PostgreSQL Views

A **view** is a named query that provides another way to present data in the database tables.

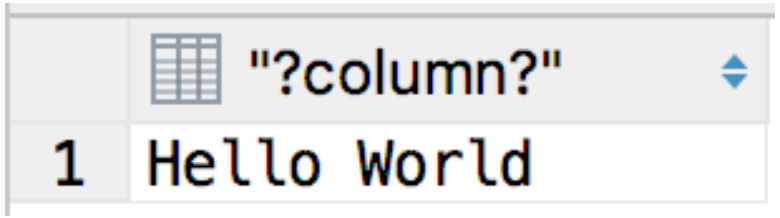
A **view** is a stored query. A view can be accessed as a virtual table in PostgreSQL. In other words, a PostgreSQL view is a logical table that represents data of one or more underlying tables through a SELECT statement.

Views

```
CREATE VIEW vista AS SELECT 'Hello World';
```

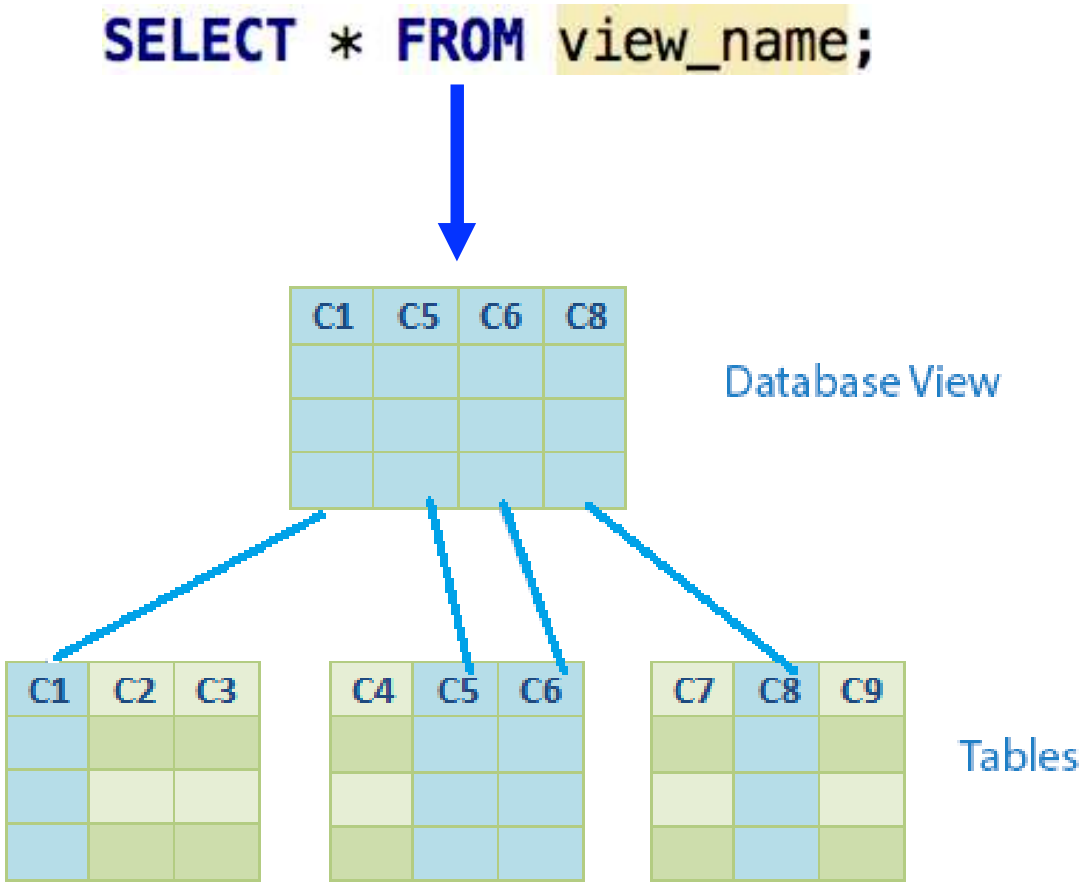
```
SELECT * FROM vista;
```

psql -U username -d myDataBase -a -f myInsertFile



	"?column?"
1	Hello World

Views


SELECT * FROM view_name;
C1
C5
C6
C8
Database View
C1
C2
C3
C4
C5
C6
C7
C8
C9
Tables

Parameter

- *CREATE OR REPLACE VIEW* is similar, but if a view of the same name already exists, it is replaced.

