MySQL

Introduction

MySQL is the most popular and free Open-Source Relational Database Management System **(RDBMS).**

MySQL uses Structured Query Language **(SQL)** to store, manage and retrieve data, and control the accessibility to the data.

MySQL is written in C and C++. Its SQL parser is written in yacc, but it uses a home-brewed lexical analyzer.

History of MySQL

- MySQL was developed by Swedish company **MySQL AB,** founded by **David Axmark, Allan Larsson,** and **Michael Monty Widenius** in 1994.
- First internal release on 23rd May 1995.
- Windows Version was released on the 8th of January 1998 for Windows 95 and NT.
- Version 3.23: beta from June 2000, production release January 2001.
- Version 4.0: beta from August 2002, production release March 2003 (unions).
- Version 4.1: beta from June 2004, production release October 2004.
- Version 5.0: beta from March 2005, production release October 2005.
- Sun Microsystems acquired MySQL AB on 26th February 2008.
- Version 5.1: production release 27th November 2008.
- Oracle acquired Sun Microsystems on 27th January 2010.
- Version 5.5: general availability on 3rd December 2010
- Version 5.6: general availability on 5th February 2013
- Version 5.7: general availability on 21st October 2015
- Version 8.0: general availability on 19th April 2018
- Version 8.1, 8.2: general availability in 2023.
- Version 8.3, 8.4, 9.0: general availability in 2024.

MySQL Installation guide

Installation Guide 1 Installation Guide 2

MySQL Server connection using terminal (command prompt)

- open bin path of MySQL server folder -> copy the path.
- open terminal -> cd C:\Program Files\MySQL\MySQL Server 8.0\bin -> press enter.
- use command MySQL -u root -p ->C:\Program Files\MySQL\MySQL Server 8.0\bin>MySQL -u root -p -> press enter.
- now enter your root password: Enter password: *******-> press enter.
- you are connected to MySQL: start writing queries -> MySQL>

Data Types in SQL

1. String

2. Number

3. Date & Time

String

Data Type	Description	Size Range
`char`	Fixed-length string	1 - 255 characters
`varchar`	Variable-length string	1 - 65535 characters
`tinytext`	Small text string	1 - 255 characters
`text`	Text string	1 - 65535 characters
`mediumtext`	Medium-length text string	1 - 16777215 characters
`longtext`	Long text string	1 - 4294967295 characters
`binary`	Fixed-length binary data	1 - 255 bytes
`varbinary`	Variable-length binary data	1 - 65535 bytes
`blob`	Binary Large Object	1 - 65535 bytes
`mediumblob`	Medium-length binary object	1 - 16777215 bytes
`longblob`	Long binary object	1 - 4294967295 bytes
`enum`	Enumeration	1 to 65535 values
`set`	Set of values	0 to 64 values

Number

Data Type	Description	Range
`bit`	Bit-field	1 - 64 bits
`tinyint`	Small integer	-128 to 127 / 0 to 255 (unsigned)
`smallint`	Small integer	-32768 to 32767 / 0 to 65535 (unsigned)
`mediumint`	Medium integer	-8388608 to 8388607 / 0 to 16777215 (unsigned)
`int`	Integer	-2147483648 to 2147483647 / 0 to 4294967295 (unsigned)
`bigint`	Large integer	-9223372036854775808 to 9223372036854775807 / 0 to 18446744073709551615 (unsigned)
`float`	Floating-point number	(size, d) -> d (0 to 24)
`double`	Double-precision float	(size, d) -> d (25 to 53)
`decimal`	Exact numeric value	(size, d) -> size (1 to 65), d (0 to 30)
`bool`	Boolean	0 / 1 => false / true

Date & Time

Data Type	Description	Format
`date`	Date	'YYYY-MM-DD' (1000-01-01 to 9999-12-31)
`datetime`	Date and time	'YYYY-MM-DD HH:MM ' (1000-01-01 00:00:00 to 9999-12-31 23:59:59)
`timestamp`	Timestamp	'YYYY-MM-DD HH:MM ' (1970-01-01 00:00:01 UTC to 2038-01-19 03:14:07 UTC)
`time`	Time	'HH:MM ' (-838:59:59 to 838:59:59)
`year`	Year	'YYYY' (1901 to 2155)

MySQL SQL

SQL stands for **Structured Query Language**, and it is the standard language for dealing with relational databases.

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

SQL keywords are **not case-sensitive:** SELECT is same as select.

Using **Semicolon (;)** is the standard way to separate each SQL statements where more than one SQL statement is to be executed and some databases require a **semicolon** at the end of each 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

User in MySQL

A **user** in MySQL is an account created for accessing the MySQL database server. Each user has **specific privileges** that determine what actions they can perform on the databases and tables within the server.

When MySQL is installed, a default administrative user called **root** is created and the **password for this root user** is set by us during the installation process.

Once we have logged in with our default root account, we can create **multiple user accounts** and start working on database by granting them **privileges** from our root user account.

