Beginner's Guide to PostgreSQL Hacking

Suraj Kharage

Senior Staff SDE I, EnterpriseDB

25 June 2026

What Is "PostgreSQL Hacking"?

✗ NOT This

- Breaking into databases
- Security exploits or attacks
- Bypassing authentication
- Anything illegal or unethical

vs

✓ This

- Contributing patches to PostgreSQL
- Fixing bugs in the C source code
- Adding new features to the database
- Improving docs, tests, performance
- Understanding database internals deeply

Why Hack on PostgreSQL?

- Postgres is world's most advanced open source database system
- Relational database concepts are time tested and evolved over a period of time
- Easy to understand, well commented, highly organised source code
- Highly extensible system
- BSD licensed code, giving a lot of flexibility
- Postgres internals skills are always high in demand
- Become a better programmer

Agenda

01

Building from Source

git clone → configure → make → run

02

PostgreSQL Architecture

Postmaster, backends, WAL, memory

03

Source Code Tour

1.4M lines — learn to navigate fast

04

Making Your First Patch

find it • read it • fix it • test it

05

Submitting to the Community

Commitfest, mailing list, review cycle

Building from Source

git clone → configure → make → run

Prerequisites

Required

gcc or clang (C99+)
GNU make 3.80+
bison & flex (lexer & parser generators)
readline (psql line editing)
zlib
icu

Recommended (enables key features)

libssl / OpenSSL (SSL connections)
libpam (PAM authentication)
Python 3 (PL/Python extension)
Perl (PL/Perl extension)
libxml2 (XML data type)
libxslt (XSLT functions)
…

```
Quick install (Debian/Ubuntu):  sudo apt install build-essential bison flex libreadline-dev zlib1g-dev
```

Getting the Source Code

Clone the official mirror on GitHub :

```
# Official mirror
git clone https://github.com/postgres/postgres.git

cd postgres

# Check branch
git branch

# Check out a specific stable release (optional)
git checkout REL_16_STABLE
```

Pro tips

Always work on master/main for new patches • git log --oneline -20 to see recent commits

Configure for Development

Use these flags for a developer build — they add assertions, debugging info, and disable optimization:

```
# Minimal development-focused configure
./configure \
 --prefix=/usr/local/pgsql-dev \
 --enable-debug \
 --enable-cassert \
 --enable-depend \
 CFLAGS='-O0'


# If you need SSL or other extensions add:
#   --with-openssl --with-libxml --with-python
```

Key flags explained:

--enable-debug → compile with -g for gdb symbols --enable-cassert → turn on hundreds of internal assertions -O0 → disable optimisation so stepping in gdb is sane

Build & Install

```
# Build the source code
  make

# Install to the --prefix directory
  make install

# Also build and install contrib modules (highly recommended)
  cd contrib && make && make install
```

What a successful build looks like:

Typical first build: ~2-3 minutes on a 4-core machine.
Incremental rebuilds after a small change: 10-30 seconds.

Initialize & Run Your Dev Instance

```
export PATH=/usr/local/pgsql-dev/bin:$PATH

# Initialize a new data directory
initdb -D ~/pgdata-dev

# Start the server
pg_ctl -D ~/pgdata-dev -l ~/pgdata-dev/logfile start

# Connect with psql
psql -d postgres

# Stop the server when done
pg_ctl -D ~/pgdata-dev stop
```

Quick verify it's running

psql -c 'SELECT version();'

