perplexity

explain deep please

Let’s dive deeper into the query and its components to give you a comprehensive understanding of how it works, including the PostgreSQL system catalogs involved, the logic behind each part of the query, and how PostgreSQL processes it.

Purpose of the Query

This query is designed to list user-defined data types in a PostgreSQL database. It excludes system-defined types (those in `pg_catalog` and `information_schema`) and focuses on three specific kinds of types:

1. **Base types (`b`)**: These are fundamental data types, like `integer`, `text`, or custom base types created by the user.
2. **Composite types (`c`)**: These are user-defined types that group multiple fields (like a table structure).
3. **Enumerated types (`e`)**: These are user-defined types that allow a fixed set of possible values (like enums in programming languages).

The query organizes this information by schema and type name for clarity.

Step-by-Step Explanation

1. System Catalogs and Their Role

PostgreSQL stores metadata about database objects (tables, columns, types, etc.) in special tables called **system catalogs**. This query uses two key catalogs:

a) `pg_type`

- This catalog stores information about all data types in PostgreSQL.
- Each row represents a type, with details like:
  - `oid`: The unique identifier for the type.
  - `typname`: The name of the type (e.g., `integer`, `my_enum`).
  - `typtype`: The kind of type:
    - `'b'`: Base type (e.g., `integer`, `text`).
    - `'c'`: Composite type (user-defined structured type).

        - `'e'`: Enumerated type (user-defined enum).
        - `'d'`: Domain type (a constraint-based type).
        - `'p'`: Pseudo-type (special-purpose types like `record` or `any`).
    - `typnamespace`: A reference to the schema where the type is defined (points to the OID of `pg_namespace`).

b) `pg_namespace`

- This catalog stores information about schemas in the database.
- Each row represents a schema, with details like:
    - `oid`: The unique identifier for the schema.
    - `nspname`: The name of the schema (e.g., `public`, `pg_catalog`).

The query joins these two catalogs to associate each type with its schema.

2. SELECT Clause

```
SELECT
    n.nspname AS "Schema",
    t.typname AS "Type Name",
    pg_catalog.format_type(t.oid, NULL) AS "Type"
```

a) `n.nspname AS "Schema"`

- Retrieves the name of the schema (`nspname`) from the `pg_namespace` table.
- Renames it as `"Schema"` for readability.

b) `t.typname AS "Type Name"`

- Retrieves the name of the type (`typname`) from the `pg_type` table.
- Renames it as `"Type Name"` for clarity.

c) `pg_catalog.format_type(t.oid, NULL) AS "Type"`

- The function `format_type()` converts a type's OID into a human-readable format.
- Example:
    - For a base type like an integer, it returns `"integer"`.
    - For an array type or composite type, it formats it accordingly.
- The second argument (`NULL`) means no additional modifiers are applied (e.g., precision or scale for numeric types).

3. FROM Clause

```
FROM pg_catalog.pg_type t
JOIN pg_catalog.pg_namespace n ON n.oid = t.typnamespace
```

a) `pg_catalog.pg_type t`

- Specifies that we're querying from the `pg_type` catalog (aliased as `t`).

b) Join with `pg_namespace`

- Joins the `pg_namespace` table (`n`) with the `pg_type` table (`t`) using:

  ```
  ON n.oid = t.typnamespace
  ```

  - This means we're linking each type to its schema by matching their OIDs.
  - The column `t.typnamespace` in `pg_type` references the OID of its schema in `pg_namespace`.

4. WHERE Clause

```
WHERE
    n.nspname NOT IN (
        'pg_catalog',
        'information_schema'
    )
    AND t.typtype IN ('b', 'c', 'e')
```

a) Filter out system schemas

```
n.nspname NOT IN ('pg_catalog', 'information_schema')
```

- Excludes types that belong to two system schemas:
  - `pg_catalog`: Contains built-in PostgreSQL objects like standard data types (`integer`, `text`, etc.).
  - `information_schema`: Contains metadata views about database objects.

This ensures we only retrieve user-defined schemas.

b) Filter specific type categories

```
t.typtype IN ('b', 'c', 'e')
```

- Limits results to three specific kinds of types:

- `'b'`: Base types — fundamental data types like integers or custom base types created by users.
- `'c'`: Composite types — structured user-defined types that group multiple fields.
- `'e'`: Enumerated types — user-defined enums with fixed sets of values.

Other categories like domains (`d`) or pseudo-types (`p`) are excluded.

5. ORDER BY Clause

```
ORDER BY "Schema", "Type Name"
```

This sorts the results first by schema name and then alphabetically by type name within each schema. This makes it easier to navigate and analyze.

How PostgreSQL Processes This Query

1. PostgreSQL starts by scanning the system catalogs (`pg_type` and `pg_namespace`) to retrieve all rows.
2. It performs an inner join between these tables using their OIDs (`n.oid = t.typnamespace`). This links each type to its corresponding schema.
3. The result set is filtered based on:
   - Excluding system schemas (`pg_catalog`, `information_schema`).
   - Including only certain kinds of types (`b`, `c`, and `e`).
4. For each remaining row, PostgreSQL applies the formatting function (`format_type()`) to make the output more human-readable.
5. Finally, it sorts the results by schema name and then by type name.

Example Output

Suppose you have a database with:

1. A user-defined enum called `"status_enum"` in schema `"public"`.
2. A composite type called `"address_type"` in schema `"my_schema"`.
3. A base custom type called `"custom_int"` in schema `"public"`.

The query would return something like:

Schema	Type Name	Type
my_schema	address_type	my_schema.address_type
public	custom_int	public.custom_int
public	status_enum	public.status_enum

Why Is This Query Useful?

1. **Database Introspection**: Helps you understand what custom data structures exist in your database.
2. **Excludes Noise**: Filters out built-in PostgreSQL objects so you can focus on user-defined ones.
3. **Schema Organization**: Groups results by schema, making it easier to see where each type is defined.

This is particularly valuable for developers working on complex databases with many custom data structures!