**What is a database**

A database is an organized collection of structured data stored electronically in a computer system.

When the computer was first invented, it was mainly used for scientific researches to perform calculations quickly.

Since the computer was adopted more and more, the requirements were also increased to require the computer to store a larger volume of data for fast retrieval.

Before the database system was invented, the flat file structure was commonly used to store data. For example, here is the comma-separated value (CSV) file that stores employee information:

first name, last name, phone

John, Doe, (408)-245-2345

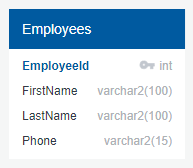
Jane, Doe, (503)-234-2355

...

The CSV file has three columns which are known as fields and rows which are known as records. When the number of rows in the flat file is increased e.g., million rows, it becomes unmanageable.

In the 1970s, Dr. Ted Codd, a computer scientist, invented the relational model for database management. The relational model deals with many issues caused by the flat file model. According to his model, data is organized in entities and attributes, instead of combining everything in a single structure.

An entity is a person, place, or thing and attributes describe the person, place, and thing. For example, you can use the relational model to organize the employee information into an employee entity with the attributes: first name, last name, and phone:



**What is DBMS**

**Database Management System**

The software which is used to manage databases is called Database Management System (DBMS). For Example, MySQL, Oracle, etc. are popular commercial DBMS used in different applications. DBMS allows users the following tasks:

* **Data Definition:** It helps in the creation, modification, and removal of definitions that define the organization of data in the database.
* **Data Updation:** It helps in the insertion, modification, and deletion of the actual data in the database.
* **Data Retrieval:** It helps in the retrieval of data from the database which can be used by applications for various purposes.
* **User Administration:** It helps in registering and monitoring users, enforcing data security, monitoring performance, maintaining data integrity, dealing with concurrency control, and recovering information corrupted by unexpected failure.

DBMS provides an environment to store and retrieve the data in convenient and efficient manner.

Key Features of DBMS

* **Data modeling:** A DBMS provides tools for creating and modifying data models, which define the structure and relationships of the data in a database.
* **Data storage and retrieval:**A DBMS is responsible for storing and retrieving data from the database, and can provide various methods for searching and querying the data.
* **Concurrency control:** A DBMS provides mechanisms for controlling concurrent access to the database, to ensure that multiple users can access the data without conflicting with each other.
* **Data integrity and security:** A DBMS provides tools for enforcing data integrity and security constraints, such as constraints on the values of data and access controls that restrict who can access the data.
* **Backup and recovery:** A DBMS provides mechanisms for backing up and recovering the data in the event of a system failure.
* **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.

A database is a collection of interrelated data which helps in the efficient retrieval, insertion, and deletion of data from the database and organizes the data in the form of tables, views, schemas, reports, etc. For Example, a university database organizes the data about students, faculty, admin staff, etc. which helps in the efficient retrieval, insertion, and deletion of data from it.

**Database Languages**

* Data Definition Language
* Data Manipulation Language
* Data Control Language
* Transactional Control Language

**Data Definition Language**

**DDL** is the short name for Data Definition Language, which deals with database schemas and descriptions, of how the data should reside in the database.

* **CREATE:** to create a database and its objects like (table, index, views, store procedure, function, and triggers)
* **ALTER:**alters the structure of the existing database
* **DROP:** delete objects from the database
* **TRUNCATE:** remove all records from a table, including all spaces allocated for the records are removed
* **COMMENT:**add comments to the data dictionary
* **RENAME:** rename an object

**Data Manipulation Language**

**DML** is the short name for Data Manipulation Language which deals with data manipulation and includes most common SQL statements such SELECT, INSERT, UPDATE, DELETE, etc., and it is used to store, modify, retrieve, delete and update data in a database. **Data query language(DQL)** is the subset of “Data Manipulation Language”. The most common command of DQL is **SELECT** statement. SELECT statement help on retrieving the data from the table without changing anything in the table.

* **SELECT:**retrieve data from a database
* **INSERT:** insert data into a table
* **UPDATE:** updates existing data within a table
* **DELETE:**Delete all records from a database table
* **MERGE:** UPSERT operation (insert or update)
* **CALL:**call a PL/SQL or Java subprogram
* **EXPLAIN PLAN:** interpretation of the data access path
* **LOCK TABLE:**concurrency Control

**Data Control Language**

**DCL**is short for Data Control Language which acts as an access specifier to the database.(basically to grant and revoke permissions to users in the database

* **GRANT:**grant permissions to the user for running DML(SELECT, INSERT, DELETE,…) commands on the table
* **REVOKE:**revoke permissions to the user for running DML(SELECT, INSERT, DELETE,…) command on the specified table

**Transactional Control Language**

**TCL** is short for Transactional Control Language which acts as an manager for all types of transactional data and all transactions. Some of the command of TCL are

* **Roll Back:** Used to cancel  or Undo changes made in the database
* **Commit:** It is used to apply or save changes in the database
* **Save Point:** It is used to save the data on the temporary basis in the database

**Data Query Language (DQL):**

**Data query language(DQL)** is the subset of **“Data Manipulation Language”**. The most common command of DQL is 1the **SELECT statement**. SELECT statement helps us in retrieving the data from the table without changing anything or modifying the table. DQL is very important for retrieval of essential data from a database.

**What is a Database Table?**

A table is a collection of related data entries, and it consists of columns and rows.

A column holds specific information about every record in the table.

A record (or row) is each individual entry that exists in a table.

Look at a selection from the "Customers" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
|  |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 5021 | Mexico |  |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 5023 | Mexico |  |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |  |
|  |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |  |

The columns in the "Customers" table above are: CustomerID, CustomerName, ContactName, Address, City, PostalCode and Country. The table has 5 records (rows).

**What is a Relational Database?**

A relational database defines database relationships in the form of tables. The tables are related to each other - based on data common to each.

Look at the following three tables "Customers", "Orders", and "Shippers" from the database:

**Customers Table**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
|  |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 5021 | Mexico |  |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 5023 | Mexico |  |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |  |
|  |
| 5 | kumar | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |  |

The relationship between the "Customers" table and the "Orders" table is the CustomerID column:

**Orders Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **EmployeeID** | **OrderDate** | **ShipperID** |
| 10278 | 5 | 8 | 12-08-1996 | 2 |
|  |
| 10280 | 5 | 2 | 14-08-1996 | 1 |  |
| 10308 | 2 | 7 | 18-09-1996 | 3 |  |
| 10355 | 4 | 6 | 15-11-1996 | 1 |  |
|  |
| 10365 | 3 | 3 | 27-11-1996 | 2 |  |
| 10383 | 4 | 8 | 16-12-1996 | 3 |  |
| 10384 | 5 | 3 | 16-12-1996 | 3 |  |

The relationship between the "Orders" table and the "Shippers" table is the ShipperID column:

**Shippers Table**

|  |  |  |
| --- | --- | --- |
| **ShipperID** | **ShipperName** | **Phone** |
| 1 | Speedy Express | (503) 555-9831 |
| 2 | United Package | (503) 555-3199 |
| 3 | Federal Shipping | (503) 555-9931 |

**What is SQL?**

SQL is the standard language for dealing with Relational Databases.

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

**Syntax rules:**

* SQL keywords are NOT case sensitive: select is the same as SELECT

For Best practice we will write all SQL keywords in upper-case.

Semicolon after SQL Statements?

* Some database systems require a semicolon at the end of each SQL statement.
* Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server.

For best practice we will use semicolon at the end of each SQL statement.

Some of The Most Important SQL Commands

* **SELECT** - extracts data from a database
* **UPDATE** - updates data in a database
* **DELETE** - deletes data from a database
* **INSERT INTO** - inserts new data into a database
* **CREATE DATABASE** - creates a new database
* **ALTER DATABASE** - modifies a database
* **CREATE TABLE** - creates a new table
* **ALTER TABLE** - modifies a table
* **DROP TABLE** - deletes a table
* **CREATE INDEX** - creates an index (search key)
* **DROP INDEX** - deletes an index

**Basics:**

USE sql\_store;

SELECT \*

FROM customers

WHERE state = ‘CA’

ORDER BY first\_name

LIMIT 3;

• SQL is **not** a case-sensitive language.

• In MySQL, every statement must be terminated with a semicolon.

**MySQL SELECT Statement**

The MySQL SELECT Statement

* The SELECT statement is used to select data from a database.
* The data returned is stored in a result table, called the result-set.

SELECT Syntax

*SELECT column1, column2, ...  
FROM table\_name;*

Here, column1, column2, ... are the field names of the table you want to select data from. If you want to select all the fields available in the table, use the following syntax:

*SELECT \* FROM table\_name;*

SELECT \* FROM Customers;

Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

**SELECT Columns Example**

The following SQL statement selects the "CustomerName", "City", and "Country" columns from the "Customers" table:

SELECT CustomerName, City, Country FROM Customers;  
  
Selecting all columns in million records is a performance & network load on DB

SELECT first\_name, second\_name, points  
FROM customers

**The MySQL WHERE Clause**

The WHERE clause is used to filter records.