postgresql.conf tweaks

logging_collector = on
log_statement = 'all'
log_min_duration_statement = 0

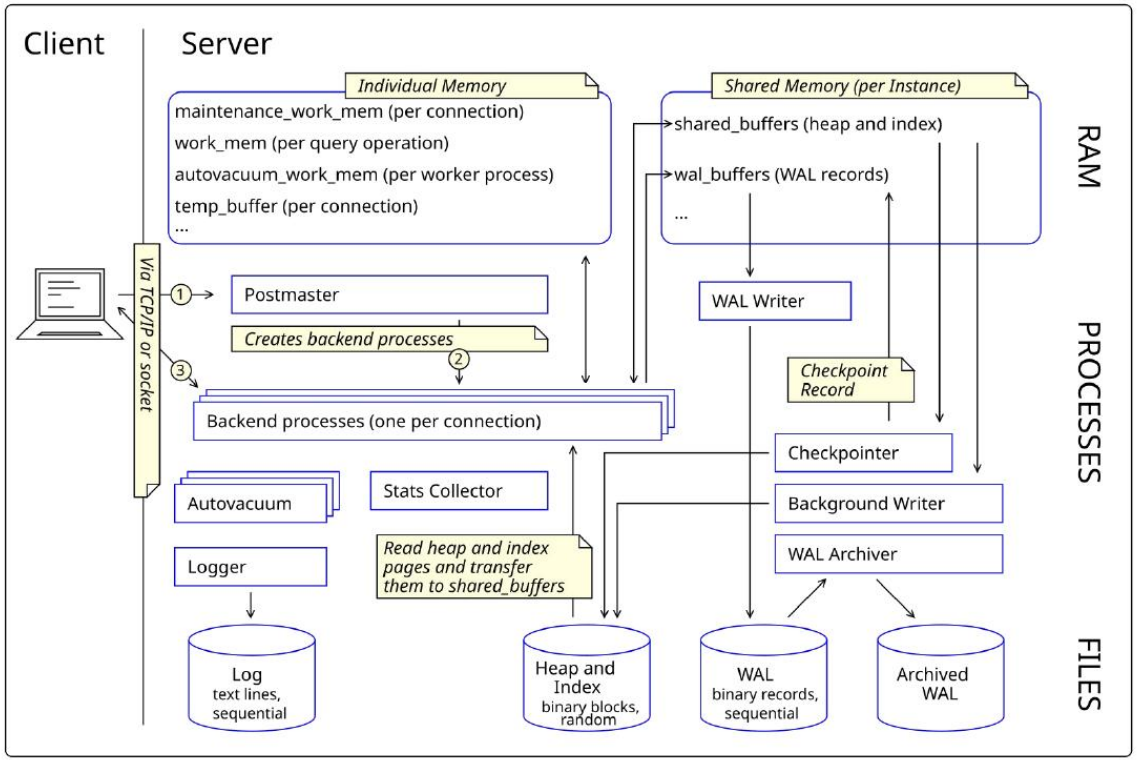
Reload without restart

pg_ctl reload -D ~/pgdata-dev

PostgreSQL Architecture

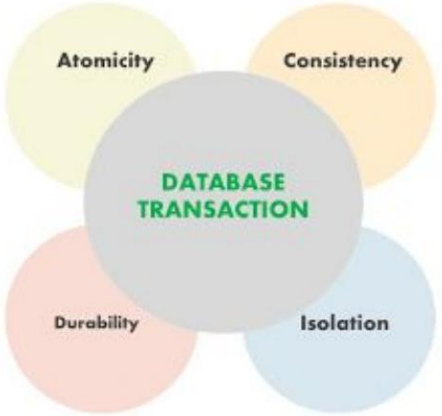
Postmaster • backends • WAL • shared memory

PostgreSQL Architecture Overview



Transaction Management

- Transactions are at the core of any database system
- Postgres supports transactions and subtransactions, and
  a variety of serializability modes
- MVCC allows readers to read without waiting for writers to finish
- Write-ahead-logs (WAL) for transaction durability
- **src/backend/access/transam/**



Memory Management

- Postgres provides robust infrastructure for memory management
- Allocations are tracked via `MemoryContext`, which are hierarchical in nature
  - `palloc/palloc0`
  - `MemoryContextAlloc()`
- Allocated memory can be freed in a single shot by deleting or resetting the `MemoryContext`
  - `MemoryContextReset()`
  - `MemoryContextDelete()`
- Or it can be explicitly freed
  - `pfree()`

System Caches

- Postgres maintains a cache of system objects for quick and fast lookup
  - No need to read the catalog from storage everytime
  - Cache in the backend local memory
  - Invalidated when system objects are modified

  src/backend/utils/cache/catcache.c
- lsyscache
  - A set of convenience routines to lookup objects
  - E.g. get_relname_relid(), get_rel_relkind(), get_rel_relispartition(), get_func_name()

  src/backend/utils/cache/lsyscache.c

Error Handling

- User level errors are reported via `ereport()`.
  - Allows additional information to be included (SQLSTATE, detail, hint, backtrace etc)
- Implemented via `longjmp`.
  - Control is passed back to the top level error handler
  - Transaction is aborted, resources are released (locks, memory, buffers, open files etc) and backend is prepared to handle next set of commands (usually ROLLBACK first)
- Developers can write their own exception handlers via `PG_TRY/PG_CATCH` blocks.
- Assertions to detect unexpected states
  - Postgres uses assertions freely (enabled with `–enable-cassert` flag)
  - Turned off in production builds

