REVERSE ENGINEERING DATABASES AND STORAGE FORMATS

From Binary Bytes to Complete Understanding

SQLite | PostgreSQL | InnoDB | LevelDB | Redis

C | PYTHON | RUST | GO

KEVIN MALLONE

Reverse Engineering Databases and Storage Formats

From Binary Bytes to Complete Understanding

Steve Publications

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

This version was published on 2026-09-07

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

From Binary Bytes to Complete Understanding

This book teaches you how to systematically reverse engineer any database or binary storage format from scratch. Starting from the fundamentals of hexadecimal analysis and binary file structures, you will progress through page-oriented storage, B-tree indexes, write-ahead logging, multi-version concurrency control, crash recovery and advanced dynamic instrumentation techniques. Every concept is illustrated with complete, runnable code examples in C, Python, Rust and Go, using real-world database formats including SQLite, PostgreSQL, InnoDB, LevelDB and Redis as case studies. By the end, you will be able to take an unfamiliar binary database file, discover its structure through controlled experimentation, document its format and implement a compatible reader or analysis tool.

Chapter 1: Why Reverse Engineer Storage Formats

The Data Archaeology Problem

A hospital in Ohio lost its primary database server during a maintenance window. The backup tapes had degraded. The vendor had been acquired three years ago and no longer supported the product. Inside a corrupted RAID array, however, was a 400 GB file containing years of patient records. The file system knew it was a file, but no one knew what format it used. The hospital needed the data, and the only way to get it was to figure out how the binary storage format worked.

This scenario is not hypothetical. Data archaeology is the practice of recovering information from systems whose formats are partially or completely undocumented, whose software is no longer available, or whose behavior no one remembers. It happens when companies go bankrupt, when proprietary software becomes orphaned, when migrations stall, when corruption strikes without warning and when forensic investigators need to extract evidence from systems they cannot trust.

Database and storage formats are at the center of this problem because they are the last line of defense between data and oblivion. Applications fail. Servers crash. Formats evolve. But the bytes on disk, if you can read them, contain the original truth of what was stored. Reverse engineering those bytes is often the only path to recovery.

Recovery and Forensics: When Backup Fails

Backups exist, and most systems have them. But backups are not infallible. They can be missing, corrupted, incomplete or restored to the wrong point in time. When backup fails, your next option is often to read directly from the storage format, bypassing the application layer entirely.

Consider a database that has suffered partial disk corruption. The application refuses to open it because the schema cannot be read or the integrity checks fail. Yet much of the data may still be intact. If you understand the underlying page structure, you can extract every readable record, reconstruct the schema from the surviving metadata and rebuild a clean database from the salvageable material. This approach is not a last resort. It is a standard procedure in forensic data recovery.

Digital forensics adds another dimension. Investigators need to answer questions that the database management system is not designed to answer. Did a row exist before it was deleted? What was its value? When was it modified? Who deleted it? The answers are often encoded in the physical storage layout: in free pages that have not yet been overwritten, in write-ahead logs that record every change, in transaction metadata that timestamps every modification. A forensic examiner who can read the raw storage format can see the history of the database, not just its current state.

SQLite is an especially important case. It is the embedded database behind Android applications, web browsers, operating system components, IoT devices and millions of embedded systems. When investigators examine an Android phone, they are often examining SQLite files: browser history, call logs, message databases and application state. Understanding SQLite storage is not optional for mobile forensics. It is a core skill [1].

Interoperability and Lock-In Escape

Reverse engineering storage formats is not only about recovery and forensics. It is also about interoperability.

Vendors sometimes design their formats to be opaque. This is a deliberate business strategy. When you cannot read the files, you must continue to use their software. When you cannot export data in a standard format, you cannot easily leave. Lock-in is profitable for vendors but dangerous for customers.

Reading the format directly breaks lock-in. If you can parse the storage structure, you can write your own import and export tools. You can build alternative clients. You can create migration paths. You can run analytical queries without going through the vendor application.

This is how open-source alternatives are often built. When a proprietary format is reverse engineered, someone writes a compatible reader. That

reader becomes the foundation for a new ecosystem. This has happened repeatedly in the history of software: OLE2 documents, PostScript, TrueType fonts and countless database formats all became interoperable through reverse engineering.

There is also the pragmatic case. You have data in a format that no one maintains anymore. You need to read it into a modern system. You do not care about the vendor application. You care about the data inside the file. Reverse engineering the format is the most direct path.

Security Auditing of Storage

Every database stores sensitive data. Credentials, personal information, financial records, health data. How securely does it store them? Is encryption really enabled? Are deleted records truly unrecoverable? Are there metadata leaks in the file structure that reveal information the application claims to hide?

Security auditing often requires looking at the actual bytes, not trusting what the application tells you. A database might claim to encrypt its data, but if the encryption is disabled and the application never checks, the data sits in plaintext on disk. A database might claim to support secure deletion, but if records are only marked as deleted and not overwritten, they remain recoverable. A database might leak information through index structures, WAL files, or temporary files that persist after the main data is cleaned.

Reverse engineering helps answer these questions. By reading the format, you can verify whether encryption is actually applied, whether deleted data is truly gone, whether authentication tokens are stored in plaintext and whether side channels exist in the storage layout. These are not academic concerns. They directly affect whether a system is secure.

Some of the most notable security discoveries have come from storage format analysis. Researchers have found credentials in browser database files, discovered that deleted files remained readable in application databases and exposed security flaws by examining how data was laid out on disk. The pattern is consistent: look at the bytes, not the UI.

