Postgres Internals Hiding in Plain Sight

Postgres has an awesome amount of data collected in its own internal tables. Postgres hackers know all about this - but software developers and folks working with day to day Postgres tasks often miss out the good stuff.

The Postgres catalog is how Postgres keeps track of itself. Of course, Postgres would do this in a relational database with its own schema. Throughout the years several nice features have been added to the internal tables like psql tools and views that make navigating Postgres' internal tables even easier.

Today I want to walk through some of the most important Postgres internal data catalog details. What they are, what is in them, and how they might help you understand more about what is happening inside your database.

psql's catalog information

The easiest way to get at some of Postgres' internal catalogs is to use the built-in psql commands that begin \d generally. Here's some common Postgres ones users should be comfortable using:

`\d {tablename}` : describes a specific table. \d will do a lot of things if you qualify \d with a table or view name.

`\di` : list all your indexes

`\dx` : list installed extensions

`\dp` : to show access privileges

`\dp+` : tables and views with the roles and access details

`\dconfig` : your current configuration settings

`\dt {tablename}` : describe a table

`\dti+` : tables and indexes with sizes

`\dg+` : show role names

`\df` : show your functions

`\dv {view name}` : describe a view

`\l` : lists all your databases

Important Postgres catalog views

Postgres exposes many of the complex internals of the database system in easy-to-query views. These host a wealth of information about what is going on inside your database and direct SQL access to answer in the moment emergency questions like “what is taking up all my CPU” and more long term questions like “what are my 10 slowest queries”.

pg_stat_activity

Shows current database activity, including running queries, state, and client information. Essential for troubleshooting and getting process ids (pid) for bad actors.

```
SELECT pid, usename, datname, client_addr, application_name, state, query
FROM pg_stat_activity
WHERE state != 'idle'
ORDER BY state, query_start DESC;
```

pg_stat_statements

This requires the pg_stat_statements extension - but it is part of the contrib library and ships with Postgres, so doesn't require separate installation.

This view tracks execution statistics for all queries executed by all databases. It's incredibly powerful for identifying slow or frequently executed queries.

```
-- pg_stat_statements 10 longest running queries
SELECT query, calls, total_exec_time, mean_exec_time, rows
FROM pg_stat_statements
ORDER BY total_exec_time DESC
LIMIT 10;
```

pg_stat_database

This view provides database-wide statistics, such as the number of connections, transactions, and I/O. It's useful for a high-level overview of database activity and health.

```
-- high leve db stats for the postgres db
SELECT datname,numbackends, xact_commit, xact_rollback, blks_read, blks_hit
FROM pg_stat_database
WHERE datname = 'postgres';
```

pg_locks

This view displays information about locks held by active processes. This is the go to place for troubleshooting locking issues, deadlocks, and contention within the database. We have a great blog on locking and **how to find the source of the lock in Postgres**.

```
-- locks joined with the activity table. Shows not granted locks, typically those that could not be granted because th
SELECT a.datname, l.pid, l.locktype, l.relation::regclass, l.mode, l.granted
FROM pg_locks l
JOIN pg_stat_activity a ON l.pid = a.pid
WHERE NOT l.granted;
```

pg_stat_user_tables

This view offers statistics on tables, including sequential scans, index scans, and row-level operations (inserts, updates, deletes). It's great for identifying tables with heavy activity or those that need vacuuming.

```
-- see sequence scans and index scans by table
SELECT relname AS table_name, seq_scan, idx_scan
FROM pg_stat_user_tables
WHERE seq_scan > 0 OR idx_scan > 0 ORDER BY seq_scan DESC;
```

pg_stat_user_indexes

This view provides statistics on user indexes, such as how often they're used and how many tuples are read. This is particularly herpful for finding unused or underutilized indexes.

```
-- Never used indexes in Postgres sorted by size
SELECT s.schemaname, s.relname AS table_name, s.indexrelname AS index_name, pg_size_pretty(pg_relation_size(s.indexrel
FROM pg_stat_user_indexes AS s
JOIN pg_index AS i ON s.indexrelid = i.indexrelid
WHERE s.idx_scan = 0 AND i.indisunique IS FALSE
ORDER BY pg_relation_size(s.indexrelid) DESC;
```

