# Learning Postgres the Hard Way

## Build Extensions to Explore the Engine's Internals

A (hands-on) journey from SQL to C, from functions to kernel hooks

# About Me

## Why I talk about this stuff?

- Developer by Profession for over 35 years
- Database Engines Nerd by Heart
- Running the SQL Server and PostgreSQL Meetups in Hamburg
- Founder of PSG Projekt Service GmbH (since 1994)

# Our Journey Today

- **The Why:** Understanding Postgres Extensibility
- **Level 1:** SQL-Only Extensions (The Gateway)
- **Level 2:** Basic C Functions (The Performance Leap)
- **Level 3:** Custom Data Types (Teaching New Math)
- **Level 4:** Hooks (Intercepting the Engine)
- **Level 4.1:** Real Auditing with SPI (Writing Back)
- **Level 5:** Query Store lite (Shared Memory)
- [**Level 6:** Rust with pgrx (Modern DX + Safety)]

Each level builds on the previous, revealing more internals

# Why Extensions?

Postgres is designed from the ground up to be extensible

# The Philosophy

## WITHOUT EXTENSIONS

- Fork the core
- Maintain patches
- Upgrade nightmares
- No sharing

## WITH EXTENSIONS

- Clean API
- Easy upgrades
- Share on PGXN
- Mix & match

# Real-World Examples

- **PostGIS:** Spatial database in an extension
- **pg_stat_statements:** Query performance tracking
- **Citus:** Distributed Postgres
- **TimescaleDB:** Time-series optimization
- **pgvector:** Vector embeddings for AI/ML

These aren't plugins—they're first-class features!

# What Can Extensions Do?

- **New Functions:** Custom business logic
- **New Data Types:** Complex numbers, JSON, geometry
- **New Operators:** @>, <<, custom indexes
- **Hooks:** Intercept queries, planning, execution
- **Background Workers:** Continuous processing
- **Foreign Data Wrappers:** Query external systems

# Prerequisites & Environment

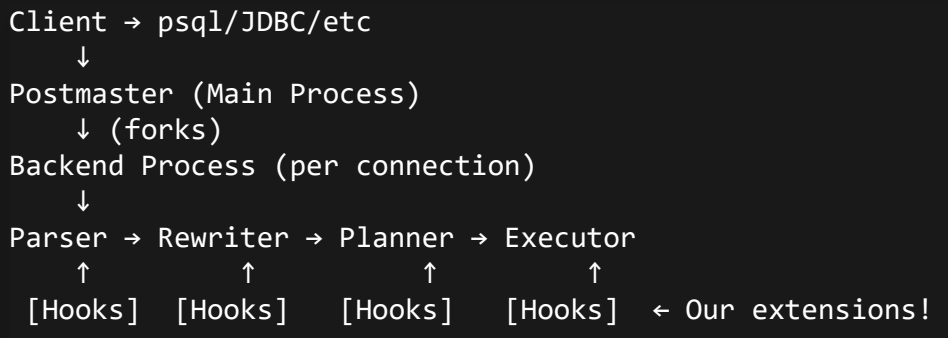
What do you need to build C extensions?

# Required Tools

- **C Compiler:** `gcc` or `clang`
- **Postgres Headers:** `postgresql-server-dev-xx`
- **Build Tool:** GNU `make`
- **PGXS:** Postgres Extension Build Infrastructure (included)

**Dev Container Ready:** All tools pre-installed in our environment

# Postgres Architecture




Each phase has hooks we can intercept

# The Build Process

1. **Write Code:** `.c` files with Postgres API calls
2. **Define Metadata:** `.control` file (name, version, etc.)
3. **SQL Wrapper:** `.sql` script to expose functions
4. **Compile:** `make` → generates `.so` (Shared Object)
5. **Install:** `make install` → copies to PG directories

# Extension File Structure

```
my_extension/
├── Makefile              # PGXS build configuration
├── my_extension.c        # C source code
├── my_extension.control  # Extension metadata
└── my_extension--1.0.sql # SQL installation script
```

Each file has a specific purpose in the extension ecosystem

# Where Extensions Live

```
# After 'make install', files go to:

$(pg_config --sharedir)/extension/
    pg_hello_sql.control
    pg_hello_sql--1.0.sql

$(pg_config --pkglibdir)/
    pg_hello_c.so          # Compiled shared library

# View directories:
pg_config --sharedir
pg_config --pkglibdir
```

PGXS handles all paths automatically!

