\_Salary** | **Emp\_City** |
| --- | --- |
| 9000 | Delhi |
| 8000 | Jaipur |

#### Example 2: Using COUNT Aggregate Function

Consider the Student\_details table:

| **Roll\_No** | **Name** | **Marks** | **Age** |
| --- | --- | --- | --- |
| 1 | Rithik | 91 | 20 |
| 2 | Kapil | 60 | 19 |
| 3 | Arun | 82 | 17 |
| 4 | Ram | 92 | 18 |
| 5 | Anuj | 50 | 20 |
| 6 | Suman | 88 | 18 |
| 7 | Sheetal | 57 | 19 |
| 8 | Anuj | 64 | 20 |

**Query: Count of students by age**

sql

Copy code

SELECT COUNT(Roll\_No) AS Student\_Count, Age

FROM Student\_details

GROUP BY Age;

**Result:**

| **Student\_Count** | **Age** |
| --- | --- |
| 3 | 20 |
| 2 | 19 |
| 1 | 17 |
| 2 | 18 |

**Query: Ages with student count greater than or equal to 2**

sql

Copy code

SELECT COUNT(Roll\_No) AS Student\_Count, Age

FROM Student\_details

GROUP BY Age

HAVING COUNT(Roll\_No) >= 2;

**Result:**

| **Student\_Count** | **Age** |
| --- | --- |
| 3 | 20 |
| 2 | 19 |
| 2 | 18 |

#### Example 3: Using MIN and MAX Aggregate Functions

Consider the Employee table:

| **Emp\_ID** | **Name** | **Emp\_Salary** | **Emp\_Dept** |
| --- | --- | --- | --- |
| 1001 | Anuj | 9000 | Finance |
| 1002 | Saket | 4000 | HR |
| 1003 | Raman | 3000 | Coding |
| 1004 | Renu | 6000 | Coding |
| 1005 | Seenu | 5000 | HR |
| 1006 | Mohan | 10000 | Marketing |
| 1007 | Anaya | 4000 | Coding |
| 1008 | Parul | 8000 | Finance |

**Query: Minimum salary by department**

sql

Copy code

SELECT MIN(Emp\_Salary) AS Min\_Salary, Emp\_Dept

FROM Employee

GROUP BY Emp\_Dept;

**Result:**

| **Min\_Salary** | **Emp\_Dept** |
| --- | --- |
| 8000 | Finance |
| 4000 | HR |
| 3000 | Coding |
| 10000 | Marketing |

**Query: Departments with minimum salary greater than 4000**

sql

Copy code

SELECT MIN(Emp\_Salary) AS Min\_Salary, Emp\_Dept

FROM Employee

GROUP BY Emp\_Dept

HAVING MIN(Emp\_Salary) > 4000;

**Result:**

| **Min\_Salary** | **Emp\_Dept** |
| --- | --- |
| 8000 | Finance |
| 10000 | Marketing |

**Query: Maximum salary by department**

sql

Copy code

SELECT MAX(Emp\_Salary) AS Max\_Salary, Emp\_Dept

FROM Employee

GROUP BY Emp\_Dept;

**Result:**

| **Max\_Salary** | **Emp\_Dept** |
| --- | --- |
| 9000 | Finance |
| 5000 | HR |
| 6000 | Coding |
| 10000 | Marketing |

**Query: Departments with maximum salary less than 8000**

sql

Copy code

SELECT MAX(Emp\_Salary) AS Max\_Salary, Emp\_Dept

FROM Employee

GROUP BY Emp\_Dept

HAVING MAX(Emp\_Salary) < 8000;

**Result:**

| **Max\_Salary** | **Emp\_Dept** |
| --- | --- |
| 5000 | HR |
| 6000 | Coding |

#### Example 4: Using AVG Aggregate Function

Consider the Employee\_Dept table:

| **Emp\_ID** | **Name** | **Emp\_Salary** | **Emp\_Dept** |
| --- | --- | --- | --- |
| 1001 | Anuj | 8000 | Finance |
| 1002 | Saket | 4000 | HR |
| 1003 | Raman | 3000 | Coding |
| 1004 | Renu | 6000 | Coding |
| 1005 | Seenu | 5000 | HR |
| 1006 | Mohan | 10000 | Marketing |
| 1007 | Anaya | 4000 | Coding |
| 1008 | Parul | 6000 | Finance |

**Query: Average salary by department**

sql

Copy code

SELECT AVG(Emp\_Salary) AS Avg\_Salary, Emp\_Dept

FROM Employee\_Dept

GROUP BY Emp\_Dept;

**Result:**

| **Avg\_Salary** | **Emp\_Dept** |
| --- | --- |
| 7000 | Finance |
| 4500 | HR |
| 6500 | Coding |
| 10000 | Marketing |

**Query: Departments with average salary greater than or equal to 6500**

sql

Copy code

SELECT AVG(Emp\_Salary) AS Avg\_Salary, Emp\_Dept

FROM Employee\_Dept

GROUP BY Emp\_Dept

HAVING AVG(Emp\_Salary) >= 6500;

**Result:**

| **Avg\_Salary** | **Emp\_Dept** |
| --- | --- |
| 7000 | Finance |
| 6500 | Coding |
| 10000 | Marketing |

### Conclusion

The HAVING clause is a powerful tool for filtering groups of records in SQL, especially when working with aggregate functions. It allows for more precise and meaningful data analysis by enabling conditions to be applied to aggregated data.

:

# SQL ORDER BY Clause

Whenever we want to sort the records based on the columns stored in the tables of the SQL database, we consider using the ORDER BY clause in SQL.

## Purpose of ORDER BY Clause

The ORDER BY clause in SQL helps us sort the records based on a specific column of a table. This means that all the values stored in the column on which we apply the ORDER BY clause will be sorted, and the corresponding column values will be displayed in the sequence obtained.

Using the ORDER BY clause, we can sort the records in ascending or descending order as per our requirement. The records will be sorted in ascending order when the ASC keyword is used with the ORDER BY clause. The DESC keyword will sort the records in descending order. If no keyword is specified after the column based on which we have to sort the records, the sorting will be done by default in ascending order.

## Syntax

**To sort records in ascending order:**

sql

Copy code

SELECT ColumnName1, ..., ColumnNameN FROM TableName ORDER BY ColumnName ASC;

**To sort records in descending order:**

sql

Copy code

SELECT ColumnName1, ..., ColumnNameN FROM TableName ORDER BY ColumnName DESC;

**To sort records in ascending order without using ASC keyword:**

sql

Copy code

SELECT ColumnName1, ..., ColumnNameN FROM TableName ORDER BY ColumnName;

## Examples

Consider we have a customers table with the following records:

| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| --- | --- | --- | --- | --- |
| 1 | Himani Gupta | 21 | Modinagar | 22000 |
| 2 | Shiva Tiwari | 22 | Bhopal | 21000 |
| 3 | Ajeet Bhargav | 45 | Meerut | 65000 |
| 4 | Ritesh Yadav | 36 | Azamgarh | 26000 |
| 5 | Balwant Singh | 45 | Varanasi | 36000 |
| 6 | Mahesh Sharma | 26 | Mathura | 22000 |
| 7 | Rohit Shrivastav | 19 | Ahemdabad | 38000 |
| 8 | Neeru Sharma | 29 | Pune | 40000 |
| 9 | Aakash Yadav | 32 | Mumbai | 43500 |
| 10 | Sahil Sheikh | 35 | Aurangabad | 68800 |

### Example 1: Sorting by Customer Names in Ascending Order

**Query:**

sql

Copy code

SELECT \* FROM customers ORDER BY Name ASC;

**Output:**

| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| --- | --- | --- | --- | --- |
| 9 | Aakash Yadav | 32 | Mumbai | 43500 |
| 3 | Ajeet Bhargav | 45 | Meerut | 65000 |
| 5 | Balwant Singh | 45 | Varanasi | 36000 |
| 1 | Himani Gupta | 21 | Modinagar | 22000 |
| 6 | Mahesh Sharma | 26 | Mathura | 22000 |
| 8 | Neeru Sharma | 29 | Pune | 40000 |
| 4 | Ritesh Yadav | 36 | Azamgarh | 26000 |
| 7 | Rohit Shrivastav | 19 | Ahemdabad | 38000 |
| 10 | Sahil Sheikh | 35 | Aurangabad | 68800 |
| 2 | Shiva Tiwari | 22 | Bhopal | 21000 |

### Example 2: Sorting by Addresses in Ascending Order

**Query:**

sql

Copy code

SELECT \* FROM customers ORDER BY Address;

**Output:**

| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| --- | --- | --- | --- | --- |
| 7 | Rohit Shrivastav | 19 | Ahemdabad | 38000 |
| 10 | Sahil Sheikh | 35 | Aurangabad | 68800 |
| 4 | Ritesh Yadav | 36 | Azamgarh | 26000 |
| 2 | Shiva Tiwari | 22 | Bhopal | 21000 |
| 6 | Mahesh Sharma | 26 | Mathura | 22000 |
| 3 | Ajeet Bhargav | 45 | Meerut | 65000 |
| 1 | Himani Gupta | 21 | Modinagar | 22000 |
| 9 | Aakash Yadav | 32 | Mumbai | 43500 |
| 8 | Neeru Sharma | 29 | Pune | 40000 |
| 5 | Balwant Singh | 45 | Varanasi | 36000 |

### Example 3: Sorting by Salary in Descending Order

**Query:**

sql

Copy code

SELECT \* FROM customers ORDER BY Salary DESC;

**Output:**

| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| --- | --- | --- | --- | --- |
| 10 | Sahil Sheikh | 35 | Aurangabad | 68800 |
| 3 | Ajeet Bhargav | 45 | Meerut | 65000 |
| 9 | Aakash Yadav | 32 | Mumbai | 43500 |
| 8 | Neeru Sharma | 29 | Pune | 40000 |
| 7 | Rohit Shrivastav | 19 | Ahemdabad | 38000 |
| 5 | Balwant Singh | 45 | Varanasi | 36000 |
| 4 | Ritesh Yadav | 36 | Azamgarh | 26000 |
| 6 | Mahesh Sharma | 26 | Mathura | 22000 |
| 1 | Himani Gupta | 21 | Modinagar | 22000 |
| 2 | Shiva Tiwari | 22 | Bhopal | 21000 |