Source Code Tour

The PostgreSQL Codebase — By the Numbers

~1.4M Lines of C code	30+ Years of development	200+ Active contributors	1 Major releases / year
1,500+ Commits per year	500+ Regression tests	~300 Source .c files	4 Minor releases / year

Repository Layout at a Glance

`src/`	All source code — backend, interfaces, tools, libraries
`src/backend/`	The database server itself (where you'll spend most of your time)
`src/include/`	All header files (.h) shared across the codebase
`src/bin/`	Client programs: psql, pg_dump, pg_ctl, initdb …
`src/pl/`	Procedural languages: PL/pgSQL, PL/Python, PL/Perl
`contrib/`	Optional extensions shipped with PostgreSQL (pg_stat_statements, etc.)
`doc/`	Documentation source (SGML format)
`src/interfaces/`	Client library drivers (libpq for C, ecpg)

The Heart: src/backend/ Subdirectories

`parser/`	Lexer (scan.l), parser (gram.y), raw parse tree	`storage/`	Buffer manager, file manager, freespace, page layout
`nodes/`	Node definitions and copy/equal/out infrastructure	`catalog/`	System catalog tables (pg_class, pg_attribute…), cache
`optimizer/`	Query planner: path generation, cost estimation, join ordering	`replication/`	Streaming replication, logical replication infrastructure
`executor/`	Runtime execution of plan nodes (SeqScan, Sort, HashJoin...)	`commands/`	SQL DDL: CREATE TABLE, ALTER, COPY, VACUUM, etc.
`access/`	Table & index access methods: heap, B-tree, GIN, GiST, BRIN	`utils/`	Error reporting (elog), memory contexts, hash tables, sorts

The Life of a SQL Query

1. Client	2. Parser	3. Analyser	4. Rewriter	5. Planner	6. Executor	7. Results
libpq sends query string over the wire protocol	Tokenise → parse → raw AST	Resolve names, types → Query node	Expand rules & views	Cost paths → cheapest PlannedStmt	Pull tuples node by node	Send rows back to client via protocol

```
Try it yourself: SET enable_seqscan = off;  EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM pg_class
WHERE relname = 'orders';
```

Key source files to trace a query:

```
tcop/postgres.c  (exec_simple_query)  →  parser/analyze.c  (transformStmt)  →  optimizer/
planner.c  (standard_planner)  →  executor/execMain.c  (ExecutorRun)
```

Start here: src/backend/tcop/postgres.c — exec_simple_query() is the entry point for every interactive SQL statement

Finding Your Way Around the Codebase

```
# You can use Emacs, gvim, Visual Studio editor.

# ctags — jump to definitions in vim/emacs
ctags -R src/

# In vim: Ctrl+] to jump, Ctrl+t to come back

# cscope — find all callers of a function
cscope -b -R -s src/

# grep with context — find where ExecSeqScan is called
grep -rn 'ExecSeqScan' src/backend/

# Find which file defines a struct
grep -rn 'typedef struct SeqScanState' src/include/

#Debugger
gdb, lldb
```

Debugging with GDB

```
# Find your backend PID
SELECT pg_backend_pid();  -- in psql

# Attach GDB to the running backend
gdb -p <pid>


# Useful GDB commands
(gdb) break ExecSeqScan     # set breakpoint
(gdb) next                  # step over a line
(gdb) step                  # step into a code
(gdb) continue              # resume execution
(gdb) pprint nodeToString(plan)  # pretty-print a Node
(gdb) bt                    # backtrace


# Attach core dump
gdb postgres -c <core_file>

# See the backtrace
(gdb) bt
```

Log-based debugging

Add elog(LOG, "value = %d", x);
then watch PGDATA/logfile.
Use DEBUG1..DEBUG5 levels.
Set log_min_messages=debug1 in postgresql.conf.

