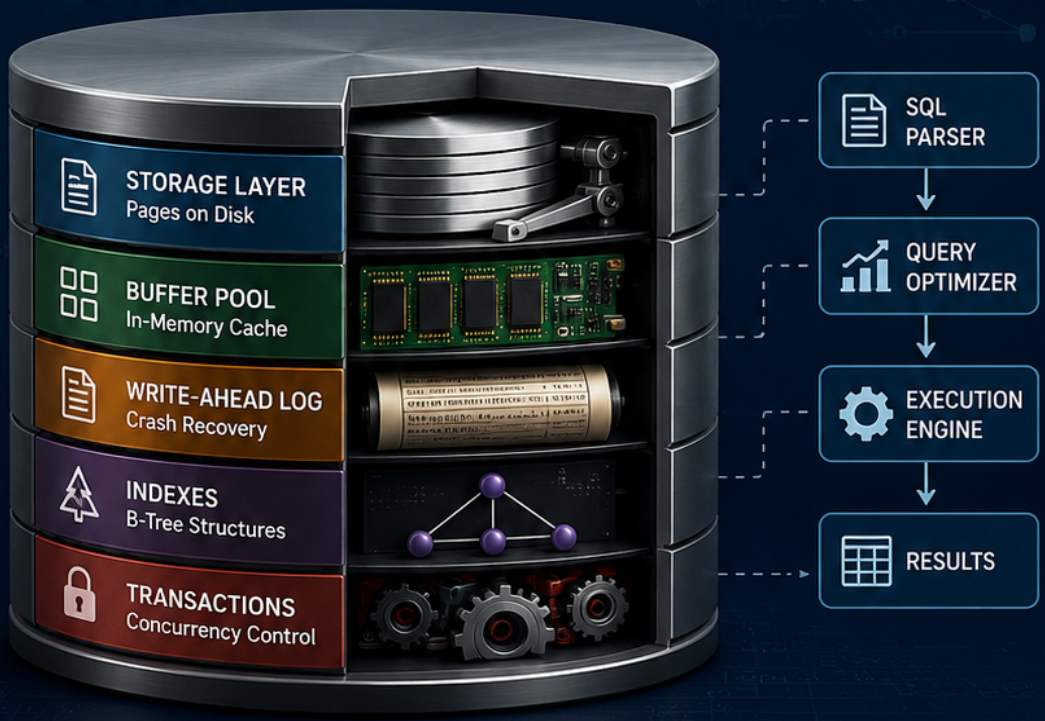
BUILDING DATABASE ENGINES FROM SCRATCH

BUILD A MODERN RELATIONAL DATABASE IN RUST

A HANDS-ON GUIDE TO STORAGE, CONCURRENCY, QUERY PROCESSING, AND RELIABILITY IN SYSTEMS PROGRAMMING



PERSISTENT STORAGE | CONCURRENCY & ISOLATION | CRASH RECOVERY | B-TREE INDEXES | COST-BASED OPTIMIZATION | JOINS, SORTS & AGGREGATIONS

JASON REED

Building Database Engines from Scratch

A Hands-On Guide to Storage, Concurrency, Query Processing, and Reliability in Systems Programming

Steve Publications

This book is available at
https://leanpub.com/buildingdatabaseenginesfromscratch

This version was published on 2026-08-30

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

A Hands-On Guide to Storage, Concurrency, Query Processing, and Reliability in Systems Programming

This book teaches you how modern database systems work by having you build one. Over fifteen chapters we will design and implement a complete relational database engine in Rust: storage layers that read and write pages to disk, a buffer pool manager that caches hot data in memory, write-ahead logging for crash recovery, B-tree indexes for fast lookups, a transaction system with proper isolation and concurrency control, a SQL parser, a cost-based query optimizer, and an execution engine that runs joins, sorts, and aggregations. Every chapter adds working code to the same project so that by the end you have a coherent, functional database application rather than a collection of isolated examples. If you are a software engineer who wants to understand what happens between typing SELECT * FROM users and seeing results on screen, this is the book for you.

Chapter 1: Why Build a Database?

The Hidden Complexity of Every Database Query

A simple query hides enormous engineering. When you type SELECT id, name FROM users WHERE age > 25 ORDER BY name LIMIT 10 into any modern database and press Enter, something remarkable happens in milliseconds. The system parses your text into a structured representation, validates that the table and columns exist, decides how to find the matching rows most efficiently, possibly consults one or more indexes, reads data from storage while managing concurrent access with other queries, applies filters and sorts, and returns exactly ten rows in alphabetical order. If another connection is modifying users at the same time, your query still sees a consistent view of the data. If the power fails mid-execution, committed transactions are never lost and incomplete ones never partially appear. None of this is magic. It is the product of decades of research and engineering distilled into layers of carefully designed subsystems that work together under strict invariants.

Most developers interact with databases as black boxes. You write SQL, you get results, and occasionally you add an index when something runs slow. This works fine for using a database but leaves you unable to answer deeper questions. Why does one query plan outperform another? What actually happens when you commit a transaction? How does the system guarantee durability across crashes? Why do LSM trees excel at write-heavy workloads while B-trees dominate OLTP systems? When should you choose PostgreSQL over SQLite or RocksDB?

The answers lie inside the engine. Every architectural decision in a database solves a concrete problem under real constraints: latency, throughput, memory pressure, disk I/O limits, correctness guarantees, and failure modes. You cannot fully internalize these trade-offs by reading about them abstractly. The most effective way to learn is to build one yourself.

This approach has proven its value across systems programming education. People who have built their own web servers understand HTTP, connection management, and request routing in ways that users never will. People who

have implemented a filesystem understand journaling, allocation strategies, and directory structures at an intuitive level. The same is true for databases. When you write the code that implements a buffer pool replacement algorithm, you understand cache behavior more deeply than any diagram could convey. When you implement crash recovery from a write-ahead log, durability stops being an abstract ACID property and becomes a series of concrete operations with real failure modes you must handle.