### Example 4: Sorting by Age in Descending Order

**Query:**

sql

Copy code

SELECT \* FROM customers ORDER BY Age DESC;

**Output:**

| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| --- | --- | --- | --- | --- |
| 3 | Ajeet Bhargav | 45 | Meerut | 65000 |
| 5 | Balwant Singh | 45 | Varanasi | 36000 |
| 4 | Ritesh Yadav | 36 | Azamgarh | 26000 |
| 10 | Sahil Sheikh | 35 | Aurangabad | 68800 |
| 9 | Aakash Yadav | 32 | Mumbai | 43500 |
| 8 | Neeru Sharma | 29 | Pune | 40000 |
| 6 | Mahesh Sharma | 26 | Mathura | 22000 |
| 2 | Shiva Tiwari | 22 | Bhopal | 21000 |
| 1 | Himani Gupta | 21 | Modinagar | 22000 |
| 7 | Rohit Shrivastav | 19 | Ahemdabad | 38000 |

## Another Example with Agents Table

Consider another table named agents with the following records:

| **AID** | **Name** | **WorkArea** | **Profit\_Percent** | **ContactNumber** | **Salary** |
| --- | --- | --- | --- | --- | --- |
| 1 | Gurpreet Singh | Bangalore | 1 | 9989675432 | 43000 |
| 2 | Sakshi Kumari | Chennai | 5 | 8190567342 | 25000 |
| 3 | Prachi Desai | Mumbai | 2 | 9056123432 | 60000 |
| 4 | Shivani More | Pune | 3 | 8894236789 | 35500 |
| 5 | Pallavi Singh | Delhi | 4 | 7798092341 | 38700 |
| 6 | Rohini Kulkarni | Ambala | 8 | 7890945612 | 25670 |
| 7 | Shweta Dixit | Chandigarh | 6 | 8898786453 | 31670 |
| 8 | Sonakshi Tiwari | Udaipur | 2 | 9809453421 | 25050 |
| 9 | Anushka Tripathi | Ujjain | 9 | 8909124326 | 38000 |
| 10 | Devika Sharma | Goa | 7 | 7864523145 | 44050 |

### Example 1: Sorting by Agent Names in Ascending Order

**Query:**

sql

Copy code

SELECT \* FROM agents ORDER BY Name ASC;

**Output:**

| **AID** | **Name** | **WorkArea** | **Profit\_Percent** | **ContactNumber** | **Salary** |
| --- | --- | --- | --- | --- | --- |
| 9 | Anushka Tripathi | Ujjain | 9 | 8909124326 | 38000 |
| 10 | Devika Sharma | Goa | 7 | 7864523145 | 44050 |
| 1 | Gurpreet Singh | Bangalore | 1 | 9989675432 | 43000 |
| 5 | Pallavi Singh | Delhi | 4 | 7798092341 | 38700 |
| 3 | Prachi Desai | Mumbai | 2 | 9056123432 | 60000 |
| 6 | Rohini Kulkarni | Ambala | 8 | 7890945612 | 25670 |
| 2 | Sakshi Kumari | Chennai | 5 | 8190567342 | 25000 |
| 4 | Shivani More | Pune | 3 | 8894236789 | 35500 |
| 7 | Shweta Dixit | Chandigarh | 6 | 8898786453 | 31670 |
| 8 | Sonakshi Tiwari | Udaipur | 2 | 9809453421 | 25050 |

### Example 2: Sorting by Work Area in Descending Order

**Query:**

sql

Copy code

SELECT \* FROM agents ORDER BY WorkArea DESC;

**Output:**

| **AID** | **Name** | **WorkArea** | **Profit\_Percent** | **ContactNumber** | **Salary** |
| --- | --- | --- | --- | --- | --- |
| 9 | Anushka Tripathi | Ujjain | 9 | 8909124326 | 38000 |
| 8 | Sonakshi Tiwari | Udaipur | 2 | 9809453421 | 25050 |
| 4 | Shivani More | Pune | 3 | 8894236789 | 35500 |
| 3 | Prachi Desai | Mumbai | 2 | 9056123432 | 60000 |
| 10 | Devika Sharma | Goa | 7 | 7864523145 | 44050 |
| 5 | Pallavi Singh | Delhi | 4 | 7798092341 | 38700 |
| 2 | Sakshi Kumari | Chennai | 5 | 8190567342 | 25000 |
| 7 | Shweta Dixit | Chandigarh | 6 | 8898786453 | 31670 |
| 1 | Gurpreet Singh | Bangalore | 1 | 9989675432 | 43000 |
| 6 | Rohini Kulkarni | Ambala | 8 | 7890945612 | 25670 |

### Example 3: Sorting by Salary in Ascending Order

**Query:**

sql

Copy code

SELECT \* FROM agents ORDER BY Salary;

**Output:**

| **AID** | **Name** | **WorkArea** | **Profit\_Percent** | **ContactNumber** | **Salary** |
| --- | --- | --- | --- | --- | --- |
| 2 | Sakshi Kumari | Chennai | 5 | 8190567342 | 25000 |
| 8 | Sonakshi Tiwari | Udaipur | 2 | 9809453421 | 25050 |
| 6 | Rohini Kulkarni | Ambala | 8 | 7890945612 | 25670 |
| 7 | Shweta Dixit | Chandigarh | 6 | 8898786453 | 31670 |
| 4 | Shivani More | Pune | 3 | 8894236789 | 35500 |
| 9 | Anushka Tripathi | Ujjain | 9 | 8909124326 | 38000 |
| 5 | Pallavi Singh | Delhi | 4 | 7798092341 | 38700 |
| 1 | Gurpreet Singh | Bangalore | 1 | 9989675432 | 43000 |
| 10 | Devika Sharma | Goa | 7 | 7864523145 | 44050 |
| 3 | Prachi Desai | Mumbai | 2 | 9056123432 | 60000 |

### Example 4: Sorting by Salary in Descending Order

**Query:**

sql

Copy code

SELECT \* FROM agents ORDER BY Salary DESC;

**Output:**

| **AID** | **Name** | **WorkArea** | **Profit\_Percent** | **ContactNumber** | **Salary** |
| --- | --- | --- | --- | --- | --- |
| 3 | Prachi Desai | Mumbai | 2 | 9056123432 | 60000 |
| 10 | Devika Sharma | Goa | 7 | 7864523145 | 44050 |
| 1 | Gurpreet Singh | Bangalore | 1 | 9989675432 | 43000 |
| 5 | Pallavi Singh | Delhi | 4 | 7798092341 | 38700 |
| 9 | Anushka Tripathi | Ujjain | 9 | 8909124326 | 38000 |
| 4 | Shivani More | Pune | 3 | 8894236789 | 35500 |
| 7 | Shweta Dixit | Chandigarh | 6 | 8898786453 | 31670 |
| 6 | Rohini Kulkarni | Ambala | 8 | 7890945612 | 25670 |
| 8 | Sonakshi Tiwari | Udaipur | 2 | 9809453421 | 25050 |
| 2 | Sakshi Kumari | Chennai | 5 | 8190567342 | 25000 |

These examples demonstrate how the ORDER BY clause can be used to sort records in SQL databases in both ascending and descending order based on different columns.

ORDER BY MULTIPLE CONDITIONS:

Here's an overview and examples of using the ORDER BY clause in SQL to sort records in descending order:

### Overview:

The ORDER BY clause in SQL is used to sort the result-set by one or more columns either in ascending (ASC) or descending (DESC) order. If neither ASC nor DESC is specified, ASC (ascending) is assumed by default.

### Syntax:

sql

Copy code

SELECT column1, column2, ...

FROM table\_name

ORDER BY column1 DESC, column2 ASC, ...;

### Examples:

Let's consider two tables for demonstration: customers and agents.

#### Example 1: Sorting customers table by Name in descending order

sql

Copy code

SELECT \*

FROM customers

ORDER BY NAME DESC;

Output:

css

Copy code

ID Name Age Address Salary

2 Shiva Tiwari 22 Bhopal 21000

10 Sahil Sheikh 35 Aurangabad 68800

7 Rohit Shrivastav 19 Ahemdabad 38000

4 Ritesh Yadav 36 Azamgarh 26000

8 Neeru Sharma 29 Pune 40000

6 Mahesh Sharma 26 Mathura 22000

1 Himani Gupta 21 Modinagar 22000

5 Balwant Singh 45 Varanasi 36000

3 Ajeet Bhargav 45 Meerut 65000

9 Aakash Yadav 32 Mumbai 43500

#### Example 2: Sorting agents table by Salary in descending order

sql

Copy code

SELECT \*

FROM agents

ORDER BY Salary DESC;

Output:

Copy code

AID Name WorkArea Profit\_Percent ContactNumber Salary

3 Prachi Desai Mumbai 2 9056123432 60000

10 Devika Sharma Goa 7 7864523145 44050

1 Gurpreet Singh Bangalore 1 9989675432 43000

5 Pallavi Singh Delhi 4 7798092341 38700

9 Anushka Tripathi Ujjain 9 8909124326 38000

4 Shivani More Pune 3 8894236789 35500

7 Shweta Dixit Chandigarh 6 8898786453 31670

6 Rohini Kulkarni Ambala 8 7890945612 25670

8 Sonakshi Tiwari Udaipur 2 9809453421 25050

2 Sakshi Kumari Chennai 5 8190567342 25000

#### Example 3: Sorting customers table by Age in descending order

sql

Copy code

SELECT \*

FROM customers

ORDER BY Age DESC;

Output:

css

Copy code

ID Name Age Address Salary

3 Ajeet Bhargav 45 Meerut 65000

5 Balwant Singh 45 Varanasi 36000

4 Ritesh Yadav 36 Azamgarh 26000

10 Sahil Sheikh 35 Aurangabad 68800

9 Aakash Yadav 32 Mumbai 43500

8 Neeru Sharma 29 Pune 40000

6 Mahesh Sharma 26 Mathura 22000

2 Shiva Tiwari 22 Bhopal 21000

1 Himani Gupta 21 Modinagar 22000

7 Rohit Shrivastav 19 Ahemdabad 38000

# ORDER BY RAND():

o fetch random rows from a database table, different SQL dialects provide varying approaches. Here's how you can achieve this with different databases:

### MySQL

To select a single random row from a table in MySQL:

sql

Copy code

SELECT \* FROM items ORDER BY RAND() LIMIT 1;

To select all rows from a table in random order in MySQL:

sql

Copy code

SELECT \* FROM items ORDER BY RAND();

### PostgreSQL

To select a single random row from a table in PostgreSQL:

sql

Copy code

SELECT \* FROM items ORDER BY RANDOM() LIMIT 1;

To select all rows from a table in random order in PostgreSQL:

sql

Copy code

SELECT \* FROM items ORDER BY RANDOM();

### SQL Server

To select a single random row from a table in SQL Server:

sql

Copy code

SELECT TOP 1 \* FROM items ORDER BY NEWID();

### Oracle

To select a single random row from a table in Oracle:

sql

Copy code

SELECT \* FROM (

SELECT \* FROM items ORDER BY dbms\_random.value

) WHERE rownum = 1;

### IBM DB2

To select a single random row from a table in IBM DB2:

sql

Copy code

SELECT \* FROM items ORDER BY RAND() FETCH FIRST 1 ROWS ONLY;

### Explanation:

* **MySQL**: Uses RAND() function to generate a random number and sorts by it.
* **PostgreSQL**: Uses RANDOM() function similarly to MySQL's RAND().
* **SQL Server**: Utilizes NEWID() to generate a new uniqueidentifier, effectively randomizing the order.
* **Oracle**: Uses dbms\_random.value function within a subquery and filters with rownum to fetch a single random row.
* **IBM DB2**: Uses RAND() and FETCH FIRST 1 ROWS ONLY to limit the result to a single row.

### Usage scenarios:

Fetching random rows is useful in applications where you want to present varied data to users, such as displaying random items in an e-commerce site or showing a random article from a database.

### Considerations:

* **Performance**: Sorting entire result sets randomly (ORDER BY RAND()) can be inefficient for large datasets compared to fetching a single random row.
* **Consistency**: Results may vary each time the query is executed due to the randomness, which is suitable for randomization needs but not for predictable ordering.

By leveraging these SQL techniques, you can efficiently retrieve random data from your database tables according to the specific requirements and capabilities of each database management system.

# SQL INSERT:

### SQL INSERT Statement

The SQL INSERT statement is used to insert a single or multiple records into a table. There are several ways to insert data into a table:

#### 1. Direct Insertion into a Table

**Method 1: Without Specifying Column Names**

sql

Copy code

INSERT INTO table\_name

VALUES (value1, value2, value3, ...);

**Method 2: Specifying Column Names**

sql

Copy code

INSERT INTO table\_name (column1, column2, column3, ...)

VALUES (value1, value2, value3, ...);

### Examples:

Consider a table named STUDENTS:

sql

Copy code

CREATE TABLE STUDENTS (

ROLL\_NO INT,

NAME VARCHAR(50),

AGE INT,

CITY VARCHAR(50)

);

**Inserting Single Records:**

sql

Copy code

INSERT INTO STUDENTS (ROLL\_NO, NAME, AGE, CITY)

VALUES (1, 'ABHIRAM', 22, 'ALLAHABAD');

INSERT INTO STUDENTS (ROLL\_NO, NAME, AGE, CITY)

VALUES (2, 'ALKA', 20, 'GHAZIABAD');

INSERT INTO STUDENTS (ROLL\_NO, NAME, AGE, CITY)

VALUES (3, 'DISHA', 21, 'VARANASI');

INSERT INTO STUDENTS (ROLL\_NO, NAME, AGE, CITY)

VALUES (4, 'ESHA', 21, 'DELHI');

INSERT INTO STUDENTS (ROLL\_NO, NAME, AGE, CITY)

VALUES (5, 'MANMEET', 23, 'JALANDHAR');

The table STUDENTS will have the following records:

| **ROLL\_NO** | **NAME** | **AGE** | **CITY** |
| --- | --- | --- | --- |
| 1 | ABHIRAM | 22 | ALLAHABAD |
| 2 | ALKA | 20 | GHAZIABAD |
| 3 | DISHA | 21 | VARANASI |
| 4 | ESHA | 21 | DELHI |
| 5 | MANMEET | 23 | JALANDHAR |

**Inserting Another Record:**

sql

Copy code

INSERT INTO STUDENTS

VALUES (6, 'PRATIK', 24, 'KANPUR');

The table STUDENTS will now include:

| **ROLL\_NO** | **NAME** | **AGE** | **CITY** |
| --- | --- | --- | --- |
| 6 | PRATIK | 24 | KANPUR |

### 2. Inserting Data through SELECT Statement

**Syntax:**

sql

Copy code

INSERT INTO table\_name

[(column1, column2, ...)]

SELECT column1, column2, ...

FROM table\_name

[WHERE condition];

### 3. Inserting Multiple Rows

**Example 1:**

To create a table named student in the dbs database:

sql

Copy code

USE dbs;

CREATE TABLE student (

ID INT,

Name VARCHAR(20),

Percentage INT,

Location VARCHAR(20),

DateOfBirth DATE

);

Inserting multiple records:

sql

Copy code

INSERT INTO student (ID, Name, Percentage, Location, DateOfBirth) VALUES

(1, 'Manthan Koli', 79, 'Delhi', '2003-08-20'),

(2, 'Dev Dixit', 75, 'Pune', '1999-06-17'),

(3, 'Aakash Deshmukh', 87, 'Mumbai', '1997-09-12'),

(4, 'Aaryan Jaiswal', 90, 'Chennai', '2005-10-02'),

(5, 'Rahul Khanna', 92, 'Ambala', '1996-03-04'),

(6, 'Pankaj Deshmukh', 67, 'Kanpur', '2000-02-02'),

(7, 'Gaurav Kumar', 84, 'Chandigarh', '1998-07-06'),

(8, 'Sanket Jain', 61, 'Shimla', '1990-09-08'),

(9, 'Sahil Wagh', 90, 'Kolkata', '1968-04-03'),

(10, 'Saurabh Singh', 54, 'Kashmir', '1989-01-06');

Verifying the records:

sql

Copy code

SELECT \* FROM student;

Result:

| **ID** | **Name** | **Percentage** | **Location** | **DateOfBirth** |
| --- | --- | --- | --- | --- |
| 1 | Manthan Koli | 79 | Delhi | 2003-08-20 |
| 2 | Dev Dixit | 75 | Pune | 1999-06-17 |
| 3 | Aakash Deshmukh | 87 | Mumbai | 1997-09-12 |
| 4 | Aaryan Jaiswal | 90 | Chennai | 2005-10-02 |
| 5 | Rahul Khanna | 92 | Ambala | 1996-03-04 |
| 6 | Pankaj Deshmukh | 67 | Kanpur | 2000-02-02 |
| 7 | Gaurav Kumar | 84 | Chandigarh | 1998-07-06 |
| 8 | Sanket Jain | 61 | Shimla | 1990-09-08 |
| 9 | Sahil Wagh | 90 | Kolkata | 1968-04-03 |
| 10 | Saurabh Singh | 54 | Kashmir | 1989-01-06 |

**Example 2:**

Creating items\_tbl in the dbs database:

sql

Copy code

USE dbs;

CREATE TABLE items\_tbl (

ID INT,

Item\_Name VARCHAR(20),

Item\_Quantity INT,

Item\_Price INT,

Purchase\_Date DATE

);

Inserting multiple records:

sql

Copy code

INSERT INTO items\_tbl (ID, Item\_Name, Item\_Quantity, Item\_Price, Purchase\_Date) VALUES

(1, 'Soap', 5, 200, '2021-07-08'),

(2, 'Toothpaste', 2, 80, '2021-07-10'),

(3, 'Pen', 10, 50, '2021-07-12'),

(4, 'Bottle', 1, 250, '2021-07-13'),

(5, 'Brush', 3, 90, '2021-07-15'),

(6, 'Notebooks', 10, 1000, '2021-07-26'),

(7, 'Handkerchief', 3, 100, '2021-07-28'),

(8, 'Chips Packet', 5, 50, '2021-07-30'),

(9, 'Marker', 2, 30, '2021-08-13'),

(10, 'Scissors', 1, 60, '2021-08-13');

Verifying the records:

sql

Copy code

SELECT \* FROM items\_tbl;

Result:

| **ID** | **Item\_Name** | **Item\_Quantity** | **Item\_Price** | **Purchase\_Date** |
| --- | --- | --- | --- | --- |
| 1 | Soap | 5 | 200 | 2021-07-08 |
| 2 | Toothpaste | 2 | 80 | 2021-07-10 |
| 3 | Pen | 10 | 50 | 2021-07-12 |
| 4 | Bottle | 1 | 250 | 2021-07-13 |
| 5 | Brush | 3 | 90 | 2021-07-15 |
| 6 | Notebooks | 10 | 1000 | 2021-07-26 |
| 7 | Handkerchief | 3 | 100 | 2021-07-28 |
| 8 | Chips Packet | 5 | 50 | 2021-07-30 |
| 9 | Marker | 2 | 30 | 2021-08-13 |
| 10 | Scissors | 1 | 60 | 2021-08-13 |

These examples illustrate the flexibility and power of the SQL INSERT statement for adding both single and multiple records into a table.

# SQL UPDATE:

### SQL UPDATE

The SQL UPDATE statement is used to modify existing records in a table. The rows that are updated are determined by a condition specified in the WHERE clause. If the WHERE clause is omitted, all rows in the table are updated.

#### Basic Syntax

The basic syntax for the UPDATE statement is as follows:

sql

Copy code

UPDATE table\_name

SET column\_name1 = value1, column\_name2 = value2, ...

