Phi-Database: Building a Relational Database from Scratch

Abstract

Phi-Database is a relational database I built in C++20 to support SQL-92 operations. This report goes over the development process, what I built, the problems I ran into, and what I learned. The project covers storage management, query processing (lexing, parsing, execution), and transaction processing.

1. Introduction

1.1 Why I Built This

I wanted to actually understand how databases work under the hood. PostgreSQL, MySQL, SQLite - they all hide the complexity of what's actually happening when you run a query. I wanted to know:

- How do SQL queries get turned into something executable?
- How is data actually stored on disk?
- How does everything connect together?
- How do transactions maintain consistency?

Beyond databases, I also intended to use this project to improve my system design skills and get deeper into C++ implementation details. I never actually had the chance to build an end-to-end system before, as I have always spent my time improving on existing systems. Because I already had experience taking CS 316 and spending quite a lot of time working with databases at internships, I figured that a database would be a great project to actually design out. Secondly, I wanted to improve my C++ skills. I spent most of my time at Duke programming in memory-managed languages like Go, Java, and Python. While they are great for programming velocity, I believe it is essential for me to become much more acquainted with a memory-unsafe language like C/C++. This ties into my final desire to dive deeper into Linux. I want to understand what happens at the kernel level and read the manpages to gain a better understanding of the implementation underlying all the syscalls I make. In essence, by getting a first-principles view on how to create a database, I was hoping to improve my memory management skills, system design abilities, and gain a deeper understanding of Linux.

1.2 What I Built

- SQL Compiler (lexer + recursive descent parser that outputs Relational Algebra trees)
- Storage Manager (heap files with page directories, table operations)
- Page Manager (file I/O with caching)
- Query Executor (RA Tree to storage operations translator)
- Transaction Processor (Scheduler)

1.3 Tech Stack

- C++20
- CMake 3.24+
- Address Sanitizer for catching memory bugs

- Linux/GCC 11

2. Architecture

The architecture of my database is intentionally simplified compared to production systems. After building the prototype, I realized the problem space splits into two broad goals: processing millions of transactions concurrently or reducing latency for an individual query. Each goal contains countless subproblems, and I chose to focus on the latter. I initially planned to add a cost-based SQL query optimizer, but time constraints meant I concentrated on delivering a correct system with a few targeted storage optimizations instead.

The system is organized into four main layers. SQL queries are tokenized and parsed into a relational algebra tree. The query executor turns that tree into an operator pipeline and runs it against the storage layer, which performs all disk I/O through fixed-size pages. Transaction processing sits alongside these layers to schedule operations.

Component Breakdown

The SQL Compiler spans 4 files (lexer, parser, AST definitions). The Query Executor handles the RA tree to operator translation. Storage Manager covers tables, heap files, and rows. Page Manager handles file I/O and caching. Transaction processing handles scheduling. The rest is shared types and utilities.

3. Implementation

3.1 SQL Compiler

Lexer

The lexer tokenizes SQL into a stream of tokens. It handles 40+ keywords, identifiers, literals (int, float, string, bool), operators, and both comment styles. This component was straightforward because it simply converts a SQL query into a list of tokens for the parser to consume.

```
std::vector<Token> tokenize_query(string& query);
```

Parser

The parser is a recursive descent parser that took a surprisingly long time to get right. This is a key step in any compiler, and while it is logically straightforward, it ended up being a hassle to nail down the SQL grammar. I spent the most time on the FROM clause, especially the JOIN portion. Initially I tried a binary-search-style approach where I recursively combined joins from halves because I assumed the AST would be a balanced tree. That was wrong—the grammar composes joins strictly from left to right. After rewriting the grammar to match SQL-92's left-associative JOINs, I started to follow the published grammar closely. At first I thought the SQL grammar was insufficient, so I bounced between documentation sets; only after fixing the bug did I realize that implementing a language grammar is literally about following each production as written. For example, the SQL-92 BNF for FROM constrains the tokens to commas, table references, or collection member declarations; the parser just has to check which token appears, consume the corresponding production, and advance. Building this parser demystified parsing for me. It highlighted

how recursion mirrors the grammar and why recursive descent feels more natural than a loop-based approach here.

