Bit-Level Columnar Coding of JSON Trees with Append-Only Delta Chains

Tomoya Nagasawa

November 2026

ABSTRACT

Self-describing binary JSON formats such as MessagePack and CBOR work at byte granularity, and they interleave structure with values in document order. We specify a JSON encoding that drops byte alignment in favour of variable-width bit fields. One pre-order scan splits a document into twelve column streams: structure links, key metadata, key strings, value types, booleans, numbers and strings. Each column is coded with small width-selecting integer codes, a ±3 delta window and run-length packing. The only count the stream writes is the number of leaves, because every other column length follows from earlier columns. We prove that the integer codes are prefix-free and give their lengths in closed form. We also prove that the number of structure slots equals the largest value link, and that the slot and leaf sequences determine the tree, so decoding needs no field markers. The same grammar serves as an update format: an edit is a structured payload addressed against the materialized state, and a length-framed chain of such payloads replays to every version. We computed sizes from the specification on seeded synthetic data and compared them with the shortest MessagePack and CBOR encodings. Arrays of records take 52–55% of MessagePack's size and 36–39% of JSON's. A flat array of two-decimal floats takes 34% of MessagePack's size, and 1000 random documents take 13.3% fewer bytes in total than in MessagePack. An interleaved *{t, v}* time series is 21% larger than MessagePack, because a single shared number chain sees no small differences in it and most of its key links fall outside the delta window.

Keywords: JSON, binary serialization, prefix codes, bit packing, columnar encoding, delta encoding, version history

1 Introduction

JSON [8] is a common interchange format on the web, and some JSON is stored *permanently*. Examples are content-addressed archives, append-only ledgers and permanent storage networks, where every byte is paid for once and kept forever. Two properties matter in that setting that matter less in RPC: how small a single encoded document is, and how cheaply a document's *history* can be stored as it changes.

The common binary JSON formats (MessagePack [1], CBOR [2]) are built for streaming, byte-aligned decoding. Every value is a tagged byte sequence emitted in document order. This design is simple, but it spends at least one whole byte on every value, flag or length, even one that needs only 2 bits. It also cannot exploit the repetition that structured data has *across* records (the same keys, the same types, slowly increasing ids) without a separate general-purpose compressor.

We study the opposite trade-off. Our contributions are:

- **Width-selecting integer codes**, in which a 2-bit selector picks a short payload or falls back to LEB128. We prove that they are prefix-free and give their lengths in closed form (§3).
- **Columnar restructuring of a single document** into twelve streams in a fixed order. We prove that the slot count equals the largest value link and that slots and leaves can be interleaved back into pre-order, so every column length except the leaf count is implied (§4).

- **An append-only delta chain** whose updates are payloads of the same grammar, addressed against the materialized document, with framing and prefix-replay properties (§5).
- **A size study** computed from the specification on seeded generators, against the shortest encodings that MessagePack and CBOR permit. It reports where the encoding is smaller, where it is larger, and why, down to individual columns (§6).

2 Background and related work

Byte-oriented binary JSON. MessagePack encodes small integers (−32…127) in a single byte, and strings of up to 31 bytes behind a 1-byte header. Larger values get a 1-byte tag and a fixed 1-, 2-, 4- or 8-byte payload, and floats are IEEE-754 binary32 or binary64. CBOR [2] has a similar major-type/argument scheme and also admits binary16 floats. Both are *row-oriented*: a record's keys are written again for every record.

Variable-length integers. LEB128 [3] stores 7 payload bits per byte plus a continuation bit, so it never uses fewer than 8 bits. Bit-level universal codes such as the γ code [5] and Golomb-type codes [4] avoid that floor. The encoding here uses a simple selector-plus-payload code that suits the small values found in structural metadata (Theorem 1 gives its exact cost).

Columnar layouts. Columnar layouts for nested records [6] store each field path as its own column, with repetition and definition levels describing the structure. Grouping similar values makes neighbouring values alike, which makes delta and run-length coding effective. The encoding applies the same idea *inside a single document*, but it groups values by *kind* (link, key, type, boolean, number, string) rather than by field path. §6.5 examines what this choice costs.

Patches and histories. JSON Patch [7] describes edits as text operations. Here an edit is a payload of the document grammar itself, addressed by position in the document's own structure tables.

3 Integer codes

All fields are written MSB-first into one continuous bitstream. Only the end of a payload is byte-aligned, by zero padding. Write $b(n)$ for the bit length of n, with $b(0) = 1$. LEB128 writes $\lceil b(n)/7 \rceil$ groups of 8 bits. Each group carries 7 value bits, least significant group first, and a top bit that is 1 exactly when more groups follow. Here the groups are written unaligned. Three selector codes cover nearly every field (Table 1). Each starts with a 2-bit selector, and the selector 11 escapes to LEB128.

code	00	01	10	11
short (counts, lengths)	2-bit	3-bit	4-bit	LEB128
uint (single-value mode)	3-bit	4-bit	6-bit	LEB128
num (number column)	Δ code, 3-bit	4-bit	6-bit	LEB128

Table 1. Width-selecting integer codes. The encoder always picks the narrowest payload that holds the value: a *short* uses 2 bits below 4, 3 bits below 8 and 4 bits below 16.