WHERE condition;

### Examples

#### Example 1: Updating a Single Column

Let's consider a students table:

**Source Table:**

Copy code

Student\_Id FirstName LastName User\_Name

1 Ada Sharma sharmili

2 Rahul Maurya sofamous

3 James Walker jonny

**SQL Statement:**

sql

Copy code

UPDATE students

SET User\_Name = 'beinghuman'

WHERE Student\_Id = 3;

**Result:**

Copy code

Student\_Id FirstName LastName User\_Name

1 Ada Sharma sharmili

2 Rahul Maurya sofamous

3 James Walker beinghuman

#### Example 2: Updating Multiple Columns

**SQL Statement:**

sql

Copy code

UPDATE students

SET User\_Name = 'beserious', FirstName = 'Johnny'

WHERE Student\_Id = 3;

**Result:**

Copy code

Student\_Id FirstName LastName User\_Name

1 Ada Sharma sharmili

2 Rahul Maurya sofamous

3 Johnny Walker beserious

### MySQL Syntax for Updating a Table

The MySQL syntax for updating multiple fields is:

sql

Copy code

UPDATE table\_name

SET field1 = new\_value1, field2 = new\_value2

[WHERE condition];

### SQL UPDATE with SELECT

You can use a SELECT statement to update records through an UPDATE statement.

#### Example:

sql

Copy code

UPDATE tableDestination

SET tableDestination.col = value

WHERE EXISTS (

SELECT col2.value

FROM tblSource

WHERE tblSource.join\_col = tableDestination.join\_col

AND tblSource.constraint = value

);

#### Another Example:

sql

Copy code

UPDATE Table

SET

Table.column1 = othertable.column1,

Table.column2 = othertable.column2

FROM

Table

INNER JOIN

other\_table

ON

Table.id = other\_table.id;

### MySQL Syntax for UPDATE with SELECT

For MySQL, you can use the following syntax to update using a SELECT statement:

#### Example with Two Tables:

* First table: categories with columns cat\_id, cat\_name
* Second table: related\_categories with columns rel\_cat\_id, rel\_cat\_name

sql

Copy code

UPDATE categories

SET cat\_name = (

SELECT rel\_cat\_name

FROM related\_categories

WHERE related\_categories.rel\_cat\_id = categories.cat\_id

);

### Updating Single and Multiple Columns

#### Updating a Single Column:

sql

Copy code

UPDATE students

SET student\_id = 001

WHERE student\_name = 'AJEET';

#### Updating Multiple Columns:

sql

Copy code

UPDATE students

SET student\_name = 'AJEET',

Religion = 'HINDU'

WHERE student\_name = 'RAJU';

This will update the student\_name to 'AJEET' and Religion to 'HINDU' where student\_name is 'RAJU'.

### Summary

The UPDATE statement is a powerful SQL command that allows you to modify existing data within a table. By using the WHERE clause, you can target specific rows for updates. Multiple columns can be updated in a single statement by separating the assignments with commas. Additionally, using SELECT statements within UPDATE allows for more complex updates based on data from other tables.

### create database joins;

### use joins;

### create table student(

### s\_id int,

### name varchar(30),

### c\_id int,

### marks int);

### create table course(

### c\_id int,

### course\_name varchar(30),

### c\_credits int);

### insert into student

### values

### (1,"hansi1",1,59),

### (2,"hansi2",2,67),

### (3,"hansi3",3,77),

### (4,"hansi4",4,87);

### insert into course

### values

### (11,"ds",3),

### (22,"ds",4),

### (3,"ds",5),

### (44,"ds",6);

### /\*

### NOTE:

### 1)JOIN AND INNERJOIN and CROSS JOIN ALL ARE SAME BUT NOTE:if condition is specified in one OF THE JOIN it should be ther in all joins to be same

### --ex;

### select \* from student join course;

### select \* from student cross join course;

### select \* from student inner join course;

### u will get same output for all above 3 codes

### select \* from student join course on student.c\_id=course.c\_id;

### select \* from student cross join course on student.c\_id=course.c\_id;

### select \* from student inner join course on student.c\_id=course.c\_id;

### similarly the output of these also will be same

### but if u see the output of below two code it will different

### select \* from student join course;

### select \* from student cross join course on student.c\_id=course.c\_id;

### and

### select \* from student join course;

### select \* from student inner join course on student.c\_id=course.c\_id;

### similarly in all other cases of with and without condition statements if u compare u wont get same output

### \*/

### -- cross join

### select \* from student join course; -- here each attributes of course will me merged with all attributes of student

### -- inner join

### select \* from student join course on student.c\_id=course.c\_id; -- returns rows by combining attributes of both tables when the condition is satisfied

### -- join

### select \* from student inner join course; -- NOTE:both inner join and join are same

### /\* left outer join:

### 1)firstly merges the rows of 2 table like

### 1st row of 1st table +(with) 1st row of 2nd table

### similarly 2nd row of 1st table +(with) 2nd row of 2nd table

### ........and so on till last count of row number not like 1st row of 1st table with

### all rows of 2nd table as in join,crossjoin,innerjoin

### 2)in that merging if there is col with same name in the two rows they are displayed individually

### 3)after merging as it left outer join the column values of all rows of left table are displayed but in

### w.r.t right table column values for the rows where condition is satisfied the column values of the respected row

### are displayed

### \*/

### select \* from student left join course on student.c\_id=course.c\_id;

### /\* right outer join:

### 1)firstly merges the rows of 2 table like

### 1st row of 1st table +(with) 1st row of 2nd table

### similarly 2nd row of 1st table +(with) 2nd row of 2nd table

### ........and so on till last count of row number not like 1st row of 1st table with

### all rows of 2nd table as in join,crossjoin,innerjoin

### 2)in that merging if there is col with same name in the two rows they are displayed individually

### 3)after merging as it right outer join the column values of all rows of right table are displayed but in

### w.r.t left table column values for the rows where condition is satisfied the column values of the respected row

### are displayed

### \*/

### select \* from student right join course on student.c\_id=course.c\_id;

### # theta join(>=,<=,>,<)

### select \* from student join course on student.c\_id<=course.c\_id;

### #equi join(=)

### select \* from student join course on student.c\_id=course.c\_id;

### #natural join

### select \* from student natural join course ; #it is like checks for same col in two tables then checks for same values in similar col of two tables if values are similar then that row of 2 tables are merged with no repeatition of columns

### primary key

Sure, let's go through the concept of primary keys in SQL with examples to illustrate how they work, including how to create, alter, and drop primary keys.

### Primary Key Basics

A primary key is a column or a set of columns that uniquely identifies each row in a table. It ensures that the column or set of columns contains unique values and does not accept NULL values.

### Creating a Primary Key on a Single Column

#### Example in MySQL

sql

Copy code

CREATE TABLE students (

S\_Id INT NOT NULL,

LastName VARCHAR(255) NOT NULL,

FirstName VARCHAR(255),

Address VARCHAR(255),

City VARCHAR(255),

PRIMARY KEY (S\_Id)

);

In this example:

* The column S\_Id is designated as the primary key.
* It uniquely identifies each record in the students table.

#### Example in SQL Server, Oracle, MS Access

sql

Copy code

CREATE TABLE students (

S\_Id INT NOT NULL PRIMARY KEY,

LastName VARCHAR(255) NOT NULL,

FirstName VARCHAR(255),

Address VARCHAR(255),

City VARCHAR(255)

);

### Creating a Composite Primary Key (Multiple Columns)

#### Example in MySQL, SQL Server, Oracle, MS Access

sql

Copy code

CREATE TABLE students (

S\_Id INT NOT NULL,

LastName VARCHAR(255) NOT NULL,

FirstName VARCHAR(255),

Address VARCHAR(255),

City VARCHAR(255),

CONSTRAINT pk\_StudentID PRIMARY KEY (S\_Id, LastName)

);

In this example:

* The primary key is composed of two columns: S\_Id and LastName.
* This combination of columns uniquely identifies each record in the table.

### Adding a Primary Key to an Existing Table

If you have already created a table and you want to add a primary key constraint, you can use the ALTER TABLE statement.

#### Adding a Primary Key on One Column

sql

Copy code

ALTER TABLE students

ADD PRIMARY KEY (S\_Id);

#### Adding a Composite Primary Key (Multiple Columns)

sql

Copy code

ALTER TABLE students

ADD CONSTRAINT pk\_StudentID PRIMARY KEY (S\_Id, LastName);

### Dropping a Primary Key Constraint

If you need to remove a primary key constraint, the syntax varies slightly between different database systems.

#### Example in MySQL

sql

Copy code

ALTER TABLE students

DROP PRIMARY KEY;

#### Example in SQL Server, Oracle, MS Access

sql

Copy code

ALTER TABLE students

DROP CONSTRAINT pk\_StudentID;

### Practical Examples

#### Creating the Students Table

sql

Copy code

CREATE TABLE students (

S\_Id INT NOT NULL,

LastName VARCHAR(255) NOT NULL,

FirstName VARCHAR(255),

Address VARCHAR(255),

City VARCHAR(255),

PRIMARY KEY (S\_Id)

);

-- Inserting some data into the students table

INSERT INTO students (S\_Id, LastName, FirstName, Address, City) VALUES

(1, 'Smith', 'John', '123 Main St', 'Springfield'),

(2, 'Doe', 'Jane', '456 Elm St', 'Shelbyville'),

(3, 'Black', 'Will', '789 Oak St', 'Ogdenville');

#### Adding a Composite Primary Key

First, let's drop the existing primary key and add a composite key.

sql

Copy code

ALTER TABLE students DROP PRIMARY KEY;

ALTER TABLE students ADD CONSTRAINT pk\_StudentID PRIMARY KEY (S\_Id, LastName);

#### Resulting Table Schema

| **S\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| --- | --- | --- | --- | --- |
| 1 | Smith | John | 123 Main St | Springfield |
| 2 | Doe | Jane | 456 Elm St | Shelbyville |
| 3 | Black | Will | 789 Oak St | Ogdenville |

### Summary

* **Primary Key**: Ensures unique identification of each row and does not accept NULL values.
* **Single Column Primary Key**: Uses one column to uniquely identify each row.
* **Composite Primary Key**: Uses multiple columns to uniquely identify each row.
* **Adding/Dropping Primary Keys**: Can be done using ALTER TABLE statements.