- *TEMPORARY or TEMP* - If specified, the view is created as a temporary view. Temporary views are automatically dropped at the end of the current session.

- *query* - A SELECT or VALUES command which will provide the columns and rows of the view.

Parameter

- *WITH [CASCADED | LOCAL] CHECK OPTION* - This option controls the behavior of automatically updatable views. When this option is specified, *INSERT* and *UPDATE* commands on the view will be checked to ensure that new rows satisfy the view-defining condition.

Parameters of Check option

- *LOCAL* - New rows are only checked against the conditions defined directly in the view itself. Any conditions defined on underlying base views are not checked (unless they also specify the CHECK OPTION).

Parameters of Check option

- *CASCADED - New rows are checked against the conditions of the view and all underlying base views. If the CHECK OPTION is specified, and neither LOCAL nor CASCADED is specified, then CASCADED is assumed.*

Advantages of Using Views

- **Simplicity**: Abstracts complex queries and makes them easier to use.
- **Security**: Restricts access to specific columns or rows in a table.
- **Consistency**: Ensures data presentation consistency across applications.

Types of Views

- **Simple Views:** These are based on a single table and do not contain any functions, groupings, or aggregations.
- **Complex Views:** These can include joins, subqueries, and aggregations, allowing for more complex data retrieval.
- **Materialized Views:** Unlike standard views, materialized views store the result set physically in the database, which can improve performance for large datasets or complex queries. They need to be refreshed manually or on a schedule to update their data.

Creating PostgreSQL Views

To create a view, we use **CREATE VIEW** statement. The simplest syntax of the **CREATE VIEW** statement is as follows:

CREATE VIEW view_name AS query;

PostgreSQL CREATE VIEW example

```
CREATE VIEW passengers_info AS
SELECT
    passenger_id,
    gender,
    date_of_birth,
    CONCAT(first_name, ' ', last_name) AS full_name
FROM
    passengers;
```

```
SELECT * FROM passengers_info;
```

	passenger_id integer	gender character varying (50)	date_of_birth date	full_name text
1	1	Female	2000-01-03	Hilde Irnis
2	2	Male	1974-06-09	Arvy Sparsholt
3	3	Male	1982-06-07	Reinald Pococke
4	4	Female	1986-10-17	Con Borrel
5	5	Male	1996-04-22	Wayne Bangs

PostgreSQL CREATE VIEW example

```
CREATE VIEW flight_schedule AS
SELECT
    f.flight_id,
    f.scheduled_departure,
    f.scheduled_arrival,
    a.airline_name,
    dep.airport_name AS departure_airport,
    arr.airport_name AS arriving_airport,
    f.departing_gate,
    f.arriving_gate
FROM
    Flights f
JOIN
    Airline a ON f.airline_id = a.airline_id
JOIN
    Airport dep ON f.departure_airport_id = dep.airport_id
JOIN
    Airport arr ON f.arrival_airport_id = arr.airport_id;
```

Filtering Data from a View

```
SELECT *
FROM flight_schedule
WHERE departure_airport = 'Longana Airport';
```

flight_id integer	scheduled_departure date	scheduled_arrival date	airline_name character varying (50)	departure_airport character varying (50)	arriving_airport character varying (50)	departing_gate character varying (50)	arriving_ charact
10	2023-11-25	2023-07-26	NHT	Longana Airport	Zephyrhills Municipal Airport	593	555
80	2023-11-02	2023-08-12	YBQ	Longana Airport	Lime Acres Finsch Mine Airport	137	[null]
100	2023-08-27	2024-02-27	SOZ	Longana Airport	Bermuda Dunes Airport	95	915
102	2024-02-08	2023-03-25	MXW	Longana Airport	Fort Worth Alliance Airport	432	103
104	2023-08-01	2023-11-29	KGF	Longana Airport	Zephyrhills Municipal Airport	107	538