Theorem 1 (lengths). *Let $L_{LEB}(n) = 8\lceil b(n)/7 \rceil$. Then $L_{short}(n)$ is 4, 5 or 6 for n below 4, 8 or 16 respectively, and 2 + $L_{LEB}(n)$ for $n \geq 16$. $L_{uint}(n)$ is 5, 6 or 8 for n below 8, 16 or 64, and 2 + $L_{LEB}(n)$ otherwise. In the number column a delta code costs 5 bits, and a literal costs 6 or 8 bits below 16 or 64 and 2 + $L_{LEB}(n)$ otherwise.*

Proof. Each codeword is a 2-bit selector followed by the payload that Table 1 names, and LEB128 writes $\lceil b(n)/7 \rceil$ groups of 8 bits. ∎

Consequently $L_{short}(n) - L_{LEB}(n)$ is −4, −3 and −2 bits for n below 4, 8 and 16, and +2 bits at every $n \geq 16$ (Figure 1). The selector code is therefore shorter exactly when the values are below 16. In the worked examples of §4.6 and §5, every length, link delta and run count is below 16.

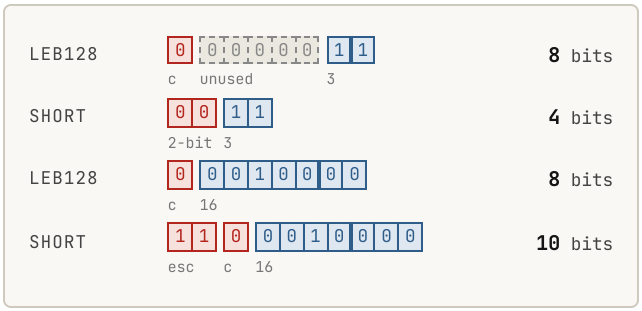


Figure 1. The integers 3 and 16 as LEB128 and as the *short* code. Below 16 the selector saves 2–4 bits. From 16 up it costs 2 extra bits at every magnitude (Theorem 1).

Theorem 2 (prefix-freeness). *The codeword sets of LEB128, short, uint and the number-column code are each prefix-free. Hence a concatenation of codewords, read left to right with the code known at each position, has exactly one parse.*

Proof. In a LEB128 codeword every group except the last has top bit 1, and the last has top bit 0. A codeword that was a proper prefix of another would have to end at a group boundary of the longer one. Its final group would then have top bit 0 while sitting at a non-final position of the longer codeword, where the top bit is 1. For the selector codes, two codewords with different selectors differ in their first two bits. Two codewords with the same selector carry payloads of the same fixed width, or two LEB128 codewords, which are prefix-free by the first part. In the number column, the codewords that begin `00 111` are run headers (below), and they are distinct from the delta codewords `00`·c with $c \le 6$. ∎

Deltas. Links and numbers are delta-coded against the previous value in their column. For a difference $d = v - p$, the *window* is $w(d) = d$ for $0 \le d \le 3$ and $w(d) = |d| + 3$ for $-3 \le d < 0$. A difference outside $[-3, 3]$ is written as a literal.

Theorem 3 (window). *w is a bijection from $\{-3,\dots,3\}$ onto $\{0,\dots,6\}$, with inverse $c \mapsto c$ for $c \le 3$ and $c \mapsto 3 - c$ for $c > 3$. The 3-bit value 7 is never a delta code.*

Proof. w maps $\{0,\dots,3\}$ onto itself and $\{-1,-2,-3\}$ onto $\{4,5,6\}$, and the stated inverse undoes each branch. ∎

The unused value 7 serves as the run escape in the number column. In the link columns codes are written as $c + 1$, which frees 0 instead. Appendix A machine-checks, for every $n < 2^{16}$, the LEB128, short and uint parts of Theorem 1 and the unique-decoding property behind Theorem 2 for those three codes, and it checks Theorem 3 in full. The number-column code is not covered there.

4 Wire format

The first bit of a payload selects one of two modes. **Single-value mode** (bit 1) covers primitives and empty containers. **Structured mode** (bit 0) covers every non-empty array or object.

4.1 Single-value mode

A single-value payload is `1 · u · code(6)`, where `u` is 1 exactly for a non-negative integer,, optionally followed by a payload (Table 2). Non-negative integers below 63, the literals `null`, `true`, `false`, `""`, `[]` and `{}`, and any single ASCII letter each fit in exactly one byte. Integer-valued doubles are written as exact integers.

Floats. Let v be a finite double that is not an integer, and let D be the shortest decimal string that rounds to v. The *precision* p is the number of digits of D after the decimal point when D is written without an exponent. The *mantissa* m is the digit string of $|D|$ read as an integer. For example, 3.14 has $p = 2$ and $m = 314$, and $1.5{\cdot}10^{-7}$ has $p = 8$ and $m = 15$.

Lemma 1. *The double nearest to $m{\cdot}10^{-p}$ is $|v|$.*

Proof. $m{\cdot}10^{-p} = |D|$ exactly, and D rounds to v by its definition. ∎

So `3.14159` takes 5 bytes and `-3.14` takes 4. MessagePack needs 9 bytes for each, since neither value is exact in binary32.

value	bits	hex
`null` / `true` / `{}`	`1 0 000000` / `…000001` / `…000101`	80 / 81 / 85
int 0 ≤ v < 63	`1 1 v(6)`	3 → c3
int v ≥ 63	`1 1 111111 leb(v-63)`	63 → ff00
int v < 0	`1 0 000110 uint(-v)`	−1 → 8608
float + / −	`1 0 000111 / 001000 uint(p) uint(m)`	3.14 → 87177404 −3.14 → 88177404
one letter	`1 0 (idx+9)(6)`	"a" → a3
one other character	`1 0 111101 leb(unit)`	"!" → bd21
base64url string	`1 0 111110 short(n) 6·n`	"ab" → be269b
other string	`1 0 111111 short(n) n·leb(unit)`	"a b" → bf36120620

Table 2. Single-value codes. Letters are indexed A–Z = 0–25 and a–z = 26–51. The base64url alphabet is A–Z, a–z, 0–9, -, _, in that order. Other strings are written as UTF-16 code units. All hex values are computed from the grammar.

