PostgreSQL views

In PostgreSQL views are `virtual` tables that represent data of the underlying tables.

A views is a named query stored in a `PostgreSQL database server`.

A view is defined based on `one` and `more` tables wich are known as `base table`, and the query that defines the view is referred to as a defining query.

After creating view you can query data from it as you would from a regular table.

`Behind the schenes:`

PostgreSQL will rewrite the query against the view and its defining qeury, executing it to retrieve data from the base tables.

- A view is a virtual table that doesn't store data (except materialized view).
- It is a saved SQL query that defines a result set.

`Advantage:`

- `Data abstraction:` Hide the complexity of underlying tables.
- `Data security:` Restrict access to specific rows and columns.
- `Data independence:` Changes to underlying tables don't always affect view definitions.
- `Query simplification:` Create compex queries as simple view.
- `Improved redability:` Make complex queries more understandable.

```sql
-- list database views
\dv -- display views

-- or listing view using sql statments
SELECT table_name, table_schema
FROM information_schema.views
WHERE table_schema NOT IN (
    'information_schema','pg_catalog'
)
ORDER BY table_schem,table_name;

-- list all materialized view
SELECT * FROM pg_matviews;
```

Create views

The `CREATE VIEW` statement used to create a new view.

```sql
-- display the view info in psql
\d+ view_name;
```

```sql
-- syntax
CREATE VIEW view_name
AS
  query;


-- Create simple view
CREATE VIEW contact_view AS
SELECT first_name,  last_name,  email
FROM customer;

-- Create complex view
CREATE VIEW film_actor_category
AS
SELECT f.title as film, a.first_name || a.last_name as actor,
l.name, C.NAME as category FROM actor a
JOIN film_actor fa USING(actor_id)
JOIN film f USING(film_id)
JOIN language as l  USING(language_id)
JOIN film_category USING(film_id)
JOIN CATEGORY C USING(category_id);

-- Create a view based on another view
CREATE VIEW animation_film
AS
SELECT *
FROM film_actor_category
WHERE category='Animation';
```

`Query from view:`

```sql
-- syntax
SELECT * FROM view_name;

-- example
SELECT * FROM film_actor_category;

SELECT * FROM animation_film;
```

`Replacing a view:`

To change the defining query of a view, you use the `CREATE OR REPLACE VIEW` statement.

```sql
-- syntax
CREATE OR REPLACE view_name
AS
    query;
```

If the view already exists, the statement replaces the existing view; otherwise, it creates a new view.

```
-- change the contact_view defining query
CREATE OR REPLACE VIEW contact_view AS
SELECT
  first_name,
  last_name,
  email,
  phone
FROM
  customer
INNER JOIN address USING (address_id);
```

Drop view

The `DROP VIEW` statement allow you to remove view from the databse server.

```
-- syntax
DROP VIEW [IF EXISTS] view_name
[CASCADE | RESTRICT]
```

`Steps:`

- `First,` specify the view name which you want to remove.
- `Second,` use `if exists` to prevent an error if the view doesn't exist.
- `Third,` use `CASCADE` option to remove dependent object along with view or the `RESTRICT` option to reject the removal of the view.

`Droping multiple views:`

```
DROP VIEW IF EXISTS view1,view2, view3, ...
[CASCADE | RESTRICT];

-- To execute the DROP VIEW statement, you need to be the owner of the view or
have a DROP privilege on it.
```

Updatable view

A view can be updatable if it meets certain conditions. this means that you can `insert, update and delete` data from the underlying table via the view.

`Conditions:`

- `First,` The defining query of the view must have exactly one entry in the `FORM` clause, Which can be a table and another updatable view.

- `Second,` The defining query must not contain one of the following clauses at the top level.
    - GROUP BY
    - HAVING
    - LIMIT
    - OFFSET FETCH
    - DISTINCT
    - WITH
    - UNION
    - INTERSECT
    - EXCEPT
- `Third,` The section list of the defining query must not contain any:
    - Window functions
    - Set-returing function.
    - Aggregate functions.

An `updatable` view may contain both `updatable` and `non-updatable` columns. If you attempt to modify a `non-updatable` column, `PostgreSQL` will raise an `error`.