This should give you a clear understanding of primary keys in SQL, along with practical examples of creating, altering, and dropping them.

### Sql FOREIGN KEY:

### Understanding SQL FOREIGN KEY

A foreign key is a column or a set of columns in one table that refers to the primary key columns in another table. This relationship helps maintain referential integrity between the two tables.

### Example Scenario

Consider two tables:

1. students
2. orders

#### students Table

| **S\_Id** | **LastName** | **FirstName** | **City** |
| --- | --- | --- | --- |
| 1 | MAURYA | AJEET | ALLAHABAD |
| 2 | JAISWAL | RATAN | GHAZIABAD |
| 3 | ARORA | SAUMYA | MODINAGAR |

#### orders Table

| **O\_Id** | **OrderNo** | **S\_Id** |
| --- | --- | --- |
| 1 | 99586465 | 2 |
| 2 | 78466588 | 2 |
| 3 | 22354846 | 3 |
| 4 | 57698656 | 1 |

In this example:

* The S\_Id column in the students table is the primary key.
* The S\_Id column in the orders table is a foreign key that references the S\_Id column in the students table.

### Creating a Foreign Key Constraint

#### While Creating the Table

##### MySQL

sql

Copy code

CREATE TABLE students (

S\_Id INT NOT NULL,

LastName VARCHAR(255) NOT NULL,

FirstName VARCHAR(255),

City VARCHAR(255),

PRIMARY KEY (S\_Id)

);

CREATE TABLE orders (

O\_Id INT NOT NULL,

Order\_No INT NOT NULL,

S\_Id INT,

PRIMARY KEY (O\_Id),

FOREIGN KEY (S\_Id) REFERENCES students(S\_Id)

);

##### SQL Server / Oracle / MS Access

sql

Copy code

CREATE TABLE students (

S\_Id INT NOT NULL PRIMARY KEY,

LastName VARCHAR(255) NOT NULL,

FirstName VARCHAR(255),

City VARCHAR(255)

);

CREATE TABLE orders (

O\_Id INT NOT NULL PRIMARY KEY,

Order\_No INT NOT NULL,

S\_Id INT,

FOREIGN KEY (S\_Id) REFERENCES students(S\_Id)

);

#### Adding a Foreign Key to an Existing Table

If the orders table already exists, you can add a foreign key constraint using the ALTER TABLE statement.

##### MySQL / SQL Server / Oracle / MS Access

sql

Copy code

ALTER TABLE orders

ADD CONSTRAINT fk\_PerOrders

FOREIGN KEY (S\_Id) REFERENCES students(S\_Id);

### Dropping a Foreign Key Constraint

To remove a foreign key constraint, you use the ALTER TABLE statement followed by DROP FOREIGN KEY for MySQL or DROP CONSTRAINT for SQL Server, Oracle, and MS Access.

#### MySQL

sql

Copy code

ALTER TABLE orders

DROP FOREIGN KEY fk\_PerOrders;

#### SQL Server / Oracle / MS Access

sql

Copy code

ALTER TABLE orders

DROP CONSTRAINT fk\_PerOrders;

### Primary Key vs. Foreign Key

1. **Primary Key**:
   * Cannot be null.
   * Must be unique.
   * Uniquely identifies each record in a table.
   * A table can have only one primary key.
   * Automatically creates a clustered index.
2. **Foreign Key**:
   * Can be null.
   * Can have duplicate values.
   * Ensures referential integrity between two tables.
   * A table can have multiple foreign keys.
   * Does not automatically create an index (needs to be created manually).

### Practical Example

Let's walk through a practical example using MySQL.

1. **Creating the students Table**:

sql

Copy code

CREATE TABLE students (

S\_Id INT NOT NULL PRIMARY KEY,

LastName VARCHAR(255) NOT NULL,

FirstName VARCHAR(255),

City VARCHAR(255)

);

INSERT INTO students (S\_Id, LastName, FirstName, City) VALUES

(1, 'MAURYA', 'AJEET', 'ALLAHABAD'),

(2, 'JAISWAL', 'RATAN', 'GHAZIABAD'),

(3, 'ARORA', 'SAUMYA', 'MODINAGAR');

1. **Creating the orders Table with Foreign Key**:

sql

Copy code

CREATE TABLE orders (

O\_Id INT NOT NULL PRIMARY KEY,

Order\_No INT NOT NULL,

S\_Id INT,

FOREIGN KEY (S\_Id) REFERENCES students(S\_Id)

);

INSERT INTO orders (O\_Id, Order\_No, S\_Id) VALUES

(1, 99586465, 2),

(2, 78466588, 2),

(3, 22354846, 3),

(4, 57698656, 1);

1. **Querying the Tables**:

To join the tables and display data from both:

sql

Copy code

SELECT s.S\_Id, s.LastName, s.FirstName, o.Order\_No

FROM students s

INNER JOIN orders o ON s.S\_Id = o.S\_Id;

#### Result:

| **S\_Id** | **LastName** | **FirstName** | **Order\_No** |
| --- | --- | --- | --- |
| 1 | MAURYA | AJEET | 57698656 |
| 2 | JAISWAL | RATAN | 99586465 |
| 2 | JAISWAL | RATAN | 78466588 |
| 3 | ARORA | SAUMYA | 22354846 |

This example demonstrates how foreign keys maintain relationships between tables, ensuring data integrity and enabling meaningful data retrieval through joins.

4o

Primary Key vs. Foreign Key Primary Key:

Cannot be null. Must be unique. Uniquely identifies each record in a table. A table can have only one primary key. Automatically creates a clustered index. Foreign Key:

Can be null. Can have duplicate values. Ensures referential integrity between two tables. A table can have multiple foreign keys. Does not automatically create an index (needs to be created manually). here foreign key references to the primary key of other table as primary key wont be having null values how caome foreign key contains null values explain with an example

ChatGPT

In a relational database, a foreign key is used to link two tables together. While a primary key uniquely identifies each record in a table and cannot be null, a foreign key in another table can indeed have null values. This is because the foreign key can represent an optional relationship.

### Example

Consider the following two tables: employees and departments.

#### departments Table

| **dept\_id** | **dept\_name** |
| --- | --- |
| 1 | HR |
| 2 | IT |
| 3 | Finance |

Here, dept\_id is the primary key of the departments table.

#### employees Table

| **emp\_id** | **emp\_name** | **dept\_id** |
| --- | --- | --- |
| 1 | John Doe | 1 |
| 2 | Jane Smith | 2 |
| 3 | Jim Brown | NULL |
| 4 | Lucy Liu | 3 |

In the employees table, dept\_id is a foreign key that references dept\_id in the departments table.

### Explanation

* **Primary Key (dept\_id in departments table)**:
  + Cannot be null.
  + Must be unique.
  + Uniquely identifies each department.
* **Foreign Key (dept\_id in employees table)**:
  + Can be null (indicating that an employee is not currently assigned to any department).
  + Can have duplicate values (multiple employees can belong to the same department).

### Why Foreign Key Can Be Null

The foreign key in the employees table can have null values because an employee might not be assigned to any department. This allows the employees table to accurately reflect real-world scenarios where certain relationships are optional.

### SQL to Create and Populate the Tables

sql

Copy code

CREATE TABLE departments (

dept\_id INT NOT NULL PRIMARY KEY,

dept\_name VARCHAR(255) NOT NULL

);

CREATE TABLE employees (

emp\_id INT NOT NULL PRIMARY KEY,

emp\_name VARCHAR(255) NOT NULL,

dept\_id INT,

FOREIGN KEY (dept\_id) REFERENCES departments(dept\_id)

);

INSERT INTO departments (dept\_id, dept\_name) VALUES

(1, 'HR'),

(2, 'IT'),

(3, 'Finance');

INSERT INTO employees (emp\_id, emp\_name, dept\_id) VALUES

(1, 'John Doe', 1),

(2, 'Jane Smith', 2),

(3, 'Jim Brown', NULL),

(4, 'Lucy Liu', 3);

### Query to Join the Tables

sql

Copy code

SELECT e.emp\_id, e.emp\_name, d.dept\_name

FROM employees e

LEFT JOIN departments d ON e.dept\_id = d.dept\_id;

#### Result

| **emp\_id** | **emp\_name** | **dept\_name** |
| --- | --- | --- |
| 1 | John Doe | HR |
| 2 | Jane Smith | IT |
| 3 | Jim Brown | NULL |
| 4 | Lucy Liu | Finance |

In the result, you can see that Jim Brown has no department assigned (NULL in dept\_name), demonstrating how the foreign key in the employees table can be null while referencing the primary key in the departments table. This reflects an optional relationship where some employees may not belong to any department.

# SQL COMPOSITE KEY

### QL Composite Key

A composite key is a combination of two or more columns in a table that together uniquely identify a record. This is useful when a single column is not sufficient to guarantee uniqueness. Each individual column in a composite key does not necessarily need to be unique by itself, but the combination of values across these columns must be unique.

### Example with Tables

Let's create an example with two tables: student\_courses and courses.

#### student\_courses Table

This table keeps track of which student is enrolled in which course.

| **student\_id** | **course\_id** |
| --- | --- |
| 1 | 101 |
| 1 | 102 |
| 2 | 101 |
| 3 | 103 |

#### courses Table

This table contains details about the courses.

| **course\_id** | **course\_name** |
| --- | --- |
| 101 | Mathematics |
| 102 | English |
| 103 | Computer Science |

In this scenario, the combination of student\_id and course\_id in the student\_courses table serves as a composite key because together they uniquely identify each record of student enrollment in a course.