The database field is particularly rich for this kind of learning because the problems are well-defined and the solutions have been studied exhaustively. Unlike many areas of software engineering where best practices evolve rapidly and opinions diverge wildly, core database algorithms are mature and thoroughly understood. The B-tree index structure dates to 1970 [1]. Write-ahead logging was formalized in the 1980s through foundational work by Jim Gray, Andreas Reuter, and others [2]. Multiversion concurrency control traces back to David Reed's 1978 dissertation [3]. These algorithms are not relics. They power every major database system in production today: PostgreSQL, MySQL/InnoDB, SQLite, Oracle, SQL Server, CockroachDB, TiDB, Redis, RocksDB, and countless others all use variants of these same core ideas.

Building a database also teaches you systems programming at its most demanding. A database engine must manage memory carefully without garbage collection interference, perform I/O efficiently while respecting storage hardware characteristics, coordinate concurrent threads without deadlocks, handle partial failures gracefully, and maintain strict invariants under all conditions. These are the skills that separate competent engineers from exceptional ones.

What You Will Build and Why It Matters

Over the course of this book we will implement a complete relational database engine called mydb. It will be written in Rust for reasons we will discuss in the next chapter, and it will grow incrementally across all fifteen chapters. Each chapter adds substantial working functionality that integrates with everything built before it. By the end you will have a single coherent codebase that implements:

A storage layer that reads and writes fixed-size pages to disk files using proper binary serialization. A buffer pool manager that caches frequently

accessed pages in memory, uses a replacement algorithm to evict cold pages, tracks dirty pages, and flushes them back to disk with appropriate write policies. A write-ahead logging system that records all modifications before they are applied to data pages, enabling crash recovery through redo and undo operations. A B-tree index implementation that supports fast key lookups, range scans, and concurrent access patterns. A transaction manager providing ACID semantics through locking protocols or multiversion concurrency control with configurable isolation levels. A SQL parser that tokenizes input text, builds abstract syntax trees for a practical subset of the SQL language, and performs semantic validation against schemas. A query planner and optimizer that transforms parsed queries into logical plans, applies rewrite rules, estimates costs using statistics, and selects efficient physical execution strategies. An execution engine implementing core relational operators: scans, filters, projections, sorts, aggregations, and multiple join algorithms including nested-loop, hash, and merge joins.

This is not a toy. While we will simplify certain aspects for educational clarity and explicitly call out where production systems do more, the core algorithms you implement are the same ones used in real databases. The B-tree you build will use the same split and merge operations as PostgreSQL's indexes. The write-ahead log will follow the same protocol that guarantees durability in SQLite and MySQL. The buffer pool replacement policy will address the same fundamental problem that every database faces: how to keep hot data in memory while cold data is evicted to disk.

The project structure is designed so that you can actually use mydb as a working command-line database application. You will be able to create tables, insert data, build indexes, run queries with WHERE clauses and JOINs, execute transactions with proper isolation, and recover from simulated crashes. Along the way we will include benchmarking utilities so you can measure performance characteristics and see the impact of optimizations firsthand.

Why does this matter for your career? Even if you never ship a database engine into production, the understanding you gain will make you a better engineer in every domain. You will understand why certain data structures excel under specific access patterns. You will develop intuition about memory hierarchies and cache behavior that applies to any performance-sensitive system. You will learn how to reason about concurrent execution and failure modes systematically. You will see how abstract correctness properties translate into concrete implementation requirements. These are transferable skills

that improve your ability to design, debug, and optimize any complex software system.

A Tour of the Architecture We Are Going to Construct

Before we start coding, let us take a high-level tour of the architecture. Understanding how pieces fit together will help you see why each component exists and what problems it solves. Think of a database engine as a layered stack where each layer provides abstractions that hide complexity from layers above.

At the lowest level sits physical storage: disks or SSDs where bytes are persisted across reboots. Storage hardware has specific characteristics that fundamentally shape database design. Reading data from disk is orders of magnitude slower than reading from memory, so databases minimize I/O through caching and batching. Sequential reads are dramatically faster than random reads, so databases organize data for scan efficiency. Writes have their own constraints: SSDs wear out with excessive writes, and fsync operations to guarantee durability introduce latency spikes. Every database architect must design around these hardware realities.

Above storage sits the storage manager layer, which organizes raw bytes into structured pages. A page is a fixed-size block of data, typically 4096 bytes, that serves as the fundamental unit of I/O. The storage manager handles reading pages from disk files into memory buffers and writing modified pages back to disk. It manages file formats, record layouts within pages, free space tracking, and basic operations like creating new pages or finding existing ones by page number. This layer knows nothing about SQL or transactions. It simply moves pages between disk and memory according to requests from above.

The buffer pool manager sits on top of the storage manager and provides a crucial optimization: caching. Instead of reading every page directly from disk for every query, the buffer pool keeps frequently accessed pages resident in memory. When a client requests a page, the buffer pool first checks its cache. If the page is present (a cache hit), it returns the in-memory copy immediately. If not (a cache miss), it reads from disk into an available buffer frame. The buffer pool must decide which pages to evict when memory fills up, track which pages have been modified and need flushing to disk, and coordinate access between concurrent readers and writers through pinning semantics

and latches. Buffer pool design is one of the most impactful areas for database performance because cache hit rates directly determine how many expensive disk I/O operations are needed.

The logging layer provides durability through write-ahead logging (WAL). The fundamental insight behind WAL is elegant: instead of writing every modification directly to data pages on disk, we first append a log record describing the change to an append-only log file and flush that log to stable storage. Only after the log is safely on disk do we consider the change durable. The actual data page can be updated in memory and flushed to disk later at our convenience. This approach transforms random writes scattered across many pages into sequential appends to a single log file, which is dramatically more efficient. More importantly, it enables crash recovery: if the system crashes, we replay the log on restart to reconstruct all committed changes that were not yet written to data pages.

