Building High-Performance Compression Systems

bzip3 and Beyond

A practical guide to modern compression from core algorithms to real-world performance tuning

HASSAN BROOKS

Building High-Performance Compression Systems

bzip3 and Beyond

Steve Publications

This book is available at
https://leanpub.com/buildinghigh-performancecompressionsystems

This version was published on 2026-09-07

Contents

bzip3 and Beyond

Compression is not a single algorithm but a composable pipeline. Modern compressors achieve their results through layered transformations, each optimizing for specific statistical properties of data. This book explores that pipeline in depth, using bzip3 (a block-sorting compressor that improves on the classic bzip2 architecture) as a central case study. Through bzip3 we examine entropy coding, data transformations, match finding, parallel processing, and the engineering trade-offs that define real-world compression software. By the end, you will understand how to analyze a compression workload, design a compression pipeline, implement the core components in C, and tune the result for production performance.

Introduction

In 2008, the Wikimedia Foundation began storing its database dumps in compressed format. By 2018 the compressed size of those dumps exceeded one terabyte. In the same decade, the number of cores on mainstream server processors multiplied by four, while memory bandwidth improved more slowly. These trends created a pressure point: compressing and decompressing large datasets was consuming increasing shares of CPU time, and the single-threaded compressors inherited from the 1990s could not scale.

That is one motivation for a project called bzip3.

bzip3 is a compressor designed as a spiritual successor to bzip2, a classic Unix compression tool from 1996. Like bzip2, it uses the Burrows-Wheeler transform and achieves particularly good compression on text and source code. But where bzip2 was designed for a single-core era with 32-bit integers and Huffman coding, bzip3 uses modern arithmetic coding, suffix-array-based transforms, and parallel block processing. It compresses text more efficiently than bzip2, faster than many LZMA-based tools, and with better ratio than Zstandard at comparable speeds on certain workloads.

bzip3 is not famous. It does not have the brand recognition of gzip or the ubiquity of Zstandard. It is not included by default in any major operating system. But it is a carefully engineered compressor, and its architecture illustrates design decisions that are broadly relevant to anyone building or selecting compression software.

This book uses bzip3 as a lens. It does not just describe bzip3; it uses bzip3 to teach the general principles of high-performance compression system design.

The Economics of Compression

At its core, compression is an engineering bargain. You trade CPU cycles and memory for storage space or bandwidth. The bargain is worthwhile when the savings in I/O outweigh the cost of computation, and the details of that trade-off determine which algorithm you choose.

Consider a web server that serves one hundred gigabytes of HTML and JavaScript every hour. If compression reduces that traffic to fifty gigabytes, you save fifty gigabytes of bandwidth. If your upstream bandwidth costs two dollars per gigabyte, you save one hundred dollars per hour. Compression paid for itself many times over in the cost of CPU cycles required to do it.

Now consider a database that must decompress data on every read. If decompression is slow, queries become latency-bound on CPU rather than on disk. In that case a slower-to-compress but faster-to-decompress algorithm may be preferable. LZ4 exists because such trade-offs matter.

Compression choices ripple outward. A slow compressor may become a bottleneck in build systems, CI pipelines, or data ingestion jobs. A bad choice may cause systems to scale poorly because network or storage costs grow faster than CPU costs shrink. Getting compression right is rarely the most glamorous part of systems engineering, but it is often one of the highest-leverage decisions.

The Compression Landscape

Modern compression algorithms fall into several families, each with a different emphasis.

The LZ family dominates general-purpose compression. LZ77, proposed by Abraham Lempel and Jacob Ziv in 1977, replaced repeated substrings with back-references. Its descendants include gzip and deflate, which use LZ77 followed by Huffman coding and remain the default compression on the web and in many archives. LZ4, developed by Yann Collet at Facebook in 2011, pushes LZ77 to the speed extreme: compression speeds exceeding five hundred megabytes per second per core, with decompression often at gigabyte-per-second rates. Zstandard, also from Yann Collet, balances speed and ratio more carefully, using LZ77 with adaptive compression levels and Finite State Entropy coding. LZMA, from Igor Pavlov in the late 1990s, targets ratio above all else, using a large dictionary and range encoding to achieve some of the best general-purpose compression ratios available.

The BWT family takes a different approach. Instead of looking for repeated substrings in a sliding window, these compressors reorder data to make it easier for entropy coders to compress it. The Burrows-Wheeler transform rearranges each block of input so that similar symbols cluster together. After

transformation, symbols with predictable patterns appear in long runs, and entropy coding can compress those runs efficiently. bzip2, released by Julian Seward in 1996, made this approach practical for general use. bzip3 continues and extends this tradition.

