#### ****Introduction to Views****

1. **Definition**: A SQL **View** is a database object that represents the result of an SQL query.
   * **Characteristics**:
     + A view does not store data but acts as a **virtual table**.
     + Each time a view is queried, the underlying SQL query it represents is executed.
2. **Example Tables**: The tutorial uses the following tables:
   * **Customer Table (**tb\_customer\_data**)**: Includes customer\_id, name, phone, email, and address.
   * **Product Table (**tb\_product\_info**)**: Contains product\_id, product\_name, brand, and price.
   * **Order Table (**tb\_order\_details**)**: Records order\_id, product, quantity, customer\_id, discount, and order\_date.

#### ****Use Case: Generating an Order Summary****

* **Scenario**: A supervisor requests an **order summary report** for clients or vendors.
  + Required Columns: order\_id, order\_date, product\_name, customer\_name, and cost.
  + **Solution**: Write an SQL query that joins the three tables and calculates the cost.

sql

Copy code

SELECT

orders.order\_id, orders.order\_date,

products.product\_name, customers.name AS customer\_name,

(products.price \* orders.quantity - orders.discount) AS cost

FROM

tb\_order\_details orders

JOIN

tb\_product\_info products ON orders.product\_id = products.product\_id

JOIN

tb\_customer\_data customers ON orders.customer\_id = customers.customer\_id;

* The query generates the desired output, which can be saved as a view for dynamic access by external parties.

#### ****Creating a View****

1. **Create Command**:
   * Syntax:

sql

Copy code

CREATE VIEW order\_summary AS

[SQL Query];

* + Example:

sql

Copy code

CREATE VIEW order\_summary AS

SELECT

orders.order\_id, orders.order\_date,

products.product\_name, customers.name AS customer\_name,

(products.price \* orders.quantity - orders.discount) AS cost

FROM

tb\_order\_details orders

JOIN

tb\_product\_info products ON orders.product\_id = products.product\_id

JOIN

tb\_customer\_data customers ON orders.customer\_id = customers.customer\_id;

1. **Querying the View**:
   * Access data in a view as you would a table:

sql

Copy code

SELECT \* FROM order\_summary;

* + Internally, SQL executes the query defined in the view.

#### ****Advantages of Views****

1. **Security**:
   * Use views to **restrict access** to sensitive data in base tables.
   * Example: Create a user (James) and grant access only to the order\_summary view:

sql

Copy code

GRANT SELECT ON order\_summary TO James;

* + This way, the user cannot access or view table structures and sensitive fields, such as email and phone in the tb\_customer\_data table.

1. **Simplify Complex Queries**:
   * For complex queries spanning multiple tables, create views to simplify access.
   * Example:
     + A 100-line SQL query can be stored in a view for ease of reuse and understanding.

#### ****Creating or Replacing a View****

* **Command**:

sql

Copy code

CREATE OR REPLACE VIEW view\_name AS

[SQL Query];

* **Behavior**:
  + If the view exists, SQL **replaces** it; if not, it creates a new one.
  + **Rules**:
    - Cannot change column names, data types, or order of existing columns.

#### ****Examples: Rules for CREATE OR REPLACE****

1. **Column Name Change**: Fails if column names are modified:
   * Incorrect:

sql

Copy code

CREATE OR REPLACE VIEW order\_summary AS

SELECT orders.order\_date AS order\_date;

* + - Error: Cannot change name of view column.

1. **Adding Columns**:
   * Allowed only at the **end** of the column list:

sql

Copy code

CREATE OR REPLACE VIEW order\_summary AS

SELECT

orders.order\_id, orders.order\_date,

products.product\_name, customers.name AS customer\_name,

(products.price \* orders.quantity - orders.discount) AS cost,

customers.customer\_id;

#### ****Understanding View Structure****

* **Stored Structure**: Views store the structure (columns, data types, etc.) of the query when created.
* **Example**:
  + If a new column is added to a base table, it won’t appear in the view until the view is refreshed:

sql

Copy code

CREATE OR REPLACE VIEW view\_name AS [SQL Query];

This first part highlights the introduction to SQL Views, their creation, advantages, and specific rules for creating or replacing views. It also emphasizes practical examples like creating user-restricted views, handling security, and simplifying complex queries. Let me know if this summary aligns with your expectations before proceeding to part two!