The indexing layer provides fast access to data without scanning every page. A B-tree index maintains sorted keys with pointers to data, enabling logarithmic-time lookups instead of linear scans. For a table with one million rows stored across 250,000 pages, a full table scan requires reading all 250,000 pages while a B-tree lookup typically touches only three or four pages. Indexes trade write overhead for read speed: every insert, update, or delete must also modify affected indexes. Beyond B-trees we will explore hash indexes for equality lookups and discuss alternative structures like LSM trees that excel under different workload patterns.

The transaction manager coordinates concurrent access and provides ACID semantics. Atomicity ensures that either all operations in a transaction complete or none do. Consistency means transactions preserve database integrity constraints. Isolation prevents concurrent transactions from interfering with each other in harmful ways. Durability guarantees that committed changes survive crashes. The transaction manager implements these properties through concurrency control mechanisms like locking protocols or multiversion concurrency control, and coordinates with the logging layer for recovery.

The SQL processing layers handle query language parsing, planning, optimization, and execution. The parser transforms text into an abstract syntax tree representing the query structure. Semantic analysis validates that tables and columns exist and types are compatible. The planner converts the validated AST into a logical plan of relational operations. The optimizer applies

rewrite rules and cost models to select an efficient physical execution strategy among many alternatives. Finally, the execution engine runs the chosen plan using operators that scan data, apply filters, perform joins, sort results, and compute aggregations.

At the top sits the client interface: either a command-line tool or network server that accepts SQL statements from users and returns results. This is the only part users interact with directly, but all the complexity we have described operates beneath it to deliver correct, efficient, and reliable behavior.

Implementation Roadmap: From Empty Project to Working Database

The book follows a deliberate progression from low-level storage up through query execution. Each chapter builds on previous work so that code accumulates into a functioning system rather than remaining as isolated examples. Here is the roadmap:

Chapters 2 through 5 establish the foundation: project setup, storage manager, file format with slotted pages for variable-length records, and buffer pool management. After Chapter 5 you will be able to store records on disk and retrieve them efficiently through a caching layer.

Chapters 6 through 8 add reliability and concurrency: write-ahead logging with crash recovery, B-tree indexes for fast lookups, and transaction management with locking and MVCC. After Chapter 8 your database can handle concurrent transactions with proper isolation while surviving crashes without data loss.

Chapters 9 through 11 bring SQL processing to life: parsing text into abstract syntax trees, planning and optimizing queries using cost models and statistics, and executing plans through a pipeline of relational operators. After Chapter 11 you will have end-to-end SQL support for SELECT, INSERT, UPDATE, and DELETE with WHERE clauses, JOINs, GROUP BY, ORDER BY, and more.

Chapters 12 through 14 focus on advanced topics: performance engineering including profiling and optimization techniques, reliability engineering covering crash consistency and corruption recovery, and distributed database concepts including replication and sharding. Chapter 15 concludes by reviewing the complete architecture we built and discussing how it compares to production systems and where you might extend it further.

This progression mirrors how real databases are architected: storage first, then durability and concurrency, then query processing on top. Each layer depends on abstractions provided by layers below, so building bottom-up ensures a solid foundation.

How to Read This Book and Use the Code

This book assumes you are comfortable programming in at least one language and have basic familiarity with data structures and algorithms. Prior database experience is helpful but not required: if you know how to write SQL queries, that is enough background. Some chapters touch on operating system concepts like file I/O, memory mapping, threading, and synchronization, so comfort with systems-level thinking will help.

Every chapter includes complete, runnable Rust code designed to integrate with previous chapters into a single project. The code lives in a consistent repository structure with clear module boundaries. You can follow along by cloning the companion repository and building incrementally as we progress, or you can read through and study the implementations conceptually. Either approach works because every code example is self-contained enough to understand while also fitting into the larger architecture.

Some chapters present simplified versions of algorithms for educational clarity before showing how production systems handle additional complexity. I will always call these simplifications out explicitly so you know where our implementation diverges from what you would find in PostgreSQL or SQLite. This approach keeps individual chapters focused while giving you accurate mental models of real systems.

The book uses inline citations [n] to reference sources for factual claims, historical information, and externally derived material. A consolidated bibliography appears at the end with full references and URLs. You are encouraged to follow these links to primary sources when you want deeper understanding on any topic.

Now let us begin building. In the next chapter we will set up our Rust project, discuss why Rust is a strong choice for database implementation, and write our first code: initializing an empty database file with a proper header structure.

Chapter 2: Project Setup and Systems Foundations

Choosing Rust for Database Implementation

The language you choose for a database engine shapes everything: how you manage memory, how you handle errors, how you reason about concurrency, and ultimately how reliable the resulting system is. For mydb we will use Rust. This is a deliberate choice with trade-offs that deserve explicit discussion.

Rust provides memory safety without garbage collection through its ownership system and borrow checker. Database engines are among the most demanding consumers of memory in any software stack. They manage large buffer pools, maintain complex data structures across threads, and operate for weeks or months without restart. Memory bugs in this context are catastrophic: a use-after-free in a buffer pool can corrupt indexes silently, a double-free can crash the entire process, and dangling pointers can introduce security vulnerabilities that persist undetected for years. Rust's compile-time guarantees eliminate entire classes of these bugs before they reach production.

The absence of garbage collection is equally important. Garbage collectors introduce unpredictable pauses that are unacceptable in latency-sensitive database operations. When a user commits a transaction, they expect a deterministic response time, not one that depends on whether the GC happens to be running a major collection cycle. Rust's manual memory management through ownership gives you control over allocation lifetimes without the safety hazards of C or C++.

Rust also provides excellent support for systems-level programming. You have direct access to file I/O, memory mapping, threading primitives, and atomic operations. The standard library includes robust implementations of common data structures like hash maps and binary heaps. The type system catches many bugs at compile time that would require extensive testing in other languages. Error handling through Result types forces you to consider failure modes explicitly rather than relying on exceptions or silent failures.

