PostgreSQL

Created by Dakshil Gorasiya

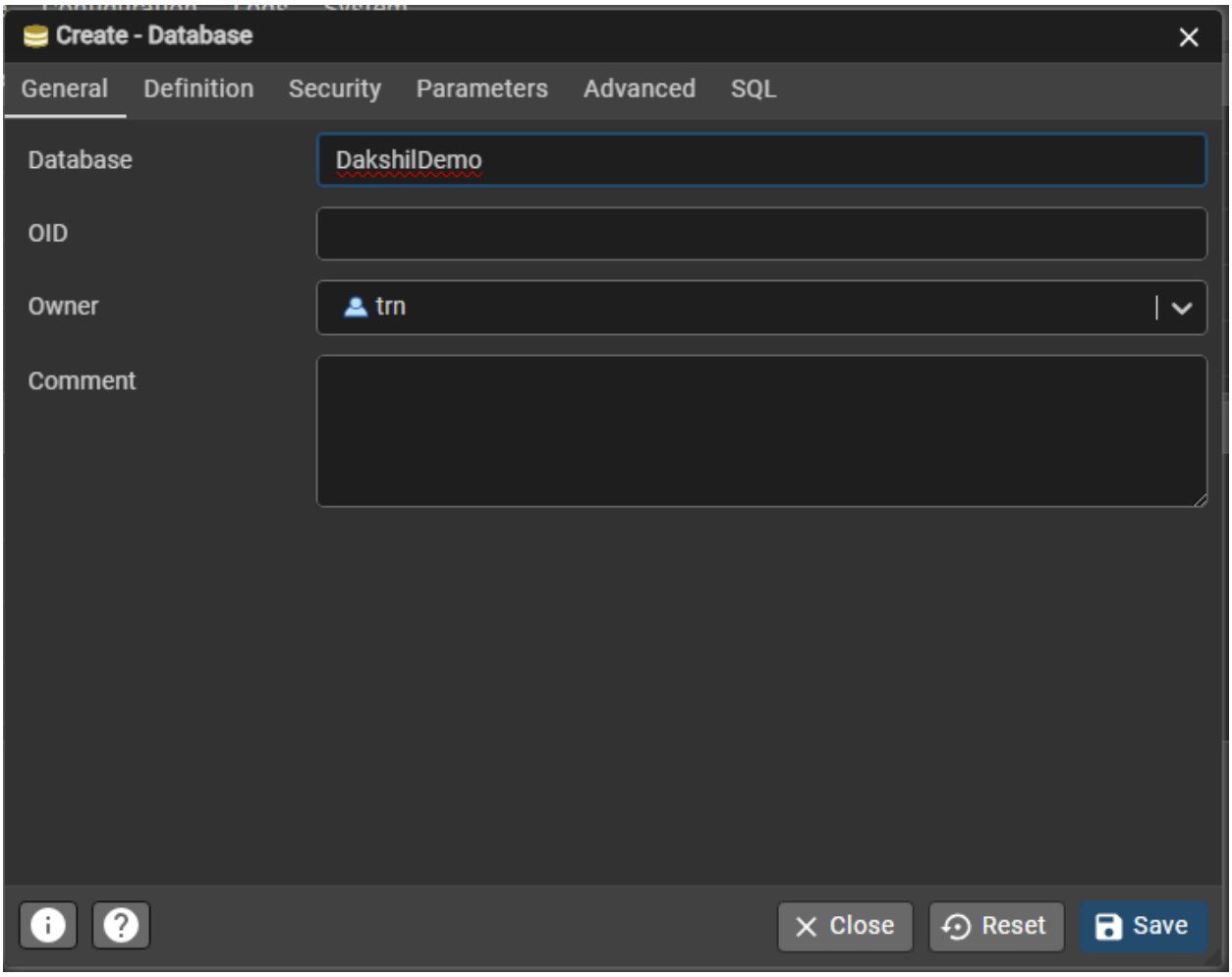
PostgreSQL is a powerful, open source object-relational database system with over 35 years of active development that has earned it a strong reputation for reliability, feature robustness, and performance.