### Creating Tables with Composite Key

#### MySQL

sql

Copy code

CREATE TABLE student\_courses (

student\_id INT,

course\_id INT,

PRIMARY KEY (student\_id, course\_id)

);

CREATE TABLE courses (

course\_id INT PRIMARY KEY,

course\_name VARCHAR(255)

);

#### SQL Server

sql

Copy code

CREATE TABLE student\_courses (

student\_id INT,

course\_id INT,

PRIMARY KEY (student\_id, course\_id)

);

CREATE TABLE courses (

course\_id INT PRIMARY KEY,

course\_name NVARCHAR(255)

);

### Inserting Data into Tables

sql

Copy code

INSERT INTO student\_courses (student\_id, course\_id) VALUES

(1, 101),

(1, 102),

(2, 101),

(3, 103);

INSERT INTO courses (course\_id, course\_name) VALUES

(101, 'Mathematics'),

(102, 'English'),

(103, 'Computer Science');

### Querying the Data

You can join the two tables to get a complete list of students and the courses they are enrolled in.

sql

Copy code

SELECT sc.student\_id, sc.course\_id, c.course\_name

FROM student\_courses sc

JOIN courses c ON sc.course\_id = c.course\_id;

#### Result

| **student\_id** | **course\_id** | **course\_name** |
| --- | --- | --- |
| 1 | 101 | Mathematics |
| 1 | 102 | English |
| 2 | 101 | Mathematics |
| 3 | 103 | Computer Science |

### Explanation

* **Composite Key (student\_id, course\_id)**:
  + The combination of student\_id and course\_id uniquely identifies each record in the student\_courses table.
  + Neither student\_id nor course\_id alone can uniquely identify a record in this table, but together they ensure uniqueness.

### Benefits of Composite Keys

* **Uniqueness**: Ensures that the combination of columns is unique, which is crucial in many-to-many relationships.
* **Data Integrity**: Helps maintain the integrity of the data by preventing duplicate records of the same combination.
* **Logical Representation**: Often represents real-world scenarios more accurately, such as students enrolling in courses, products in orders, etc.

By using composite keys, you can effectively manage and enforce uniqueness in your database where a single column is not sufficient to uniquely identify records.

# Unique Key in SQL

A unique key in SQL is a constraint that ensures all values in a column or a combination of columns are unique across the database table. Unlike primary keys, which do not allow NULL values, unique keys can accept a single NULL value, but no duplicates.

### Characteristics of Unique Keys

* **Uniqueness**: Ensures no two rows have the same value for the unique key column(s).
* **Null Values**: Allows one NULL value per column (except in Oracle, where NULL values are treated as distinct).
* **Multiple Unique Keys**: A table can have multiple unique keys, but only one primary key.
* **Indexes**: Unique keys automatically create non-clustered indexes to enforce the uniqueness.

### Creating Unique Key Constraints

#### Creating a Table with a Unique Key on a Single Column

sql

Copy code

-- SQL Server / Oracle / MS Access

CREATE TABLE students (

S\_Id INT NOT NULL UNIQUE,

LastName VARCHAR(255) NOT NULL,

FirstName VARCHAR(255),

City VARCHAR(255)

);

-- MySQL

CREATE TABLE students (

S\_Id INT NOT NULL,

LastName VARCHAR(255) NOT NULL,

FirstName VARCHAR(255),

City VARCHAR(255),

UNIQUE (S\_Id)

);

#### Creating a Table with a Unique Key on Multiple Columns

sql

Copy code

-- MySQL / SQL Server / Oracle / MS Access

CREATE TABLE students (

S\_Id INT NOT NULL,

LastName VARCHAR(255) NOT NULL,

FirstName VARCHAR(255),

City VARCHAR(255),

CONSTRAINT uc\_studentId UNIQUE (S\_Id, LastName)

);

### Adding a Unique Key Constraint to an Existing Table

#### Adding a Unique Key on a Single Column

sql

Copy code

-- MySQL / SQL Server / Oracle / MS Access

ALTER TABLE students

ADD UNIQUE (S\_Id);

#### Adding a Unique Key on Multiple Columns

sql

Copy code

-- MySQL / SQL Server / Oracle / MS Access

ALTER TABLE students

ADD CONSTRAINT uc\_StudentId UNIQUE (S\_Id, LastName);

### Dropping a Unique Key Constraint

#### Dropping a Unique Key in MySQL

sql

Copy code

ALTER TABLE students

DROP INDEX uc\_studentID;

#### Dropping a Unique Key in SQL Server / Oracle / MS Access

sql

Copy code

ALTER TABLE students

DROP CONSTRAINT uc\_studentID;

### Example: Unique Key with NULL Values

Consider a books table where the ISBN must be unique, but the publisher field can be NULL and still be unique.

#### Creating the Table

sql

Copy code

CREATE TABLE books (

BookID INT PRIMARY KEY,

Title VARCHAR(255),

ISBN VARCHAR(13) UNIQUE,

Publisher VARCHAR(255)

);

#### Inserting Data

sql

Copy code

INSERT INTO books (BookID, Title, ISBN, Publisher) VALUES

(1, 'Book One', '978-3-16-148410-0', 'Publisher A'),

(2, 'Book Two', '978-1-56619-909-4', NULL),

(3, 'Book Three', '978-0-262-13472-9', NULL);

#### Table Data

| **BookID** | **Title** | **ISBN** | **Publisher** |
| --- | --- | --- | --- |
| 1 | Book One | 978-3-16-148410-0 | Publisher A |
| 2 | Book Two | 978-1-56619-909-4 | NULL |
| 3 | Book Three | 978-0-262-13472-9 | NULL |

* The ISBN column is unique and cannot have duplicates.
* The Publisher column can have NULL values but each NULL is considered unique by the unique constraint.

### Summary

* **Primary Key**: Uniquely identifies each record and does not allow NULL values.
* **Unique Key**: Ensures unique values in the column(s) it is applied to and allows one NULL value.
* A table can have one primary key and multiple unique keys.
* Both constraints improve data integrity and performance by ensuring uniqueness and creating indexes.

Understanding unique keys and how to implement them is crucial for maintaining data integrity and designing efficient and robust database systems.

# Alternate Key in SQL

An **alternate key** is a candidate key that was not chosen to be the primary key of a table. It is still unique and can be used to identify a row in the table, but it is not the primary key.

### Key Concepts

* **Primary Key**: The main key used to uniquely identify each row in a table. There can be only one primary key in a table.
* **Candidate Key**: Any column or set of columns that can uniquely identify a row in a table. There can be multiple candidate keys in a table.
* **Alternate Key**: Any candidate key that is not selected as the primary key. It is still unique but is not the main identifier for the table.

### Example Scenario

Consider a students table with the following columns:

* StudentID (Primary Key)
* RollNo (Candidate Key)
* Name
* Class

Here, StudentID is chosen as the primary key. The RollNo is a candidate key but not the primary key, making it an alternate key.

### Table Creation

#### Creating the Table with a Primary Key and a Unique Constraint for Alternate Key

sql

Copy code

CREATE TABLE students (

StudentID INT PRIMARY KEY,

RollNo INT UNIQUE,

Name VARCHAR(255),

Class VARCHAR(50)

);

#### Inserting Data

sql

Copy code

INSERT INTO students (StudentID, RollNo, Name, Class) VALUES

(1, 101, 'Alice', '10A'),

(2, 102, 'Bob', '10B'),

(3, 103, 'Charlie', '10C');

#### Table Data

| **StudentID** | **RollNo** | **Name** | **Class** |
| --- | --- | --- | --- |
| 1 | 101 | Alice | 10A |
| 2 | 102 | Bob | 10B |
| 3 | 103 | Charlie | 10C |

### Summary of Key Types

* **Primary Key (StudentID)**: Uniquely identifies each student.
* **Alternate Key (RollNo)**: Uniquely identifies each student but is not the primary key.

### Additional Considerations

* **Composite Keys**: In some cases, an alternate key might be a composite key, which is a combination of two or more columns that together uniquely identify a record.

#### Example with Composite Key

If StudentID and Class together uniquely identify a record, they can be used as a composite alternate key.

sql

Copy code

CREATE TABLE students (

StudentID INT,

RollNo INT,

Name VARCHAR(255),

Class VARCHAR(50),

PRIMARY KEY (StudentID),

UNIQUE (StudentID, Class)

);

In this case, StudentID remains the primary key, and the combination of StudentID and Class is a composite unique key (an alternate key).

### Conclusion

An alternate key in SQL is any candidate key that is not chosen as the primary key. It provides an additional unique way to identify a row in a table, ensuring data integrity and supporting database operations. Understanding and using alternate keys effectively can enhance the design and functionality of your database.

# Like

The LIKE operator in SQL is used to search for a specified pattern in a column. It is commonly used in SELECT statements to filter rows based on matching patterns in string columns. The LIKE operator is case-insensitive in some SQL databases (like MySQL) and case-sensitive in others (like PostgreSQL).

### Syntax

sql

Copy code

SELECT column1, column2, ...

FROM table\_name

WHERE column\_name LIKE pattern;

### Wildcards

The LIKE operator uses two wildcards:

1. **Percent (%)**: Represents zero, one, or multiple characters.
2. **Underscore (\_)**: Represents a single character.

### Examples

Let's consider a table named Employees with the following data:

| **id** | **name** | **department** |
| --- | --- | --- |
| 1 | John Doe | HR |
| 2 | Jane Smith | Finance |
| 3 | Jack White | IT |
| 4 | Julie Green | HR |

#### Using % Wildcard

* **Example 1**: Find employees whose names start with 'J'.

sql

Copy code

SELECT name

FROM Employees

WHERE name LIKE 'J%';

**Result**:

| **name** |
| --- |
| John Doe |
| Jane Smith |
| Jack White |
| Julie Green |

* **Example 2**: Find employees whose names end with 'e'.

sql

Copy code

SELECT name

FROM Employees

WHERE name LIKE '%e';

**Result**:

| **name** |
| --- |
| Jane Smith |
| Julie Green |

* **Example 3**: Find employees whose names contain 'it'.

sql

Copy code

SELECT name

FROM Employees

WHERE name LIKE '%it%';

**Result**:

| **name** |
| --- |
| Jane Smith |
| Jack White |

#### Using \_ Wildcard

* **Example 4**: Find employees whose names have 'J' as the second character.

sql

Copy code

SELECT name

FROM Employees

WHERE name LIKE '\_J%';

**Result**:

| **name** |
| --- |
| (no rows match this pattern) |

* **Example 5**: Find employees whose names are exactly 5 characters long.

sql

Copy code

SELECT name

FROM Employees

WHERE name LIKE '\_\_\_\_\_';

**Result**:

| **name** |
| --- |
| (no rows match this pattern) |

### Combining Wildcards

* **Example 6**: Find employees whose names start with 'J' and end with 'e'.

sql

Copy code

SELECT name

FROM Employees

WHERE name LIKE 'J%e';

**Result**:

| **name** |
| --- |
| Jane Smith |
| Julie Green |

### Negating the LIKE Operator

You can use the NOT operator to find rows that do not match a pattern.