# The Deployment

## Once installed, it's just SQL:

```
-- Load the extension into the current database
CREATE EXTENSION my_extension;

-- Use it immediately!
SELECT my_function(42);

-- Update to a new version
ALTER EXTENSION my_extension UPDATE TO '1.1';

-- Remove it cleanly
DROP EXTENSION my_extension;
```

All objects created by the extension are tracked and removed together

# Versioning & Upgrades

## Version Files:

```
my_ext--1.0.sql      # Initial
my_ext--1.0--1.1.sql # Upgrade
my_ext--1.1--2.0.sql # Upgrade
```

## Usage:

```
-- Install v1.0
CREATE EXTENSION my_ext;

-- Upgrade to 1.1
ALTER EXTENSION my_ext
  UPDATE TO '1.1';
```

Postgres automatically finds and applies upgrade scripts

# Level 1: SQL-Only Extensions

## The Gateway Drug

Let's start with the simplest possible extension

# Why Start with SQL-Only?

- **No compilation required** - just SQL scripts
- **Perfect for:** Packaging utility functions for reuse
- **Easy distribution:** Copy 2-3 text files
- **Learn the mechanics:** Control files, versioning, dependencies

Many production extensions start here before evolving to C!

# The Anatomy

## 1. Control File (`pg_hello_sql.control`):

```
comment = 'Simple SQL-only extension'
default_version = '1.0'
relocatable = true
```

## 2. SQL Script (`pg_hello_sql--1.0.sql`):

```
CREATE OR REPLACE FUNCTION hello_sql()
RETURNS text AS $$
    SELECT 'Hello from SQL-only extension!';
$$ LANGUAGE sql;
```

# The Makefile (PGXS Magic)

```
EXTENSION = pg_hello_sql
DATA = pg_hello_sql--1.0.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

**PGXS** is Postgres's built-in extension build infrastructure

It knows where to install files for your Postgres version

# Installation & Usage

```
# Install to Postgres directories
make install

# Connect to Postgres
psql -U postgres

# Create the extension
CREATE EXTENSION pg_hello_sql;

# Use it!
SELECT hello_sql();
-- Returns: "Hello from SQL-only extension!"
```

# What Just Happened?

1. `make install` copied files to `$SHAREDIR/extension/`
2. `CREATE EXTENSION` found the `.control` file
3. Postgres read the `default_version` (1.0)
4. It executed `pg_hello_sql--1.0.sql`
5. Tracked all created objects in `pg_extension` catalog

**Clean uninstall:** `DROP EXTENSION pg_hello_sql;` removes everything

# Demo Time!

Let's see it live

# Level 2: Basic C Functions

## The Performance Leap

When SQL isn't enough, go native!

# Why C?

## PERFORMANCE

- No SQL parsing overhead
- Direct memory access
- Compiler optimizations
- 10-100x faster for compute

## CAPABILITIES

- System calls
- External libraries
- Complex algorithms
- Hardware features

# The Essential Boilerplate

```c
#include "postgres.h"  // Core Postgres types and macros
#include "fmgr.h"      // Function Manager interface

PG_MODULE_MAGIC;       // REQUIRED: Version check
```

`PG_MODULE_MAGIC` ensures ABI compatibility

Without it, Postgres refuses to load your extension!

# A Simple Function

```
PG_FUNCTION_INFO_V1(add_integers);

Datum
add_integers(PG_FUNCTION_ARGS)
{
    int32 arg1 = PG_GETARG_INT32(0);
    int32 arg2 = PG_GETARG_INT32(1);

    PG_RETURN_INT32(arg1 + arg2);
}
```

**Datum:** Postgres's universal "any type" container

# Understanding the Macros

| Macro | Purpose |
| --- | --- |
| `PG_FUNCTION_INFO_V1(name)` | Declares function metadata for Postgres |
| `PG_FUNCTION_ARGS` | Standard function signature (hides complexity) |
| `PG_GETARG_INT32(n)` | Extract argument N as int32 |
| `PG_RETURN_INT32(x)` | Return int32 value to Postgres |

Macros handle version differences and type conversions

# The SQL Wrapper

We still need SQL to expose the C function:

```
-- pg_hello_c--1.0.sql
CREATE FUNCTION add_integers(integer, integer)
    RETURNS integer
AS 'MODULE_PATHNAME', 'add_integers'
LANGUAGE C STRICT;
```

- `MODULE_PATHNAME` → auto-replaced with .so path
- `STRICT` → return NULL if any arg is NULL

# The Makefile (with C)