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

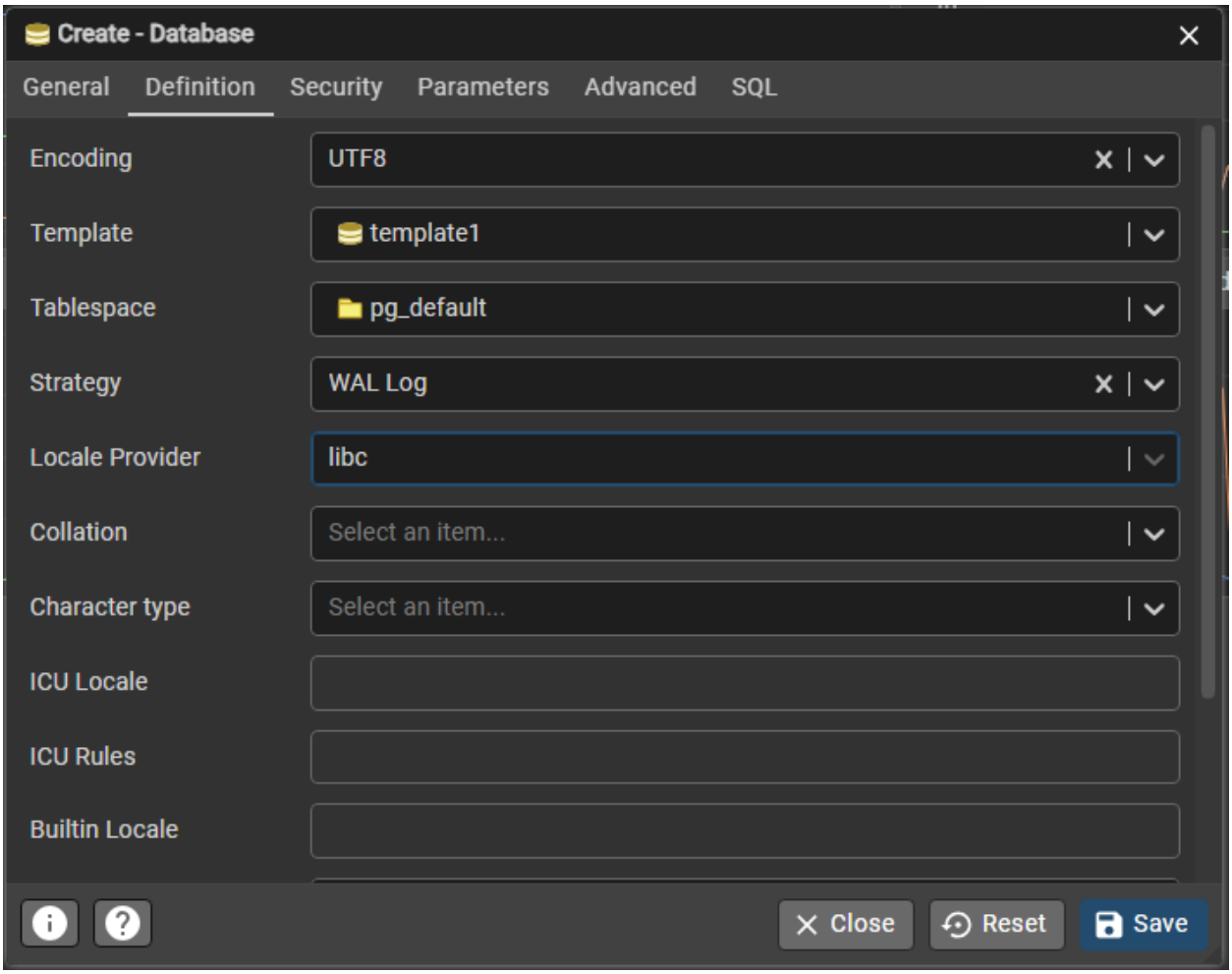


`Database`: The name of a database to create.

`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

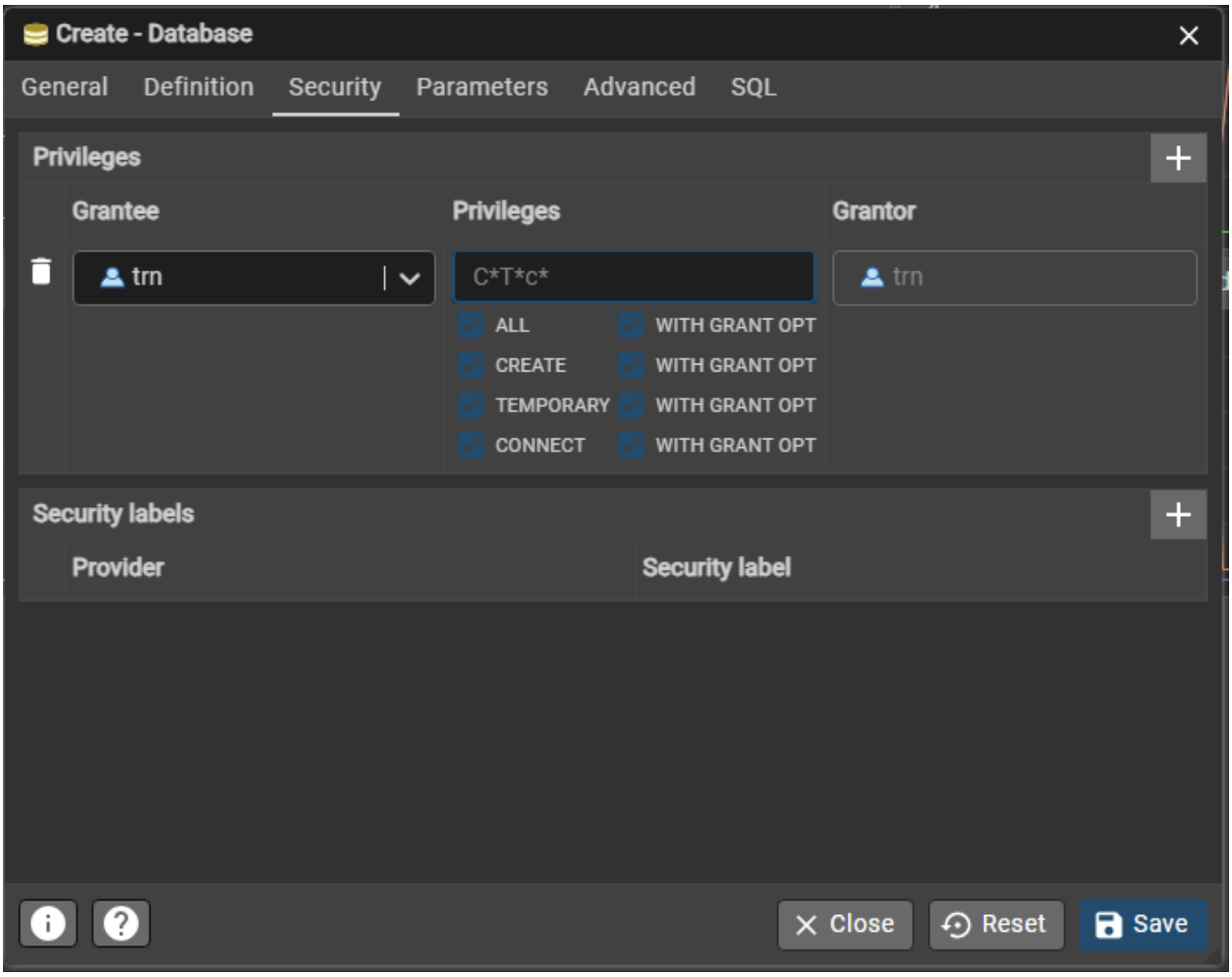
`ICU Locale`: Used **only if Locale Provider = ICU,** Defines Unicode language rules.