4.2 Structured mode: slots, leaves and links

A structured document is walked in pre-order. Every container (the root included, empty or not) and every object key is allocated a *slot*, numbered 0, 1, 2, … in the order of the walk. Every primitive is a *leaf*. Every empty container also contributes one *marker leaf*, which the container owns itself. Each slot $s > 0$ records its parent through a **key link** (parent + 1). The parent of a key is its object. The parent of a container is the key slot of the member it is the value of, or the array that holds it. Each leaf records its owner through a **value link** (owner + 1). The owner is the key slot for an object member's value, the array slot for an element, or the container itself for a marker. Arrays carry no indices. The only count written explicitly is the number of leaves, R.

Theorem 4 (slot count). *In a structured payload that encodes a complete document (an anchor, not a delta payload), with n slots and value links $v_1, \ldots, v_R$, $n = \max_j v_j$.*

Proof. Every owner is a slot, so every $v_j \le n$. Let $s = n - 1$ be the last slot in pre-order. If s is a key, its value follows it in pre-order. A container value would receive a slot after s, so the value is a leaf owned by s. If s is an empty container, it owns its marker. If s is a non-empty container, its children follow it. None of them is a key or a container, because each of those would take a slot after s. So s is an array, and its elements are leaves that s owns. In every case some $v_j = s + 1 = n$. ∎

Lemma 2. *The subtree of every slot contains a leaf.*

Proof. By induction on height. An empty container owns its marker, and a non-empty array or object has an element or a key whose subtree contains a leaf. A key's value is either a leaf or a container. ∎

Theorem 5 (pre-order reconstruction). *Let $o_k = v_k - 1$ be the owner of the k-th leaf in pre-order. Then slot t precedes leaf k in the pre-order sequence of slots and leaves if and only if $t \le \max_{j \le k} o_j$.*

Proof. (⇐) If $t \le o_j$ for some $j \le k$, then slot t is slot o_j or precedes it. Slot o_j precedes the leaf j that it owns, and leaf j is leaf k or precedes it. (⇒) Suppose slot t precedes leaf k. If leaf k lies in the subtree of t, its owner lies in that subtree too, and so $o_k \ge t$. Otherwise, the subtree of t is a contiguous block of the pre-order that starts at t and ends before leaf k. By Lemma 2 the block contains a leaf $j < k$, whose owner lies in the subtree of t, so $o_j \ge t$. ∎

Theorem 5 lets the decoder interleave the slot sequence and the leaf sequence back into one pre-order walk. Key links give each slot's parent, value links give each leaf's owner, and the interleaving gives the order of children, including array elements. The tree therefore needs no indices or end markers.



Figure 2. Structured-mode encoding. One deterministic pass produces twelve column streams, which are concatenated in a fixed order and zero-padded to a byte boundary.

4.3 Link columns

Value links and key links are coded the same way, each in its own column, against the previous link in that column. The first link is coded against 0. A link v whose difference d from its predecessor lies in [−3, 3] is *delta-flagged* and written as $w(d) + 1$ in 3 bits. Any other link is *literal-flagged* and written as $v + 1$ in W bits, where W is a running width that starts at 1. Suppose c slots have been allocated when a literal link is produced. Before writing it, the encoder raises W to $b(c + 1)$ by emitting one all-zero group at the current width for each extra bit.

Proposition 6. *A literal link always fits in W bits and is never zero, so a decoder that reads W bits and widens W by one on every zero group tracks the encoder's W exactly.*

Proof. A value link produced after c slots names an owner $\le c - 1$. A key link for slot c names a parent $\le c - 1$. In both cases $v \le c$, so $1 \le v + 1 \le c + 1 < 2^{b(c+1)}$. A zero group is never a literal, so it is always a widening step. ∎

Runs. Let x be a link's delta code if it is delta-flagged and its value if it is literal-flagged. A maximal stretch of $L \ge 4$ consecutive links with the same x whose first link is delta-flagged is written `000 · short(L) · (x+1)`, and W does not change inside a run. Within a run, each link is read through its own flag, so a literal-flagged link in a run has the value x. If such a stretch starts with a literal-flagged link, that link is written on its own and the stretch is reconsidered from the next link. The first link of a run is delta-flagged, so the decoder meets the escape `000` only where a 3-bit delta code is expected, and there it cannot be a delta codeword.

Flag columns. The value flags, the key flags and the booleans each begin with a 2-bit mode: `00` all zero and `01` all one, both with no body, or `10` mixed, followed by the raw bits. A flag column in which every link has the same flag therefore costs 2 bits whatever the document size. In the worked example of §4.6 both flag columns are uniform.

4.4 Keys and strings

The `keys` column has one 2-bit kind per slot: `00` array, `01` object, `10` base64url key or dictionary reference, `11` other key. A new key of ℓ characters adds `short(ℓ+1)`, and its characters go to the `kvals` column: 6 bits each for a base64url key, or one LEB128 per UTF-16 unit otherwise. The empty key is kind `11` with `short(1)`. A key seen before is kind `10` with `short(0)`, and its dictionary index i goes to `kvals` as `short(i)`. A single dictionary is shared by keys and string values. It lists distinct strings in pre-order of first occurrence, and by Theorem 5 the decoder can rebuild that order.