Supports:

- SELECT with DISTINCT, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT
- INSERT (single and multi-row)
- UPDATE with SET and WHERE
- DELETE with WHERE
- CREATE TABLE with constraints
- DROP TABLE

Expression precedence goes OR (lowest), AND, NOT, comparisons, add/sub, mult/div (highest).

The parser outputs RA nodes directly instead of an intermediate SQL AST:

```
enum class RANodeType {
    TABLE_SCAN,
    PROJECT,
    SELECT_OP,
    INNER_JOIN,
    LEFT_JOIN, RIGHT_JOIN, FULL_JOIN,
    CROSS_PRODUCT,
    GROUP_BY,
    SORT,
    LIMIT_OP,
    // ...
};
```

3.2 Storage Manager

The two building blocks I used for storage are heap files and fixed-size pages. Each heap file contains a fixed number of pages, and each page has a configurable size (128 bytes for early testing, scaled up for realistic workloads). That tiny size is obviously impractical in production but kept the concepts manageable. Tables expose insert/read-row operations, and those operations sit on top of HeapFile, which is responsible for locating the correct page or appending new ones. Every heap file keeps a small metadata header with the record count and a pointer to the next heap file when the table grows beyond a single file.

I realized late that this layout is not very I/O optimal: if a record lives in one of the last heap files, the engine may need to traverse each file sequentially before issuing yet another I/O to read the page itself. I originally assumed the page cache would hide most of that cost, but that assumption turned out to be wrong. Another design debate was whether HeapFile should consult the page cache directly. I ultimately decided the table layer owns cache lookups because HeapFile is conceptually an interface to disk, not memory. The page cache behaves like the database's memory hierarchy, and the table code should try it first, but HeapFile should not peek into it. Building this component reinforced how similar database storage is to an OS design except that the emphasis is "memory first."

3.3 Query Executor

This component essentially acts like a generator function in Python. After some research I chose the Volcano iterator model because it is ubiquitous and maps cleanly to relational algebra: each operator exposes open/next/close and pulls tuples from its children. I implemented the basic operators like SeqScan, FilterOp, ProjectOp, LimitOp, CrossProductOp, and NestedLoopJoin.

```
struct StorageOps {
    virtual void open() = 0;
    virtual std::vector<Row*> next() = 0;
    virtual void close() = 0;
};
```

3.4 Page Manager

When I designed the lower file-system layer, I intended DbFile to be a thin wrapper over the POSIX API. DbFile is a singleton that manages all file descriptors and exposes `read_at()`/`write_at()` helpers so a Page can be written at a particular offset safely. I preferred this over sprinkling raw POSIX calls everywhere because it prevents mistakes in byte counts and forces all I/O to use page-sized buffers. It also handily ensures the backing directories exist. The PageCache sits directly on top, caching pages on reads and writes and evicting them in an LRU order when full. This layer was relatively straightforward to implement.

3.5 Transaction Processing

Transaction Scheduler

The scheduler ended up being very bare bones. It was a simple 2PL locking system that I learned from the CS316 lecture. I did not want to overcomplicate this part especially since when I was testing it would be on single queries. I didn't have much trouble in this section as the algorithm of 2PL locking is very well defined, but this scheduler could definitely be much more refined.

```
void TScheduler::schedule_transaction() {
    // Groups operations by table
    // Adds LOCK operations before table access
    // Adds UNLOCK operations after table operations complete
    // Adds COMMIT at end of transaction
}
```

5. Problems I Ran Into

5.1 Bazel to CMake

Started with Bazel because I thought it would be cool. It wasn't. The biggest issue was Bazel's sandboxing - it creates its own isolated environment for builds, which means you can't easily access the files that get created during the build process. For a database project where I constantly needed to inspect heap files and debug page layouts on disk, this was a nightmare. I couldn't just look at what was being written without jumping through hoops.

Bazel also has a steep learning curve with its BUILD files and dependency management. Way too complex for what I needed. Switched to CMake and everything got simpler - I could just build and run, then inspect the output files directly.

5.2 Table vs HeapFile

This was a tricky system design issue where I didn't know how to bridge the gap between the notion of a table and the page cache. While it seems like an easy decision in hindsight, the table should not manipulate pages directly; it should issue logical read/write requests. That abstraction hides the fact that tables span multiple pages and lets HeapFile manage the details, which keeps the implementation much cleaner.