Web-specific compressors have emerged to exploit the structure of modern web content. Brotli, developed by Google and specified in RFC 7932, uses LZ77 with a large static dictionary of common English words and HTML patterns, second-order context modeling for entropy coding, and a sixteen-megabyte sliding window. On web content it typically beats gzip by fifteen to twenty-five percent.

All of these compressors share a fundamental architecture: they split compression into a pipeline of stages. Each stage transforms the data in a way that makes the next stage more effective. The LZ family typically runs match finding, then encodes the resulting literals and back-references. The BWT family runs run-length encoding, applies a transform, and then entropy-codes the result. Zstandard and Brotli mix elements of both.

Understanding these pipelines, and why each stage is chosen, is the main purpose of this book.

Why bzip3 Matters

Why choose bzip3 as the central case study?

First, bzip3 exemplifies the block-based approach. It processes data in fixed-size blocks, each compressed independently. This design supports parallel compression and decompression, random access, and error localization. Understanding bzip3 teaches you how to structure compression as a pipeline that can be parallelized and extended.

Second, bzip3 combines several classic techniques in a modern way. It uses run-length encoding as a first pass, then an LZ77-inspired predictor called LZP, then the Burrows-Wheeler transform, and finally an arithmetic coder with context mixing. Each stage removes a different kind of redundancy, and understanding how they compose helps you evaluate similar combinations in other compressors.

Third, bzip3 is accessible as a reference implementation. The codebase is small enough to read end to end but large enough to be production-quality. It

uses well-chosen external libraries, such as libsais for suffix-array-based BWT construction. Reading and implementing bzip3 gives you practical experience with real compression code, not just theoretical descriptions of algorithms.

Fourth, bzip3 demonstrates trade-offs that matter. It does not try to be the fastest compressor or the highest-ratio compressor. It targets a middle ground on the speed-versus-ratio curve, optimized for text and source code. That makes it a good example of how to think about workload-specific optimization.

What This Book Is Not

This book is not a survey of every compression algorithm ever invented. There is no exhaustive treatment of arithmetic coding variants, no complete bibliography of PPM-family compressors, no deep dive into range-encoder patent history.

This book is not primarily about theory. While we will establish the information-theoretic foundations of compression, our focus is on implementation. The reader should walk away able to write C code that compresses and decompresses data, not just able to describe how compression works in abstract terms.

This book does not cover compression for specific media types in depth. Image compression, audio compression, and video compression involve domain-specific techniques and perceptual models that are largely orthogonal to the general-purpose text and binary compression covered here.

This book is not a tutorial for beginners. It assumes you are already comfortable with C, with the basics of CPU architecture and memory hierarchy, and with systems programming concepts such as threading and I/O.

How to Use This Book

The book is organized in four parts.

The first part establishes foundations. Chapter 2 introduces the information theory that gives us limits on how well any compressor can perform. Chapters 3 through 6 explain the core building blocks: entropy coding schemes such as Huffman and range coding, LZ-style match finding, the Burrows-Wheeler transform, and move-to-front coding.

The second part uses those building blocks to understand bzip2 and then bzip3. Chapter 7 explains block-based design principles. Chapters 8 and 9 dive into range coding implementation and context modeling. Chapters 10 and 11 describe bzip3's architecture and compression pipeline in detail.

The third part is about engineering. Chapters 12 through 17 cover implementation topics: writing a reference compressor in C, using SIMD instructions, adding parallelism, handling streaming data, managing memory and cache behavior, and integrating with I/O subsystems.

The fourth part zooms out. Chapters 18 through 24 address file format design, benchmarking methodology, comparison of bzip3 with other compressors, security considerations, API design, production deployment, and emerging techniques.

Each chapter is written to stand on its own but also to build on previous material. If you are already familiar with the basics of compression and want to jump straight to bzip3 implementation, you can begin at Chapter 12. If you want to understand how the field as a whole works, read sequentially from the beginning.

All code examples in this book are written in portable C and are designed to compile on mainstream toolchains with standard optimization flags. The code is not optimized to the point of illegibility, but it is structured as production-quality software would be.

Let us begin.

Chapter 1: Information Theory Foundations

A compressor is nothing more than a program that exploits predictability. Where data is predictable, you can describe it with fewer bits. Where data is unpredictable, no amount of algorithmic cleverness will compress it. The mathematical framework that makes this intuition precise is information theory, pioneered by Claude Shannon in the late 1940s.

This chapter establishes that framework. We will not prove Shannon's theorems in full generality, but we will explain what they mean for compression and why they matter for the algorithms we will implement later.

Entropy and Its Meaning

Suppose you receive a stream of symbols from some source. Each symbol is drawn from a known alphabet, and each symbol has some probability of occurring. How many bits per symbol, on average, do you need to encode this stream without loss?