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

WHERE Syntax

SELECT *column1*,*column2, ...*  
FROM *table\_name*  
WHERE *condition*;  
  
  
Example:  
  
SELECT \*

FROM Customer

WHERE state = ‘va’

**Text Fields vs. Numeric Fields**

SQL requires single quotes around text values (most database systems will also allow double quotes).

However, numeric fields should not be enclosed in quotes:

SELECT \*

FROM Customer

WHERE points = 2273

SELECT \*

FROM Customer

WHERE state = ‘va’

**Operators in The WHERE Clause**

The following operators can be used in the WHERE clause:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Equal | [Try it](https://www.w3schools.com/mysql/trymysql.asp?filename=trysql_op_equal_to) |
| > | Greater than | [Try it](https://www.w3schools.com/mysql/trymysql.asp?filename=trysql_op_greater_than) |
| < | Less than | [Try it](https://www.w3schools.com/mysql/trymysql.asp?filename=trysql_op_less_than) |
| >= | Greater than or equal | [Try it](https://www.w3schools.com/mysql/trymysql.asp?filename=trysql_op_greater_than2) |
| <= | Less than or equal | [Try it](https://www.w3schools.com/mysql/trymysql.asp?filename=trysql_op_less_than2) |
| <> | Not equal. **Note:** In some versions of SQL this operator may be written as != | [Try it](https://www.w3schools.com/mysql/trymysql.asp?filename=trysql_op_not_equal_to) |
| BETWEEN | Between a certain range | [Try it](https://www.w3schools.com/mysql/trymysql.asp?filename=trysql_op_between) |
| LIKE | Search for a pattern | [Try it](https://www.w3schools.com/mysql/trymysql.asp?filename=trysql_op_like) |
| IN | To specify multiple possible values for a column |  |

SELECT \*

FROM Customer

WHERE birth\_date  > ‘1990-01-01’  
 **Equal:**  
  
SELECT \*

FROM Customer

WHERE state = ‘va’

**Not Equal:**

SELECT \*

FROM Customer

WHERE state ! = ‘va’  
Same as above

SELECT \*

FROM Customer

WHERE state <> ‘va’

**Between:**  
SELECT \* FROM Products

WHERE unit\_price BETWEEN 1.21 AND 2.53;  
  
**LIKE:**% represent any number of characters  
\_ reperstenst single chrcter

SELECT \* FROM Customers

WHERE state LIKE ‘s%’;  
  
Example:  
Let’s search for customer whose last name starts with letter “b”.  
If you place b after % then it will search for customers names ends with “b”.  
  
SELECT \*

FROM customers

WHERE last\_name LIKE 'b%'   
  
  
Let’s search for customer whose last name starts with word “brush” .  
SELECT \*

FROM customers

WHERE last\_name LIKE 'brush%'  
  
Let’s search for customer whose last name contains “b” letter anyware like fornt back or middle.  
  
SELECT \*

FROM customers

WHERE last\_name LIKE '%B%'  
  
  
Now let’s discuss “\_“ symbol which is equal to single character.  
  
Let’s search for customer whose last name ends with y and word is 6 characters long.  
SELECT \*

FROM customers

WHERE last\_name LIKE '\_\_\_\_\_y'

Let’s search for customer whose first name starts with ‘b’ and last name ends with y with 6 characters long.  
  
SELECT \*

FROM customers

WHERE last\_name LIKE 'b\_\_\_\_y'  
  
Conditional operators with Like.  
  
SELECT \*

FROM customers

WHERE address LIKE '%trail%' OR address LIKE '%avenue%'  
  
Customers with phone number ends with 9  
  
SELECT \*

FROM customers

WHERE phone LIKE '%9'

With NOT:

SELECT \*

FROM customers

WHERE phone NOT LIKE '%9'

**IN**:  
SELECT \* FROM Customers WHERE state IN ('Orlando','Hampton');

**The MySQL ORDER BY Keyword**

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

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

ORDER BY Syntax

SELECT *column1*,*column2, ...*  
FROM *table\_name*  
ORDER BY *column1, coulmn2,…*ASC|DESC;  
  
**Example:**  
  
SELECT \* FROM Customers

ORDER BY first\_name

SELECT \* FROM Customers

ORDER BY first\_name DESC;

Exercise:  
Get the orders placed this year  
  
SELECT \*

FROM orders

WHERE order\_date  > = ‘2024-01-01’  
  
  
here based on state and more than one customer has same state then second column order will execute.  
SELECT \*

FROM customers

order by state, first\_name  
  
SELECT \*

FROM customers

order by state DESC, first\_name ASC  
  
  
Lets trt to fetch 2 order with new total price column as alias with unit \* unit price in ascending order.  
  
SELECT \* , quantity \* unit\_price AS total\_price

FROM order\_items

WHERE order\_id = 2 ORDER BY total\_price ASC

**The MySQL AND, OR and NOT Operators**

The WHERE clause can be combined with AND, OR, and NOT operators.

The AND and OR operators are used to filter records based on more than one condition:

* The AND operator displays a record if all the conditions separated by AND are TRUE.
* The OR operator displays a record if any of the conditions separated by OR is TRUE.

Combine multiple search conditions when filtering data:

AND, OR and NOT Operator.  
  
Need customers details born after 1990 and points greater than 1000 points  
  
SELECT \*

FROM customers

WHERE birth\_date  > = ‘1990-01-01’ AND points  > 1000  
  
SELECT \*

FROM orders

WHERE order\_date  > = ‘1990-01-01’ OR points  > 1000  
  
Need customers details born after 1990 or point greater than 1000 points and lives in Virginia.  
  
SELECT \*

FROM customers

WHERE birth\_date  > ‘1990-01-01’ OR points  > 1000 AND state = ‘VA’  
  
when combining multiple logical operators we need to aware order of this operators.

*The general order of operator precedence in SQL is as follows:*

1. Parentheses
2. Unary operators (e.g., NOT)
3. Division, Multiplication and modulus operators
4. Addition and subtraction operators
5. Comparison operators (e.g., <, >, =)
6. Logical operators (e.g., AND, OR)

SELECT \*

FROM customers

WHERE birth\_date  >  ‘1990-01-01’ OR

      (points  > 1000 AND site = ‘VA’)  
  
Customers born after 1990 OR points greater than 1000

SELECT \*

FROM customers

WHERE birth\_date  >  ‘1990-01-01’ OR points  > 1000  
  
Result

1,3,5  
  
NOT operator to negate the above condition:  
born before 1990 and less than 1000 points

SELECT \*

FROM customers

WHERE NOT (birth\_date  >  ‘1990-01-01’ OR points  > 1000)  
  
Result  
2,4,10  
  
Same query without not  
  
SELECT \*

FROM customers

WHERE birth\_date  <=  ‘1990-01-01’ AND points  <= 1000)  
  
 **Regular expressions (Regexp):**MySQL supports another type of pattern matching operation based on the regular expressions and the REGEXP operator.  
A regular expression is a sequence of characters that forms a search pattern.

1. It provide a powerful and flexible pattern match that can help us implement power search utilities for our database systems.  
     
   Example below query will retrive customer details whose last name contains “field” string with like operator  
     
   SELECT \*

FROM customers

WHERE last\_name LIKE '%field%'

Using REGEXP we can achieve this functionality without % symbol.  
  
SELECT \*

FROM customers

WHERE last\_name REGEXP 'field'  
  
  
We can use ^ carrot symbol to indicate beginning of the string.  
  
SELECT \*

FROM customers

WHERE last\_name REGEXP '^field'  
  
no customers starts with ‘field’.  
  
We can use $ dollar symbol to indicate end of the string.

SELECT \*

FROM customers

WHERE last\_name REGEXP 'field$'  
  
Lets try to use pipe symbol” |” or vertical bar which represents multiple search patterns.  
  
SELECT \*

FROM customers

WHERE last\_name REGEXP 'rose|field|mac'

Lets combine multiple search pattens.  
  
Here we have added carrot symbol before field and it should check for lastname starts with filed. It skips brushfield row since the last name starts with brush string not with filed.  
  
SELECT \*

FROM customers

WHERE last\_name REGEXP 'rose|^field|mac'  
  
try with $ symbol.  
  
SELECT \*

FROM customers

WHERE last\_name REGEXP 'rose|field$|mac'  
  
Lets try to learn brackets symbol. Its serach for customers who have ‘e’ in there last name.  
  
SELECT \*

FROM customers

WHERE last\_name REGEXP 'e'  
  
Now lets make sure before ‘e’ we should either have letter ‘g’ or an ‘i’. here we use square brackets.   
  