Syntax to create a User:

```
CREATE USER 'username'@'host' IDENTIFIED BY 'password';
```

Example:

- *CREATE USER 'gfguser1'@'localhost' IDENTIFIED BY 'abcd';*
- *GRANT ALL PRIVILEGES ON mydatabase.* TO 'gfguser1'@'localhost';*
- *FLUSH PRIVILEGES;*

CREATE USER: Creates a new user with a specified password.

GRANT ALL PRIVILEGES: Grants all permissions on a specified database to the user.

FLUSH PRIVILEGES: Applies the changes made to user privileges immediately.

Steps to add new User in MySQL workbench:

- ➔ Launch MySQL Workbench on your machine.
- ➔ Click on the + icon next to MySQL Connections to create a new connection.
- ➔ **Name:** Enter a name for your connection (e.g., gfguser1 Connection).
- ➔ **Connection Method**: Leave it as Standard (TCP/IP).
- ➔ **Hostname**: Enter localhost.
- ➔ **Port**: Default is 3306 (Change it if required)
- ➔ **Username**: Enter username (gfguser1).
- ➔ **Password**: Click on Store in Vault... (or Store in Keychain... on macOS) and enter the password (abcd).
- ➔ Click the Test Connection button to check if the connection works. If the connection is successful, you will see a success message. If there is an error, ensure the user credentials and host details are correct.
- ➔ If the test is successful, click OK to save the connection.
- ➔ You can now select this connection from the MySQL Workbench home screen and click on it to connect to the MySQL server as gfguser1.

Example Queries to change existing User password:

```
ALTER USER 'gfguser1'@'localhost' IDENTIFIED BY 'newpassword';
```

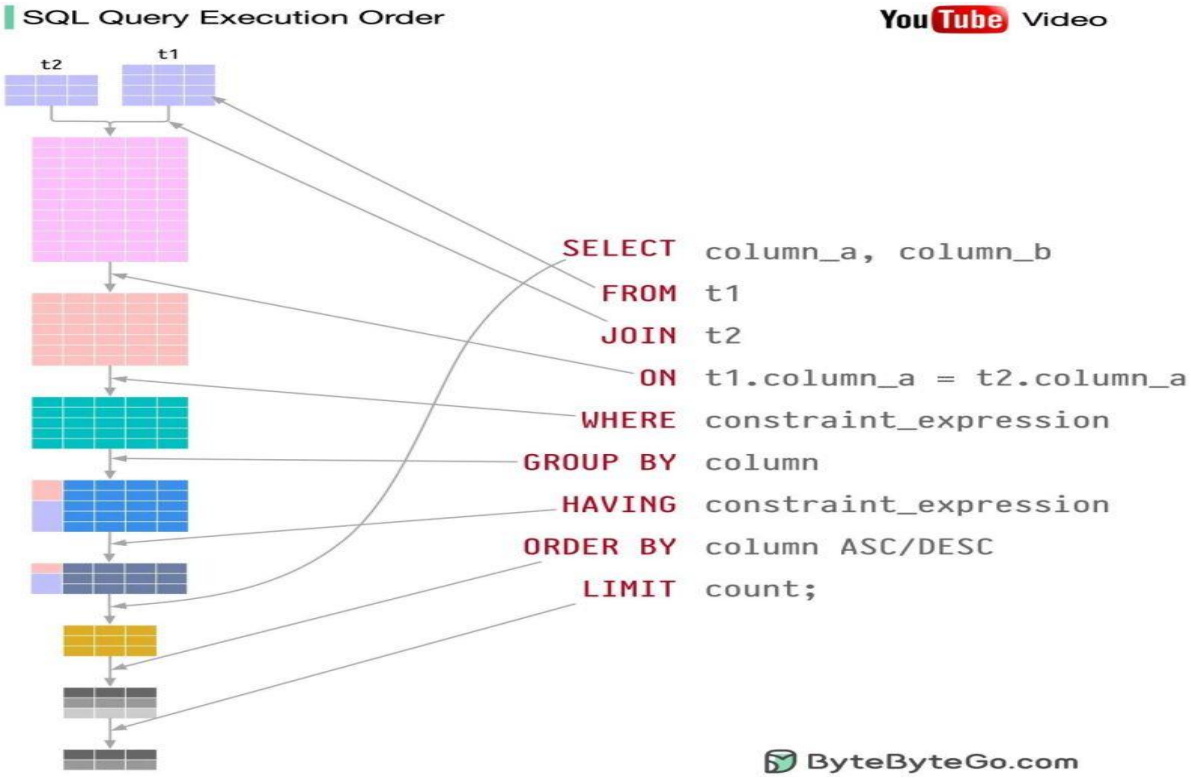
```
SET PASSWORD FOR 'gfguser1'@'localhost' = 'newpassword';
```

Important Notes

- Always run **GRANT PRIVILEGES** after **CREATING USER** and run **FLUSH PRIVILEGES;** after granting privileges to **apply changes.**
- You can only **change the password** of logged in user only: you cannot change **gfguser2's password from gfguser1 account.**
- You cannot directly add a new user from **workbench** by clicking on the **+ icon,** first you need to create a new user with **create user command** then **grant privileges** and after **flushing privileges** only you can **add** the new user.
- The databases and tables that were available in another user account will **not be copied or moved** to newly created user account directly, only the **database (mydatabase)** that you mentioned in **GRANT PRIVILEGE** will be moved or given access.