Useful trick

\set VERBOSITY verbose

```
ERROR: 42703: column "usr_id" does not exist
LINE 1: SELECT * FROM users WHERE usr_id = 1;
                                      ^
LOCATION: errorMissingColumn, parse_relation.c:3594
```

Making Your First Patch

find it • read it • fix it • test it • format it

Finding Something to Work On

Documentation

Easiest starting point — typos, unclear explanations, missing examples in the official docs. Every patch counts.

Fix bugs

plpgsql-bugs — Fix a small bug with a confirmed reproducer.

TODO list

PostgreSQL TODO wiki page list known wanted improvements.

Error messages

Improve vague error messages to be more helpful: better wording, add HINT or DETAIL fields.

Tests

Add regression tests for untested code paths. Improve pg_regress test coverage in src/test/regress/.

Review patches

plpgsql-hackers - Pick any feature/email that you are interested and review it.

Reading & Understanding the Code

- **Before writing a single line — read the area thoroughly**
  - Find all callers and callees of the function you want to change
  - Read the surrounding comments — PostgreSQL has excellent inline documentation
  - Check git log -p <file> to understand past changes and their rationale
- **Use EXPLAIN to verify your mental model**
  - EXPLAIN (VERBOSE, ANALYZE, BUFFERS) your target queries
  - Match what you see in the plan to the executor node source files
- **Write a test case first**
  - Craft a SQL test that demonstrates the current bug or missing feature
  - Place it in src/test/regress/sql/ — run make check and watch it fail
  - Then write the fix until the test passes
- **Ask on the mailing list before deep work**
  - Send a short proposal to pgsql-hackers — avoid duplicating ongoing work
  - Committers are generally helpful in guiding approach

PostgreSQL Coding Style

```
/*
 * get_role_oid - Given a role name, look up the role's OID.
 *
 * If missing_ok is false, throw an error if role name not found.
 * If true, just return InvalidOid.
 */
Oid
get_role_oid(const char *rolname, bool missing_ok)
{
    Oid       oid;

    oid = GetSysCacheOid1(AUTHNAME, Anum_pg_authid_oid,
                          CStringGetDatum(rolname));

    if (!OidIsValid(oid) && !missing_ok)
        ereport(ERROR,
                (errcode(ERRCODE_UNDEFINED_OBJECT),
                 errmsg("role \"%s\" does not exist",rolname)));

    return oid;
}
```

Key rules:

- ✓ Tabs for indentation (not spaces)
- ✓ Column limit: 80 chars per line
- ✓ Opening brace on new line for functions
- ✓ No C99 // comments — use /* ... */
- ✓ All variables declared at block top
- ✓ Run pgindent before submitting

Testing Your Changes

```
# Core regression suite (~230 tests, ~5 minutes)
make check
make check-world

# Test a contrib module (e.g. pg_stat_statements)
cd contrib/pg_stat_statements && make check

# Full install-check (uses an already-running server)
make installcheck PGPORT=5433

# In case of failures, check
cat src/test/regress/regressions.diffs

# TAP test cases for complicated scenarios.
# enable those using —enable-tap-tests
# written in perl
e.g. src/test/recovery/t/
```

Where test files live

sql/ — input SQL
expected/ — expected output
results/ — actual output
regressions.diffs — what changed

Adding a new test

1. Create sql/mytest.sql
2. Run once to capture output:
3. Copy results/ → expected/
4. Include in schedule

Making a Clean Patch

Reviewers want to see ✓

- ✓ One logical change per patch
- ✓ Commits on top of current master
- ✓ Regression tests that exercise the change
- ✓ Updated documentation where applicable
- ✓ pgindent run on modified files
- ✓ Clear commit message explaining WHY

Common mistakes ✗

- ✗ Mixing unrelated changes in one diff
- ✗ Patch against stale 6-month-old checkout
- ✗ No tests — trust me it works
- ✗ Docs unchanged — reviewer has to guess
- ✗ Inconsistent whitespace / tab vs space
- ✗ Commit message: 'fix bug'

Creating the Patch File

Always work on a master/feature branch:

```
# Create a feature branch from current master
git checkout -b fix-explain-analyze-timing

# Make your changes, then commit
git add src/backend/commands/explain.c
git commit -m 'Fix EXPLAIN ANALYZE timing for CTEs'

# Create a patch file ready for email submission
git format-patch -v 6 HEAD~1

# Apply the patch
git apply <patch_name>
git am <patch_name>
```

Good commit message anatomy

First line: header ≤72 chars.
Body: explain WHY. Reference bug IDs, upstream commit etc.

