PostgreSQL

Created by Dakshil Gorasiya

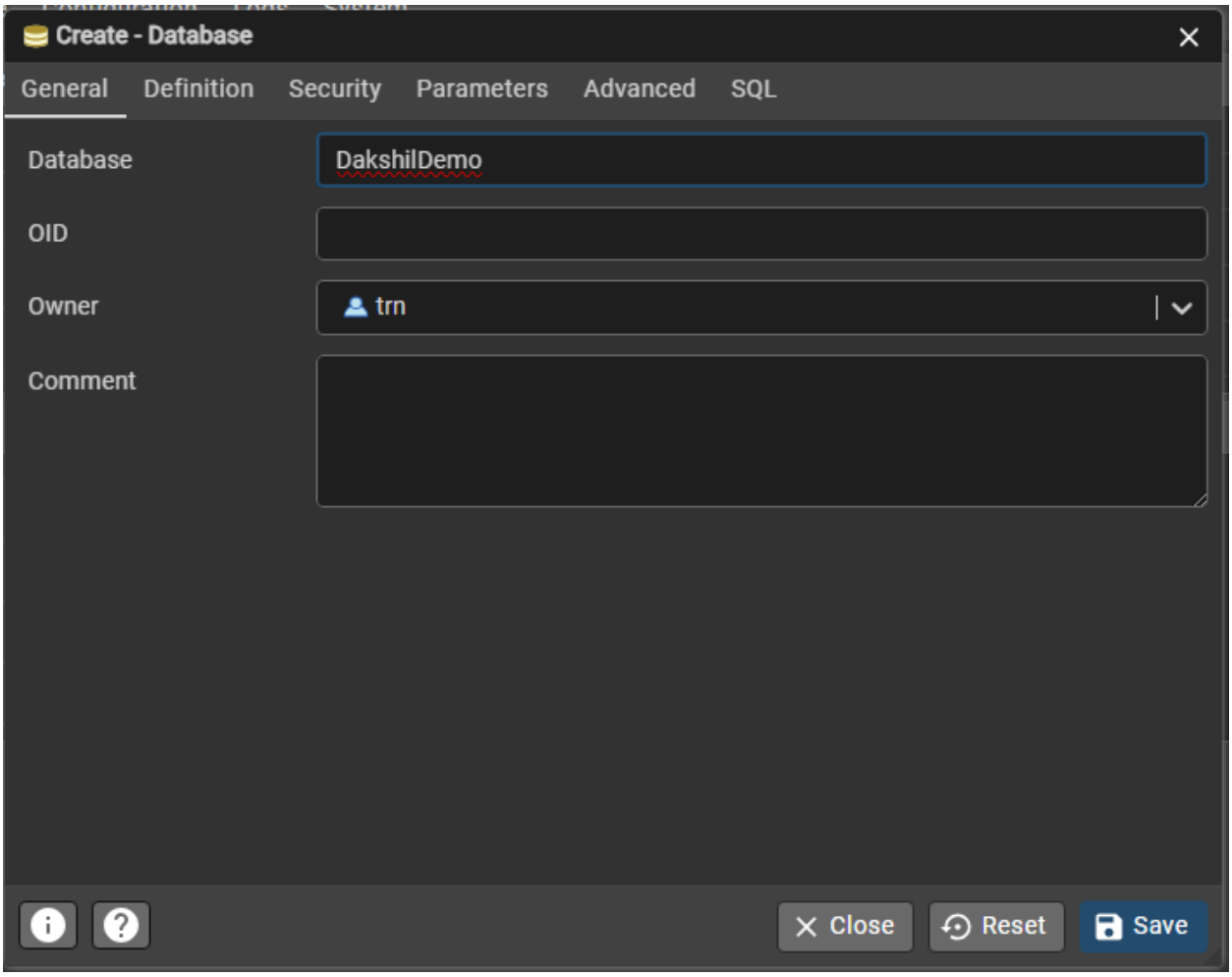
PostgreSQL is a powerful, open source object-relational database system with over 35 years of active development that has earned it a strong reputation for reliability, feature robustness, and performance.

PostgreSQL has earned a strong reputation for its proven architecture, reliability, data integrity, robust feature set, extensibility, and the dedication of the open source community behind the software to consistently deliver performant and innovative solutions.

