MySQL Constraints

The constraint in MySQL is used to specify the rule that allows or restricts what values/data will be stored in the table. They provide a suitable method to ensure data accuracy and integrity inside the table. It also helps to limit the type of data that will be inserted inside the table. If any interruption occurs between the constraint and data action, the action is failed.

Types of MySQL Constraints

Constraints in MySQL is classified into two types:

1. **Column Level Constraints:** These constraints are applied only to the single column that limits the type of particular column data.
2. **Table Level Constraints:** These constraints are applied to the entire table that limits the type of data for the whole table.

How to create constraints in MySQL

We can define the constraints during a table created by using the CREATE TABLE statement. MySQL also uses the ALTER TABLE statement to specify the constraints in the case of the existing table schema.

Syntax

The following are the syntax to create a constraints in table:

1. **CREATE** **TABLE** new\_table\_name (
2. col\_name1 datatype **constraint**,
3. col\_name2 datatype **constraint**,
4. col\_name3 datatype **constraint**,
5. .........
6. );

Constraints used in MySQL

The following are the most common constraints used in the MySQL:

* NOT NULL
* DEFAULT
* PRIMARY KEY
* AUTO\_INCREMENT
* UNIQUE
* INDEX
* ENUM
* FOREIGN KEY

Let us discuss each of these constraints in detail.

NOT NULL Constraint

This constraint specifies that the column cannot have NULL or empty values. The below statement creates a table with NOT NULL constraint.

1. **CREATE** **TABLE** Student(

Id **INTEGER**,

LastName TEXT NOT NULL,

FirstName TEXT NOT NULL,

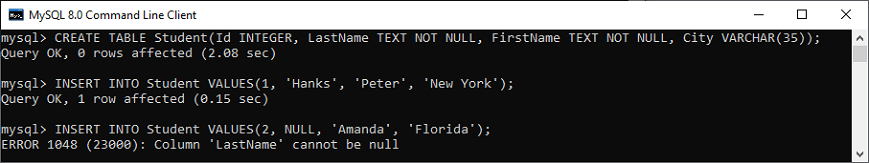
City **VARCHAR**(35)

);

Execute the queries listed below to understand how it works:

1. **INSERT** **INTO** Student **VALUES**(1, 'Hanks', 'Peter', 'New York');
3. **INSERT** **INTO** Student **VALUES**(2, NULL, 'Amanda', 'Florida');

**Output**



In the above image, we can see that the first INSERT query executes correctly, but the second statement fails and gives an error that says column LastName cannot be null.

UNIQUE Constraint

This constraint ensures that all values inserted into the column will be unique. It means a column cannot stores duplicate values. MySQL allows us to use more than one column with UNIQUE constraint in a table. The below statement creates a table with a UNIQUE constraint:

1. **CREATE** **TABLE** ShirtBrands(

Id **INTEGER**,

BrandName **VARCHAR**(40) **UNIQUE**,

**Size** **VARCHAR**(30)

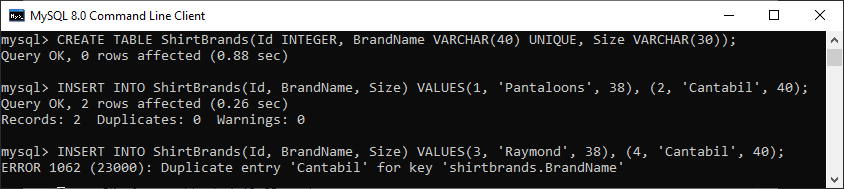
);

Execute the queries listed below to understand how it works:

1. **INSERT** **INTO** ShirtBrands(Id, BrandName, **Size**) **VALUES**(1, 'Pantaloons', 38), (2, 'Cantabil', 40);
3. **INSERT** **INTO** ShirtBrands(Id, BrandName, **Size**) **VALUES**(1, 'Raymond', 38), (2, 'Cantabil', 40);

**Output**

In the below output, we can see that the first INSERT query executes correctly, but the second statement fails and gives an error that says: Duplicate entry 'Cantabil' for key BrandName.



DEFAULT Constraint

This constraint is used to set the default value for the particular column where we have not specified any value. It means the column must contain a value, including NULL.