A string value goes to the `vals` column. A new string of ℓ units is written `short(ℓ)` followed by its characters, in 6-bit base64url (type 2) when $\ell > 0$ and every character is in the alphabet, and in LEB128 units (type 7) otherwise. A repeated string is written `short(0) · 0 · short(i)` (type 2). After `short(0)` in type 2, the bit value 1 is reserved for a string-diff reference in delta payloads. A key that recurs with dictionary index i therefore costs exactly $6 + L_{\text{short}}(i)$ bits in `keys` and `kvals` (10 bits for $i < 4$), plus its key link.

4.5 Types, booleans and numbers

Every leaf has a 3-bit type: 1 null, 2 base64url or referenced string, 3 boolean, 4 non-negative integer, 5 negative integer, 6 float or empty-container marker, 7 other string. The code `000` is an escape. `000 · short(L) · t` with $L \geq 1$ stands for L leaves of type t, and the encoder uses it for every maximal run with $L \geq 4$. `000 · short(0)` introduces the delta-only leaves of §5. Booleans take one bit each in a flag column.

Numbers share one chain in leaf order. An integer contributes its magnitude, since the sign is in the type. A float contributes a header h, then $p + 1$ if $p \geq 3$, then its mantissa m. The header is $h = p + 1$ for $p \leq 2$ and $h = 0$ for $p \geq 3$, plus 4 when the float is negative. A marker contributes the single item 1 for `[]` or 5 for `{}`. Each item x_i is coded against x_{i-1}, with $x_0 = 0$. If both are below 2^{32} and their difference lies in [−3, 3], the item is the delta codeword `00`·$w(d)$. Otherwise it is a literal (Table 1). Three or more consecutive identical codewords are packed as `00 111 · short(L) · codeword`. The array of integers 1 to 1000 therefore encodes in 13 bytes: every item is the delta codeword for +1, and they form a single run.

4.6 Column order and a worked example

The twelve columns are emitted as `dc` (the mode bit 0 and `short(R)`), vflags, vlinks, kflags, klinks, keys, kvals, types, bools, nums, vals and strdiffs, followed by padding. A column with nothing to hold is empty. In particular, kflags is absent when $n = 1$ and bools is absent when no leaf has type 3. Figure 3 decomposes `{"a":1,"b":2,"c":[3,4]}`, which encodes to 100 bits (13 bytes). The same document is 12 bytes in MessagePack and in CBOR, and 23 in JSON.

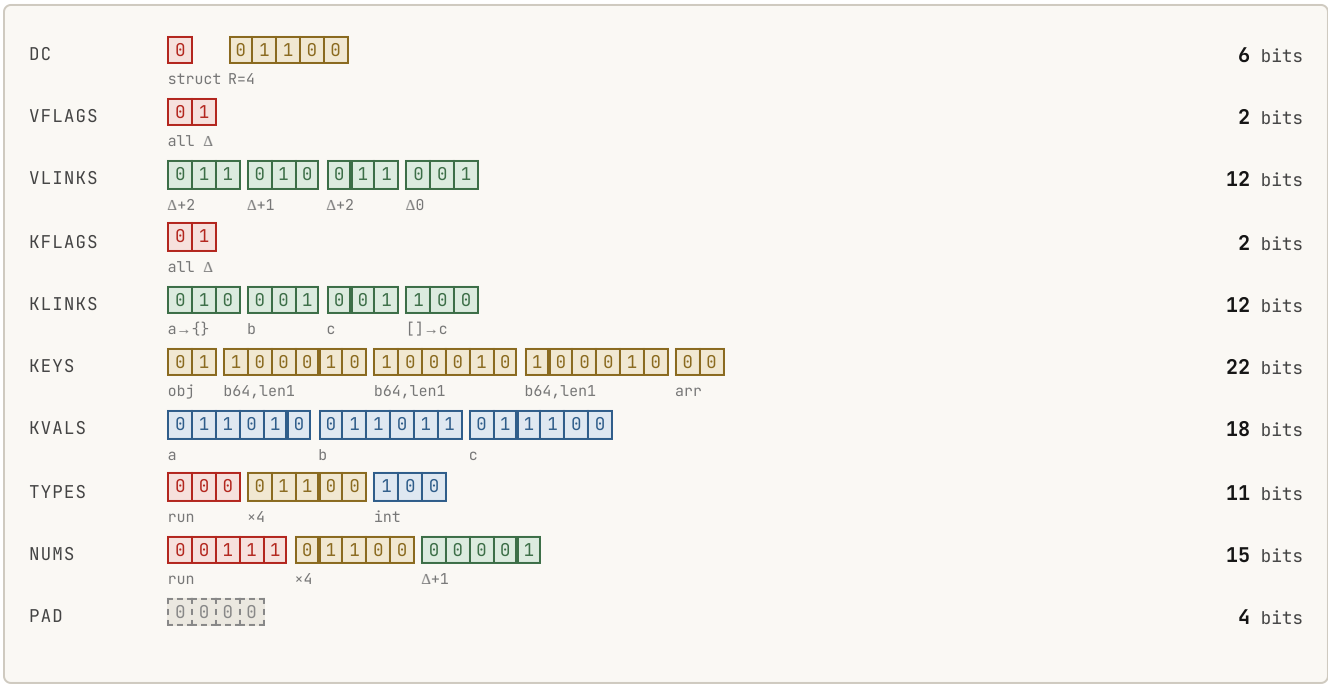


Figure 3. Complete bit decomposition of `{"a":1,"b":2,"c":[3,4]}`: 100 data bits + 4 pad bits = 13 bytes (`3169951318a28869b70321d810`). Slots: 0 the object, 1–3 the keys, 4 the array. The value links 2, 3, 5, 5 give $n = 5$ (Theorem 4). The four values 1, 2, 3, 4 become a single run of "Δ+1" in the number column.