### ****Summary of the Transcript: Part 2****

The second part of the transcript continues to delve into the creation, management, and practical uses of SQL views, focusing on modifying views, handling structural changes, and creating updatable views with rules. Here's the detailed summary:

### ****1. Modifying Views****

Views can be modified using specific SQL commands depending on the changes required.

#### ****a.**** CREATE OR REPLACE VIEW

* This command recreates a view if it already exists.
* **Key Rules**:
  + You **cannot** change the column order, names, or data types in an existing view using CREATE OR REPLACE VIEW.
  + However, you **can**:
    - Add new columns to the end of the column list.
    - Modify the SELECT clause (e.g., add/remove WHERE or ORDER BY clauses).
* **Example**: Adding a new column to a view:

sql

Copy code

CREATE OR REPLACE VIEW order\_summary AS

SELECT

orders.order\_id, orders.order\_date,

products.product\_name, customers.name AS customer\_name,

(products.price \* orders.quantity - orders.discount) AS cost,

customers.customer\_id -- New column added at the end

FROM

tb\_order\_details orders

JOIN

tb\_product\_info products ON orders.product\_id = products.product\_id

JOIN

tb\_customer\_data customers ON orders.customer\_id = customers.customer\_id;

#### ****b.**** ALTER VIEW

* Use ALTER VIEW to rename columns or the view itself.
* **Example 1**: Renaming a column:

sql

Copy code

ALTER VIEW order\_summary RENAME COLUMN order\_date TO order\_placed\_date;

* **Example 2**: Renaming the view:

sql

Copy code

ALTER VIEW order\_summary RENAME TO order\_summary\_v2;

#### ****c.**** DROP VIEW

* Removes a view from the database:

sql

Copy code

DROP VIEW order\_summary\_v2;

### ****2. Handling Structural Changes in Underlying Tables****

Views do not automatically adapt to changes in the underlying table structure.

#### ****a. Adding a New Column to the Table****

* If a column is added to a table, it will not automatically appear in the view unless the view is refreshed using CREATE OR REPLACE VIEW.
* **Example**:
  + Adding a column to the tb\_product\_info table:

sql

Copy code

ALTER TABLE tb\_product\_info ADD COLUMN prod\_config VARCHAR(100);

* + Refreshing the view to include the new column:

sql

Copy code

CREATE OR REPLACE VIEW expensive\_products AS

SELECT \* FROM tb\_product\_info WHERE price > 1000;

#### ****b. Data Changes in Tables****

* Views always show the **latest data** from the underlying tables because they execute the original query each time they are called.
* However, structural changes (e.g., new columns) require explicit updates to the view.

### ****3. Updatable Views****

Updatable views allow changes (e.g., INSERT, UPDATE, or DELETE) to propagate to the underlying tables. However, not all views can be updated.

#### ****a. Rules for Updatable Views****

1. **Single Table Only**:
   * The view must be based on a single table.
   * Example: A view like this is updatable:

sql

Copy code

CREATE VIEW expensive\_products AS

SELECT \* FROM tb\_product\_info WHERE price > 1000;

1. **No** DISTINCT **Clause**:
   * Views containing DISTINCT cannot be updated.
   * Example:

sql

Copy code

CREATE VIEW unique\_brands AS

SELECT DISTINCT brand FROM tb\_product\_info;

-- Not updatable

1. **No** GROUP BY **Clause**:
   * Views with GROUP BY cannot be updated.
   * Example:

sql

Copy code

CREATE VIEW order\_summary\_by\_date AS

SELECT order\_date, COUNT(\*) AS total\_orders

FROM tb\_order\_details

GROUP BY order\_date;

-- Not updatable

1. **No** WITH **Clause**:
   * Views created with common table expressions (WITH) cannot be updated.
2. **No Window Functions**:
   * Views containing window functions (e.g., ROW\_NUMBER, RANK) are not updatable.

#### ****b. Updating Data in Updatable Views****

* If the view is based on a single table and satisfies the rules above, you can update data through the view.
* **Example**:

sql

Copy code

UPDATE expensive\_products

SET product\_name = 'Updated Product Name'

WHERE product\_id = 1;

#### ****c. Non-Updatable Views****

* For views that are not naturally updatable, you can use **INSTEAD OF triggers** to handle updates.

### ****4. The**** WITH CHECK OPTION ****Clause****

The WITH CHECK OPTION clause ensures that any changes (e.g., INSERT or UPDATE) made through a view must satisfy the view's WHERE clause.

