Views in MySQL

VIEWS are virtual tables that do not store any data of their own but display data stored in other tables. In other words, VIEWS are nothing but SQL Queries. A view can contain all or a few rows from a table. A MySQL view can show data from one table or many tables.

MySQL Views syntax

Let's now look at the basic syntax used to create a view in MySQL.

CREATE VIEW `view_name` AS SELECT statement;
WHERE

- **"CREATE VIEW `view_name`"** tells MySQL server to create a view object in the database named `view_name`
- **"AS SELECT statement"** is the SQL statements to be packed in the MySQL Views. It can be a SELECT statement can contain data from one table or multiple tables.

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are a type of virtual tables allow users to do the following −

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

Creating Views

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic **CREATE VIEW** syntax is as follows −

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

You can include multiple tables in your SELECT statement in a similar way as you use them in a normal SQL SELECT query.

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  +----+----------+-----+-----------+----------+
```

Following is an example to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS;
```

Now, you can query CUSTOMERS_VIEW in a similar way as you query an actual table. Following is an example for the same.

```
SQL > SELECT * FROM CUSTOMERS_VIEW;
```

This would produce the following result.

```
+----------+-----+
| name     | age |
+----------+-----+
| Ramesh   |  32 |
| Khilan   |  25 |
| kaushik  |  23 |
| Chaitali |  25 |
```

MySQL Triggers

A MySQL **trigger** is a stored program (with queries) which is executed automatically to respond to a specific event such as insertion, updation or deletion occurring in a table.
There are 6 different types of triggers in MySQL:

1. Before Update Trigger:
As the name implies, it is a trigger which enacts before an update is invoked. If we write an update statement, then the actions of the trigger will be performed before the update is implemented.
Example:
Considering tables
: create table customer (acc_no integer primary key,

cust_name varchar(20),

avail_balance decimal);

create table mini_statement (acc_no integer,

avail_balance decimal,

foreign key(acc_no) references customer(acc_no) on delete cascade);

Inserting values in them:

insert into customer values (1000, "Fanny", 7000);

insert into customer values (1001, "Peter", 12000);

Trigger to insert (old) values into a mini_statement record (including account number and available balance as parameters) before updating any record in customer record/table:

delimiter //

create trigger update_cus

-> before update on customer

-> for each row

-> begin

-> insert into mini_statement values (old.acc_no, old.avail_balance);

-> end; //

Making updates to invoke trigger:

delimiter;

update customer set avail_balance = avail_balance + 3000 where acc_no = 1001;

update customer set avail_balance = avail_balance + 3000 where acc_no = 1000;

Output:

select *from mini_statement;

```
+--------+---------------+
| acc_no | avail_balance |
+--------+---------------+
|   1001 |         12000 |
|   1000 |          7000 |
+--------+---------------+
```

2 rows in set (0.0007 sec)

2. After Update Trigger:

As the name implies, this trigger is invoked after an updation occurs. (i.e., it gets implemented after an update statement is executed.).

Example:

We create another table:

```
create table micro_statement (acc_no integer,
                              avail_balance decimal,
        foreign key(acc_no) references customer(acc_no) on delete cascade);
```

Insert another value into customer:

insert into customer values (1002, "Janitor", 4500);

Query OK, 1 row affected (0.0786 sec)

Trigger to insert (new) values of account number and available balance into micro_statement record after an update has occurred:

```
delimiter //
create trigger update_after
    -> after update on customer
    -> for each row
    -> begin
    -> insert into micro_statement values(new.acc_no, new.avail_balance);
    -> end; //
```

Making an update to invoke trigger:

```
delimiter ;
update customer set avail_balance = avail_balance + 1500 where acc_no = 1002;
```

Output:

```
select *from micro_statement;
+--------+---------------+
| acc_no | avail_balance |
+--------+---------------+
|   1002 |          6000 |
+--------+---------------+
1 row in set (0.0007 sec)
```