4.7 Decodability

Theorem 7. *Every column length is determined by R and the columns before it. Hence a structured payload is decoded by reading its columns in order, with no per-column counts or field markers.*

Proof. `dc` gives R, and vflags then holds R flags. The link column is consumed flag by flag, using runs

(§4.3) and Proposition 6. Theorem 4 gives n, so kflags holds $n - 1$ flags, which in turn determine klinks. keys holds n kinds, and their lengths and reference flags determine kvals. types is read until R leaves are typed. The number of type-3 leaves determines bools. The types 4–6, together with the headers h, determine how many number items follow. The types 2 and 7 determine vals. Every field is a codeword of a prefix-free code (Theorem 2). ∎

Round trip. Take any JSON value whose numbers are finite doubles other than −0. Decoding inverts each column (Theorems 2–3, Proposition 6). It rebuilds the tree (Theorems 4–5) and the dictionary (by pre-order). It recovers every float exactly (Lemma 1). Decoding the encoding therefore returns the original value, with member order preserved. As a computational check, a decoder transcribed from this specification reproduced every input of §6, as well as the first $2{\cdot}10^4$ documents of the Appendix B generator (which include the 1000 of §6.4) and $2{\cdot}10^5$ random trees of depth at most 6 with integer leaves 0–3 and keys `k0`–`k9`.

5 Delta chains

The grammar's second use is storing a document's history. A *state* S is the decoded table of a document: its slots with kinds and key links, its leaves with types, values and value links, and the string dictionary. Let S have n slots. A *delta payload* against S is a structured payload in which links may name the existing slots 0, …, $n - 1$. New slots are numbered from n, the literal width starts at $b(n + 1)$, and kflags, klinks and keys cover only the new slots. (Theorems 4 and 7 are stated for anchors. This report does not specify how a decoder obtains the number of new slots in a delta payload.) Dictionary references may name entries of S.

Leaf semantics. A leaf owned by an existing key slot replaces that member's value. A leaf owned by a new key slot adds a member, and a leaf owned by an existing array slot appends an element. Two leaf forms exist only in deltas and use the type escape. `000 · short(0) · 0` is a *deletion leaf*, which removes the member or element it is linked to. `000 · short(0) · 1 · short(i) · short(r) · t` is a *positional leaf* for array index i: with $r = 0$ it inserts a leaf of type t before index i, and with $r = 1$ it replaces element i, or removes it when $t = 000$. Neither form can be confused with a run, because runs have $L \geq 1$. Applying a payload merges it into S, drops unreachable slots and dictionary entries, and renumbers both densely in pre-order. When an edit cannot be expressed against S, such as a change of root type, the chain ends, and a new chain starts with a new anchor.

Chains. A chain is a byte sequence `LEB128(|p₀|) · p₀ · LEB128(|p₁|) · p₁ ⋯`, where p_0 is a full document (the *anchor*) and every later p_k is a delta payload. The states are $S_0 = \text{decode}(p_0)$ and $S_k = \text{apply}(S_{k-1}, p_k)$.

Proposition 8 (framing). *A chain splits into its entries in exactly one way, whether or not the payload grammar is self-delimiting.*

Proof. LEB128 is prefix-free (Theorem 2). Each length prefix is therefore read unambiguously, and it states exactly how many bytes the payload occupies. ∎

Proposition 9 (prefix replay). *For every k, the prefix of a chain that ends after entry k replays to S_k.*

Proof. By induction on k: S_k depends only on S_{k-1} and p_k, and Proposition 8 recovers p_0, …, p_k from the prefix. ∎

A payload is not meaningful on its own. Compaction renumbers slots, so an address in p_k refers to a position in S_{k-1} and nothing else, and entries must be replayed in order.

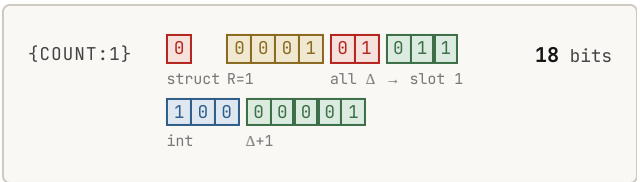


Figure 4. The delta that turns `{count:0}` into `{count:1}`: 18 bits, or 3 bytes after padding. Its one leaf links to the existing key slot 1 and replaces its value. No slots are new, so kflags, klinks, keys and kvals are empty.

Example. The anchor `{count:0}` takes 62 bits. These are `dc` 5, vflags 2, vlinks 3, kflags 2, klinks 3, keys 2 + 7 (object kind, then kind `10` and `short(6)`), kvals 30 (five 6-bit characters), types 3 and nums 5, which pad to 8 bytes. A delta that sets the counter to x is Figure 4 with the number codeword for x against 0. That codeword is 5 bits for $x \le 3$, 10 bits for $x = 100$ and 26 bits for $x = 100000$, so the payloads are 3, 3, 3 and 5 bytes. Table 3 compares the chain with storing every version in full.

version	full encoding	chain entry
`{count:0}` (anchor)	8	1 + 8
`{count:1}`	8	1 + 3
`{count:2}`	8	1 + 3
`{count:100}`	9	1 + 3
`{count:100000}`	11	1 + 5
total	44	27

Table 3. Bytes for five versions of a counter, derived from the grammar. Chain entries include their 1-byte length prefix. The five full encodings total 44 bytes, or 49 with the same framing.