#### ****a. Example Without**** WITH CHECK OPTION

* Creating a view for products of a specific brand:

sql

Copy code

CREATE VIEW apple\_products AS

SELECT \* FROM tb\_product\_info WHERE brand = 'Apple';

* A supplier accidentally inserts data for a non-Apple product:

sql

Copy code

INSERT INTO apple\_products (product\_id, product\_name, brand, price)

VALUES (11, 'Non-Apple Product', 'Samsung', 500);

* The product is added to the underlying table but does not appear in the view, which violates the purpose of the view.

#### ****b. Example With**** WITH CHECK OPTION

* Adding the WITH CHECK OPTION ensures that only rows meeting the view’s condition can be inserted or updated:

sql

Copy code

CREATE OR REPLACE VIEW apple\_products AS

SELECT \* FROM tb\_product\_info WHERE brand = 'Apple'

WITH CHECK OPTION;

* Now, if someone tries to insert a non-Apple product, they get an error:

sql

Copy code

INSERT INTO apple\_products (product\_id, product\_name, brand, price)

VALUES (12, 'Non-Apple Product', 'Samsung', 500);

-- Error: new row violates check option

### ****5. Summary of the Second Part****

* **Modifying Views**: Use CREATE OR REPLACE, ALTER VIEW, and DROP VIEW for changes.
* **Structural Changes in Tables**: Views require manual updates to reflect new columns.
* **Updatable Views**: Can only be updated if they follow specific rules (e.g., no DISTINCT, GROUP BY).
* WITH CHECK OPTION: Prevents invalid data from being added through views.

This concludes the explanation of the second part of the transcript.

before going to the second part give me the code for creating the example table and fill each table with 10 rows of values in postgres sql.

###### ChatGPT said:

Here is the SQL code to create the example tables and populate them with 10 rows of sample data in PostgreSQL:

sql

Copy code

-- Create Customer Table

CREATE TABLE tb\_customer\_data (

customer\_id SERIAL PRIMARY KEY,

name VARCHAR(100) NOT NULL,

phone VARCHAR(15) NOT NULL,

email VARCHAR(100) NOT NULL,

address VARCHAR(255) NOT NULL

);

-- Populate Customer Table

INSERT INTO tb\_customer\_data (name, phone, email, address) VALUES

('Alice Johnson', '1234567890', 'alice@example.com', '123 Main St'),

('Bob Smith', '9876543210', 'bob@example.com', '456 Elm St'),

('Charlie Davis', '5555555555', 'charlie@example.com', '789 Oak St'),

('Diana Brown', '1112223333', 'diana@example.com', '101 Pine St'),

('Edward Wilson', '4446668888', 'edward@example.com', '202 Cedar St'),

('Fiona Adams', '3337779999', 'fiona@example.com', '303 Maple St'),

('George Clark', '2223334444', 'george@example.com', '404 Birch St'),

('Hannah Moore', '8889990000', 'hannah@example.com', '505 Walnut St'),

('Ian Hall', '6667778888', 'ian@example.com', '606 Cherry St'),

('Julia King', '9990001111', 'julia@example.com', '707 Spruce St');

-- Create Product Table

CREATE TABLE tb\_product\_info (

product\_id SERIAL PRIMARY KEY,

product\_name VARCHAR(100) NOT NULL,

brand VARCHAR(50) NOT NULL,

price NUMERIC(10, 2) NOT NULL

);

-- Populate Product Table

INSERT INTO tb\_product\_info (product\_name, brand, price) VALUES

('Laptop A', 'BrandX', 1000.00),

('Smartphone B', 'BrandY', 800.00),

('Tablet C', 'BrandZ', 600.00),

('Monitor D', 'BrandX', 300.00),

('Headphones E', 'BrandY', 150.00),

('Keyboard F', 'BrandZ', 50.00),

('Mouse G', 'BrandX', 40.00),

('Charger H', 'BrandY', 20.00),

('Desk Lamp I', 'BrandZ', 70.00),

('Speaker J', 'BrandX', 120.00);

-- Create Order Table

CREATE TABLE tb\_order\_details (

order\_id SERIAL PRIMARY KEY,

product\_id INTEGER NOT NULL REFERENCES tb\_product\_info(product\_id),

quantity INTEGER NOT NULL,

customer\_id INTEGER NOT NULL REFERENCES tb\_customer\_data(customer\_id),

discount NUMERIC(10, 2) NOT NULL,

order\_date DATE NOT NULL

);