MySQL Queries

Queries can be understood as the commands which interacts with database tables to work around with data.



CREATE QUERY

It is used to create databases, tables, indexes, etc.

Syntax: CREATE TABLE TABLE_NAME (ATTRIBUTE_NAME1 Datatype, ATTRIBUTE_NAME2 Datatype…)

Example

CREATE TABLE student (name char(25), id int, cgpa float)

INSERT QUERY

It is used to insert data into existing tables.

Syntax: INSERT INTO TABLE_NAME (ATTRIBUTE_NAME1, ATTRIBUTE_NAME2, ATTRIBUTE_NAME3) VALUES (value1, value2, value3)

Example

INSERT INTO student (name, id, cgpa) VALUES (‘kalyan’, 30959, 9.46);

DELETE QUERY

It is used to delete data from tables.

Syntax: DELETE FROM TABLE_NAME WHERE attribute_name = value

Example

DELETE FROM student WHERE id = 30959;

UPDATE QUERY

It is used to update existing data.

Syntax: UPDATE TABLE_NAME SET attribute_name = value WHERE condition

Example

UPDATE student SET cgpa = 9.5 WHERE id = 30959;

SELECT QUERY

It is used to select data from a database and the data returned is stored in a table which is called result-set.

SELECT SYNTAX TO RETRIVE ALL RECORDS

Select * from Table _Name.

Example

Select * from student;

SELECT SYNTAX TO GET SPECIFIC COLUMNS

Select column1, column2 from Table_Name;

Example

Select name, id from student;

SELECT SYNTAX TO GET DISTINCT VALUES

Select distinct column_name from table_name

Example

Select distinct name from student;

SELECT DISINCT TO GET NAMES WITH EVEN ID NUMBERS

Select distinct s.name from student s where MOD(s.id,2) =0;

MySQL WHERE Clause

It is used to filter the records based on a condition.

Syntax: Select column_names from table_name WHERE condition

Example

Select id, name from student where cgpa=9.46;

Operators in The WHERE Clause

The following operators can be used in the `WHERE` clause:

Operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<>	Not equal. **Note:** In some versions of SQL this operator may be written as !=
BETWEEN	Between a certain range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

MySQL BETWEEN Example

Select * from student where id BETWEEN 30900 and 31000;

MySQL LIKE

It is used in a WHERE clause to search for a specified pattern in a column.

Examples

- Select * from student where name like 'a%';

  ➔ This will retrieve all records whose name starts with a.

- Select * from student where name like '%a';

  ➔ This will retrieve all records whose name ends with a.

- Select * from student where name like '%ab%';

  ➔ This will retrieve all records whose name contains ab.

MySQL NOT Examples

- Select * from city where not name = 'kalyan';

  ➔ This will retrieve all the records where the name is not Kalyan.

- Select * FROM student WHERE id <> 18;

  ➔ This will retrieve all the records where the id is not 18.

❖ The NOT can be used with NOT LIKE, NOT BETWEEN, etc.

MySQL ORDER BY

- It is used to order the records.
- Its default order is ascending.
- The values can be ordered in ascending order using ASC.
- The values can be ordered in descending order using DESC.

Examples

- Select * from student ORDER BY id, name;

  ➔ This will retrieve all the records by sorting the id and name column values in ascending order.

- Select * from student ORDER BY id ACS, name DESC;

  ➔ This will retrieve all the records by sorting id column values in ascending order and name column values in descending order.

MySQL IS NULL

- It is used to check if a value is null or not.
- It returns the rows which are null based on condition.

Example

- Select * from student id **is null**;

  ➔ This will retrieve all the records from student table where id is null.

MySQL IS NOT NULL

- It is used to check if a value is not null or not.
- It returns the rows which are not null based on condition.

Example

- Select * from student id **is not null**;

  ➔ This will retrieve all the records from student table where id is not null.

MySQL SELECT TOP

- It is like MySQL limit.
- It retrieves records up to a provided range.

Example

- Select Top 3 * from student; (SQL)

  ➔ This will retrieve the first 3 records from the student table.

- Select * from student **LIMIT** 3; (MySQL)

  ➔ This will retrieve the first 3 records from the student table.

- Select * from student **LIMIT** (select **count**(*) * 0.5 from student);

  ➔ This will retrieve the first 50 percent of rows data from the table.

- Select * from student where id > 30959 **LIMIT** 3;

  ➔ This will retrieve the top 3 rows from student table where id > 30959.

MySQL MIN & Max

- The min and max functions in MySQL are used to get the min and max values in the specified column.
- It works with numeric data types and date time values.

Examples

- Select min(id) from student;

  ➔ This will return the minimum id number from student table.

- Select max(id) from student;

  ➔ This will return the maximum id number from student table.

- Select min(id) as minimum_id from student table;

  ➔ This will return the minimum id along with as alias for minimum_id.