6 Size evaluation

6.1 Method

All inputs are generated by the deterministic procedures of Appendix B from the 32-bit generator defined there. The seed is `0x5eed2026` plus a per-dataset offset: users +0, time series +4, random documents +5, sensor +7. Sizes for this encoding are computed from the specification and include the final padding. Sizes for the other formats are computed from their specifications. JSON is the UTF-8 length of the text with no insignificant whitespace, with numbers in shortest round-trip form [9]. MessagePack is the shortest encoding that its specification permits [1]: the narrowest integer family, the narrowest length header (a length packed into the type byte, or an 8-, 16- or 32-bit length) the specification allows for each string, array and map, and binary32 exactly when the value is representable in it, binary64 otherwise. CBOR uses preferred serialization [2], with shortest-form arguments and the narrowest of binary16, binary32 and binary64 that represents the value exactly. Every size is a deterministic function of the input. Every input listed here was decoded back from its encoding and compared with the original.

6.2 Size

input	JSON	this work	MessagePack	CBOR	ratio to MessagePack
{a:1,b:2,c:[3,4]}	23	13	**12**	**12**	1.083
−3.14 (single value)	5	**4**	9	9	0.444
3.14159 (single value)	7	**5**	9	9	0.556
users, N = 10	790	**305**	557	567	0.548
users, N = 100	7 989	**2 895**	5 526	5 680	0.524
users, N = 1000	80 873	**30 632**	56 920	57 882	0.538
sensor floats, N = 1000 (2 dp)	5 885	**3 015**	8 787	8 709	0.343
time series {t, v}, N = 1000	24 954	14 040	**11 617**	12 003	1.209
integers 1…1000	3 894	**13**	2 621	2 725	0.005

Table 4. Encoded size in bytes. The smallest in each row is bold. *users* records are `{id, name, age, active, score, tags[]}` with sequential ids. In the time series `t` increases by 60 and `v` follows an integer random walk. Appendix B defines every generator.

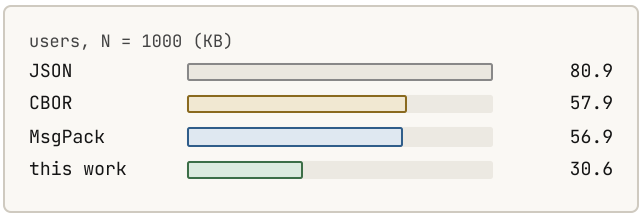


Figure 5. Encoded size of 1000 user records, with bars scaled to JSON = 100%. Repeated keys become dictionary references, and sequential ids and repeated types become runs (Table 5).

6.3 Column accounting

Because the columns are separate, the size of an input splits exactly by column (Table 5). In *users* all 6756 value links are delta-coded, so vflags costs 2 bits. The 8000 key links have mixed flags, and 2998 of them are literals. In every record after the first, three key links differ from the preceding key link by more than 3: the object's link (to the root array), the first key's link (to the new object) and the tags array's link (to its key). The kflags column therefore stores its 8000 bits raw.

column	users N=1000	time series N=1000	sensor N=1000
dc	19	19	19
vflags + vlinks	2 + 19 268	2 + 6 000	2 + 27
kflags + klinks	8 002 + 47 019	3 002 + 24 350	0 + 0
keys + kvals	40 007 + 27 117	14 002 + 8 004	2 + 0
types	20 268	24	162
bools	1 002	0	0
nums	50 393	56 912	23 903
vals	31 956	0	0
total bits (slots / leaves)	245 053 (8001 / 6756)	112 315 (3001 / 2000)	24 115 (1 / 1000)

Table 5. Bits per column. strdiffs is empty in every anchor. Each total, padded to whole bytes, gives the corresponding size in Table 4.

6.4 Random documents

To avoid tuning for favourable inputs, we generated 1000 random documents (Appendix B). 854 are objects with 1–8 keys from a 16-word pool, and 146 are two-element arrays. Values nest to depth 4 and mix integers of five magnitude scales, two-decimal and full-precision floats, strings, booleans and nulls. Each key of a nested object is, with probability 0.2, a random string of printable non-digit characters. The JSON texts range from 7 to 582 bytes, with a median of 89. Against MessagePack, the encoding was smaller on 791 documents, equal on 88 and larger on 121. The per-document size ratio ranged from 0.474 to 2.000, with a median of 0.896 and a geometric mean of 0.895. Against CBOR it was smaller on 781, equal on 96 and larger on 123. In total the encoding used 64 127 bytes, against 74 004 for MessagePack (−13.3%), 73 907 for CBOR and 113 350 for JSON. The extremes are small. The ratio 2.000 is `[{},[]]`, which takes 6 bytes here against 3, while `[83.84,-24.23]` takes 9 bytes against 19.

6.5 Where the encoding is larger

Interleaved numeric fields. All numbers share one chain in leaf order. In the time series, consecutive items alternate between a timestamp near $1.7{\cdot}10^9$ and a value $v \in [85, 201]$, so no difference falls in the window and every item is a literal. Each timestamp is a 2-bit selector plus a 5-byte LEB128 (42 bits). Each v is 10 bits below 128 (386 rows) and 18 bits otherwise (614 rows). Together these give the 56 912 bits of nums in Table 5, 7 114 bytes in all, while MessagePack spends 5 bytes on each `t` and 1–2 on each `v`. Key

links add 24 350 bits: 1997 of the 3000 key links are literals, because from the third record on each record's object and first key jump outside the window (in the second record only the first key does). Grouping numbers by kind rather than by field path is what loses this case. Within the field `t` alone, consecutive differences are all exactly 60 and second differences are all 0, so a chain per field path with second-order differences would give `t` a delta code of 0 from its third row on.