`Setting up a sample table:`

```
CREATE TABLE cities (
    id SERIAL PRIMARY KEY ,
    name VARCHAR(255),
    population INT,
    country VARCHAR(50)
);

INSERT INTO cities (name, population, country)
VALUES
    ('New York', 8419600, 'US'),
    ('Los Angeles', 3999759, 'US'),
    ('Chicago', 2716000, 'US'),
    ('Houston', 2323000, 'US'),
    ('London', 8982000, 'UK'),
    ('Manchester', 547627, 'UK'),
    ('Birmingham', 1141816, 'UK'),
    ('Glasgow', 633120, 'UK'),
    ('San Francisco', 884363, 'US'),
    ('Seattle', 744955, 'US'),
    ('Liverpool', 498042, 'UK'),
    ('Leeds', 789194, 'UK'),
    ('Austin', 978908, 'US'),
    ('Boston', 694583, 'US'),
    ('Manchester', 547627, 'UK'),
    ('Sheffield', 584853, 'UK'),
    ('Philadelphia', 1584138, 'US'),
    ('Phoenix', 1680992, 'US'),
    ('Bristol', 463377, 'UK'),
    ('Detroit', 673104, 'US');

SELECT * FROM cities;
```

`Create an updatable views:`

```

-- creat [city_us] view
CREATE VIEW city_us
AS
    SELECT *
    FROM cities
    WHERE country='US';

-- insert new row
INSERT INTO city_us(name,population,country)
VALUES ('San Jose', 983459, 'US');

-- check new entry available in both
SELECT * FROM CITIES;
SELECT * FROM city_us;


-- update data
UPDATE city_us
SET population = 1000000
WHERE name = 'New York';

-- delete data
DELETE FROM city_us
WHERE id = 21;
```

WITH CHECK OPTION

A simple view can be `updatable`.

To ensure that any data modification made through a view adheres to certain conditions in the view's definition, you use the `WITH CHECK OPTION` clause.

Typically, you specify the `WITH CHECK OPTION` when creating a view using the `CREATE VIEW` statement:

```

-- syntax
CREATE VIEW view_name AS
query
WITH CHECK OPTION;
```

When you create a view `WITH CHECK OPTION`, PostgreSQL will ensure that you can only modify data of the view that satisfies the condition in the view’s defining query (query).

`Scope of check:`

In PostgreSQL, you can specify a scope of check.

- `LOCAL:`

  The LOCAL scope restricts the check option enforcement to the current view only. It does not enforce the check to the views that the current view is based on.

  ```
  CREATE VIEW view_name AS
  query
  WITH LOCAL CHECK OPTION;
  ```

- `CASCADE:`

  The `CASCADED` scope extends the check option enforcement to all underlying views of the current view.

  ```
  CREATE VIEW view_name AS
  query
  WITH CASCADED CHECK OPTION;
  ```

To change the scope of check for an existing view, you can use the `ALTER VIEW` statement.

`Example:`

```
-- setting up a simple table and insert few data
CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    department_id INT,
    employee_type VARCHAR(20)
       CHECK (employee_type IN ('FTE', 'Contractor'))
);

INSERT INTO employees (first_name, last_name, department_id, employee_type)
VALUES
    ('John', 'Doe', 1, 'FTE'),
    ('Jane', 'Smith', 2, 'FTE'),
    ('Bob', 'Johnson', 1, 'Contractor'),
    ('Alice', 'Williams', 3, 'FTE'),
    ('Charlie', 'Brown', 2, 'Contractor'),
    ('Eva', 'Jones', 1, 'FTE'),
    ('Frank', 'Miller', 3, 'FTE'),
    ('Grace', 'Davis', 2, 'Contractor'),
    ('Henry', 'Clark', 1, 'FTE'),
    ('Ivy', 'Moore', 3, 'Contractor');

-- create view
```

```sql
CREATE OR REPLACE VIEW fte AS
SELECT
  id,
  first_name,
  last_name,
  department_id,
  employee_type
FROM
  employees
WHERE
  employee_type = 'FTE';

-- insert a new row into the employees table via the fte view:

INSERT INTO fte(first_name, last_name, department_id, employee_type)
VALUES ('John', 'Smith', 1, 'Contractor');
```

It succeeds,

The `issue` is that we can insert an `employee` with the type of `Contractor` into the employee table via the view that exposes the employee to the type of `FTE`.