-- Populate Order Table

INSERT INTO tb\_order\_details (product\_id, quantity, customer\_id, discount, order\_date) VALUES

(1, 2, 1, 50.00, '2023-11-01'),

(2, 1, 2, 30.00, '2023-11-02'),

(3, 3, 3, 20.00, '2023-11-03'),

(4, 1, 4, 0.00, '2023-11-04'),

(5, 2, 5, 10.00, '2023-11-05'),

(6, 1, 6, 5.00, '2023-11-06'),

(7, 2, 7, 0.00, '2023-11-07'),

(8, 1, 8, 2.00, '2023-11-08'),

(9, 3, 9, 15.00, '2023-11-09'),

(10, 2, 10, 25.00, '2023-11-10');

### Instructions for Use:

1. Open a PostgreSQL client or query editor.
2. Copy and paste the above SQL script.
3. Run the script to create the tables and populate them with the specified data.

Let me know if you need any additional modifications or explanations!

##### You said:

I want to create these table in the postgres-practice database, when i just create table tb\_customer\_data how the sql knows to which database it must access.

###### ChatGPT said:

When you are working with multiple databases in PostgreSQL (like in your postgres-practice workspace shown in the screenshot), the SQL editor knows which database to use based on the **active connection**. Here's how it works:

### ****Steps to Direct SQL Commands to the Right Database****

1. **Active Connection**:
   * The database you are connected to is indicated at the top of the SQL editor in tools like **DBeaver**, as shown in your screenshot (postgres-practice is the active database).
   * Any SQL query you execute in that editor will apply to the active database.
2. **Explicit Connection**:
   * If you want to specify the database explicitly in your SQL scripts (useful in certain setups), you can use the \c command in **psql** or specify it explicitly in your **connection settings**:

sql

Copy code

\c postgres-practice

1. **Default Connection**:
   * If you open a new SQL editor in tools like DBeaver, it will prompt you to connect to a database. The SQL commands will apply to the selected database.
2. **Schema Context**:
   * By default, tables are created in the public schema unless you specify another schema. This schema belongs to the connected database.
3. **Fully Qualified Names**:
   * You can explicitly define the schema and database (though this is less common in PostgreSQL):

sql

Copy code

CREATE TABLE public.tb\_customer\_data (...);

1. **Switching Connections in Tools**:
   * In tools like DBeaver, you can switch the active database by selecting a different database in the **Database Navigator** panel or by right-clicking the database and choosing Set Active.

### ****How SQL Identifies the Database****

When you run a CREATE TABLE command like:

sql

Copy code

CREATE TABLE tb\_customer\_data (

customer\_id SERIAL PRIMARY KEY,

name VARCHAR(100) NOT NULL,

phone VARCHAR(15) NOT NULL,

email VARCHAR(100) NOT NULL,

address VARCHAR(255) NOT NULL

);

The tool or client uses the current **connection's active database**. Since you are connected to postgres-practice, this table will be created in that database.

If you want to confirm or explicitly control the connection:

* Use the SHOW current\_database; command.
* Or check the connection in your tool's interface (as shown in your screenshot, under postgres-practice).

Let me know if you need more details or examples!

##### You said:

SERIAL what is meant by serial primary key why we are using in the postgres, if it is wrong if i only use primary key instead of serial primary key.

###### ChatGPT said:

### What is a SERIAL Primary Key in PostgreSQL?

In PostgreSQL, the SERIAL keyword is used to create an **auto-incrementing integer column**. When combined with the PRIMARY KEY constraint, it ensures that each value in the column is unique and automatically generated. Here's a detailed explanation:

### ****How**** SERIAL ****Works****

1. **Auto-Incrementing Column**:
   * A SERIAL column automatically generates sequential values starting from 1 by default.
   * Each time a new row is inserted, PostgreSQL automatically assigns the next available integer value to the SERIAL column.
2. **Backed by a Sequence**:
   * Under the hood, PostgreSQL creates a **sequence** for the column. This sequence is used to generate unique values for each row.
   * For example, the following command:

sql

Copy code

CREATE TABLE example (

id SERIAL PRIMARY KEY

);

Internally does three things:

* + - Creates the column id as an integer.
    - Creates a sequence (example\_id\_seq) to auto-generate values for the column.
    - Sets the default value of the column to fetch the next value from the sequence:

sql

Copy code

nextval('example\_id\_seq')

1. **Why Use** SERIAL PRIMARY KEY**?**
   * Ensures automatic, unique, and sequential values for the primary key.
   * Reduces manual work; you don't need to explicitly handle unique ID generation.