Tiny values. In single-value mode, for integers in [−70 000, 70 000] the encoding is larger than MessagePack exactly on [−32 768, −64], [−32, −1], [63, 127], [191, 255] and [16 447, 65 535], and smaller on [65 536, 70 000]. Tiny structured documents also carry a few bits of fixed column overhead: the worked example takes 13 bytes, against 12 in MessagePack and CBOR.

Decoding work. Fields are not byte-aligned, so every read or write of a field needs shifts and masks across byte boundaries. Columns carry no offsets, so the start of a column is known only after all earlier columns have been decoded (Theorem 7). Byte-aligned, row-oriented formats avoid both kinds of work.

7 Discussion

Once a structured document's structure and values are stored as separate streams of small integers, much of its regularity becomes visible to simple codes. Key dictionaries, type runs, delta links and two-bit flag columns together bring record arrays to 52–55% of MessagePack's size *without* a general-purpose compressor and without a schema. The same machinery gives an edit format in which small changes cost a few bytes (Table 3), which helps wherever every historical version is kept.

The costs are just as precise. The selector codes lose 2 bits on every value from 16 up (Theorem 1). Grouping by kind rather than by field path loses on interleaved numeric series, and Table 5 shows that the loss sits in nums and klinks. Delta payloads are meaningful only against the exact preceding state (Proposition 9). The structural claims of §§3–4 are proved and every size is computed from the format's definition on stated inputs. The round-trip property is argued and tested rather than proved, and the delta semantics of §5 are specified only informally.

8 Conclusion

We specified a deterministic, bit-granular, column-oriented JSON encoding that also serves as an append-only update format. We proved that its integer codes are prefix-free, gave their lengths in closed form, and proved that its tree can be rebuilt from two link columns and a single count. On arrays of records it takes 52–55% of MessagePack's size, a five-version counter history fits in 27 bytes, and on interleaved numeric series it is 21% larger than MessagePack.

References

1. *MessagePack specification*. msgpack.org, 2008–.
2. *Concise Binary Object Representation (CBOR)*. RFC 8949, 2020.
3. *DWARF Debugging Information Format, Version 5*, §7.6 (LEB128), 2017.
4. Run-length encodings. *IEEE Trans. Inf. Theory* 12(3):399–401, 1966.
5. Universal codeword sets and representations of the integers. *IEEE Trans. Inf. Theory* 21(2):194–203, 1975.
6. Interactive analysis of web-scale datasets. *PVLDB* 3(1–2):330–339, 2010.
7. *JavaScript Object Notation (JSON) Patch*. RFC 6902, 2013.
8. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259, 2017.
9. *ECMAScript Language Specification* (ECMA-262), Number::toString.

A Machine-checked statements

The following Lean 4 file (core library only) defines the codes of §3 over bit lists, together with their decoders. It checks Theorem 1, the unique-decoding property behind Theorem 2 (each decoder consumes exactly one codeword and leaves any following bits untouched) and the comparison with LEB128 for every $n < 2^{16}$, and it checks Theorem 3 in full.

```
/-- `w`-bit big-endian representation of `v` (low `w` bits). -/
def fixed (v : Nat) : Nat → List Bool
  | 0 ⇒ []
  | w + 1 ⇒ v.testBit w :: fixed v w

/-- Read `w` bits big-endian. -/
def readFixed : Nat → List Bool → Option (Nat × List Bool)
  | 0, s ⇒ some (0, s)
  | w + 1, b :: s ⇒ (readFixed w s).map fun (x, r) ⇒ ((if
b then 2 ^ w else 0) + x, r)
  | _ + 1, [] ⇒ none

/-- LEB128: 7-bit groups, least significant first; top bit 1
= more groups follow. -/
def leb (n : Nat) : List Bool :=
  if n < 128 then fixed n 8 else fixed (128 + n % 128) 8 ++
leb (n / 128)
decreasing_by omega

def readLeb : Nat → List Bool → Option (Nat × List Bool)
  | 0, _ ⇒ none
  | fuel + 1, s ⇒ do
    let (g, r) ← readFixed 8 s
    if g < 128 then pure (g, r) else do
      let (hi, r') ← readLeb fuel r
      pure (g - 128 + 128 * hi, r')

/-- The `short` code: selector 00/01/10 picks a 2/3/4-bit
payload, 11 escapes to LEB128. -/
def short (n : Nat) : List Bool :=
  if n < 4 then fixed 0 2 ++ fixed n 2
  else if n < 8 then fixed 1 2 ++ fixed n 3
  else if n < 16 then fixed 2 2 ++ fixed n 4
  else fixed 3 2 ++ leb n

def readShort (s : List Bool) : Option (Nat × List Bool) :=
do
  let (sel, r) ← readFixed 2 s
  match sel with
  | 0 ⇒ readFixed 2 r
  | 1 ⇒ readFixed 3 r
  | 2 ⇒ readFixed 4 r
  | _ ⇒ readLeb 64 r

/-- The `uint` code: 3/4/6-bit payloads, then LEB128. -/
def uint (n : Nat) : List Bool :=
  if n < 8 then fixed 0 2 ++ fixed n 3
  else if n < 16 then fixed 1 2 ++ fixed n 4
  else if n < 64 then fixed 2 2 ++ fixed n 6
  else fixed 3 2 ++ leb n

def readUint (s : List Bool) : Option (Nat × List Bool) :=
do
  let (sel, r) ← readFixed 2 s
  match sel with
  | 0 ⇒ readFixed 3 r
  | 1 ⇒ readFixed 4 r
  | 2 ⇒ readFixed 6 r
  | _ ⇒ readLeb 64 r

/-- Bit length `b(n)`, with `b(0) = 1`. -/
def b (n : Nat) : Nat := if n < 2 then 1 else n.log2 + 1

def lenLeb (n : Nat) : Nat := 8 * ((b n + 6) / 7)

/-- A tail that must survive decoding untouched. -/
def tail : List Bool := [true, false, true, true]

-- Theorem 1 (length functions), checked for every n < 2^16.
example : (List.range (2 ^ 16)).all (fun n ⇒
    (leb n).length = lenLeb n &&
    (short n).length = (if n < 4 then 4 else if n < 8 then
5 else if n < 16 then 6
                          else 2 + lenLeb n) &&
    (uint n).length = (if n < 8 then 5 else if n < 16 then
6 else if n < 64 then 8
                        else 2 + lenLeb n)) := by
native_decide

-- Theorem 2 (unique decodability): each decoder consumes
exactly one codeword and
-- leaves any following bits untouched, checked for every n
< 2^16.
example : (List.range (2 ^ 16)).all (fun n ⇒
    readShort (short n ++ tail) = some (n, tail) &&
    readUint (uint n ++ tail) = some (n, tail) &&
    readLeb 64 (leb n ++ tail) = some (n, tail)) := by
native_decide

-- The short code against LEB128: -4, -3, -2 bits below 16
and +2 bits from 16 on.
example : (List.range (2 ^ 16)).all (fun n ⇒
    ((short n).length : Int) - lenLeb n =
      (if n < 4 then -4 else if n < 8 then -3 else if n < 16
then -2 else 2)) := by
  native_decide

/-- The delta window: d ∈ [-3, 3] ↦ code in 0..6. -/
def window (d : Int) : Nat := if d ≥ 0 then d.toNat else (3
- d).toNat
def unwindow (c : Nat) : Int := if c ≤ 3 then c else -((c :
Int) - 3)

-- Theorem 3: the window is a bijection [-3, 3] → {0, …, 6};
code 7 is never produced.
example : ∀ d : Fin 7, unwindow (window ((d : Int) - 3)) =
(d : Int) - 3 := by decide
example : ∀ c : Fin 7, window (unwindow c) = c ∧ window
(unwindow c) < 7 := by decide
```