MySQL COUNT

- It returns the number of records based on column.

Examples

- Select **count**(id) from student;

  ➔ It returns the number of id's.

- Select count(distinct id) from student;
  ➔ It returns the number of unique id's.

- Select count(*) as [number of records] from student;

  ➔ It returns the number of records with an alias number of records.

SQL SUM

- It returns the sum of all records of a specific column.

Examples

- Select sum(id) from student;

  ➔ It will return the sum of all id's.

- Select sum(id) from student where name = 'kalyan';

  ➔ It will return the sum of id's where name is kalyan.

SQL AVG

- It returns the average of a column.

Example

- Select avg(cgpa) as [Average CGPA] from student;

  ➔ It will return the average cgpa from student table;

SQL LIKE

➔ It is used with WHERE clause to search for a specified pattern in a column.
➔ It uses two wildcard characters to search for patterns.
➔ The first wildcard character is '%' which represents 0, 1 or multiple characters.
➔ The second wildcard character is '_' which represents single character.

Example

- Select * from student where name like '%a';
- Select * from student were name REGEXP '[a]$';

➔ This will retrieve all the records where student name ends with ‘a’.

- Select * from student where name like ‘a%’; (SQL)
- Select * from student where name REGEXP ‘^[a]’; (MySQL)

➔ It will retrieve all the records where student name starts with ‘a’.

- Select * from student where name like ‘%an%’;

➔ It will retrieve all the records where name contains ‘an’.

- Select * from student name like ‘k_l%’;

➔ It will retrieve all the records where name is kalyan, kali, etc.
➔ Here the ‘_’ represents any single character.
➔ Here the ‘%’ represents all characters after the pattern.

SQL WildCard Characters

➔ A wildcard character is used to substitute one or more characters in a stirng.
➔ They are used with LIKE operator.

Examples

- Select * from student where name like ‘[rvk]%’; (SQL)
- Select * from student where name REGEXP ‘[rvk]’; (MySQL)

➔ This will retrieve all the records where name starts with ‘r’ or ‘v’ or ‘k’.

- Select * from student where name like ‘[a-c]%’;

➔ This will retrieve all the records where name starts with ‘a’ or ‘b’ or ‘c’.

SQL IN

➔ It allows us to specify multiple values in a where clause.
➔ It is a shortcut for or operations.

Example

- Select * from student where cgpa in 9.46;

  ➔ This will retrieve all the students having cgpa as 9.46.

- Select * from student where id in (30959,30976);

  ➔ This will return all the student details with id as 30959 and 30976.

SQL NOT IN

➔ It is the opposite of IN.

Example

- Select * from student where id NOT IN (30959, 30976);

  ➔ This will return all the student details with id except 30959 and 30976.

- Select * from student where id NOT IN(select sid from certified_students);

  ➔ This will return all the records where student ids are not in certified_students.

SQL BETWEEN

➔ It is used to retrieve a range of records based on where condition.

Example

- Select * from student where id BETWEEN 30950 and 30960;

  ➔ This will retrieve those records where id is between 30950 and 30960.

- Select * from student where DOB BETWEEN '2003-01-01' and '2003-12-31';

  ➔ This will retrieve the records where DOB is between January 2003 and December 2003.

SQL ALIASES

➔ Aliases are used to give temporary names to a table, column in a table.
➔ An alias only exists for the duration of that query only.
➔ An alias is created with **AS** keyword.

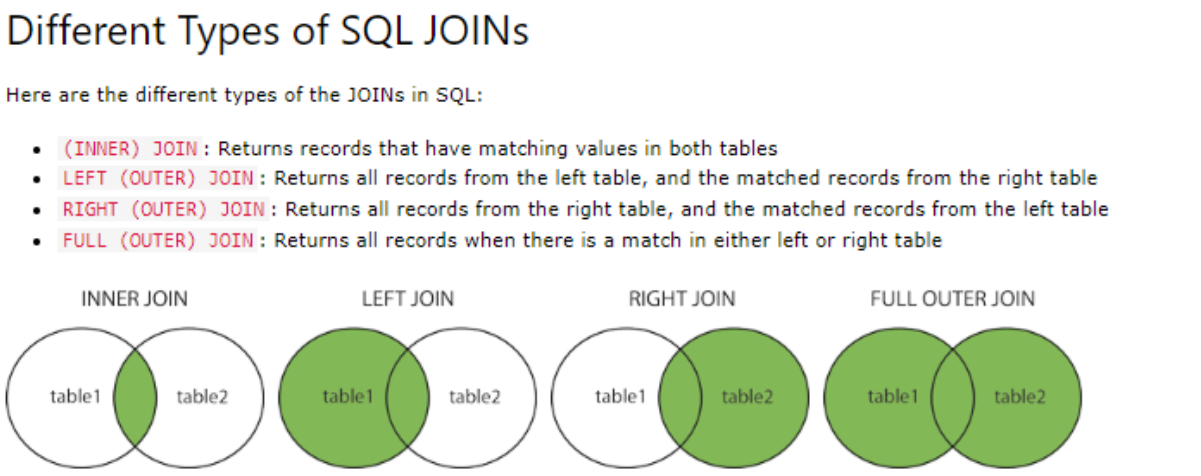
Example

- Select id as Student_ID, name as STUDENT_NAME from student;

  ➔ This will retrieve id values and name values from student table by adding column names as STUDENT_ID and STUDENT_NAME.