The answer is given by the Shannon entropy of the source. For a discrete random variable X taking values in a finite alphabet A with probability distribution p(x), the entropy H(X) is:

H(X) = - sum over all x in A of p(x) log2 p(x)

The logarithm base 2 gives us bits. If you change the base to e you get nats; base 10 gives hartleys. For compression we use bits.

Consider a simple example: a source that emits A and B with equal probability. The entropy is:

H = -(0.5 log2(0.5) + 0.5 log2(0.5)) = -(0.5 × -1 + 0.5 × -1) = 1 bit

This means we need exactly one bit per symbol on average. Using fixed-length codes (A=0, B=1) is optimal in this case.

Now consider a source that emits A with probability 0.9 and B with probability 0.1. The entropy is:

$$H = -(0.9 \log_2(0.9) + 0.1 \log_2(0.1)) \approx -(0.9 \times -0.152 + 0.1 \times -3.322) \approx 0.469 \text{ bits}$$

The entropy is less than one bit because the source is predictable: most symbols are A. Any encoding that uses one bit per symbol is wasting space. A good encoder should exploit the imbalance and use shorter codes for A.

Entropy measures the average uncertainty of a random variable. Equivalently, it measures the minimum average number of bits required to encode the symbols from that distribution without loss. It is not about the symbols themselves but about the probabilities. A stream of English letters has different entropy depending on the letter distribution, even though the alphabet is the same.

A few important properties follow from the formula:

First, entropy is zero when one symbol has probability 1. There is no uncertainty, no information to convey, and no bits needed. Second, entropy is maximized when all symbols have equal probability. In that case no symbol is more likely than any other, and no encoding can exploit imbalance. Third, entropy is concave: spreading probability mass more evenly always increases entropy.

For an alphabet of size N, the maximum entropy is $\log_2(N)$ bits per symbol. For 8-bit bytes, maximum entropy is 8 bits per byte. Random data has entropy close to 8 bits per byte and is incompressible. Text typically has lower entropy because some letters are more common than others and because letter sequences are correlated.

Shannon's Source Coding Theorem

Shannon's source coding theorem states that for any memoryless source with entropy H, there exists an encoding scheme that achieves an average code length L satisfying:

$$H \leq L < H + 1$$

and conversely, no encoding scheme can achieve $L < H$ without loss.

The first inequality gives us a lower bound: no lossless compressor can compress below the entropy of the source. The second inequality says there

always exists a practical scheme that comes within one bit per symbol of the entropy.

The one-bit gap arises from the discreteness of bits. If the optimal code length for a symbol is 0.469 bits, you cannot emit a fraction of a bit. You must round up, and rounding introduces at most one bit of overhead per symbol. As we will see in later chapters, arithmetic coding and range coding can approach the entropy more closely than discrete-length codes like Huffman by spreading the fractional bits across many symbols.

The theorem has an important corollary for compression: if a compressor claims to compress random data, it is either lying or compressing only by exploiting structure in the test suite. Random data by definition has maximum entropy, so no lossless compressor can reduce its size. Some compressors may appear to compress random data on small samples because they are exploiting the particular seed of the random number generator, but they cannot do so in general.

Measuring Compression: Ratio, Bits-per-Byte, and Entropy Gap

Engineers measure compression quality in three common ways.

Compression ratio is the most intuitive: original size divided by compressed size. A ratio of 3.0 means the compressed data is one-third the original size. This is easy to explain to stakeholders but obscures detail when comparing compressors across different data sets.

Bits-per-byte is more precise: the number of bits in the compressed output divided by the number of bytes in the input. A bits-per-byte of 5.0 means five bits of compressed data for each input byte, which corresponds to a compression ratio of 8/5 = 1.6. This metric lets you compare compression directly against entropy: if the entropy of the source is 4.2 bits per byte and your compressor achieves 5.0 bits per byte, you are within 0.8 bits per byte of the theoretical limit.

Entropy gap is the difference between the achieved bits-per-byte and the entropy of the source. A smaller gap means better compression efficiency. In practice, measuring entropy is not straightforward because it depends on knowing the true probability distribution of the source, and real data sources are not memoryless.

To estimate entropy for practical purposes, engineers often use empirical methods. For example, one can count symbol frequencies in a block of data, compute the zero-order entropy from those frequencies, and use that as a lower bound. One can also use higher-order models that estimate the probability of each symbol conditioned on previous symbols. These methods give approximate entropy values that are useful for comparison.

For the work in this book, we will use bits-per-byte as the primary metric and compression ratio as a secondary. We will also refer to entropy estimates where they help explain compressor behavior.

The Gap Between Theory and Practice

Shannon's theorem gives us a lower bound for memoryless sources. Real data sources are not memoryless. They have long-range correlations, structure at multiple levels, and patterns that change over time. This means two things: the theoretical lower bound is harder to compute, and practical compressors must use models that exploit structure beyond simple symbol frequencies.