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

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

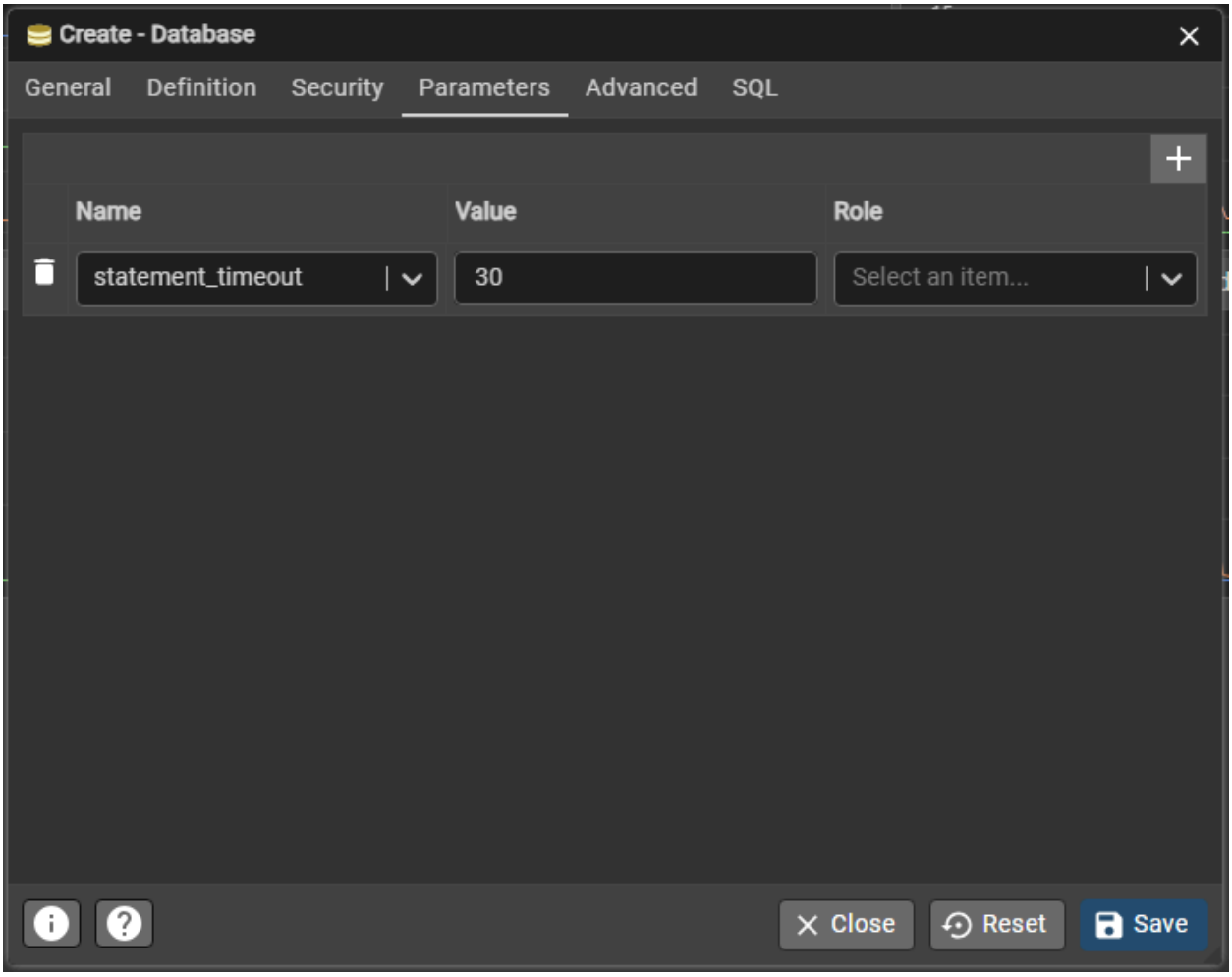
`Connection Limit`: Maximum number of **simultaneous connections** to this database.