The ecosystem is growing rapidly for database work specifically. DuckDB, one of the fastest analytical databases available, is written primarily in Rust and C++. PolarDB-X from Alibaba uses Rust for its distributed transaction coordinator. Many new storage engines and data processing systems choose Rust as their implementation language. The sqlparser-rs crate provides a mature SQL parser used by multiple projects. Crates like tempfile, serde, and criterion make testing, serialization, and benchmarking straightforward.

C++ remains the dominant language in the database world: PostgreSQL, MySQL/InnoDB, MongoDB, Elasticsearch, and many others are C++ codebases. C++ offers comparable performance with a richer history of database-specific libraries and patterns. However, C++ also gives you far more ways to shoot yourself in the foot: undefined behavior from uninitialized memory, data races from unsynchronized access, subtle lifetime bugs from raw pointer misuse. Modern C++ with smart pointers and RAII mitigates many issues but does not eliminate them. For an educational project where correctness matters as much as performance, Rust's stricter guarantees are worth the learning curve.

Go is another option that some database projects use, notably CockroachDB and etcd. Go simplifies concurrency with goroutines and channels, has fast compilation, and produces reliable code. However, its garbage collector introduces pauses that make it less suitable for latency-critical storage operations, and its lack of generics until recently (and still limited compared to Rust) makes implementing generic data structures more cumbersome. For a book focused on internals where memory behavior matters, Go's abstractions hide too much.

The trade-offs are real. Rust has a steeper learning curve than C or Go. The borrow checker can be frustrating when you first encounter it, especially for patterns involving shared mutable state that databases rely heavily on. Compilation times are slower than many alternatives. Some advanced database techniques like lock-free data structures require unsafe Rust blocks, which reintroduce the possibility of memory unsafety (though confined to auditable boundaries). We will address all these challenges as we encounter them and show how idiomatic Rust patterns handle common database engineering problems.

Project Structure and Build Configuration

Let us create our project structure. We will organize mydb as a Cargo workspace with clear module boundaries that mirror the architectural layers

described in Chapter 1. This structure will evolve as we add subsystems, but establishing it now prevents refactoring pain later.

Create a new Rust project:

```
cargo new mydb --lib
cd mydb
```

The initial Cargo.toml looks like this:

```
[package]
name = "mydb"
version = "0.1.0"
edition = "2021"

[dependencies]
thiserror = "1.0"
anyhow = "1.0"
log = "0.4"
env_logger = "0.10"

[dev-dependencies]
tempfile = "3.8"
criterion = "0.5"

[[bench]]
name = "storage_bench"
harness = false
```

We use thiserror for typed error enums, anyhow for ergonomic error handling in application code, log and env_logger for structured logging throughout the engine, tempfile for tests that need temporary database files, and criterion for benchmarking. As we add features we will include more dependencies: a tokenizer crate for SQL lexing, maybe an LRU cache implementation for the buffer pool, and serialization utilities.

Now let us set up the directory structure in src/:

```
src/
├── lib.rs              # Library root, public API exports
├── main.rs             # CLI entry point (in examples/ or as binary)
├── storage/            # Storage layer: files, pages, I/O
│   ├── mod.rs
│   ├── file_manager.rs # Database file operations
│   └── page.rs         # Page abstraction and layout
├── buffer_pool/        # Buffer pool manager
│   ├── mod.rs
│   └── replacer.rs     # Page replacement algorithms
├── log/                # Write-ahead logging
│   ├── mod.rs
│   └── record.rs       # Log record format
├── index/              # Index implementations
│   ├── mod.rs
│   ├── btree.rs        # B+ tree index
│   └── hash_index.rs   # Hash index
├── transaction/        # Transaction management
│   ├── mod.rs
│   ├── lock_manager.rs # Locking protocols
│   └── mvcc.rs         # Multiversion concurrency control
├── types/              # Data types and serialization
│   ├── mod.rs
│   ├── value.rs        # Cell values (INTEGER, TEXT, etc.)
│   └── schema.rs       # Table schemas
├── sql/                # SQL processing
│   ├── mod.rs
│   ├── lexer.rs        # Tokenizer
│   ├── parser.rs       # Recursive descent parser
│   └── ast.rs          # Abstract syntax tree
├── planner/            # Query planning and optimization
│   ├── mod.rs
│   ├── logical_plan.rs # Logical operators
│   ├── physical_plan.rs # Physical execution strategies
│   └── optimizer.rs    # Cost-based optimization
├── executor/           # Query execution engine
│   ├── mod.rs
│   ├── operator.rs     # Operator trait and base types
│   ├── scan.rs         # Table and index scans
│   ├── join.rs         # Join implementations
│   └── aggregate.rs    # Aggregation operators
└── engine.rs           # Top-level database engine API
```

This structure is intentionally layered. The storage module knows nothing about SQL or transactions. The buffer pool module depends only on storage for page I/O. The transaction manager depends on buffer pool and logging. The SQL modules depend on types and schema information but not directly on storage internals. These boundaries prevent circular dependencies and make

each component independently testable.

Create the examples directory for our CLI tool:

```
mkdir examples
touch examples/cli.rs
```

Add a binary target to Cargo.toml:

```
[[bin]]
name = "mydb-cli"
path = "examples/cli.rs"
```

Now let us write the library root. Create src/lib.rs:

```
//! mydb - A relational database engine built from scratch.
//!
//! This crate implements a complete database storage engine with support for
//! persistent storage, buffered I/O, write-ahead logging, B-tree indexing,
//! transactional concurrency control, and SQL query processing.

pub mod storage;
pub mod buffer_pool;
pub mod log;
pub mod index;
pub mod transaction;
pub mod types;
pub mod sql;
pub mod planner;
pub mod executor;
pub mod engine;

// Re-export commonly used types at the crate root for convenience.
pub use engine::DatabaseEngine;
pub use types::value::Value;
pub use types::schema::{Schema, ColumnType};
```

Each module starts empty and grows as we implement its functionality. This forward declaration ensures the project compiles throughout development even before every component is fully implemented. We will fill in each module incrementally across chapters.

