1
The Complete Journey

When we type `SELECT * FROM users WHERE id = 42;` and press Enter, what happens inside PostgreSQL? Our query triggers a complex series of operations involving multiple processes, sophisticated memory management, and decades of optimization research.

Understanding PostgreSQL's query processing transforms how we write efficient queries, diagnose performance issues, and master database development. This book uses PostgreSQL's actual implementation to illuminate how modern databases work.

In this chapter, we'll explore:

- PostgreSQL's process-per-connection architecture
- The six-stage query processing pipeline
- Hierarchical memory context system
- Pull-based execution model

Let's begin with a high-level view of the complete journey from SQL text to query results.

The Big Picture: From SQL to Results

Before we dive into specifics, let's establish the complete journey our query will take. Understanding this roadmap helps us appreciate why each component exists and how they work together.

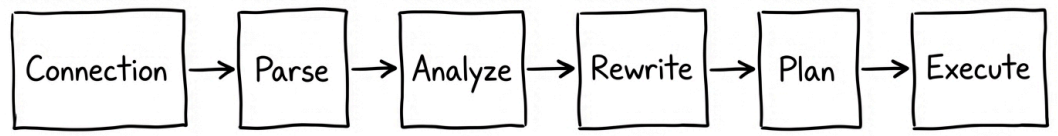

Figure 1.1: PostgreSQL Query Processing Pipeline

PostgreSQL transforms every query through six fundamental phases, as the diagram above illustrates. First, our client establishes a connection to PostgreSQL's postmaster process. The parser then converts raw SQL text into a structured parse tree. Next, the analyzer transforms this into a semantically validated query tree. The rewriter applies rules and views. The planner creates an execution plan. Finally, the executor produces results.

To understand how this pipeline works, we first need to examine the architectural foundation that makes it possible. We'll walk through the complete process step by step in this chapter, but before diving into each phase, let's establish the underlying architecture that enables PostgreSQL's query processing system.

PostgreSQL Architecture Overview

PostgreSQL's architecture is built around a fundamental design decision: process isolation. Unlike many modern applications that use threads for concurrency, PostgreSQL creates a separate operating system process for each client connection. This process-per-connection model provides strong isolation—if one client's session crashes or has a memory issue, it cannot affect other clients.

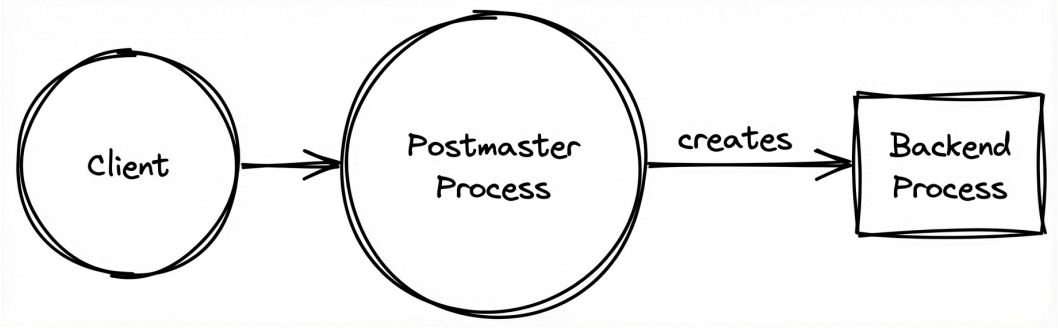

Figure 1.2: PostgreSQL Process-Per-Connection Architecture

The architecture centers on two main process types: the postmaster process and backend processes. The postmaster process acts as the server's main coordinator, listening for incoming connections and managing the overall system. When a client connects, the postmaster creates a dedicated backend process that handles all operations for that specific client connection. This backend process has its own memory space and executes queries independently of other clients.

The postmaster also manages various background processes that handle housekeeping tasks like automatic vacuuming, writing "dirty pages" (modified data currently in memory) to disk, and managing the write-ahead log (the journal that ensures data safety during crashes) for recovery. These background processes work continuously to maintain database health and performance without interfering with client query processing.

PostgreSQL uses a sophisticated memory management system based on hierarchical contexts. Think of memory contexts as nested containers where each container can hold both data and smaller containers inside it. When PostgreSQL destroys a parent context, it automatically destroys all child contexts contained within it, ensuring automatic cleanup and preventing memory leaks.