Rebase before submitting

Submitting to the Community

Commitfest • pgsql-hackers • review cycle

The Commitfest — How PG Development Works

- **Commitfest = the submission + review window**
  - Five commitfests per year: July, September, November, January, March
  - Patches must be registered at commitfest.postgresql.org
  - Each CF lasts roughly one month — community reviews registered patches
- **PostgreSQL releases on a ~1 year cadence**
  - Feature freeze in April, beta in June, release in September/October
  - Minor/security releases every ~3 months
- **cfbot (cfbot.cputube.org) auto-tests every patch**
  - Applies your patch, builds, runs make check — watch for red cfbot flags

Mailing List Etiquette — pgsql-hackers

Sending your patch

Subject: [PATCH] Fix EXPLAIN ANALYZE timing for CTEs
Describe the problem, your approach, and testing done.
Attach the .patch file — do NOT paste inline.

Replying to review

Always reply-all on the same thread.
Address every point raised by the reviewer.
Be patient — reviewers are volunteers.
Send a new patch as an attachment.

DOs

- Use plain text email (no HTML)
- CC the reviewer on replies
- Quote only the relevant part
- Reference prior thread links

DON'Ts

- Don't top-post
- Don't attach the whole mailing list archive
- Don't send MIME HTML
- Don't repeat the whole patch in the email body
- Don't go dark for weeks without a status update

Subscribe: www.postgresql.org/list/pgsql-hackers/ — Low volume is fine; search archives first before asking.

Submitting Your Patch — Step by Step

1

Send to pgsql-hackers

Email with subject [PATCH] <short description>. Attach the .patch file. Explain the problem, your fix, and how you tested it.

2

Register on commitfest.postgresql.org

Log in with your postgresql.org account (free). Add a new entry for your patch with the mailing list thread URL.

3

Watch cfbot

cfbot.cputube.org auto-applies every registered patch. Fix any build failures or test failures it reports within a few days.

4

Respond to review feedback

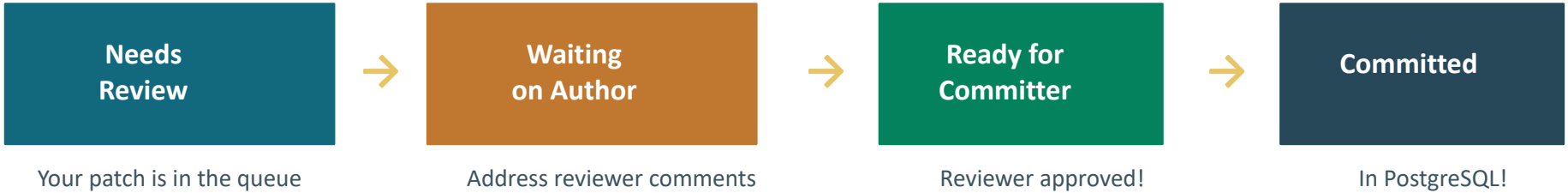
Reply on the same thread. Send v2 with changes. Update the commitfest entry to 'Needs Review' again after respin.

5

Repeat until committed

PostgreSQL moves slowly — quality over speed. A patch may go through 3-5 rounds of review. Persistence pays off.

What Happens During Review



- **What reviewers check:**
  - Correctness — does the logic work for all edge cases?
  - Style — does it match PostgreSQL conventions?
  - Tests — is coverage adequate?
  - Documentation — is user-visible behaviour documented?
  - Performance — measured impact for hot code paths?

Tips for Getting Your Patch Accepted

[Pkg] Keep patches small

A 50-line patch gets reviewed in days. A 2,000-line patch sits for months. Split complex features into reviewable chunks.

[Fast] Respond quickly

When a reviewer asks questions, reply within a few days. Patches with fast turnaround get committed faster.

[Docs] Write great docs

Write the docs change FIRST. If you can't explain it clearly, the design may need rethinking.

[Test] Over-test

More regression tests = faster review. Test null inputs, error paths, boundary values, not just the happy path.

[Team] Review others' patches

Reviewing other patches builds relationships and goodwill. You learn the codebase deeply and committers notice active reviewers.

[Time] Be patient & persistent

Don't get discouraged — keep rebasing, keep responding, keep improving.

Thank You!