`Is_template` : Can be used as template

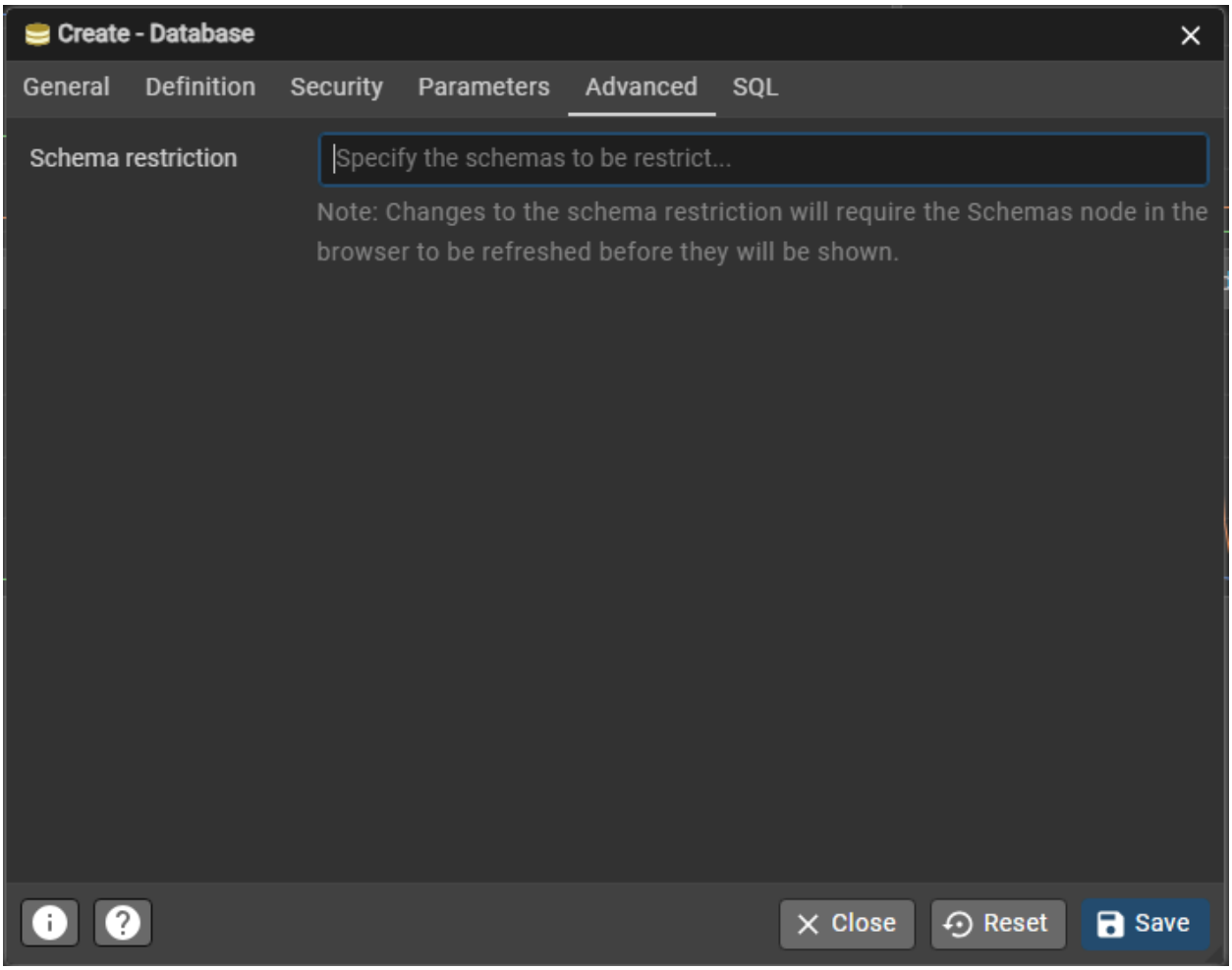


`Privileges`: permissions on a database object

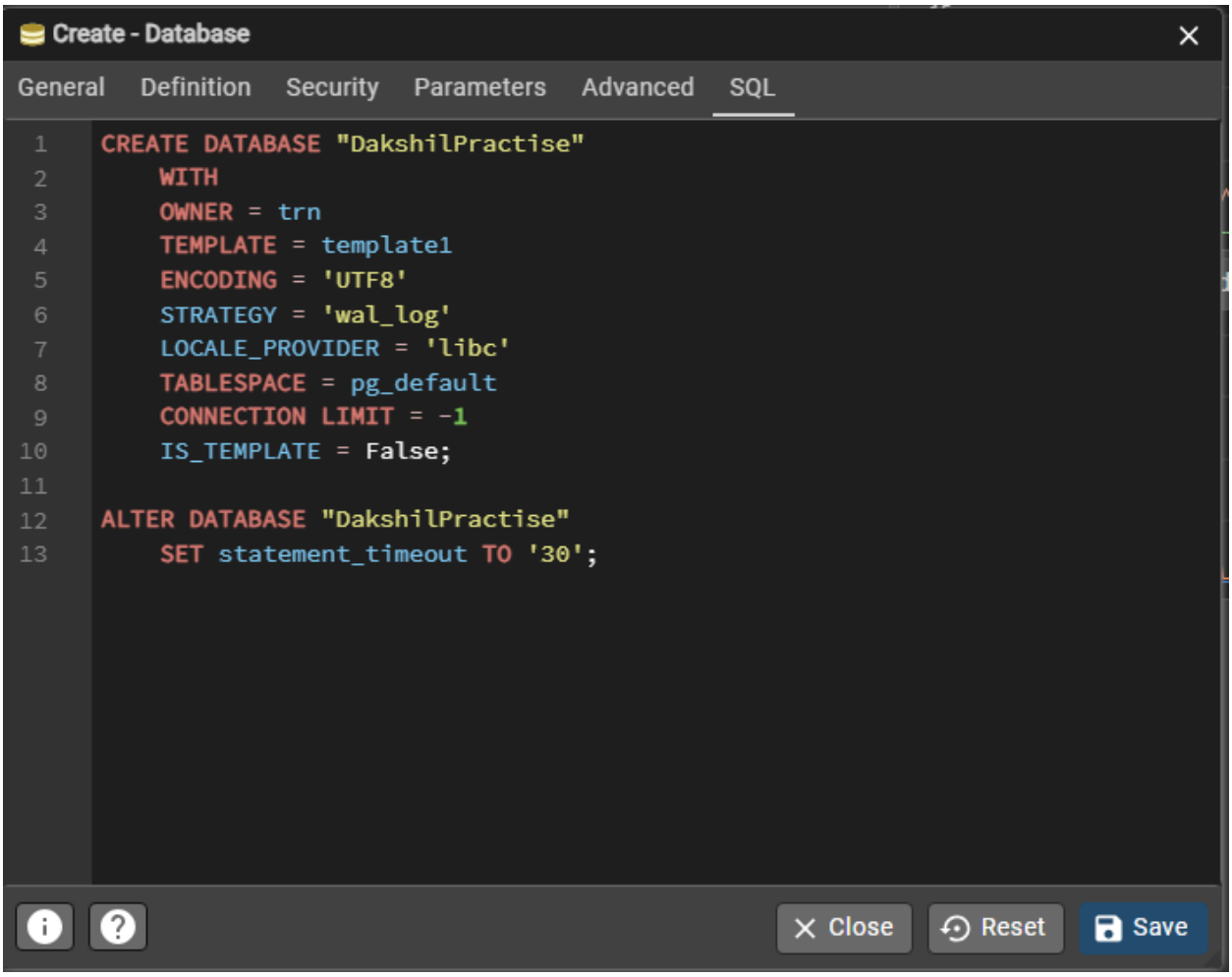
`Security labels`: metadata tags for external security systems. They do nothing by themselves. They are used with SELinux, External security extensions, Mandatory Access Control (MAC)