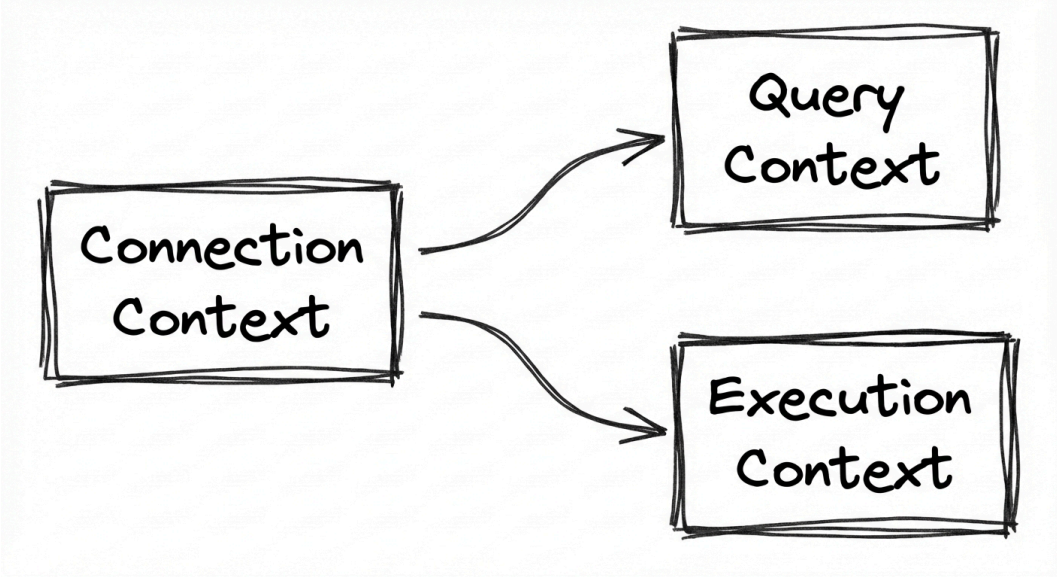

Figure 1.3: PostgreSQL Hierarchical Memory Context System

The system works like a tree of memory allocators. Each database connection has a top-level context that persists for the entire session. When processing a query, PostgreSQL creates temporary contexts for different phases of query processing. When the query completes, these temporary contexts are destroyed, automatically freeing all memory allocated within them. This design prevents memory leaks even when queries encounter errors or are cancelled mid-execution.

PostgreSQL also maintains shared memory structures that all processes can access. The most important is the shared buffer pool, which caches frequently accessed database pages in memory. This shared cache dramatically improves performance by reducing disk I/O, as multiple clients can access the same cached data without redundant disk reads.

With this architectural foundation in place, let’s prepare our environment for the practical examples throughout this book.

Setting Up Your Environment

To follow along with the examples in this book, you'll need PostgreSQL installed and the pagila sample database loaded. Pagila contains data for a fictional DVD rental store with tables like film, customer, rental, and payment—perfect for exploring query processing.

PostgreSQL Version: This book covers PostgreSQL 18, and we recommend using version 18 for the best experience. That said, most examples will work with earlier versions (PostgreSQL 12 and later), though some newer features and output formats may differ slightly.

Loading the pagila database:

```
# Download pagila
git clone https://github.com/devrimgunduz/pagila.git
cd pagila

# Create and populate the database
createdb pagila
psql pagila < pagila-schema.sql
psql pagila < pagila-data.sql

# Verify (should show 1000 films)
psql pagila -c "SELECT COUNT(*) FROM film;"
```

Throughout this book, we'll use pagila tables in our examples. You can explore the schema anytime with `\dt` in psql or check table structures with `\d table_name`.

With our environment ready, let's trace how a query actually enters and flows through the system.

The Journey Begins: Query Submission

Every query starts with a client connection that follows PostgreSQL's wire protocol for communication.

When our application connects to PostgreSQL, it establishes a TCP connection to the postmaster process. The postmaster accepts the connection and performs authentication. After successful authentication, the postmaster creates a dedicated backend process to handle this client session.

Once connected, our client sends SQL query text using PostgreSQL's wire protocol. The protocol supports both simple queries (like ad-hoc SQL statements) and extended queries (which enable prepared statements with parameters). The backend process receives this text and performs initial validation to check for proper encoding and potential security issues.

This connection model means each client gets dedicated resources and isolation, but it also means PostgreSQL doesn't scale to thousands of concurrent connections as efficiently as some other databases. For high-connection scenarios, connection pooling becomes essential.

Once our query text reaches the backend process, the real work begins. PostgreSQL must transform our SQL into an executable plan through a sophisticated multi-stage pipeline.

The Processing Pipeline

Each phase of query processing transforms the query representation and adds semantic information plus optimization decisions that guide execution. The processing pipeline diagram below shows how PostgreSQL transforms our SQL statement through distinct stages.

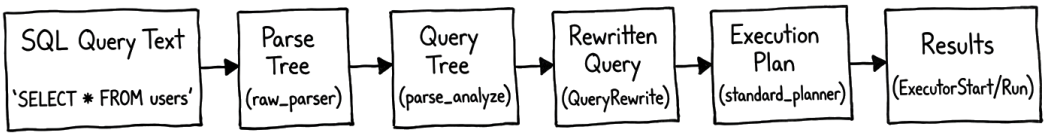

Figure 1.4: Detailed PostgreSQL Query Processing Pipeline

This pipeline architecture allows PostgreSQL to maintain clean separation between different concerns. The parser focuses purely on syntax, the analyzer handles semantics and type checking, and the planner focuses on optimization. This modular design makes the codebase maintainable and allows each component to be optimized independently.