To ensure that we can insert only employees with the type `FTE` into the `employees` table via the `fte` view, you can use the `WITH CHECK OPTION`:

```sql

-- insert this new data
INSERT INTO fte(first_name, last_name, department_id, employee_type)
VALUES ('John', 'Smith', 1, 'Contractor');

-- It will throw,
-- ERROR:  new row violates check option for view "fte"
```

The reason is that the `employee_type` Contractor does not satisfy the condition defined in the defining query of the view:

`employee_type = 'FTE';`

But if you modify the row with the employee type `FTE`, it'll be fine.

```sql

-- update
UPDATE fte
SET last_name = 'Doe'
WHERE id = 2;
```

It works as expected.

`Using WITH LOCAL CHECK OPTOION:`

Recreate the `fte` view without using the `WITH CHECK OPTION`

```
-- recreate view
CREATE OR REPLACE VIEW fte AS
SELECT id,first_name,last_name,department_id,employee_type
FROM employees
WHERE employee_type='FTE';

-- create new [fte_1] view based on [fte] view returns the employees of department
1, with the WITH LOCAL CHECK OPTION.
CREATE OR REPLACE VIEW fte_1 AS
SELECT id,first_name,last_name,department_id,employee_type
FROM fte
WHERE department_id=1
WITH LOCAL CHECK OPTION;

-- now retrieve the data
SELECT * FROM fte_1;

-- Since we use the WITH LOCAL CHECK OPTION, PostgreSQL checks only the fte_1

-- now insert data using fte_1
INSERT INTO fte_1(first_name, last_name, department_id, employee_type)
VALUES ('Miller', 'Jackson', 1, 'Contractor');

-- It succeeded. The reason is that the INSERT statement inserts a row with
department 1 that satisfies the condition in the fte_1 view
```

`Using WITH CASCADE CHECK OPTOION:`

Recreate the `fte_1` view with the `WITH CASCADE CHECK OPTION`

```
-- recreate fte_1
CREATE OR REPLACE VIEW fte_1 AS
SELECT id,first_name,last_name,department_id,employee_type
FROM fte
WHERE department_id=1
WITH CASCADE CHECK OPTION;

-- now retrieve the data
SELECT * FROM fte_1;

-- now insert data using fte_1
INSERT INTO fte_1(first_name, last_name, department_id, employee_type)
VALUES ('Miller', 'Jackson', 1, 'Contractor');

-- Error:
-- new row violates check option for view "fte".
-- The WITH CASCADED CHECK OPTION instructs PostgreSQL to check the constraint on
the fte_1 view and also its base view which is the fte view.
```

Alter view

The `ALTER VIEW` statement allow you to change various properties of view.

If you want to change the view's defining query, use the `CREATE OR REPLACE VIEW` statement.

```
-- syntax
ALTER VIEW view_name
    [ RENAME TO new_view_name ]
    [ OWNER TO new_owner ]
    [ COLUMN old_column_name TO new_column_name ]
    [ AS new_query ]
    [ WITH [ LOCAL | CASCADED ] CHECK OPTION ]
    [ [ NOT ] ENCRYPTED ]
    [ [ NOT ] DEFINED ]
```

`Key options:`

- `RENAME TO:` Changes the name of the view.
- `OWNER TO:` Changes the owner of the view.
- `COLUMN old_column_name TO new_column_name:` Renames a column within the view.
- `AS new_query:` Modifies the underlying query that defines the view.
- `WITH [ LOCAL | CASCADED ] CHECK OPTION:` Ensures that data inserted or updated through the view satisfies the WHERE clause of the view's definition.
- `[ NOT ] ENCRYPTED:` Enables or disables encryption for the view.
- `[ NOT ] DEFINED:` Marks the view as defined or undefined.

  ```
  -- full example
  ALTER VIEW CustomerOrders
      RENAME TO CustomerOrders_New
      OWNER TO another_user
      COLUMN OrderID TO OrderNumber
      AS
          SELECT CustomerID, OrderID AS OrderNumber
          FROM Orders
          WHERE OrderDate > '2023-01-01'
      WITH CASCADED CHECK OPTION
      ENCRYPTED;
  ```