```
MODULES = pg_hello_c          # Compile pg_hello_c.c
EXTENSION = pg_hello_c
DATA = pg_hello_c--1.0.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

`MODULES` tells PGXS to compile the .c file into a .so

# Compilation Flow

```
pg_hello_c.c
    ↓  [gcc -c -fPIC ...]
pg_hello_c.o
    ↓  [gcc -shared ...]
pg_hello_c.so  ← Shared Library
    ↓  [make install]
/usr/lib/postgresql/16/lib/pg_hello_c.so
```

PGXS handles all compiler flags automatically

# Usage

```
# Compile and install
make install

# In Postgres
CREATE EXTENSION pg_hello_c;

SELECT add_integers(10, 20);
-- Returns: 30
```

The C function is now callable from any SQL query!

# Demo Time!

Let's compile and run native code

# Level 3: Custom Data Types

## Teaching Postgres New Math

Extending the type system itself!

# Why Custom Types?

- **Domain modeling:** Express business concepts natively
- **Performance:** Store data in optimized format
- **Type safety:** Let Postgres enforce rules
- **Operators:** Create custom @ << >> operators
- **Indexes:** Build specialized index methods

Examples: `point`, `inet`, `json`, `cube`, `geometry`

# Our Example: Complex Numbers

Mathematical complex numbers (a + bi)

```
Complex Number:  3 + 4i

Representation:  (3, 4)
                  ↑  ↑
                real imaginary
```

We'll store them efficiently and make them queryable

# The C Structure

```c
typedef struct Complex {
    double x;  // Real part
    double y;  // Imaginary part
} Complex;
```

Just 16 bytes (2 × double precision floats)

Stored directly in table pages, no overhead!

# Required Functions

Every custom type needs at minimum:

| Function | Purpose | Example |
|---|---|---|
| **Input** | String → Internal format | "(3,4)" → Complex{3,4} |
| **Output** | Internal → String | Complex{3,4} → "(3,4)" |

Optional: Send/Receive (binary I/O for network protocol)

# Input Function (Parsing)

```c
PG_FUNCTION_INFO_V1(complex_in);


Datum
complex_in(PG_FUNCTION_ARGS)
{
    char *str = PG_GETARG_CSTRING(0);
    double x, y;
    Complex *result;


    // Parse "(x, y)" format
    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for complex: \"%s\"", str)));


    // Allocate memory in Postgres context
    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
```

**Key:** Use `palloc()` not `malloc()`!

# Output Function (Formatting)

```c
PG_FUNCTION_INFO_V1(complex_out);


Datum
complex_out(PG_FUNCTION_ARGS)
{
    Complex *complex = (Complex *) PG_GETARG_POINTER(0);
    char *result;


    // Format as string
    result = psprintf("(%g,%g)", complex->x, complex->y);


    PG_RETURN_CSTRING(result);
}
```

`psprintf()` is Postgres's safe sprintf

# SQL Type Registration

```sql
-- Declare the type exists
CREATE TYPE complex;

-- Register input/output functions
CREATE FUNCTION complex_in(cstring) RETURNS complex
    AS 'MODULE_PATHNAME' LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex) RETURNS cstring
    AS 'MODULE_PATHNAME' LANGUAGE C IMMUTABLE STRICT;

-- Define the type's properties
CREATE TYPE complex (
    internallength = 16,        -- sizeof(Complex)
    input = complex_in,
    output = complex_out,
    alignment = double          -- Memory alignment
);
```

# Using Our New Type

```
-- Create a table with our type!
CREATE TABLE signals (
    id serial PRIMARY KEY,
    frequency complex,
    amplitude complex
);

-- Insert complex numbers
INSERT INTO signals (frequency, amplitude) VALUES
    ('(440, 0)', '(1.0, 0.5)'),
    ('(880, 100)', '(0.8, 0.2)');

-- Query them
SELECT * FROM signals;
-- Returns: (440,0) and (880,100)
```

# What's Possible Next?

- **Operators:** `complex + complex`, `complex * complex`
- **Functions:** `abs(complex)`, `conjugate(complex)`
- **Aggregates:** `avg(complex)`
- **Casts:** `point::complex`
- **Indexes:** GiST/SP-GiST for spatial queries

PostGIS does this for 100+ geometric types!