It is known for:

- **Strong consistency & ACID compliance**
- **Advanced SQL features**
- **Extensibility**
- **High reliability & data integrity**

Create database

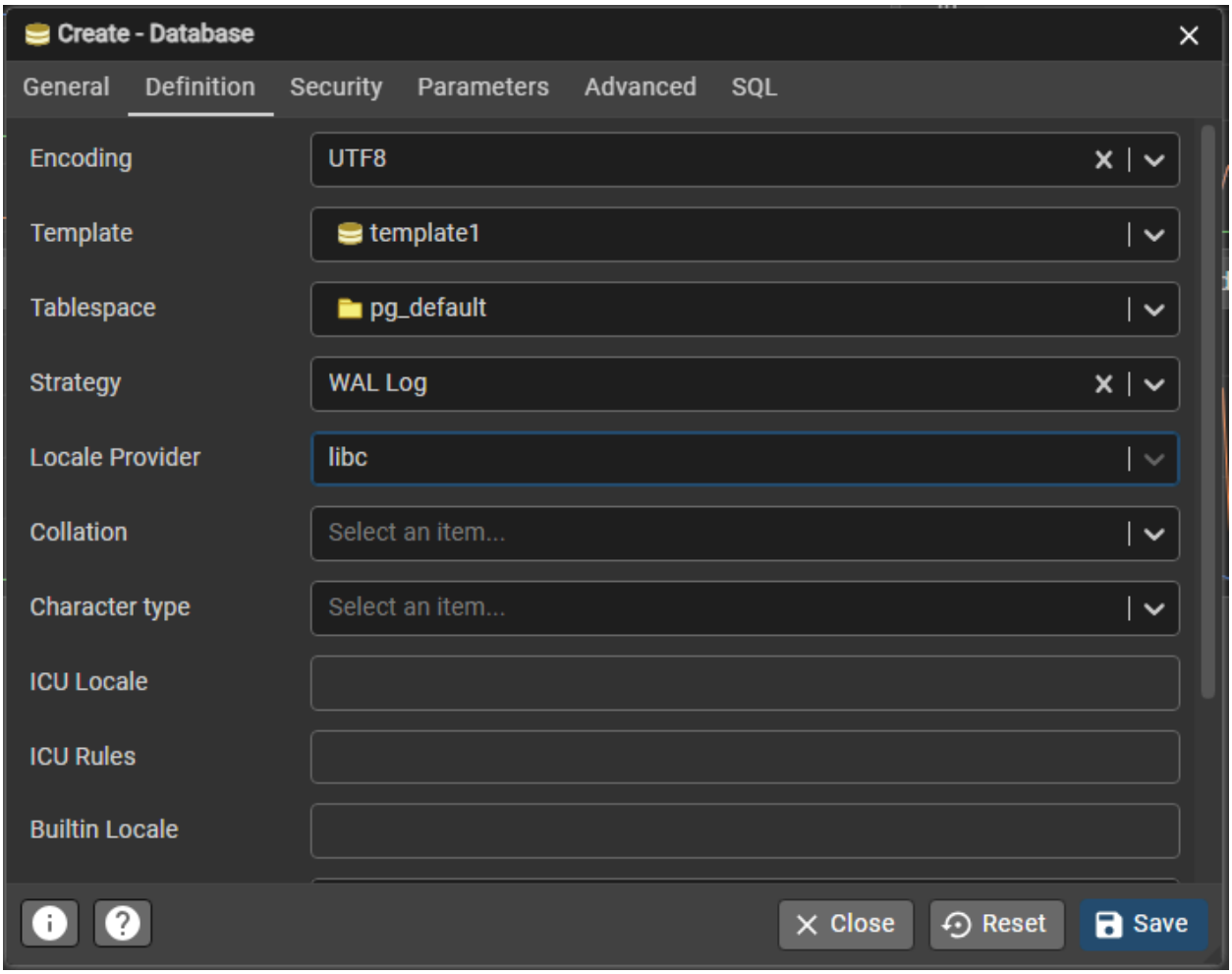


`Database`: The name of a database to create.

`OID`: The object identifier to be used for the new database. If this parameter is not specified, `PostgreSQL` will choose a suitable OID automatically. This parameter is primarily intended for internal use by pg_upgrade, and only pg_upgrade can specify a value less than 16384.

`Owner`: The **PostgreSQL role (user)** who owns this database.

`Comment`: A **human-readable description** of the database.



`Encoding`: Character set encoding to use in the new database. Specify a string constant (e.g., `'SQL_ASCII'`), or an integer encoding number, or `DEFAULT` to use the default encoding (namely, the encoding of the template database).

`Template`: The name of the template from which to create the new database, or `DEFAULT` to use the default template (`template1`).

`Tablespace`: Defines **where database files are stored on disk**.