  
SELECT \*

FROM customers

WHERE last\_name REGEXP '[gim]e'  
  
here it will search for   
ge  
ie  
me  
Lets try range in brackets as shown below. Here we are trying combinations from a alphabet to h ex: ae,be,ce,----- he  
  
SELECT \*

FROM customers

WHERE last\_name REGEXP '[a-h]e'  
  
  
SELECT \*

FROM customers

WHERE last\_name REGEXP '[a-z]e'  
  
  
EXERCISE:

Customer first\_name ELKA or AMBUR  
Customers last\_name ends with EY or ON  
Customers last\_name starts with MY or contains SE  
Customers last\_name contains B followed by R or U  
  
SELECT \*

FROM customers

WHERE last\_name REGEXP 'ELKA|AMBUR'  
  
  
  
SELECT \*

FROM customers

WHERE last\_name REGEXP 'ey$|on$'  
  
SELECT \*

FROM customers

WHERE last\_name REGEXP '^my|se'  
  
  
SELECT \*

FROM customers

WHERE last\_name REGEXP 'b[ru]'  
  
else  
  
SELECT \*

FROM customers

WHERE last\_name REGEXP 'br|bu'

**NULL Operator**  
Null operator is used to check records with attribute null.  
Null means absence of a value.  
  
If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.  
  
IS NULL Syntax

SELECT *column\_names*FROM *table\_name*  
WHERE *column\_name* IS NULL;

IS NOT NULL Syntax

SELECT *column\_names*FROM *table\_name*  
WHERE *column\_name* IS NOT NULL;

Now lets search for customers whose phone number was not set.  
  
SELECT \*

FROM customers

WHERE phone IS NULL  
  
Not null returns customers with phone number  
  
SELECT \*

FROM customers

WHERE phone IS NOT NULL  
  
Exercise:

Need orders that are not shipped.  
  
  
  
SELECT \*

FROM orders

WHERE shipper\_id IS NULL  
  
OR  
  
SELECT \*

FROM orders

WHERE shipped\_date IS NULL  
  
NOTE:  
  
In MySQL, there isn't a built-in function to check all columns for NULL values without explicitly specifying each column. However, you can work around this limitation by using dynamic SQL or writing a script in a language like Python, PHP, or even a MySQL stored procedure to generate the query for you.  
  
**AND \OR in NULL**  
  
  
The given query is not value. Comments is string and shipper\_id is integer and bitwise and is not possible between string and number. So it will give unexpected result.  
  
SELECT \*

FROM orders

WHERE (shipper\_id AND comments) IS NULL;

The SQL engine tries to evaluate shipper\_id AND comments as a bitwise AND operation and then checks if the result is NULL, which is not the intended logic. You want to check if either shipper\_id or comments is NULL, which requires using separate IS NULL checks with OR:

sql

SELECT \*

FROM orders

WHERE shipper\_id IS NULL

OR comments IS NULL;

**Example: Bitwise AND with 12 and 10**

Let’s take the numbers 12 and 10:

* **12 in binary**: 1100
* **10 in binary**: 1010

**Bitwise AND Operation:**

sql

1100 (12 in binary)

& 1010 (10 in binary)

------

1000 (8 in binary)

**Explanation:**

* The bitwise AND compares each bit of the numbers:
  + The first bit of 12 (1) and the first bit of 10 (1) result in 1.
  + The second bit of 12 (1) and the second bit of 10 (0) result in 0.
  + The third bit of 12 (0) and the third bit of 10 (1) result in 0.
  + The fourth bit of 12 (0) and the fourth bit of 10 (0) result in 0.
* The result of 1100 AND 1010 is 1000, which is 8 in decimal.

**Summary:**

* **12 in decimal**: 1100 in binary
* **10 in decimal**: 1010 in binary
* **Result of 12 AND 10**: 1000 in binary, which is 8 in decimal

This operation is useful in situations like setting or checking specific bits in flags or permissions, where each bit represents an on/off state (like whether a permission is granted or not).

**LIMIT**The LIMIT clause is used to specify the number of records to return.

The LIMIT clause is useful on large tables with thousands of records. Returning a large number of records can impact performance.

LIMIT Syntax

SELECT *column\_name(s)*  
FROM *table\_name*WHERE *condition*  
LIMIT *number*;  
  
  
  
  
SELECT \*

FROM customers

LIMIT 3

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

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

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

Example

SELECT \* FROM Customers  
LIMIT 3 OFFSET 3;  
  
Without offset keyword it will take offset as first and limit second parameter  
SELECT \* FROM Customers

LIMIT 2,2;  
  
  
Exercise:  
  
Select top 3 Loyal customers.  
  
  
  
SELECT \*

FROM Customers

ORDER BY points DESC

LIMIT 3  
  
  
  
  
  
**MySQL Joins**

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.  
  
  
Supported Types of Joins in MySQL

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

**MySQL INNER JOIN Keyword**

The INNER JOIN keyword selects records that have matching values in both tables.



INNER JOIN Syntax

SELECT *column\_name(s)*  
FROM *table1*  
INNER JOIN *table2*ON *table1.column\_name*=*table2.column\_name*;

Now I want to join both the tables.We can ue customerID which is common in both the tables.

Let's look at a selection from the "Orders" table:

|  |  |  |
| --- | --- | --- |
| **OrderID** | **CustomerID** | **OrderDate** |
| 10308 | 2 | 1996-09-18 |
| 10309 | 37 | 1996-09-19 |
| 10310 | 77 | 1996-09-20 |

Then, look at a selection from the "Customers" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mexico |

Notice that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.

Then, we can create the following SQL statement (that contains an INNER JOIN), that selects records that have matching values in both tables:  
  
SELECT \*

FROM orders

INNER JOIN customers ON orders.customer\_id = customers.customer\_id  
  
Now simply with order id , first name and last name  
  
SELECT order\_id, first\_name, last\_name

FROM orders

JOIN customers ON orders.customer\_id = customers.customer\_id  
  
Lets print customer ID also   
  
SELECT order\_id, customer\_id, first\_name, last\_name

FROM orders

JOIN customers ON orders.customer\_id = customers.customer\_id  
  
But here we will get error saying customer\_id is ambigious. Because it is there in two tables. So we need to mention explicitly from which table it should come.  
  
SELECT order\_id, orders.customer\_id, first\_name, last\_name

FROM orders

JOIN customers ON orders.customer\_id = customers.customer\_id  
  
If you see here we have repeated the word order in multiple times. Same for customer word also. We can get rid of this repetition by using alias.  
  
SELECT order\_id, o.customer\_id, first\_name, last\_name

FROM orders o

JOIN customers c ON o.customer\_id = c.customer\_id  
  
  
Exercise:

I want you to look at the order items table. So, in this table we have these columns, order ID, column ID, product id, quantity, and unit price. Now I want you to write a query and join this table with the products table so for each order return both the product id as well as this name, followed by the quantity, and the unit price form the order items table. And by the way make sure to use an alias to simplify your code,  
  
here unit\_price is from order\_item table because unit price will change in products. We need to select from order\_item table(captured at the time of purchase)

SELECT order\_id, oi.product\_id, name, quantity, oi.unit\_price

FROM order\_items oi

JOIN products p ON oi.product\_id = p.product\_id  
  
  
**Join across databases**In real world as a DBA we ned to deal with multiple databases. SO here we need to combine columns from tables in multiple databases  
  
Imagine products table is in different database.  
Now join both the tables in different databases.  
  
Prefix the tables that are not present in the database.  
  
SELECT \*

FROM order\_items oi

JOIN sql\_inventory.products p ON oi.product\_id = p.product\_id

**Joining multiple tables**

For example, back to our SQL database, look at the orders table, now you know how to write a query to join this table with the customers table to return information about the customer to place each order. But here we also have another column, status, which is similar to the customer ID status. So the name of the status's are not stored in this table, they are somewhere else in the order status's table. Let's have a quick look here.

Our orders can be either processed, shipped or delivered. And these are the identifiers for each of these status's. Now back to our orders table. In the status column you store status id. So now we should write a query to join the orders table, be two tables. The customers table and orders status's table.  
  
-- SELECT \* FROM sql\_store.orders;  
  
SELECT o.order\_id, o.order\_date, c.first\_name, os.name AS status

FROM orders o

JOIN customers c ON o.customer\_id = c.customer\_id

JOIN order\_statuses os ON o.status = os.order\_status\_id  
  
  
  
Exercise:  
  
Alright, for exercise, take a look at the SQL invoicing database. Here we have this table, payments, and these are the payments that each client has made towards either invoice

Let's take a look at the data, so we have these columns, like client id, that identifies the client, so we can join this table with the client's table to see the name of the clients. Next we have invoice ID, we also have date, payment method. So similarly we can join this table with the payment method table here, let's have a look at the data in this table, these are the payment methods, credit card, cash, PayPal, wire transfer. So, back to the payments table, I want you to write a query and join this table with the payment methods table as well as the client's table.