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

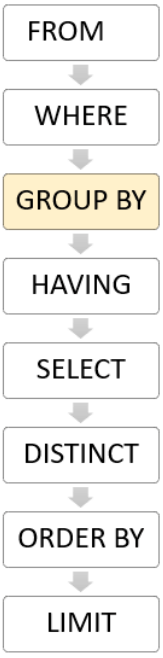
```
SELECT
    *
```

```
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;
$$
```

Partition

In PostgreSQL partition table doesn’t automatically created like MySQL

To create table with partition

```
CREATE TABLE SALES (
    ID BIGINT,
    SALE_DATE DATE,
    COUNTRY TEXT,
    AMOUNT NUMERIC
```

```
)
PARTITION BY
    RANGE (SALE_DATE);
```

To create partition

```
CREATE TABLE SALES_2024 PARTITION OF SALES FOR
VALUES
FROM
    ('2024-01-01') TO ('2025-01-01');

CREATE TABLE SALES_2025 PARTITION OF SALES FOR
VALUES
FROM
    ('2025-01-01') TO ('2026-01-01');
```

To delete partition

```
DROP TABLE SALES_2024;
```

To detach partition but preserve data

```
ALTER TABLE SALES DETACH PARTITION SALES_2024;
```

To reattach that partition

```
ALTER TABLE SALES ATTACH PARTITION SALES_2024 FOR
VALUES
FROM
    ('2024-01-01') TO ('2025-01-01');
```

INDEX

To create index

```
CREATE INDEX IDX_ADDRESS_PHONE ON ADDRESS (PHONE);
```

To create unique index

```
CREATE UNIQUE INDEX index_name
ON table_name (column [, ...])
[ NULLS [ NOT ] DISTINCT ];
```

NULLS DISTINCT → will allow multiple null values

NULLS NOT DISTINCT → will allow only one null value in column

To create index on expression

```
CREATE INDEX IDX_LC_LAST_NAME ON CUSTOMER (LOWER(LAST_NAME));
```

To create partial index

```
CREATE INDEX CUSTOMER_ACTIVE ON CUSTOMER (ACTIVE)
WHERE
    ACTIVE = 0;
```

To create multi-column index

```
CREATE INDEX [IF NOT EXISTS] index_name
ON table_name(column1, column2, ...);
```

if a query does *not* constrain the first column of the index, PostgreSQL must evaluate whether a full index scan on this index is more efficient than alternative indexes or a table scan.

REINDEX

In practice, an index can become corrupted and no longer contain valid data due to hardware failures or software bugs.

Additionally, when you create an index without the `CONCURRENTLY` option, the index may become invalid if the index build fails.

In such cases, you can rebuild the index. To rebuild the index, you can use the `REINDEX` statement as follows:

```
REINDEX [ ( option, ...) ]
{ INDEX | TABLE | SCHEMA | DATABASE | SYSTEM }
name;
```

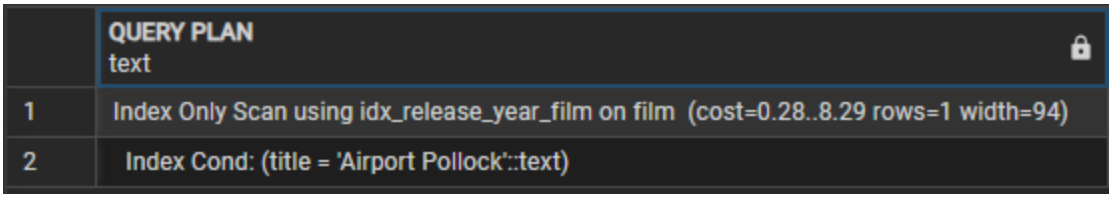
The `option` can be one or more values:

- `VERBOSE [boolean]` – show the progress as each index is reindexed.
- `TABLESPACE new_tablespace` – specify the new tablespace on which the indexes will be rebuilt.
- `CONCURRENTLY` – rebuild the index without taking any locks. If not used, reindex will lock out writes but not reads on the table until it is completed.

To list indexes

```
SELECT
    tablename,
    indexname,
    indexdef
FROM
    pg_indexes
WHERE
    schemaname = 'public'
ORDER BY
    tablename,
    indexname;
```

Covering Index
A **covering index** is an index that contains **all the columns needed by a query**, so PostgreSQL can answer the query **using only the index**, without touching the table.
In explain if postgres doesn't need to touch table than output will be like



	QUERY PLAN text
1	Index Only Scan using idx_release_year_film on film (cost=0.28..8.29 rows=1 width=94)
2	Index Cond: (title = 'Airport Pollock'::text)

To add extra column to index use syntax

```
CREATE INDEX IDX_RELEASE_YEAR_FILM ON FILM (TITLE) INCLUDE (D
ESCRIPTION);
```

Description will not be sorted but only stored with index.

Index types

1. B-tree index

B-tree is a self balancing tree.

It can be user for query that uses <, ≤, =, ≥, BETWEEN, IN, IS NULL, IS NOT NULL

```
CREATE INDEX idx_users_email ON users USING btree (email);
```

2. Hash index

Hash indexes can handle only simple equality comparison (=).

```
CREATE INDEX index_name
ON table_name USING HASH (indexed_column);
```

3. BRIN

BRIN = Block Range INdex

Instead of indexing **each row**, a BRIN index stores **summary information per block range** (group of heap pages).