SQL JOINS

➔ A Join clause is used to combine rows from two or more tables based on a related common column between them.



Examples

- Select student.name, certified_students.sid from student **INNER JOIN** certified_students **on** student.id = certified_students.sid;

  ➔ This will return all the records with the ids that are available in both the tables.
  ➔ The **INNER JOIN** returns the records that match in both the tables.
- Select student.name, certified_students.cid from student **LEFT JOIN** certified_students on student.id = certified_student.id.

  ➔ This will return all the records of student table even if they don't have cid.
  ➔ If they don't have cid it will return null.
  ➔ The **LEFT JOIN** returns the all the records from left table(student), even if there are no matches in the right table(certified_students).

- Select student.name,certified_students.cid from certified_students **RIGHT JOIN** student on student.id = certified_students.sid;

  ➔ This will return all the records from certified_students even if they don't have name.
  ➔ The **RIGHT JOIN** returns the all the records from rightt table(certified_students), even if there are no matches in the left table(student).

- Select student.name, certified_students.sid from student **FULL OUTER JOIN** certified_students **on** student.id = certified_students.sid;

  ➔ The **FULL OUTER JOIN** will return all records when there is a match in left table or right table records.

SQL UNION

➔ It is used to combine result set of two more.
➔ Every select statement within **UNION** must have the same number of columns.
➔ The column must also have similar data types.
➔ The **UNION** operator returns only distinct values by default.
➔ If we want to allow duplicate values, then **UNION ALL** can be used.

Examples

- Select id from student **UNION** select sid from certified_students;

  ➔ This will return all the distinct IDs from both the tables.

- Select id from student **UNION ALL** select sid from certified_students;

  ➔ This will return all the ids from both the tables including duplicates.

SQL GROUP BY

➔ The GROUP BY statement groups rows that have the same values into summary rows like "find the number of customers in each country".
➔ The GROUP BY statement is often used with aggregate functions (COUNT( ), MAX( ), MIN(), SUM( ), AVG( ).

EXAMPLE

- Select count(customerId), country from customers Group By country;

  ➔ This will return all the records with count of each customerId in each country.

SQL HAVING Clause

➔ The SQL HAVING Clause is used because we cannot use where clause with aggregate functions like MAX( ), MIN( ), COUNT( ), etc.

Example

- Select count(customerId), country from customers group by country **having** count(customerId) > 5;

  ➔ This will return all the countries where count of customerId > 5.

SQL EXISTS

➔ The EXISTS operator is used to check the existence of any record in a subquery.
➔ The EXISTS operator returns true if the subquery returns one or more records.

Example

- Select name from student where **EXISTS** (select cid from certified_students where certified_students.id = students.id);

  ➔ This will return all the names of certified students.

SQL ANY

➔ The **any** operator allows us to perform a comparison between a single column value and a range of other values.
➔ It returns a Boolean value as a result.
➔ It returns true if any of the subquery values meet the condition.

Example

- Select name from student where id = **ANY** (select cid from certified_students where cid = id);

  ➔ This will retrieve all the names of students who are certified.

SQL ALL

➔ The **all** operator allows us to perform a comparison between a single column value and a range of other values.
➔ It returns a Boolean value as a result.
➔ It returns true if any of the subquery values meet the condition.

Example

- Select name from student where id = **ALL**(select cid from certified_students where cid = id);

  ➔ This will retrieve all the names of students who are certified.

SQL SELECT INTO

➔ The SELECT INTO statement copies values from one existing table into another new table.

Examples

- **Select** * **into** certifications from certified_students;

  ➔ This will copy all the details of certified_students table into certifications table.

- Select name **into** certified_students from students;

  ➔ This will copy the name column values form students table to certified_students.

- Select * **into** student_backup **IN** 'klu' from students;

  ➔ Here we are using IN clause to point out another database called as 'klu' where our student_backup table is going to be created.
  ➔ This will copy all the data from students table into student_backup table.

SQL INSERT INTO SELECT

➔ The **INSERT INTO SELECT** is used to insert values of one existing table into another existing table;
➔ The **INSERT INTO SELECT** is only possible if the data types of both source and destination tables matches.

Example

- **INSERT INTO** students(name,id,cgpa) **select** cname,cid,score from certified_students;

  ➔ This will copy all the values from certified_students table to students table because the data types and order of columns in both the tables matches.
- **INSERT INTO** students(name,id,cgpa) select cname,cid,score from certified_students WHERE cname = 'AWS-CP';

➔ This will copy all the records where cname = ‘AWS-CP’.

SQL CASE

➔ It is like switch case in c or java where it returns a value when a specified first condition is met.

Examples

- Select id, name,
  CASE
  WHEN cgpa > 9 **THEN** ‘cgpa is above 9’
  WHEN cgpa > 8.5 **THEN** ‘cgpa is above 8.5’
  ELSE ‘cgpa is less than 8.5’
  END AS CGPACriteria FROM student;

➔ This will retrieve id, name column values from cgpa with an additional alias column as ‘CGPACriteria’ with the message as ‘cgpa is’ based on the student cgpa.

- Select id, name from student ORDER BY (CASE WHEN ID IS NULL THEN named ELSE id);

➔ This will retrieve id,name column values based on ascending order of id, if id is null then id field will be replaced by name;

STORED PROCEDURE

➔ A stored procedure is a prepared SQL code that we can save and reuse it.

Examples

- **CREATE PROCEDURE** selectAll @ID INT **AS** Select * from students where id = @ID Go;
- **EXEC** selectAll @ID = 30959;

➔ This will return all the records where id is 30959.

- **CREATE PROCEDURE** selectAll @ID INT, @NAME nvarchar(15) **AS** Select * from students where id = @ID and name = @NAME;
- **EXEC** selectAll @ID = 30959, @NAME = 'kalyan';

  ➔ This will return all the records where id is 30959 and name is kalyan.

SQL COMMENTS

- Single line comments in sql are represented with '- - '.
- Multi line comments in sql are represented with '/* */'.

SQL CREATE Database

➔ It is used to create a database.

Example

- CREATE DATABASE klu;

SQL DROP Database

➔ It is used to drop or delete an existing database.

Example

- DROP Database klu;

SQL BACKUP Database

➔ It is used in SQL server to create a full backup of an existing database.

Example

- **BACKUP Database** klu **TO DISK =** "File path **";**

  ➔ This will create a full **BACKUP** klu database into the provide file path.

- **BACKUP Database** klu **TO DISK =** "FILE PATH" **WITH DIFFERENTIAL**;

  ➔ It will backup only the parts of database that have changed.

SQL CREATE TABLE

➔ It is used to create a table.

Example

- **CREATE TABLE** students(id int, name varchar(20), cgpa float);

  ➔ This will create a table named as student with id, name and cgpa attributes.

SQL DROP TABLE

➔ It is used to drop or delete an existing table.

Example

- **DROP TABLE** students;

  ➔ This will delete students table.

SQL TRUNCATE TABLE

➔ It is used to delete data inside a table but it won't delete the table.

Example

- **TRUNCATE TABLE** students;

  ➔ This will delete the data inside the students table.

SQL ALTER TABLE

➔ It is used to ADD, DELETE or MODIFY columns in an existing table.

➔ It is also used to ADD or DROP various constraints in an existing table.

Examples

- **ALTER TABLE** students **ADD** year int;

  ➔ This will add an extra column named as year to the existing student table.

- **ALTER TABLE** students **DROP COLUMN** year;

  ➔ This will drop the year column from the students table.

- **ALTER TABLE** students **RENAME COLUMN** year **TO** study_year;

  ➔ This will change the column name year to study_year in students table.

- **ALTER TABLE** students **MODIFY COLUMN** id bigint;

  ➔ This will change the column datatype of id from int to bigint.

SQL CONSTRAINTS

➔ SQL constraints are used to specify rules for the data in a table.
➔ They can be used to limit the type of data that go into the table.
➔ CONSTRAINTS can be table level or column level.
➔ The CONSTRAINTS are to be defined while creating or while altering a 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

EXAMPLES

NOT NULL

- CREATE TABLE stu_details(id int **NOT NULL**, name varchar(50) **NOT NULL**) ;

  ➔ This will define a **NOT NULL CONSTRAINT** on id and name attributes i.e. the id and name values must not be **NULL.**

- **ALTER TABLE** stu_details **ALTER COLUMN** age int **NOT NULL**;

  ➔ This will define a **NOT NULL CONSTRAINT** for age attribute in stu_details table.

UNIQUE

- CREATE TABLE employee (eid int, ename varchar(30), esal float, **UNIQUE** (eid));

  ➔ This will define a **UNIQUE** constraint on eid i.e. it will not allow duplicate values for eid column.

- ALTER TABLE employee **ADD UNIQUE** (EID);

  ➔ This will add an UNIQUE constraint on EID attribute in employee table.

- CREATE TABLE employee (eid int, ename varchar(30), esal float, **CONSTRAINT** empdetails **UNIQUE** (eid,ename));

  ➔ This will define a **UNIQUE** constraint on eid and ename i.e. it will not allow duplicate values for eid and ename columns.

PRIMARY KEY

➔ A primary key constraint is used to uniquely identify each record in a table.
➔ Primar key values must contain **unique** values and cannot contain **null** values.
➔ A table can have only one primary key.

- CREATE TABLE employee (eid int, ename varchar(30), esal float, **PRIMARY KEY** (eid));

  ➔ This will define a **PRIMARY KEY** constraint on eid i.e. it will not allow duplicate values for eid column and null values.