### ****Using Only**** PRIMARY KEY ****Without**** SERIAL

You can define a column as a PRIMARY KEY without using SERIAL, but the behavior will differ:

1. **Without** SERIAL:
   * You must manually specify the value of the column when inserting rows.
   * For example:

sql

Copy code

CREATE TABLE example (

id INT PRIMARY KEY

);

In this case, PostgreSQL won't auto-generate values for the id column. You must insert the id explicitly:

sql

Copy code

INSERT INTO example (id) VALUES (1), (2), (3);

1. **Challenges Without** SERIAL:
   * It's your responsibility to ensure that the values are unique and sequential.
   * If multiple users or applications interact with the table, there’s a risk of **conflicting primary key values** unless you implement your own mechanism for generating unique values.

### ****When is**** SERIAL ****Preferred?****

* When you need a **unique and sequential primary key** with minimal effort.
* It's useful for most use cases, such as when creating an id column for customers, orders, or products.

### ****Alternative to**** SERIAL****:**** IDENTITY ****(Modern Approach)****

Starting with PostgreSQL 10, a better alternative to SERIAL is GENERATED AS IDENTITY. It provides the same functionality but is more compliant with the SQL standard:

sql

Copy code

CREATE TABLE example (

id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY

);

### ****Key Differences Between**** SERIAL ****and**** PRIMARY KEY ****Alone****

| Feature | SERIAL PRIMARY KEY | PRIMARY KEY (without SERIAL) |
| --- | --- | --- |
| Value Generation | Auto-generated sequential values | Manually specified |
| Sequence Creation | Yes | No |
| Default Behavior | Uses nextval(sequence) for new rows | No automatic behavior |
| Ease of Use | Simplifies unique key generation | Requires manual handling |
| Risk of Conflicts | None (handled by the sequence) | High (if multiple users insert rows) |

### ****Conclusion****

Using SERIAL PRIMARY KEY simplifies the process of creating unique identifiers in your table. If you only use PRIMARY KEY, you lose the auto-incrementing feature and must handle unique ID generation manually.

For modern PostgreSQL projects, it’s recommended to use GENERATED AS IDENTITY, but SERIAL remains widely used and fully functional.

##### You said:

then only because i use serial primary key i am not including any integers on id manually, if i dont use serial then only primary key will not provide the autoincremented numbers i should enter values like 1,2,3 in insert it could cause confusion like giving a same integer to a unique id am i right say it crisply.

###### ChatGPT said:

Yes, you're absolutely right! If you use SERIAL PRIMARY KEY, PostgreSQL automatically generates unique, sequential numbers for the id, so you don’t have to handle it manually.

If you only use PRIMARY KEY (without SERIAL), **PostgreSQL will not auto-generate numbers**. You must manually assign values (e.g., 1, 2, 3) during inserts, which can lead to confusion or errors, such as accidentally assigning the same number to multiple rows, violating the primary key's uniqueness constraint.

Using SERIAL avoids this risk and simplifies the process.

##### You said:

SELECT orders.order\_id, orders.order\_date, products.product\_name, customers.name AS customer\_name, (products.price \* orders.quantity - orders.discount) AS cost FROM tb\_order\_details orders JOIN tb\_product\_info products ON orders.product\_id = products.product\_id JOIN tb\_customer\_data customers ON orders.customer\_id = customers.customer\_id; explain this code what is this orders means is it related to our created tables?

###### ChatGPT said:

Yes, the orders used in your query is related to the tb\_order\_details table you created. Let me explain the query step by step and clarify its components.