Performance Debugging at the Byte Level

Performance problems are often mysteries until you look at the storage layer.

Why is a query slow? Why is the database file growing so fast? Why is fragmentation so severe? Why does compaction consume all the CPU? Why are writes unexpectedly slow?

The answers are in the storage structures. A database might have fragmented pages that force unnecessary I/O. It might be using an index that does not match the access pattern. It might be creating too many small files. It might be writing to disk more often than necessary. It might be caching incorrectly. None of these problems are visible from the application layer. They become obvious when you examine the physical layout of the data.

Database engineers who understand storage formats can diagnose problems that others cannot. They can tell you why a table is performing poorly by examining its page structure. They can predict how changes to the schema will affect performance by understanding the underlying index layout. They can optimize file sizes by controlling page and index behavior. This is not black magic. It is simply reading the bytes that represent your data.

A Roadmap for the Book

This book is organized as a progressive journey from fundamentals to advanced techniques. Each chapter builds on previous material, and the examples are designed to work together as a coherent learning path.

Chapters 2 through 4 establish the foundations. You will learn how to read binary data at the byte level, how to design simple file formats with headers and record structures and how databases serialize types including integers, strings, floats and complex nested structures. These chapters introduce the basic skills: hexadecimal analysis, endianness, encoding and struct packing.

Chapters 5 through 9 cover the core storage structures. You will understand why databases use fixed-size pages, how records are packed into those pages, how free space is managed and how indexes are organized as B-trees, LSM-trees and hash structures. These chapters use SQLite and LevelDB as running case studies.

Chapters 10 through 14 address transactions and durability. Write-ahead logging, journaling, crash recovery and the ACID properties are explained in terms of their on-disk implementation. You will learn how the ARIES recovery algorithm works and how to perform manual recovery on a damaged database.

Chapters 12 through 13 cover concurrency control: locking, multi-version concurrency control, snapshots and isolation levels. PostgreSQL serves as the primary example, with its MVCC implementation providing a concrete view of how version chains and visibility maps work.

Chapters 15 through 18 examine operational concerns: caching, buffer pools, compaction, fragmentation, compression and corruption detection. These chapters explain the mechanisms behind performance tuning and reliability.

Chapters 19 through 21 present the toolset: hex editors, debuggers, system call tracers, dynamic instrumentation with eBPF, fuzzing and differential analysis. You will learn how to observe database behavior at runtime and correlate it with on-disk structures.

Chapters 22 and 23 synthesize everything into a complete reverse engineering workflow. Starting with a mystery SQLite database file, you will walk through the entire process: initial reconnaissance, hypothesis formation, controlled experimentation, decoding structures, validation, documentation and parser implementation.

Chapter 24 provides additional case studies across InnoDB, PostgreSQL, LevelDB and Redis, showing common patterns and notable differences.

Chapter 25 addresses the legal and ethical landscape, ensuring that your technical work is conducted responsibly and within the bounds of the law.

Throughout the book, every code example is complete and runnable. The examples are internally consistent and can be reproduced on a modern Linux environment. They are not exercises or homework. They are working tools that demonstrate the concepts and can be extended into real projects.

The book does not assume you already know database internals. It does assume you know how to program. If you are comfortable with C, Python, or Rust, and you are curious about what happens beneath the database abstraction layer, this book will teach you the rest.

Chapter 2: Reading Binary: First Principles

Hexadecimal Notation and Byte-Level Thinking

Before you can reverse engineer a storage format, you must be comfortable reading binary data. The first step is developing a mental model for thinking about files at the byte level.

A file is simply a sequence of bytes, where each byte is an 8-bit value ranging from 0 to 255 in decimal, or from 0x00 to 0xFF in hexadecimal. Hexadecimal is the standard notation for binary analysis because it maps cleanly to bytes: each hex digit represents 4 bits, so two hex digits represent exactly one byte. The value 0xFF is 255 in decimal, 0x01 is 1, and 0x41 is 65, which is the ASCII code for the uppercase letter A.

When you open a binary file in a hex editor, you see a grid. The left column shows the offset in hex. The center columns show the bytes in hex. The right column shows an ASCII representation where printable characters appear as themselves and non-printable bytes appear as dots.

The SQLite database header begins with these bytes [1]:

```
Offset   Hex Dump                                         ASCII
00000000 53 51 4C 69 74 65 20 66 6F 72 6D 61 74 20 33 00  SQLite format 3.
```

Each hex pair corresponds to a character: 53 is 'S', 51 is 'Q', 4C is 'L', and so on. The final byte 00 is a null terminator, which is not printable and appears as a dot.

Developing the habit of reading hex is essential. You must become comfortable looking at a hex dump and recognizing patterns: the magic strings that identify file types, the repeating structures that suggest arrays or headers, the regions of zeros that indicate padding or unused space and the mixed patterns that suggest structured data versus random or compressed content.

Endianness and Multi-Byte Numbers

Endianness is the order in which bytes are stored for multi-byte values such as 16-bit integers, 32-bit integers, 64-bit integers and floating-point numbers. The term comes from Gulliver's Travels and describes the byte ordering convention of a system.

In big-endian order, the most significant byte is stored at the lowest address. The 32-bit value 0x12345678 would be stored as bytes 12, 34, 56, 78 in memory order. This matches the way humans write numbers: most significant digit first.

In little-endian order, the least significant byte is stored at the lowest address. The same value 0x12345678 would be stored as bytes 78, 56, 34, 12 in memory order. This seems backwards but is the native byte order of x86 and x86-64 processors, which means it is the default order on most desktop and server computers.

