Graphical user interface, website

Description automatically generated

Graphical user interface, text

Description automatically generated

First, create two tables called members and committees:

CREATE TABLE members1 (

member\_id INT AUTO\_INCREMENT,

name VARCHAR(100),

PRIMARY KEY (member\_id)

);

CREATE TABLE committees1 (

committee\_id INT AUTO\_INCREMENT,

name VARCHAR(100),

PRIMARY KEY (committee\_id)

);

INSERT INTO members1(name)

VALUES('Harish'),('Jane'),('Praveen'),('David'),('Wasim');

INSERT INTO committees1(name)

VALUES('Harish'),('Praveen'),('Wasim'),('Joe');

**MySQL INNER JOIN clause**

* The inner join clause joins two tables based on a condition which is known as a join predicate.
* The inner join clause compares each row from the first table with every row from the second table.
* If values from both rows satisfy the join condition, the inner join clause creates a new row whose column contains all columns of the two rows from both tables and includes this new row in the result set. In other words, the inner join clause includes only matching rows from both tables.
* If the join condition uses the equality operator (=) and the column names in both tables used for matching are the same, and you can use the USING clause instead:

**SELECT column\_list**

**FROM table\_1**

**INNER JOIN table\_2 USING (column\_name);**

SELECT

m.member\_id,

m.name AS member,

c.committee\_id,

c.name AS committee1

FROM

members1 m

INNER JOIN committees1 c ON c.name = m.name;

Diagram, venn diagram

Description automatically generated

Because both tables use the same column to match, you can use the USING clause as shown in the following query:

SELECT

m.member\_id,

m.name AS member,

c.committee\_id,

c.name AS committee

FROM

members1 m

INNER JOIN committees1 c USING(name);

**MySQL LEFT JOIN clause**

* Like an inner join, a left join also requires a join predicate. When joining two tables using a left join, the concepts of left and right tables are introduced.
* The left join selects data starting from the left table. For each row in the left table, the left join compares with every row in the right table.
* If the values in the two rows satisfy the join condition, the left join clause creates a new row whose columns contain all columns of the rows in both tables and includes this row in the result set.
* If the values in the two rows are not matched, the left join clause still creates a new row whose columns contain columns of the row in the left table and NULL for columns of the row in the right table.
* In other words, the left join selects all data from the left table whether there are matching rows exist in the right table or not.
* In case there are no matching rows from the right table found, the left join uses NULLs for columns of the row from the right table in the result set.

**SELECT column\_list**

**FROM table\_1**

**INNER JOIN table\_2 USING (column\_name);**

Diagram

Description automatically generated

Because both tables use the same column to match, you can use the USING clause as shown in the following query:

SELECT

m.member\_id,

m.name AS member,

c.committee\_id,

c.name AS committee

FROM

members1 m

LEFT JOIN committees1 c USING(name);

**To find members who are not the committee members, you add a WHERE clause and IS NULL operator as follows:**

**SELECT**

**m.member\_id,**

**m.name AS member,**

**c.committee\_id,**

**c.name AS committee**

**FROM**

**members1 m**

**LEFT JOIN committees1 c USING(name)**

**WHERE c.committee\_id IS NULL;**

**Generally, this query pattern can find rows in the left table that do not have corresponding rows in the right table.**

Diagram, venn diagram

Description automatically generated

**MySQL RIGHT JOIN clause**

The right join clause selects all rows from the right table and matches rows in the left table.

If a row from the right table does not have matching rows from the left table, the column of the left table will have NULL in the result set.

Here is the syntax of the right join:

SELECT column\_list

FROM table\_1

RIGHT JOIN table\_2 ON join\_condition;

SELECT column\_list

FROM table\_1

RIGHT JOIN table\_2 USING (column\_name);

**SELECT**

**m.member\_id,**

**m.name AS member,**

**c.committee\_id,**

**c.name AS committee**

**FROM**

**members1 m**

**RIGHT JOIN committees1 c on c.name = m.name;**

Diagram

Description automatically generated

**SELECT**

**m.member\_id,**

**m.name AS member,**

**c.committee\_id,**

**c.name AS committee**

**FROM**

**members m**

**RIGHT JOIN committees c USING(name);**

**To find the committee members who are not in the members table, you use this query:**

**SELECT**

**m.member\_id,**

**m.name AS member,**

**c.committee\_id,**

**c.name AS committee**

**FROM**

**members1 m**

**RIGHT JOIN committees1 c USING(name)**

**WHERE m.member\_id IS NULL;**

Diagram, venn diagram

Description automatically generated

**MySQL CROSS JOIN clause**

Unlike the inner join, left join, and right join, the cross join clause does not have a join condition.

The cross join makes a Cartesian product of rows from the joined tables. The cross join combines each row from the first table with every row from the right table to make the result set.

Suppose the first table has n rows and the second table has m rows. The cross join that joins the tables will return nxm rows.