Consider English text. The letter frequencies alone give a zero-order entropy of about 4.1 bits per character. But English has strong higher-order correlations: Q is almost always followed by U, and the bigram frequency distribution is far from uniform. If we condition each symbol on its predecessor, the first-order entropy drops to about 3.3 bits per character. Conditioning on more context gives even lower entropy: studies have estimated the entropy of English at between one and one-and-a-half bits per character when context is fully exploited.

A good compressor builds a model that approximates this conditional structure. LZ77-based compressors implicitly capture higher-order correlations by finding repeated substrings: if a substring appears twice, the second occurrence is predictable given the first. The Burrows-Wheeler transform groups similar contexts together, making the symbol sequence more predictable. Context modeling explicitly estimates symbol probabilities from surrounding symbols.

But every model introduces overhead. The compressor must transmit information about the model so that the decompressor can reconstruct it. A model that is too complex may compress better but add so much overhead that the net gain is negative. This is a fundamental tension in compression design.

Another practical constraint is computational complexity. Shannon's theorem does not bound the time or memory required to encode or decode. A compressor could theoretically achieve very low entropy gap by using an exhaustive search for repeated patterns, but such a compressor might require exponential time. Practical compressors must balance compression quality with speed and memory usage.

Practical Implications for Compressor Design

The information theory foundation translates directly into design principles for compressors.

First, know your data. Different data types have different structures and different effective entropies. Text benefits from higher-order modeling. Binary data may benefit more from LZ-style match finding. Images benefit from transforms that exploit spatial locality. A compressor that excels on one type may be mediocre on another.

Second, use layered transformations. Each transformation should remove a different kind of redundancy. Run-length encoding removes run-length redundancy. The Burrows-Wheeler transform rearranges data to cluster similar symbols. LZ matching removes repetition. Entropy coding removes statistical redundancy. The pipeline approach lets each stage do one thing well.

Third, model adaptively. Real data is not stationary. A compressor that assumes a fixed symbol distribution will perform poorly when the distribution changes. Adaptive models update their probability estimates as they process data, tracking changes in the source distribution.

Fourth, account for overhead. Every bit of model information you transmit reduces your net compression. Design your model so that the overhead is small compared to the compression benefit.

Fifth, respect computational constraints. The best theoretical model is useless if it cannot run within your time or memory budget. Choose algorithms and parameters that fit your deployment environment.

These principles guide the rest of this book. When we examine bzip3's pipeline, we will see each principle in action. When we implement compression components, we will apply them. The mathematics gives us the goals; engineering gives us the means.

In the next chapter we will move from theory to the concrete mechanisms that compressors use: symbol coding schemes such as Huffman coding and arithmetic/range coding. These are the workhorse algorithms that convert probability distributions into bit streams, and they are essential building blocks for everything that follows.

Chapter 2: Symbol Coding: Huffman and Beyond

Entropy gives us a theoretical limit. Symbol coding schemes are the practical mechanisms that approach that limit. A symbol coder takes a sequence of symbols with known or estimated probabilities and produces a bit stream such that more probable symbols use fewer bits on average.

This chapter explains the three main families of symbol coding used in modern compression: Huffman coding, arithmetic coding, and range coding. We will explain how each works, when each is appropriate, and why range coding has largely replaced arithmetic coding in new systems.

Prefix Codes and the Kraft-McMillan Theorem

Before building any symbol coder, we must understand the constraint that all lossless prefix codes must satisfy.

A prefix code is a set of codewords such that no codeword is a prefix of another. This property guarantees that you can decode a bit stream without ambiguity or lookahead: as soon as you recognize a complete codeword, you know the corresponding symbol, and you can immediately start decoding the next one.

For example, the codewords 0, 10, and 11 form a prefix code. You can decode the stream 01011 as the sequence A, B, C. The codewords 0, 01, and 10 do not form a prefix code: after reading the first bit 0, you cannot tell whether the codeword is complete or whether you should read another bit.

The Kraft-McMillan theorem gives a necessary and sufficient condition for the existence of a prefix code with given codeword lengths. Let l_i be the length of the codeword for symbol i in an alphabet of size N. A prefix code with those lengths exists if and only if:

sum over i of 2^(-l_i) ≤ 1

This inequality has an intuitive interpretation. Each codeword of length l occupies a fraction 2^(-l) of the code space (the set of all infinite bit sequences). The Kraft sum is the total fraction of code space used. If it exceeds 1, you need more space than exists.

The theorem is useful because it lets us reason about optimal code lengths before constructing the codes themselves. From the entropy formula, we know that the optimal length for symbol i is approximately -log2 p(i). The Kraft inequality is satisfied if we define l_i = ceil(-log2 p(i)), guaranteeing that an optimal prefix code exists.