# Demo Time!

Let's create complex numbers in Postgres

ACT III

# THE MAIN EVENT

## Intercepting the Engine

**Demo 04** — Hooks & the Executor

**Demo 05** — SPI & Query Audit

**Demo 06** — Shared Memory Query Store

*“This is where we go from extending Postgres… to becoming part of it”*

# The Query Execution Pipeline

```
Client SQL text
    ↓ Parser        →  raw parse tree
    ↓ Analyzer      →  Query tree (resolve names, OIDs, types)
    ↓ Rewriter      →  apply rules, views, RLS
    ↓ Planner       →  PlannedStmt (cost-optimized)
    ↓ Executor      →  result tuples   ← Demo 04 hooks here
```

By the time our hook fires at `ExecutorStart`, the query has been fully parsed, analyzed, and planned

Each stage has hookable entry points, not just the executor

# Hooks: Beautifully Simple, Brutally Powerful

The actual declaration from the Postgres source:

```
typedef void (*ExecutorStart_hook_type)(QueryDesc *queryDesc, int eflags);
extern PGDLLIMPORT ExecutorStart_hook_type ExecutorStart_hook;
```

**That's it.** A global function pointer. `NULL` by default.

No registration API. No event bus. No framework.
Just: assign your function to the pointer.

- Zero overhead when not installed, just a `NULL` check
- Chaining is cooperative, each hook passes control down the chain
- No thread-safety issue, Postgres is multi-process, not multi-threaded

*“The simplicity IS the feature”*

# Four Hooks, Four Phases

| Phase | Hook | Purpose |
|---|---|---|
| **Start** | `ExecutorStart_hook` | Creates executor state. Used by EXPLAIN. |
| **Run** | `ExecutorRun_hook` | Fetches tuples. Multiple calls for cursors. |
| **Finish** | `ExecutorFinish_hook` | Fires deferred triggers. |
| **End** | `ExecutorEnd_hook` | Teardown. `auto_explain` grabs timing here. |

For query timing: hook at **Start** (capture timestamp), hook at **End** (calculate duration)

# `_PG_init`: Install Your Hooks Here

```c
static ExecutorStart_hook_type prev_ExecutorStart = NULL;

void _PG_init(void)
{
    // Save whatever was there before
    prev_ExecutorStart = ExecutorStart_hook;
    // Install OUR hook
    ExecutorStart_hook = pg_query_logger_ExecutorStart;
}
```

- `_PG_init` is called after `dlopen` succeeds and `PG_MODULE_MAGIC` is verified
- `_PG_fini` almost never called in practice, shared libraries aren't dynamically unloaded

# The Chaining Contract: Never Break the Chain

```c
static void
pg_query_logger_ExecutorStart(QueryDesc *queryDesc, int eflags)
{
    // YOUR CODE runs here (before execution)
    if (queryDesc->sourceText)
        elog(LOG, "Query: %s", queryDesc->sourceText);

    // CRITICAL: ALWAYS chain to previous or standard!
    if (prev_ExecutorStart)
        prev_ExecutorStart(queryDesc, eflags);  // Chain!
    else
        standard_ExecutorStart(queryDesc, eflags);  // Default
}
```

Forgetting this ONE call silently breaks all other hooks!

# Inside `QueryDesc`: What We Can See

| Field | Contains |
|---|---|
| `sourceText` | Original SQL string as the user typed it |
| `plannedstmt` | Optimized execution plan tree |
| `params` | Bound parameter values |
| `plannedstmt->queryId` | Query fingerprint hash (normalized) |
| `totaltime` | Timing (if instrumentation active) |

`queryId`: same query, different literal parameters → **same ID**.
Foundation of `pg_stat_statements`.

# Two Ways to Load Extensions with Hooks

## LOAD (SESSION-LEVEL)

```
LOAD 'pg_query_logger';
```

- Quick testing
- No restart needed
- One connection only
- Lost on disconnect

## SHARED_PRELOAD_LIBRARIES

```
shared_preload_libraries =
  'pg_query_logger'
```

- All connections
- Persistent
- Requires restart

"For testing: `LOAD`. For production: `shared_preload_libraries`."

This distinction becomes **critical** for shared memory (Demo 06)

# 20+ Places to Hook Into Postgres

### Query Pipeline

- post_parse_analyze_hook
- planner_hook (pg_hint_plan)
- ExecutorStart/Run/Finish/End

### DDL

- ProcessUtility_hook

### Authentication

- ClientAuthentication_hook
- check_password_hook

### Access Control

- object_access_hook
- row_security_policy hook

### Infrastructure

- shmem_request_hook
- shmem_startup_hook
- emit_log_hook

# Demo 04: `pg_query_logger`

40 lines of C that intercept every query

- `make install` in `04-hooks/`
- `LOAD 'pg_query_logger'`
- Run some queries: `SELECT 42`, `UPDATE ...`
- Check the server logs → see every intercepted query
- Notice: DDL not captured (`ProcessUtility_hook` missing)
- Show the 37 lines of C doing the work

Let's intercept some queries

# Demo 05:
# The Engine Talking to Itself

## SPI: Server Programming Interface

`pg_query_logger` proved we can intercept queries.
Now: can we store the audit trail in a **TABLE**?