`Strategy`: Strategy to be used in creating the new database. If the `WAL_LOG` strategy is used, the database will be copied block by block and each block will be separately written to the write-ahead log. This is the most efficient strategy in cases where the template database is small, and therefore it is the default. The older `FILE_COPY` strategy is also available. This strategy writes a small record to the write-ahead log for each tablespace used by the target database. Each such record represents copying an entire directory to a new location at the filesystem

level. While this does reduce the write-ahead log volume substantially, especially if the template database is large, it also forces the system to perform a checkpoint both before and after the creation of the new database. In some situations, this may have a noticeable negative impact on overall system performance.
The `FILE_COPY` strategy is affected by the **file_copy_method** setting.

`Locale Provider`: Defines **which system library handles language rules**. (libc: os based, icu: modern custom)

`Collation`: Defines **how strings are sorted and compared**.

`Character type`: Upper/lowercase behavior

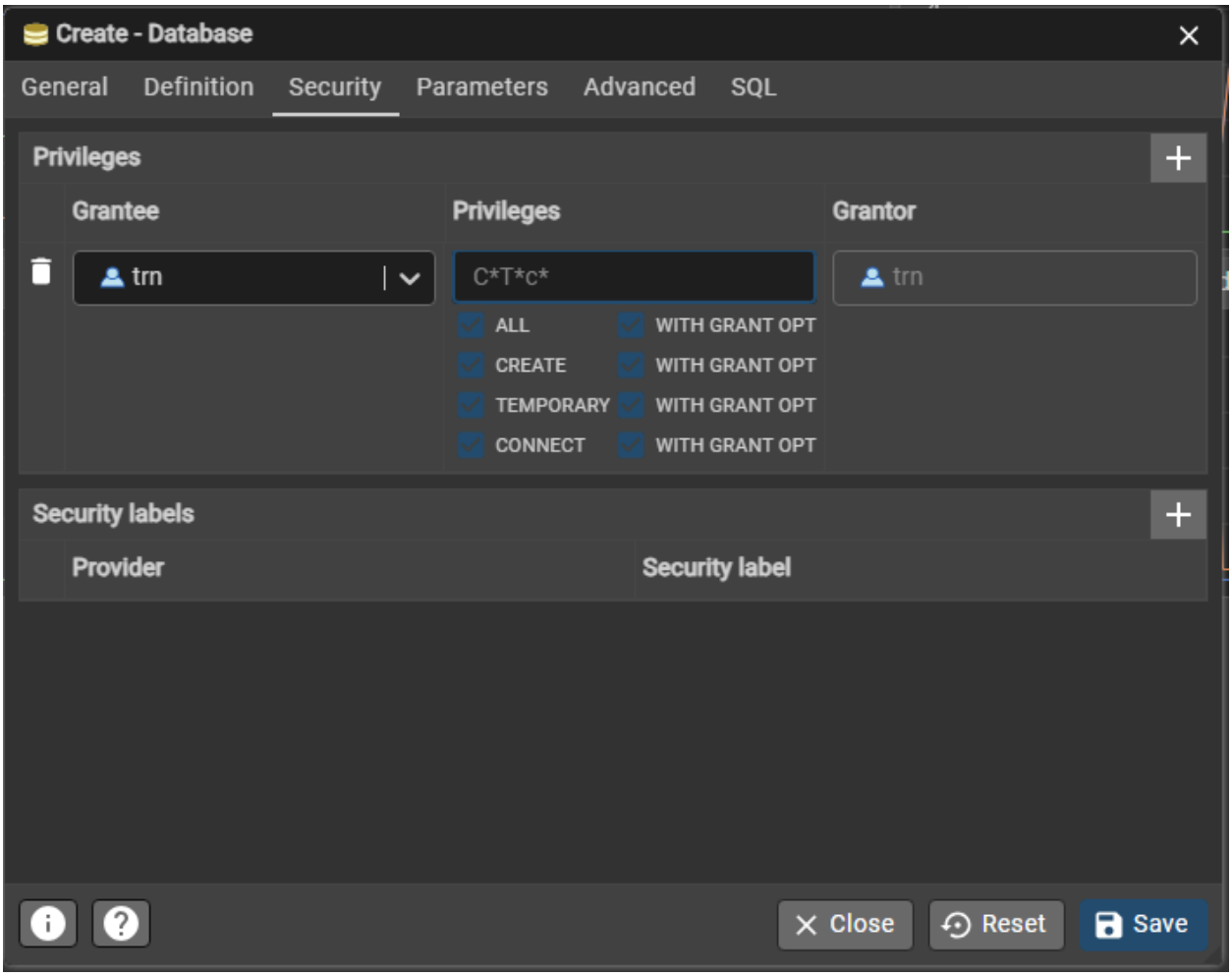
`ICU Locale`: Used **only if Locale Provider = ICU,** Defines Unicode language rules.

`ICU Rules`: Advanced custom rules for sorting & comparison.

`Builtin Locale`: PostgreSQL's internal locale instead of OS locale.