* **Example 7**: Find employees whose names do not start with 'J'.

sql

Copy code

SELECT name

FROM Employees

WHERE name NOT LIKE 'J%';

**Result**:

| **name** |
| --- |
| (no rows match this pattern) |

### Practical Tips

1. **Case Sensitivity**: The LIKE operator is case-insensitive in some databases (e.g., MySQL) and case-sensitive in others (e.g., PostgreSQL). Use ILIKE for case-insensitive matching in PostgreSQL.
2. **Performance**: The LIKE operator can be slow on large datasets, especially with patterns starting with %, as it can prevent the use of indexes.
3. **Escaping Special Characters**: If you need to search for the actual % or \_ characters, you can use an escape character.

sql

Copy code

SELECT name

FROM Employees

WHERE name LIKE 'A\\_%' ESCAPE '\';

This example escapes the underscore, so it searches for names that start with 'A\_'.

The LIKE operator is a powerful tool for pattern matching in SQL. Understanding how to use it effectively can greatly enhance your ability to query databases based on text patterns.

### Problem solving:

### Problem link(https://leetcode.com/problems/rising-temperature/editorial/?envType=study-plan-v2&envId=top-sql-50)

# DATEDIFF(date\_part, start\_date, end\_date)

* date\_part: Specifies the part of the date difference you want to calculate (e.g., day, month, year).
* start\_date: The starting date or timestamp.
* end\_date: The ending date or timestamp.

**Examples:**

1. Calculate the difference in days between two dates:

sql

Copy code

SELECT DATEDIFF(day, '2023-01-15', '2023-02-10') AS day\_diff;

Output: 26 (difference in days between January 15 and February 10, 2023)

1. Calculate the difference in months between two dates:

sql

Copy code

SELECT DATEDIFF(month, '2022-05-20', '2023-03-15') AS month\_diff;

Output: 10 (difference in months between May 20, 2022, and March 15, 2023)

1. Calculate the difference in years between two dates:

sql

Copy code

SELECT DATEDIFF(year, '2000-01-01', '2023-01-01') AS year\_diff;

Output: 23 (difference in years between January 1, 2000, and January 1, 2023)

SELECT w1.id FROM Weather w1 JOIN Weather w2 ON DATEDIFF(w1.recordDate, w2.recordDate) = 1 WHERE w1.temperature > w2.temperature;

give me clarity in this datediff whether it will be commparing like value in first row of w1.recorddate with all the w2.recorddate in all rows of w2 table of

The DATEDIFF(w1.recordDate, w2.recordDate) function in the query you provided calculates the difference in days between w1.recordDate and w2.recordDate. Let me clarify how this works in the context of your query:

### Query Breakdown:

sql

Copy code

SELECT

w1.id

FROM

Weather w1

JOIN

Weather w2

ON

DATEDIFF(w1.recordDate, w2.recordDate) = 1

WHERE

w1.temperature > w2.temperature;

### Explanation:

1. **Self-Join (JOIN Weather w2)**:
   * The Weather table is aliased as w1 and w2, creating a self-join. This means you're joining the Weather table with itself.
2. **Join Condition (ON DATEDIFF(w1.recordDate, w2.recordDate) = 1)**:
   * This condition specifies that you want to join rows where the difference in days (DATEDIFF) between w1.recordDate and w2.recordDate is exactly 1 day.
3. **Comparison in WHERE Clause (w1.temperature > w2.temperature)**:
   * After joining, this condition filters the result set to include only rows where the temperature in w1 is greater than the temperature in w2. This ensures you're selecting records where the temperature has increased from one day to the next.

### How DATEDIFF Works:

* DATEDIFF(w1.recordDate, w2.recordDate) computes the difference in days between w1.recordDate and w2.recordDate.
* When DATEDIFF(w1.recordDate, w2.recordDate) = 1, it means that w2.recordDate is exactly 1 day before w1.recordDate.

### Clarification:

* **Comparison with All Rows of w2**: Yes, this condition compares each record in w1 with every record in w2 where the date difference is exactly 1 day. Essentially, for each record in w1, it finds all records in w2 that are exactly one day earlier.
* **Temporal Comparison**: The query checks for temperature increases day by day. For each record in w1, it compares temperatures with the corresponding record(s) in w2 that are exactly one day earlier.

### Example Scenario:

Assume the Weather table has these records:

yaml

Copy code

+----+------------+-------------+

| id | recordDate | temperature |

+----+------------+-------------+

| 1 | 2023-06-15 | 25 |

| 2 | 2023-06-16 | 26 |

| 3 | 2023-06-17 | 24 |

| 4 | 2023-06-18 | 27 |

+----+------------+-------------+

* For w1, the query would first consider the record with id=2 (2023-06-16, temperature=26).
* It then finds records in w2 where recordDate is 2023-06-15 (1 day before 2023-06-16).
* If the temperature in w1 (26) is greater than the temperature in w2 (25), then w1.id (2) would be selected.

This process continues for each record in w1, comparing it with all corresponding records in w2 where the date difference is exactly 1 day.

The query effectively identifies instances where the temperature has increased from one day to the next, based on the defined criteria.

Output before where condtion

SELECT \* FROM Weather w1 JOIN Weather w2 ON DATEDIFF(w1.recordDate, w2.recordDate) = 1;

sql

Copy code

SELECT

\*

FROM

Weather w1

JOIN

Weather w2

ON

DATEDIFF(w1.recordDate, w2.recordDate) = 1;

Assuming the Weather table contains the data as follows:

yaml

Copy code

+----+------------+-------------+

| id | recordDate | temperature |

+----+------------+-------------+

| 1 | 2023-06-15 | 25 |

| 2 | 2023-06-16 | 26 |

| 3 | 2023-06-17 | 24 |

| 4 | 2023-06-18 | 27 |

+----+------------+-------------+

### Query Output:

The query performs a self-join on the Weather table (w1 and w2), joining records where the recordDate of w1 is exactly 1 day after the recordDate of w2. Here's what the output would look like:

yaml

Copy code

+------+------+------------+-------------+------+------+------------+-------------+

| id | id | recordDate | temperature | id | id | recordDate | temperature |

+------+------+------------+-------------+------+------+------------+-------------+

| 2 | 1 | 2023-06-16 | 26 | 1 | 2023-06-15 | 25 |

| 3 | 2 | 2023-06-17 | 24 | 2 | 2023-06-16 | 26 |

| 4 | 3 | 2023-06-18 | 27 | 3 | 2023-06-17 | 24 |

+------+------+------------+-------------+------+------+------------+-------------+

### Explanation of Output:

* **Row 1**: w1.id=2 (June 16, temperature 26) is joined with w2.id=1 (June 15, temperature 25).
* **Row 2**: w1.id=3 (June 17, temperature 24) is joined with w2.id=2 (June 16, temperature 26).
* **Row 3**: w1.id=4 (June 18, temperature 27) is joined with w2.id=3 (June 17, temperature 24).

Each row in the output represents a pair of records from Weather, where w1.recordDate is exactly 1 day after w2.recordDate. This is determined by the condition DATEDIFF(w1.recordDate, w2.recordDate) = 1 in the ON clause of the JOIN.

# Date\_format:

he DATE\_FORMAT function in SQL is used to format a date value according to a specified format string. This function is available in MySQL and MariaDB. In other SQL databases, you might use similar functions like TO\_CHAR in PostgreSQL and Oracle or FORMAT in SQL Server.

### Syntax:

sql

Copy code

DATE\_FORMAT(date, format)

* date: The date value to format.
* format: The format string that specifies the output format.

### Common Format Specifiers:

* %Y: Four-digit year (e.g., 2024)
* %y: Two-digit year (e.g., 24)
* %m: Two-digit month (01-12)
* %c: Month (1-12)
* %M: Full month name (January-December)
* %b: Abbreviated month name (Jan-Dec)
* %d: Day of the month with leading zero (01-31)
* %e: Day of the month without leading zero (1-31)
* %H: Hour (00-23)
* %k: Hour (0-23)
* %i: Minutes (00-59)
* %S: Seconds (00-59)
* %p: AM or PM

### Examples:

Assume you have a table Queries with a query\_date column of type DATE.

sql

Copy code

+------------+-------------------+----------+--------+------------+

| query\_name | result | position | rating | query\_date |

+------------+-------------------+----------+--------+------------+

| Dog | Golden Retriever | 1 | 5 | 2023-01-15 |

| Dog | German Shepherd | 2 | 5 | 2023-01-20 |

| Dog | Mule | 200 | 1 | 2023-02-10 |

| Cat | Shirazi | 5 | 2 | 2023-02-05 |

| Cat | Siamese | 3 | 3 | 2023-03-10 |

| Cat | Sphynx | 7 | 4 | 2023-03-15 |

+------------+-------------------+----------+--------+------------+

### Formatting Dates:

1. **Format date as YYYY-MM**:

sql

Copy code

SELECT

query\_name,

DATE\_FORMAT(query\_date, '%Y-%m') AS formatted\_date

FROM Queries;

**Output**:

sql

Copy code

+------------+---------------+

| query\_name | formatted\_date|

+------------+---------------+

| Dog | 2023-01 |

| Dog | 2023-01 |

| Dog | 2023-02 |

| Cat | 2023-02 |

| Cat | 2023-03 |

| Cat | 2023-03 |

+------------+---------------+

1. **Format date as Month Day, Year**:

sql

Copy code

SELECT

query\_name,

DATE\_FORMAT(query\_date, '%M %d, %Y') AS formatted\_date

FROM Queries;