Internally it will create block of 128 page and store min and max value like this

Block range	Min value	Max value
0–127	2022-01-01	2022-01-31
128–255	2022-02-01	2022-02-28
256–383	2022-03-01	2022-03-31

BRIN is perfect when data is:

- **Very large** (millions/billions of rows)
- **Physically ordered or correlated**
- **Append-only or mostly insert-at-end**

BRIN performs poorly if:

- Data is **randomly distributed**
- Frequent **UPDATEs** move rows
- Queries need **exact matches**

JSON/JSONB

Feature	JSON	JSONB
Storage	Textual representation	Binary storage format
Size	Typically larger because it retains the whitespace in JSON data	Typically smaller
Indexing	Full-text search indexes	Binary indexes
Performance	Slightly slower	Generally faster
Query performance	Slower due to parsing	Faster due to binary storage
Parsing	Parse each time	Parse once, store in binary format
Data manipulation	Simple and easy	More complex
Ordering of keys	Preserved	Not preserved
Duplicate keys	Allow duplicate key, the last value is retained	Do not allow duplicate keys.
Use cases	Storing configuration data, log data, simple JSON documents	Storing JSON documents where fast querying and indexing are required

Operators

Operator	Syntax	Meaning
`->`	`jsonb->'key'`	Extract the value of the 'key' from a JSON object as a JSONB value
`->>`	`jsonb->>'key'`	Extract the value of the 'key' from a JSON object as a text string
`@>`	`jsonb @> jsonb → boolean`	Return true if the first JSONB value contains the second JSONB value or false otherwise.
`<@`	`jsonb <@ jsonb → boolean`	Return true if the first JSONB value is contained in the second one or false otherwise.
`?`	`jsonb ? text → boolean`	Return true if a text string exists as a top-level key of a JSON object or as an element of a JSON array or false

Operator	Syntax	Meaning
		otherwise.
`?|`	`jsonb ?| text[] → boolean`	Return true if any text string in an array exists as top-level keys of a JSON object or as elements of a JSON array.
`?&`	`jsonb ?& text[] → boolean`	Return true if all text strings in an array exist as top-level keys of a JSON object or as elements of a JSON array.
`||`	`jsonb || jsonb → jsonb`	Concatenate two JSONB values into one.
`-`	`jsonb - text → jsonb`	Delete a key (and its value) from a JSON object, or matching string value(s) from a JSON array.
`-`	`jsonb - text[] → jsonb`	Delete all matching keys or array elements from the left operand.
`-`	`jsonb - integer → jsonb`	Delete the array element with specified index (negative integers count from the end of the array).
`#-`	`jsonb #- text[] → jsonb`	Delete the field or array element at the specified path.
`@?`	`jsonb @? jsonpath → boolean`	Return true if a JSON path returns any item for the specified JSONB value.
`@@`	`jsonb @@ jsonpath → boolean`	Evaluate a JSON path against a JSONB value and return a boolean result based on whether the JSON path matches any items within the JSONB value

To create GIN index on json data

```
CREATE INDEX IDX_GIN_JSON_DATA ON PRODUCTS USING GIN (DATA);
```

We can also specify operator class for json

```
CREATE INDEX index_name
ON table_name
USING GIN(jsonb_column jsonb_path_ops);
```

Name	Indexable Operators
`array_ops`	`&& (anyarray,anyarray)`
	`@> (anyarray,anyarray)`
	`<@ (anyarray,anyarray)`
	`= (anyarray,anyarray)`
`jsonb_ops`	`@> (jsonb,jsonb)`
	`@? (jsonb,jsonpath)`
	`@@ (jsonb,jsonpath)`
	`? (jsonb,text)`
	`?| (jsonb,text[])`
	`?& (jsonb,text[])`
`jsonb_path_ops`	`@> (jsonb,jsonb)`
	`@? (jsonb,jsonpath)`
	`@@ (jsonb,jsonpath)`
`tsvector_ops`	`@@ (tsvector,tsquery)`

Window Function

Similar to an aggregate function, a window function operates on a set of rows. However, it does not reduce the number of rows returned by the query.

The term *window* describes the set of rows on which the window function operates. A window function returns values from the rows in a window.

Syntax

```
window_function(arg1, arg2,..) OVER (
   [PARTITION BY partition_expression]
   [ORDER BY sort_expression [ASC | DESC] [NULLS {FIRST | LAS
T }])
```

Name	Description
CUME_DIST	Return the relative rank of the current row.
DENSE_RANK	Rank the current row within its partition without gaps.
FIRST_VALUE	Return a value evaluated against the first row within its partition.
LAG	Return a value evaluated at the row that is at a specified physical offset row before the current row within the partition.
LAST_VALUE	Return a value evaluated against the last row within its partition.
LEAD	Return a value evaluated at the row that is offset rows after the current row within the partition.
NTILE	Divide rows in a partition as equally as possible and assign each row an integer starting from 1 to the argument value.
NTH_VALUE	Return a value evaluated against the nth row in an ordered partition.
PERCENT_RANK	Return the relative rank of the current row (rank-1) / (total rows – 1)
RANK	Rank the current row within its partition with gaps.
ROW_NUMBER	Number the current row within its partition starting from 1.

When using window function like FIRST_VALUE, LAST_VALUE use

```
PARTITION BY group_name
    ORDER BY price
    RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
```

as default is

```
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

So frame ends to current row and result will be unexpected

CUME_DIST

Cumulative Distribution: What fraction of rows are less than or equal to the current row (based on ORDER BY)? It returns value between $0 < \text{value} \leq 1$

Ex:

price	rows ≤ price	CUME_DIST
10	1	1/4 = 0.25
20	3	3/4 = 0.75
20	3	3/4 = 0.75
40	4	4/4 = 1.00

NTILE

`NTILE(n)` divides ordered rows into **n equal groups (buckets)**.

It assigns a **bucket number (1 to n)** to each row.

Ex:

price	bucket
10	1
20	1
30	2
40	2
50	3
60	3
70	4
80	4

PERCENT_RANK

`PERCENT_RANK()` gives the **relative rank of a row as a percentage** within its partition.

Ex.

price	Rank	PERCENT_RANK
10	1	0.00
20	2	(2-1)/3 = 0.33
20	2	0.33
40	4	(4-1)/3 = 1.00

EXPLAIN