Welcome to SPI, the mechanism that powers every procedural language in Postgres

# SPI: Not a New Connection. A Nested One.

SPI lets C code running *inside* the backend open a nested execution context and run SQL

Powers: PL/pgSQL `EXECUTE`, PL/Python `plpy.execute()`, PL/Perl, C triggers

SPI is **NOT** a separate connection. It re-enters the executor recursively, in the **SAME process**, sharing the SAME memory, locks, and transaction state.

```
SPI_connect()            → push new execution context
SPI_execute_with_args()  → parse, plan, execute in current txn
SPI_finish()             → pop context, free SPI memory
                           (after this: all result pointers are DANGLING)
```

# The Infinite Loop That Would Crash Your Database

```
1. User runs: SELECT * FROM users
      ↓
2. ExecutorStart hook fires
      ↓
3. We run: INSERT INTO audit_log ... (via SPI)
      ↓
4. INSERT is also a query
      ↓
5. ExecutorStart hook fires AGAIN
      ↓
6. We run: INSERT INTO audit_log ... (via SPI)
      ↓
7. ↺ INFINITE RECURSION ↺
      ↓
Stack overflow → backend crash
```

# One Boolean Saves Everything

```c
static bool is_logging = false;  /* The guard */

static void pg_audit_ExecutorStart(QueryDesc *queryDesc, int eflags)

{

    if (is_logging) {

        /* We're in our own SPI call -- skip logging */

        if (prev_ExecutorStart)

            prev_ExecutorStart(queryDesc, eflags);

        else

            standard_ExecutorStart(queryDesc, eflags);

        return;

    }

    if (queryDesc->sourceText) {

        is_logging = true;     /* SET before SPI */

        /* ... SPI INSERT ... */

        is_logging = false;    /* CLEAR after SPI */

    }

    standard_ExecutorStart(queryDesc, eflags);

}
```

# What Actually Happens

```
Client: SELECT * FROM users;
  └─ Executor starts
       └─ Our hook fires
            └─ is_logging = false → proceed
            └─ SPI_connect()
            └─ SPI INSERT INTO audit_log (SPI call)
                 └─ Nested Executor starts
                      └─ Our hook fires again
                           └─ is_logging = true → skip!
                      └─ INSERT runs normally
                 └─ SPI done
            └─ SPI_finish()
       └─ Original SELECT runs
  └─ Results returned to client
```

Every PL/pgSQL function works this way too. SPI all the way down.

# Parameterized Queries: Safe Even in C

```c
char *query = "INSERT INTO public.audit_log(query) VALUES($1)";
Oid   argtypes[1] = { TEXTOID };
Datum values[1]   = { CStringGetTextDatum(queryDesc->sourceText) };
char  nulls[1]    = { ' ' };   /* ' ' = NOT NULL */
SPI_execute_with_args(query, 1, argtypes, values, nulls, false, 0);
```

The query text is passed as a parameter **value**,
never spliced into SQL

Same mechanism as `PREPARE`/`EXECUTE`. Even C code gets first-class parameterization.

Never build SQL strings with `sprintf`. Even in C internals.

# Error Handling: PG_TRY / PG_CATCH

```
is_logging = true;
PG_TRY();
{
    SPI_connect();
    /* ... SPI operations ... */
    SPI_finish();
}
PG_CATCH();
{
    is_logging = false;  /* MUST reset or audit locks up forever */
    PG_RE_THROW();       /* Propagate the error */
}
PG_END_TRY();
is_logging = false;
```

PG_TRY sets up a sigjmp_buf for Postgres's longjmp-based error handling. If you don't reset is_logging in PG_CATCH, after any error your extension stops logging **permanently**

# Your Audit Log Can Lie

The audit `INSERT` is in the **SAME transaction** as the user's query

```
BEGIN;
UPDATE accounts SET balance = balance - 1000000;  -- logged
-- realizes it was wrong
ROLLBACK;  -- The audit entry also rolls back!
```

The `UPDATE` was logged. The `ROLLBACK` removes the audit entry.

This is different from `log_statement = 'all'` which writes to the OS log file **regardless of transaction outcome**.