For example, the following statement creates a table "Persons" that contains DEFAULT constraint on the "City" column. If we have not specified any value to the City column, it inserts the default value:

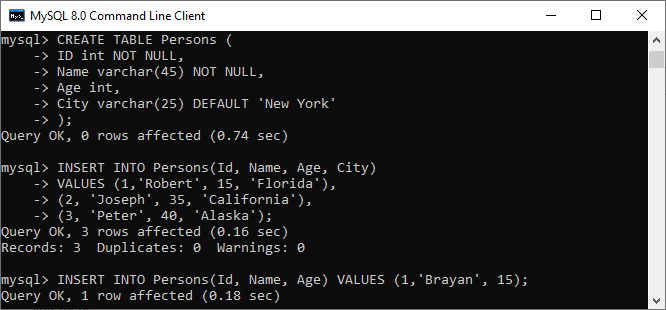
1. **CREATE** **TABLE** Persons (
2. ID **int** NOT NULL,
3. **Name** **varchar**(45) NOT NULL,
4. Age **int**,
5. City **varchar**(25) **DEFAULT** 'New York'
6. );

Execute the listed queries to insert the values into the table:

1. **INSERT** **INTO** Persons(Id, **Name**, Age, City)
2. **VALUES** (1,'Robert', 15, 'Florida'),
3. (2, 'Joseph', 35, 'California'),
4. (3, 'Peter', 40, 'Alaska');
6. **INSERT** **INTO** Persons(Id, **Name**, Age) **VALUES** (1,'Brayan', 15);

**Output**

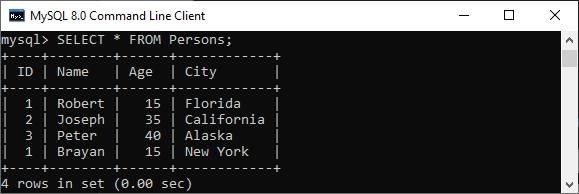
In the below output, we can see that the first insert query that contains all fields executes successfully, while the second insert statement does not contain the "City" column but also executed successfully. It is because it has a default value.



Now, executes the following statement to validate the default value for the 4th column:

1. **SELECT** \* **FROM** Persons;

We can see that it works perfectly. It means default value "New York" stored automatically in the City column.



PRIMARY KEY Constraint

This constraint is used to identify each record in a table uniquely. If the column contains primary key constraints, then it cannot be null or empty. A table may have duplicate columns, but it can contain only one primary key. It always contains unique value into a column.

The following statement creates a table "Person" and explains the use of this primary key more clearly:

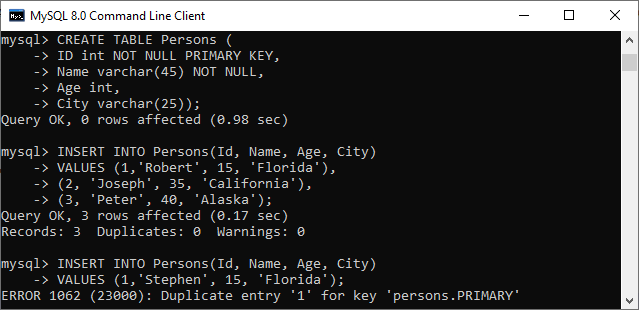
1. **CREATE** **TABLE** Persons (
2. ID **int** NOT NULL **PRIMARY** **KEY**,
3. **Name** **varchar**(45) NOT NULL,
4. Age **int**,
5. City **varchar**(25));

Next, use the insert query to store data into a table:

1. **INSERT** **INTO** Persons(Id, **Name**, Age, City)
2. **VALUES** (1,'Robert', 15, 'Florida') ,
3. (2, 'Joseph', 35, 'California'),
4. (3, 'Peter', 40, 'Alaska');
6. **INSERT** **INTO** Persons(Id, **Name**, Age, City)
7. **VALUES** (1,'Stephen', 15, 'Florida');

**Output**

In the below output, we can see that the first insert query executes successfully. While the second insert statement fails and gives an error that says: Duplicate entry for the primary key column.



AUTO\_INCREMENT Constraint

This constraint automatically generates a unique number whenever we insert a new record into the table. Generally, we use this constraint for the primary key field in a table.

We can understand it with the following example where the id column going to be auto-incremented in the Animal table:

1. **CREATE** **TABLE** Animals(
2. id **int** NOT NULL AUTO\_INCREMENT,
3. **name** **CHAR**(30) NOT NULL,
4. **PRIMARY** **KEY** (id));

Next, we need to insert the values into the "Animals" table:

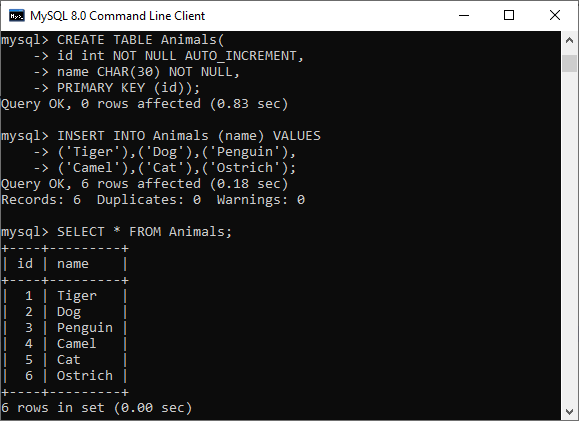
1. **INSERT** **INTO** Animals (**name**) **VALUES**
2. ('Tiger'),('Dog'),('Penguin'),
3. ('Camel'),('Cat'),('Ostrich');

Now, execute the below statement to get the table data:

1. **SELECT** \* **FROM** Animals;

**Output**

In the output, we can see that I have not specified any value for the auto-increment column, so MySQL automatically generates a unique number in the sequence order for this field.