Produce a report that shows the payments, with more details, such as the name of the client, and the payment method.  
  
-- SELECT \* FROM sql\_invoicing.payments;

SELECT c.client\_id, c.name, p.amount, p.date, p.invoice\_id, pm.name as payment\_type

FROM payments p

JOIN clients c ON p.client\_id = c.client\_id

JOIN payment\_methods pm ON p.payment\_method = pm.payment\_method\_id

**Self Join**MySQL Self Join

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

Self Join Syntax

SELECT *column\_name(s)*  
FROM *table1 T1, table1 T2*  
WHERE *condition*;  
  
  
There you go. So here we have these columns, employee id, first name, last name, salary and reports 2. This is the id of the manager for this person or this employee. Now once again, you don't want to repeat the managers information here, like the phone number, the address because this information can change in the future, so we are only using your identifier or their id to refer to them in this table, now where can we find information about this This manager is actually an employee of the same organization, so, look at this example, the manager ID is 37 370. Now if you look on the website, here is the ID of that manager which is another employee. That was the manager for this employee. We don't have any values here so the value for the sale is null. So this employee doesn't have a manager and that means they are the CEO. So let's go ahead and write a query to join this table with itself so we can select the name of each employee and their manager.

USE sql\_hr;

SELECT e.first\_name, e.last\_name, m.first\_name AS manager

FROM employees e

JOIN employees m ON e.reports\_to = m.employee\_id

**MySQL LEFT JOIN Keyword**

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



LEFT JOIN Syntax

SELECT *column\_name(s)*  
FROM *table1*  
LEFT JOIN *table2*ON *table1.column\_name*=*table2.column\_name*;

**Note:** The LEFT JOIN keyword returns all records from the left table(paymnets) , even if there are no matches in the right table(clients) .  
  
USE sql\_invoicing;

SELECT \*

FROM payments

LEFT JOIN clients ON payments.client\_id = clients.client\_id ORDER BY payments.payment\_id  
 **MySQL RIGHT JOIN Keyword**

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



RIGHT JOIN Syntax

SELECT *column\_name(s)*  
FROM *table1*  
RIGHT JOIN *table2*ON *table1.column\_name*=*table2.column\_name*;

USE sql\_invoicing;

SELECT \*

FROM payments

RIGHT JOIN clients ON payments.client\_id = clients.client\_id ORDER BY payments.payment\_id  
  
 **SQL CROSS JOIN Keyword**

The CROSS JOIN keyword returns all records from both tables (table1 and table2).



**Note:** CROSS JOIN can potentially return very large result-sets!  
  
CROSS JOIN Syntax

SELECT *column\_name(s)*  
FROM *table1*  
CROSS JOIN *table2*;

**Note:** CROSS JOIN can potentially return very large result-sets!

The CROSS JOIN keyword returns all matching records from both tables whether the other table matches or not. So, if there are rows in "Customers" that do not have matches in "Orders", or if there are rows in "Orders" that do not have matches in "Customers", those rows will be listed as well.  
  
SELECT customers.first\_name,customers.customer\_id, orders.order\_id

FROM customers

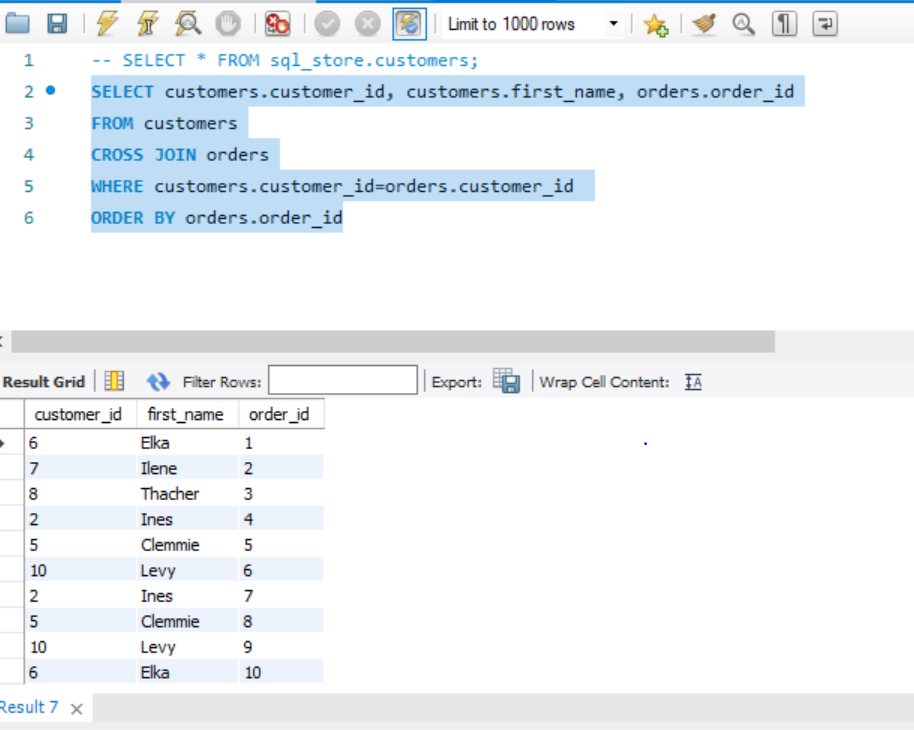
CROSS JOIN orders ORDER BY orders.order\_id  
  
A screenshot of a computer

Description automatically generated  
  
  
  
If you add a WHERE clause (if table1 and table2 has a relationship), the CROSS JOIN will produce the same result as the INNER JOIN clause:  
  
SELECT customers.customer\_id, customers.first\_name, orders.order\_id

FROM customers

CROSS JOIN orders

WHERE customers.customer\_id=orders.customer\_id

ORDER BY orders.order\_id  
  
  
  
  


**The MySQL UNION Operator**

IN joins we can combine columns with multiple tables. In sql we can also combine rows with multiple tables.  
  
The UNION operator is used to combine the result-set of two or more SELECT statements.

* Every SELECT statement within UNION must have the same number of columns
* The columns must also have similar data types
* The columns in every SELECT statement must also be in the same order

UNION Syntax

SELECT *column\_name(s)* FROM *table1*  
UNION  
SELECT *column\_name(s)* FROM *table2*;  
  
  
  
  
  
  
  
  
SELECT first\_name

FROM customers

UNION

SELECT name

FROM shippers  
  
First we have a quick look at our orders table, and select everything from the order table. Now if you look at the data, we can see that the order was placed in the current year, 2024, all the other orders were placed in previous years. Now let's say you want to create a report, get all the orders, and next to each order, add a label. If the order is placed in the current year, the label is going to be active, and if the order is placed in previous years, you want to label it as archives.  
  
  
Orders place in this year.  
  
SELECT \*

FROM orders

WHERE order\_date >= '2024-01-01'  
  
  
Specific columns for clear understanding we will add string ‘Active’ in last column.  
  
SELECT order\_id,

Order\_date,

"Active"

FROM orders

WHERE order\_date >= '2024-01-01'  
  
change “active ” string as Status.  
  
SELECT order\_id,

Order\_date,

'Active' AS Status

FROM orders

WHERE order\_date >= '2024-01-01'  
  
  
Now retrive ordes olderthan current year.  
  
SELECT order\_id,

Order\_date,

‘Archived’ AS Status

FROM orders

WHERE order\_date < '2024-01-01'  
  
NOW using Union WE will combine both select statements  
  
SELECT order\_id,

Order\_date,

'Active' AS Status

FROM orders

WHERE order\_date >= '2024-01-01'

UNION

SELECT order\_id,

Order\_date,

'Archived' AS Status

FROM orders

WHERE order\_date < '2024-01-01'  
  
  
Exercise:  
  
In customer table using union get customer details based on points. If points less than 2000 means bronze type , points less between 2000 and 3000 means silver and points greater than 3000 gold.  
  
Bronz  
  
SELECT customer\_id, first\_name, points, 'Bronz' AS Type

FROM customers

WHERE points <2000  
  
Silver  
  
SELECT customer\_id, first\_name, points, 'Silver' AS Type

FROM customers

WHERE points BETWEEN 2000 AND 3000  
  
  
Gold  
  
SELECT customer\_id, first\_name, points, 'Gold' AS Type

FROM customers

WHERE points > 3000  
  
  
With Union:

SELECT customer\_id, first\_name, points, 'Bronz' AS Type

FROM customers

WHERE points <2000

UNION

SELECT customer\_id, first\_name, points, 'Silver' AS Type

FROM customers

WHERE points BETWEEN 2000 AND 3000

UNION

SELECT customer\_id, first\_name, points, 'Gold' AS Type

FROM customers

WHERE points > 3000

ORDER BY first\_name  
  
  
**MySQL INSERT INTO Statement**

The INSERT INTO statement is used to insert new records in a table.

INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two ways:

1. Specify both the column names and the values to be inserted:

INSERT INTO *table\_name* (*column1*,*column2*,*column3*, ...)  
VALUES (*value1*,*value2*,*value3*, ...);  
  
INSERT INTO orders (order\_id,customer\_id,order\_date,status,comments,shipped\_date,shipper\_id)

VALUES (12, 6, '2024-03-01',1,'sample','2024-03-15',1);

2. If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. Here, the INSERT INTO syntax would be as follows:

INSERT INTO *table\_name*  
VALUES (*value1*,*value2*,*value3*, ...);  
  
INSERT INTO orders

VALUES (11, 6, '2024-02-01',1,'itemx','2024-02-15',5);  
  
  
**Insert Data Only in Specified Columns**

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

The following SQL statement will insert a new record, but only insert data in the "Customer id", "order\_date", shipped\_date and shipper\_id columns (orderID will be updated automatically):  
  
INSERT INTO orders (customer\_id,order\_date,shipped\_date,shipper\_id)

VALUES (6, '2024-06-01','2024-06-15',1);  
  
  
**Insert multiple records/rows**

Insert Multiple records in a table. Here Shippers table shipper id is auto increment so name alone we can insert.  
  
INSERT INTO shippers (name)

VALUES ('Shipper1'),  
('Shipper2'),   
('Shipper3');  
  
  
INSERT INTO products (name,quantity\_in\_stock,unit\_price)

VALUES ('Product1',10,1.32),

('Product2',15,2.5),

('Product3',20,1.5);  
  
  
**Inserting Hierarchical rows**

If we observe orders table actual items in the order are not in this table and they are in the order\_item table.  
  
Actual order has one or more order\_items. This is we called parent child relationship.  
Here order table is parent and order item table is child. One row in order table can have one or more children in order item table.  
  
here we will see how to insert an order and all its items.  
  
**How to insert data in multiple tables.**  
  
In orders table schema first four columns are NN and first column is AI. So customer\_id, order\_date and status are required.  
  
A screenshot of a computer

Description automatically generated  
  
In order items table we have below columns and 4 are NN values.  
  
A screenshot of a computer

Description automatically generated  
Here one order(parent) is having multiple order items(children’s) Now we will see how to insert an order and all its items, so we will learn *how to insert data into multiple tables.*  
  
INSERT INTO orders (customer\_id,order\_date,status)

VALUES (1,'2024-01-19',1)  
  
AS soon as we insert an order Mysql will generate new Id for order and we need that value to be inserted in order items so how we can do that.  
  
In mySQL we have bunch of built in functions. A function is a piece of code that we use ourselfs. One of the functions is “LAST\_INSERT\_ID()” which provides last inserted ID.  
  
SELECT LAST\_INSER\_ID()  
  
NOW query to insert   
  
INSERT INTO orders (customer\_id,order\_date,status)

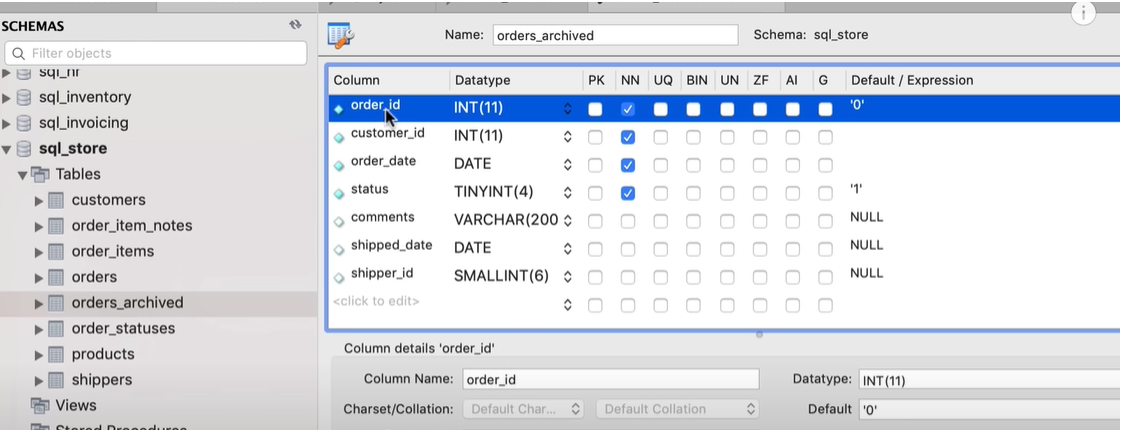
VALUES (2,'2024-01-19',1);

INSERT INTO order\_items

VALUES (LAST\_INSERT\_ID(),4,3,1.5),

(LAST\_INSERT\_ID(),2,1,2.5),

(LAST\_INSERT\_ID(),3,4,2.5);  
  
  
  
**CREATING COPY OF A TABLE**  
  
How to copy data from one table to other table.  
  
First create new table orders\_archieved by copying orders.  
  
CREATE TABLE orders\_archieved AS

SELECT \* FROM orders  
  
here we used subquery(select statement) in create table query.  
  
When we create table like this technique mysql will ignore PK and AI attribute.   
  
  
  
WE can also use sub query statement while inserting tables.  
  
Lets truncate orders\_archived table by right click and truncate.  
  
WE will insert rows from order table where orders are older than current year.  
  
SELECT \*  
FROM orders  
WHERE order\_date < ‘2024-01-01’  
  
Final Query:  
  
INSERT INTO orders\_archieved

SELECT \*

FROM orders

WHERE order\_date < ‘2024-01-01’  
  
  
**Exercise**:  
Create invoices archived table with inner join and condition un paid payments.  
  
USE sql\_invoicing;

CREATE TABLE invoices\_archieved

SELECT i.invoice\_id,

i.number,

c.name AS Client,

i.invoice\_total,

i.payment\_total,

i.invoice\_date,

i.payment\_date,

i.due\_date

FROM invoices i

JOIN clients c

ON c.client\_id = i.client\_id

WHERE payment\_date IS NULL  
  
 **The SQL UPDATE Statement**

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

UPDATE Syntax

UPDATE *table\_name*  
SET *column1*=*value1*,*column2*=*value2*, ...  
WHERE *condition*;

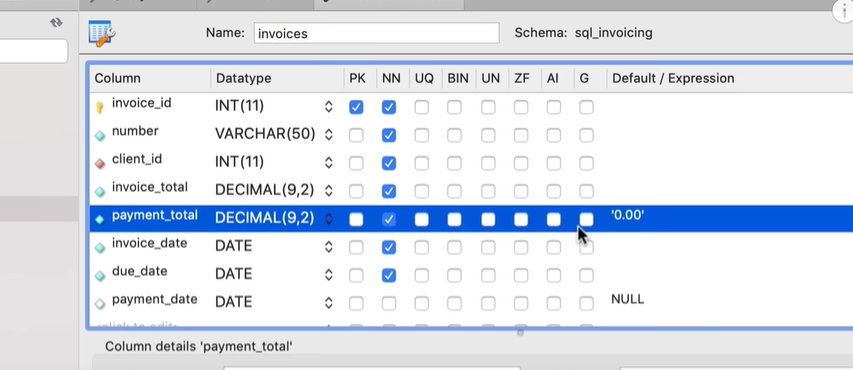
**Note:** Be careful when updating records in a table! Notice the WHERE clause in the UPDATE statement. The WHERE clause specifies which record(s) that should be updated. If you omit the WHERE clause, all records in the table will be updated!  
  
Example:

UPDATE invoices

SET payment\_total = 10, payment\_date = '2019-04-20'

WHERE invoice\_id = 1  
  
Lets think if we updated wrong row so revert the data in the row like below.  
  
UPDATE invoices

SET payment\_total = 0, payment\_date = NULL

WHERE invoice\_id = 1  
  
We try in another way lets observe the table schema here “payment\_total” column has default value and “payment\_date” accepts null.  
  
  
  
  
We can use mysql default “DEFAULT” keyword instead of zero as show below.

UPDATE invoices

SET payment\_total = DEFAULT, payment\_date = NULL

WHERE invoice\_id = 1  
  
  
  
Example:  
  
  
USE sql\_store;

UPDATE customers

SET phone = '123-096-986', city= 'Frankfurt'

WHERE customer\_id = 1;  
 **UPDATE Multiple Records**

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

The following SQL statement will update the points " for all records where state is "CA":

Example

UPDATE Customers  
SET points= points+100  
WHERE state=’CA’;

With IN more than one record:  
  
USE sql\_store;

UPDATE customers

SET points= points+100

WHERE customer\_id IN (1,2)  
  
  
Exercise:  
  
Add 50 points to customers who are born before 1990.  
  