Core Abstractions: Pages, Records, and Offsets

Before writing any storage code, let us define the core abstractions that will appear throughout the entire engine. These concepts are fundamental to how databases think about data organization.

A page is a fixed-size block of bytes that serves as the atomic unit of I/O between memory and disk. Our implementation uses 4096-byte pages, matching the typical filesystem block size and CPU cache line alignment. This choice is not arbitrary: reading or writing in multiples of the filesystem block size avoids partial-block operations that waste bandwidth. Every page has a unique page ID within a database file, starting from zero.

Why do databases think in pages rather than individual records or raw bytes? The answer lies in hardware characteristics. Disk I/O has enormous latency compared to memory access: spinning disks require mechanical seeking and rotation (milliseconds per operation), while even fast SSDs introduce tens of microseconds of overhead per read. Memory access takes nanoseconds. If you read one byte at a time from disk, the overhead dominates completely. By reading in fixed-size pages, you amortize the I/O cost across many bytes of useful data. Furthermore, operating systems and storage drivers are optimized for block-sized transfers, so aligning with page boundaries gives you better performance.

Within each page lives structured data organized according to the page type. Data pages store table records using a slotted format: a header describing the page state, an array of slot pointers indicating where each record begins, and the actual record payloads packed into available space. Index pages store keys and child pointers arranged for efficient binary search. Special pages hold metadata like database headers, free space maps, or transaction logs. Each page type defines its own internal layout but shares the same fundamental size and I/O semantics.

A record (also called a row or tuple) represents a single logical entity stored in a table. Records can have variable length because they contain fields of different sizes: a fixed-width integer column occupies four bytes while a text column might hold anywhere from zero to thousands of bytes depending on the actual value. Variable-length records create complexity for storage management because you cannot compute record positions by simple arithmetic. Instead, databases use directory structures within pages (slot

arrays) that track where each record starts and how much space it occupies.

Offsets are integer positions within a page or file that locate specific data. Page offsets measure bytes from the beginning of a page: offset 0 is the first byte, offset 4095 is the last byte of a 4096-byte page. File offsets measure bytes from the beginning of the database file and can be converted to page IDs by dividing by the page size. Understanding offsets is critical for implementing storage correctly because every read and write operation must specify exact positions.

Let us formalize these concepts in code. Create src/storage/page.rs:

```rust
//! Page abstraction for fixed-size blocks of data.
//!
//! Pages are the fundamental unit of I/O in mydb. Each page is PAGE_SIZE bytes,
//! identified by a unique PageId within a database file, and can hold structured
//! data depending on its type (data page, index page, metadata page, etc.).

use std::fmt;

/// Default page size: 4096 bytes, matching typical filesystem block size.
pub const PAGE_SIZE: usize = 4096;

/// Unique identifier for a page within a database file.
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct PageId(u64);

impl PageId {
    /// Create a new PageId from a raw number.
    pub fn new(id: u64) -> Self {
        Self(id)
    }

    /// Get the raw numeric ID.
    pub fn as_u64(&self) -> u64 {
        self.0
    }

    /// Compute the byte offset of this page within a database file.
    pub fn file_offset(&self) -> u64 {
        self.0 * PAGE_SIZE as u64
    }
}

impl fmt::Display for PageId {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "Page({})", self.0)
```

```
    }
}

/// A page buffer holding PAGE_SIZE bytes of data in memory.
///
/// This is the in-memory representation of a disk page. The buffer pool manager
/// allocates PageBuffer instances and fills them by reading from disk or
/// initializing with zeros for new pages. Higher-level components interpret
/// the bytes according to the appropriate page format (slotted page, B-tree
/// node, etc.).
#[derive(Clone)]
pub struct PageBuffer {
    /// The page ID this buffer represents.
    pub page_id: PageId,
    /// Raw byte contents of the page.
    pub data: [u8; PAGE_SIZE],
}

impl PageBuffer {
    /// Create a new empty page buffer filled with zeros.
    pub fn new(page_id: PageId) -> Self {
        Self {
            page_id,
            data: [0u8; PAGE_SIZE],
        }
    }

    /// Get a mutable slice of the page data.
    pub fn as_mut_slice(&mut self) -> &mut [u8] {
        &mut self.data
    }

    /// Get an immutable slice of the page data.
    pub fn as_slice(&self) -> &[u8] {
        &self.data
    }
}

impl fmt::Debug for PageBuffer {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.debug_struct("PageBuffer")
            .field("page_id", &self.page_id)
            .field("data_len", &PAGE_SIZE)
            .finish()
    }
}

/// Types of pages that can exist in a database file.
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum PageType {
```

```rust
    /// Unknown or uninitialized page.
    Unused,
    /// Database header and metadata (page 0).
    Header,
    /// Data page storing table records using slotted format.
    Data,
    /// Internal B-tree node page (non-leaf index page).
    IndexInternal,
    /// Leaf B-tree node page (stores actual index entries).
    IndexLeaf,
}

impl PageType {
    /// Encode this page type as a single byte for storage in page headers.
    pub fn as_byte(self) -> u8 {
        match self {
            Self::Unused => 0x00,
            Self::Header => 0x01,
            Self::Data => 0x10,
            Self::IndexInternal => 0x20,
            Self::IndexLeaf => 0x30,
        }
    }

    /// Decode a page type from a stored byte value.
    pub fn from_byte(byte: u8) -> Self {
        match byte {
            0x01 => Self::Header,
            0x10 => Self::Data,
            0x20 => Self::IndexInternal,
            0x30 => Self::IndexLeaf,
            _ => Self::Unused,
        }
    }
}
```

This code establishes the foundational types. PageId wraps a u64 to prevent accidental misuse of raw integers throughout the codebase. PageBuffer holds exactly PAGE_SIZE bytes and tracks which page it represents. PageType enumerates the kinds of pages we support with byte-level encoding for on-disk storage. Every subsequent storage component will use these types consistently.