Building a Huffman Tree

Huffman coding, described by David A Huffman in 1952, constructs an optimal prefix code for a known symbol distribution. Optimal here means no other prefix code has a smaller average length for the same distribution.

The algorithm is simple and greedy:

1. Create a leaf node for each symbol, weighted by its frequency.
2. Put all nodes into a priority queue keyed by weight.
3. While more than one node remains: a. Remove the two nodes with smallest weight. b. Create a new internal node with those two as children and weight equal to their sum. c. Insert the new node back into the queue.
4. The remaining node is the root of the Huffman tree.

To encode a symbol, traverse the tree from the root to the leaf for that symbol, emitting a bit for each edge (conventionally 0 for left, 1 for right). To decode, start at the root and follow edges based on input bits until you reach a leaf.

Here is a compact C implementation:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_FREQ 65536
#define MAX_SYMBOLS 256

typedef struct Node {
    int symbol;       // -1 for internal nodes
    int freq;
    struct Node *left, *right;
} Node;

typedef struct {
    Node *nodes[MAX_SYMBOLS];
    int count;
} PriorityQueue;

static void pq_init(PriorityQueue *pq) {
    pq->count = 0;
}

static void pq_push(PriorityQueue *pq, Node *n) {
    int i = pq->count;
    while (i > 0 && pq->nodes[(i-1)/2]->freq > n->freq) {
        pq->nodes[i] = pq->nodes[(i-1)/2];
        i = (i-1)/2;
    }
    pq->nodes[i++] = n;
    pq->count = i;
}

static Node *pq_pop(PriorityQueue *pq) {
    Node *min = pq->nodes[0];
    pq->nodes[0] = pq->nodes[--pq->count];
    int i = 0;
    while (1) {
        int l = 2*i+1, r = 2*i+2, smallest = i;
        if (l < pq->count && pq->nodes[l]->freq < pq->nodes[smallest]->freq)
            smallest = l;
        if (r < pq->count && pq->nodes[r]->freq < pq->nodes[smallest]->freq)
            smallest = r;
        if (smallest == i) break;
        pq->nodes[i] = pq->nodes[smallest];
        i = smallest;
    }
    return min;
}

static Node *new_node(int symbol, int freq) {
```

```
    Node *n = malloc(sizeof(Node));
    n->symbol = symbol;
    n->freq = freq;
    n->left = n->right = NULL;
    return n;
}

Node *huffman_build(const int *freq, int nsymbols) {
    PriorityQueue pq;
    pq_init(&pq);
    for (int i = 0; i < nsymbols; i++) {
        if (freq[i] > 0)
            pq_push(&pq, new_node(i, freq[i]));
    }
    if (pq.count == 0) return NULL;
    if (pq.count == 1) {
        Node *leaf = pq_pop(&pq);
        Node *parent = new_node(-1, leaf->freq);
        parent->left = leaf;
        return parent;
    }
    while (pq.count > 1) {
        Node *l = pq_pop(&pq);
        Node *r = pq_pop(&pq);
        Node *parent = new_node(-1, l->freq + r->freq);
        parent->left = l;
        parent->right = r;
        pq_push(&pq, parent);
    }
    return pq_pop(&pq);
}

void huffman_print_codes(Node *root, int depth, unsigned code, FILE *out) {
    if (!root) return;
    if (root->symbol >= 0) {
        fprintf(out, "%d: ", root->symbol);
        for (int i = 7; i >= 0; i--)
            fputc((code >> i) & 1 ? '1' : '0', out);
        fprintf(out, " (len=%d)\n", depth);
        return;
    }
    huffman_print_codes(root->left, depth+1, code << 1, out);
    huffman_print_codes(root->right, depth+1, (code << 1) | 1, out);
}
```

This implementation uses a simple priority queue with array-based heap operations. For a symbol alphabet of 256 bytes, the build step is negligible in cost.

Huffman coding is widely used because it is simple, correct, and nearly optimal. On typical English text it achieves within a few percent of the entropy. But it has limitations: it can only assign integer-length codes, so it cannot exploit fractional bits-per-symbol. If a symbol should use 1.7 bits but Huffman assigns 2, you lose compression efficiency. This limitation becomes significant for high-probability symbols or highly skewed distributions.

Canonical Huffman Codes

A practical variant of Huffman coding is the canonical form, used in DEFLATE and Brotli. Canonical Huffman codes have a specific structure: all codes of the same length are assigned in numeric order, and code lengths follow a deterministic rule. This allows the decoder to reconstruct the code table from just the set of code lengths, without transmitting the full tree.

To construct a canonical Huffman code:

1. Count how many symbols have each code length.
2. Assign the first code of length l as 0 if l is the shortest length, otherwise as (prev_code + 1) << (l - prev_length).
3. Assign subsequent codes of the same length by incrementing.

Here is the construction code:

```
#define MAX_CODE_LEN 16

void huffman_canonical(const Node *root, int *code_len, int *code, int maxsym)
    {
    // First pass: compute code lengths
    memset(code_len, 0, sizeof(int) * maxsym);
    if (!root) return;
    int count[MAX_CODE_LEN] = {0};
    // Collect lengths from tree
    void collect(Node *n, int d) {
        if (!n) return;
        if (n->symbol >= 0) {
            code_len[n->symbol] = d;
            if (d < MAX_CODE_LEN) count[d]++;
        } else {
            collect(n->left, d+1);
            collect(n->right, d+1);
        }
```

```
    }
    collect(root, 0);
    // Second pass: assign canonical codes
    int next_code[MAX_CODE_LEN] = {0};
    for (int i = 1; i < MAX_CODE_LEN; i++)
        next_code[i] = (next_code[i-1] + count[i-1]) << 1;
    for (int s = 0; s < maxsym; s++) {
        if (code_len[s] > 0) {
            code[s] = next_code[code_len[s]]++;
        }
    }
}
```

Canonical codes are important for interoperability: two implementations that independently construct Huffman trees for the same distribution will produce different tree structures but the same canonical codes, making the format deterministic.

Arithmetic Coding: The Idea

Arithmetic coding removes the integer-length limitation by encoding the entire message as a single number in a range. Instead of assigning a fixed code to each symbol, arithmetic coding maintains a current range [low, high) representing the interval consistent with all symbols seen so far. For each new symbol, it subdivides the range proportionally to symbol probabilities, keeping the sub-range that corresponds to that symbol.

Consider encoding the sequence “ABA” with p(A)=0.7, p(B)=0.3. Start with range [0, 1):

- Encode A: range becomes [0, 0.7)
- Encode B: subdivide [0, 0.7) into [0, 0.49) for A and [0.49, 0.7) for B. We take [0.49, 0.7)
- Encode A: subdivide [0.49, 0.7) into [0.49, 0.643) for A and [0.643, 0.7) for B. We take [0.49, 0.643)

Any number in [0.49, 0.643) uniquely identifies the sequence “ABA”. We can choose the shortest binary fraction in that range, for example 0.101 binary (0.625 decimal), which is three bits.

The length of the code approaches -log2 P(sequence), where P(sequence) is the product of symbol probabilities. This gives near-optimal compression because it does not require integer code lengths per symbol.

Arithmetic coding is theoretically elegant but historically difficult to implement efficiently. The main challenges are:

1. Precision: the range narrows exponentially, requiring high-precision arithmetic.
2. Overflow: as the range narrows, the high bits of low and high may agree and can be emitted, but handling this carefully while avoiding overflow is nontrivial.
3. Patent restrictions: arithmetic coding was patented in several countries for much of the late 1980s and 1990s. bzip2 switched from arithmetic to Huffman coding partly because of these patents.

These implementation and legal challenges led to the development of range coding as a practical alternative.

Range Coding: Why It Is Better Than Arithmetic Coding

Range coding is essentially arithmetic coding implemented in a way that is faster and simpler on standard hardware. The term "range coding" was popularized by G.N.N. Martin in 1996, though the core idea predates that. In practice, range coders are so similar to arithmetic coders that the distinction is more about implementation convention than mathematical difference.

A typical 64-bit range coder maintains two state variables: low (the low end of the current range) and range (the width of the current range). Range is kept within a fixed bit-width, typically 64 bits. When encoding a symbol with cumulative frequency count low_count and total frequency freq_total, the coder:

1. Computes the new range as range * sym_count / freq_total
2. Updates low as low + (range_old * low_count / freq_total)
3. Normalizes by shifting out high bits of low when range becomes too small

Here is a compact range encoder:

```
#include <stdint.h>
#include <string.h>

typedef struct {
    uint64_t low;
    uint64_t range;
    uint8_t buf[65536];
    int pos;
    int pending_bytes;
} RangeEncoder;

void rc_init(RangeEncoder *rc) {
    rc->low = 0;
    rc->range = UINT64_C(0xFFFFFFFFFFFFFFFF);
    rc->pos = 0;
    rc->pending_bytes = 0;
    memset(rc->buf, 0, sizeof(rc->buf));
}

static void rc_write_byte(RangeEncoder *rc, uint8_t b) {
    if (rc->pos < (int)sizeof(rc->buf))
        rc->buf[rc->pos++] = b;
}

static void rc_shift(RangeEncoder *rc) {
    uint64_t top = rc->low >> 56;
    uint64_t carry = top + rc->pending_bytes;
    if (carry != top) {
        for (int i = 0; i < 64; i++) {
            uint64_t c = (carry >> 8) & 0xFF;
            carry = carry >> 8;
            if (c != 0xFF || carry != top) {
                rc_write_byte(rc, (uint8_t)c);
                break;
            }
        }
        rc->pending_bytes = 0;
    } else {
        rc->pending_bytes++;
    }
    rc_write_byte(rc, (uint8_t)(top));
    rc->low = (rc->low << 8) & UINT64_C(0xFFFFFFFFFFFFFFFF);
    rc->range <<= 8;
}

void rc_encode_symbol(RangeEncoder *rc, int low_count, int sym_count, int
↪  total) {
    uint64_t range_lo = rc->range / total;
    uint64_t range_hi = range_lo * sym_count;
    rc->low += range_lo * low_count;
```