pg_settings

This is a prebuilt view that is super useful for viewing configuration parameters, their current values, and their descriptions. Qualify with **`ILIKE`** to see exact parameters you're looking for.

```
-- find shared_buffer or work_mem settings
SELECT name, setting, unit, short_desc
FROM pg_settings
WHERE name LIKE '%shared_buffers%' OR name LIKE '%work_mem%';
```

pg_roles

This view describes all system roles, which include users and groups. It's useful for checking permissions, login capabilities, and role memberships.

```
-- This query lists all roles, showing their names, whether they can log in, and their password expiration date.
SELECT rolname, rolcanlogin, rolvaliduntil
FROM pg_roles
ORDER BY rolname;
```

pg_database

This view contains all databases in the cluster. It provides key metadata for each database, including its owner, character encoding, and access privileges. We have a lot of folks now that create dozens and sometimes hundreds of databases for development, so this is a good high level view.

```
-- This query lists all Postgres databases, their sizes, and owners.
SELECT d.datname AS database_name, pg_size_pretty(pg_database_size(d.datname)) AS database_size, pg_get_userbyid(d.dat
FROM pg_database AS d
WHERE d.datistemplate = false;
```

Postgres catalog tables

Behind the Postgres metacommands and views - there are several core catalog tables. Many of the psql commands match up with the catalog tables. Something roughly like this:

psql command	what data	catalog tables
\d	tables and table objects	pg_class
\di	indexes	pg_class, pg_index
\dx	installed extensions	pg_extension
\dp	tables and privileges	pg_class, pg_roles, pg_attribute
\l	databases	pg_database
\df	available functions	pg_proc

Let's look at these and how you might want to use them.

pg_stats

The pg_stats table collects all the details about your columns - things like cardinality - are there many items in this column or a few? Postgres uses a lot of the details in pg_stats to make decisions for the query planner and efficiently. In some cases, giving **pg_stats more information can make your queries faster.**

```
-- table column data like cardinality
SELECT * FROM pg_stats
WHERE tablename = 'table_name'
AND attname = 'column_name';
```

pg_class

pg_class contains a row for every table, index, sequence, view, materialized view, and other "relation-like" objects in the database. Sometimes this is a nice high level view of an entire table's accoutrements.

```
SELECT c.relname, pg_get_userbyid(c.relowner) AS owner
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE n.nspname = 'public' AND c.relkind = 'r'
ORDER BY c.relname;
```

pg_type

This table stores all data types that exist. It's confusing though - in Postgres, every table has an associated composite type that defines the structure of its rows. So if you do a `select *` you'll see all the table names here and all the data types. If you filter a bit, you can see all your custom data types, domains, and enums.

```
-- see your custom data types in Postgres
SELECT
    t.typname AS type_name,
    n.nspname AS schema_name,
    t.typtype AS type_class
FROM
    pg_type AS t
JOIN
    pg_namespace AS n ON t.typnamespace = n.oid
LEFT JOIN pg_class c ON typrelid = c.oid
WHERE
    t.typtype IN ('e', 'd', 'c') -- 'e' for enum, 'd' for domain, 'c' for composite types.
    AND n.nspname NOT IN ('pg_catalog', 'information_schema', 'pg_toast')
    AND (t.typtype <> 'c' OR c.relkind = 'c')
```

```
ORDER BY
    schema_name, type_name;
```

pg_proc

This is the catalog of all functions and stored procedures that Postgres can use. It contains metadata about each routine. Made a function last week but can't find it now? Just scan through all of them.

```
-- This query finds all functions, triggers, and stored procedures.
SELECT proname AS function_name, proargnames AS argument_names, pg_catalog.format_type(prorettype, NULL) AS return_typ
FROM pg_proc
ORDER BY proname;
```

pg_attribute

This table stores information about table columns and there is one row in **`pg_attribute`** for every column in every table. While indexes and other objects that have an entry in **`pg_class`** .