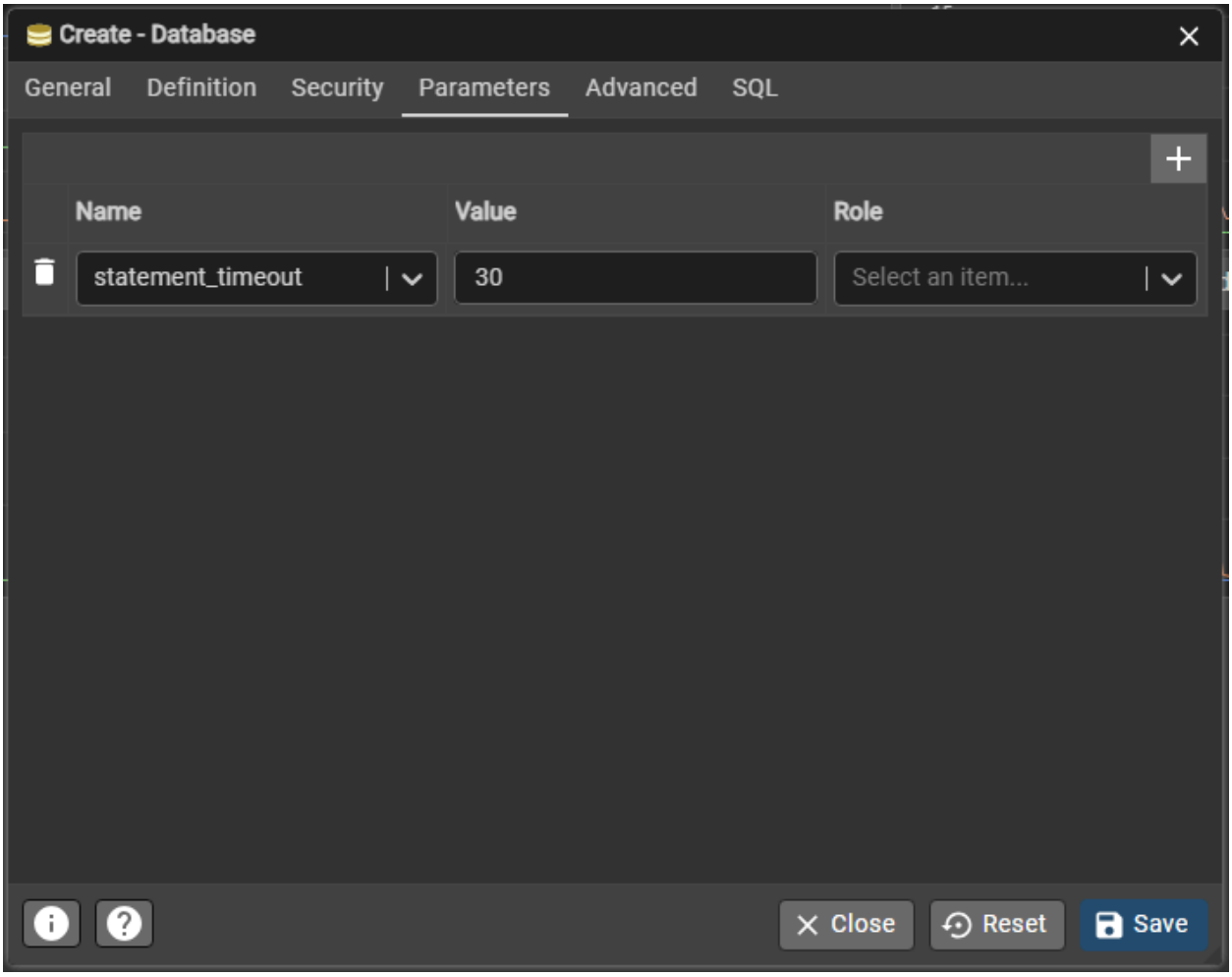
`Connection Limit`: Maximum number of **simultaneous connections** to this database.

`Is_template` : Can be used as template

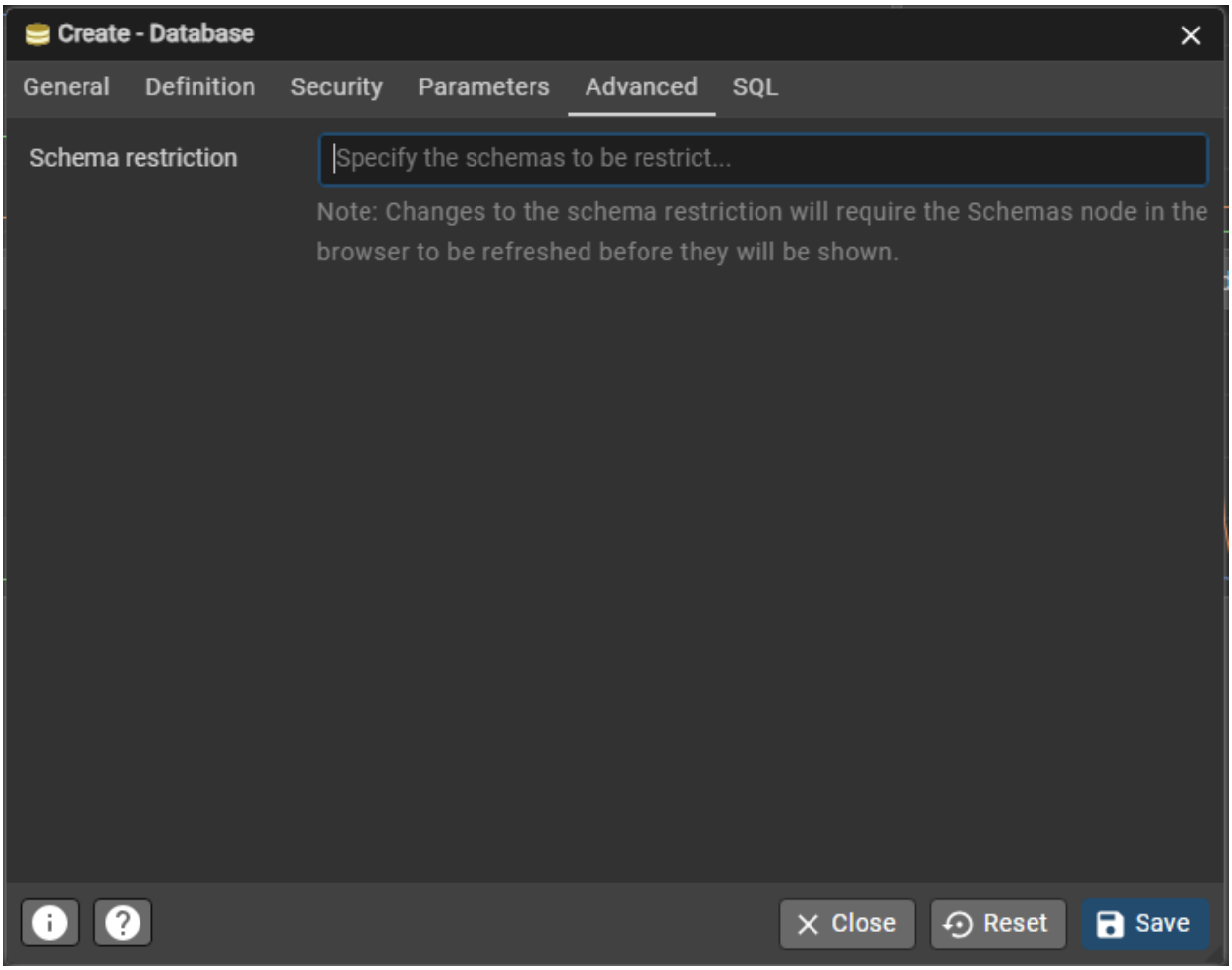


`Privileges`: permissions on a database object

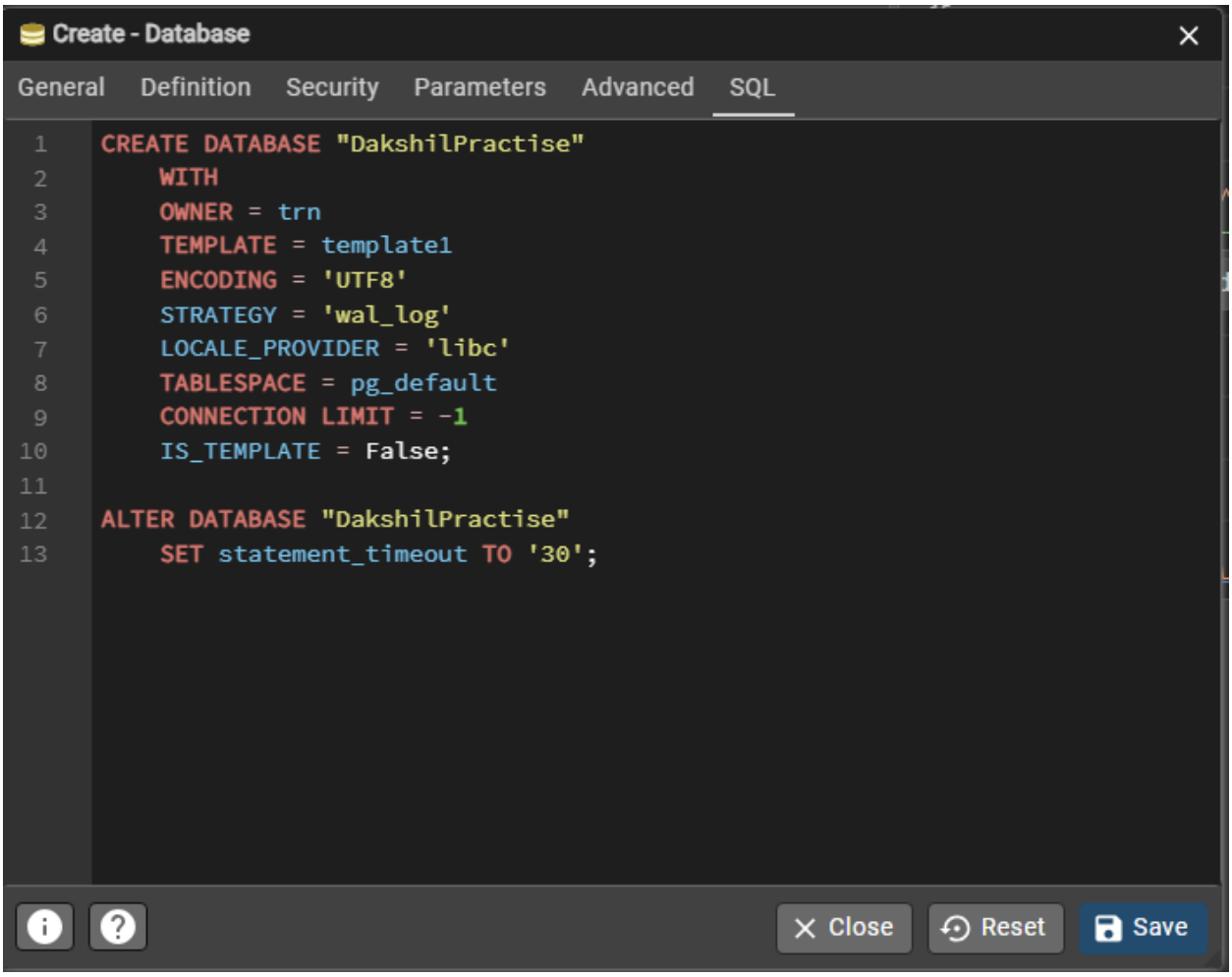
`Security labels`: metadata tags for external security systems. They do nothing by themselves. They are used with SELinux, External security extensions, Mandatory Access Control (MAC)



Set parameter at database level without touching `postgresql.conf`



Restrict current user from accessing schemas stronger than `grant`



To review SQL query

Roles

Roles in `PostgreSQL` is way different than `MySQL`

There is no concept of users in PostgreSQL
Roles are defined at the **database cluster level**, not per individual database. This means if you create a role named `app_user`, it exists for every database in that instance.
A role is just an entity. What makes it a "user" is the `LOGIN` attribute. What makes it a "group" is simply the ability to have other roles as members.

Roles own database objects (tables, functions) and can assign privileges to other roles.

Ex:

```
-- Creating a "Group" role (cannot log in)
CREATE ROLE engineering_team;

-- Creating a "User" role (can log in)
CREATE ROLE alice WITH LOGIN PASSWORD 'secure_password';

-- Adding Alice to the Engineering group
GRANT engineering_team TO alice;
```

To give permission to a role

```
GRANT <PRIVILEGE> ON <OBJECT> TO <ROLE>;
```

Internal working