USE sql\_store;

UPDATE customers

SET points = points+50

WHERE birth\_date < '1990-01-01'  
  
  
**USING subqueries in updates**UPDATE orders

SET comments = 'GOLD customers'

WHERE customer\_id IN (SELECT customer\_id

from customers

WHERE points > 3000)  
  
 **The SQL DELETE Statement**

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

DELETE Syntax

DELETE FROM *table\_name*WHERE *condition*;

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

WHERE client\_id = 2  
  
 **Delete a Table**

To delete the table completely, use the DROP TABLE statement:

Example

Remove the Customers table:

DROP TABLE customers;

**MySQL Data Types**

A Data Type specifies a particular type of data, like integer, floating points, Boolean, etc. It also identifies the possible values for that type, the operations that can be performed on that type, and the way the values of that type are stored. In MySQL, each database table has many columns and contains specific data types for each column.  
  
 **Numeric Data Type**

MySQL has all essential SQL numeric data types. These data types can include the exact numeric data types (For example, integer, decimal, numeric, etc.), as well as the approximate numeric data types (For example, float, real, and double precision). It also supports BIT datatype to store bit values. In MySQL, numeric data types are categories into two types, either signed or unsigned except for bit data type.

|  |  |
| --- | --- |
| **Data Type Syntax** | **Description** |
| TINYINT | It is a very small integer that can be signed or unsigned. If signed, the allowable range is from -128 to 127. If unsigned, the allowable range is from 0 to 255. We can specify a width of up to 4 digits. It takes 1 byte for storage. |
| SMALLINT | It is a small integer that can be signed or unsigned. If signed, the allowable range is from  -32768 to 32767. If unsigned, the allowable range is from 0 to 65535. We can specify a width of up to 5 digits. It requires 2 bytes for storage. |
| MEDIUMINT | It is a medium-sized integer that can be signed or unsigned. If signed, the allowable range is from -8388608 to 8388607. If unsigned, the allowable range is from 0 to 16777215. We can specify a width of up to 9 digits. It requires 3 bytes for storage. |
| INT | It is a normal-sized integer that can be signed or unsigned. If signed, the allowable range is from -2147483648 to 2147483647. If unsigned, the allowable range is from 0 to 4294967295. We can specify a width of up to 11 digits. It requires 4 bytes for storage. |
| BIGINT | It is a large integer that can be signed or unsigned. If signed, the allowable range is from -9223372036854775808 to 9223372036854775807. If unsigned, the allowable range is from 0 to 18446744073709551615. We can specify a width of up to 20 digits. It requires 8 bytes for storage. |
| FLOAT(m,d) | It is a floating-point number that cannot be unsigned. You can define the display length (m) and the number of decimals (d). This is not required and will default to 10,2, where 2 is the number of decimals, and 10 is the total number of digits (including decimals). Decimal precision can go to 24 places for a float type. It requires 2 bytes for storage. |
| DOUBLE(m,d) | It is a double-precision floating-point number that cannot be unsigned. You can define the display length (m) and the number of decimals (d). This is not required and will default to 16,4, where 4 is the number of decimals. Decimal precision can go to 53 places for a double. Real is a synonym for double. It requires 8 bytes for storage. |
| DECIMAL(m,d) | An unpacked floating-point number that cannot be unsigned. In unpacked decimals, each decimal corresponds to one byte. Defining the display length (m) and the number of decimals (d) is required. Numeric is a synonym for decimal. |
| BIT(m) | It is used for storing bit values into the table column. Here, M determines the number of bit per value that has a range of 1 to 64. |
| BOOL | It is used only for the true and false condition. It considered numeric value 1 as true and 0 as false. |
| BOOLEAN | It is Similar to the BOOL. |

**Date and Time Data Type:**

This data type is used to represent temporal values such as date, time, datetime, timestamp, and year. Each temporal type contains values, including zero. When we insert the invalid value, MySQL cannot represent it, and then zero value is used.

The following table illustrates all date and time data types that support in MySQL:

|  |  |  |
| --- | --- | --- |
| **Data Type Syntax** | **Maximum Size** | **Explanation** |
| YEAR[(2|4)] | Year value as 2 digits or 4 digits. | The default is 4 digits. It takes 1 byte for storage. |
| DATE | Values range from '1000-01-01' to '9999-12-31'. | Displayed as 'yyyy-mm-dd'. It takes 3 bytes for storage. |
| TIME | Values range from '-838:59:59' to '838:59:59'. | Displayed as 'HH:MM:SS'. It takes 3 bytes plus fractional seconds for storage. |
| DATETIME | Values range from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. | Displayed as 'yyyy-mm-dd hh:mm:ss'. It takes 5 bytes plus fractional seconds for storage. |
| TIMESTAMP(m) | Values range from '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' TC. | Displayed as 'YYYY-MM-DD HH:MM:SS'. It takes 4 bytes plus fractional seconds for storage. |

**String Data Types:**

The string data type is used to hold plain text and binary data, for example, files, images, etc. MySQL can perform searching and comparison of string value based on the pattern matching such as LIKE operator, Regular Expressions, etc.

The following table illustrates all string data types that support in MySQL:

|  |  |  |
| --- | --- | --- |
| **Data Type Syntax** | **Maximum Size** | **Explanation** |
| CHAR(size) | It can have a maximum size of 255 characters. | Here size is the number of characters to store. Fixed-length strings. Space padded on the right to equal size characters. |
| VARCHAR(size) | It can have a maximum size of 255 characters. | Here size is the number of characters to store. Variable-length string. |
| TINYTEXT(size) | It can have a maximum size of 255 characters. | Here size is the number of characters to store. |
| TEXT(size) | Maximum size of 65,535 characters. | Here size is the number of characters to store. |
| MEDIUMTEXT(size) | It can have a maximum size of 16,777,215 characters. | Here size is the number of characters to store. |
| LONGTEXT(size) | It can have a maximum size of 4GB or 4,294,967,295 characters. | Here size is the number of characters to store. |
| BINARY(size) | It can have a maximum size of 255 characters. | Here size is the number of binary characters to store. Fixed-length strings. Space padded on the right to equal size characters. (introduced in MySQL 4.1.2) |
| VARBINARY(size) | It can have a maximum size of 255 characters. | Here size is the number of characters to store. Variable-length string. (introduced in MySQL 4.1.2) |
| ENUM | It takes 1 or 2 bytes that depend on the number of enumeration values. An ENUM can have a maximum of 65,535 values. | It is short for enumeration, which means that each column may have one of the specified possible values. It uses numeric indexes (1, 2, 3…) to represent string values. |
| SET | It takes 1, 2, 3, 4, or 8 bytes that depends on the number of set members. It can store a maximum of 64 members. | It can hold zero or more, or any number of string values. They must be chosen from a predefined list of values specified during table creation |

**MySQL Constraints**

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

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

Syntax

CREATE TABLE *table\_name*(  
*column1 datatype* *constraint*,  
*column2 datatype* *constraint*,  
*column3 datatype* *constraint*,  
    ....  
);  
  
MySQL Constraints

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

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

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

The following constraints are commonly used in SQL:

NOT NULL - Ensures that a column cannot have a NULL value

UNIQUE - Ensures that all values in a column are different

PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table

FOREIGN KEY - Prevents actions that would destroy links between tables

CHECK - Ensures that the values in a column satisfies a specific condition

DEFAULT - Sets a default value for a column if no value is specified

CREATE INDEX - Used to create and retrieve data from the database very quickly

**The MySQL CREATE DATABASE Statement**

The CREATE DATABASE statement is used to create a new SQL database.

Syntax

CREATE DATABASE *databasename*;

Example:  
CREATE DATABASE sampleDB;

**The MySQL DROP DATABASE Statement**

The DROP DATABASE statement is used to drop an existing SQL database.

Syntax

DROP DATABASE *databasename*;

Example:

DROP DATABASE samleDB;

**The MySQL CREATE TABLE Statement**

The CREATE TABLE statement is used to create a new table in a database.

Syntax

CREATE TABLE *table\_name*(  
*column1 datatype*,  
*column2 datatype*,  
*column3 datatype*,  
   ....  
);  
  
The column parameters specify the names of the columns of the table.

The datatype parameter specifies the type of data the column can hold (e.g. varchar, integer, date, etc.).