```
    rc->range = range_hi;
    while ((rc->range & UINT64_C(0x8000000000000000)) == 0) {
        rc_shift(rc);
    }
}

void rc_finish(RangeEncoder *rc) {
    // Flush remaining bits
    for (int i = 0; i < 8; i++) {
        uint64_t top = rc->low >> 56;
        uint64_t carry = top + rc->pending_bytes;
        if (carry != top) {
            uint64_t c = (carry >> 8) & 0xFF;
            rc_write_byte(rc, (uint8_t)c);
            rc->pending_bytes = 0;
            break;
        }
    }
    for (int i = 0; i < 8; i++) {
        uint64_t top = rc->low >> 56;
        rc_write_byte(rc, (uint8_t)top);
        rc->low <<= 8;
    }
}

int rc_get_size(RangeEncoder *rc) {
    return rc->pos;
}

uint8_t *rc_get_data(RangeEncoder *rc) {
    return rc->buf;
}
```

The decoder mirrors the encoder, maintaining the same low and range variables and using the same frequency table to decode each symbol:

```
typedef struct {
    uint64_t low;
    uint64_t range;
    const uint8_t *data;
    int pos;
    int length;
} RangeDecoder;

void rd_init(RangeDecoder *rd, const uint8_t *data, int length) {
    rd->data = data;
    rd->pos = 0;
    rd->length = length;
    rd->low = 0;
```

```
    rd->range = UINT64_C(0xFFFFFFFFFFFFFFFF);
}

static uint8_t rd_read_byte(RangeDecoder *rd) {
    if (rd->pos >= rd->length) return 0;
    return rd->data[rd->pos++];
}

static void rd_shift(RangeDecoder *rd) {
    rd->low = ((rd->low << 8) & UINT64_C(0xFFFFFFFFFFFFFFFF)) |
    ↪  rd_read_byte(rd);
    rd->range <<= 8;
}