```
-- Which 3 film categories have generated the highest total r
evenue?
EXPLAIN
SELECT
    CATEGORY.NAME,
    SUM(AMOUNT) AS TOTAL_REVENUE
FROM
    CATEGORY
    INNER JOIN FILM_CATEGORY ON CATEGORY.CATEGORY_ID = FILM_C
ATEGORY.CATEGORY_ID
    INNER JOIN FILM ON FILM.FILM_ID = FILM_CATEGORY.FILM_ID
    INNER JOIN INVENTORY ON INVENTORY.FILM_ID = FILM.FILM_ID
    INNER JOIN RENTAL ON RENTAL.INVENTORY_ID = INVENTORY.INVE
NTORY_ID
    INNER JOIN PAYMENT ON PAYMENT.RENTAL_ID = RENTAL.RENTAL_I
D
GROUP BY
    CATEGORY.NAME
ORDER BY
    TOTAL_REVENUE DESC
LIMIT
    3;
```

```
"QUERY PLAN"
"Limit  (cost=1390.09..1390.10 rows=3 width=100)"
"  ->  Sort  (cost=1390.09..1390.13 rows=16 width=100)"
"        Sort Key: (sum(payment.amount)) DESC"
"        ->  HashAggregate  (cost=1389.68..1389.88 rows=16 wi
dth=100)"
```

```
"               Group Key: category.name"
"               ->  Hash Join  (cost=785.37..1316.70 rows=1459
6 width=74)"
"                     Hash Cond: (inventory.film_id = film.fil
m_id)"
"                     ->  Hash Join  (cost=639.06..969.70 rows
=14596 width=8)"
"                           Hash Cond: (rental.inventory_id =
inventory.inventory_id)"
"                           ->  Hash Join  (cost=510.99..803.2
8 rows=14596 width=10)"
"                                 Hash Cond: (payment.rental_i
d = rental.rental_id)"
"                                 ->  Seq Scan on payment  (co
st=0.00..253.96 rows=14596 width=10)"
"                                 ->  Hash  (cost=310.44..310.
44 rows=16044 width=8)"
"                                       ->  Seq Scan on rental
(cost=0.00..310.44 rows=16044 width=8)"
"                           ->  Hash  (cost=70.81..70.81 rows=
4581 width=6)"
"                                 ->  Seq Scan on inventory
(cost=0.00..70.81 rows=4581 width=6)"
"                     ->  Hash  (cost=133.81..133.81 rows=1000
width=74)"
"                           ->  Hash Join  (cost=111.86..133.8
1 rows=1000 width=74)"
"                                 Hash Cond: (film_category.fi
lm_id = film.film_id)"
"                                 ->  Hash Join  (cost=1.36..2
0.67 rows=1000 width=70)"
"                                       Hash Cond: (film_categ
ory.category_id = category.category_id)"
"                                       ->  Seq Scan on film_c
ategory  (cost=0.00..16.00 rows=1000 width=4)"
"                                       ->  Hash  (cost=1.16..
```

```
1.16 rows=16 width=72)"
"                                                  ->  Seq Scan on
category  (cost=0.00..1.16 rows=16 width=72)"
"                                    ->  Hash  (cost=98.00..98.00
rows=1000 width=4)"
"                                          ->  Seq Scan on film
(cost=0.00..98.00 rows=1000 width=4)"
```

How to read it

start from most right arrow

ex. Seq scan on inventory means it read entire table sequentially

In above it is Hash means it will prepare internal hash table for inventory table

In above there is hash join means generated hash table is used for join

at top there is limit which is used for limit, sort which is used for order by, hashaggregate which is used for group by

If do explain with analyze we see
" Buffers: shared hit=45"

means ram is used for reading if we see dirty there means there is something which should be write back to disk so a background job will be created

Red flags to see in plan

1. Check for seq scan it should be not be used when finding small number of rows from large number of rows (Ex. to find 5 rows it check 5 million rows)
2. Check for sort, HashAggregate if is we see Disk:..KB means it is using disk to store intermediate stage
3. Check number of estimated rows and actual rows if large difference found db may be choosing wrong method for query to get correct estimate rows run query `ANALYZE table_name` to calculate statistics again.

Vacuum/Analyze

VACUUM and **ANALYZE** are essential maintenance tasks that keep your database healthy and fast. Unlike some databases that overwrite old data in place, PostgreSQL uses a system called **MVCC** (Multi-Version Concurrency Control). When you `UPDATE` or `DELETE` a row, PostgreSQL doesn't physically remove it from the disk immediately. Instead, it marks the old version as "dead." Over time, these dead rows accumulate, leading to **bloat**, which slows down queries and wastes space.

Vacuum

The primary job of `VACUUM` is to reclaim the space occupied by these "dead" rows so it can be reused by new data.

- **Prevents Bloat:** By cleaning up dead rows, it keeps the physical size of your tables manageable.
- **Transaction ID Wraparound:** It prevents a critical failure where the database runs out of unique transaction IDs.
- **Performance Impact:** It reduces the amount of data the system has to scan during a query, leading to faster I/O.

Analyze

`ANALYZE` doesn't touch the data itself; instead, it collects statistics about the distribution of data within a table.

- **Planner Statistics:** It updates internal tables with information like how many rows are in a table and how many unique values exist in a column.
- **Optimizing Queries:** The **Query Planner** uses these statistics to decide the most efficient way to run a query (e.g., "Should I use an index or a full table scan?").

- **Performance Impact:** Without up-to-date statistics, the planner might choose a slow, inefficient path to retrieve your data.

Autovaccum

To show settings of autovaccum

```
SHOW autovacuum_vacuum_cost_limit;
SHOW autovacuum_vacuum_cost_delay;
```

To find out which table has more dead rows use query