# The Observer Effect

Querying `audit_log`... gets logged.

```
SELECT * FROM audit_log;
-- This SELECT is itself intercepted and logged!
-- Now audit_log has one more row...
-- SELECT * FROM audit_log again?
-- One more row!
```

Every observation changes what you're observing

*"The measuring instrument affects the measurement"*

Production solution: filter audit queries by `application_name` or `current_user` in the hook!? Kind of messy?

# Demo 05: `pg_audit`

- `CREATE EXTENSION pg_audit;` — creates the `audit_log` table
- `LOAD 'pg_audit';`
- Run queries: `SELECT 100`, `UPDATE`, `DELETE`
- `SELECT * FROM audit_log` — see the entries
- Show the observer effect live
- `BEGIN; DELETE everything; ROLLBACK` → no audit entry
- `\sf` a PL/pgSQL function — SPI is exactly how PL/pgSQL works

Let's audit some queries.

# Demo 06: Shared Memory

## How Sessions Actually Share Data

`pg_query_store`: a simplified `pg_stat_statements` built from scratch

Everything so far lived in ONE backend (process!).
Now: all backends, all connections, simultaneously.

# The Fundamental Truth: Postgres Is Multi-Process

PostgreSQL is **NOT multi-threaded.** The postmaster forks a new backend for each connection. Each backend has its own **private address space**.

```
Postmaster (supervisor)
    ├─ Backend 1 (Connection 1) — private heap
    ├─ Backend 2 (Connection 2) — private heap
    ├─ Backend 3 (Connection 3) — private heap
    └─ ... up to max_connections
```

A global variable in your extension lives in **ONE** backend's heap. Other backends can't see it.

*"If 5 pgbench connections are running queries, how do they contribute to the same counters?"*

Answer: Shared Memory

# Why 'LOAD' Is Not Enough

LOAD runs in a single backend **AFTER** the postmaster has already started

Shared memory allocation happens **DURING** postmaster startup, before any backends exist

```
void _PG_init(void)
{
    if (!process_shared_preload_libraries_in_progress)
        return;  /* We're in a backend -- shmem already allocated */
    /* ... register hooks ... */
}
```

THIS is why `pg_stat_statements` requires a server restart.

# The Shared Memory Lifecycle

```
1. Postmaster starts
        ↓ reads shared_preload_libraries
        ↓ calls _PG_init() for each extension
2. shmem_request_hook fires
        ↓ RequestAddinShmemSpace(bytes)
        ↓ RequestNamedLWLockTranche(name, count)
3. Postmaster allocates all shared memory
4. shmem_startup_hook fires
        ↓ ShmemInitStruct() — carve out named region
5. Backends forked (fork())
        ↓ INHERIT shared memory mapping
        ↓ Pointer is valid in ALL backends
6. Server shutdown → shared memory released
```

Every backend inherits the mapping via `fork()`. The address is valid in **ALL** processes.

# Who Initializes the Shared Memory?

```
LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
shared_state = ShmemInitStruct("pg_query_store",
                               sizeof(QueryStoreSharedState),
                               &found);
if (!found) {
    /* First backend to get here -- do initialization */
    memset(shared_state, 0, sizeof(QueryStoreSharedState));
    shared_state->lock =
        &(GetNamedLWLockTranche("pg_query_store"))->lock;
}
/* else: already initialized -- just use the existing pointer */
LWLockRelease(AddinShmemInitLock);
```

`ShmemInitStruct` does a name lookup. First call: allocates + `found=false`. Subsequent calls: returns existing pointer + `found=true`. The `found` flag elegantly solves "who initializes" without any leader election protocol. You cannot `palloc` inside shared memory, `palloc` is backend-local. Shared data must be flat and fixed-size.

The `found` flag elegantly solves "who initializes" without any leader election protocol

# LWLock: The Shared Memory Workhorse

| Property | LWLock | Heavyweight Lock |
|---|---|---|
| **Speed** | Fast (spin-then-sleep) | Slower (deadlock detection) |
| **Modes** | `LW_SHARED`, `LW_EXCLUSIVE` | Many modes (AccessShare, RowEx…) |
| **Use case** | Shared memory structs | Database objects (tables, rows) |

```
LWLockAcquire(shared_state->lock, LW_EXCLUSIVE);
/* modify shared data */
LWLockRelease(shared_state->lock);
```

NEVER hold an LWLock during I/O or SPI calls. Every other backend accessing that lock stalls.