int rd_decode_symbol(RangeDecoder *rd, const int *cumfreq, int total) {
    uint64_t scaled = ((rd->low - rd->range + 1) * total - 1) / rd->range;
    // Binary search in cumulative frequency table
    int lo = 0, hi = total;
    while (lo + 1 < hi) {
        int mid = (lo + hi) / 2;
        if (cumfreq[mid] <= scaled)
            lo = mid;
        else
            hi = mid;
    }
    int sym = lo;
    uint64_t range_lo = rd->range / total;
    uint64_t range_hi = range_lo * (cumfreq[sym+1] - cumfreq[sym]);
    rd->range = range_hi;
    rd->low -= range_lo * cumfreq[sym];
    while ((rd->range & UINT64_C(0x8000000000000000)) == 0) {
        rd_shift(rd);
    }
    return sym;
}
```

Range coding has several advantages over classical arithmetic coding implementations:

1. Speed: it renormalizes byte by byte rather than bit by bit, roughly doubling throughput on byte-oriented architectures.
2. Simplicity: the algorithm is straightforward to implement correctly, without the edge cases of some arithmetic coding implementations.
3. No patent issues: by the time range coding became popular, the arithmetic coding patents had expired in most jurisdictions.
4. Accuracy: 64-bit range coding approaches the entropy as closely as practical arithmetic coding.

The slight theoretical inefficiency of range coding compared to pure arithmetic coding (a few bits per kilobyte in practice) is negligible compared to the benefits. This is why LZMA, bzip3, and many modern compressors use range coding or variants.

In the next chapter we will turn to the other major family of compression algorithms: dictionary-based methods in the LZ family, which complement entropy coding by finding and eliminating repeated substrings.

Chapter 3: Dictionary Methods and LZ Algorithms

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

LZ77: Sliding Window and Match Vectors

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

LZ78 and the LZ Family Tree

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

Deflate's Approach to Dictionary Compression

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

Match Finding: Hash Chains vs Binary Search vs FM-Index

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

The Cost of Match Finding

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

Chapter 4: Block Transformations: BWT and MTF

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

Why Transform Before Compressing

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

The Burrows-Wheeler Transform Explained

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

Inverting the BWT: The LF-Mapping Property

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

Move-to-Front Coding

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

How BWT+MTF Creates Compressible Distributions

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

Chapter 5: The bzip2 Architecture

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

Block Size as a Design Parameter

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

Sorting Runs: The In-place Suffix Sort

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

Run-Length Encoding Before BWT

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

Huffman Coding with Multiple Passes

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

Where bzip2 Falls Short

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

Chapter 6: Block-Based Design Principles

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

Why Blocks at All

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

Block Size vs Compression Ratio vs Memory

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

Independence and Parallelism

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

Streaming with Blocks

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

Error Localization and Recovery

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

Summary: The Block Philosophy

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

Chapter 7: Range Coding Implementation

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

The 64-bit Range Coder

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

Handling Underflow and Renormalization

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

Byte Output Buffering

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

The Decoder Mirror

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

Testing Range Coder Correctness

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

Optimization Notes

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

Chapter 8: Context Modeling

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

Order-0 vs Order-N Models

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

Context Mixing and Blending

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

Adaptive Probability Updates

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

State Machines in Context Modeling

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

Context Modeling for BWT Output

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

The Cost of Context Modeling

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

Chapter 9: bzip3 Design Philosophy

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

Motivations Behind bzip3

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

The Modular Pipeline Approach

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

LZH vs BWT Modes

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

Choosing Between Pretransformation Strategies

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

The Role of Preprocessing

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

Chapter 10: The bzip3 Compression Pipeline

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

Chunking and Block Partitioning

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

Pretransformation and Dictionary Compression

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

Transform Selection and Encoding

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

Context Modeling the Transformed Stream

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

Range Encoding to Final Bits

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

Why This Pipeline Works

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

Chapter 11: Implementing a Reference bzip3-style Compressor

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

Project Layout and Build System

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

The Block Structure and API

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

BWT Implementation in C

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

MTF and Context Modeling Code

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

Putting It All Together

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

Chapter 12: SIMD and Vectorization

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

Where SIMD Helps in Compression

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

Vectorizing Match Finding

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

SIMD-Accelerated Sorting

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

Vectorized Range Coding

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

Portability and Auto-Vectorization

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

Measuring SIMD Benefits

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

Chapter 13: Parallel Compression Architectures

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

Parallel Patterns in Compression

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

Block-Level Parallelism

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

Shared Memory and Lock-Free Queues

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

Producer-Consumer Pipeline Design

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

Scaling Limits and Amdahl's Law

Chapter 14: Streaming Compression Design

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

Stream vs Block: Reconciling the Tension

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

Chunk Boundaries and Backreferences

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

Flushing and Partial Results

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

Seeking and Indexing

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

Handling Stream Corruption

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

API Design for Streaming

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

Chapter 15: Memory Engineering

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

Memory Footprint Analysis

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

Cache Line Behavior and Data Layout

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

Arena Allocators and Pooling

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

Reducing Allocation in Hot Paths

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

Low-Memory Mode Trade-offs

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

Memory and Parallelism

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

Chapter 16: I/O and System Integration

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

The I/O Bottleneck

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

Buffering Strategies

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

Asynchronous I/O with Compression

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

Pipe and Network Throughput

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

Compression on the Fly vs Batch

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

Chapter 17: File Format Design

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

Magic Numbers and Headers

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

Metadata and Flags

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

CRC and Integrity Checking

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

Forward Compatibility

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

bzip3 Format Decoded

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

Chapter 18: Benchmarking and Profiling

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

Compression Benchmarks: Tools and Standards

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

Designing Fair Comparisons

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

Measuring Throughput and Latency

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

Profiling with perf and Valgrind

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

Interpreting Benchmark Results

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

Chapter 19: The Broader Landscape

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

LZ4 and the Speed Extreme

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

Zstandard: Adaptive Compression

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

Brotli: Static Dictionary Power

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

xz/LZMA: Ratio Focused

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

When to Use Which Compressor

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

Chapter 20: Security and Correctness

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

Decompression Bombs and Limits

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

Bounds Checking Everywhere

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

Integer Overflow and Wraparound

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

Fuzz Testing Compression Code

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

Side Channels in Compression

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

CVEs and Real-World Vulnerabilities

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

Chapter 21: API Design and Library Engineering

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

Stream vs Batch APIs

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

Error Handling Semantics

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

Thread Safety and Initialization

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

ABI Stability

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

Example API Design

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

Chapter 22: Production Deployment

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

Compression on Servers and Storage

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

Network Compression Trade-offs

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

Choosing Levels and Parameters

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

Monitoring and Alerting

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

Migration Strategies

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

Chapter 23: Beyond bzip3

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

Neural and Learned Compression

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

Compression for Non-Text Workloads

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

Hardware Acceleration

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

The End of the Compression Line

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

Open Problems

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

Conclusion

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

References

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