B Data generators

Generator. The state a is a 32-bit word initialised to the seed. Each draw sets $a \leftarrow a$ + 0x6D2B79F5 mod 2^{32}. It then computes $t \leftarrow (a \oplus (a \gg 15)) \otimes (a \mid 1)$ and $t \leftarrow ((t + (t \oplus (t \gg 7)) \otimes (t \mid 61)) \bmod 2^{32}) \oplus t$, and returns $r = ((t \oplus (t \gg 14)) \bmod 2^{32})/2^{32}$. Here $\otimes$ is multiplication mod 2^{32} and $\gg$ is a logical shift. One draw of int(a, b) gives $a + \lfloor r \cdot (b - a + 1) \rfloor$. two($x$) is round($100x$)/100 in binary64, where round goes to the nearest integer with ties toward $+\infty$, and -0 is replaced by 0. Draws are made in the order written.

- **users** (seed +0). For i = 1…N: name = NAMES[int(0,15)], age = int(18,77), active = (r < 0.5), score = two(100r), k = int(0,3), then k tags TAGS[int(0,7)]. The record is `{id: i, name, age, active, score, tags}`. NAMES = ant, bee, cat, dog, eel, fox, gnu, hen, ibis, jay, kiwi, lark, mole, newt, owl, puma. TAGS = red, green, blue, admin, beta, pro, new, vip.
- **time series** (seed +4). Row i = 0…N−1 is `{t: 1700000000 + 60i, v: v_i}`, with v_0 = 100 and $v_{i+1} = v_i$ + int(−5,5).
- **sensor** (seed +7). N elements two(20 + 10r).
- **random documents** (seed +5). Each document: if r < 0.85, an object whose keys are keys(int(1,8), 1), with each value value(1) in key order; otherwise [value(1), value(1)].
  value(d): if d < 4 and r < 0.25, then, if r < 0.5, an object with keys keys(int(0,6), 0.8) and values value(d+1), else an array of int(0,4) values value(d+1). Otherwise prim().
  prim(): draw $x = r$. If x < 0.2: $u = r$, M = (16, 256, 65536, 2^{31}, 2^{48})[int(0,4)], $m = \lfloor uM \rfloor$, $s = r$, giving $-m$ if s < 0.3 and $m \neq 0$, else m. If x < 0.35: two(200r − 100). If x < 0.45: 1000(2r − 1). If x < 0.7: str(). If x < 0.85: r < 0.5. Otherwise null.
  str(): ℓ = int(0,12), b = (r < 0.7), then ℓ characters, each B64[int(0,63)] if b, else P[int(0,94)].
  keys(k, q): draw until k distinct keys are collected or 100 draws are made. Each draw is POOL[int(0,15)] if r < q, else a string of int(1,8) characters K[int(0,84)]. A key is kept if it is new.
  P is printable ASCII 0x20–0x7E in order, K is P without the digits, and B64 is the alphabet of Table 2. POOL = id, name, type, value, count, items, data, meta, user, time, status, tags, size, flag, memo, ref.