There is a table named `pg_authid` which has field like `rolname`, `rolsuper`, `rolcanlogin`, `rolcreaterole`, `rolpassword` , `oid`

`pg_auth_members`: to store relationship like `group_id`, `member_id`

To switch role

```
SET ROLE devops;
```

RLS (Row level security)

We can restrict user from accessing some rows in a table

```
CREATE POLICY us_only_policy ON sales_data
FOR SELECT
TO intern_role
USING ( true );
```

In using we can apply condition same as select's where

Data type

Data Type	Description	Example
`boolean`	Stores logical true/false values	`TRUE`, `FALSE`
`char(n)`	Fixed-length character string	`CHAR(5) → 'Hello'`
`varchar(n)`	Variable-length character string with limit	`VARCHAR(50) → 'PostgreSQL'`
`text`	Variable-length text, no limit	`'This is long text'`
`numeric(p,s)` / `decimal`	Exact precision decimal number (used for money)	`NUMERIC(10,2) → 12345.67`
`double precision`	8-byte floating-point number (approximate)	`123.456789`
`float`	Floating-point number (precision varies)	`FLOAT → 3.14`
`real`	4-byte floating-point number	`REAL → 3.14159`
`smallint`	2-byte integer	`32767`
`integer` / `int`	4-byte integer	`100000`
`bigint`	8-byte integer	`9223372036854775807`
`date`	Stores date (YYYY-MM-DD)	`'2025-02-04'`
`timestamp`	Date and time (without timezone)	`'2025-02-04 10:30:00'`
`timestamptz`	Date and time with timezone	`'2025-02-04 10:30:00+05:30'`
`interval`	Time span or duration	`'2 days 3 hours'`
`time`	Time of day (no date)	`'14:45:00'`