- ALTER TABLE employee **ADD PRIMARY KEY** (EID);

  ➔ This will add an **PRIMARY KEY** constraint on EID attribute in employee table.
- CREATE TABLE employee (eid int, ename varchar(30), esal float, **CONSTRAINT** empdetails **PRIMARY KEY**(eid,ename));

  ➔ This will define a **PRIMARY KEY** constraint on eid and ename i.e. it will not allow duplicate values as well as null values for eid and ename columns.

FOREIGN KEY

➔ It is used to prevent actions that would destroy links between tables.
➔ It is a field or collection of fields in one table that refers to **primary key** in another table.
➔ The table with the **foreign key** is called **child table** and the table with **primary key** is called **parent table.**

Examples

- CREATE TABLE certified_studnets(cid int, cname varchar(30), **PRIMARY KEY** (cid),
  FOREIGN KEY (id) **REFERENCES** students(id);

➔ This will create a table named as certified_students where cid is PRIMARY KEY and id as FOREIGN KEY which is a PRIMARY KEY in students table.

- ALTER TABLE certified_students **ADD FOREIGN KEY** (id) **REFERENCES** students(id);

  ➔ This will alter a table named as certified_students where cid is PRIMARY KEY and id as FOREIGN KEY which is a PRIMARY KEY in students table.

CHECK

➔ It is used to **LIMIT** the value range that can be placed in a column.
➔ If we define a **CHECK** constraint on a column, it will allow only certain values for this column.

Example

- **Create table** voters (vid int, name varchar(20), age int **CHECK (AGE >= 18)**);

  ➔ This will create a **CHECK** for age stating that the age should be >= 18.
  ➔ During insertion if we provide age below 18, it will throw an error and the record will not get inserted.

DEFAULT

➔ It is used to set a default value for a column.
➔ The specified default value will be added to all new records if no value is specified.

Examples

- Create table stu_det(id int, name varchar(30) **DEFAULT 'student'**, cgpa float);

➔ This will create a table named as stu_det with id, name attributes where name having DEFAULT value as 'student'.

- ALTER TABLE stu_det ALTER id **SET DEFAULT** 21000000;

  ➔ This will alter a table named as stu_det where id is altered with DEFAULT value as 21000000.

AUTO INCREMENT

➔ It generates a unique number automatically when a new record is inserted.
➔ Often this is the primary key field that we would like to be created automatically whenever a new record is inserted.
➔ By default the value of AUTO INCREMENT starts from 1.

Examples

- Create table stu (id int **AUTO_INCREMENT**, name varchar (30));

  ➔ This will create a table stu where we need not to insert id values.

- Insert Into stu ('kalyan');

  ➔ This will inserd a records with id as 1 and name as kalyan.

- ALTER TABLE stu **AUTO_INCREMENT** = 100;

  ➔ This will start the **AUTO_INCREMENT** from 100.

SQL VIEWS

➔ A view is a virtual table based on the result-set of an SQL statement.
➔ A view contains columns and rows just like a real table.

Example

- **CREATE VIEW [TOP STUDENTS DETAILS] AS** SELECT name, id from students where cgpa > 9;

  ➔ This will create a virtual table or **view** AS TOP STUDENTS DETAILS with name, id as attributes consisting records of students where cgpa > 9.

- Select * from [TOP STUDENTS DETAILS];

  ➔ This will retrieve all the records from TOP STUDENTS DETAILS **view**.

SQL DATA TYPES

➔ The data type of column defines what type of value it should hold.
➔ Each column should have a name and a data type.

String Data Types

Data type	Description
CHAR(size)	A FIXED length string (can contain letters, numbers, and special characters). The *size* parameter specifies the column length in characters - can be from 0 to 255. Default is 1
VARCHAR(size)	A VARIABLE length string (can contain letters, numbers, and special characters). The *size* parameter specifies the maximum string length in characters - can be from 0 to 65535
BINARY(size)	Equal to CHAR(), but stores binary byte strings. The *size* parameter specifies the column length in bytes. Default is 1
VARBINARY(size)	Equal to VARCHAR(), but stores binary byte strings. The *size* parameter specifies the maximum column length in bytes.
TINYBLOB	For BLOBs (Binary Large Objects). Max length: 255 bytes
TINYTEXT	Holds a string with a maximum length of 255 characters
TEXT(size)	Holds a string with a maximum length of 65,535 bytes
BLOB(size)	For BLOBs (Binary Large Objects). Holds up to 65,535 bytes of data
MEDIUMTEXT	Holds a string with a maximum length of 16,777,215 characters
MEDIUMBLOB	For BLOBs (Binary Large Objects). Holds up to 16,777,215 bytes of data
LONGTEXT	Holds a string with a maximum length of 4,294,967,295 characters
LONGBLOB	For BLOBs (Binary Large Objects). Holds up to 4,294,967,295 bytes of data
ENUM(val1, val2, val3, ...)	A string object that can have only one value, chosen from a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted. The values are sorted in the order you enter them
SET(val1, val2, val3, ...)	A string object that can have 0 or more values, chosen from a list of possible values. You can list up to 64 values in a SET list

Date and Time Data Types

Data type	Description
DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'
DATETIME(*fsp*)	A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time
TIMESTAMP(*fsp*)	A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT_TIMESTAMP and ON UPDATE CURRENT_TIMESTAMP in the column definition
TIME(*fsp*)	A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'
YEAR	A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000. MySQL 8.0 does not support year in two-digit format.