# `queryId`: The Query Fingerprint

When `compute_query_id = on`, Postgres assigns a hash that normalizes away literal constants

```
-- These get the SAME queryId:
SELECT * FROM orders WHERE id = 1;
SELECT * FROM orders WHERE id = 42;
SELECT * FROM orders WHERE id = 999;

-- This gets a DIFFERENT queryId:
SELECT * FROM orders WHERE customer = 'Alice';
```

Foundation of `pg_stat_statements`, aggregate ALL variants of the same query pattern

In `pg_query_store`: `queryDesc->plannedstmt->queryId`

# `BufferUsage`: Is Your Cache Working?

`pgBufferUsage` is a per-backend global the buffer manager increments on every buffer access. Snapshot before/after execution isolates the query's contribution.

"Same technique as EXPLAIN (BUFFERS)"

| Metric | Means |
|---|---|
| `shared_blks_hit` | Found in shared buffer cache (fast!) |
| `shared_blks_read` | Had to read from disk (slow) |
| `shared_blks_written` | Wrote to shared buffers |

High `shared_blks_hit / (hit + read)` ratio = good. Low ratio = increase `shared_buffers` or fix the query.

# The Pattern: Copy–Release–Write

The critical performance insight for shared memory + I/O:

```
/* CRITICAL SECTION: microseconds */
LWLockAcquire(lock, LW_EXCLUSIVE);
memcpy(local_stats, shared_state->stats, ...);   /* copy out */
memset(shared_state->stats, 0, ...);             /* reset */
LWLockRelease(lock);

/* SLOW SECTION: milliseconds -- no lock held */
SPI_connect();
for (each stat in local_stats) {
    SPI_execute_with_args("INSERT INTO query_store_intervals ...");
}
SPI_finish();
```

*Analogy: 50 people adding scores to a whiteboard. Instead: photograph the board (1ms), erase it (1ms), step away, write scores into a notebook at your desk.*

# Setup: This One Actually Needs a Restart

```
# Edit postgresql.conf:
shared_preload_libraries = 'pg_query_store'
compute_query_id = on
```

```
# Restart Postgres (required!)
sudo service postgresql restart

# Then:
psql -c "CREATE EXTENSION pg_query_store;"
```

This IS a live demonstration of the constraint. The restart requirement isn't a design quirk, it's the multi-process architecture showing itself.

# We Just Built a Tiny `pg_stat_statements`

| Feature | `pg_query_store` (ours) | `pg_stat_statements` |
|---|---|---|
| Hook | `ExecutorEnd_hook` | `ExecutorEnd_hook` |
| Storage | Fixed array (100 queries) | Dynamic hash table |
| Locking | Single LWLock | Per-bucket LWLock |
| Metrics | count, duration, buffers | Much more |
| Output | Table flush | SRF (set-returning) |
| Query ID | `queryDesc->plannedstmt->queryId` | Same |

# Demo 06: `pg_query_store`

- Show `postgresql.conf` setup (`shared_preload_libraries`, `compute_query_id`)
- `sudo service postgresql restart`
- `CREATE EXTENSION pg_query_store`
- Run `stress_test.sh` (pgbench: 5 connections, 10 seconds)
- `SELECT pg_query_store_flush()` — persist to DB
- Query `query_store_intervals`: top queries by execution count
- `SELECT pg_query_store_reset()`

Let's stress test the engine

# Act III Reveals…

1. Postgres has a formal query pipeline, and every stage is hookable
2. Hooks are global function pointers. Zero abstraction. The simplicity IS the feature.
3. The chaining contract keeps extensions coexisting: always save `prev_*`, always call it
4. SPI is the executor talking to itself, the foundation of ALL procedural languages
5. The recursion guard is a one-boolean firewall against infinite loops
6. Postgres is multi-process, shared memory with LWLock is the IPC mechanism
7. `shared_preload_libraries` runs at postmaster startup, the only window for shared memory

# Level 6: Rust with pgrx

## Modern Developer Experience

Native Postgres extensions with Rust safety and crates ecosystem

# Why Rust for Extensions?

### CLASSIC C PATH

- Maximum control
- Low-level APIs
- More memory-safety risk

### RUST + PGRX PATH

- Type-safe APIs
- Modern tooling + crates
- Fewer common footguns

# Demo 7: Hello pgrx

```rust
use pgrx::prelude::*;

pgrx::pg_module_magic!();

#[pg_extern]
fn hello_pgrx() -> &'static str {
    "Hello from pgrx!"
}

#[pg_extern]
fn add_integers_rust(a: i32, b: i32) -> i32 {
    a + b
}
```

Same SQL-facing behavior as our C demo, with simpler signatures

# Demo 8: HTTP + JSON in Extension

```rust
use reqwest::blocking::Client;
use serde_json::Value;


#[pg_extern]
fn fetch_url_data(url: &str) -> String {
    Client::builder()
        .timeout(std::time::Duration::from_secs(5))
        .build()
        .and_then(|c| c.get(url).header("User-Agent", "pgrx-demo/1.0").send())
        .and_then(|r| r.text())
        .unwrap_or_else(|e| format!("Request failed: {}", e))
}
```

Crates ecosystem inside Postgres: `reqwest`, `serde_json`, and more

# Demo 9: Background Worker Builder

```rust
#[pg_guard]
pub extern "C-unwind" fn _PG_init() {
    BackgroundWorkerBuilder::new("Heartbeat Worker")
        .set_function("heartbeat_main")
        .set_library("pg_heartbeat_worker")
        .enable_spi_access()
        .set_start_time(BgWorkerStartTime::RecoveryFinished)
        .load();
}
```

Cleaner setup than raw C BGW structs, still full Postgres power

Requires `shared_preload_libraries='pg_heartbeat_worker'` + restart

# Rust Trade-offs (Realistic View)

- **Pros:** Memory safety, modern DX, rich ecosystem
- **Pros:** Faster onboarding for many app developers
- **Trade-off:** Longer initial compile times
- **Trade-off:** Need Rust skills in team

# What We Learned

From simple SQL to kernel internals

# The Journey Recap

| Level | Concepts | Skills |
|---|---|---|
| **1: SQL-Only** | Extensions, PGXS, Control files | Packaging, Distribution |
| **2: C Functions** | C API, Datum, Macros | Native code, Performance |
| **3: Data Types** | Type system, I/O functions | Domain modeling |
| **4: Hooks** | Function pointers, Execution flow | Interception, Monitoring |
| **4.1: SPI** | Server Programming Interface | SQL from C, Recursion handling |
| **5: Query Store** | Shared memory, LWLocks, Concurrency | Aggregation, Deadlock prevention |

| | | |
|---|---|---|
| **6: Rust + pgrx** | Typed APIs, crates, BGW builder | Safer extension development |

# Key Takeaways

- **Start Simple:** SQL-only extensions teach 80% of concepts
- **PGXS is Magic:** Handles complexity of building/installing
- **Hooks are Powerful:** But always chain properly!
- **Shared Memory is Fast:** But requires careful synchronization
- **Rust + pgrx is Practical:** Modern DX without losing native speed
- **Study the Source:** `src/backend/` has all the answers

# Common Pitfalls

- Forgetting `PG_MODULE_MAGIC`
- Not chaining hooks properly
- Using `malloc()` instead of `palloc()`
- Holding locks during I/O operations
- Not handling recursion in hooks
- Ignoring memory contexts

# Resources for Deep Diving

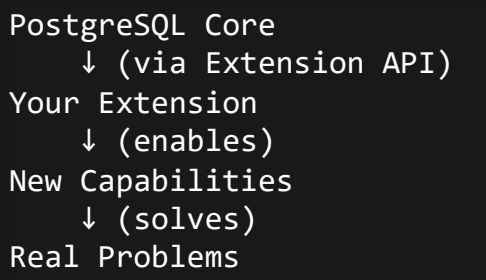
- **Documentation:** postgresql.org/docs/current/extend.html
- **Source Code:** github.com/postgres/postgres
- **PGXN:** pgxn.org (Extension Network)
- **Mailing Lists:** pgsql-hackers, pgsql-general
- **Books:** "PostgreSQL Server Programming" (Packt)
- **Examples:** contrib/ directory in Postgres source

# What's Next?

Ideas for your own extensions:

- **Custom Aggregates:** Median, percentiles, statistical functions
- **Background Workers:** Continuous maintenance tasks
- **Foreign Data Wrappers:** Query APIs, NoSQL databases
- **Custom Scan Providers:** Alternative execution strategies
- **Index Access Methods:** Specialized indexes
- **Procedural Languages:** PL/YourLanguage

# The Extension Ecosystem




You're now part of the extensibility story!

# Final Thoughts

> *"The best way to learn Postgres internals is to try to extend it. You'll quickly discover how well-designed and thoughtful the architecture is."*

Start building. Break things. Read source code. Ask questions.

That's the hard way—and the best way.

# Questions?

Thank you!

Contact: lorenz@psg.de