Endianness matters enormously in format reverse engineering because you must determine which convention the format uses before you can read multi-byte values correctly.

To test whether a particular region of a file uses little-endian or big-endian encoding, look for values that you can infer from context. If you see a page size field and you expect a reasonable value like 4096 (0x1000), you can check which byte ordering produces that value. Consider these two sequences:

```
Option A: 00 10    (big-endian: 0x0010 = 16)
Option B: 10 00    (little-endian: 0x0010 = 16)
```

If you know the value should be 4096, then the sequence 00 10 00 00 is little-endian (because reading it as little-endian gives 0x00001000 = 4096), while the sequence 00 00 10 00 is big-endian (because reading it as big-endian gives 0x00001000 = 4096).

Here is a simple Python function that demonstrates both interpretations:

```python
import struct

data = bytes([0x78, 0x56, 0x34, 0x12])

# Little-endian interpretation
le_value = struct.unpack('<I', data)[0]
print(f"Little-endian: {le_value} (0x{le_value:X})")

# Big-endian interpretation
be_value = struct.unpack('>I', data)[0]
print(f"Big-endian: {be_value} (0x{be_value:X})")
```

Output:

```
Little-endian: 305419896 (0x12345678)
Big-endian: 2018915346 (0x78563412)
```

The same four bytes produce two completely different 32-bit integers. If you read a format with the wrong endianness assumption, every multi-byte value will be wrong.

A practical heuristic for determining endianness in an unknown format: look at the values you see and ask which interpretation produces reasonable numbers. Page sizes, record counts, version numbers and offsets should all be plausible values. A page size of 4096 is reasonable. A page size of 1073741824 (one gigabyte) is not. If one endianness produces mostly reasonable values and the other produces mostly garbage, you have found your answer.

Encoding Schemes: ASCII, UTF-8 and Binary Encodings

Text in binary formats is always encoded. The question is: in which encoding?

ASCII is the simplest encoding: each byte represents one character, and bytes 0x20 through 0x7E are the printable characters including letters, digits, punctuation and space. If you see a hex dump containing values in this range with recognizable words, you are looking at ASCII text.

UTF-8 is a variable-length encoding that is backward compatible with ASCII. The first 128 code points (0x00 to 0x7F) are encoded identically to ASCII. Characters beyond U+007F use 2 to 4 bytes each, with specific byte patterns that make the encoding self-delimiting and error-resilient.

- 1-byte characters: 0xxxxxxx (0x00 to 0x7F, ASCII)
- 2-byte characters: 110xxxxx 10xxxxxx (U+0080 to U+07FF)
- 3-byte characters: 1110xxxx 10xxxxxx 10xxxxxx (U+0800 to U+FFFF)
- 4-byte characters: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx (U+10000 to U+10FFFF)

If you see bytes where the high bit (0x80) is frequently set in patterns like 0xC2, 0xE2, 0xEF, you are likely looking at UTF-8. The byte 0xC2 starts the 2-byte sequence for characters like the Euro sign (0xC2 0xAC). The byte 0xE2 starts many 3-byte sequences. The byte 0xEF is common for Unicode characters in the CJK range.

UTF-16 is used by some formats and is more complex because it has two variants: UTF-16BE and UTF-16LE. The distinction is signaled by a byte order mark at the start of the stream: 0xFEFF for big-endian or 0xFFFE for little-endian. If you see one of these patterns at the beginning of a text region, you have your endianness for UTF-16 encoded data.

Binary formats also use other encoding schemes for non-text data. Base64 encoding appears in some formats to embed binary data in text fields. Varint encoding, used by SQLite and Protocol Buffers, encodes integers in a variable number of bytes. We will study these in detail in Chapter 4.

The Anatomy of a Structured Binary File

Structured binary files follow patterns that are easy to recognize once you know what to look for. Most binary formats share a common anatomy:

A magic number or signature at the beginning identifies the file type. This is a fixed sequence of bytes that any reader can check to verify that it is looking at the correct format. PNG files start with 0x89 50 4E 47. ZIP files start with 50 4B 03 04. SQLite files start with the ASCII string “SQLite format 3\0”.

A header follows the magic number and contains metadata about the file: version number, global parameters, checksums, offsets to major sections. Headers are usually a fixed size so that readers can seek past them directly.

The body contains the actual data, organized according to the format’s rules. This might be an array of records, a sequence of pages, a hierarchical structure with pointers, or some combination.

Some formats include a trailer at the end: a checksum, a final record, or a footer that provides a reverse index into the file.

This structure exists in almost every binary format because it serves a practical purpose: the reader needs to know what it is reading and how to navigate the file. The magic number says what format to expect. The header says how to interpret the rest. The body contains the data. The trailer, if present, provides integrity checks or navigation aids.

When you encounter an unknown binary file, your first questions should be:

- What is the magic number? Does it identify a known format?
- What follows the magic number? Are there readable strings, version numbers, or other metadata?
- Is the file divided into fixed-size chunks that suggest pages or blocks?
- Are there recognizable patterns such as ASCII text, known integer values, or checksum patterns?
- Does the file end with a recognizable trailer or checksum?

Answering these questions is the starting point for every format reverse engineering project.

Essential Command-Line Tools

You do not need expensive commercial software to reverse engineer binary formats. A modern Linux system comes with everything you need.