```
SELECT
    relname AS table_name,
    n_live_tup AS live_rows,
    n_dead_tup AS dead_rows,
    CAST(n_dead_tup AS FLOAT) / NULLIF(n_live_tup, 0) AS dead
_ratio
FROM pg_stat_user_tables
ORDER BY dead_ratio DESC;
```

To change autovaccume for global system either edit `postgresql.conf` or run these commands

```
-- Make autovacuum 5x more aggressive globally and permanentl
y
ALTER SYSTEM SET autovacuum_vacuum_cost_limit = 1000;
ALTER SYSTEM SET autovacuum_vacuum_cost_delay = '2ms';

-- After running this, you MUST reload the configuration to a
pply it
SELECT pg_reload_conf();
```

To change autovaccum for a specific table use query

```
ALTER TABLE sessions SET (
  autovacuum_vacuum_scale_factor = 0.01,   -- Clean after onl
y 1% change
  autovacuum_analyze_scale_factor = 0.005, -- Update stats af
ter 0.5% change
  autovacuum_vacuum_cost_limit = 1000      -- Give this speci
fic job more "power"
);
```

Locks and Transaction

ACID - Atomicity, Consistency, Isolation, and Durability.

Atomicity - Either transaction is fully completed or aborted it cannot be in intermediate state for longer time

Consistency - When transaction completes (committed or rollbacked) it move database from one consistence state to other

Isolation - It is about parallel transaction behavior.

Durability - Once transaction is committed its effect should be there always.

Isolation

transaction isolation level defines how visible changes made by one transaction are to other concurrent transactions.

PostgreSQL support 4 isolation level (3 effectively)

1. Read uncommitted

In the **Read Uncommitted** level, Transaction B *can* see the uncommitted changes made by Transaction A. This is called a **Dirty Read**.

It's "dirty" because Transaction A might eventually rollback. If that happens, Transaction B is essentially working with data that "never technically existed." PostgreSQL is a bit unique here: it actually treats *Read Uncommitted* the same as the next level up, effectively skipping this dangerous behavior.

2. Read committed

This is the default transaction level of PostgreSQL

It allows to read data that is already committed so withing one transaction if read data 2 times we might get two different result

3. Repeatable Read

It take snapshot of db when we start new transaction so withing transaction we always get same result even if some other transaction changed data.

4. Serializable

Let's look at a classic example: **The Hospital Shift**. 🏥

- **The Rule:** The hospital requires at least **1** doctor to be on call at all times.
- **Current State:** Doctor A and Doctor B are both currently "On Call."

Now, imagine this happens simultaneously in `REPEATABLE READ` mode:

1. **Doctor A** checks the schedule. She sees **2** doctors on call. She thinks, "Great, if I leave, there will still be 1 left." So she updates her status to "Off Duty."
2. **Doctor B** checks the schedule (using his own snapshot). He *also* sees **2** doctors on call. He thinks, "Great, if I leave, there will still be 1 left." So he updates his status to "Off Duty."

Since they are updating *different* rows (Alice updates Alice, Bob updates Bob), there is no row-lock conflict.

and at last hospital will have no active doctor

Serializable solves this

PostgreSQL monitors the data that each transaction "read" (even if it didn't lock it). It realizes that if **Transaction A** (Doctor A) finishes, it invalidates the premise of **Transaction B** (Doctor B's count of doctors).

Because the database cannot mathematically make both transactions happen safely at the same time, it will **kill one of them**.

- **Doctor A:** Transaction Commits. Success! (She goes off duty).
- **Doctor B:** Tries to commit, but gets an error: `ERROR: could not serialize access due to read/write dependencies among transactions`.

How it handles it as there is no row level lock breach?
When you run a query in `SERIALIZABLE` mode, PostgreSQL places invisible markers called **SIREAD locks** (or Predicate Locks) on the data you *looked* at.

- **Doctor A** runs `SELECT count(*) FROM doctors WHERE on_call = true`.
  - Postgres attaches a "tripwire" to the set of rows that match `on_call = true`. It essentially notes: *"Transaction A cares about the set of on-call doctors."*
- **Doctor B** does the same thing.
  - Postgres notes: *"Transaction B also cares about the set of on-call doctors."*

Now, when they try to update:

1. **Doctor A** tries to update her own status (change `on_call` to `false`).

- Postgres sees that Doctor A is changing a row that **Transaction B is watching** (via that invisible tripwire). It records a "dependency."

2. **Doctor B** tries to update *his* status.
    - Postgres sees that Doctor B is changing a row that **Transaction A is watching**.

Postgres detects a **cycle** in these dependencies: A's action affects B's view, and B's action affects A's view. Since both cannot be true based on what they originally read, Postgres immediately aborts one of them to break the cycle.

```
BEGIN ISOLATION LEVEL SERIALIZABLE;
BEGIN ISOLATION LEVEL REPEATABLE READ;
BEGIN ISOLATION LEVEL READ COMMITTED;
```

Advisory lock

Think of an Advisory Lock as a **global semaphore** or a "flag" stored in the database.

- **Standard Locks:** "I am locking Row #5 in the Users table." (The database enforces this).
- **Advisory Locks:** "I am locking the concept of 'PDF Report Generation'." (The database holds the flag, but **your code** decides what it means).

It allows your database to act as the traffic controller for your application logic, completely independent of the actual data in your tables.

To acquire lock

```
SELECT pg_advisory_lock(5000);
```

To release lock

```
SELECT pg_advisory_unlock(5000);
```

Here 5000 is unique ID for lock which application agree to use

There is one crucial detail here. Standard locks (like row updates) release automatically when a transaction commits (`COMMIT;`).

Advisory locks are different. They are usually bound to your **Session** (your connection to the database), *not* the transaction.

If your code crashes *after* taking the lock but *before* unlocking it, but the connection to the database stays open (like in a connection pool), lock will not be released.

To avoid this we can use transaction level advisory lock

```
select pg_advisory_xact_lock(4000);
select pg_advisory_xact_unlock(4000);
```