Create src/storage/mod.rs:

```rust
//! Storage layer: manages database files, pages, and disk I/O.
//!
//! This module provides the lowest level abstractions in mydb. It handles
//! opening and creating database files, reading and writing pages at specific
//! offsets, tracking file size, and managing free space for new pages.

pub mod file_manager;
pub mod page;

pub use page::{PageId, PageBuffer, PageType, PAGE_SIZE};
pub use file_manager::FileManager;
```

Error Handling Strategy Across the Engine

Database engines must handle errors carefully because failures can corrupt data or lose transactions if mishandled. Rust's type system gives us excellent tools for this: Result types force callers to consider error cases explicitly, and enum-based error types let us distinguish between recoverable and fatal conditions.

We will use a two-layer error strategy. At the library level, modules define specific error enums using thiserror that describe what can go wrong in that domain. At the application level (CLI tool, test harnesses), we use anyhow::Result for ergonomic error propagation without repeating boilerplate type annotations everywhere.

Create src/storage/file_manager.rs with our first concrete storage implementation:

```rust
//! Database file management: open, create, read, write pages to disk.
//!
//! The FileManager is responsible for all direct interaction with the database
//! file on disk. Higher-level components never open files themselves; they
//! request page reads and writes through this interface.

use std::fs::{File, OpenOptions};
use std::io::{Read, Seek, SeekFrom, Write};
use std::path::{Path, PathBuf};
use thiserror::Error;

use super::page::{PageBuffer, PageId, PageType, PAGE_SIZE};

/// Errors that can occur during file management operations.
#[derive(Error, Debug)]
```

```rust
pub enum FileManagerError {
    /// A low-level I/O error occurred.
    #[error("I/O error: {0}")]
    Io(#[from] std::io::Error),

    /// Attempted to access a page beyond the current file size.
    #[error("Page {page_id} does not exist in database")]
    PageNotFound { page_id: PageId },

    /// Database file is corrupted or has invalid header.
    #[error("Database file corrupted: {0}")]
    Corrupted(String),

    /// File path or permissions error.
    #[error("File access error: {0}")]
    FileAccess(String),
}

/// Manages the database file on disk.
///
/// The FileManager maintains an open handle to the database file and provides
/// methods for reading and writing pages at specific offsets. It tracks the
/// current file size in pages so that new pages can be allocated by extending
/// the file. All I/O operations go through this struct to ensure consistency
/// and enable future optimizations like memory-mapped access or async I/O.
pub struct FileManager {
    /// The underlying file handle.
    file: File,
    /// Path to the database file for error reporting.
    path: PathBuf,
    /// Current number of pages in the database file.
    num_pages: u64,
}

impl FileManager {
    /// Open an existing database file or create a new one if it does not exist.
    ///
    /// If creating a new file, initializes page 0 with a valid database header.
    pub fn open_or_create<P: AsRef<Path>>(path: P) -> Result<Self,
↪   FileManagerError> {
        let path = path.as_ref().to_path_buf();
        let exists = path.exists();

        let file = OpenOptions::new()
            .read(true)
            .write(true)
            .create(true)
            .truncate(false)
            .open(&path)?;