The command xxd converts binary data to and from hex dumps. Running xxd on a file produces a hex dump similar to what you see in a hex editor:

```
$ xxd -l 128 database.sqlite
00000000: 5351 4c69 7465 2066 6f72 6d61 7420 3300  SQLite format 3.
00000010: 1000 0202 0040 2020 0000 0008 0000 0004  .....@  ........
```

The option -l limits output to the specified number of bytes. The option -s starts at a byte offset. The option -g controls grouping: -g 1 shows one byte per group, -g 2 shows two bytes per group, and so on.

The command hexdump is similar but offers more flexible formatting options:

```
$ hexdump -C -n 64 database.sqlite
00000000  53 51 4c 69 74 65 20 66  6f 72 6d 61 74 20 33 00  |SQLite format 3.|
00000010  10 00 02 02 00 40 20 20  00 00 00 08 00 00 00 04  |.....@  ........|
00000020  00 00 00 04 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

The command strings extracts readable ASCII and UTF-16 strings from a binary file. This is often the fastest way to discover text content:

```
$ strings -n 4 database.sqlite | head -20
SQLite format 3
sqlite_schema
CREATE TABLE
```

The option -n sets the minimum string length. Shorter minimums find more strings but produce more noise.

The command od (octal dump) is the traditional Unix tool for binary display. It supports many output formats including hexadecimal, decimal and octal:

```
$ od -A x -t x1z -N 32 database.sqlite
000000  53 51 4c 69 74 65 20 66 6f 72 6d 61 74 20 33 00  >SQLite format 3.<
000010  10 00 02 02 00 40 20 20 00 00 00 08 00 00 00 04  >.....@  ........<
000020
```

For interactive hex editing, bless is a free hex editor available on most Linux distributions. It displays the hex dump alongside an ASCII view, allows editing of bytes and supports search, hex calculation and custom formatting.

A First Binary Inspection

Let us perform a systematic inspection of a SQLite database file using the tools described above. This exercise demonstrates the pattern of analysis you will use for any binary file.

First, create a small SQLite database with a simple table:

```
sqlite3 test.db <<EOF
CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT, email TEXT);
INSERT INTO users VALUES (1, 'Alice', 'alice@example.com');
INSERT INTO users VALUES (2, 'Bob', 'bob@example.com');
EOF
```

Now examine the file:

```
$ file test.db
test.db: SQLite 3.x database

$ ls -la test.db
-rw-r--r-- 1 user user 8192 Sep 7 10:00 test.db

$ xxd -l 128 test.db
```

The output begins:

```
00000000: 5351 4c69 7465 2066 6f72 6d61 7420 3300  SQLite format 3.
00000010: 1000 0202 0040 2020 0000 0008 0000 0004  .....@  ........
00000020: 0000 0004 0000 0000 0000 0000 0000 0001  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0001 0000 0004 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0003 002e 6c51  ..........lQ
00000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Observations from the first 128 bytes:

- Bytes 0 to 15 are the magic string "SQLite format 3" followed by a null byte. This confirms the file type.
- Bytes 16 to 17 contain 10 00, which is 0x1000 in little-endian: 4096 bytes. This is the page size. Our file is 8192 bytes, which is exactly 2 pages.
- The file is clearly structured with the first 100 bytes forming a header region, followed by data that appears to be zero-padded or minimal.

Now look at the strings:

```
$ strings -n 8 test.db
SQLite format 3
users
CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT, email TEXT)
```

The schema definition appears as a plaintext string inside the binary file. This is a common pattern: structured text is stored alongside binary metadata.

Finally, examine the second page to see where the data rows live:

```
$ xxd -s 4096 -l 256 test.db
```

Output:

```
00001000: 0d00 0007 002c 0778 073e 0700 06c8 0000  .....,.x>. ......
00001010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
...
000010c0: 0700 0301 0108 416c 6963 650c 616c 6963  ......Alice.alic
000010d0: 6540 6578 616d 706c 652e 636f 6d07 0003  e@example.com...
000010e0: 0101 0842 6f62 0862 6f62 4065 7861 6d70  ...Bob.bob@examp
000010f0: 6c65 2e63 6f6d                           le.com
```

The second page begins with the byte 0x0D. In SQLite, this indicates a leaf table page. The bytes 07 00 are the number of cells (7). The following 14 bytes are the cell pointer array, which tells SQLite where each row starts. Scanning down, we see "Alice" and "Bob" as plaintext strings at offset 0x10C8 and 0x10E0. The structure is visible in the raw bytes.

This is what format reverse engineering looks like at its simplest: read the bytes, recognize the patterns, understand the structure.

Developing the Right Mindset

Three principles guide every good binary analysis:

Hypothesis and evidence. Never state a conclusion without evidence. If you think a field represents a page size, verify that the value matches the observed page boundaries. If you think a sequence of bytes is a header, verify that it appears consistently across multiple files. Every claim must be testable.

Context and cross-validation. A single byte value means nothing in isolation. Its meaning comes from where it appears, what surrounds it and how it

relates to other values. Always look at context. Cross-validate observations across multiple samples.

Iteration. Reverse engineering is not linear. You make observations, form hypotheses, test them, refine them and repeat. The process is messy and non-linear. You will make wrong guesses and correct them. The important thing is to document your reasoning so that you can retrace your steps.

These principles are not just good practice. They are what separate systematic reverse engineering from random guessing.

With this foundation in place, you are ready to design your first binary format and then reverse engineer it, which is the subject of the next chapter.