**Output**:

sql

Copy code

+------------+--------------------+

| query\_name | formatted\_date |

+------------+--------------------+

| Dog | January 15, 2023 |

| Dog | January 20, 2023 |

| Dog | February 10, 2023 |

| Cat | February 05, 2023 |

| Cat | March 10, 2023 |

| Cat | March 15, 2023 |

+------------+--------------------+

1. **Extracting the year and month separately**:

sql

Copy code

SELECT

query\_name,

DATE\_FORMAT(query\_date, '%Y') AS year,

DATE\_FORMAT(query\_date, '%m') AS month

FROM Queries;

**Output**:

sql

Copy code

+------------+------+-------+

| query\_name | year | month |

+------------+------+-------+

| Dog | 2023 | 01 |

| Dog | 2023 | 01 |

| Dog | 2023 | 02 |

| Cat | 2023 | 02 |

| Cat | 2023 | 03 |

| Cat | 2023 | 03 |

+------------+------+-------+

1. **Grouping by month and year**:

sql

Copy code

SELECT

query\_name,

DATE\_FORMAT(query\_date, '%Y-%m') AS month\_year,

COUNT(\*) AS total\_queries

FROM Queries

GROUP BY query\_name, month\_year;

**Output**:

sql

Copy code

+------------+------------+--------------+

| query\_name | month\_year | total\_queries|

+------------+------------+--------------+

| Dog | 2023-01 | 2 |

| Dog | 2023-02 | 1 |

| Cat | 2023-02 | 1 |

| Cat | 2023-03 | 2 |

+------------+------------+--------------+

This query groups the queries by both the query\_name and the formatted month\_year, providing the total number of queries for each group.

### Summary:

The DATE\_FORMAT function is highly flexible and allows you to format date values in many different ways to suit your needs, making it easier to display, group, and analyze date-related data.

# String fuctions(final)

Certainly! Let's go through the explanations for each string function in SQL with clear examples and their immediate table outputs:

### 1. CONCAT()

**Function**: Concatenates two or more strings together.

**Syntax**: CONCAT(string1, string2, ...)

**Example and Explanation**:

sql

Copy code

SELECT CONCAT('Hello', ' ', 'World') AS concatenated\_string;

* **Explanation**: Concatenates the strings 'Hello', ' ' (space), and 'World' together.
* **Output**:

| **concatenated\_string** |
| --- |
| Hello World |

### 2. SUBSTRING()

**Function**: Extracts a substring from a string.

**Syntax**: SUBSTRING(string FROM start\_position [FOR length])

**Example and Explanation**:

sql

Copy code

SELECT SUBSTRING('Hello World', 1, 5) AS substring\_result;

* **Explanation**:
  + string: 'Hello World' - The original string from which to extract.
  + start\_position: 1 - Starting position to begin extraction (1-based index).
  + length: 5 - Number of characters to extract from the starting position.
* **Output**:

| **substring\_result** |
| --- |
| Hello |

### 3. UPPER()

**Function**: Converts all characters in a string to uppercase.

**Syntax**: UPPER(string)

**Example and Explanation**:

sql

Copy code

SELECT UPPER('hello world') AS uppercase\_string;

* **Explanation**: Converts the string 'hello world' to uppercase.
* **Output**:

| **uppercase\_string** |
| --- |
| HELLO WORLD |

### 4. LOWER()

**Function**: Converts all characters in a string to lowercase.

**Syntax**: LOWER(string)

**Example and Explanation**:

sql

Copy code

SELECT LOWER('HELLO WORLD') AS lowercase\_string;

* **Explanation**: Converts the string 'HELLO WORLD' to lowercase.
* **Output**:

| **lowercase\_string** |
| --- |
| hello world |

### 5. LENGTH()

**Function**: Returns the length of a string in characters.

**Syntax**: LENGTH(string)

**Example and Explanation**:

sql

Copy code

SELECT LENGTH('Hello World') AS string\_length;

* **Explanation**: Returns the number of characters in the string 'Hello World'.
* **Output**:

| **string\_length** |
| --- |
| 11 |

### 6. TRIM()

**Function**: Removes leading and trailing spaces from a string.

**Syntax**: TRIM([LEADING | TRAILING | BOTH] trim\_character FROM string)

**Example and Explanation**:

sql

Copy code

SELECT TRIM(' Hello World ') AS trimmed\_string;

* **Explanation**: Removes leading and trailing spaces from the string ' Hello World '.
* **Output**:

| **trimmed\_string** |
| --- |
| Hello World |

### 7. REPLACE()

**Function**: Replaces occurrences of a substring within a string.

**Syntax**: REPLACE(string, search\_string, replacement\_string)

**Example and Explanation**:

sql

Copy code

SELECT REPLACE('Hello World', 'World', 'SQL') AS replaced\_string;

* **Explanation**: Replaces occurrences of 'World' with 'SQL' in the string 'Hello World'.
* **Output**:

| **replaced\_string** |
| --- |
| Hello SQL |

### 8. LEFT()

**Function**: Extracts a specified number of characters from the left of a string.

**Syntax**: LEFT(string, length)

**Example and Explanation**:

sql

Copy code

SELECT LEFT('Hello World', 5) AS left\_string;

* **Explanation**: Extracts the first 5 characters from the left of the string 'Hello World'.
* **Output**:

| **left\_string** |
| --- |
| Hello |

### 9. RIGHT()

**Function**: Extracts a specified number of characters from the right of a string.

**Syntax**: RIGHT(string, length)

**Example and Explanation**:

sql

Copy code

SELECT RIGHT('Hello World', 5) AS right\_string;

* **Explanation**: Extracts the last 5 characters from the right of the string 'Hello World'.
* **Output**:

| **right\_string** |
| --- |
| World |

### 10. POSITION() or LOCATE()

**Function**: Returns the position of a substring within a string.

**Syntax**: POSITION(substring IN string) or LOCATE(substring, string)

**Example and Explanation**:

sql

Copy code

SELECT POSITION('l' IN 'Hello World') AS position\_result;

* **Explanation**: Returns the position of the first occurrence of 'l' in 'Hello World'.
* **Output**:

| **position\_result** |
| --- |
| 3 |

### Immediate Table Output

Here's the combined table showing the results of each example query:

| **concatenated\_string** | **substring\_result** | **uppercase\_string** | **lowercase\_string** | **string\_length** | **trimmed\_string** | **replaced\_string** | **left\_string** | **right\_string** | **position\_result** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hello World | Hello | HELLO WORLD | hello world | 11 | Hello World | Hello SQL | Hello | World | 3 |

Each function provides different capabilities for manipulating and extracting information from strings in SQL queries. Adjust the examples and parameters according to your specific SQL dialect and requirements for accurate results.

3.5

what if i want only 1st letter uppercase and all lower cae

ChatGPT

To achieve capitalizing only the first letter of a string

# Subqueries:

Type of subqueries:

### Example Tables

1. **employees**

diff

Copy code

+----+--------+---------+---------+

| id | name | dept\_id | salary |

+----+--------+---------+---------+

| 1 | John | 1 | 50000 |

| 2 | Alice | 2 | 60000 |

| 3 | Bob | 1 | 55000 |

| 4 | Carol | 3 | 70000 |

+----+--------+---------+---------+

1. **departments**

diff

Copy code

+---------+-------------+

| dept\_id | dept\_name |

+---------+-------------+

| 1 | HR |

| 2 | IT |

| 3 | Marketing |

+---------+-------------+

### Scalar Subqueries

A scalar subquery returns a single value. For example, to find employees whose salary is greater than the average salary in the company:

sql

Copy code

SELECT name, salary

FROM employees

WHERE salary > (SELECT AVG(salary) FROM employees);

**Explanation**: The subquery (SELECT AVG(salary) FROM employees) returns the average salary, and the outer query selects employees with a salary greater than this average.

### Row Subqueries

A row subquery returns a single row with multiple columns. For example, to find the employee with the highest salary in each department:

sql

Copy code

SELECT e1.name, e1.salary

FROM employees e1

WHERE (e1.salary, e1.dept\_id) IN

(SELECT MAX(e2.salary), e2.dept\_id

FROM employees e2

GROUP BY e2.dept\_id);

**Explanation**: The subquery (SELECT MAX(e2.salary), e2.dept\_id FROM employees e2 GROUP BY e2.dept\_id) returns the highest salary and corresponding department ID for each department. The outer query matches these values to find the respective employees.

### Column Subqueries

A column subquery returns a single column from multiple rows. For example, to find employees who work in the 'IT' department:

sql

Copy code

SELECT name

FROM employees

WHERE dept\_id IN (SELECT dept\_id FROM departments WHERE dept\_name = 'IT');

**Explanation**: The subquery (SELECT dept\_id FROM departments WHERE dept\_name = 'IT') returns the department IDs of all 'IT' departments. The outer query selects employees whose dept\_id is in this list.

### Table Subqueries

A table subquery is used in the FROM clause to create a derived table. For example, to find the average salary per department and list those departments with an average salary greater than 55000:

sql

Copy code

SELECT dept\_name, avg\_salary

FROM

(SELECT d.dept\_name, AVG(e.salary) AS avg\_salary

FROM employees e

JOIN departments d ON e.dept\_id = d.dept\_id

GROUP BY d.dept\_name) AS dept\_avg

WHERE avg\_salary > 55000;

**Explanation**: The subquery (SELECT d.dept\_name, AVG(e.salary) AS avg\_salary FROM employees e JOIN departments d ON e.dept\_id = d.dept\_id GROUP BY d.dept\_name) calculates the average salary for each department and returns a table. The outer query filters this table to include only departments with an average salary greater than 55000.

### Summary

* **Scalar Subqueries**: Return a single value. Used in places where a single value is expected.
* **Row Subqueries**: Return a single row with multiple columns. Used in places where multiple column values are expected.
* **Column Subqueries**: Return a single column from multiple rows. Used where a single column list is required.
* **Table Subqueries**: Return a complete table. Used in the FROM clause to create derived tables for complex queries.

These examples demonstrate how subqueries can be effectively utilized to perform a variety of tasks in SQL.