Numeric Data Types

Data type	Description
BIT(*size*)	A bit-value type. The number of bits per value is specified in *size*. The *size* parameter can hold a value from 1 to 64. The default value for *size* is 1.
TINYINT(*size*)	A very small integer. Signed range is from -128 to 127. Unsigned range is from 0 to 255. The *size* parameter specifies the maximum display width (which is 255)
BOOL	Zero is considered as false, nonzero values are considered as true.
BOOLEAN	Equal to BOOL
SMALLINT(*size*)	A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The *size* parameter specifies the maximum display width (which is 255)
MEDIUMINT(*size*)	A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The *size* parameter specifies the maximum display width (which is 255)
INT(*size*)	A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The *size* parameter specifies the maximum display width (which is 255)
INTEGER(*size*)	Equal to INT(size)
BIGINT(*size*)	A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The *size* parameter specifies the maximum display width (which is 255)
FLOAT(*size*, *d*)	A floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. This syntax is deprecated in MySQL 8.0.17, and it will be removed in future MySQL versions
FLOAT(*p*)	A floating point number. MySQL uses the *p* value to determine whether to use FLOAT or DOUBLE for the resulting data type. If *p* is from 0 to 24, the data type becomes FLOAT(). If *p* is from 25 to 53, the data type becomes DOUBLE()
DOUBLE(*size*, *d*)	A normal-size floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter
DOUBLE PRECISION(*size*, *d*)	
DECIMAL(*size*, *d*)	An exact fixed-point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. The maximum number for *size* is 65. The maximum number for *d* is 30. The default value for *size* is 10. The default value for *d* is 0.
DEC(*size*, *d*)	Equal to DECIMAL(size,d)

STANDARD EXAMPLES

_INNER JOIN

- Given the **CITY** and **COUNTRY** tables, query the sum of the populations of all cities where the CONTINENT is 'Asia'.
- CITY. CountryCode and COUNTRY. Code are matching key columns .

➔ **select sum (CITY. Population) from COUNTRY inner join CITY on country. Code = city. CountryCode where country. Continent = 'Asia';**

CONCAT

Query the following two values from the **STATION** table:

1. The sum of all values in LAT_N rounded to a scale of 2 decimal places.
2. The sum of all values in LONG_W rounded to a scale of 2 decimal places.

➔ **SELECT CONCAT(ROUND(SUM(LAT_n), 2), ' ', ROUND(SUM(LONG_W), 2)) AS ConcatenatedResult FROM Station;**

STRING LENGTH

- Query the two cities in **STATION** with the shortest and longest CITY names, as well as their respective lengths (i.e.: number of characters in the name). If there is more than one smallest or largest city, choose the one that comes first when ordered alphabetically.

➔ **SELECT CITY, LENGTH(CITY) FROM STATION ORDER BY LENGTH(CITY) ASC, CITY LIMIT 1;**

➔ **SELECT CITY, LENGTH(CITY) FROM STATION ORDER BY LENGTH(CITY) DESC, CITY LIMIT 1;**

NESTED SELECTS WITH ROUND

- Query the Western Longitude (LONG_W) for the largest Northern Latitude (LAT_N) in **STATION** that is less than 137.2345. Round your answer to decimal places.

  ➔ **select round(LONG_W,4) from STATION where LAT_N = (select max(LAT_N) from station where LAT_N < 137.2345);**

MULTIPLE ORDER BY

- Query the Name of any student in **STUDENTS** who scored higher than 75 Marks. Order your output by the last three characters of each name. If two or more students both have names ending in the same last three characters (i.e.: Bobby, Robby, etc.), secondary sort them by ascending ID.

  ➔ Select name from students where marks > 75 ORDER BY RIGHT(name,3), ID ASC;

MAX WITH MULIPLICATION

- We define an employee's total earnings to be their monthly salary * months worked, and the maximum total earnings to be the maximum total earnings for any employee in the **Employee** table. Write a query to find the maximum total earnings for all employees as well as the total number of employees who have maximum total earnings. Then print these values as 2 space-separated integers.

  ➔ SELECT MAX(months * salary), COUNT(employee_id) FROM EMPLOYEE WHERE
  (months * salary) = (SELECT MAX(months * salary) FROM EMPLOYEE);

CEIL AND REPLACE FUNCTIONS

- Samantha was tasked with calculating the average monthly salaries

  for all employees in the **EMPLOYEES** table, but did not realize her

keyboard's 0 key was broken until after completing the calculation. She wants your help finding the difference between her miscalculation (using salaries with any zeros removed), and the actual average salary.

Write a query calculating the amount of error (i.e.: average monthly salaries), and round it up to the next integer.

➔ SELECT CEIL(AVG(Salary) – AVG(REPLACE(Salary,0,””))) from EMPLOYEES;