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 a much simpler version than how difficult databases are. After building this database, the problem can be split into two: processing millions of transactions at once or improving latency once a query is received. There are of course a plethora of problems that are involved when addressing these two problems, and I chose to tackle the second one. Even within the second problem, there are various sections that can massively improve the system. I initially attempted to introduce a more intelligent SQL Query Optimizer given information about the tables, but I did not have enough time to do so. Thus, I focused a majority of my efforts on simply building a working system and some latency improvements on the storage operation side.

The system can be split into four main layers. SQL Querying at the top which gets tokenized and parsed into a Relational Algebra tree. Then the query executor builds an operator tree from that and runs it against the storage layer, which handles all the disk I/O through fixed-size pages. Transaction processing sits alongside to handle operation scheduling.

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. Handles 40+ keywords, identifiers, literals (int, float, string, bool), operators, and both comment styles. This was very easy to implement as it essentially converts a SQL query into a list of tokens that can be used by the parser to create the AST.

```
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 quite a hassle to find and implement the SQL grammar. I ended up actually spending the most time on the parser as I had trouble with the FROM statement grammar in particular the JOINing part of it. Initially I tried to almost do a binary search

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 stored things in my database were heap files and pages. Every heap file contained a fixed number of pages and each page was a set amount of data which I set to 128 bytes as default for testing and scaled higher for rows and such. This is obviously an obscenely small amount of bytes to set a page, but conceptually it served its purpose. The way it would work is that the table had insert and read row operations that served as the basic operations of read and write that every storage op needs. These then work on top of the HeapFile to basically determine where the data actually is to read it or concatenate on top of the existing HeapFiles with rows to allow future reads find it. The heap files each had a small metadata header where it would keep track of how many records are currently in the file and a pointer to the heap file when the table is larger than the size of a heap file. I realized late that this wasn't very IO optimal as if data was in one of the last heap files then I would sequentially have to do that many IOs to get to it. In addition, I would then have to do the additional IO to actually pull the page from disk into memory. At the time, I believed that I could heavily rely on the Page Cache to reduce the IOs made, but looking back in hindsight, this assumption was not true. During the creation of HeapFile, I had to grapple with another problem which was whether the HeapFile should always read from disk or should it always try to pass through and read from the PageCache first. I ended up deciding that the table was in charge of checking the cache for a cheaper read from memory and when inserting a row the table would write through the cache but ultimately use the HeapFile path. I chose to do it in this fashion because the HeapFile should not be an interface over the Page Cache but over the disk. The Page Cache in a sense is "memory" for a database, and as such the table should always attempt to go through it, but ultimately it is not the responsibility of the HeapFile to peek into the cache to get the data. From building this, I realized that the system of a database really is very similar to the model of an OS except it is a "memory first" type of design.

3.3 Query Executor

This essentially acted as a Generator function in python. Through some basic research, I decided to just use the Volcano iterator model as it is the most common iterator model, and it basically runs through the RA tree applying operations in a sort of functional paradigm. I implemented the basic operations like SeqScan, IndexScan, FilterOp, ProjectOp, LimitOp, CrossProductOp, NestedLoopJoin.

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

3.4 Page Manager

When I was designing the lower file system management, I intended DbFile to be a thin wrapper over the POSIX API. DbFile became a singleton that manages all file descriptors and it handles the read_at() and write_at() functions where given a Page as an input it will right it at a particular offset. This was used over the normal POSIX file api because I did not want to accidentally assign the wrong amount of bytes to write and the base unit of disk reads/writes are gonna come in at page size anyway. In addition it serves the nice purpose of automatically creating directories for heapfiles and such. Overall it was intended to be very simple. The PageCache lays right on top of it where it caches all the pages written and read. When the cache is full, the page cache will evict on a LRU basis. This part was pretty straightforward to implement and I did not have much trouble here.

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 weird system design issue I had where I didn't know how to bridge the gape between the notion of a Table and the Page Cache. While it seems like an easy design choice, the table should not work on pages at all and simply make requests to read or write. This abstracts the notion of tables being multiple pages large where the heapfile can operate on all of that making the entire implementation a lot cleaner.

5.3 Struct Alignment

A big issue I grappled with structs is that structs are padded by the compiler to align with the memory line. This is especially finnicky because when I went to read the files I would see random bytes that I did not specify from the struct, and thus I would be very confused where they come from. I didn't realize that to actually keep a struct the way it was you needed to write **attribute** before it to signal to Clang that it can keep it the way it is.

5.4 Pre Optimizations

Probably the most important thing i learned working on this project was the dangers of pre optimizations. I really never learned how slow programming could be until I started worrying about all the small things before the big system could even be built. A big example of this was when I was building page cache. I really really over thought about it at first to make sure that every page was clearly in memory and I was using the minimum number of IOs as possible when in reality it all was going to change since my entire project structure changed. My intended approach was to go bottom up because I thought the top would simply be easy to interface, but waht I realized was that I had to keep changing the things I already built to accomadate the inputs that were being fed to it. This essentially made all the optimizations I did useless as I had to delete my work and redo it. This was in fact the biggest lesson I learned in that I need to get a MVP working product out before I think about any optimizations.

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 using garbage collection was a bit of a change. I heavily relied on Address Sanitizer to catch cases where I forgot to malloc and as the size of the repo grew I had to structure the codebase well so I could hunt down memory leaks easily. I really wanted to follow a more C like structure coding style and avoided using smart pointers which would have made this problem a LOT easier.

5.7 POSIX IO issues

There were a couple issues about POSIX API that I did not realize and tripped me up. The first one is the on write() the returned number of bytes can be LESS than the requested number of bytes to write. This definitely was not something I expected, as usually in these cases I would think the writes would simply fail. This caused a really frustrating bug where there were non deterministic writes that did not fully write all the bytes into a file and when I checked the bytes were not what I requested. This is because when I checked my writes I was not actually checking to see if it wrote all of them correctly and just checking that it did not write no bytes at all.

6. Results

The database can:

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

```sql
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 Orcal White Paper
3. Duke Devils Database (DDB)
4. SQL-92 Specification
5. Phoenix SQL Grammar Documentation
6. https://craftinginterpreters.com/introduction.html