Query columns and data types for any table with a query like this:

```
SELECT
    a.attname AS column_name,
    pg_catalog.format_type(a.atttypid, a.atttypmod) AS data_type
FROM
    pg_catalog.pg_attribute a
WHERE
    a.attrelid = 'orders'::regclass
    AND a.attnum > 0
    AND NOT a.attisdropped
ORDER BY
    a.attnum;
```

pg_catalog schema

The pg_catalog is the schema holding the system tables, so you will either need to include `pg_catalog` in your `search_path` (the default), or any query you issue will need to be qualified with `pg_catalog`.

Here's a summary of the internal catalog tables:

pg_catalog	schema holding all the catalog tables
pg_stats	table and column statistics, like cardinality
pg_attribute	row for every table column
pg_class	every table, index, view, materialized view, forgien table
pg_type	data types, built in and custom

Exploring system tables with `ECHO_HIDDEN` or `-E`

Sometimes navigating these tables and views can be confusing and require browsing through a mix of docs and source code. If you want to have some fun exploring how the catalog is connected, you can connect to your database with `-E` argument to psql (or do `\set ECHO_HIDDEN` on if you're already connected). Postgres will echo each psql the command that's run with SQL so you can grab the underlying SQL and edit from there.

Elizabeth Christensen
Nov 7, 2025 • 9 min read
More by this author

LATEST ARTICLES

Postgres Internals Hiding in Plain Sight

PostGIS Performance: Improve Bounding Boxes with Decompose and Subdivide

Temporal Joins

PostGIS Performance: pg_stat_statements and Postgres tuning

Is Postgres Read Heavy or Write Heavy? (And Why You Should Care)

For example, echoing **`\dt+`** will show me a query and the results.

```
SELECT n.nspname as "Schema",
  c.relname as "Name",
  CASE c.relkind WHEN 'r' THEN 'table' WHEN 'v' THEN 'view' WHEN 'm' THEN 'materialized view' WHEN 'i' THEN 'index' WH
  pg_catalog.pg_get_userbyid(c.relowner) as "Owner",
  CASE c.relpersistence WHEN 'p' THEN 'permanent' WHEN 't' THEN 'temporary' WHEN 'u' THEN 'unlogged' END as "Persisten
  am.amname as "Access method",
  pg_catalog.pg_size_pretty(pg_catalog.pg_table_size(c.oid)) as "Size",
  pg_catalog.obj_description(c.oid, 'pg_class') as "Description"
FROM pg_catalog.pg_class c
     LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
     LEFT JOIN pg_catalog.pg_am am ON am.oid = c.relam
WHERE c.relkind IN ('r','p','')
      AND n.nspname <> 'pg_catalog'
      AND n.nspname !~ '^pg_toast'
      AND n.nspname <> 'information_schema'
  AND pg_catalog.pg_table_is_visible(c.oid)
ORDER BY 1,2;

List of tables
-[ RECORD 1 ]-+--------------
Schema        | public
Name          | articles
Type          | table
Owner         | dba
Persistence   | permanent
Access method | heap
Size          | 16 kB
Description
```

Getting to Postgres internals

1. The easiest way to see internals is to start with the psql `\d` commands
2. The prebuilt views like `pg_stat_activity`, `pg_stat_statements`, `pg_locks`, and `pg_stat_user_indexes` are ready to go for easy querying and searching.
3. Going a step deeper, you can access the underlying internal Postgres tables, housed in the pg_catalog schema. `-E` `echo_hidden` can help you see the tables involved if you echo psql commands.

Crunchy Postgres
Crunchy Postgres for Kubernetes
Crunchy Bridge
Crunchy Certified PostgreSQL
Crunchy PostgreSQL for Cloud Fou...
Crunchy MLS PostgreSQL
Crunchy Spatial

Enterprise PostgreSQL Sup...
Ansible
Red Hat Partner
Trusted PostgreSQL
Crunchy Data Subscription

Migrate to Crunchy Data

Migrate from RDS
Migrate from Heroku

Customer Portal
Documentation
Postgres Tutorials
Crunchy Bridge Walkthrough
Postgres Operator Walkthro...
Blog
Events

About
Team
News
Contact Us
Newsletter
Branding
Privacy Notice – Recently Upda...
Security
Cookies Settings

Subscribe to the Crunchy Data Newsletter
and receive Postgres content every month.

Enter your email
Subscribe