### ****Explanation of the Query****

#### ****Query Breakdown****

1. **SELECT Clause**:
   * This specifies the columns you want in the output:
     + orders.order\_id: Fetches the order\_id from the tb\_order\_details table.
     + orders.order\_date: Fetches the order\_date from the tb\_order\_details table.
     + products.product\_name: Fetches the product\_name from the tb\_product\_info table.
     + customers.name AS customer\_name: Fetches the name from the tb\_customer\_data table and renames it to customer\_name in the output.
     + (products.price \* orders.quantity - orders.discount) AS cost: Calculates the cost of each order:
       - products.price \* orders.quantity: Total price of the products in the order.
       - - orders.discount: Subtracts the discount from the total.
       - Renames the calculated column as cost.
2. **FROM Clause**:
   * This specifies the base table for the query:
     + tb\_order\_details orders: The query starts with the tb\_order\_details table. The alias orders is given to this table for easier reference.
3. **JOIN Clauses**:
   * These are used to combine data from multiple tables based on related columns:
     + **First JOIN**: Combines tb\_order\_details (orders) with tb\_product\_info (products) using the product\_id column.
       - products ON orders.product\_id = products.product\_id: Ensures only rows where the product\_id matches in both tables are included.
     + **Second JOIN**: Combines the result with tb\_customer\_data (customers) using the customer\_id column.
       - customers ON orders.customer\_id = customers.customer\_id: Ensures only rows where the customer\_id matches in both tables are included.
4. **Aliases**:
   * orders, products, and customers are aliases for the tables tb\_order\_details, tb\_product\_info, and tb\_customer\_data, respectively.
   * Using aliases shortens the query and makes it more readable.

### ****What Does**** orders ****Mean?****

* In the query:

sql

Copy code

tb\_order\_details orders

* + tb\_order\_details is the table name.
  + orders is an **alias** for the table, which allows you to refer to the table as orders throughout the query.
* For example:
  + Instead of writing tb\_order\_details.order\_id, you can simply write orders.order\_id because of the alias.

Aliases are purely for convenience and readability; they do not create new tables or objects.

### ****Relation to Your Created Tables****

* The query uses the following tables:
  + tb\_order\_details (aliased as orders): Provides information about orders (e.g., order\_id, product\_id, customer\_id, quantity, discount, order\_date).
  + tb\_product\_info (aliased as products): Provides information about products (e.g., product\_name, price).
  + tb\_customer\_data (aliased as customers): Provides information about customers (e.g., name).

The JOIN clauses link these tables based on their relationships (product\_id and customer\_id) to generate a consolidated order summary.

### ****Purpose of the Query****

This query retrieves an **order summary** with details such as:

* Order ID
* Order Date
* Product Name
* Customer Name
* Total Cost (after considering quantity and discount)

The orders alias simply refers to the tb\_order\_details table in the context of this query.

##### You said:

what is this tod and tpi means when i write orders near that will cause any conflits first of all why postgres sql automatically giving this tod.

###### ChatGPT said:

In the screenshot, tod and tpi are aliases (shortened names) that are being used for the tables tb\_order\_details and tb\_product\_info, respectively. Here's a breakdown of what's happening and why these aliases are being used:

### ****1. What are**** tod ****and**** tpi****?****

* tod: An alias for the table tb\_order\_details.
* tpi: An alias for the table tb\_product\_info.

Aliases are used to make table references shorter and more readable in SQL queries, especially when:

* The table names are long (e.g., tb\_order\_details).
* Multiple tables are being used in joins, and the aliases help avoid repetition of long names.

### ****2. Why are**** tod ****and**** tpi ****Automatically Appearing?****

In PostgreSQL or SQL tools (like DBeaver, pgAdmin, etc.), some IDEs suggest or autocomplete aliases for tables when writing queries. These autogenerated aliases (tod for tb\_order\_details and tpi for tb\_product\_info) are often based on:

* The initials or abbreviations of the table names.
* Naming conventions or patterns detected by the SQL editor.

This is purely for convenience and doesn't affect the query's execution unless you explicitly use these aliases in the query.

### ****3. Does Writing**** orders ****Next to**** tod ****Cause Conflicts?****

Yes, it can cause **syntax issues**. Writing multiple aliases for the same table without separating them properly is not valid. For example:

sql

Copy code

FROM tb\_order\_details tod orders

In this statement:

* tod is being assigned as the alias for tb\_order\_details.
* orders appears to be another alias for the same table but is not separated correctly, causing confusion.

If you intended to use orders as the alias, you can replace tod with orders, like this:

sql

Copy code

FROM tb\_order\_details orders

JOIN tb\_product\_info tpi

Here, orders is the alias for tb\_order\_details.

