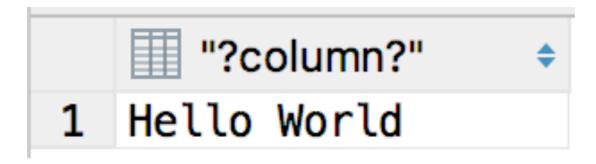
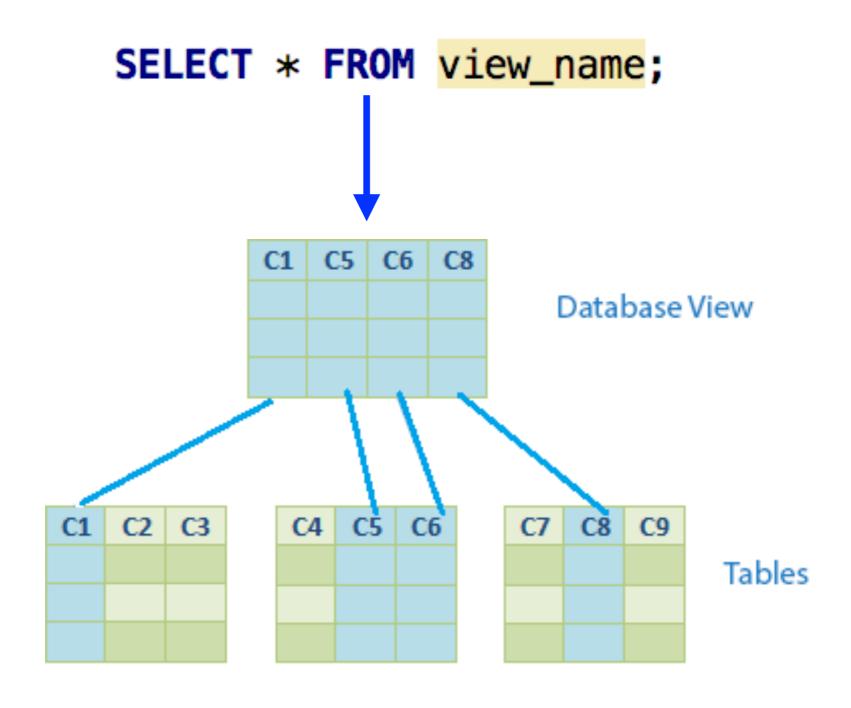
- A view is named query that provides another way to present data in the database tables.
- A view is defined based on one or more tables, which are known as base tables.
- When you create a view, you basically create a query and assign it a name
- Therefore a view is useful for wrapping a commonly used complex query.

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





```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] VIEW name [ ( column_name [, ...] ) ]

AS query
[ WITH_ [ CASCADED | LOCAL ] CHECK OPTION ]
```

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.

Films table

id	title	kind	classification
1	Film1	Drama	PG
2	Film2	Comedy	U
3	Film3	Comedy	PG
4	Film4	Fantasy	D

Examples

Create a view consisting of all comedy films:

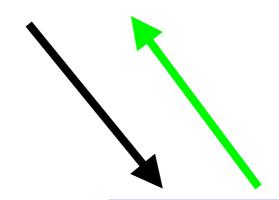
```
CREATE VIEW comedies AS

SELECT *

FROM films
WHERE kind = 'Comedy';
```

Example

SELECT * **FROM** comedies;



id	title	kind	classificat ion
1	Film1	Drama	PG
2	Film2	Comed	U
3	Film3	Comed	PG
4	Film4	Fantasy	D



```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';
```

Examples

```
CREATE VIEW universal_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'U'
    WITH LOCAL_CHECK OPTION;
```

```
INSERT INTO universal_comedies VALUES (DEFAULT, 'Film5', 6, 'Comedy', 'U');
INSERT INTO universal_comedies VALUES (DEFAULT, 'Film5', 6, 'Comedy', 'D');
```

Examples

```
CREATE VIEW pg_comedies AS

SELECT *

FROM comedies

WHERE classification = 'PG'
WITH CASCADED CHECK OPTION;
```

```
INSERT INTO pg_comedies VALUES (DEFAULT, 'Film5', 6, 'Comedy', 'PG');
INSERT INTO pg_comedies VALUES (DEFAULT, 'Film5', 6, 'Drama', 'PG');
INSERT INTO pg_comedies VALUES (DEFAULT, 'Film5', 6, 'Comedy', 'U');
```

- 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.

 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;
```

- 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.

Example

```
CREATE MATERIALIZED VIEW rental_by_category
AS
 SELECT c.name AS category,
    sum(p.amount) AS total_sales
   FROM ((((payment p
     JOIN rental r ON ((p.rental_id = r.rental_id)))
     JOIN inventory i ON ((r.inventory_id = i.inventory_id)))
     JOIN film f ON ((i.film_id = f.film_id)))
     JOIN film_category fc ON ((f.film_id = fc.film_id)))
     JOIN category c ON ((fc.category_id = c.category_id)))
  GROUP BY c. name
  ORDER BY sum(p.amount) DESC
WITH NO DATA;
```

 Because we used the WITH NO DATA option, we cannot query data from the view. If we try to do so, we will get an error message as follows:

```
SELECT * FROM rental_by_category;
```

```
[Err] ERROR: materialized view "rental_by_category" has not been populated HINT: Use the REFRESH MATERIALIZED VIEW command.
```

 PostgreSQL is very nice to give us a hint to ask for loading data into the view. Let's do it by executing the following statement:

REFRESH MATERIALIZED VIEW rental_by_category;

 Now, if we query data again, we will get the result as expected.

category	total_sales	
▶ Sports	4892.19	
Sci-Fi	4336.01	
Animation	4245.31	
Drama	4118.46	
Comedy	4002.48	
New	3966.38	
Action	3951.84	
Foreign	3934.47	
Games	3922.18	
Family	3830.15	
-Docupantant	3740.65	

 From now on, we can refresh the data in the rental_by_category view using the REFRESH MATERIALIZED VIEW statement. However, to refresh it with CONCURRENTLY option, we need to create a UNIQUE index for the view first.

```
CREATE UNIQUE INDEX rental_category
ON rental_by_category (category);
```

Let's refresh data concurrently for the rental_by_category view.

REFRESH MATERIALIZED VIEW CONCURRENTLY rental_by_category;

Modifying 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

Modifying views

To change the definition of a view, you use the ALTER VIEW statement.

```
ALTER VIEW customer_master

RENAME TO customer_info;
```

Remove views

 To remove an existing view in PostgreSQL, you use DROP VIEW statement as follows:

```
DROP VIEW [ IF EXISTS ] view_name;
```

Updatable views

A PostgreSQL view is updatable when it meets the following conditions:

- The defining query of the view must has exactly one entry in the FROM clause, which can be a table or another updatable view.
- The defining query must not contain one of the following clauses at top level: GROUP BY, HAVING, LIMIT, OFFSET, DISTINCT, WITH, UNION, INTERSECT, and EXCEPT.
- The selection list must not contain any window function or set-returning function or any aggregate function such as SUM, COUNT, AVG, MIN, MAX, etc.

Updatable views

- An updatable view may contain both updatable and nonupdatable columns.
- If you try to insert or update a non-updatable column, PostgreSQL will raise an error.
- When you execute an update operation such as INSERT, UPDATE or DELETE, PosgreSQL will convert this statement into the corresponding statement of the underlying table.

Updatable views

```
CREATE VIEW test_update AS SELECT code, value FROM boxes;
INSERT INTO test_update (code, value, contents) VALUES (DEFAULT, 1, 'Rocks');
INSERT INTO test_update (code, value) VALUES (DEFAULT, 1);
```

[42703] ERROR: column "contents" of relation "test_update" does not exist

Позиция: 26

- 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.

- 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 <u>CREATE ROLE</u> SQL command:

• To create a role use the **CREATE ROLE** SQL command:

CREATE ROLE name;

DROP ROLE name;

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

SELECT rolname FROM pg_roles;

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.

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;

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.

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-bedropped to a single other role.

- REASSIGN OWNED BY doomed_role TO successor_role;
- DROP OWNED BY doomed_role;
- DROP ROLE doomed_role;

Questions