Changing PostgreSQL Views

To change the defining query of a view, you use the CREATE VIEW statement with OR REPLACE addition as follows:

CREATE OR REPLACE view_name

AS

query

ALTER VIEW

To change the definition of a view, you use the **ALTER VIEW** statement. For example, you can change the name of the view from customer_master to customer_info by using the following statement:

```
ALTER VIEW customer_master RENAME TO customer_info;
```

PostgreSQL Drop View

The **DROP VIEW** statement removes a view from the database. The following illustrates the syntax of the DROP VIEW statement:

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

To remove multiple views using a single statement, you specify a comma-separated list of view names after the DROP VIEW keywords like this:

```
DROP VIEW [IF EXISTS] view_name1, view_name2, ...;
```

Materialized views

- PostgreSQL extends the view concept to a next level that allows views to store data physically, and we call those views are materialized views.

- A materialized view caches the result of a complex expensive query and then allow you to refresh this result periodically.

- The materialized views are useful in many cases that require fast data access therefore they are often used in data warehouses or business intelligent applications.

Materialized views

- To create a materialized view, you use the CREATE MATERIALIZED VIEW statement as follows:

```
CREATE MATERIALIZED VIEW view_name
AS
SELECT column_list FROM table_name;
WITH [NO] DATA;
```

Materialized views

- If you want to load data into the materialized view at the creation time, you put `WITH DATA` option, otherwise you put `WITH NO DATA`.

- In case you use `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.

PostgreSQL Materialized Views

The PostgreSQL **materialized views** that allow you to store the result of a query physically and update the data periodically.

To create a materialized view, you use the CREATE MATERIALIZED VIEW statement as follows:

```
CREATE MATERIALIZED VIEW view_name
AS
query
WITH [NO] DATA;
```

Example

```sql
CREATE MATERIALIZED VIEW active_bookings AS
SELECT
    b.booking_id,
    p.first_name || ' ' || p.last_name AS passenger_name,
    b.status
FROM
    Booking b
JOIN
    Passengers p ON b.passenger_id = p.passenger_id
WHERE
    b.status = 'Active';
```

```sql
SELECT * FROM active_bookings;
REFRESH MATERIALIZED VIEW active_bookings;
```

Refreshing data for materialized views

To load data into a materialized view, you use the **REFRESH MATERIALIZED VIEW** statement as shown below:

REFRESH MATERIALIZED VIEW view_name;

Removing materialized views

Removing a materialized view is pretty straightforward as we have done for tables or views. This is done using the following statement:

```
DROP MATERIALIZED VIEW view_name;
```

Roles

Roles

- PostgreSQL manages database access permissions using the concept of *roles*.
- A role can be thought of as either a database user, or a group of database users, depending on how the role is set up.
- Roles can own database objects (for example, tables and functions) and can assign privileges on those objects to other roles to control who has access to which objects.

Roles

- Database roles are conceptually completely separate from operating system users.
- In practice it might be convenient to maintain a correspondence, but this is not required.
- Database roles are global across a database cluster installation (and not per individual database). To create a role use the CREATE ROLE SQL command:

Roles

- To determine the set of existing roles, examine the pg_roles system catalog, for example

```
SELECT rolname FROM pg_roles;
```

Roles

- To create a role use the CREATE ROLE SQL command:

```
CREATE ROLE name;
```

```
DROP ROLE name;
```

Role Attributes

- A database role can have a number of attributes that define its privileges and interact with the client authentication system.