Example:  
  
CREATE TABLE products (

product\_id int(11) NOT NULL AUTO\_INCREMENT,

name varchar(50) NOT NULL,

quantity\_in\_stock int(11) NOT NULL,

unit\_price decimal(4,2) NOT NULL,

PRIMARY KEY (product\_id)

)  
  
  
INSERT INTO products VALUES (1,'Foam Dinner Plate',70,1.21),

(2,'Pork - Bacon,back Peameal',49,4.65),

(3,'Lettuce - Romaine, Heart',38,3.35),

(4,'Brocolinni - Gaylan, Chinese',90,4.53),

(5,'Sauce - Ranch Dressing',94,1.63),

(6,'Petit Baguette',14,2.39),

(7,'Sweet Pea Sprouts',98,3.29),

(8,'Island Oasis - Raspberry',26,0.74),

(9,'Longan',67,2.26),

(10,'Broom - Push',6,1.09)

Default value example:  
  
CREATE TABLE example\_table (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100) NOT NULL,

status ENUM('active', 'inactive') DEFAULT 'active',

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP );  
  
  
CREATE TABLE employees (

employee\_id INT AUTO\_INCREMENT PRIMARY KEY,

first\_name VARCHAR(50) NOT NULL,

last\_name VARCHAR(50) NOT NULL,

hire\_date DATE DEFAULT CURRENT\_DATE,

is\_active BOOLEAN DEFAULT TRUE,

department VARCHAR(50) DEFAULT 'General' );  
  
**Create Table Using Another Table**

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

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

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

Syntax:

CREATE TABLE *new\_table\_name* AS  
    SELECT *column1, column2,...*  
    FROM *existing\_table\_name*  
    WHERE ....;  
  
  
Example:  
  
USE `store`;

CREATE TABLE TestTable AS

SELECT \* FROM sql\_store.customers;  
  
  
  
**The MySQL DROP TABLE Statement**

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

Syntax

DROP TABLE *table\_name*;  
  
Example:  
DROP TABLE Test;  
  
 **MySQL TRUNCATE TABLE**

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

Syntax

TRUNCATE TABLE *table\_name*;  
  
Example:  
  
TRUNCATE TABLE *test*;

**MySQL ALTER TABLE Statement**

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.

The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

**ALTER TABLE - ADD Column**

To add a column in a table, use the following syntax:

ALTER TABLE *table\_name*  
ADD *column\_name datatype*;

The following SQL adds an "Email" column to the "Customers" table:

Example

ALTER TABLE customers  
ADD Email varchar(255);

**ALTER TABLE - DROP COLUMN**

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

ALTER TABLE *table\_name*  
DROP COLUMN *column\_name*;

The following SQL deletes the "Email" column from the "Customers" table:

Example

ALTER TABLE Customers  
DROP COLUMN Email;

**ALTER TABLE - MODIFY COLUMN**

To change the data type of a column in a table, use the following syntax:

ALTER TABLE *table\_name*  
MODIFY COLUMN *column\_name datatype*;  
  
ALTER TABLE customers  
ADD DateOfBirth date;

**Change Data Type Example**

Now we want to change the data type of the column named "DateOfBirth" in the "customers" table.

We use the following SQL statement:

Example

ALTER TABLE customers  
MODIFY COLUMN DateOfBirth year;

Notice that the "DateOfBirth" column is now of type year and is going to hold a year in a two- or four-digit format.

**DROP COLUMN Example**

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

We use the following SQL statement:

Example

ALTER TABLE customers  
DROP COLUMN DateOfBirth;

**RENAME COLUMN Example**

Next, we want to rename the column name "DateOfBirth" to “DOB”in the "customers" table.

We use the following SQL statement:

USE sql\_store;

ALTER TABLE customers

RENAME COLUMN DateOfBirth TO DOB

**The MySQL GROUP BY Statement**

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

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

GROUP BY Syntax

SELECT *column\_name(s)*  
FROM *table\_name*  
WHERE *condition*  
GROUP BY *column\_name(s)*ORDER BY *column\_name(s);*

USE sql\_store;

SELECT COUNT(customer\_id), state

FROM customers

GROUP BY state

ORDER BY COUNT(customer\_id) DESC;  
  
  
Example2:  
  
USE test;

CREATE TABLE sales (

id INT AUTO\_INCREMENT PRIMARY KEY,

product\_id INT NOT NULL,

quantity INT NOT NULL,

price DECIMAL(10, 2) NOT NULL,

sale\_date DATE NOT NULL

)

INSERT INTO sales (product\_id, quantity, price, sale\_date) VALUES

(101, 5, 10.00, '2024-09-01'),

(102, 3, 20.00, '2024-09-01'),

(101, 2, 10.00, '2024-09-02'),

(103, 7, 15.00, '2024-09-02'),

(102, 1, 20.00, '2024-09-03'),

(101, 4, 10.00, '2024-09-03');

**Total Quantity Sold by Product**

If you want to find the total quantity sold for each product, you would use the GROUP BY clause like this:

SELECT product\_id, SUM(quantity) AS total\_quantity

FROM sales

GROUP BY product\_id;

This query groups the rows by product\_id and calculates the total quantity for each product.   
  
**Total Revenue by Date**

If you want to calculate the total revenue for each sale date, you can do it like this:

SELECT sale\_date, SUM(quantity \* price) AS total\_revenue

FROM sales

GROUP BY sale\_date;

This query groups the rows by sale\_date and calculates the total revenue for each date.

**Average Price per Product**

To find the average price of each product, you can use the following query:

SELECT product\_id, AVG(price) AS average\_price

FROM sales

GROUP BY product\_id;

This query groups the rows by product\_id and calculates the average price for each product.

**The MySQL HAVING Clause**

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

HAVING Syntax

SELECT *column\_name(s)*  
FROM *table\_name*  
WHERE *condition*  
GROUP BY *column\_name(s)*HAVING *condition*ORDER BY *column\_name(s);*

The HAVING clause is used to filter the results of a GROUP BY query. It is similar to the WHERE clause, but WHERE is used to filter rows before grouping, while HAVING is used to filter groups after grouping.

**Example Scenario**

Using the same sales table, let’s say you want to find products that have a total quantity sold greater than 6. You would use the HAVING clause to filter the results after grouping.

**Query with HAVING**

Here’s a query that groups the data by product\_id, calculates the total quantity sold for each product, and then filters to include only those products where the total quantity is greater than 6:

SELECT product\_id, SUM(quantity) AS total\_quantity

FROM sales

GROUP BY product\_id

HAVING SUM(quantity) > 6;

**Explanation**

1. **SELECT product\_id, SUM(quantity) AS total\_quantity**: This selects the product\_id and the sum of quantity for each product.
2. **FROM sales**: Specifies the table to query.
3. **GROUP BY product\_id**: Groups the results by product\_id.
4. **HAVING SUM(quantity) > 6**: Filters the grouped results to include only those where the total quantity is greater than 6.

**The MySQL EXISTS Operator**

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

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

EXISTS Syntax

SELECT *column\_name(s)*  
FROM *table\_name*  
WHERE EXISTS  
(SELECT *column\_name*FROM *table\_name* WHERE *condition*);  
  
Exampe table:

CREATE TABLE products ( product\_id INT PRIMARY KEY, product\_name VARCHAR(50) );

INSERT INTO products (product\_id, product\_name) VALUES (101, 'Widget A'), (102, 'Widget B'), (103, 'Widget C');

Exists:  
  
Products That Have Been Sold  
  
SELECT product\_id, product\_name

FROM products p

WHERE EXISTS ( SELECT 1 FROM sales s WHERE s.product\_id = p.product\_id );

**Explanation**

* **SELECT product\_id, product\_name FROM products p**: Selects the product\_id and product\_name from the products table.
* **WHERE EXISTS**: Filters the products based on the existence of rows in the sales table.
* **Subquery**: The subquery selects 1 from the sales table where s.product\_id matches p.product\_id. If the subquery returns at least one row, the EXISTS condition is true.

**Products with Total Sales Quantity Greater Than 5**

SELECT product\_id, product\_name

FROM products p

WHERE EXISTS (

SELECT 1

FROM sales s

WHERE s.product\_id = p.product\_id

GROUP BY s.product\_id

HAVING SUM(s.quantity) > 5

);

**Explanation**

* **SELECT product\_id, product\_name FROM products p**: Selects the product\_id and product\_name from the products table.
* **WHERE EXISTS**: Filters the products based on the result of the subquery.
* **Subquery**:
  + **SELECT 1 FROM sales s**: Selects rows from the sales table.
  + **WHERE s.product\_id = p.product\_id**: Joins the sales and products tables on product\_id.
  + **GROUP BY s.product\_id**: Groups the results by product\_id.
  + **HAVING SUM(s.quantity) > 5**: Filters groups where the total quantity sold is greater than 5.

**The MySQL CASE Statement**

The CASE statement goes through conditions and returns a value when the first condition is met (like an if-then-else statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.

If there is no ELSE part and no conditions are true, it returns NULL.