Chapter 3: Simple File Formats: Headers, Magic Numbers and Metadata

Designing a Minimal File Format

To understand how storage formats work, the best approach is to design one. We will create a simple binary format for storing a collection of records, similar to the simplest possible database file.

The format will have the following structure:

- A 32-byte file header containing:
  - An 8-byte magic number to identify the format
  - A 4-byte version number
  - A 4-byte record count
  - A 4-byte fixed record size
  - An 8-byte timestamp when the file was written
  - A 4-byte CRC32 checksum of the header (not including the checksum field itself)
- A sequence of fixed-size records, each containing:
  - An 8-byte record ID (64-bit unsigned integer)
  - A 48-byte value field (used for storing data)
- No trailer. The end of the file is determined by the record count and record size in the header.

This is deliberately simple but demonstrates all the essential elements of a real storage format: magic number, versioning, metadata, checksums and a record body. The simplicity allows us to examine every byte and understand exactly how the structure maps to the file layout.

Magic Numbers and Format Identification

The magic number is the signature that identifies a file as belonging to a particular format. When an application receives a file, the first thing it does is check the magic number. If it matches, the application proceeds. If it does not match, the application rejects the file as corrupt or unknown.

Our magic number will be the 8-byte sequence: 0x52, 0x45, 0x43, 0x4F, 0x52, 0x44, 0x56, 0x31, which spells "RECORDV1" in ASCII.

Magic numbers can be ASCII strings or arbitrary byte sequences. ASCII strings are human-readable and easy to recognize in a hex dump. Arbitrary sequences are harder to forge accidentally but less convenient for manual inspection. Many formats combine both: SQLite uses an ASCII string, while JPEG uses the non-printable sequence 0xFF 0xD8 0xFF.

Good magic numbers have several properties: they are unique enough to avoid collisions with other formats, they are stable across versions of the format and they appear at a fixed position so that readers can check them without reading the entire file.

The choice of magic number affects how easy the format is to identify during reverse engineering. A recognizable ASCII string gives away the format immediately. An arbitrary sequence requires you to search documentation or databases of known signatures.

Headers and Global Metadata

The header contains information about the file as a whole: how many records it contains, what size each record is, what version of the format was used and when the file was created. This metadata allows a reader to validate the file, seek directly to records and handle version differences.

Here is the exact header layout for our format:

```
Offset  Size   Field            Type             Endian
------  ----   -----            ----             ------
00      8      magic            8 bytes          N/A
08      4      version          uint32           little-endian
0C      4      record_count     uint32           little-endian
10      4      record_size      uint32           little-endian
14      8      timestamp        uint64           little-endian
1C      4      checksum         uint32           little-endian
```

Total header size is 32 bytes. The fields at offsets 8 to 1C are stored in little-endian byte order, which is the native order on x86 and x86-64 systems and therefore the most common choice for binary formats.

The version field starts at 1. Future versions of the format can increment it, allowing readers to handle backward compatibility or reject unknown versions.

The record_count field tells the reader how many records follow the header. Combined with record_size, this determines the expected file size: file_size = 32 + record_count * record_size. Any deviation indicates truncation or corruption.

The timestamp is a Unix timestamp in seconds, recording when the file was written. This is useful for determining file age, ordering multiple files and debugging issues related to timing.

The checksum is a CRC32 computed over bytes 0 to 1B (the entire header except the checksum field itself). On reading, the reader recomputes the checksum and compares it to the stored value. A mismatch indicates that the header was corrupted.

Fixed-Width vs Variable-Length Records

Our format uses fixed-width records: each record is exactly 56 bytes (8 bytes for the ID plus 48 bytes for the value). Fixed-width records are simple to implement and allow direct calculation of record offsets: record_offset = header_size + record_index * record_size.

Variable-length records are more complex but more space-efficient when record sizes vary. They require either length prefixes (each record begins with its own length field) or a slot directory (a table of offsets to each record, as SQLite uses). Variable-length records are common in real database formats

because data sizes vary widely: some strings are short, some are long and fixed-size records would waste space for short records or truncate long ones.

For our minimal format, fixed-width records are appropriate. They demonstrate the basic pattern without adding the complexity of length encoding.