Login privileges

- Only roles that have the `LOGIN` attribute can be used as the initial role name for a database connection.
- A role with the `LOGIN` attribute can be considered the same as a “database user”. To create a role with login privilege, use either:

```
CREATE ROLE name LOGIN;

CREATE USER name;
```

Superuser status

- A database superuser bypasses all permission checks, except the right to log in.
- This is a dangerous privilege and should not be used carelessly; it is best to do most of your work as a role that is not a superuser.
- To create a new database superuser, use:
- **CREATE ROLE*name* SUPERUSER**.
- You must do this as a role that is already a superuser.

Database creation

- A role must be explicitly given permission to create databases.
- To create such a role, use:
- `CREATE ROLE name CREATEDB`.

Initiating replication

- A role must explicitly be given permission to initiate streaming replication.
- A role used for streaming replication must have LOGIN permission as well.
- To create such a role, use:
- `CREATE ROLE name REPLICATION LOGIN`.

Role creation

- A role must be explicitly given permission to create more roles.
- To create such a role, use:
- **`CREATE ROLE name CREATEROLE`**.
- A role with CREATEROLE privilege can alter and drop other roles, too, as well as grant or revoke membership in them.
- However, to create, alter, drop, or change membership of a superuser role, superuser status is required;

Password

- A password is only significant if the client authentication method requires the user to supply a password when connecting to the database.
- The password and md5 authentication methods make use of passwords.
- Database passwords are separate from operating system passwords.
- Specify a password upon role creation with:
- `CREATE ROLE name PASSWORD 'string'.`

Modify role

- A role's attributes can be modified after creation with **`ALTER ROLE`**.

```
ALTER ROLE role_specification [ WITH ] option [ ... ]
```

Modify role

- where *option* can be:

```
  SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| CONNECTION LIMIT connlimit
| [ ENCRYPTED ] PASSWORD 'password'
```

Modify role

- where *option* can be:

```
  SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| CONNECTION LIMIT connlimit
| [ ENCRYPTED ] PASSWORD 'password'
```

Examples

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

```
ALTER ROLE davide WITH PASSWORD NULL;
```

```
ALTER ROLE miriam CREATEROLE CREATEDB;
```

Role Membership

- It is frequently convenient to group users together to ease management of privileges: that way, privileges can be granted to, or revoked from, a group as a whole.

- In PostgreSQL this is done by creating a role that represents the group, and then granting *membership* in the group role to individual user roles.

Role Membership

- Once the group role exists, you can add and remove members using the GRANT and REVOKE commands:

- `GRANT group_role TO role1, ... ;`
- `REVOKE group_role FROM role1, ... ;`

Role Membership

- You can grant membership to other group roles, too (since there isn't really any distinction between group roles and non-group roles).

- The database will not let you set up circular membership loops.

Drop role

- To destroy a group role, use DROP ROLE:
- `DROP ROLE name;`

Ownership

- Ownership of objects can be transferred one at a time using ALTER commands, for example:
- `ALTER TABLE bobs_table OWNER TO alice;`

Ownership

- Alternatively, the REASSIGN OWNED command can be used to reassign ownership of all objects owned by the role-to-be- dropped to a single other role.
  - `REASSIGN OWNED BY doomed_role TO successor_role;`
  - `DROP OWNED BY doomed_role;`
  - `DROP ROLE doomed_role;`

Example

Adding a new table to our airport database

```
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    role VARCHAR(50),
    department VARCHAR(50)
);
```

```
CREATE ROLE booking_department;-- role for booking dep
CREATE ROLE dispatcher_department;-- role for dispatcher
CREATE ROLE admin_department;-- role for admin
```

```
GRANT SELECT, INSERT, UPDATE ON TABLE passengers TO booking_department;--giving read and write permissions
GRANT SELECT ON TABLE flights TO dispatcher_department;-- only to read permission
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO admin_department;--full access
```

Creating users and assigning them roles

```
CREATE USER booking_user WITH PASSWORD 'password123';-- user for the reservations
GRANT booking_department TO booking_user;

CREATE USER dispatcher_user WITH PASSWORD 'password456';--dispatcher user
GRANT dispatcher_department TO dispatcher_user;

CREATE USER admin_user WITH PASSWORD 'password789';--admin user
GRANT admin_department TO admin_user;
```