5.3 Struct Alignment

A big issue I grappled with was struct padding. Compilers align structs to natural boundaries, which meant that when I inspected files I saw seemingly random bytes I never wrote. I eventually realized I needed to mark these structs with `__attribute__((packed))` (or equivalent) so Clang would keep the layout I expected.

5.4 Pre Optimizations

Probably the most important thing I learned on this project was the danger of premature optimization. I never realized how much progress it could destroy until I obsessed over small details before the system even ran end to end. The page cache is the best example: I overthought eviction policies and I/O counts even though the surrounding components were still in flux, which meant I rewrote or discarded most of that work. I originally attacked the stack from the bottom up because I assumed the upper layers would be easy to interface with, but in reality I kept changing the lower layers to accommodate new requirements. The big lesson was to build an MVP first and worry about optimizations later.

5.5 Build Management

At first I used Bazel as my build system as a lot of my friends recommended it to me from their experience. To be honest, it was a very nice system to use and built very quickly, but an issue I had was that Bazel's sandboxed execution model made it impossible to check the created files I made were correct. When I was working on DbFiles and wanted to ensure the test bytes I wrote were exactly what I wrote, I spent around 2-3 days wondering where the file was actually stored. I realized too late that it was actually in a random sandboxed folder I couldn't really access and this was the intended implementation of Bazel. Coming from using basic MakeFiles this was a big surprise, so I ended up making the switch to CMake just to be able to easily test file writes and know where my built executables were.

5.6 Memory Management Bugs

Working on memory bugs after years of garbage-collected languages was an adjustment. I relied heavily on AddressSanitizer to catch cases where I forgot to allocate or freed the wrong thing, and as the codebase grew I had to structure it carefully so I could hunt down leaks. I intentionally stuck to a C-like style and avoided smart pointers, which made this both harder and more educational.

5.7 POSIX IO issues

There were a couple of POSIX API details I didn't know that tripped me up. The big one is that `write()` may return fewer bytes than requested without being an error. I assumed the call either succeeded fully or failed, so I never checked the return value carefully. That led to a nasty bug where writes were nondeterministic and the bytes on disk didn't match what I thought I had written. Adding explicit loops to retry partial writes fixed it.

6. Results

The database can:

1. Parse simple SQL queries into RA trees. A query like:

```
SELECT department, COUNT(*) as emp_count
FROM employees WHERE salary > 50000
GROUP BY department HAVING COUNT(*) > 5
ORDER BY emp_count DESC LIMIT 10
```

2. Create tables with schemas
3. Execute simple queries through the operator pipeline
4. Schedule transactions with proper lock ordering

Operators return batches of 64 rows for cache efficiency. All disk access goes through fixed-size pages. AddressSanitizer is on for all builds.

7. What I Learned

Technical

- Relational Algebra is the key abstraction between SQL and execution
- The Volcano iterator model composes really well
- Don't assume you can memcpy structs with complex types
- CMake > Bazel for projects this size
- Two-phase locking is straightforward to implement but has concurrency limits

Process

- Switching between components when stuck prevents burnout
- Integration testing matters - things that work alone break together
- Writing things down (like Parser.md) helps clarify thinking

Database Stuff

- SQL has a lot of edge cases
- Page size and row format affect everything above them
- The catalog (table metadata) is needed everywhere
- Transaction ordering is tricky to get right

9. Conclusion

I heavily underestimated the detail that goes into building a database. From a first glance, it genuinely didn't seem too difficult, and it certainly didn't help that I lacked deep familiarity with C++ and Linux internals going in. Over the course of researching and building different components (disk management, buffer pools) I gained a real understanding of what happens under the hood. While I didn't tackle harder challenges like concurrency control, query optimization, or full ACID compliance, building even a simplified version from first principles was a valuable learning experience that gave me a much deeper appreciation for production database systems.

References

1. "Database System Concepts" by Silberschatz, Korth, and Sudarshan
2. Berkeley DB Architecture Oracle White Paper
3. Duke Devils Database (DDB)
4. SQL-92 Specification
5. Phoenix SQL Grammar Documentation
6. https://craftinginterpreters.com/introduction.html