ENUM Constraint

The ENUM data type in MySQL is a string object. It allows us to limit the value chosen from a list of permitted values in the column specification at the time of table creation. 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.

The following illustration creates a table named "shirts" that contains three columns: id, name, and size. The column name "size" uses the ENUM data type that contains small, medium, large, and x-large sizes.

1. **CREATE** **TABLE** Shirts (
2. id **INT** **PRIMARY** **KEY** AUTO\_INCREMENT,
3. **name** **VARCHAR**(35),
4. **size** ENUM('small', 'medium', 'large', 'x-large')
5. );

Next, we need to insert the values into the "Shirts" table using the below statements:

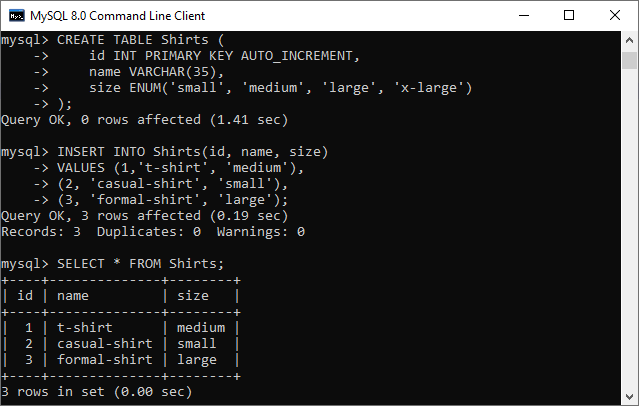
1. **INSERT** **INTO** Shirts(id, **name**, **size**)
2. **VALUES** (1,'t-shirt', 'medium'),
3. (2, 'casual-shirt', 'small'),
4. (3, 'formal-shirt', 'large');

Now, execute the SELECT statement to see the inserted values into the table:

1. **SELECT** \* **FROM** Shirts;

**Output**

We will get the following output:



INDEX Constraint

This constraint allows us to create and retrieve values from the table very quickly and easily. An index can be created using one or more than one column. It assigns a ROWID for each row in that way they were inserted into the table.

The following illustration creates a table named "shirts" that contains three columns: id, name, and size.

1. **CREATE** **TABLE** Shirts (
2. id **INT** **PRIMARY** **KEY** AUTO\_INCREMENT,
3. **name** **VARCHAR**(35),
4. **size** ENUM('small', 'medium', 'large', 'x-large')
5. );

Next, we need to insert the values into the "Shirts" table using the below statements:

1. **INSERT** **INTO** Shirts(id, **name**, **size**)
2. **VALUES** (1,'t-shirt', 'medium'),
3. (2, 'casual-shirt', 'small'),
4. (3, 'formal-shirt', 'large');

Now, execute this statement for creating index:

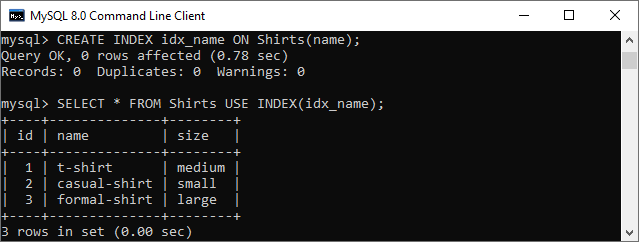
1. **CREATE** **INDEX** idx\_name **ON** Shirts(**name**);

We can use the query below to retrieve the data using the index column:

1. **SELECT** \* **FROM** Shirts USE **INDEX**(idx\_name);

**Output**

The following output appears:



Foreign Key Constraint

This constraint is used to link two tables together. It is also known as the referencing key. A foreign key column matches the primary key field of another table. It means a foreign key field in one table refers to the primary key field of another table.

Let us consider the structure of these tables: Persons and Orders.

**Table: Persons**

1. **CREATE** **TABLE** Persons (
2. Person\_ID **int** NOT NULL **PRIMARY** **KEY**,
3. **Name** **varchar**(45) NOT NULL,
4. Age **int**,
5. City **varchar**(25)
6. );

**Table: Orders**

1. **CREATE** **TABLE** Orders (
2. Order\_ID **int** NOT NULL **PRIMARY** **KEY**,
3. Order\_Num **int** NOT NULL,
4. Person\_ID **int**,
5. **FOREIGN** **KEY** (Person\_ID) **REFERENCES** Persons(Person\_ID)
6. );

In the above table structures, we can see that the "Person\_ID" field in the "Orders" table points to the "Person\_ID" field in the "Persons" table. The "Person\_ID" is the PRIMARY KEY in the "Persons" table, while the "Person\_ID" column of the "Orders" table is a FOREIGN KEY.

**Output**

Our table contains the following data:

