

Article

Combinadic Compression: An Order-0 Enumerative Byte-Stream Codec with Size-Preserving Encryption

Anthony J. Rasmussen 

Shack C Technologies, Thornlands, QLD 4164, Australia; anthony@shackc.tech

Abstract

Order-0 enumerative coding represents a fixed-length block not as a self-delimiting bitstream but as a short tuple of integer fields over known finite ranges: each value's positions form a bitset, reduced to its colexicographic rank, leaving per-symbol counts and ranks. We study this *representation*, not a particular codec: one structural fact—finite-range integer fields—affords what a self-delimiting bitstream cannot. It is order-0-optimal, reaching the entropy bound up to an $O(A \log N)$ per-block overhead; at matched block sizes it is the most compact *block-resetting* order-0 coder (Huffman, range, asymmetric numeral systems (ANS)) on narrow alphabets, though a globally modeled coder wins at small blocks. Because every field is an integer over a known range, it also yields an exact compressed size known before encoding, independent random access, and an in-place size-preserving cipher (addition modulo each range) reducing to a standard keystream. Leaving the counts in clear exposes only a block's count spectrum—an adjustable leakage equal to its entropy, small on the homogeneous streams it best suits. Slower than streaming coders at equal ratio—35–199 MB/s compressing; 9–38 MB/s decompressing—it is not a general-purpose compressor but a structurally transparent design point, suited to record stores and constrained-alphabet data.

Keywords: lossless compression; enumerative coding; combinatorial number system; binomial coefficient; colexicographic ordering; size-preserving encryption; joint compression and encryption

1. Introduction

All digital data—photographs, documents, database tables—is ultimately a stream of bytes, and there is a great deal of it. Despite cheap, abundant storage and bandwidth, it remains worthwhile to compress data before writing it to disk or sending it over a network, and there is no shortage of utilities that do so: `gzip`, `bzip2`, `xz`, `zstd`, and many others.

The problem this paper addresses is deliberately narrow. When the unit of storage is a small, fixed-length, independently addressable block—a record in a store, a packet, a short read—the streaming order-0 coders in production (Huffman, range, ANS) still apply, but they forfeit, by their self-delimiting nature, three properties such settings want together: an exact compressed size known *before* encoding, random access to a block without its neighbors, and encryption that does not change the block's size. We ask what a single representation can deliver all three while remaining order-0-optimal, and take enumerative coding's finite-range-integer representation—not any particular codec—as the answer we characterize. In scope are order-0 compactness, these structural properties, and a size-preserving cipher; out of scope are higher-order modeling (dictionary, block-sorting, and



Academic Editor: Bruno Carpentieri

Received: 15 June 2026

Revised: 3 July 2026

Accepted: 29 July 2026

Published: 1 August 2026

Copyright: © 2026 by the author.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

context-mixing methods, against which we compare only to locate the order-0 frontier) and raw throughput, at which streaming coders are and remain faster.

Broadly, modern lossless compressors combine two ingredients. A *model* predicts or removes redundancy—dictionary and match-based methods such as LZ77/LZ78 [1,2], the Burrows–Wheeler transform [3], and adaptive context models [4]—and an *entropy coder* represents the residual symbols in close to their information-theoretic minimum. The entropy stage is itself a mature field: Huffman coding [5], arithmetic coding [6], range coding [7], and the more recent asymmetric numeral systems (ANS) [8] all approach the order-0 entropy of their input, differing mainly in speed and in how the coding state is carried.

This article concerns a less common member of the entropy-coding family: *enumerative coding* [9]. The idea is exact rather than statistical—one assigns every admissible sequence a unique index in the lexicographic (or, here, colexicographic) ordering of the set it belongs to, and transmits that index in place of the sequence. For the set of binary sequences of a fixed weight, the index is a sum of binomial coefficients, and both the index and its inverse are computed in closed form [10,11]. Enumerative coding meets the order-0 entropy bound by construction and requires no probability model, but it has seen limited use as a general byte-stream compressor, in part because the indices are large integers whose arithmetic is slower than a streaming coder's. A second limitation has kept it marginal as a *byte-stream* method: prior realizations operate on binary constant-weight sequences (bit by bit), a granularity at which byte-oriented data is near-incompressible (Section 2); coding the full byte alphabet through the *multinomial* factorization of the rank removes it and is what makes the representation a practical byte-stream codec.

Our object of study is the *representation* this coding induces, more than any particular codec. A stream is divided into fixed-length blocks; within each block the positions of every byte value form disjoint bitsets, which—after a length-reduction step that re-expresses each bitset in the coordinate space left by those already encoded—are replaced by their colexicographic ranks. A block is thereby a short tuple of integer fields—the per-symbol counts and these ranks—each over a known finite range. It is this one structural fact—finite-range integer fields rather than a self-delimiting bitstream—that affords a family of properties a streaming entropy coder cannot, and which we realize as a codec, characterize, and analyze.

The gap is thus specific. The order-0 entropy coders in production are streaming, self-delimiting bitstreams, and none yields from one representation an exact pre-encoding size, independent random access, and size-preserving encryption; enumerative coding's finite-range representation yields all three, but has been studied chiefly as a *binary* primitive rather than realised and characterised as a byte-stream codec. This paper closes that gap.

The contributions of this paper are:

- A realization of this representation as a byte-stream codec—the per-symbol ranking is classical (Cover; Schalkwijk), applied in its multinomial form block-wise over the full byte alphabet, the most-frequent value recovered by elimination (Sections 3–5);
- A characterization of the representation against mainstream coders—an allocation-light, block-parallel implementation (Section 7) whose compressed size and throughput we measure against the entropy coders of a modern compressor on the *enwik9* benchmark, finding it order-0-optimal and, on narrow-alphabet inputs, the most compact of these *block-resetting* coders at a matched block size, while on wide-alphabet data its per-block housekeeping overtakes the gain (Section 8);
- The setting these properties suit—the *homogeneous record store*, where small, externally fixed blocks are the unit of storage and access. There the representation supplies as a single package what a streaming entropy coder cannot: order-0-compact ratios that

lead the *block-resetting* order-0 coders on constrained alphabets, independent random access, an exact compressed size known before encoding, and the cipher below—with embedded pipelines, packet and record stores, and short-read genomics the natural settings (Section 9);

- A size-preserving cipher the representation admits directly (Section 6): because each field is an integer over a known range, every field is encrypted in place at no size cost. Full encryption matches per-record AES-CTR on size, access, and leakage; leaving the counts in clear gives a tunable, keyless computability whose disclosure we characterize information-theoretically as $I(\text{content}; \text{leak}) = H(S)$, the entropy of the count spectrum—small, with its query value, on homogeneous data. We present this as a structural property of the representation, with stated assumptions and leakage, not as a deployed feature.

The remainder of the article is organized as follows. Section 2 situates the method among related compression and joint-encryption schemes. Section 3 fixes notation and the ranking machinery; Sections 4 and 5 present the encoder and decoder; Section 6 develops the size-preserving cipher and its security; Section 7 describes the implementation; Section 8 reports experiments; Section 9 discusses applications; Section 10 weighs limitations and positioning; and Section 11 concludes.

2. Related Work

2.1. Enumerative Coding

The technique on which this work rests is enumerative source coding [9]: a sequence is represented by its index in the ordered enumeration of the set S of admissible sequences, so that storing the index achieves compression of $(\log_2 |S|)/n$ bits per symbol. Cover gives closed-form indices for two cases of particular interest, sequences of a fixed weight and sequences with a prescribed empirical Markov property, and an efficient inverse. The fixed-weight case had been treated earlier by Lynch [10] and Schalkwijk [11]—hence “Lynch-Davisson” or combinatorial coding—and was later subsumed by Rissanen’s arithmetic framework [12]. Our per-symbol step is exactly the enumerative coding of a constant-weight bitset, and the full block decomposition is the classical multinomial enumeration of a multiset permutation; the present contribution is therefore not the enumerative kernel but its realization as a practical byte-stream codec—the block-parallel implementation, the systematic comparison against mainstream coders, and the joint-encryption variant.

Borysenko and co-workers have independently developed a closely related line of work under the name *binomial compression* [13–15]. Their scheme encodes fixed-length blocks of binary sequences as binomial numbers—the colexicographic rank of a constant-weight bitset—and an adaptive variant adds an early-exit rule that bypasses the ranking step for blocks whose unit count falls outside an efficient compression range, recovering throughput at the cost of a ratio penalty of up to three to five percent [15]. The present work differs in three respects. First, where their scheme treats only binary (constant-weight) sequences, ours codes the full byte alphabet via the multinomial factorization of the rank—a product of binomial coefficients over all distinct symbol values. This distinction is consequential for compression quality: a bit-stream coder cannot capture correlations between bit positions within a byte, so for typical byte-oriented data such as text the bit-level entropy is near 1 bit per bit, and the binary scheme achieves almost no compression, whereas the byte-stream formulation attains the (substantially lower) order-0 byte entropy at every block size. Second, it provides a systematic experimental comparison against mainstream streaming coders—Huffman, range, and ANS—at matched block sizes on standard corpora. Third, it derives a joint compression-and-encryption construction from the finite-range property of the rank (Section 6). The binomial-number-system line has

itself gestured at protecting information from unauthorized access through the structure of the encoding, but that protection rests on the obscurity of the number system rather than a key; our construction is instead a keyed, size-preserving cipher whose confidentiality reduces to that of a standard keystream, with stated assumptions and leakage.

2.2. Statistical Entropy Coders

The practical alternatives to enumerative coding at the same (order-0) bound are the streaming statistical coders. Huffman coding is optimal among prefix codes but integral-length; arithmetic coding [6] and range coding remove the integrality penalty by maintaining a fractional interval; and asymmetric numeral systems [8] attain arithmetic-coding ratios at table-driven speeds and now underlie several mainstream compressors. Like enumerative coding, arithmetic coding maps a message to a number; the essential differences are that it is adaptive and renormalized, and that our index is an exact rank over a finite combinatorial set rather than a point in a real interval. As our experiments confirm (Section 8), these coders reach the same compressed size as our method while running substantially faster, so our motivation lies in the structural properties of the combinatorial representation rather than in throughput. Higher-order methods—block-sorting (the Burrows–Wheeler transform [3]) and context-mixing coders (the PPM [4] and PAQ/TPAQ family)—model redundancy above order 0 and so compress general text and genomic data substantially further; they are complementary rather than competing, and we include the strongest (TPAQ/TPAQX) in Section 8 only to locate the order-0 frontier our method targets.

2.3. Succinct Combinatorial Representations

The same combinatorial index that we use to compress a bitset is the basis of succinct data structures. The indexable-dictionary scheme of Raman, Raman and Rao [16] encodes a bitvector as a (class, offset) pair—the class being its weight and the offset its enumerative index—and supports constant-time rank and select queries on the compressed form. Our encoder produces the same kind of object per byte value, which makes the construction a natural building block for compressed-but-queryable representations such as bitmap indexes and inverted-index posting lists.

2.4. Joint Compression and Encryption

Because the entropy stage of a compressor maps data onto a near-uniform code, it is a natural place to introduce a cipher. Randomized arithmetic coding [17] encrypts by making the coder’s interval assignment depend on a secret key, achieving compression and encryption together at negligible cost in efficiency; related schemes randomize binary arithmetic coders or Huffman trees. Our variant follows the same principle but exploits a property specific to enumerative coding: every field of a block is an integer over a known finite range, so it can be encrypted by a modular cipher that is exactly size-preserving (Section 6), rather than by randomizing the internal state of a streaming coder.

Format-preserving encryption (FPE) is the direct point of comparison. It encrypts a value so that the ciphertext inhabits the same domain—an integer in $[0, M)$ to an integer in $[0, M)$. The standard constructions are pseudorandom permutations over the domain, built as Feistel networks: the FFX mode of Bellare, Rogaway and Spies [18], standardized by NIST as FF1 and FF3 in SP 800-38G [19], descends from the problem of ciphers over arbitrary finite domains [20]. That line has proved delicate: FF3 fell to Durak and Vaudenay [21], its repaired variant FF3-1 to Beyne [22], and a 2025 draft revision of the standard removes FF3 altogether, leaving FF1. Our cipher achieves the same domain preservation—each field’s ciphertext stays in its own range $[0, M)$, so the encrypted block is exactly the size of the plaintext—and we call it *size-preserving encryption*, reserving “format-preserving” for the permutation-based family above. It is a different primitive: not a deterministic

pseudorandom permutation but a randomized additive stream cipher over $\mathbb{Z}_M$ (a one-time pad keyed by a nonce-addressed keystream), whose confidentiality reduces to that of a standard stream cipher rather than to a bespoke Feistel network. The trade is deliberate. We give up the deterministic, equality-preserving behavior that permutation-based FPE provides, and that some tokenization settings rely on, in exchange for semantic security and a reduction to a well-understood primitive that has not needed the repeated repair the FFX line has; the cost is that an additive cipher is malleable and so wants per-block authentication, which we treat as deferred hardening (Section 6).

These FPE breaks are all black-box cryptanalysis; a distinct line concerns *side-channel* and hardware leakage—timing, cache, and power analysis—of cipher *implementations*. That axis is orthogonal to the present construction: our threat model (Section 6) is the honest-but-curious storage adversary and explicitly excludes side-channel attackers, so a constant-time, hardened deployment is—like authenticated integrity—a matter for the vetted implementation we defer to.

We develop the full construction—its tunable leakage–queryability dial and a security analysis—in Section 6.

3. Mathematical Preliminaries

Let a bitset contain N bits and have a population count of k . Bitset b is one of many possible bitsets associated with the tuple (N, k) , or $b \in B_{N,k} = \{b_0, b_1, \dots\}$. The cardinality of $B_{N,k}$ is given by the usual binomial coefficient

$$|B_{N,k}| = \binom{N}{k} = \frac{N!}{k!(N-k)!}. \quad (1)$$

Using colexicographic ordering, there is a ranking function to obtain the index value of each b in $B_{N,k}$:

$$r(b) = \sum_{i=1}^k \binom{p(b,i)}{i} \quad (2)$$

where $p(b,i) \in [0, N-1]$ gives the i th *on* position within b . This rank is the *combinadic* of b —its representation in the combinatorial number system [9,11,23], which gives this paper its title.

Clearly, for a known value of N , bitset compression can be achieved with a ranking function whenever

$$\underbrace{\lceil \log_2(N+1) \rceil}_{\text{cost of storing } k} + \underbrace{\lceil \log_2 \binom{N}{k} \rceil}_{\text{cost of storing } r} < N. \quad (3)$$

We can recover the ordered set $\{p(b,k), \dots, p(b,1)\}$ from $r(b)$ and k by unranking as follows

$$p(b,i) = \max \left\{ j : \binom{j}{i} \leq r(b) - \sum_{\ell=i+1}^k \binom{p(b,\ell)}{\ell} \right\}. \quad (4)$$

4. Compression Algorithm

For a large file, it is tempting to consider a single compression step as prescribed in Equation (2). This is rarely practical, however, as it leads to the expensive computation of binomial coefficients with very large arguments. Instead, it is prudent to consider compressing smaller, fixed-width blocks.

The next temptation is to apply compression to all raw bits in every block. While this is certainly less expensive than single-step compression, more efficiencies become available by analyzing the bytes within each block.

Consider a block containing 20 bytes in which only five byte values occur: d b c b c e a e e c b a c d c e c b e b. We map each byte value to a bitset marking the positions it occupies within the block—the per-value bitsets in the second stage of Figure 1.

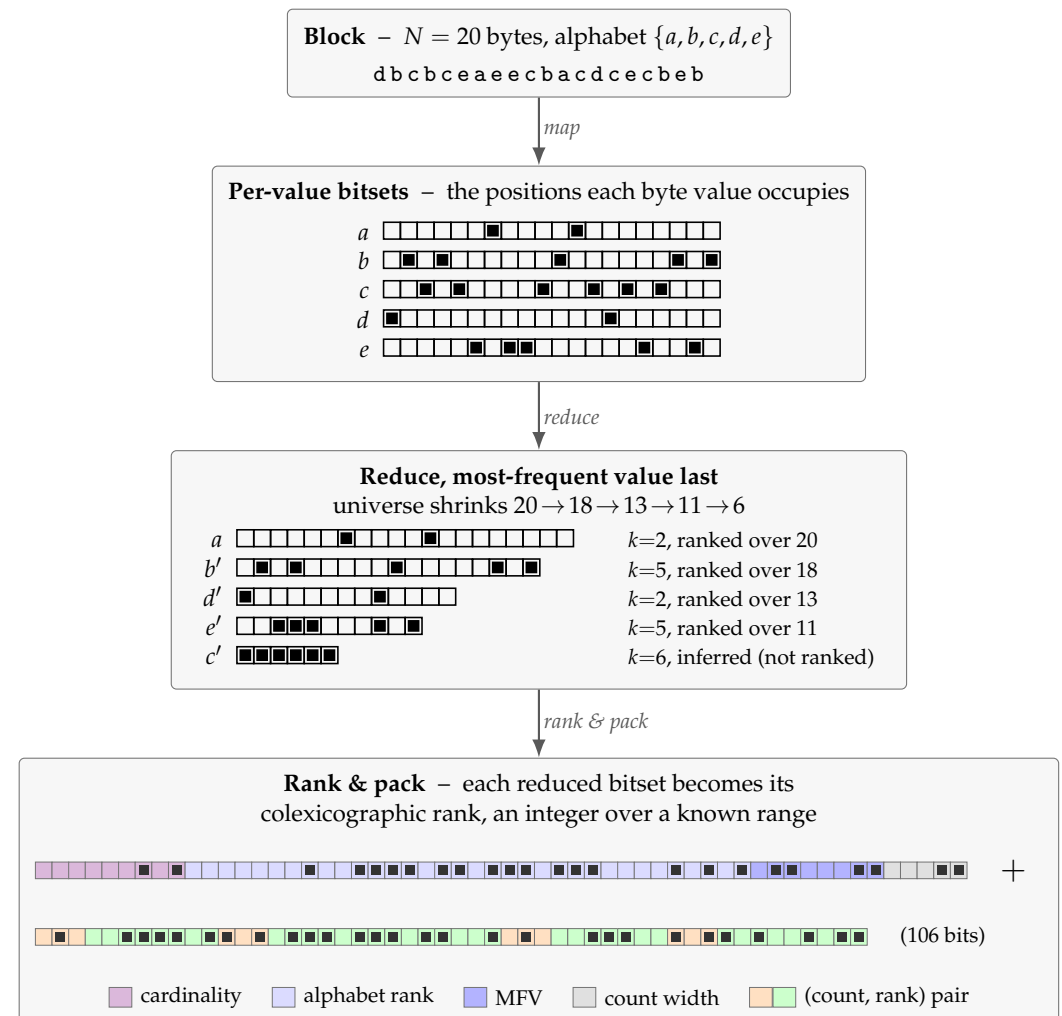


Figure 1. The per-block pipeline on the running example, the 20-byte block d b c b c e a e e c b a c d c e c b e b over $\{a, b, c, d, e\}$: each byte value's positions form a bitset, reduced in most-frequent-last order and replaced by its colexicographic rank. The final stage shows the *actual* packed block bit for bit (filled = ■), tinted by field—the header (cardinality, alphabet rank, most-frequent value (MFV), count width) and the per-value (count, rank) pairs, 106 bits in all.

To reduce these disjoint bitsets, we traverse them in a fixed order and drop, from each, the zeroes that fall on positions already claimed by the bitsets before it (their cumulative disjunction); each is then ranked over only the positions that remain, and the final bitset can be inferred rather than ranked. Taken in natural sort order, this costs 15 binomial coefficients. A small change lowers it to 14: handle the most frequently occurring value last—value c here, with six occurrences—so the largest bitset is the one left to inference, recovered from the positions no other value claims (the reduction stage of Figure 1).

Each reduced bitset is ranked over a universe whose size is the number of positions not yet claimed by the values before it. As the values are processed, this universe shrinks—in the example from 20 to 18, 13, and 11 positions—so the binomial arguments, and with them the ranks, become progressively smaller.

To invert this process, the decoder must also be told which byte values occur. The set of occurring values is itself a subset of the 256 possible bytes, so it is encoded with the same ranking function: we store its cardinality A and the rank of that subset among the $\binom{256}{A}$ subsets of that size (in the example $A = 5$). Successive blocks of a homogeneous stream usually share most of their alphabet, which a second, *stream* mode exploits to amortise the alphabet across the whole stream (Section 4.1); the description so far is the self-contained *per-record* mode.

A compressed block is therefore the concatenation of the alphabet (its cardinality A and rank); the identity of the most frequent value, stored verbatim as the byte that the omitted final bitset represents (it cannot be deduced from the alphabet alone, since the counts are not yet known to the decoder); a single per-block count width; and, for each remaining value in turn, its population count k followed by the rank of its reduced bitset. The counts share that width— $\lceil \log_2(k_{\max} + 1) \rceil$ bits, where $k_{\max}$ is the largest stored count, hence at most $\lceil \log_2(N + 1) \rceil$, and the width is sent once per block—while each rank takes exactly $\lceil \log_2 \binom{n}{k} \rceil$ bits for the current universe size n , the fewest that can distinguish the $\binom{n}{k}$ possibilities, after which n is reduced by k before the next value is processed—the packed fields of the final stage of Figure 1. Decompression simply reverses these steps (Section 5).

The reason this construction is order-0-optimal is that the per-value ranks telescope exactly onto the block's multinomial coefficient. Because each value is ranked over the universe *left by* those before it, the ranges multiply to the number of distinct arrangements of the block's multiset, so their summed widths are the exact cost of naming the arrangement once the counts are known.

Proposition 1 (Order-0 optimality). *Let a block consist of N bytes in which the distinct values $v_1, \dots, v_A$ occur with multiplicities $k_1, \dots, k_A$, processed in the encoder's order so that the most frequent value is v_A . Write $n_1 = N$ and $n_{i+1} = n_i - k_i$. The encoder spends on the ranks alone*

$$\sum_{i=1}^{A-1} \left\lceil \log_2 \binom{n_i}{k_i} \right\rceil \text{ bits,} \quad (5)$$

the final value v_A requiring no rank. Dropping the ceilings, the ranges telescope,

$$\prod_{i=1}^{A-1} \binom{n_i}{k_i} = \binom{N}{k_1, \dots, k_A} = \frac{N!}{k_1! \cdots k_A!}, \quad (6)$$

the multinomial coefficient—exactly the number of distinct arrangements of the block given its counts. The integer ceilings inflate the payload by at most $A - 1$ bits per block, and the counts and alphabet add a further $O(A \log N)$ header bits, so the per-byte cost is

$$\frac{1}{N} \log_2 \frac{N!}{k_1! \cdots k_A!} + O\left(\frac{A \log N}{N}\right) \xrightarrow{N \rightarrow \infty} H_0, \quad (7)$$

where $H_0 = -\sum_v \frac{k_v}{N} \log_2 \frac{k_v}{N}$ is the block's empirical order-0 entropy. The scheme thus attains the order-0 bound, the housekeeping term vanishing as the block grows or the alphabet narrows.

The telescoping is the identity $\prod_i \binom{n_i}{k_i} = N! / \prod_i k_i!$ with $n_{i+1} = n_i - k_i$; the omitted final factor $\binom{k_A}{k_A} = 1$ is why the most-frequent value is recovered by elimination at no cost. This accounting also locates the method's only inefficiency relative to the entropy floor: the $\leq A - 1$ bits of per-block ceiling loss and the $O(A \log N)$ header, both proportionally largest for small blocks over wide alphabets—precisely the regime the experiments of Section 8 probe. The header term is a known quantity rather than a peculiarity of our scheme: the counts are the block's *type*, and as an N -symbol string over an A -symbol alphabet has at

most $(N + 1)^{A-1}$ types, naming one costs at most $(A - 1) \log_2(N + 1)$ bits—the standard method-of-types bound [24], which our $O(A \log N)$ matches in order. The header is itself compressible: the count vector is a composition of N into A parts, an enumerable set of size $\binom{N-1}{A-1}$, so the counts could be ranked exactly as the arrangements are—trading the fixed-width $O(A \log N)$ header for $\lceil \log_2 \binom{N-1}{A-1} \rceil$ bits—the header’s last remaining slack, the alphabet already being ranked. We leave this refinement to future work.

Because each block is encoded entirely independently of the others, a stream may be compressed in parallel, one block per worker, and the compressed blocks reassembled in their original order. The per-block cost is dominated by the large-integer additions that accumulate each rank, as discussed in Section 7.

4.1. Two Modes: Stream and Per-Record

The treatment of the alphabet is the codec’s one point of contact between blocks, and it admits two modes of different scope. *Per-record* mode treats every block in isolation: each stores its own alphabet—the cardinality A and the rank of the occurring-value subset—with no reference to any other block. This is the right mode when the units compressed are independent: a single packet, a lone record, a message whose neighbors are unrelated. Here the block is genuinely the unit of storage, access, and—as Section 6 develops—encryption.

Stream mode exploits the common case in which the blocks form a *homogeneous stream*: successive records of one kind that share an alphabet—a column of a table, a log of like-formatted entries, packets of a single protocol, sequencing reads over $\{A, C, G, T\}$. A reference alphabet is derived once from the start of the stream and transmitted a single time; each block then stores only the symmetric difference in its alphabet against that reference, usually a handful of bits, amortizing the cost of naming the alphabet across the whole stream. This costs nothing in independence: given the reference (read once and cached), any block still decodes, decrypts, and is accessed without its neighbors, so the parallelism above and the random access of Section 6 are retained.

The two modes share the entire ranking machinery and differ only in this single field, selected by a flag. They suit different data—per-record for independent units, stream for homogeneous record streams—and Section 8.2 quantifies the amortization stream mode buys and shows that it reverses, per-record mode winning, when a block’s alphabet varies too much for a fixed reference to fit.

5. Decompression Algorithm

Decompression reverses the encoder field by field, reconstructing each block from its header and ranks. The decoder first reads the 9-bit population count of the block’s alphabet and the rank of that alphabet, and unranks it over the 256 possible byte values to recover the set of values present, undoing the symmetric difference against the reference alphabet. It then reads the most-frequent byte value and the field width used for the per-symbol counts, and orders the alphabet exactly as the encoder did, with the most-frequent value placed last.

For each byte value in turn (save the last), the decoder reads the count k and then the rank r stored in $\lceil \log_2 \binom{n}{k} \rceil$ bits, where n is the number of positions still unfilled. The rank is unranked into a reduced bitset by the inverse of (2): working from the most significant position down, it repeatedly applies the unranking recurrence (4), each step locating the largest j whose binomial coefficient does not exceed the running remainder r and subtracting that coefficient. Because the half-diagonal cache stores coefficients in descending order for a fixed lower index (Section 7), this search is a sequential scan; techniques for shortening it have been studied [25].

The reduced bitset gives the symbol's positions in the coordinate space of the *currently unfilled* slots. The decoder therefore rehydrates them by walking the unoccupied positions in order and assigning the symbol to those whose ordinal the reduced bitset selects, inverting the length-reduction of the encoder; the chosen slots are then marked occupied so that subsequent symbols see the same shrinking universe the encoder did. After all but one value have been placed, the most-frequent value is written into every remaining slot, which requires no stored rank. The block is thereby reconstructed exactly, and the appended XXH3 hash [26] confirms the round trip.

6. Size-Preserving Encryption and Queryable Ciphertext

The same representation that yields order-0-optimal compression also yields a cipher. A compressed block is a concatenation of integer fields, each over a range known to both encoder and decoder: the alphabet rank over $[0, \binom{256}{A})$, the most-frequent value as an index over $[0, A)$, each count k_i over $[0, n_i]$, and each position rank r_i over $[0, \binom{n_i}{k_i})$. Any such field, a value in some $[0, M)$, can be encrypted in place by an additive cipher over $\mathbb{Z}_M$,

$$y = (r + s) \bmod M, \quad r = (y - s) \bmod M, \quad (8)$$

where $s \in [0, M)$ is a key-derived keystream value. The ciphertext again lies in $[0, M)$ and so occupies exactly $\lceil \log_2 M \rceil$ bits: compression and encryption are obtained together at no increase in size. This is the size-preserving property, and it is specific to the enumerative representation—a streaming entropy coder emits a self-delimiting bitstream rather than an integer over a known range, and does not admit the construction directly.

6.1. Every Field, in One Pass

The construction is not limited to the position rank. The decisive structural fact is that the decoder reads a block strictly front to back, and *each field's modulus depends only on the block length and the fields already decoded, never on the field's own value*: the cardinality A fixes the alphabet modulus, and the running universe n_i , fixed by the earlier counts, fixes every subsequent modulus. The fields can therefore be decrypted in the very pass the decoder already performs, and in principle the *entire* block can be encrypted, not merely its position ranks. The “field widths derive from the counts” obstruction one might fear applies only to random access *within* a block—reading a field without its predecessors—which the format never requires.

6.2. A Leakage-Queryability Dial

Both settings preserve random access *across* blocks—each block decrypts on its own given the reference alphabet—so the choice between them is not about reaching a record but about what an *untrusted* party can compute without the key. That choice is a dial between confidentiality and queryability, with the counts as the pivot.

At the confidential end, every field is encrypted (the full-encryption setting above): a block is opaque but for its compressed *length*—essentially its order-0 codelength, and unavoidable for any size-preserving joint scheme—and is decrypted strictly front to back by the key holder.

At the queryable end, the per-symbol counts are left in clear while the alphabet rank and the position ranks are encrypted. Because the rank fields have data-dependent widths $\lceil \log_2 \binom{n_i}{k_i} \rceil$, the cleartext counts form a *skip-table* over the field boundaries, so a storage server or index that never holds the key can still compute over the ciphertext: a block's count *spectrum*—its histogram shape, hence its order-0 entropy and size—and the field offsets needed to repack, route, deduplicate, or filter on that spectrum. The byte *labels* stay hidden, since the alphabet rank is encrypted, and so does the arrangement, since the

position ranks are; what leaks is exactly the spectrum. This is the trade that adjustable, query-exposing encryption makes in encrypted databases such as CryptDB [27]: a bounded, deliberate leak buys computation over data that is never decrypted. That bargain must be struck with eyes open—the inference attacks of Naveed, Kamara and Wright [28] showed that frequency and property leakage in encrypted databases is often far more revealing than early designs assumed, and the count spectrum is exactly such material: a per-block histogram that can fingerprint a block’s contents even with its byte labels hidden. The dial makes that disclosure explicit and optional rather than safe; where it is unacceptable, full encryption leaves only the per-block length. The same skip-table lets a key holder decrypt one rank field in isolation, but that is a side effect—one byte value’s occurrence set is seldom a meaningful unit on its own—not the reason for the setting.

The design point is therefore narrow and worth stating plainly: if no semi-trusted party must compute over the store, full encryption—equivalently, per-record AES-CTR (Section 8.6)—suffices, and the cipher’s value is then its structural integration into the coding pass alone; the dial earns its place precisely when an untrusted intermediary needs to work over the encrypted records and the count spectrum is an acceptable disclosure.

6.3. What It Conceals, on What Assumption, and Against Whom

Threat model. The adversary is honest-but-curious: a storage server, backup, or index that holds the ciphertext—and, in the queryable setting of Section 6.2, the cleartext counts—but never the key. It may read and compute over what it holds; we do not assume it provides authenticated integrity, which we add separately as per-block authentication (below). The question is how much it learns about a block’s plaintext from the ciphertext.

The per-field guarantee is Shannon’s. For a fixed field with modulus M , the map $r \mapsto (r + s) \bmod M$ with s uniform on $[0, M)$ is a one-time pad over $\mathbb{Z}_M$: the ciphertext Y is uniform and independent of the plaintext R , so the mutual information $I(R; Y) = 0$ and $H(R | Y) = H(R)$ —a single field leaks nothing about its value beyond the public modulus M . The keystream value s must be unique to each (key, nonce) pair—reuse degenerates to a two-time pad—so we make the nonce explicit rather than leave it to discipline. The reference implementation (the accompanying Go codec, Section 7) packs a domain tag, block index, and field index into one 64-bit nonce, [domain:16 | block:32 | field:16], and keys a stream cipher (AES-CTR or ChaCha20, or the SHA-256 reference generator measured in Section 8.6) with a per-file subkey derived from the master key (raw key material—ideally 32 random bytes, reduced to a 32-byte key by SHA-256 otherwise) and a random per-file salt stored in the header, so every (block, field) pair within a file gets a distinct nonce and identical nonces across files encrypt under different keys. With a pseudorandom rather than truly uniform keystream, the guarantee becomes computational: the construction is IND-CPA (indistinguishability under chosen-plaintext attack) under the standard PRG (pseudorandom generator) assumption, the standard result for an additive stream cipher with unique nonces [29]; Supplementary Materials S1 states and proves this precisely, as an IND-CPA guarantee relative to the leakage function below.

What remains is a small, explicit *leakage budget*, and it is exactly what the dial of Section 6.2 trades. Under full encryption, the only leak is the per-block compressed length—essentially the block’s order-0 codelength, the $\log_2$ of its multinomial, and unavoidable for any size-preserving joint scheme; it is removable only by padding to a fixed size. The queryable setting additionally exposes each field’s modulus M , a function of the counts, so that what leaks is the count *spectrum*: the histogram shape, hence the block’s order-0 entropy H_0 and size, but neither the byte labels (the alphabet rank stays encrypted) nor the arrangement (the position ranks do). The arrangement is protected twice over—each position rank is encrypted, and length-reduction expresses each reduced bitset in the

coordinate space left by its predecessors, so recovering any value's absolute positions needs the entire encrypted prefix. A keyless party thus gains exactly $I(\text{content}; \text{leak}) = H(S)$ bits about a record, the entropy of its count spectrum, which is also what upper-bounds the queryability the disclosure buys; on the homogeneous stores the codec targets records share an alphabet and near-identical counts, so $H(S)$ is small by construction, and the dial is most informative on heterogeneous data. Supplementary Materials S1 makes the confidentiality claim precise—a formal IND-CPA guarantee relative to a defined leakage function, with proof—and quantifies $H(S)$ across block sizes and skew.

Encrypting the counts too—each k_i a value in $[0, n_i]$, a range fixed by the decrypted prefix—closes the spectrum, leaving only the length. An exactly uniform keystream requires rejection sampling on the reduction modulo M ; a few extra keystream bytes before reduction make the modular bias cryptographically negligible.

6.4. Practicalities

Two changes complete the full-encryption setting. The stream-global reference alphabet, against which each block's alphabet is a symmetric difference, is itself the stream's alphabet identity and must also be encrypted—a single 256-bit value, decrypted once when the stream is opened and cached, at negligible cost. And the per-block count-width header—the single field width shared by all counts—is itself encrypted, so that the field boundaries still follow from the decrypted prefix: the decoder recovers the width before it parses the counts. Each fixed-width field (the cardinality, the most-frequent value, the count-width header, and every count) is then encrypted over its field modulus exactly as the variable-width ranks are, so the block remains the same size as its unencrypted form. The reference implementation integrates this cipher into both compression modes and a command-line tool, and its exact size-preservation and throughput cost are measured in Section 8.6. The additive cipher is malleable, as any stream cipher is, so a wrong key or a tampered block should fail cleanly rather than decode to garbage; Supplementary Materials S2 shows why strict size-preservation *precludes* integrity (a length-preserving map is a bijection, with no reject state) and gives an optional, costed authenticated mode. A hardened realization with a vetted stream cipher remains the natural next step, and any deployment should be reviewed by a cryptographer.

7. Implementation

7.1. Language and Arithmetic

The reference implementation is written in Go, using the standard library's `math/big` package for arbitrary-precision integers. The dominant cost of compression is large-integer addition—the accumulation of binomial coefficients that forms each rank $r(b)$ in (2). We found this to perform comparably whether backed by Go's `math/big` or by the GMP (GNU Multiple Precision Arithmetic Library) C library [30], so the portable standard-library backend is retained.

7.2. Caching Binomial Coefficients

Both ranking and unranking evaluate $\binom{n}{k}$ for many arguments, so the coefficients are precomputed once into a table generated by Pascal's rule. Rather than store the full triangle, we keep only its *half-diagonals*: by the symmetry $\binom{n}{k} = \binom{n}{n-k}$ every query is served from the smaller half, halving the storage. The table is oriented so that each row is a diagonal of the triangle. This matches the access pattern of the decompressor's unranking loop, which, for a fixed i , scans descending $\binom{j}{i}$ in search of the largest j with $\binom{j}{i} \leq r$ (Section 5); the diagonal layout turns that search into a sequential, cache-friendly walk. Alongside each coefficient we cache its bit length. This serves two ends: it supplies the exact field width $\lceil \log_2 \binom{N}{k} \rceil$ for

storing a rank, and it enables a fast magnitude test—two big integers can be ordered by bit length alone whenever those lengths differ, so the expensive full comparison is reached only on a tie.

This cache is also what sets the method’s practical block ceiling, and the limit is memory rather than the 16-bit block field. The half-triangle holds $\Theta(N^2)$ coefficients whose individual widths grow with N , so its footprint grows faster than quadratically—about $0.0154 N^3$ bytes, cubic in the block size (Supplementary Materials S3 derives this)—the moderate blocks of our experiments ($N \leq 8192$) are comfortable, but blocks in the tens of thousands are infeasible long before the format’s nominal 65,535 maximum. We cache the bit length of each coefficient in a 16-bit field, which suffices precisely because the memory wall keeps N well below the point at which a central coefficient’s bit length would reach 2^{16} ; the implementation enforces this with an explicit bound so that the failure mode is a clear error rather than silent overflow.

7.3. Reducing and Ranking Positions

After a symbol’s absolute positions are reduced into the shrinking residual universe, their rank is taken with (2). Because the magnitude of that rank is bounded by the corresponding $\binom{n}{k}$, whose bit length is already known, the big-integer accumulator is grown to that capacity *once* before the additions begin, avoiding the repeated reallocation that would otherwise occur as it fills word by word. Positions are grouped by byte value with a counting sort—a single counting pass followed by a prefix-sum scatter into one contiguous array—rather than 256 separately growing lists; this keeps the grouping allocation-free and cache-local, and the per-byte frequencies it produces also identify the most-frequent value, which is handled last. Asymptotically, ranking a block costs $N - k_A$ big-integer additions on an accumulator that grows to the rank’s width of $O(N)$ bits— $O(N^2/w)$ machine-word operations at word width w —while the counting sort and bit packing are $O(N)$ and the cached coefficient lookups $O(1)$; the quadratic big-integer term is what makes throughput fall as N grows, and decompression adds the downward unranking search of Section 5.

7.4. Bit Packing

Each block is emitted as a tightly packed bit stream: a 9-bit population count for the block’s alphabet, the colexicographic rank of that alphabet within the 256 possible byte values, the most-frequent byte, a per-block count width, and then, for every remaining symbol, its population count k followed by its rank r . We store each k in $\lceil \log_2(k_{\max} + 1) \rceil$ bits— $k_{\max}$ the block’s largest stored count, at most N , the width itself written once per block—and each r in exactly $\lceil \log_2 \binom{n}{k} \rceil$ bits, the minimum that distinguishes the $\binom{n}{k}$ possibilities. Bits are written most-significant-first through a small hand-rolled accumulator and read back by its exact inverse, so packing a field allocates nothing and stays branch-light.

7.5. Contiguous Runs

When a reduced position set contains a run of consecutive values, the corresponding terms of (2) form a contiguous diagonal segment summable in closed form by a hockey-stick identity [31], trading run detection for fewer big-integer additions. It is implemented and validated but disabled by default: detection outweighs the saving on the small-to-moderate blocks measured here, and it pays off only on long runs in large blocks.

7.6. Parallelism and Integrity

Because blocks are compressed independently, the implementation pipelines them across a fixed pool of worker goroutines: a reader dispatches blocks, the workers rank them concurrently, and a single collector writes the results back in block order. To amortize inter-thread hand-off when blocks are small, blocks travel through the pipeline in batches whose

size scales inversely with the block length, keeping the in-flight memory roughly constant across block sizes. Per-block working memory—bitsets, index buffers, the accumulator, and the bit writer—is drawn from a reusable pool, so steady-state compression performs no per-block heap allocation. Finally, a 64-bit XXH3 hash of the original stream is appended and re-verified on decompression, guaranteeing that the round trip is lossless.

8. Computational Experiments

Because our method is a per-block order-0 enumerative coder, the fair point of comparison is an entropy coder alone—and at a *matched block size*, since both our method and a block-based entropy coder pay a per-block overhead that amortizes only as the block grows. We preview where the comparison lands, so the figures below are read in context: the codec is order-0-optimal and, at a matched block size, the most compact of the block-resetting streaming coders on narrow-alphabet data, but a coder granted one global model is more compact still at small blocks; its case rests on the structural package of Section 9, not on ratio supremacy. We evaluate on the standard `enwik9` benchmark—the first 10⁹ bytes of an English Wikipedia dump (1 GB)—against Kanzi 2.4.0 [32] with its transforms disabled, running its order-0 entropy coders at block sizes matched to our block *N*, and, as a large-block optimum, at Kanzi’s default “auto” block. All measurements were taken on an Apple M3 (eight logical cores), Go 1.26.3, with both compressors using eight concurrent jobs. Compressed fractions are deterministic (bit-identical run to run); throughputs are wall-clock trimmed means over ten runs, discarding the slowest and fastest; the coefficient of variation of the retained samples is at most 6% for the codec across all block sizes (and comparable for the baselines), so the throughput differences reported below reflect genuine gaps rather than run-to-run noise. Unless stated otherwise, the codec runs in *stream* mode (Section 4.1); the per-file Silesia results below isolate the alphabet amortization this mode provides over per-record mode. Each table in this section is reproduced by a named script under `experiments/scripts/` of the reference implementation, with the saved outputs in `experiments/results/`; the repository README maps every table to its script. The computational experiments reported in this section were designed by the author and were implemented and scripted with the assistance of Claude (Anthropic); the author ran them, verified every reported figure against the reference implementation, and is solely responsible for their design and interpretation. Table 1 sets our codec against all three order-0 streaming coders. At small blocks, Huffman achieves the best compressed fraction among them—its simpler per-block header incurs less overhead than ANS0’s coding tables—while ANS0 and Range converge to the order-0 entropy floor first at large blocks.

Table 1. Compression of `enwik9` (1 GB) on an Apple M3, eight jobs, no transform, at matched block size *N*. Lower fraction is better; throughput (comp./decomp.) is in MB/s, higher being faster. Throughput for Huffman and Range is comparable to ANS0 at each *N* and is omitted for brevity. Kanzi’s minimum block is 1024; its “auto” block is the large-block optimum at which it attains the order-0 entropy.

N	This Work			Kanzi ANS0			Frac. Only	
	Frac.	Comp.	Decomp.	Frac.	Comp.	Decomp.	Huffman	Range
512	0.6835	199	38	-	-	-	-	-
1024	0.6595	186	30	0.6994	80	127	0.6670	0.6767
2048	0.6461	119	21	0.6657	140	202	0.6504	0.6572
4096	0.6400	57	12	0.6485	224	300	0.6426	0.6464
8192	0.6376	35	9	0.6406	334	402	0.6396	0.6396
auto	-	-	-	0.6375	1019	917	0.6398	0.6374

On `enwik9`, at every matched block size our codec produces a *smaller* compressed fraction than all three streaming coders—an advantage that, as Section 8.1 shows on a heterogeneous corpus, is specific to narrow-alphabet inputs such as text. Its per-block housekeeping—the alphabet and the symbol counts—is lighter than any block-based entropy coder’s header, an overhead that Kanzi’s store mode (None) exposes directly: 0.5% expansion at $N = 1024$, falling to nothing at the auto block. At $N = 1024$ we reach 0.6595 against 0.6670 for Huffman—the most compact of the streaming coders at this block size—and 0.6767 for Range and 0.6994 for ANS0. The advantage persists, narrowing as the block grows. Our codec and the streaming ANS0 and Range coders all converge to about 0.638—below `enwik9`’s static order-0 entropy of 0.6446, because all three are *block-adaptive*, re-estimating symbol frequencies locally rather than coding against one global table, which on this non-stationary stream pays off: the streaming coders reach it only at their large auto block (0.637–0.640), whereas our method reaches 0.638 already at $N = 8192$. This confirms that the combinatorial ranking is *order-0-optimal*, and that at equal granularity it is the most compact of the *block-resetting* order-0 coders on this text input. A coder that instead shares one global model is a sterner baseline, which we add next (Figure 2); Section 8.1 makes the alphabet-width qualification precise.

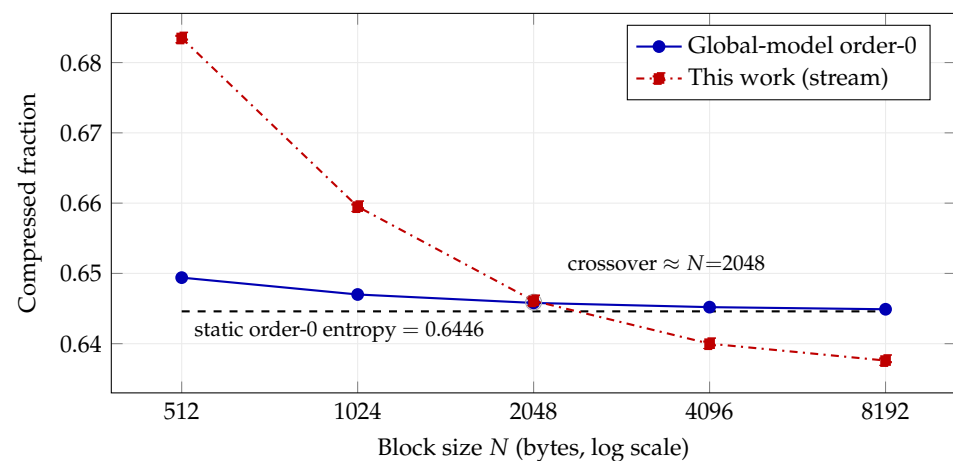


Figure 2. Compressed fraction versus block size on `enwik9`: the idealized globally modeled order-0 baseline (one byte-frequency model shared over the whole stream, each block independently framed) against our codec’s stream mode. The two cross near $N = 2048$. The static global model is more compact at small blocks, where it avoids the codec’s $O(A \log N)$ per-block header; the block-adaptive codec wins at large blocks by tracking the stream’s non-stationarity, dropping below the corpus’s static order-0 entropy (0.6446, dashed) that lower-bounds any single-model coder. Underlying fractions (global-model order-0/this work, stream) at $N = 512, 1024, 2048, 4096, 8192$: 0.6494/0.6835, 0.6458/0.6595, 0.6452/0.6461, 0.6449/0.6400, 0.6449/0.6376.

One aspect of this comparison deserves to be stated plainly, because it separates two effects the matched-block framing can blur. Our codec does not treat each block’s alphabet in isolation: a reference alphabet is derived once from the start of the stream and transmitted a single time, and every block then stores only the *symmetric difference* of its own alphabet against that reference (Section 4), amortizing the cost of naming the alphabet across the whole stream—whereas a block-based entropy coder at the same block size re-pays its table or alphabet overhead in every block. Part of the small-block advantage is therefore this amortization, not the coding kernel. The clean, kernel-only comparison is *per-record* mode, in which the codec—like the block-resetting streaming coders—names each block’s alphabet from scratch: there (Section 8.2) it reaches 0.6676 at $N = 1024$, still bettering Range (0.6767) and ANS0 (0.6994) and tying Huffman (0.6670). The coding kernel is thus competitive even with the model-sharing scope removed; the additional edge stream

mode shows is alphabet amortization, which a globally modeled baseline would equally enjoy. The reader should read the matched-block gap in that light, not as a coding-kernel difference alone.

That sterner baseline grants a static coder the *same* global model-sharing scope our stream mode takes: one byte-frequency model fixed over the whole stream, each block coded against it and independently framed, so per-block access is retained. It is an idealized globally modeled order-0 coder—a lower bound for any real static arithmetic or range coder of this kind. Figure 2 sets it against our stream mode at matched blocks, and the two cross over near $N = 2048$: below it the global model is the more compact, because our per-block counts are overhead it does not pay; above it our codec wins, its per-block counts adapting to the stream's non-stationarity faster than a single global table can. Against a model-sharing coder, then, the small-block ratio is not where the method's case lies—there its value is the structural package of Section 9—while at larger blocks its per-block adaptivity earns a clear edge.

The price is throughput, and the picture is two-sided. Our compression falls from 199 to 35 MB/s, and decompression—which additionally searches during unranking—from 38 to 9 MB/s, as the block grows from 512 to 8192, because the per-block ranking is super-linear in N . Kanzi's coders run the other way: at small blocks their fixed per-block cost dominates, so at $N = 1024$ all three streaming coders compress at only ≈ 80 MB/s (ANS0; Huffman and Range comparable), *slower* than our 186; but they scale up with block size, and at the auto block where they reach their best ratio they run near 1 GB/s—one to two orders of magnitude faster than our method at the matching ratio. The streaming coders' speed advantage is thus realized precisely at the large blocks our method does not require for ratio. At the small blocks where its ratio is best, our compression happens to outrun Kanzi's—but only because Kanzi's per-block setup cost makes *its* order-0 coders slow there; against the fastest table-driven entropy stages (an FSE/tANS implementation such as *zstd*'s), it would not, and our decoding stays slower at every block (Table 1). We therefore make no throughput claim: the method's standing rests on ratio and structure, not speed. Section 9 identifies practical contexts in which the block size is an external constraint, and the small- N ratio advantage becomes the operative comparison.

Codecs that model context reach smaller fractions only by exploiting structure our order-0 method does not, and are not order-0 baselines; for reference, at the auto block the order-1 ANS1 reaches 0.4828, FPAQ 0.5925 and CM 0.4956, and the context-mixing TPAQ and TPAQX 0.1767 and 0.1703—the last below 8 MB/s.

8.1. Heterogeneous Corpora

To test whether the small-block advantage on *enwik9* generalizes beyond text, we repeat the comparison on the Silesia corpus—twelve files of deliberately mixed type (English and Polish text, program source, an executable, a database, and medical images), 212 MB in total. Each file is compressed independently at matched block sizes against Kanzi's Huffman coder, the most compact streaming order-0 coder at small blocks (Table 1). Table 2 gives the per-file fractions at $N = 1024$.

The outcome is a clean split along alphabet width. On every text and source file—*dickens*, *reymont*, *webster*, *xml*, *samba*—and on the structured *nci*, our method is the more compact; it loses only on the dense, full-alphabet binaries *mr*, *ooffice*, *osdb* and *x-ray*. This is exactly what Proposition 1 predicts: the per-block housekeeping is $O(A \log N)$, negligible when the alphabet A is small but, on a near-256-symbol source, large enough to overtake the gain from exact arrangement coding. The prediction is quantitative—the deficit on the full-alphabet *x-ray* image falls from 0.048 at $N = 1024$ to 0.014 at $N = 8192$ as the fixed header amortizes over a larger block—and the aggregate gap to Huffman is

only +0.0004, +0.0028, +0.0022, +0.0004 at $N = 1024, 2048, 4096, 8192$: a tie to within 0.4% on a corpus chosen to be hostile, with our method winning the majority of files outright.

Two further points connect this to the rest of the section. First, the text files independently reproduce the enwik9 result—on narrow-alphabet data the method beats Huffman—so that advantage is a property of the alphabet, not of a single benchmark. Second, the per-file figures isolate the amortization effect noted above: compressed as one concatenated stream the corpus yields 0.6501 at $N = 1024$ against 0.6477 per file, the 0.0024 difference being the cost of the stream-global alphabet once the byte distribution shifts between files. The practical reading is that this codec is the most compact of the *block-resetting* order-0 coders for constrained-alphabet inputs at small blocks, and is competitive rather than preferable on wide-alphabet, heterogeneous data.

Table 2. Per-file compressed fraction on the Silesia corpus at $N = 1024$, each file compressed independently, against Kanzi’s Huffman coder (the most compact streaming order-0 coder at this block). Lower is better; the smaller of each pair is in bold. The method wins on every text and source file and loses only on the dense, full-alphabet binaries; the aggregate is a tie.

File (Type)	Size (MB)	This Work	Huffman
dickens (text)	10.2	0.6028	0.6106
reymont (text)	6.6	0.6350	0.6392
webster (text)	41.5	0.6595	0.6638
xml (markup)	5.3	0.6482	0.6558
samba (source)	21.6	0.6611	0.6703
nci (structured)	33.6	0.3277	0.3309
mozilla (executable)	51.2	0.7079	0.7116
sao (binary)	7.3	1.0342	1.0360
mr (medical)	10.0	0.5210	0.4890
ooffice (library)	6.2	0.8388	0.8263
osdb (database)	10.1	0.9142	0.9015
x-ray (image)	8.5	0.8857	0.8376
Total	211.9	0.6477	0.6473

These per-file wins are over Huffman, which both the integer-length penalty and per-block table resets handicap. Against the sterner globally modeled order-0 baseline introduced above (Figure 2), the picture refines along the same stationarity axis: on the *stationary* text files (dickens, reymont, webster) one global model is the better fit and edges our codec, whereas on the structured, non-stationary files—xml, the source tree samba, the executable mozilla—our per-block counts adapt to the shifting local statistics and our codec is the more compact. The codec’s ratio advantage over a strong order-0 baseline is thus specific to non-stationary data; on stationary text it is the structural package, not ratio, that sets it apart.

The classic Canterbury corpus (2.8 MB, eleven mostly textual files, each at $N = 1024$) reproduces the pattern—the method wins nine of eleven files and the aggregate (0.4803 against Huffman’s 0.4806), losing only the wide-alphabet binaries—and the larger Canterbury files extend it, the four-symbol E. coli genome showing the widest narrow-alphabet margin (0.256 against 0.279). The most telling single file is the bilevel fax image ptt5: narrow and highly skewed, it reaches 0.150 where Huffman manages 0.224. That gap has a different cause from the housekeeping differences seen elsewhere—Huffman must spend a whole bit on the dominant symbol whatever its probability, whereas the exact rank pays only its true information content—so on skewed sources the method’s edge over Huffman

is the avoidance of the integer-length-code penalty an arithmetic coder also escapes, here realized by combinatorial ranking.

8.2. Stream Versus Per-Record Mode

Section 4.1 distinguishes a *stream* mode, which amortises a reference alphabet across the blocks, from a *per-record* mode in which each block names its own alphabet. Compressing the same input both ways isolates exactly what that amortization is worth. On enwik9—one homogeneous text stream—stream mode is the more compact at every block size, and the margin grows as the block shrinks and the per-block alphabet cost is paid more often (Table 3): from 0.3% of the input at $N = 8192$ to 1.5% at $N = 512$. The stream column reproduces the “this work” fractions of Table 1, as it must, those experiments being stream mode.

Table 3. Stream versus per-record mode on enwik9: compressed fraction at each block size. Δ is per-record minus stream—the alphabet amortisation stream mode provides—and grows as the block shrinks, the per-block alphabet cost then being paid more often.

N	Stream	Per-Record	Δ
512	0.6835	0.6938	+0.0103
1024	0.6595	0.6676	+0.0082
2048	0.6461	0.6528	+0.0067
4096	0.6400	0.6442	+0.0042
8192	0.6376	0.6397	+0.0021

The Silesia files, compressed one at a time, show that the better mode tracks alphabet homogeneity rather than being universal. On the text and source files, the reference fits and stream mode wins, by up to 0.010 at $N = 512$ (reymont; similarly dickens, webster, xml, samba). On the dense, wide-alphabet binaries the per-block alphabet varies so much that encoding each block’s symmetric difference against a fixed reference costs *more* than naming its alphabet outright, and per-record mode is the more compact—mr by 0.015 (2.9%) and mozilla by 0.006 at $N = 512$, with ooffice, sao, and x-ray at best neutral. These are the same wide-alphabet files on which the method trails Huffman (Section 8.1): where the order-0 housekeeping is already proportionally large, a mismatched reference only adds to it. The rule is the one Section 4.1 states—stream mode for homogeneous record streams, per-record for independent or alphabet-varying units—now with the cost of choosing wrong made explicit.

8.3. A Record Store

Section 9 argues that the method’s home is the homogeneous record store, so we test one on real data: the timestamp column of the HADOOP log set from the loghub collection [33]—180,897 records, the ISO-8601 [34] field YYYY-MM-DD HH:MM:SS,mmm that opens each log line, 23 bytes each over a 14-symbol alphabet—compressed at record granularity with the block equal to the record. The defining property of a store is that records are written, addressed, encrypted, and read *individually*, so the only admissible coders are those that preserve per-record access; Table 4 compares them, with whole-stream gzip included only as a ceiling.

Table 4. A record store of 180,897 real timestamp records from the loghub HADOOP log set (23 bytes each, 14-symbol alphabet), compressed at record granularity (block = record). Only coders preserving per-record access are candidates; a static global-model order-0 coder is the most compact of them, but—unlike the method—offers no in-place encryption, count-indexed queryability, or exact predictable per-block size.

Coder	Per-Record Access	Fraction
Global-model order-0	yes	0.5544
This work, stream mode	yes	0.8048
This work, per-record mode	yes	1.0689
gzip-9, per record	yes	1.8679
gzip-9, whole stream	no	0.1082

Among the access-preserving coders, the static global-model coder is the most compact (0.554): coding each record against one stream-wide frequency table, it pays no per-record model cost. Our stream mode (0.805) trails it—it re-derives an alphabet symmetric difference and per-symbol counts for every record—while per-record mode (1.069, the full alphabet on each 23-byte record) and per-record gzip (1.868, a multi-byte header each) expand outright. On ratio alone, then, the codec is not the coder of choice here: a global model wins, and—to be fair to the strongest alternative—that global model can itself be encrypted record by record with AES-CTR, matching our full mode on size, per-record access, and length-only leakage. Two things that pairing still does not provide are an exact per-block size known *before* encoding (fixed by (N, k) in our representation, useful for fixed-rate channels and buffer budgeting) and the count-indexed *queryable* dial of Section 6, which lets a keyless party compute over the ciphertext at no size cost. Those two properties, not ratio and not encryption *per se*, are the reason to choose the method in this setting. Whole-stream gzip reaches 0.108, far below all of them, because the timestamps span a short window and recur heavily (only 64% are distinct) and gzip exploits that inter-record redundancy; but it forfeits per-record access entirely—it cannot decompress, re-encrypt, or address one record without processing the rest—so it is a ceiling, not a substitute. On record stores whose records are larger or whose alphabets are constrained—short-read genomics being the worked example of Section 8.5—the method is additionally competitive on ratio.

8.4. Effect of Alphabet Size

Because the method is order-0, its compressed fraction on an input drawn uniformly from an alphabet of A symbols should approach the order-0 entropy $\frac{1}{8} \log_2 A$. To verify this, we generated 64 MiB of uniform random bytes restricted to A distinct values for several A and compressed each at two block sizes, alongside Kanzi’s Huffman coder at its default auto block (Table 5).

The fraction tracks $\frac{1}{8} \log_2 A$ across the range, confirming order-0 optimality on controlled inputs; at $N = 4096$ it sits within 0.001–0.010 of Huffman, the residual being per-block housekeeping that shrinks as the block grows or the alphabet narrows. Two points carry to the rest of the paper: compression improves as the alphabet narrows—a four-symbol source compresses to the two-bit limit 0.25, while a full-byte uniform source is incompressible at order 0 and both coders expand slightly—and the housekeeping is proportionally largest exactly when the payload is smallest (small blocks, wide alphabets). The right-hand columns expose the throughput cost: the method runs 5–30× below Huffman, and its speed tracks the magnitude of the ranks rather than the alphabet alone—a narrow alphabet is quickest at a small block yet slowest at a large one, where each rank approaches the central binomial coefficient and big-integer arithmetic dominates.

Table 5. The method on uniform random bytes over an A -symbol alphabet, at two block sizes, alongside Kanzi’s Huffman coder at its auto block (64 MiB for the compressed fraction, 256 MiB for throughput, the latter excluding our one-time cache build). The $N = 1024$ and $N = 4096$ columns are this work. The compressed fraction tracks the order-0 entropy $\frac{1}{8} \log_2 A$ —within 0.001–0.010 of Huffman at $N = 4096$, the gap largest at $A = 256$ where per-block housekeeping is proportionally most significant—while throughput runs 5–30× below Huffman, the more so as the block grows.

A	$\frac{1}{8} \log_2 A$	Compressed Fraction			Throughput (MB/s)		
		$N = 1024$	$N = 4096$	Huff.	$N = 1024$	$N = 4096$	Huff.
2	0.1250	0.1315	0.1267	0.1256	244	78	1695
4	0.2500	0.2573	0.2519	0.2506	179	48	1497
16	0.5000	0.5107	0.5032	0.5010	185	57	924
64	0.7500	0.7675	0.7564	0.7518	173	105	1580
256	1.0000	1.0537	1.0131	1.0028	135	132	1225

8.5. Real Constrained-Alphabet Data

The synthetic sweep above draws bytes uniformly over an A -symbol alphabet; real constrained-alphabet data lets us confirm the same behavior outside the laboratory and separates two regimes the method treats very differently. We extract the nucleotide stream—the bases only, with quality scores and headers excluded—from short-read sequencing data and compress it as an $\{A, C, G, T\}$ byte stream at the read granularity.

On *Escherichia coli* K-12 (accession SRR2627175, Illumina MiSeq; Illumina, Inc., San Diego, CA, USA), the base composition is almost perfectly uniform: the four bases occur with frequencies 0.244–0.256 and the empirical order-0 entropy is 1.9999 bits/base, an ideal fraction of 0.2500. At $N = 1024$ the method compresses the 540 MB base stream to 0.2569—reproducing, on real data, the synthetic $A = 4$ figure of 0.2573 (Table 5) to three decimals, a direct confirmation that the alphabet model of Proposition 1 describes real sequences. But a uniform four-symbol source is exactly the case in which the order-0 floor coincides with trivial two-bit packing: the method *matches* the floor and, at small blocks, pays a slight housekeeping premium over packing (0.2569 against 0.2500). On near-uniform genomes it therefore has no ratio advantage over bit-packing—only exactness and the structural properties of Section 9. (The assembled genome of the same organism appears as a static file in the Canterbury corpus of Section 8.1, where its identically near-uniform composition again leaves the method just above the two-bit floor.)

The advantage appears when the base composition is skewed. The genome of *Plasmodium falciparum* is among the most AT-rich known ($\approx 80\%$ A + T genome-wide), and even after the GC-enrichment of short-read sequencing its reads remain markedly non-uniform. On an Illumina run of this organism (accession ERR006177) the base stream has frequencies $A = 0.327$, $T = 0.359$, $C = 0.162$, $G = 0.153$ and an order-0 entropy of 1.897 bits/base—an ideal fraction of 0.237, well below the two-bit ceiling. At $N = 1024$ the method compresses the 279 MB base stream to 0.2436, against 0.25 for a two-bit pack: here exact arrangement coding is strictly the more compact, because the single-symbol skew it captures is precisely what fixed-width packing cannot. The residual 0.007 over the 0.237 floor is the same per-block housekeeping seen throughout, and shrinks as the block grows. The comparison drawn here is against fixed-width packing and gzip, not against the global-model order-0 baseline of Figure 2: a single base-frequency model shared over the stream would capture the same single-symbol skew and reach a comparable fraction, so on this near-stationary source the method’s case is again the structural package (exact read sizing, in-place encryption) rather than a ratio win over a global model.

Two caveats temper this. First, the gains live in the base composition alone: the order-0 model captures single-symbol skew, not the repeats and context that dominate genomic redundancy. On this shuffled stream of short reads, a small-window LZ coder finds little such structure—gzip-9 reaches only 0.252, which the exact ranking in fact edges—but a strong general compressor with a larger window does find it: zstd-19 reaches 0.132 on the *E. coli* stream and 0.170 on *P. falciparum*, well below both the exact coder and the two-bit floor, and reference-based, format-aware genomic compressors such as CRAM and Genozip exploit alignment and context to compress sequencing data far further still. These remain the right choice when ratio is the goal. Second, quality scores and headers, the bulk of a FASTQ file, are a different distribution we do not address. The genomics result is thus a validation of the constrained-alphabet behavior and an illustration of where exact order-0 coding beats both bit-packing and an order-0-grade general compressor, not a claim to competitive sequence compression.

These results position the present method as a simple, exactly invertible order-0 coder that provably meets the order-0 entropy bound, rather than as a competitor to context-modeling entropy coders. Its distinguishing properties lie elsewhere: the encoding is a pure combinatorial rank, every block is independent—enabling the straightforward parallelism of Section 7—and the same machinery lends itself to encryption.

8.6. Encryption Overhead

The size-preserving cipher of Section 6 adds no bytes: in every setting the encrypted block is bit-for-bit the same size as the plaintext block, a property our round-trip tests verify exactly. Its only cost is throughput, which Table 6 measures on a representative mildly skewed block over a 40-symbol alphabet (Apple M3), comparing the plaintext codec against the two cipher settings: *queryable* (only the alphabet rank and the position ranks encrypted, the counts left in clear) and *full* (every field encrypted).

Table 6. Throughput cost of the size-preserving cipher on a representative block (Apple M3), plaintext versus the queryable and full settings. The compressed size is identical in all three—the cipher adds exactly 0 bytes. Throughput is in MB/s; the parenthesized factor is the slowdown relative to plaintext. The plaintext figures are not comparable with Table 1: this is a different, deliberately small representative block (a 40-symbol alphabet at the sizes shown), chosen to isolate the cipher’s relative cost rather than absolute codec speed. The keystream is a reference SHA-256 construction, so these figures are an upper bound on the overhead.

<i>N</i>	Operation	Plaintext	Queryable	Full
1024	compress	68.7	38.3 (1.8×)	31.7 (2.2×)
1024	decompress	11.1	9.7 (1.1×)	9.3 (1.2×)
4096	compress	50.8	40.6 (1.3×)	38.6 (1.3×)
4096	decompress	10.0	9.4 (1.1×)	9.2 (1.1×)

Three things stand out, all consistent with the construction. First and most important, encryption is *exactly* size-preserving: every field is replaced by a ciphertext over the same modulus, so the block length is unchanged and the only information an encrypted store leaks is that length (Section 6.2). At the whole-file level, the encrypted format adds a single fixed-size header—a one-byte mode flag and a 16-byte random salt—independent of the input size and of the block count, so the per-block size cost remains exactly zero. Second, the overhead falls almost entirely on *compression at small blocks*, where the plaintext codec is fastest, and the per-field keystream is therefore proportionally most expensive: up to 2.2× for full mode at $N = 1024$. Decompression, already dominated by the downward search of the unranking step—which the cipher never touches—absorbs the keystream cost

almost entirely, slowing by at most $1.2\times$. Third, full mode costs only modestly more than queryable: the bulk of the work is encrypting the variable-width ranks, which both settings share, while full mode adds one short keystream per count. As the block grows, the relative overhead shrinks, the per-field cost amortizing over more bytes.

Two qualifications apply. The keystream is a reference SHA-256 counter construction, chosen for clarity rather than speed, generated and reduced per field behind a fresh big-integer allocation each time. Measured directly, that reference generator runs at a flat ≈ 0.7 GB/s irrespective of output size, whereas a hardware-accelerated AES stream reaches an order of magnitude more—but only once the output is large enough to amortize the cipher’s per-call key schedule. At the small per-field sizes this implementation requests, that setup cost dominates and the cipher family barely matters; the order-of-magnitude gain is unlocked only by generating one keystream per block and slicing the fields from it. The table is therefore a deliberate *upper bound*, not the operative cost: the headline $2.2\times$ reflects the reference SHA-256 per-field generator, and a per-block keystream paired with a vetted cipher (AES where hardware support exists, ChaCha20 as the portable alternative) would cut it substantially. The reader should read these figures as the worst case the construction admits rather than its intended cost. The figures are also for the cipher in isolation: a production encrypted store would add per-block authentication (Section 6), whose cost is not included here.

It is worth being precise about what the construction does and does not buy over a conventional cipher. The obvious alternative is to compress, then encrypt with a stream cipher such as AES-CTR, which is itself size-preserving. Two variants matter (Table 7). Encrypting the *whole* compressed stream makes every record opaque *and* unaddressable—the record boundaries vanish into the ciphertext—so it is no use to a store. Encrypting *each record* independently, however (nonce = record index, the length framing left in clear), is size-preserving, per-record addressable, and leaks only the per-record length—matching our *full*-encryption mode on every axis, and doing so more simply and faster, since it avoids the per-field big-integer keystream whose cost Table 6 reports. Full mode is therefore best understood as a *structurally integrated* alternative to per-record AES-CTR—encryption folded into the coding pass, matching it on size, access, and leakage—not a more capable one; the queryable mode below is where the construction does something no whole-record cipher can.

What no whole-record cipher can do is our *queryable* mode. Leaving the per-record counts in clear turns their cleartext field widths into a skip-table, so a keyless storage server can read each record’s count spectrum and field offsets and act on them—repack and defragment the store, tier records by their (leaked) order-0 size, or pre-filter before an authenticated equality check—without decrypting a byte or learning a symbol label. That bounded, keyless computability at zero size cost is the construction’s distinctive offering, and it is exactly the deliberate trade of Section 6.2: justified only where such a query is wanted and the count-spectrum disclosure (Naveed–Kamara–Wright) is acceptable, not as a default.

Table 7. The joint cipher versus stream-cipher alternatives. All four are exactly size-preserving ($1.00\times$); they differ only in what a keyless party can compute. Per-record AES-CTR matches full mode on every axis—size, per-record access, and length-only leakage—and is simpler and faster; only the queryable dial exposes the count spectrum and field offsets to a keyless party, which no whole-record cipher provides. Whole-stream AES-CTR additionally forfeits per-record addressability.

Encrypted Form	Computable Without the Key
Joint cipher, full mode	per-block length only
Per-record AES-CTR	per-block length only
Joint cipher, queryable mode	count spectrum + field offsets
Whole-stream AES-CTR	nothing (records not addressable)

8.7. Resource Use and Parallel Scaling

Two costs distinguish the method from the streaming baselines beyond throughput: memory and how it scales with threads.

Peak memory. The codec’s footprint is dominated by the binomial-coefficient cache (Section 7), which holds $\approx N^2/4$ coefficients of growing width for a footprint of $\approx 0.0154\,N^3$ bytes (Supplementary Materials S3). Measured peak resident memory (Table 8) rises from 66 MB at $N = 512$ to of order 10 GB at $N = 8192$, while the entropy-coder baselines, which keep only a block-sized buffer, stay flat at 14–16 MB regardless of N . These runs compress `enwik8`; because the input is streamed rather than held resident—the 66 MB peak at $N = 512$ is already below the 100 MB corpus—peak memory is fixed by the block size rather than the data, and would be identical on the `enwik9` used for the ratio experiments. Memory, like the $O(A \log N)$ header, is a cost the representation pays for its structural properties, and it is what sets the practical block ceiling; the $N = 8192$ resident peak varies with the host’s memory pressure (macOS compressing it from ~ 10 to ~ 5 GB under load). As a first-order estimate for constrained targets, the codec’s compute cost is 20–115 cycles/byte compressing and 107–450 cycles/byte decompressing (the decode penalty being the unranking search), and at the small blocks such a store uses ($N \leq 1024$) the cache is tens of MB—feasible on a modest device, with on-target measurement left as future work.

Table 8. Peak resident memory (Apple M3): the codec at several block sizes versus the order-0 Kanzi baselines. The codec’s cache-bound footprint grows with N ; the streaming coders are flat.

Method	N	Peak RSS (MB)
this work	512	66
this work	1024	131
this work	2048	370
this work	4096	1709
this work	8192	$\sim 10,240$
Kanzi Huffman	1024/4096	15.1/14.6
Kanzi ANS0	1024/4096	16.1/15.1
Kanzi Range	1024/4096	14.1/13.5

Parallel scaling. Because the batch size scales inversely with block length (Section 7), the per-block hand-off is amortized precisely where it would otherwise dominate: at $N = 512$ compression scales near-linearly—efficiency $\approx 100\%$ through four workers and $5.65\times$ at eight—whereas at $N = 8192$ it reaches only $3.38\times$ at eight workers (42% efficiency), becoming memory-bandwidth-bound on the wide-integer accumulations that draw on the large cache. The falloff at eight workers also reflects the test machine’s heterogeneous cores. Thread management is thus not the small-block bottleneck; throughput is limited by big-integer compute and memory bandwidth.

9. Applications

The codec’s throughput is lower than streaming entropy coders at the same ratio, so its value lies not in general-purpose compression but in a setting the combinatorial representation suits unusually well: *homogeneous record stores*—collections of small, like-formatted records (database columns, log and packet stores, sequencing reads) compressed and accessed one record at a time. Such records share an alphabet, so the stream-mode reference of Section 4.1 is fair; the block is the record, so the small-block regime is the operative one rather than a contrivance; and, because each block is encoded as integer fields over known finite ranges, the representation delivers as one package what a streaming coder cannot: order-0-compact ratios, independent and parallel coding, random access

to any record, exact predictable size, and the size-preserving cipher with count-indexed queryability of Section 6. The subsections below develop this use case and the structural properties it draws on.

9.1. Compressed Record Stores

Here the small-block regime is not a contrivance but the only admissible one—the block *is* the externally fixed record—and it is exactly where this method is the most compact of the *block-resetting* order-0 coders on narrow-alphabet inputs (Section 8). Section 8.3 measures this on 180 897 real log timestamps, where a static global model wins on ratio but only the method supplies the encryption, queryability, and exact sizing a store needs.

Short-read sequencing is a second setting the experiments reach directly. Platforms produce reads of roughly 50–300 bases, each a natural compression unit—fixed-length, small, and over a four-symbol alphabet for which the method provably reaches the order-0 floor in one pass with no model state. On real reads (Section 8.5) it tracks that floor and, on the AT-rich *P. falciparum*, beats both fixed-width packing and gzip on the base stream; dedicated formats such as CRAM still win overall through higher-order modeling, but for an embedded sequencer or streaming edge device a simple, exactly invertible coder at the read granularity may be preferable to a complex pipeline.

Other naturally small-record settings fit the same profile, though we report them as motivated rather than measured: embedded and latency-bounded pipelines, where a large amortizing block simply will not fit; and packet or log stores—DNS responses, CAN frames, fixed-width database rows—where the record is the only available granularity and the per-block housekeeping is proportionally smallest relative to the payload.

One limitation applies across all these settings: the decoder is markedly slower than the encoder—roughly four to five times so (Table 1), owing to the downward search in the unranking step—so read-heavy deployments such as record stores and sequencing archives, written once and read many times, must weigh that asymmetry against the ratio gain; accelerating the unranking search (Section 5) is the natural remedy.

9.2. Further Structural Uses

Beyond storage, the exactness has uses of its own. A block's compressed size is fixed by (N, k) and known before encoding—convenient for fixed-rate channels and buffer budgeting—and the coding is exact integer arithmetic with no renormalization, a clean reference against which to validate faster, approximate coders; the underlying $O(k)$ rank/unrank bijection is independently useful for compact set identifiers and sampling. And because encoding an occurrence bitset as its colexicographic rank is exactly the construction underlying succinct rank/select bitvectors, the codec doubles as a building block for compressed-but-queryable structures—bitmaps, inverted-index posting lists, and similar set- or permutation-structured data, for which an order-0 model is already near-optimal.

10. Discussion

The experiments and the construction together place the method precisely, and it is worth being explicit about where that is, and is not.

Proposition 1 and the experiments agree that the codec attains the order-0 entropy of its input: its arrangement coding is exact, free of the integer-length penalty that costs Huffman on skewed sources—the ptt5 case of Section 8.1—and of the renormalisation state a streaming coder carries. What it cannot do is model structure above order 0. On data whose redundancy is largely higher-order—natural text beyond its symbol frequencies, genomic reads with repeats—context-modeling and dictionary coders (the CM/TPAQ

family, CRAM, Genozip, or simply gzip exploiting matches) compress substantially further and remain the right tools when ratio is the only concern. The method's compression claim concerns the order-0 frontier, not the absolute one.

Within that regime, the codec is the most compact of the *block-resetting* order-0 coders on narrow-alphabet inputs at matched block sizes, and ptt5 (Section 8.1) shows that the edge can be large where a skewed distribution defeats Huffman's integer-length codes. Three bounds, established with the data in Section 8 and not re-argued here, keep that edge in proportion: on a heterogeneous corpus it is a fraction of a percent and reverses on wide-alphabet binaries as the $O(A \log N)$ housekeeping predicts; part of it is stream mode's alphabet amortization rather than the coding kernel; and against a coder granted one global model, it reverses outright below the crossover (Figure 2). Ratio alone is therefore not the reason to choose the method. The reason is structural: because each block is a tuple of finite-range integers rather than a self-delimiting bitstream, one representation packages independent, randomly-accessible blocks—which block-resetting coders also allow—with two things a self-delimiting bitstream does not give: an exact size known before encoding, and the size-preserving, queryable cipher of Section 6. That package, paired with order-0 compactness, is why the method's home is the homogeneous record store, shown here on real production-log timestamps and genomic reads.

The price is throughput. Big-integer arithmetic makes the codec slower than table-driven coders at the same ratio, and decoding—which additionally searches during unranking—is several times slower than encoding, a real consideration for read-heavy stores. The encryption, finally, is now given a formal treatment (Supplementary Materials S1): its confidentiality is proven relative to a defined leakage function by reduction to standard keystream security, with the residual leaks (the per-field modulus and the per-block length) stated explicitly. What remains is engineering rather than analysis—any deployment still needs a vetted stream cipher, disciplined nonces, and independent review—so a hardened realization is the most valuable next step.

11. Conclusions and Future Research

We have realized and characterized the representation that order-0 enumerative coding induces as a lossless byte-stream codec, and located it precisely. On *enwik9* its compressed size converges onto the order-0 rate of the Huffman, range, and ANS coders as the block grows, and at a matched block size its stream mode is the most compact of these *block-resetting* coders on narrow alphabets; on the heterogeneous Silesia corpus that edge holds on every text and source file but reverses on full-alphabet binaries as the $O(A \log N)$ housekeeping overtakes it, and against a coder granted one global model, it reverses below the crossover. Ratio alone is therefore not the case for the method.

Its case is structural, and its cost is throughput. Because each block is a tuple of finite-range integers rather than a self-delimiting bitstream, one representation packages independent random access, an exact size known before encoding, and a size-preserving, queryable cipher—none of which a streaming order-0 coder supplies—while running $5\text{--}30\times$ slower (35–199 MB/s compressing, 9–38 MB/s decompressing) and needing a coefficient cache that grows with the block size (Section 7). Its home is accordingly the homogeneous record store and the constrained-alphabet stream—embedded pipelines, packet and record stores, short-read genomics—where the block size is externally fixed and the structural package, not ratio or speed, is the operative advantage.

Several directions remain. The dominant decompression cost is the downward search in the unranking step, whose acceleration is worth pursuing [25]; the hockey-stick identity for contiguous runs reduces big-integer additions and may repay its detection cost on large blocks. A native or GMP-backed port is unlikely to help substantially, since the

bottleneck is large-integer addition, which performs comparably under Go’s math/big and GMP. More promising are the structural avenues: a queryable, succinct variant that supports rank and select on the compressed form without full decompression, and a hardened, production realization of the joint compression–encryption scheme of Section 6—with a vetted stream cipher and a full security analysis under a stated threat model. A dedicated hardware implementation is a further natural direction, which we frame as future work rather than a validated result: each rank is a multi-operand wide-integer sum that maps onto a carry-save adder tree, the datapath needs only addition, comparison, and table lookup, and the independence of blocks gives spatial parallelism. We report no synthesis figures—the wide, data-dependent integer widths make resource use (LUT counts, maximum operating frequency, and on-chip memory for the coefficient tables) a study in its own right—so a field-programmable gate array (FPGA) prototype and its resource characterization remain open.

Supplementary Materials: The following supporting information can be downloaded at: <https://www.mdpi.com/article/10.3390/a19080634/s1>, Supplementary Materials S1 (formal security analysis—the leakage-relative IND-CPA theorem and its proof, and the count-spectrum leakage quantification); Supplementary Materials S2 (the authenticated-encryption construction—the size-preservation/integrity impossibility result and an optional, costed authenticated mode); and Supplementary Materials S3 (analytical models for the per-block header overhead and the coefficient-cache memory).

Funding: This research received no external funding; the work was wholly self-funded.

Data Availability Statement: The corpora used in this study are publicly available: *enwik9*, the Silesia and Canterbury corpora, the loghub HADOOP log dataset [33], and the sequencing runs SRR2627175 (*E. coli*) and ERR006177 (*P. falciparum*) from the European Nucleotide Archive (ENA). The reference codec, the experiment scripts that reproduce every table, and the sources that generate this manuscript are openly available at <https://github.com/ajr74/combinadic-compression> and archived at <https://doi.org/10.5281/zenodo.20684743> (Zenodo, accessed on 30 July 2026).

Acknowledgments: During the preparation of this manuscript the author used Claude Opus 4.8 (Anthropic) to assist with drafting and editing the text and with surveying related work; its use in the computational experiments is described in Section 8. The core method—the combinadic compression algorithm and its codec—was conceived and developed independently and solely by the author, predating this assistance. The author reviewed and verified all AI-assisted output and takes full responsibility for the content of this publication.

Conflicts of Interest: The author is employed by Shack C Technologies. The author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AES	Advanced Encryption Standard
ANS	Asymmetric numeral systems
CTR	Counter mode (block-cipher mode of operation)
ENA	European Nucleotide Archive
FPE	Format-preserving encryption
FPGA	Field-programmable gate array
GMP	GNU Multiple Precision Arithmetic Library
IND-CPA	Indistinguishability under chosen-plaintext attack
PRG	Pseudorandom generator

SHA Secure Hash Algorithm

References

1. Ziv, J.; Lempel, A. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. Inf. Theory* **1977**, *23*, 337–343. <https://doi.org/10.1109/TIT.1977.1055714>.
2. Ziv, J.; Lempel, A. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Trans. Inf. Theory* **1978**, *24*, 530–536. <https://doi.org/10.1109/TIT.1978.1055934>.
3. Burrows, M.; Wheeler, D.J. *A Block-Sorting Lossless Data Compression Algorithm*; Technical Report 124; Digital Equipment Corporation Systems Research Center: Palo Alto, CA, USA, 1994.
4. Cleary, J.G.; Witten, I.H. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Trans. Commun.* **1984**, *32*, 396–402. <https://doi.org/10.1109/TCOM.1984.1096090>.
5. Huffman, D.A. A Method for the Construction of Minimum-Redundancy Codes. *Proc. IRE* **1952**, *40*, 1098–1101. <https://doi.org/10.1109/JRPROC.1952.273898>.
6. Witten, I.H.; Neal, R.M.; Cleary, J.G. Arithmetic coding for data compression. *Commun. ACM* **1987**, *30*, 520–540. <https://doi.org/10.1145/214762.214771>.
7. Martin, G.N.N. Range Encoding: An Algorithm for Removing Redundancy from a Digitised Message. In Proceedings of the Video and Data Recording Conference, Southampton, UK, 24–27 July 1979.
8. Duda, J. Asymmetric numeral systems: Entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. *arXiv* **2013**, arXiv:1311.2540. <https://doi.org/10.48550/arXiv.1311.2540>.
9. Cover, T.M. Enumerative source encoding. *IEEE Trans. Inf. Theory* **1973**, *19*, 73–77. <https://doi.org/10.1109/TIT.1973.1054929>.
10. Lynch, T.J. Sequence time coding for data compression. *Proc. IEEE* **1966**, *54*, 1490–1491. <https://doi.org/10.1109/PROC.1966.5167>.
11. Schalkwijk, J.P.M. An algorithm for source coding. *IEEE Trans. Inf. Theory* **1972**, *18*, 395–399. <https://doi.org/10.1109/TIT.1972.1054832>.
12. Rissanen, J.J. Generalized Kraft inequality and arithmetic coding. *IBM J. Res. Dev.* **1976**, *20*, 198–203. <https://doi.org/10.1147/rd.203.0198>.
13. Borysenko, O.; Matsenko, S.; Bobrovs, V. Binomial Number System. *Appl. Sci.* **2021**, *11*, 11110. <https://doi.org/10.3390/app112311110>.
14. Borysenko, O.; Matsenko, S.; Salgals, T.; Spolitis, S.; Bobrovs, V. The Lossless Adaptive Binomial Data Compression Method. *Appl. Sci.* **2022**, *12*, 9676. <https://doi.org/10.3390/app12199676>.
15. Kulyk, I.; Shevchenko, M.; Grynenko, V.; Hermes, M. Speeding up Binomial Compression Based on Binary Binomial Numbers. *East.-Eur. J. Enterp. Technol.* **2025**, *4*, 26–33. <https://doi.org/10.15587/1729-4061.2025.335729>.
16. Raman, R.; Raman, V.; Rao, S.S. Succinct indexable dictionaries with applications to encoding k -ary trees and multisets. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2002; pp. 233–242.
17. Grangetto, M.; Magli, E.; Olmo, G. Multimedia selective encryption by means of randomized arithmetic coding. *IEEE Trans. Multimed.* **2006**, *8*, 905–917. <https://doi.org/10.1109/TMM.2006.879919>.
18. Bellare, M.; Rogaway, P.; Spies, T. *The FFX Mode of Operation for Format-Preserving Encryption*; Submission to NIST; NIST: Gaithersburg, MD, USA, 2010.
19. Dworkin, M.J. *Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption*; Technical Report NIST Special Publication 800-38G; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2016. <https://doi.org/10.6028/NIST.SP.800-38G>.
20. Black, J.; Rogaway, P. Ciphers with Arbitrary Finite Domains. In *Proceedings of the Topics in Cryptology—CT-RSA 2002*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2002; Volume 2271, pp. 114–130. https://doi.org/10.1007/3-540-45760-7_9.
21. Durak, F.B.; Vaudenay, S. Breaking the FF3 Format-Preserving Encryption Standard over Small Domains. In *Proceedings of the Advances in Cryptology—CRYPTO 2017*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2017; Volume 10402, pp. 679–707. https://doi.org/10.1007/978-3-319-63715-0_23.
22. Beyne, T. Linear Cryptanalysis of FF3-1 and FEA. In *Proceedings of the Advances in Cryptology—CRYPTO 2021*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2021; Volume 12825, pp. 41–69. https://doi.org/10.1007/978-3-030-84242-0_3.
23. Knuth, D.E. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*; Section 7.2.1.3, the combinatorial number system; Addison-Wesley: Upper Saddle River, NJ, USA, 2011.
24. Csiszár, I. The Method of Types. *IEEE Trans. Inf. Theory* **1998**, *44*, 2505–2523. <https://doi.org/10.1109/18.720546>.
25. Kruchinin, V.; Shablya, Y.; Kruchinin, D.; Rulevskiy, V. Unranking Small Combinations of a Large Set in Co-Lexicographic Order. *Algorithms* **2022**, *15*, 36. <https://doi.org/10.3390/a15020036>.

26. Collet, Y. xxHash: Extremely Fast Non-Cryptographic Hash Algorithm. Available online: <https://github.com/Cyan4973/xxHash> (accessed on 7 June 2026).
27. Popa, R.A.; Redfield, C.M.S.; Zeldovich, N.; Balakrishnan, H. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*; ACM: New York, NY, USA, 2011; pp. 85–100. <https://doi.org/10.1145/2043556.2043566>.
28. Naveed, M.; Kamara, S.; Wright, C.V. Inference Attacks on Property-Preserving Encrypted Databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*; Association for Computing Machinery: New York, NY, USA, 2015; pp. 644–655. <https://doi.org/10.1145/2810103.2813651>.
29. Katz, J.; Lindell, Y. *Introduction to Modern Cryptography*, 3rd ed.; Chapman and Hall/CRC: Boca Raton, FL, USA, 2020.
30. Granlund, T.; the GMP development team. GNU MP: The GNU Multiple Precision Arithmetic Library (6.3.0). Available online: <https://gmplib.org> (accessed on 7 June 2026).
31. Graham, R.L.; Knuth, D.E.; Patashnik, O. *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed.; Addison-Wesley: Reading, MA, USA, 1994.
32. Langlet, F. Kanzi: Lossless Data Compression. Available online: <https://github.com/flanglet/kanzi-go> (accessed on 7 June 2026).
33. Zhu, J.; He, S.; He, P.; Liu, J.; Lyu, M.R. Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*; IEEE: New York, NY, USA, 2023. <https://doi.org/10.48550/arXiv.2008.06448>.
34. ISO 8601-1:2019; Date and Time—Representations for Information Interchange—Part 1: Basic Rules. International Organization for Standardization: Geneva, Switzerland, 2019.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.