The following shows the syntax of the cross join clause:

**SELECT select\_list**

**FROM table\_1**

**CROSS JOIN table\_2;**

**SELECT**

**m.member\_id,**

**m.name AS member,**

**c.committee\_id,**

**c.name AS committee**

**FROM**

**members m**

**CROSS JOIN committees c;**

**Full Outer:**

**Left outer**

**Union**

**Right outer**

**CREATE TABLE user (**

**id INT NOT NULL AUTO\_INCREMENT,**

**name VARCHAR(255) NOT NULL,**

**qual varchar(30) NOT NULL,**

**PRIMARY KEY (id)**

**);**

**LOAD DATA INFILE 'd:/example/ex1.csv'**

**INTO TABLE user**

**FIELDS TERMINATED BY ','**

**ENCLOSED BY '"'**

**LINES TERMINATED BY '\n'**

**IGNORE 1 ROWS;**

**LOAD DATA INFILE 'ex1.csv'**

**INTO TABLE user**

**FIELDS TERMINATED BY ','**

**ENCLOSED BY '"'**

**LINES TERMINATED BY '\n'**

**IGNORE 1 ROWS;**

MySQL ENUM data type

In MySQL, an ENUM is a string object whose value is chosen from a list of permitted values defined at the time of column creation.

The ENUM data type provides the following advantages:

Compact data storage. MySQL ENUM uses numeric indexes (1, 2, 3, …) to represents string values.

Readable queries and output.

To define an ENUM column, you use the following syntax:

**CREATE TABLE table\_name (**

**...**

**col ENUM ('value1','value2','value3'),**

**...**

**);**

CREATE TABLE tickets1 (

id INT PRIMARY KEY AUTO\_INCREMENT,

title VARCHAR(255) NOT NULL,

priority ENUM('Low', 'Medium', 'High') NOT NULL

);

The priority column will accept only three values Low, Medium and High. Behind the scenes, MySQL maps each enumeration member to a numeric index. In this case, Low, Medium, and High are map to 1, 2 and 3 respectively.

**Inserting MySQL ENUM values**

To insert data into an ENUM column, you use the enumeration values in the predefined list. For example, the following statement inserts a new row into the tickets table.

INSERT INTO tickets1(title, priority)

VALUES('Scan virus for computer A', 'High');

Besides the enumeration values, you can use the numeric index of the enumeration member for inserting data into an ENUM column. For instance, the following statement inserts a new ticket with the Low priority:

INSERT INTO tickets1(title, priority)

VALUES('Upgrade Windows OS for all computers', 1);

In this example, instead of using the Low enumeration value, we used value 1. Since Low is mapped to 1, it is acceptable.

Let’s add some more rows to the tickets table:

INSERT INTO tickets1(title, priority)

VALUES('Install Google Chrome for srinithi', 'Medium'),

('Create a new user for the new employee Praveen', 'High');

Because we defined the priority as a NOT NULL column, when you insert a new row without specifying the value for the priority column, MySQL will use the first enumeration member as the default value.

See the following statement:

INSERT INTO tickets(title)

VALUES('Refresh the computer of Harish');

Filtering MySQL ENUM values

SELECT

\*

FROM

tickets

WHERE

priority = 'High';

SELECT

\*

FROM

tickets

WHERE

priority = 3;

**Sorting MySQL ENUM values**

SELECT

title, priority

FROM

Tickets1

ORDER BY priority DESC;

## MySQL generated column

When you create a new table, you specify the table columns in the CREATE TABLE statement. Then, you use the INSERT, UPDATE, and DELETE statements to modify directly the data in the table columns.

MySQL 5.7 introduced a new feature called the generated column. Columns are generated because the data in these columns are computed based on predefined expressions.

DROP TABLE IF EXISTS contacts;

CREATE TABLE contacts1 (

id INT AUTO\_INCREMENT PRIMARY KEY,

first\_name VARCHAR(50) NOT NULL,

last\_name VARCHAR(50) NOT NULL,

email VARCHAR(100) NOT NULL

);

To get the full name of a contact1, you use the CONCAT() function as follows:

SELECT

id,

CONCAT(first\_name, ' ', last\_name),

email

FROM

Contacts1;

By using the MySQL generated column, you can recreate the contacts table as follows:

DROP TABLE IF EXISTS contacts1;

CREATE TABLE contacts1 (

id INT AUTO\_INCREMENT PRIMARY KEY,

first\_name VARCHAR(50) NOT NULL,

last\_name VARCHAR(50) NOT NULL,

fullname varchar(101) GENERATED ALWAYS AS (CONCAT(first\_name,' ',last\_name)),

email VARCHAR(100) NOT NULL

);

The GENERATED ALWAYS as (expression) is the syntax for creating a generated column.

To test the fullname column, you insert a row into the contacts table.

INSERT INTO contacts1(first\_name,last\_name, email)

VALUES('ketavarapu','pavan','ketavarapu.pavan@indiumsoft.com');

SELECT

\*

FROM

Contacts1;