`Renaming a view:`

```
ALTER VIEW IF EXISTS view_name
RENAME TO new_view_name;
```

`Change the view option:`

```
ALTER VIEW IF EXISTS view_name
SET (view_option_name [=view_option_value] [,...]);
```

The view_option_name can be:

- `check_option:` change the check option. the valid value is `LOCAL` OR `CASCADE`.
- `security_barrier:` change the security-barrier property of a view. The valid value is `true` or `false`.
- `security_invoker:` change the security invoker of a view. the valid value is `true` or `false`.

  ```
  ALTER VIEW actor_film
  SET (check_option = local)

  -- psql: to view the change you can use
  \d+ actor_film
  ```

`Changing the view column:`

```
-- syntax
ALTER VIEW view_name
RENAME [COLUMN] column_name TO new_column_name;

-- example
-- the following statement changes the title column of the film_rating view to
film_title
ALTER VIEW film_rating
RENAME title TO film_title;
```

`Setting the new schema:`

```
-- syntax
ALTER VIEW [ IF EXISTS ] view_name
SET SCHEMA new_schema;

-- example
ALTER VIEW film_rating
SET SCHEMA web;
```

Materialized view

In PostgreSQL,

Views are **virtual** table that represent data of underlying tables.

PostgreSQL extends the view concept to the next level which allows view to store data physically. These views are called **materialized view**.

Materialize view cache the result of an expensive query and allow you to refresh data periodically.

The materialized views can be useful in many cases that require fast data access. Therefore, you often find them in data warehouses and business intelligence applications.

`Create materialized view:`

```
-- syntax
CREATE MATERIALIZED VIEW IF NOT EXISTS view_name
AS
query
WITH [NO] DATA;
```

- `WITH DATA:` if you want to load data into the materialized view at the creation time.

  ```
  -- WITH DATA
  CREATE MATERIALIZED VIEW staff_store_address AS
  select staff_id,first_name,last_name, address,store.store_id from store
  JOIN staff ON store.manager_staff_id=staff.staff_id
  JOIN address ON address.address_id=staff.address_id
  WITH DATA;

  -- query data
  SELECT * FROM staff_store_address;
  ```

- `WITH NO DATA:` the view is flagged as unreadable. It means that you cannot query data from the view until you load data into it. In this case use `REFEERESH MATERIALIZED VIEW` statement to load the data.

  ```
  -- WITH NO DATA
  CREATE MATERIALIZED VIEW staff_store_address AS
  select staff_id,first_name,last_name, address,store.store_id from store
  JOIN staff ON store.manager_staff_id=staff.staff_id
  JOIN address ON address.address_id=staff.address_id
  WITH NO DATA;

  -- query data
  SELECT * FROM staff_store_address;
  ```

> It will not display any data. you need to refresh to load data.

`Refresh data for materialized view:`

To load data into materialized view, you use the `REFRESH MATERIALIZED VIEW` statement.

```
REFRESH MATERIALIZED VIEW view_name;
```

When you refresh data for a materialized view, PostgreSQL locks the underlying tables. Consequently, you will not be able to retrieve data from underlying tables while data is loading into the view.

To avoid this, you can use the `CONCURRENTLY` option.

```
REFRESH MATERIALIZED VIEW CONCURRENTLY view_name;
```

With the `CONCURRENTLY` option, PostgreSQL creates a temporary updated version of the materialized view, compares two versions, and performs `INSERT` and `UPDATE` only the differences.

However, to refresh it with `CONCURRENTLY` option, you need to create a `UNIQUE` index for the view first.

```
-- create unique index on staff-store_address view
CREATE UNIQUE INDEX staff_store_address_idx
ON staff_store_address (address);
```

`Drop materialized view:`

```
-- syntax
DROP MATERIALIZED VIEW view_name;
```