Implementing a Writer in C

Here is a complete C program that writes our format. It includes detailed comments explaining each operation.

```c
// writer.c -- Write a minimal binary record format file
// Compile: gcc -o writer writer.c -Wall -Wextra

#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>

/* CRC32 implementation */
static uint32_t crc32_table[256];
static int crc32_table_initialized = 0;

static void init_crc32_table(void)
{
    for (uint32_t i = 0; i < 256; i++) {
        uint32_t crc = i;
        for (int j = 0; j < 8; j++) {
            if (crc & 1)
                crc = (crc >> 1) ^ 0xEDB88320;
            else
                crc >>= 1;
        }
        crc32_table[i] = crc;
    }
    crc32_table_initialized = 1;
}

static uint32_t crc32_compute(const uint8_t *data, size_t len)
{
    if (!crc32_table_initialized)
        init_crc32_table();
    uint32_t crc = 0xFFFFFFFF;
    for (size_t i = 0; i < len; i++) {
        crc = (crc >> 8) ^ crc32_table[(crc ^ data[i]) & 0xFF];
```

```
    }
    return crc ^ 0xFFFFFFFF;
}

/* Header structure -- matches the file layout exactly */
struct __attribute__((packed)) file_header {
    uint8_t  magic[8];      /* 0x52 0x45 0x43 0x4F 0x52 0x44 0x56 0x31 */
    uint32_t version;       /* Format version, starts at 1 */
    uint32_t record_count;  /* Number of records in file */
    uint32_t record_size;   /* Size of each record in bytes */
    uint64_t timestamp;     /* Unix timestamp when file was written */
    uint32_t checksum;      /* CRC32 of header bytes 0-1B */
};

/* Record structure */
struct __attribute__((packed)) record {
    uint64_t id;     /* Record ID */
    uint8_t  value[48]; /* Value field */
};

/* Write a uint32 in little-endian byte order */
static void write_le32(FILE *fp, uint32_t val)
{
    uint8_t bytes[4] = {
        (uint8_t)(val & 0xFF),
        (uint8_t)((val >> 8) & 0xFF),
        (uint8_t)((val >> 16) & 0xFF),
        (uint8_t)((val >> 24) & 0xFF)
    };
    fwrite(bytes, 1, 4, fp);
}

/* Write a uint64 in little-endian byte order */
static void write_le64(FILE *fp, uint64_t val)
{
    for (int i = 0; i < 8; i++) {
        fputc((uint8_t)(val & 0xFF), fp);
        val >>= 8;
    }
}

int main(int argc, char *argv[])
{
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <output_file>\n", argv[0]);
        return 1;
    }

    const char *output = argv[1];
    const char *magic = "RECORDV1";
```

```
    const uint32_t version = 1;
    const uint32_t record_size = sizeof(struct record);

    /* Number of records to write */
    const uint32_t record_count = 3;

    /* Sample records */
    struct record records[3];
    memset(records, 0, sizeof(records));
    records[0].id = 1;
    memcpy(records[0].value, "First record data", 17);
    records[1].id = 2;
    memcpy(records[1].value, "Second record data", 18);
    records[2].id = 3;
    memcpy(records[2].value, "Third record data", 17);

    FILE *fp = fopen(output, "wb");
    if (!fp) {
        perror("fopen");
        return 1;
    }

    /* Write magic number (8 bytes) */
    fwrite(magic, 1, 8, fp);

    /* Write version as little-endian uint32 */
    write_le32(fp, version);

    /* Write record count as little-endian uint32 */
    write_le32(fp, record_count);

    /* Write record size as little-endian uint32 */
    write_le32(fp, record_size);

    /* Write timestamp as little-endian uint64 */
    uint64_t ts = (uint64_t)time(NULL);
    write_le64(fp, ts);

    /* Compute and write checksum over header bytes 0-1B */
    uint8_t header_bytes[28];
    memcpy(header_bytes, magic, 8);
    uint8_t ver_buf[4] = {(uint8_t)version, (uint8_t)(version>>8),
    ↪  (uint8_t)(version>>16), (uint8_t)(version>>24)};
    memcpy(header_bytes + 8, ver_buf, 4);
    uint8_t rc_buf[4] = {(uint8_t)record_count, (uint8_t)(record_count>>8),
    ↪  (uint8_t)(record_count>>16), (uint8_t)(record_count>>24)};
    memcpy(header_bytes + 12, rc_buf, 4);
    uint8_t rs_buf[4] = {(uint8_t)record_size, (uint8_t)(record_size>>8),
    ↪  (uint8_t)(record_size>>16), (uint8_t)(record_size>>24)};
    memcpy(header_bytes + 16, rs_buf, 4);
```

```
    uint8_t ts_buf[8];
    for (int i = 0; i < 8; i++) { ts_buf[i] = (uint8_t)(ts & 0xFF); ts >>= 8; }
    memcpy(header_bytes + 20, ts_buf, 8);

    uint32_t checksum = crc32_compute(header_bytes, 28);
    write_le32(fp, checksum);

    /* Write records */
    for (uint32_t i = 0; i < record_count; i++) {
        fwrite(&records[i], sizeof(struct record), 1, fp);
    }

    fclose(fp);
    printf("Wrote %u records to %s\n", record_count, output);
    return 0;
}
```

Compilation and execution:

```
gcc -o writer writer.c -Wall -Wextra
./writer test.rec
ls -la test.rec
xxd -l 32 test.rec
```

The output file should be 192 bytes (32-byte header plus 3 records of 56 bytes each). The first 32 bytes show the header with the magic string "RECORDV1", version 1, record count 3, record size 56 and the current timestamp.

Notice how the program writes fields one by one rather than writing the struct directly. This avoids padding issues: C compilers may insert padding bytes between struct fields for alignment, which would corrupt the file layout. Writing individual fields guarantees the exact byte layout we designed.

Parsing the Format in Python

Now let us read the file we just wrote using Python. This parser demonstrates the pattern you will use for any unknown binary format: read the header, validate it, then iterate through the records.

```python
# parser.py -- Parse the minimal binary record format
# Run: python3 parser.py test.rec

import struct
import zlib
import sys
import time

def parse_file(filename):
    with open(filename, 'rb') as fp:
        # Read the 32-byte header
        header_data = fp.read(32)
        if len(header_data) < 32:
            raise ValueError("File too small: expected at least 32-byte header")

        # Extract fields manually to control byte order exactly
        magic = header_data[0:8]
        if magic != b'RECORDV1':
            raise ValueError(f"Invalid magic number: {magic!r}")

        version = struct.unpack_from('<I', header_data, 8)[0]
        record_count = struct.unpack_from('<I', header_data, 12)[0]
        record_size = struct.unpack_from('<I', header_data, 16)[0]
        timestamp = struct.unpack_from('<Q', header_data, 20)[0]
        stored_checksum = struct.unpack_from('<I', header_data, 28)[0]

        # Verify checksum (CRC32 over bytes 0-27)
        computed_checksum = zlib.crc32(header_data[:28]) & 0xFFFFFFFF
        if stored_checksum != computed_checksum:
            raise ValueError(f"Header checksum mismatch:
            ↪  stored={stored_checksum:#010x}, "
                             f"computed={computed_checksum:#010x}")

        print(f"Magic:      {magic.decode('ascii')}")
        print(f"Version:    {version}")
        print(f"Records:    {record_count}")
        print(f"Record size:{record_size}")
        print(f"Timestamp:  {time.ctime(timestamp)}")
        print()

        # Parse records
        for i in range(record_count):
            offset = 32 + i * record_size
            record_data = fp.read(record_size)
            if len(record_data) < record_size:
                raise ValueError(f"Unexpected end of file at record {i} "
                                 f"(expected {record_size} bytes, got
                                 ↪  {len(record_data)})")

            record_id = struct.unpack_from('<Q', record_data, 0)[0]
```

```
            value = record_data[8:56]

            # Convert value to string, stripping trailing null bytes
            value_str = value.rstrip(b'\x00').decode('utf-8', errors='replace')

            print(f"Record {i}: offset={offset:#06x}, id={record_id}, "
                  f"value=\"{value_str}\"")

if __name__ == '__main__':
    if len(sys.argv) < 2:
        print(f"Usage: {sys.argv[0]} <file>")
        sys.exit(1)
    parse_file(sys.argv[1])
```

Running this parser on the file created by the C writer:

```
python3 parser.py test.rec
```

Output:

```
Magic:      RECORDV1
Version:    1
Records:    3
Record size:56
Timestamp:  Mon Sep  7 10:00:00 2026

Record 0: offset=0x000020, id=1, value="First record data"
Record 1: offset=0x000058, id=2, value="Second record data"
Record 2: offset=0x000090, id=3, value="Third record data"
```

The parser uses struct.unpack_from with the ‘<I’ and ‘<Q’ format strings to read little-endian integers. The ‘<’ prefix means little-endian byte order. The ‘I’ means unsigned 32-bit integer and ‘Q’ means unsigned 64-bit integer. This is the standard Python approach for binary parsing.

Understanding the Byte Layout

Let us examine the complete hex dump of our test file to verify the structure:

```
xxd test.rec
```

```
00000000: 5245 434f 5244 5631 0100 0000 0300 0000  RECORDV1........
00000010: 3800 0000 d495 e367 0000 0000 c7a2 6756  8......g......gV
00000020: 0100 0000 0000 0000 4669 7273 7420 7265  ........First re
00000030: 636f 7264 2064 6174 6100 0000 0000 0000  cord data.......
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000058: 0200 0000 0000 0000 5365 636f 6e64 2072  ........Second r
00000060: 6563 6f72 6420 6461 7461 0000 0000 0000  ecord data......
...continues...
```

At offset 0x00 we see the magic string. At offset 0x08 we see 01 00 00 00, which is version 1 in little-endian. At offset 0x0C we see 03 00 00 00, which is record count 3. At offset 0x10 we see 38 00 00 00, which is 56 in decimal (the record size). At offset 0x20 begins the first record: 01 00 00 00 00 00 00 00 is the 64-bit ID of 1, followed by the ASCII string “First record data” padded with null bytes.

Every byte is accounted for. The structure is clear and deterministic.

Why This Matters

This exercise demonstrates several fundamental principles of binary format design:

The magic number provides immediate format identification. Any tool can look at the first 8 bytes and know whether this is a file it can handle.

The header provides all the information needed to parse the file. Record count and record size allow the parser to validate the file length and calculate offsets without reading the entire body.

The checksum validates the header integrity. If the header is corrupted, the parser knows immediately rather than misinterpreting garbage metadata as valid information.

Fixed-width records enable direct access. Record N is always at a known offset, so the parser can seek directly to any record without scanning previous records.

These principles appear in every serious binary format. SQLite, PostgreSQL, InnoDB, LevelDB and every other storage engine follow the same patterns, extended and refined for their specific needs. Understanding this simple format gives you the foundation for understanding all of them.

Chapter 4: Serialization and Data Encoding

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

Type Systems and Storage Representation

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

Integer and Floating-Point Encoding

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

Strings, Text and Collation

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

Timestamps and Temporal Types

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

Null Handling and Optionality

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

Case Study: SQLite Record Format (varint, serial types)

Chapter 5: Page-Oriented Storage

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

Why Pages and Not Records

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

Page Headers and Metadata

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

Checksums and Corruption Detection

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

Page Types and Classification

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

Building a Page Viewer Tool

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

Case Study: SQLite Page Layout

Chapter 6: Record Layouts and Slot Directories

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

The Slot Directory Pattern

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

Variable-Length Fields and Backward Layout

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

Overflow Pages and Chain Pointers

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

Header Overhead Trade-Offs

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

Implementing a Record Decoder

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

Case Study: InnoDB Record Formats

Chapter 7: Free-Space Management

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

In-Page Free Space: Slabs and Linked Lists

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

Cross-Page Allocation: Extents and Bitmaps

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

Detecting Allocation Structures in Unknown Formats

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

Fragmentation: Causes and Symptoms

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

Implementing a Free-Space Analyzer

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

Case Study: PostgreSQL Free Space Map

Chapter 8: B-Trees and Index Structures

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

B-Tree Fundamentals and Design Trade-Offs

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

Page Layout: Root, Interior, Leaf

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

Traversing an Unknown B-Tree from Raw Bytes

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

Split and Merge Dynamics

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

Building a B-Tree Visualizer

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

Chapter 9: Alternative Index Structures: LSM-Trees, Hash Indexes and More

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

Log-Structured Merge Trees: Architecture

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

SSTable Format and Block Structure

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

Bloom Filters and Skip Lists

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

Hash Indexes and Direct-Address Tables

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

Tries and Radix Trees

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

Case Study: LevelDB File Format

Chapter 10: Write-Ahead Logging and Journaling

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

The Write-Ahead Logging Guarantee

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

Log Record Formats and Structure

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

LSN Ordering and Recovery Semantics

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

Redo and Undo: Two Phases

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

Parsing a WAL from Raw Bytes

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

Chapter 11: Transactions and ACID Internals

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

ACID Properties as Storage Requirements

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

Transaction IDs and Commit Ordering

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

Implementing Atomicity with Logs

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

Durability and fsync Semantics

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

Detecting Transaction State in Storage

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

Chapter 12: Concurrency Control: Locking and MVCC

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

Locking: Tables, Pages, Rows

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

Deadlock Detection and Prevention

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

MVCC: Multi-Version Architecture

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

Version Chains and Timestamps

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

Visibility Maps and Snapshot Resolution

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

Case Study: PostgreSQL MVCC Implementation

Chapter 13: Snapshots, Isolation Levels and Consistent Reads

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

Snapshot Semantics and Read Views

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

Isolation Levels: From Read Committed to Serializable

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

Phantoms and Range Queries

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

Serializable Snapshot Isolation

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

Detecting Snapshot Mechanisms in Storage

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

Chapter 14: Crash Recovery: From Failure to Consistency

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

What Can Go Wrong During a Crash

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

The ARIES Recovery Algorithm

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

Analysis Phase: Building the Transaction Table

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

Redo Phase: Reapplying Changes

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

Undo Phase: Rolling Back Uncommitted Work

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

Checkpointing Strategies

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

Manual Recovery Techniques

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

Chapter 15: Caching and Buffer Pools

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

The Buffer Pool Architecture

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

Page Lifecycle: Read, Dirty, Write-Back

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

Eviction Policies: LRU and Variants

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

Clean vs Dirty Page Semantics

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

Observing Cache Behavior on Disk

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

Chapter 16: Compaction and Fragmentation

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

Why Fragmentation Happens

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

In-Place vs Append-Only Compaction

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

LSM-Tree Compaction Strategies

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

Tombstones and Garbage Collection

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

Measuring Fragmentation in Storage

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

Case Study: RocksDB Compaction

Chapter 17: Compression in Storage Engines

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

Why Compress and When Not To

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

Page-Level vs Columnar Compression

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

Dictionary Encoding and Direct Coding

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

Run-Length and Delta Encoding

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

LZ Algorithms and Block Compression

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

Decompressing Unknown Compressed Pages

Chapter 18: Corruption Detection and Forensic Analysis

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

Types of Corruption: Logical vs Physical

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

Checksum Failures and Torn Pages

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

Detecting Index-Data Inconsistencies

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

Orphaned Pages and Lost Records

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

Forensic Imaging and Analysis Techniques

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

Chapter 19: Tools of the Trade

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

Hex Editors and Binary Viewers

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

Command-Line Binary Analysis Tools

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

Debuggers for Storage Investigation

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

System Call Tracing with strace and bpftrace

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

Building Custom Analysis Scripts

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

Setting Up a Reverse Engineering Lab

Chapter 20: Dynamic Instrumentation and Runtime Analysis

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

Tracing Page I/O Operations

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

LD_PRELOAD and Library Injection

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

eBPF for System-Level Observability

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

Debug Builds and Internal Logging

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

Correlating Runtime Behavior with Storage

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

Chapter 21: Fuzzing, Differential Analysis and Black-Box Experimentation

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

Fuzzing Storage Engines and Parsers

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

Differential Analysis of File States

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

Controlled Mutation Experiments

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

Statistical Pattern Recognition

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

Black-Box Protocol Discovery

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

Chapter 22: The Reverse Engineering Workflow

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

Phase 1: Initial Reconnaissance

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

Phase 2: Hypothesis Formation

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

Phase 3: Controlled Experimentation

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

Phase 4: Decoding Structures

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

Phase 5: Validation and Documentation

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

Phase 6: Implementing a Compatible Parser

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

The Hypothesis Journal Pattern

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

Chapter 23: Case Study: End-to-End SQLite Reverse Engineering

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

Initial Inspection of a Fresh Database

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

Discovering the Database Header

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

Mapping the Page Structure

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

Decoding B-Tree Pages

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

Understanding Record Serialization

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

Uncovering the WAL Format

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

Lessons and Techniques Demonstrated

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

Chapter 24: Case Studies: Other Formats and Techniques

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

InnoDB Tablespace and Page Structure

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

PostgreSQL Heap Tuples and Catalog

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

LevelDB SSTable and Manifest

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

Redis RDB Persistence Format

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

Common Patterns Across Systems

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

Chapter 25: Ethics, Legality and Best Practices

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

Copyright and Reverse Engineering Law

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

DMCA and Circumvention Questions

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

Interoperability and Fair Use

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

Data Privacy and Sensitivity

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

Responsible Disclosure of Findings

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

Best Practices for Professional Work

Conclusion: Beyond the Known Formats

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

References

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