```

```rust
        let metadata = file.metadata()?;
        let file_size = metadata.len();

        // Calculate number of complete pages in the file.
        let num_pages = if file_size == 0 {
            0
        } else {
            (file_size + PAGE_SIZE as u64 - 1) / PAGE_SIZE as u64
        };

        let mut manager = Self {
            file,
            path,
            num_pages,
        };

        if !exists || file_size == 0 {
            // Initialize a new database with a header page.
            manager.init_header()?;
        } else {
            // Validate existing header.
            manager.validate_header()?;
        }

        Ok(manager)
    }

    /// Initialize page 0 with the database header structure.
    fn init_header(&mut self) -> Result<(), FileManagerError> {
        let mut buffer = PageBuffer::new(PageId::new(0));

       // Write magic number: "MYDB" followed by version byte and format flags.
       // This allows us to identify valid database files and detect corruption.
        buffer.data[0..4].copy_from_slice(b"MYDB");
        buffer.data[4] = 1; // Format version
        buffer.data[5] = PageType::Header.as_byte();

        // Store page size as big-endian u32 at offset 8.
        let page_size_bytes = (PAGE_SIZE as u32).to_be_bytes();
        buffer.data[8..12].copy_from_slice(&page_size_bytes);

        // Initialize page count to 1 (just the header page).
        let page_count_bytes: [u8; 8] = 1u64.to_be_bytes();
        buffer.data[12..20].copy_from_slice(&page_count_bytes);

        // Write header page to disk.
        self.write_page(PageId::new(0), &buffer)?;
        self.num_pages = 1;

        Ok(())
```

```
115     }
116
117     /// Validate that the database file has a correct header.
118     fn validate_header(&mut self) -> Result<(), FileManagerError> {
119         let buffer = self.read_page(PageId::new(0))?;
120
121         // Check magic number.
122         if &buffer.data[0..4] != b"MYDB" {
123             return Err(FileManagerError::Corrupted(
124                 "Invalid magic number: not a mydb database file".to_string(),
125             ));
126         }
127
128         // Check version.
129         if buffer.data[4] != 1 {
130             return Err(FileManagerError::Corrupted(format!(
131                 "Unsupported format version: {}",
132                 buffer.data[4]
133             )));
134         }
135
136         // Verify page size matches our expectation.
137         let stored_page_size =
    ↪ u32::from_be_bytes(buffer.data[8..12].try_into().unwrap());
138         if stored_page_size as usize != PAGE_SIZE {
139             return Err(FileManagerError::Corrupted(format!(
140                 "Page size mismatch: file uses {} bytes, expected {}",
141                 stored_page_size, PAGE_SIZE
142             )));
143         }
144
145         // Read current page count from header.
146         let stored_count =
    ↪ u64::from_be_bytes(buffer.data[12..20].try_into().unwrap());
147         self.num_pages = stored_count.max(self.num_pages);
148
149         Ok(())
150     }
151
152     /// Read a page from disk into a PageBuffer.
153     ///
154     /// Returns an error if the page ID is beyond the current file size.
155     pub fn read_page(&mut self, page_id: PageId) -> Result<PageBuffer,
    ↪ FileManagerError> {
156         let offset = page_id.file_offset();
157         let max_offset = self.num_pages * PAGE_SIZE as u64;
158
159         if offset + PAGE_SIZE as u64 > max_offset {
160             return Err(FileManagerError::PageNotFound { page_id });
161         }
```

```
162
163         let mut buffer = PageBuffer::new(page_id);
164         self.file.seek(SeekFrom::Start(offset))?;
165         self.file.read_exact(&mut buffer.data)?;
166
167         Ok(buffer)
168     }
169
170     /// Write a page buffer to disk at its designated offset.
171     pub fn write_page(&mut self, page_id: PageId, buffer: &PageBuffer) ->
    ↪ Result<(), FileManagerError> {
172         let offset = page_id.file_offset();
173         self.file.seek(SeekFrom::Start(offset))?;
174         self.file.write_all(&buffer.data)?;
175
176         // Track file growth if we wrote a new page at the end.
177         let new_page_count = page_id.as_u64() + 1;
178         if new_page_count > self.num_pages {
179             self.num_pages = new_page_count;
180             self.update_page_count_in_header()?;
181         }
182
183         Ok(())
184     }
185
186     /// Update the page count stored in the header.
187     fn update_page_count_in_header(&mut self) -> Result<(), FileManagerError> {
188         let mut buffer = self.read_page(PageId::new(0))?;
189         let count_bytes: [u8; 8] = self.num_pages.to_be_bytes();
190         buffer.data[12..20].copy_from_slice(&count_bytes);
191         self.write_page(PageId::new(0), &buffer)?;
192         Ok(())
193     }
194
195     /// Allocate a new page at the end of the file.
196     ///
197     /// Returns the PageId of the newly allocated page and a zero-filled buffer.
198     pub fn allocate_page(&mut self) -> Result<(PageId, PageBuffer),
    ↪ FileManagerError> {
199         let page_id = PageId::new(self.num_pages);
200         let buffer = PageBuffer::new(page_id);
201
202         // Extend file by writing the new zero-filled page.
203         self.write_page(page_id, &buffer)?;
204
205         Ok((page_id, buffer))
206     }
207
208     /// Get the current number of pages in the database.
209     pub fn num_pages(&self) -> u64 {
```

```
210         self.num_pages
211     }
212
213     /// Flush all buffered data to disk using fsync.
214     ///
215     /// This ensures that all writes are durable on stable storage, surviving
216     /// a system crash or power loss. Called at transaction commit and during
217     /// checkpoint operations.
218     pub fn sync(&self) -> Result<(), FileManagerError> {
219         self.file.sync_all()?;
220         Ok(())
221     }
222
223     /// Get the file path for this database.
224     pub fn path(&self) -> &Path {
225         &self.path
226     }
227 }
```

Let us walk through this implementation carefully because every design decision here matters.

The FileManager struct holds a single File handle that remains open for the lifetime of the database connection. Opening and closing files repeatedly would be prohibitively expensive. The path field stores the file location for error messages and future operations like WAL file naming. The num_pages field tracks how many pages exist in the file so we can allocate new pages sequentially without scanning or seeking.

The open_or_create method handles both cases: opening an existing database validates its header to detect corruption, while creating a new one initializes a proper header structure. This dual behavior mirrors real databases like SQLite that create files transparently when connecting to non-existent paths.

The header format encodes several critical pieces of information. The magic number “MYDB” identifies valid database files and immediately rejects corrupted or incompatible files. The version byte allows future format evolution: if we change the file layout, we bump the version and refuse to open older formats without migration. The page size field enables cross-platform compatibility: different databases could theoretically use different page sizes, and we verify that our expected size matches what the file declares. The page count lets us quickly determine where new pages should be allocated.

All multi-byte integers in the header use big-endian byte order. This is a

deliberate choice for portability: the same database file can be read correctly on both little-endian machines (most Intel and AMD CPUs) and big-endian systems without byte-swapping. SQLite makes the same choice for exactly this reason [4].

The read_page method seeks to the correct offset within the file and reads exactly PAGE_SIZE bytes using read_exact, which fails if fewer bytes are returned (indicating truncation or corruption). The write_page method performs the reverse operation and updates the page count if we extended the file. Both methods operate on exact byte offsets computed from PageId.file_offset(), ensuring no ambiguity about where data lives.

The sync method calls fsync through Rust's sync_all, which flushes both kernel page cache buffers and filesystem metadata to physical storage. This is essential for durability guarantees: without fsync, writes might appear successful in user space while still sitting in volatile RAM that would be lost on power failure. We will call this at critical points during transaction commits.

First Binary: Initializing an Empty Database File

Now let us write our first executable program: a simple CLI tool that creates and inspects database files. This gives us something runnable to test our storage layer immediately.