CASE Syntax

CASE  
    WHEN *condition1* THEN *result1*  
    WHEN *condition2* THEN *result2*  
    WHEN *conditionN* THEN *resultN*  
    ELSE *result*  
END;  
  
Example:

SELECT id, product\_id, quantity,

CASE quantity

WHEN 1 THEN 'Low'

WHEN 2 THEN 'Medium'

WHEN 3 THEN 'High'

ELSE 'Very High'

END AS quantity\_category

FROM sales;

**Explanation**

* **CASE quantity**: Evaluates the quantity column.
* **WHEN 1 THEN 'Low'**: If quantity is 1, then the result is 'Low'.
* **WHEN 2 THEN 'Medium'**: If quantity is 2, then the result is 'Medium'.
* **WHEN 3 THEN 'High'**: If quantity is 3, then the result is 'High'.
* **ELSE 'Very High'**: For any other value, the result is 'Very High'.
* **AS quantity\_category**: Aliases the result of the CASE statement as quantity\_category.

Now let’s use a searched CASE statement to categorize products based on their quantity ranges.

SELECT id, product\_id, quantity,

CASE

WHEN quantity <= 2 THEN 'Low'

WHEN quantity BETWEEN 3 AND 5 THEN 'Medium'

WHEN quantity BETWEEN 6 AND 10 THEN 'High'

ELSE 'Very High'

END AS quantity\_category

FROM sales;

**Explanation**

* **CASE**: Starts the searched CASE statement.
* **WHEN quantity <= 2 THEN 'Low'**: If quantity is 2 or less, the result is 'Low'.
* **WHEN quantity BETWEEN 3 AND 5 THEN 'Medium'**: If quantity is between 3 and 5, the result is 'Medium'.
* **WHEN quantity BETWEEN 6 AND 10 THEN 'High'**: If quantity is between 6 and 10, the result is 'High'.
* **ELSE 'Very High'**: For quantities greater than 10, the result is 'Very High'.
* **AS quantity\_category**: Aliases the result of the CASE statement as quantity\_category

**Primary keys and foreign keys are fundamental concepts in relational database design.**

They help maintain the integrity and relationships between tables.

**Definitions**

* **Primary Key**: A column (or a set of columns) in a table that uniquely identifies each row in that table. Each table can have only one primary key, and its values must be unique and not null.
* **Foreign Key**: A column (or a set of columns) in one table that refers to the primary key of another table. Foreign keys establish and enforce a link between the data in the two tables.

**Example Scenario**

Consider a simple database for a bookstore with two tables:

1. **authors**: Contains information about book authors.
2. **books**: Contains information about books, each associated with an author.

**Table Creation with Primary and Foreign Keys**

**1. Create authors Table**

The authors table will have author\_id as the primary key.

CREATE TABLE authors (

author\_id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100) NOT NULL

);

* **author\_id**: Primary key of the authors table. It is unique for each author and auto-increments with each new record.

**2. Create books Table**

The books table will have book\_id as the primary key and author\_id as a foreign key referring to authors.

CREATE TABLE books (

book\_id INT AUTO\_INCREMENT PRIMARY KEY,

title VARCHAR(100) NOT NULL,

publication\_year INT,

author\_id INT,

FOREIGN KEY (author\_id) REFERENCES authors(author\_id)

);

* **book\_id**: Primary key of the books table. It is unique for each book and auto-increments with each new record.
* **author\_id**: Foreign key that references the author\_id in the authors table. It creates a relationship between books and authors.

**Inserting Data**

Now let’s insert some sample data into these tables.

**Insert Data into authors**

INSERT INTO authors (name) VALUES

('J.K. Rowling'),

('George R.R. Martin'),

('J.R.R. Tolkien');

**Insert Data into books**

INSERT INTO books (title, publication\_year, author\_id) VALUES

('Harry Potter and the Philosopher\'s Stone', 1997, 1),

('A Game of Thrones', 1996, 2),

('The Hobbit', 1937, 3),

('Harry Potter and the Chamber of Secrets', 1998, 1);

**Query to Verify Data and Relationships**

You can use a JOIN query to verify the relationship between the tables:

SELECT b.title, b.publication\_year, a.name AS author

FROM books b

JOIN authors a ON b.author\_id = a.author\_id;

**Explanation**

* **SELECT b.title, b.publication\_year, a.name AS author**: Selects the book title, publication year, and author name.
* **FROM books b**: Specifies the books table as the source of the data.
* **JOIN authors a ON b.author\_id = a.author\_id**: Joins the books table with the authors table using the author\_id foreign key. This ensures that each book is matched with the correct author.

This example demonstrates how to define and use primary and foreign keys to create relationships between tables in a MySQL database. The primary key ensures each record is unique within its table, while the foreign key enforces a relationship between records in different tables.

Top of Form

Bottom of Form

**Stored Procedures:**  
Stored procedures are a powerful feature in MySQL that allow you to encapsulate SQL logic in a reusable, executable unit. They can be used to perform complex operations, execute multiple SQL statements in sequence, and improve performance by reducing the amount of SQL sent to the server.

Basic Concepts

* Stored Procedure: A precompiled collection of one or more SQL statements stored under a name and executed as a unit.
* Parameters: Stored procedures can accept parameters to provide input values or return output values.

**Creating Stored Procedures**

Here's a step-by-step guide on how to create and use stored procedures in MySQL.

Before creating stored procedure. Create two tables below.

CREATE TABLE employees (

employee\_id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100),

department VARCHAR(50),

salary DECIMAL(10, 2) );

CREATE TABLE salary\_log (

log\_id INT AUTO\_INCREMENT PRIMARY KEY,

employee\_id INT,

changed\_on DATETIME,

new\_salary DECIMAL(10, 2),

FOREIGN KEY (employee\_id) REFERENCES employees(employee\_id) );  
  
Sample data:  
  
INSERT INTO employees (name, department, salary)

VALUES ('Alice', 'HR', 60000.00),

('Bob', 'IT', 75000.00),

('Charlie', 'Marketing', 30000.00);

**Creating a Stored Procedure**

Let's start with a simple stored procedure that retrieves employees by department.Stored procedures will be saved in DB.

DELIMITER //

CREATE PROCEDURE GetEmployeesByDepartment(IN dept\_name VARCHAR(50))

BEGIN

SELECT \* FROM employees

WHERE department = dept\_name;

END //

DELIMITER ;

* **DELIMITER //**: Changes the delimiter to // to avoid conflicts with the semicolons used inside the procedure.
* **CREATE PROCEDURE**: Defines a new stored procedure.
* **IN dept\_name VARCHAR(50)**: Specifies an input parameter named dept\_name.
* **BEGIN ... END**: Encloses the SQL statements that make up the body of the procedure.
* **DELIMITER ;**: Resets the delimiter back to the default ;.

**showing Stored Procedure in DB:**

SHOW PROCEDURE STATUS WHERE Db = 'test';

**Calling a Stored Procedure**

To call the stored procedure and get employees from a specific department:

CALL GetEmployeesByDepartment('HR');

**Full Example: Creating, Viewing, and Dropping a Stored Procedure**

Here’s a complete example demonstrating how to create, view, and drop a stored procedure.

**1. Create a Stored Procedure**

DELIMITER //

CREATE PROCEDURE ListAllEmployees()

BEGIN

SELECT \* FROM employees;

END //

DELIMITER ;

**2. View the Stored Procedure**

SHOW CREATE PROCEDURE ListAllEmployees;

**3. Execute the Stored Procedure**sql

CALL ListAllEmployees();

**4. Drop the Stored Procedure**

DROP PROCEDURE IF EXISTS ListAllEmployees;

**Create Stored Procedures**

-- Procedure to get employees by department

DELIMITER //

CREATE PROCEDURE GetEmployeesByDepartment(IN dept\_name VARCHAR(50))

BEGIN

SELECT \* FROM employees

WHERE department = dept\_name;

END //

-- Procedure to update salary and log the change

CREATE PROCEDURE UpdateEmployeeSalary(IN emp\_id INT, IN new\_salary DECIMAL(10, 2))

BEGIN

-- Update salary

UPDATE employees

SET salary = new\_salary

WHERE employee\_id = emp\_id;

-- Log the salary change

INSERT INTO salary\_log (employee\_id, changed\_on, new\_salary)

VALUES (emp\_id, NOW(), new\_salary);

END //

DELIMITER ;  
  
  
**Calling:**CALL UpdateEmployeeSalary(2, 80000.00);

**4. Call Stored Procedures**

-- Get employees from HR department

CALL GetEmployeesByDepartment('HR');

-- Update Bob's salary and log the change

CALL UpdateEmployeeSalary(2, 80000.00);

-- Check the updated salary and log

SELECT \* FROM employees;

SELECT \* FROM salary\_log;

**Conclusion**

Stored procedures in MySQL are a powerful tool for managing complex queries and operations. They allow you to encapsulate logic, improve performance by reducing network traffic, and maintain consistency across applications. By using parameters and handling multiple SQL statements, you can create robust and reusable database routines.