Data Type	Description	Example
`uuid`	Universally unique identifier	`'550e8400-e29b-41d4-a716-446655440000'`
`json`	Stores JSON data (text-based)	`'{"name":"Dakshil"}'`
`hstore`	Key-value pairs (extension required)	`'key => value'`
`array`	Stores array of values	`INTEGER[] → '{1,2,3}'`
`xml`	Stores XML data	`'<user><id>1</id></user>'`
`bytea`	Binary data (images, files)	`'\xDEADBEEF'`

Composite type

A **composite type** is a **user-defined data type** that groups **multiple fields (columns)** into **one structured type**, similar to:

- a **struct** in C
- a **class without methods**
- a **row type** in a table

```
CREATE TYPE address AS (
    street   TEXT,
    city     TEXT,
    pincode  INTEGER
);
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name TEXT,
    home_address address
);
INSERT INTO users (name, home_address)
VALUES ('Dakshil', ROW('MG Road', 'Ahmedabad', 380015));
SELECT
    (home_address).street,
```

```
    (home_address).city
FROM users;
```

Type creation using DOMAIN

A DOMAIN is a custom data type built on top of an existing data type, with constraints attached.

```
CREATE DOMAIN adult_age INT
CHECK (VALUE >= 18);
CREATE TABLE users (
    id SERIAL,
    name TEXT,
    age adult_age
);
```

Serial vs Identity

Serial

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name TEXT
);
```

Identity

```
CREATE TABLE users (
    id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
```

```
    name TEXT
);
```

Aspect	`SERIAL`	`IDENTITY`
Type	**Pseudo-type**	**SQL standard feature**
SQL Standard	PostgreSQL-specific	SQL:2003 standard
Sequence ownership	Loosely linked	Strictly bound to column
Manual insert allowed	Yes	`ALWAYS` → No
Override value	Always allowed	Only with `OVERRIDING SYSTEM VALUE`
Dependency handling	Can break if sequence dropped	Safer, managed automatically
Recommended today	Legacy	Modern & preferred

DQL

DISTINCT ON

The `DISTINCT ON` clause allows you to retrieve unique rows based on specified columns

The `DISTINCT ON` clause retrieves the first unique entry from each column or combination of columns in a result set.

The key factor for determining which unique entry is selected lies in the columns that appear in the `ORDER BY` clause.

Technically, you can use the `DISTINCT ON` without the `ORDER BY` clause. However, without the `ORDER BY` clause, the “first” unique entry becomes unpredictable because the table stores the rows in an unspecified order.

Ex:

Data

```
INSERT INTO student_scores (name, subject, score)
VALUES
  ('Alice', 'Math', 90),
  ('Bob', 'Math', 85),
  ('Alice', 'Physics', 92),
  ('Bob', 'Physics', 88),
  ('Charlie', 'Math', 95),
  ('Charlie', 'Physics', 90);
```

Query

```
SELECT
  DISTINCT ON (name) name,
  subject,
  score
FROM
  student_scores
ORDER BY
  name,
  score DESC;
```

Result

```
    name   | subject | score
---------+---------+-------
 Alice   | Physics |    92
 Bob     | Physics |    88
 Charlie | Math    |    95
(3 rows)
```

FETCH

Same as limit but compatible with SQL standards

```
OFFSET row_to_skip { ROW | ROWS }
FETCH { FIRST | NEXT } [ row_count ] { ROW | ROWS } ONLY
```

FIRST = NEXT, ROW = ROWS

```
SELECT
    film_id,
    title
FROM
    film
ORDER BY
    title
FETCH FIRST ROW ONLY;
```

Escape character in LIKE operator

```
SELECT * FROM t
WHERE message LIKE '%10$%%' ESCAPE '$';
```

It will match → anything before 10% anything after

JOIN

LEFT JOIN = LEFT OUTER JOIN

Left Anti join

left join that returns rows from the left table that do not have matching rows from the right table

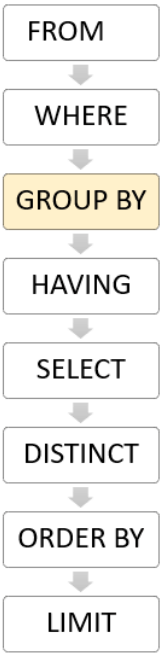
```
SELECT
    *
```

```
FROM
    T1
    LEFT JOIN T2 ON T1.ID = T2.T1_ID
WHERE
    T2.T1_ID IS NULL
```

Natural JOIN

```
SELECT select_list
FROM table1
NATURAL [INNER, LEFT, RIGHT] JOIN table2;
```

Default is INNER



View

Materialized view

Materialized view is view with data physically stored.

Materialized views cache the result set of an expensive query and allow you to refresh data periodically.

To create materialized view

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

To refresh materialized view

```
REFRESH MATERIALIZED VIEW view_name;
```

In above statement table will be locked to avoid it we can do it concurrently

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

Recursive view

A recursive view is a view whose defining query references the view name itself.

Syntax:

```
CREATE RECURSIVE VIEW view_name(columns)
AS
query;
```

Ex:

```
CREATE RECURSIVE VIEW reporting_line (employee_id, subordinat
es) AS
SELECT
  employee_id,
  full_name AS subordinates
FROM
  employees
WHERE
  manager_id IS NULL
UNION ALL
SELECT
  e.employee_id,
  (
    rl.subordinates || ' > ' || e.full_name
  ) AS subordinates
FROM
  employees e
  INNER JOIN reporting_line rl ON e.manager_id = rl.employee_
id;
```

Error messages

Syntax:

```
raise level format;
```

level

- debug
- log
- notice
- info

- warning
- exception

The `format` is a string that specifies the message. The `format` uses percentage (`%`) placeholders that will be substituted by the arguments.

The number of placeholders must be the same as the number of arguments. Otherwise, PostgreSQL will issue an error

Raising error

To raise an error, you use the `exception` level after the `raise` statement.
The `raise` statement uses the `exception` level by default.

Besides raising an error, you can add more information by using the following additional clause:

```
using option = expression
```

The `option` can be:

- `hint`: provide the hint message so that the root cause of the error is easier to discover.
- `detail`: give detailed information about the error.
- `errcode`: identify the error code, which can be either by condition name or an `SQLSTATE` code.

Ex:

```
do $$
declare
  email varchar(255) := 'john.doe@example.com';
begin
  -- check email for duplicate
  -- ...
  -- report duplicate email
  raise exception 'duplicate email: %', email
```

```
        using hint = 'check the email again';
end $$;
```

```
do $$
begin
    --...
    raise sqlstate '77777';
end $$;
```

Assertion

The `assert` statement is a useful shorthand for inserting debugging checks into PL/pgSQL code.

```
assert condition [, message];
```

The `condition` is a Boolean expression that is expected to always return `true`.

If the `condition` evaluates to `true`, the `assert` statement does nothing.

In case the `condition` evaluates to `false` or `null`, PostgreSQL raises an `assert_failure` exception.

The `message` is optional.

If you don't pass the `message`, PostgreSQL uses the "`assertion failed`" message by default. In case you pass the `message` to the `assert` statement, PostgreSQL will use it instead of the default message.

```
do $$
declare
   film_count integer;
begin
```

```
   select count(*)
   into film_count
   from film;

   assert film_count < 0, 'Film not found, check the film tab
le';
end$$;
```

PLSQL

For loop

Syntax

```
[ <<label>> ]
for loop_counter in [ reverse ] from.. to [ by step ] loop
    statements
end loop [ label ];
```

Ex

```
do
$$
begin
   for counter in 1..5 loop
            raise notice 'counter: %', counter;
   end loop;
end;
$$;
```

For loop to iterate over result set

Syntax

```
[ <<label>> ]
for target in query loop
    statements
end loop [ label ];
```

Ex

```
do
$$
declare
    f record;
begin
    for f in select title, length
           from film
           order by length desc, title
           limit 10
    loop
    raise notice '%(% mins)', f.title, f.length;
    end loop;
end;
$$
```

Dynamic query in for loop

```
do $$
declare
    -- sort by 1: title, 2: release year
    sort_type smallint := 1;
    -- return the number of films
    rec_count int := 10;
    -- use to iterate over the film
    rec record;
    -- dynamic query
```

```
    query text;
begin

    query := 'select title, release_year from film ';

    if sort_type = 1 then
        query := query || 'order by title';
    elsif sort_type = 2 then
      query := query || 'order by release_year';
    else
       raise 'invalid sort type %s', sort_type;
    end if;

    query := query || ' limit $1';   // inject variable here

    for rec in execute query using rec_count
        loop
         raise notice '% - %', rec.release_year, rec.title;
    end loop;
end;
$$
```