### ****4. Is it Necessary to Use the IDE-Suggested Aliases (****tod****,**** tpi****)?****

No, you are not required to use the suggested aliases. You can create your own meaningful aliases. For example:

sql

Copy code

FROM tb\_order\_details orders

JOIN tb\_product\_info products

* Here, orders is the alias for tb\_order\_details.
* products is the alias for tb\_product\_info.

Using meaningful aliases (orders and products) improves the readability of your query.

### ****5. Why Use Aliases in General?****

* **Simplifies Queries**: Instead of repeating long table names, aliases make the query more concise.

sql

Copy code

SELECT orders.order\_id, products.product\_name

FROM tb\_order\_details orders

JOIN tb\_product\_info products ON orders.product\_id = products.product\_id;

vs.

sql

Copy code

SELECT tb\_order\_details.order\_id, tb\_product\_info.product\_name

FROM tb\_order\_details

JOIN tb\_product\_info ON tb\_order\_details.product\_id = tb\_product\_info.product\_id;

* **Improves Readability**: Aliases clarify which table the column belongs to, especially in queries with multiple joins.

### ****Conclusion****

* The autogenerated tod and tpi aliases are suggested by your SQL editor for convenience.
* You can override these with your own aliases (e.g., orders, products) for clarity.
* Ensure that you use only **one alias per table**, as using multiple aliases incorrectly will cause syntax errors.

##### You said:

why is this error is coming and how to rectify it.

###### ChatGPT said:

The error ERROR: role "karthik" does not exist occurs because you are trying to grant access to a role or user (Karthik) that hasn't been created in the PostgreSQL database. To fix this, you need to create the role (user) first and then grant the necessary permissions.

### ****Steps to Fix the Issue****

1. **Create the Role/User:** Run the following SQL command to create the role or user:

sql

Copy code

CREATE ROLE karthik WITH LOGIN PASSWORD 'yourpassword';

* + Replace yourpassword with a secure password for the user.
  + The WITH LOGIN clause ensures that the role can log in.

1. **Grant Privileges:** Once the role karthik exists, you can grant the SELECT privilege on the order\_summary view:

sql

Copy code

GRANT SELECT ON order\_summary TO karthik;

1. **Verify Role Creation (Optional):** To confirm the role exists, you can query the pg\_roles system catalog:

sql

Copy code

SELECT rolname FROM pg\_roles;

This should display all roles, including the newly created karthik.

1. **Test the Permissions:** Log in as karthik (or switch to the role using your tool) and attempt to query the view:

sql

Copy code

SELECT \* FROM order\_summary;

### ****Why the Error Occurs****

The error happens because PostgreSQL strictly enforces roles and privileges. If the role you are trying to assign privileges to does not exist, PostgreSQL cannot grant the permission, leading to the error.

### ****Complete Example****

If you want to set up the role and permissions in one go:

sql

Copy code

-- Step 1: Create the role with login access

CREATE ROLE karthik WITH LOGIN PASSWORD 'securepassword';

-- Step 2: Grant SELECT permission on the view

GRANT SELECT ON order\_summary TO karthik;

After running these commands, the issue will be resolved. Let me know if you encounter further challenges!

##### You said:

it is success, now where to see the karthik user name.

###### ChatGPT said:

You can view the newly created karthik role in the PostgreSQL database by querying the pg\_roles system catalog or using your database management tool (like DBeaver).

### ****1. Query to View Roles in PostgreSQL****

Run the following query to see all roles in your PostgreSQL database, including the newly created karthik:

sql

Copy code

SELECT rolname FROM pg\_roles;

This will list all the roles in your database. Look for the karthik role in the result.

### ****2. Using DBeaver to See Roles****

If you are using DBeaver, you can also find the role karthik by navigating through the database interface:

1. In the **Database Navigator** pane, locate your database connection (postgres-practice in your case).
2. Expand the **Roles** section under your database.
3. You should see the karthik role listed along with other roles.

### ****3. Additional Query to Check Role Details****

To see more details about the karthik role, such as whether it has login access, run:

sql

Copy code

SELECT rolname, rolcanlogin, rolsuper FROM pg\_roles WHERE rolname = 'karthik';

* rolcanlogin: Indicates whether the role can log in.
* rolsuper: Indicates if the role has superuser privileges.

### ****Next Steps****

Now that the karthik role is created, you can log in using its credentials (karthik with password narikootam) or proceed to grant privileges to karthik on specific objects like the order\_summary view.

Let me know if you need help with further actions!