Let's examine each pipeline phase in detail, starting with how PostgreSQL converts raw SQL text into structure.

Parsing: From Text to Structure

The parsing phase converts raw SQL text into a structured parse tree. PostgreSQL's parser reads our query character by character, applying grammar rules to identify SQL keywords, table names, column references, and operators. Think of this like a language teacher analyzing a sentence—the parser identifies nouns, verbs, and clauses, but doesn't yet understand what they mean in context.

This parse tree represents our query's syntactic structure. It knows that `SELECT name FROM users` has a SELECT clause with a column reference and a FROM clause with a table reference, but it lacks semantic information about what these actually represent in our database. The parser doesn't know if the `users` table exists, if the `name` column is valid, or what data types are involved.

With the syntactic structure established, PostgreSQL needs to understand what the query actually means in the context of our database schema. This is where the analyzer steps in.

Analysis: Adding Meaning to Structure

The analyzer takes over next, transforming the parse tree into a semantically validated query tree. This is where PostgreSQL moves from understanding grammar to understanding meaning—like a teacher who not only knows the sentence structure but also understands what the words refer to in the real world.

This analysis phase resolves table and column references and applies type checking. PostgreSQL verifies that the tables exist, the columns are real, data types are compatible, and permissions allow access. The result is a representation that knows `users` refers to a specific table with specific columns, and `name` refers to a text column in that table.

Now that PostgreSQL understands what our query means, it needs to check if any automatic transformations should be applied before optimization begins.

Rewriting: Applying Rules and Transformations

With a semantically valid query tree, PostgreSQL applies any rewrite rules. Rewrite rules are transformations that PostgreSQL applies automatically to modify our query before optimization. The most common examples include:

- **View expansion**: When we query a view, the rewriter converts it into a query against the underlying base tables with appropriate WHERE clauses and JOINs
- **Security policies**: The rewriter adds row-level security policies as additional WHERE conditions

- **User-defined rules**: Custom transformation rules that developers can define

For instance, if we query `SELECT * FROM active_users` where `active_users` is a view defined as `SELECT * FROM users WHERE status = 'active'`, the rewriter transforms our query to directly access the `users` table with the appropriate filter.

With all transformations applied, PostgreSQL now has the final query that needs to be executed. The next challenge is determining the most efficient way to execute it.

Planning: Finding the Optimal Path

The query then moves to the planner, which represents the most sophisticated part of PostgreSQL's query processing. The planner's job is answering the question: "What's the most efficient way to execute this query?"

The planner considers available indexes, table statistics, and join algorithms to create an optimal execution plan. It examines multiple join orders (should we join table A to B first, or B to A?), considers different index access paths (use the index on column X or scan the whole table?), and estimates costs using collected statistics about data distribution and table sizes.

The result is an execution plan containing a tree of operations that specify exactly how to retrieve our data most efficiently. This plan might say "use the index on user_id, then join with the posts table using a hash join, then sort the results."

With the optimal execution plan in hand, PostgreSQL is finally ready to retrieve our data and return the results.

Execution: Producing Results

With an optimal plan ready, PostgreSQL executes it to produce results. The executor implements a pull-based model where each operation in the plan produces rows on demand. A pull-based model means data flows through the plan tree only when requested by the operation above it—like a waiter who only brings food when we ask for it, rather than continuously delivering dishes whether we want them or not.

In PostgreSQL's approach, the top operation requests a row, which causes its child operation to request a row, and so on down the tree until actual data is retrieved from disk or cache. This design is memory-efficient because PostgreSQL only processes data when it's actually needed.

The executor processes the plan and generates result rows one by one. PostgreSQL then formats these rows according to the wire protocol format expected by the client and transmits them over the network connection. Once all results are sent, PostgreSQL automatically cleans up any temporary memory contexts and releases locks acquired during query execution.

Summary

In this chapter, we've learned PostgreSQL's foundational architecture: the process-per-connection model that provides isolation, hierarchical memory contexts that prevent leaks, and the six-stage query pipeline that transforms SQL text into results. We've also discovered PostgreSQL's pull-based execution model.

Understanding these architectural foundations prepares us to appreciate the intricate details of each processing phase. Once we understand how these systems work together, we'll develop better intuition for query optimization and performance troubleshooting.

Now that we've surveyed the complete landscape, the following chapters explore each component in detail. Chapter 2 examines connection establishment and the wire protocol, revealing how clients communicate with PostgreSQL. We'll trace the exact path our query takes from network packet to parse tree, setting the stage for deeper exploration of parsing, planning, and execution in subsequent chapters.