Create examples/cli.rs:

```
1  //! Command-line interface for mydb.
2  //!
3  //! Usage:
4  //!   mydb-cli init <path>          Create a new database file
5  //!   mydb-cli info <path>          Show database file information
6  //!   mydb-cli shell <path>         Start interactive SQL shell (future)
7
8  use std::env;
9  use std::process;
10
11 use mydb::storage::{FileManager, PAGE_SIZE};
12
13 fn main() {
14     env_logger::init();
15
16     let args: Vec<String> = env::args().collect();
17
```

```
    if args.len() < 2 {
        eprintln!("Usage: mydb-cli <command> [arguments]");
        eprintln!("Commands: init, info");
        process::exit(1);
    }

    let command = &args[1];

    match command.as_str() {
        "init" => cmd_init(&args),
        "info" => cmd_info(&args),
        _ => {
            eprintln!("Unknown command: {}", command);
            process::exit(1);
        }
    }
}

/// Initialize a new database file at the given path.
fn cmd_init(args: &[String]) {
    if args.len() < 3 {
        eprintln!("Usage: mydb-cli init <path>");
        process::exit(1);
    }

    let path = &args[2];

    match FileManager::open_or_create(path) {
        Ok(mut fm) => {
            // Sync to ensure header is durable.
            if let Err(e) = fm.sync() {
                eprintln!("Error syncing database: {}", e);
                process::exit(1);
            }

            println!("Database initialized at {}", path);
            println!("Page size: {} bytes", PAGE_SIZE);
            println!("Pages allocated: {}", fm.num_pages());
        }
        Err(e) => {
            eprintln!("Error initializing database: {}", e);
            process::exit(1);
        }
    }
}

/// Display information about an existing database file.
fn cmd_info(args: &[String]) {
    if args.len() < 3 {
        eprintln!("Usage: mydb-cli info <path>");
```

```
        process::exit(1);
    }

    let path = &args[2];

    match FileManager::open_or_create(path) {
        Ok(fm) => {
            println!("Database file: {}", path);
            println!("Page size: {} bytes", PAGE_SIZE);
            println!("Pages: {}", fm.num_pages());
            println!(
                "File size: {} bytes ({:.2} KB)",
                fm.num_pages() * PAGE_SIZE as u64,
                (fm.num_pages() as f64 * PAGE_SIZE as f64) / 1024.0
            );
        }
        Err(e) => {
            eprintln!("Error reading database: {}", e);
            process::exit(1);
        }
    }
}
```

Build and run the project:

```
cargo build --release
./target/release/mydb-cli init test.db
./target/release/mydb-cli info test.db
```

Expected output:

```
Database initialized at test.db
Page size: 4096 bytes
Pages allocated: 1

Database file: test.db
Page size: 4096 bytes
Pages: 1
File size: 4096 bytes (4.00 KB)
```

You now have a working database file with a valid header structure. If you inspect it with a hex editor, you will see the “MYDB” magic number at offset 0, followed by version and metadata fields exactly as specified in our format. This small program already demonstrates core database behaviors: file creation, header initialization, validation on open, and durability through fsync.

In the next chapter we will dive deeper into storage fundamentals: how disks and SSDs actually work, why page-based I/O matters for performance, binary serialization details including endianness and alignment, and building a complete storage manager interface that higher layers can rely on.

Chapter 3: Storage Fundamentals — Disks, SSDs, and Pages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

How Disks and SSDs Actually Work

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Why Databases Think in Pages, Not Bytes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Building Our Page Abstraction Layer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Binary Data Layout: Endianness, Alignment, and Serialization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Implementing the Storage Manager Interface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Chapter 4: File Format and Slotted Page Records

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Designing the Database File Format

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Slotted Page Layout: Headers, Directories, and Records

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Variable-Length Record Serialization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Free-Space Management Within Pages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Implementing Page Reads and Writes with Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Chapter 5: Buffer Pool Manager — Caching at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

The Purpose of Buffer Pools and Why They Are Hard

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Pinning Semantics and Reference Counting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Dirty Pages, Write Policies, and Flush Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Handling Memory Pressure and Eviction Under Load

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Chapter 6: Logging, Durability, and Crash Recovery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

The Durability Problem and Why fsync Matters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Write-Ahead Logging: Protocol and Invariants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Designing Our Log Record Format

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Checkpointing Strategies and Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Crash Recovery: Redo and Undo in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Chapter 7: Indexing — B-Trees, Hash Indexes, and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Why Indexes Exist and What They Trade Away

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Implementing a B+ Tree from Scratch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Page Splits, Merges, and Concurrency Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Hash Indexes for Equality Lookups

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

LSM Trees and Alternative Index Architectures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Chapter 8: Transactions and Concurrency Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

ACID Properties: What They Mean in Implementation Terms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Locking Protocols and Two-Phase Locking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Deadlock Detection and Prevention

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Latches vs Locks: Protecting Data Structures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Implementing MVCC with Snapshot Isolation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Chapter 9: SQL Parsing and Semantic Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Designing Our SQL Subset and Grammar

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Lexical Analysis: Tokenizing SQL Input

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Recursive Descent Parsing and AST Construction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Symbol Tables and Schema Resolution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Chapter 10: Query Planning and Cost-Based Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

From AST to Logical Plan

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Physical Operators and Execution Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Building a Cost Model from Statistics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Cardinality Estimation: Histograms and Selectivity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Join Ordering and Query Rewrite Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Chapter 11: Query Execution Engine

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Operator Architecture: Iterators and Pipelining

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Table Scans and Index Scans

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Filtering, Projection, and Expression Evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Join Implementations: Nested-Loop, Hash, and Merge

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Aggregation and Grouping with Hash Tables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Chapter 12: Performance Engineering and Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Understanding Database Performance Characteristics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

I/O Patterns: Sequential Versus Random Access

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Cache Efficiency and Data Locality

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Vectorized Execution vs Tuple-at-a-Time Processing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Profiling, Benchmarking, and Workload Generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Systematic Optimization: Finding and Fixing Bottlenecks

Chapter 13: Reliability, Correctness, and Failure Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Crash Consistency: Guarantees and Invariants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Torn Pages and Partial Writes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Corruption Detection with Checksums and Validation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Disk-Full Conditions and Safe Failure Modes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Backup, Restore, and Integrity Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Chapter 14: Beyond Single Node — Distributed Database Concepts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Why Single-Node Databases Eventually Hit Limits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Replication Architectures: Leader-Follower and Multi-Leader

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Log Replication and Consistency Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Sharding and Distributed Query Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Consensus, Fault Tolerance, and Rebalancing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Chapter 15: Conclusion — From Educational Engine to Production System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Architecture Review: How All Subsystems Fit Together

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

What We Simplified and Why Production Systems Differ

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Comparing Our Implementation to Real Databases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

Paths Forward: Extending Your Database Engine

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

The Engineer's Mindset for Building Reliable Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingdatabaseenginesfromscratch.