Technical paper · jsonW 1.3.11

jsonW: one data model, three lossless forms and permissions inside the format

Wortic Tecnologia · November 13, 2026 · text 1.1, on jsonW 1.3.11

jsonw.org/paper/en · github.com/developer-wortic/jsonw

CONTENTS

ABSTRACT

JSON and JSONL became the common language of data, but they were designed to exchange messages between programs, not to store and query the volume and variety of data that AI systems use. Reading one field of a JSONL file means parsing the file from the start; money, measures and exact decimals have no type; vectors become text; permissions and proof of integrity live outside the data. This paper describes jsonW (json Wondrous) 1.3.11, a data model with three forms that convert without loss: row, for messages and real time; column, the `.jsw` file in blocks read through `mmap`; and text, in which every valid JSON is also valid jsonW-text. The format carries business types, vectors and tensors, nineteen built-in features (search, ranking, maintenance, similar spelling and audio, text and video signals), permission views and hash-based custody. On 100,000 synthetic records, on one machine, the `.jsw` file was 6.6 times smaller than uncompressed JSONL and read one column about 2,300 times faster than our own JSON reader. On the real INPI trademark base, 55.3 million rows and 12.53 GB of CSV, the `.jsw` was 6.5 times smaller than the CSV and 12.8 times smaller than the XML, and counted the 2.3 million records in force in 43 ms, faster than Parquet read by DuckDB.

Keywords: data format; columnar storage; JSON; vector search; access control; integrity; post-quantum key exchange.

1 Introduction

JSON [1] was born to exchange messages between programs in a text people can read. JSON Lines [2] took the idea to files: one record per line. Both became de facto standards for logs, exports, training sets and API responses. They are simple, universal and readable, and those qualities explain their success. Working with data for AI, however, asks for things they were not made to give.

1.1 What JSON and JSONL lack

- **Size.** Everything is text. A 384-dimension vector becomes thousands of characters per record: the 100,000 records of the test in Section 5 take 519.3 MB as JSONL.
- **Access.** There is no index. To read one field of one record, the file must be parsed from the start up to it; to read one column, the whole file.
- **Types.** The specification does not fix the precision of numbers: RFC 8259 leaves the limit to each implementation and advises relying only on the precision of an IEEE 754 double [1, 28]. There is no type for money with a currency, a measure with a unit, an exact decimal, a code that must not lose its leading zero, a date, a vector or a tensor. Each system invents a convention, and moving between conventions loses data.
- **Permissions.** Who may see which column and which row lives outside the data, in a separate layer, and every query must remember to apply it.
- **Custody.** Nothing in the file proves it was not altered.

In practice, the same data goes through a chain: JSON on the wire, CSV or Parquet for storage, a vector database for semantic search and a permission layer to decide who sees what. Each step costs time and memory and, sometimes, the data itself: the price that turns into text, the tax ID that loses its leading zero, the weight nobody knows was grams or kilograms.

1.2 Contributions

This paper describes jsonW (json Wondrous) version 1.3.11. Its contributions are:

1. a data model with three forms that convert without loss (row, column and text), in which every valid JSON is also valid jsonW-text (Section 2);
2. business and AI types declared in the schema itself (Section 2.3);
3. nineteen features inside the file, from search and ranking to audio, text and video signals, shown with the public tool when it has a command for them (Section 3);
4. column, row and cell permissions and hash-based custody as part of the format (Section 4);
5. reproducible measurements, with methodology, data and the limits of the measurement (Section 5), and a comparison with JSON, JSONL, MessagePack, CBOR, Protobuf, Parquet and Arrow (Section 6).

1.3 What this text does not describe

The jsonW core is written in C, with no dependency beyond the standard library, and is maintained by Wortic Tecnologia; it is not open source. What is public, and what this text uses, is the usage documentation, the C ABI (`sdk/jsonw.h`), the SDKs, the tool binaries and the observable behavior [20]. The internal binary layout, the compression codecs and the inner organization of a block are out of scope. What we state about the format and the tool can be checked with the public binaries; the internal evidence (fuzzing, hostile loads, cross-SDK comparison) is ours and marked as such. When a feature exists in the format but has no command in the public tool yet, we say so.

2 Data model

2.1 Three forms

jsonW is a single data model with three ways to store the same value (Figure 1). Moving from one form to another loses no value and changes the type of no value.

- **Row:** binary, one value after another, for messages, events and real time. It takes the place of JSON, MessagePack, CBOR and Protobuf. It is the form that runs on the wire (Section 4.5).
- **Column:** the `.jsw` file, in blocks stored by column and read straight from disk through `mmap` , with no copy. It takes the place of JSONL, CSV and Parquet.
- **Text:** jsonW-text, for people to read and write. It takes the place of JSON, TOML and YAML. Every valid JSON is already valid jsonW-text; the text adds notations for the types JSON lacks, such as `12.50 BRL` , `250 g` and `t"2026-11-09T01:00:00.000000Z"` .

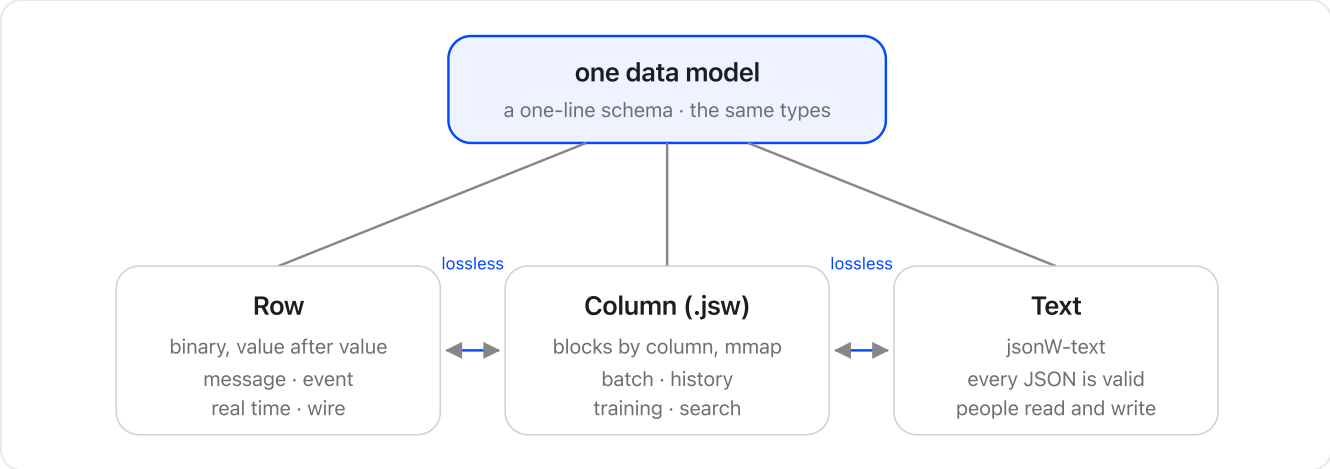


Figure 1. One model, three forms. Schema and types are the same in all three; converting between them loses no value and changes no type.

2.2 Schema

The schema is a line of text with `name:type` pairs separated by commas. The question mark accepts null, and marks after the type declare permissions (Section 4.3). The type and command names are Portuguese words: `texto` is text, `dinheiro` is money, `medida` is measure, `codigo` is code, `tempo` is time, `pessoal` is personal.

```
id:i64,produto:texto,preco:dinheiro,peso:medida[g],sku:codigo,criado:tempo,cpf:codigo?+pessoal
```

Listing 1. A schema. `cpf` (a Brazilian personal tax ID) accepts null and is personal data: without the `pessoal` scope, it comes out null.

The schema evolves without breaking readers: a new field is null in old rows, a name never changes type and the reader resolves fields by name. So an old file is read by new code, and a new file by old code [20].

2.3 Types

Table 1 lists the types of version 1.3.11. Business types are checked on write: an input that does not fit the type is rejected with the reason, instead of being stored wrong.

Table 1. jsonW 1.3.11 types [20].

Type	In the schema	What it guarantees
exact decimal	`decimal[2]`	never goes through floating point: 12.50 stays 12.50
money	`dinheiro`, `dinheiro[4]`	currency in every value, by ISO 4217 [27] code or by symbol (`12,50 BRL`, `US$`, `€`, `₿`); sums only within one currency
measure	`medida`, `medida[kg]`	89 units in 14 quantities; converts on write and rejects another quantity
code	`codigo`	exact bytes: tax IDs, keys, plates and SKUs are never normalized or trimmed
text	`texto`, `texto+nfc`	full Unicode, with accents, emoji and `\0`; `+nfc` normalizes on request [29]
repeated text	`dic`	per-file dictionary: each text stored once
time	`tempo`	microseconds, UTC
url	`url`	validated on write (`https://…`, `mailto:`, `tel:`, `urn:`; up to 8 KiB)
vector	`vf32[n]` `vf16[n]` `bits[n]` `vq8[n]` `vq4[n]`	contiguous matrix in the file, ready for semantic search
tensor	`tensor<bf16>[768,768]`	f32, f16, bf16, u8, i8, i16, i32 elements
media	`midia`	hash, type, size, duration, width and height; jsonW does not fix the codec: media stays in its own open codec
and more	`bool` `i64` `f64` `f32` `uuid` `bytes` `ref` `lista<t>` `obj{…}` `mapa<k,v>` `qualquer`	`qualquer` (any) holds open JSON; nesting up to 1000 levels

The 14 measure quantities are length, mass, volume, energy, time, area, data, speed, temperature, dimensionless, power, pressure, carbon and density. Conversion is exact between decimal multiples (12.5 mm = 0.0125 m) and rounded to the column's scale for the others (1 mi = 1609.344 m, stored as 1609 m in a `medida[m]` column, which has no decimal places; 25 °C = 77 °F).

2.4 Comma or dot

Readers accept both separators: `12,50`, `1.234,56 BRL` and `1,234.56 USD`. When both are present, the last one is the decimal. With only one, that separator is the decimal, except a comma followed by three digits in a dot currency (`USD 1,234` is one thousand) or a dot followed by three digits in a comma currency (`R$ 1.234` is one thousand). The canonical output always uses a dot. And real numbers never depend on the process locale: a JVM or PHP configured for Portuguese does not swap the separator.

2.5 A worked example

Listing 2 shows the trip from JSONL to `.jsw` and back, with the real output of the `wjsw` 1.2.1 tool.

```
# two lines of JSONL, as they arrive from any system
{"id":1,"produto":"Café especial","preco":"12,50 BRL","peso":"250 g","sku":"CAF-0250"}
{"id":2,"produto":"Pão de queijo","preco":"8,90 BRL","peso":"1 kg","sku":"PDQ-1000"}

$ wjsw de-texto "id:i64,produto:texto,preco:dinheiro,peso:medida[g],sku:codigo" vendas.jsonl
vendas.jsw
2 linhas -> vendas.jsw
$ wjsw ver vendas.jsw
{id:1,produto:"Café especial",preco:12.50 BRL,peso:250 g,sku:"CAF-0250"}
{id:2,produto:"Pão de queijo",preco:8.90 BRL,peso:1000 g,sku:"PDQ-1000"}
$ wjsw para-texto vendas.jsw --json
{"id":1,"produto":"Café especial","preco":"12.50 BRL","peso":"250 g","sku":"CAF-0250"}
{"id":2,"produto":"Pão de queijo","preco":"8.90 BRL","peso":"1000 g","sku":"PDQ-1000"}

# what does not fit the type is rejected, with the reason
$ wjsw de-texto "…,peso:medida[g],…" leite.jsonl leite.jsw     # peso: "2 L"
valor 1: medida de outra grandeza perto de ","sku":"LEI-0002"}
```

Listing 2. `1 kg` became `1000 g` because the column is `medida[g]`; a liter is volume and the column is mass, so the row was rejected ("measure of another quantity"; the message shows the input near the error). Money keeps its currency, and the SKU is stored byte for byte. The tool prints its messages in Portuguese.

3 The 19 features

The features below are part of the format and the core, not of a layer on top, and the public tool exposes most of them (Table 2). They are the pieces that, in other stacks, are spread across a search engine, a vector database, an index and a permission layer. The examples in this section were run with the public `wjsw` 1.2.1 binary for Linux x86_64 (Listing 6B with 1.3.11), and the inputs are at jsonw.org/paper/exemplos; long outputs were cut where `…` appears.

Table 2. The 19 features and where they appear in the public 1.3.11 tool and ABI.

#	Feature	What it does	In the tool
1	Term search	sparse vector with per-term (IDF) weights, the calculation search engines use	`avaliar … esparso`
2	Semantic search	cosine over dense `vf32`, `vf16`, `vq8` and `vq4` vectors	`avaliar … cosseno`
3	Multilinear grid	bilinear, trilinear and higher interpolation, straight from the file	`avaliar … grade`
4	Score by request	weights, cosine, sparse or grid in a one-line request	`avaliar`
5	Top k	heap-based, deterministic top-k: highest score first, ties by lowest row	`avaliar`, `escolher`
6	Zones	minimum and maximum per block and column: search skips whole blocks	`filtrar`, `info`
7	Dictionary	repeated text stored once per file	`dic` type
8	Key index	goes straight to the row of a key, without scanning	in the core; no command of its own
9	Lexicon	Aho-Corasick automaton read from disk; whole word, accent- and case-insensitive	`reconhecer`
10	Live writing	one process reads while another writes, never seeing half a block	in the core; no command of its own
11	Live, deleted and history	writing the same key edits; `_apagado: true` deletes; compaction rewrites only what is current	in the core; no command of its own
12	Permission view	column, row and cell, according to the credential	`ver`, `wjsw-cgi`, SDK
13	Custody	crc32c and SHA-256 (or BLAKE3) per block, chained	`conferir`, `reparar`
14	Partial remote read	with header and footer, downloads only the blocks of the rows read	`wremoto`
15	Similar spelling and sound	a 0-to-1 score from trigrams of the spelling and of a phonetic key, in Brazilian Portuguese, US English, Spanish and Chinese	`filtrar … ~~`
16	Audio signals	pitch, pauses, intensity, jitter, shimmer, HNR, formants, MFCC, spectral centroid, articulation rate and breaths	ABI: `jsonw_sinal_audio`
17	Text signals	intensifiers, hedges, polarity, absolutes, negation, passive voice, pronouns, form and irony cues, in pt, en, es and zh	ABI: `jsonw_sinal_texto`
18	Video into scenes	scene cuts, the measures and audio signals of each stretch and a frame mosaic	ABI: `jsonw_video_*`
19	Per-block and per-tenant columns	local dictionary per block, fixed-width digits, tenant zone and batch comparison	on write; `filtrar`, `calcular`

3.1 Search and ranking (features 1 to 5)

The `avaliar` (evaluate) command scores each row and returns the top k. The request can be a weighted sum over numeric columns, the cosine between a vector column and a query, the product between sparse vectors or a grid lookup. The same request exists in JSON, identical in the tool and the ABI, and every SDK returns the same result, byte for byte [20]. The computation reads only the columns in the request, in parallel. Listing 3 uses five products with a `vf16[4]` vector column and a `mapa<i64,f32>` term column.

```
$ wjsw de-texto
"id:i64,produto:texto,categoria:dic,preco:decimal[2],nota:f64,vet:vf16[4],termos:mapa<i64,f32>"
prod.jsonl prod.jsw
5 linhas -> prod.jsw

# 4 and 5. score by weights (price counts against, rating in favor) and the top 3
$ wjsw avaliar prod.jsw pesos "preco=-1,nota=10" 3
37.1    1       {id:2,produto:"Pão de queijo",categoria:"padaria",preco:8.90d,nota:4.6,…}
35.5    0       {id:1,produto:"Café especial",categoria:"bebida",preco:12.50d,nota:4.8,…}
31.1    2       {id:3,produto:"Chá verde",categoria:"bebida",preco:9.90d,nota:4.1,…}

# 2. semantic search: cosine with the query
$ wjsw avaliar prod.jsw cosseno vet "1,0,0.2,0.1" 2
0.9869597291884717      2       {id:3,produto:"Chá
verde",…,vet:vf16[0.7998046875,0,0.300048828125,0.0999755859375],…}
0.9681576829495052      0       {id:1,produto:"Café especial",…}

# 1. term search: product between sparse vectors (term 12 and term 90)
$ wjsw avaliar prod.jsw esparso termos "12=1,90=0.5" 2
1.2000000476837158      0       {id:1,produto:"Café especial",…,termos:[[12,1.2000000476837158],
[40,0.40000000596046451]]}
1.0     4       {id:5,produto:"Suco de uva",…,termos:[[12,0.30000001192092896],
[90,1.399999976158142]]}

# the same request in JSON, as in the ABI
$ wjsw avaliar prod.jsw '{"pesos":{"preco":-1}}' 2 --json
{"linha":1,"nota":-8.9,"valor":{"id":2,"produto":"Pão de queijo",…}}
{"linha":2,"nota":-9.9,"valor":{"id":3,"produto":"Chá verde",…}}

# 5. the 2 largest values of a column
$ wjsw escolher prod.jsw nota 2
4.9     3       {id:4,produto:"Bolo de milho",…}
4.8     0       {id:1,produto:"Café especial",…}
```

Listing 3. Output: score, row position and the row in jsonW-text, best to worst.

Two details of the output are worth noting. First, the vector stored as `vf16` comes back as `0.7998046875` where the input said `0.8`: half precision holds about three decimal digits. When vector exactness matters, the column should be `vf32`; `vq8` and `vq4` trade more precision for size. Second, term search is a dot product: the IDF weights [22] are stored in each row's vector, and the query brings its own.

Grid (feature 3). A grid is a table of values over axes (for example, shipping cost by distance and weight). The request `{"grade":{"campos":["x","y"],"arquivo":"g.jsw"}}` uses the row's columns as coordinates and returns, as the score, the value interpolated between the neighboring grid points: bilinear with two axes, trilinear with three, and so on. In version 1.3.11, the format of the grid table is not yet in the public documentation, so we do not show its output here.

3.2 File organization (features 6 to 8)

Zones. Each block keeps the minimum and maximum of each column. A filter that a block cannot satisfy makes the reader skip the whole block. In the 100,000-record file of Section 5, with 103 blocks:

```
$ wjsw filtrar com.jsw "id > 99990"
9 de 100000 linhas em 1.7 ms · blocos 103, pulados pela zona 102 · linhas lidas 833
$ wjsw info prod.jsw
…
dicionario 2 textos
zona      preco        890 .. 2400
zona      nota         3.8999999999999999 .. 4.9000000000000004
```

Listing 4. In the 100,000-record file (ids 0 to 99,999), zones skip 102 of 103 blocks ("pulados pela zona"); in the products file of Listing 3, `info` shows the dictionary of the `categoria` column and the zones, with reals in double precision as the tool prints them.

Dictionary. A `dic` column stores each distinct text once per file. In the 100,000 test records, the `tipo` and `fonte` columns add up to 10 distinct texts, and `wjsw info` shows `dicionario 10 textos`.

Key index. The format has an index that goes straight to the row of a key, without scanning the file. In the public 1.3.11 tool it has no command and no ABI function; the visible use of keys is `comparar` (compare), which matches the rows of two files by a column and counts the differences:

```
$ wjsw comparar ontem.jsw hoje.jsw sku
so em a 1 · so em b 1 · mudou 1 · iguais 0 (0.0 ms)
```

Only in a: 1 · only in b: 1 · changed: 1 · equal: 0.

3.3 Lexicon (feature 9)

A term table, with the schema `termo:texto,forma:dic,sinal:dic,peso:f32,idioma:dic,origem:dic`, is compiled, on open, into an Aho-Corasick automaton [21] that finds every term in a text in a single pass. Matching is by whole word, accent- and case-insensitive, and returns the original span and the sum of weights per signal: a text classifier without training, useful for triage and for routing a request to the right specialist.

```
$ wjsw reconhecer termos.jsw "Quero CANCELAR o pedido, ou então me diga quanto custa o orcamento
novo"
3 termo(s), 2 sinal(is)
  intenção de compra soma 2.2  vezes 2
  risco de perda       soma 2  vezes 1
  [6..14] CANCELAR  (termo 0, risco de perda, peso 2)
  [43..55] quanto custa  (termo 3, intenção de compra, peso 1.2)
  [58..67] orcamento  (termo 2, intenção de compra, peso 1)
```

Listing 5. A Portuguese sentence ("I want to CANCEL the order, or tell me how much the new quote costs"). "CANCELAR" matched "cancelar", and "orcamento" matched "orçamento", because matching ignores accents and case. The signals are "purchase intent" and "risk of loss".

3.4 Writing and history (features 10 and 11)

A `.jsw` file is append-only. That gives two guarantees. The first is **live writing**: one process can read the file while another writes it, like a `tail -f` that never sees half a block. The second is **history**: writing the same key again is an edit, writing the key with `_apagado: true` is a deletion, and compaction rewrites only what is current into a new file, leaving the history in the original.

In version 1.3.11, reading during writing, the per-key "current only" view and compaction have no command in `wjsw` and no function in the public ABI. With the public tool, the history shows as it is, row after row:

```
$ wjsw ver dup.jsw
{id:1,nome:"a",_apagado:null}
{id:2,nome:"b",_apagado:null}
{id:1,nome:"a2",_apagado:null}
{id:2,nome:null,_apagado:true}
```

What the public ABI does guarantee on writing is **atomic publication**: the file appears, whole, only on `fechar()` (close). A writer that exits on an error midway calls `abandonar()` (abandon), and the destination stays as it was.

3.5 Permissions, custody and remote access (features 12 to 14)

Permission views (12) and custody (13) are the subject of Section 4. **Partial remote read** (14) uses HTTP `Range` requests [24]: the reader downloads the header and footer, where the schema, the block index and the zones are, and then only the blocks it needs (Figure 4). We measured with `wremoto` 1.2.1 against a local nginx serving the 100,000-record file (the `com.jsw` of Section 5, named `cartoes.jsw`); with `file://` instead of the HTTP address, `wremoto` gives the same numbers, with no server:

```
$ wremoto info http://127.0.0.1:18099/cartoes.jsw
… 100000 linhas em 103 blocos · arquivo 79.1 MB · abrir baixou 8.7 KB (cabecalho + rodape)
$ wremoto ver http://127.0.0.1:18099/cartoes.jsw 50000 1
{id:50000,tipo:"fator de emissão",titulo:"Fator de emissão de CO₂ do combustível nº 50000",…}
baixado no total: 777.8 KB em 5 pedidos (0.98% do arquivo)
$ wremoto filtrar http://127.0.0.1:18099/cartoes.jsw "id > 99990"
…
baixado no total: 667.7 KB em 5 pedidos (0.84% do arquivo)
```

Listing 6. To read one row from the middle of a 79.1 MB file, the reader downloaded 0.98% of it ("baixado no total"); for the filter, 0.84%, because the zones discarded the other blocks before download.

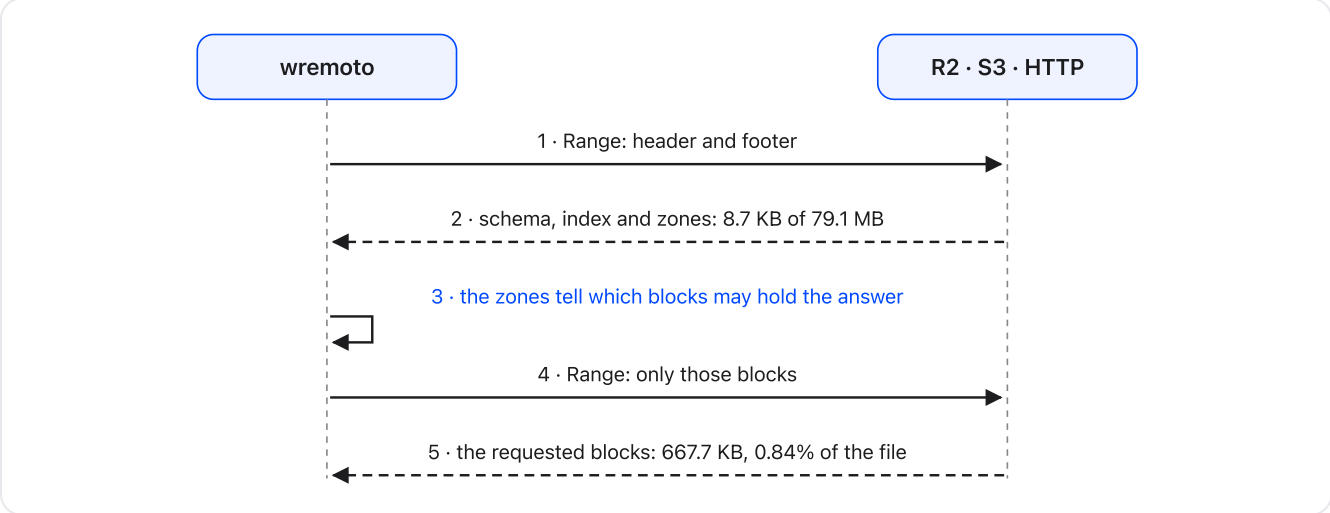


Figure 4. Partial remote read of the 100,000-record file, with the numbers measured in Listing 6. With R2 or S3, each request is signed with AWS SigV4 [25].

3.6 Spelling, signals and per-block columns (features 15 to 19)

Spelling and sound (feature 15). The filter's `~~` operator gives each value a score from 0 to 1: the higher of two Dice measures over trigrams of Unicode characters, one on the normalized spelling (no accents or case) and one on a phonetic key of the requested language. There are keys for Brazilian Portuguese, US English, Spanish and Chinese written in pinyin; for ideographs, the trigrams count character by character, and their sound key, which needs a table of readings, is not in this version yet. The filter uses no index: it scans only the requested column, in parallel (Listing 6B).

```
$ wjsw filtrar marcas.jsw "nome ~~pt Kokakola"
2 de 4 linhas em 0.0 ms · blocos 1, pulados pela zona 0 · linhas lidas 4
{id:1,nome:"Coca-Cola"}
{id:4,nome:"Kokakola Bebidas"}
```

Listing 6B. Similar spelling and sound. "Kokakola" finds "Coca-Cola": the Portuguese key maps the hard "c" and the "k" to the same sound. `~~en`, `~~es` and `~~zh` change the language; `~~pt@0.4` changes the threshold.

Communication signals (features 16 to 18). They measure how someone spoke, wrote or appeared, and decide nothing: the model that reads the row interprets it. In audio (16-bit PCM), pitch comes from normalized autocorrelation, the same method as Praat; jitter and shimmer from the glottal pulses; formants from linear prediction (LPC); and pauses from Otsu's threshold over intensity, which takes the cut from the data itself instead of a fixed number. In text, the language's lexicon (pt, en, es or zh, written by hand) is matched by the automaton of feature 9, and the form counts (caps, punctuation, emoji, laughter) use no dictionary. Chinese, which does not separate words with spaces, is matched ideograph by ideograph. In video, the caller decodes and hands over the frames and the sound; each frame becomes a thumbnail with light, contrast, color and shape, and scene cuts come from the same Otsu threshold over the difference between frames, only when the series has two clear groups. Each measurement comes out as one row of jsonW-text in the declared schema, ready to be written to a `.jsw`.

Per-block and per-tenant columns (feature 19). Three new codecs: a local dictionary for each block, with and without entropy coding, for text that repeats within the block but not across the whole file; and fixed-width digits, for codes made only of numerals, such as an application number. Each block's zone also stores the tenant, and, with a session's view, another customer's block is never opened. The filter and the aggregation compare the dictionary code in batches with vector instructions (SSE2) and evaluate "contains" over the whole block at once. A file written with these codecs does not open in 1.2.x readers. In the same version, the maximum level (`--max`) gained two codecs that store each cell and each vector as the difference from the previous row: the common prefix and suffix and only the middle that changes. The effect on the INPI base is in Tables 5C and 5D.

4 Security and custody

4.1 Custody

Each block of the file stands on its own and carries two checks: a CRC32C [11], fast, for accidental damage, and a cryptographic hash, SHA-256 [12] by default or BLAKE3 [13] as an option. The hashes form a chain that covers the whole file: any byte changed in data, a zone or the dictionary fails `conferir` (verify) (Figure 2).

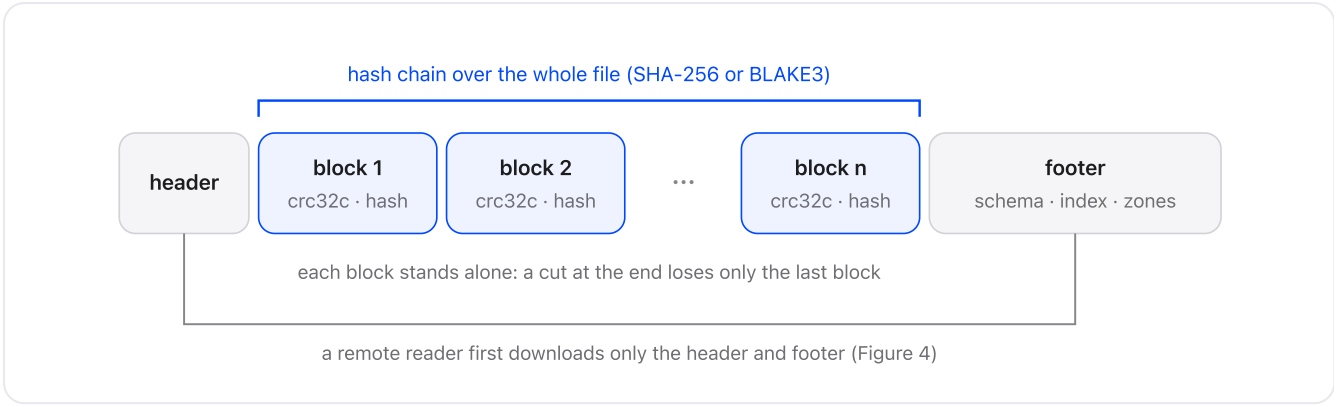


Figure 2. The `.jsw` file seen from outside, at the level described in the public documentation. The internal organization of header, blocks and footer is not published (Section 1.3).

```
$ wjsw conferir vendas.jsw
ok: 2 linhas, 1 blocos, sha256 e crc32c conferidos
# one byte changed in the middle of the file
$ wjsw conferir adulterado.jsw
FALHOU: bloco 0: nao veio ou nao confere
# the last 20 bytes cut off, as in an interrupted write
$ wjsw conferir cortado.jsw
FALHOU: sem rodape: rode wjsw reparar
$ wjsw reparar cortado.jsw && wjsw conferir cortado.jsw
ok
ok: 2 linhas, 1 blocos, sha256 e crc32c conferidos
# BLAKE3 instead of SHA-256
$ wjsw regravar vendas.jsw vendas-b3.jsw --hash blake3 && wjsw conferir vendas-b3.jsw
ok: 2 linhas, 1 blocos, blake3 e crc32c conferidos
```

Listing 7. Custody, tampering and repair, with the real output ("FALHOU" = failed; "sem rodape: rode wjsw reparar" = no footer: run wjsw reparar).

> Custody proves integrity, not authorship. An unkeyed hash can be recomputed by whoever rewrites the whole file. To prove that a file is the same one that left a source, keep the file identifier shown by `wjsw info` somewhere separate, or sign it with a separate tool, and compare.

4.2 Reading hostile files

The reader treats every file as hostile: it checks every index and every bound before use, rejects nesting deeper than 1000 levels and is tested with fuzzing (randomly generated inputs) in the test suite. Quotes, commas, slashes and any byte inside a text are data and never break reading.

4.3 Permission views

In jsonW, permissions are declared in the schema and applied by whoever serves the file, before answering: what the session may not see never leaves the server (Figure 3, Table 3).

Table 3. The layers of the permission view.

Layer	How it is declared	What happens
Column	`+escopo(financeiro)`, `+pessoal`	without the scope, the column comes out null and cannot be used even in a filter
Row	an integer column named `tenant` or `tenant_id`, or marked `+tenant`	only the rows of the credential's tenant; other tenants' blocks are not even opened; `tenant:todos` (all) opens everything; the tenant is an integer: with a text `tenant` column, no row comes out
Cell	both together	the customer's own rows, without the personal-data column

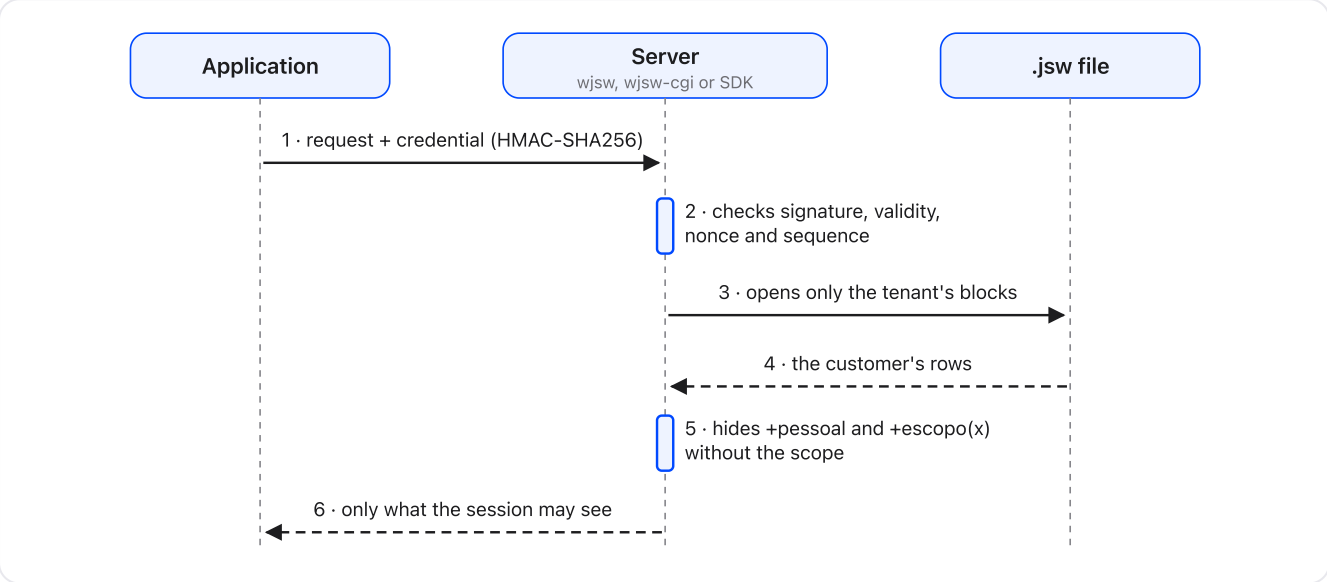


Figure 3. The permission view. Steps 2 to 5 happen inside the server; the file never goes out whole to the requester.

A clinic with two units shows the three layers. Unit 1's front desk sees unit 1's schedule without tax ID and without amount; unit 1's finance team sees the amount, still without tax ID; nobody in unit 1 sees unit 2:

```
$ wjsw de-texto "paciente:texto,cpf:codigo?
+pessoal,valor:dinheiro+escopo(financeiro),data:tempo,tenant:i64" clin.jsonl clin.jsw
# unit 1 front desk
$ WJSW_ESCOPOS="tenant:1" wjsw ver clin.jsw
{paciente:"Ana Souza",cpf:null,valor:null,data:t"2026-11-12T13:00:00.000000Z",tenant:1}
{paciente:"Carla Dias",cpf:null,valor:null,data:t"2026-11-12T15:30:00.000000Z",tenant:1}
# unit 1 finance
$ WJSW_ESCOPOS="financeiro tenant:1" wjsw ver clin.jsw
{paciente:"Ana Souza",cpf:null,valor:180.00 BRL,data:t"2026-11-12T13:00:00.000000Z",tenant:1}
{paciente:"Carla Dias",cpf:null,valor:180.00 BRL,data:t"2026-11-12T15:30:00.000000Z",tenant:1}
```

Listing 8. The same query, two credentials, two answers. The example data is fictitious.

> The view is not encryption. It protects what leaves the server; whoever holds the `.jsw` file holds all the data, and `para-texto` exports everything, because it is the copy for whoever already owns the file. To show data to a user, use `ver`, `wjsw-cgi` or the SDK with `visao()`; to protect the file at rest, use disk or storage encryption.

4.4 Credential

The credential that comes with a request is itself a jsonW value, with issuer, subject, tenant, scopes and validity, signed with HMAC-SHA256 [18]. The signature is checked in constant time, and a nonce with a sequence number prevents a request from being replayed. It travels on top of TLS, not instead of it.

4.5 Post-quantum transport

The row form runs on a wire protocol over QUIC [15] with TLS 1.3 [16] and the hybrid key exchange X25519MLKEM768 [17], which combines the X25519 curve with ML-KEM-768, standardized by NIST in FIPS 203 [14]. The post-quantum argument has two parts. The key exchange uses a mechanism designed to resist a quantum computer. And custody and credentials use only 256-bit hashes and HMAC, which do not depend on factoring or discrete logarithms; to find a preimage or the HMAC key, Grover's algorithm [19] gives a quadratic speedup and leaves about 128 bits of security; to find collisions, the known quantum bound is lower, about 2^{85} operations on SHA-256 with the Brassard, Høyer and Tapp algorithm [35], which needs an enormous quantum memory.

The last line of `wjsw medir` (Section 5) measures the row form: one test record takes 5,157 bytes as JSON and 1,073 bytes as a message, 4.8 times less, in every run.

> The public 1.3.11 package ships neither a wire server nor a wire client.

4.6 Web servers

The package includes `wjsw-cgi`, which runs on Apache, nginx, LiteSpeed, Caddy and IIS [20]. Every `.jsw` request goes through it: a file without marked columns and without a tenant goes out unfiltered, with `Range` and CORS; the others go out filtered, with the view enforced. The scope comes from the server after authentication, never from a header sent by the client, and asking for a hidden column through `campos=` or `onde=` returns 403, so nothing leaks by inference. In version 1.2.0, the sample configuration let the server deliver the raw file when the request had no parameters; 1.2.1 fixed this, and anyone who installed the old configuration should replace it [20].

5 Measurements

This section measures jsonW 1.3.11 against JSONL and states, alongside, what the measurement does not prove. The numbers were redone for this text on November 13, 2026, with the release binary, on the same machine as the November 12 measurement.

5.1 Methodology

- **Tool:** `wjsw medir` (measure), which ships in the public binary, with 100,000 rows: `wjsw medir <folder> 100000`. Binary `bin/linux-x86_64/wjsw` from the public repository, version 1.3.11 (pre-release, SHA-256 `cb119d7473e7d786…`).
- **Machine:** a virtual server with an AMD EPYC 9355P, 8 vCPUs and 18 GB of memory, Ubuntu 26.10, Linux kernel 7.3. The files lived on a `tmpfs` (memory), so the measurement excludes the disk. The machine was not isolated from other loads.
- **Data:** 100,000 synthetic records, shaped like a knowledge-base card: `id:i64,tipo:dic,titulo:texto,corpo:texto,fonte:dic,valor:decimal[4],vetor:vf16[384],tags:lista<texto>`. The texts repeat a template with the row number; `tipo` and `fonte` cycle through 5 values; the vector has 384 components that are multiples of 1/1024, which `vf16` stores without loss.

- **The JSON side:** the same set written as JSONL and read by jsonW's own text reader, which parses each line and converts the schema types. Both files are loaded into memory before the clock starts.
- **Repetition:** 10 consecutive runs. We report the median of each time and, for ratios, the median and range over the 10. The raw outputs are published with this text.
- **Check:** in every run the tool checks that both reads counted 100,000 rows with the same sum.

5.2 Time

Table 4. Time to write and read the 100,000 records. Times: median of 10 runs. Ratio: JSONL ÷ jsonW in each run, with the median and range over the 10. In the 8-thread row, the ratio is against reading JSONL on one thread. The JSONL is written and read by jsonW's own text code.

Operation	JSONL	jsonW	Ratio (range)
Write	15,211 ms	131 ms	114× (88 to 127)
Read all, every field	3,687 ms	78 ms	46× (43 to 51)
Read all, 8 threads	(no parallel version)	9.0 ms	419× (365 to 532)
Read one column (`valor`)	3,687 ms	1.59 ms	2,310× (1,592 to 2,784)
Go to row N, field `titulo`	read from the start	0.09 µs (0.05 to 0.16)	no defined ratio

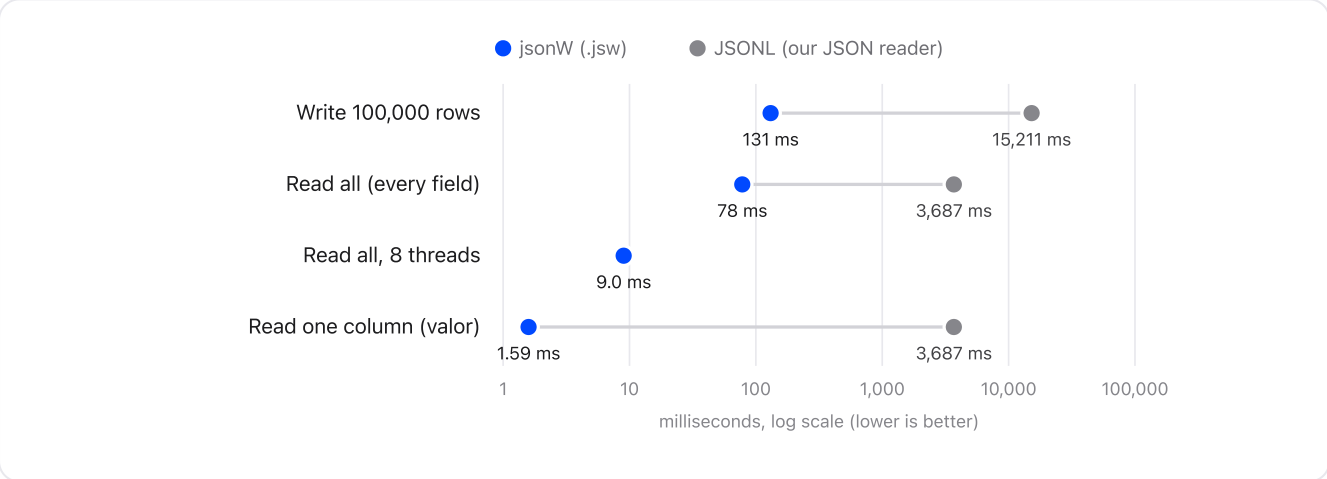


Figure 6. The times of Table 4 on a log scale (median of 10 runs). The horizontal distance between the dots of each row is the ratio.

Figure 6 shows the same times on a log scale. The largest difference is in reading one column: JSONL forces parsing each whole line to extract one field, and the columnar file reads only that column. It is a structural advantage, the same any columnar format has over a row format. For "go to row N", JSONL cannot jump: it must be read from the start up to the row, so the ratio depends on N and we do not report it.

5.3 Size

To compare size against general-purpose compressors, we rewrote the same 100,000 records: an `awk` generator, published with this text, produces the JSONL; `wjsw de-texto` writes the `.jsw`; and the JSONL returned by `wjsw para-texto --json` has 519,264,563 bytes, the same size, at MB scale, that `wjsw medir` reports for its own JSONL (519.3 MB); the `.jsw` has 79,076,300 bytes (79.1 MB in both). The generator's JSONL has 200,000 more bytes, because the decimal value is quoted there; Table 5 uses the

JSONL rewritten by jsonW, and with the generator's file the ratios change in the first decimal place (gzip: 19.7× instead of 19.8×). The JSONL was compressed with gzip 1.14 [32], zstd 1.5.7 [31] and xz 5.8.3 [33] at their usual maximum levels.

Table 5. Size of the 100,000 records, in MB (10^6 bytes), with and without the vector column. Ratio = uncompressed JSONL ÷ the size in that row.

File	With vector	Ratio	Without vector	Ratio
JSONL	519.3	1.0×	38.2	1.0×
`.jsw`, `vf32` vector	155.9	3.3×	n/a	n/a
`.jsw`, `vf16` vector, default level	79.1	6.6×	2.22	17.2×
`.jsw`, `vq8` vector (lossy)	41.1	12.6×	n/a	n/a
JSONL + gzip -9	26.3	19.8×	1.42	26.9×
JSONL + zstd -19	2.67	194.3×	0.63	60.6×
JSONL + xz -9	1.56	332.5×	0.47	81.3×
`.jsw`, `vf16` vector, `--max`	1.36	381.1×	0.0175	2,185×

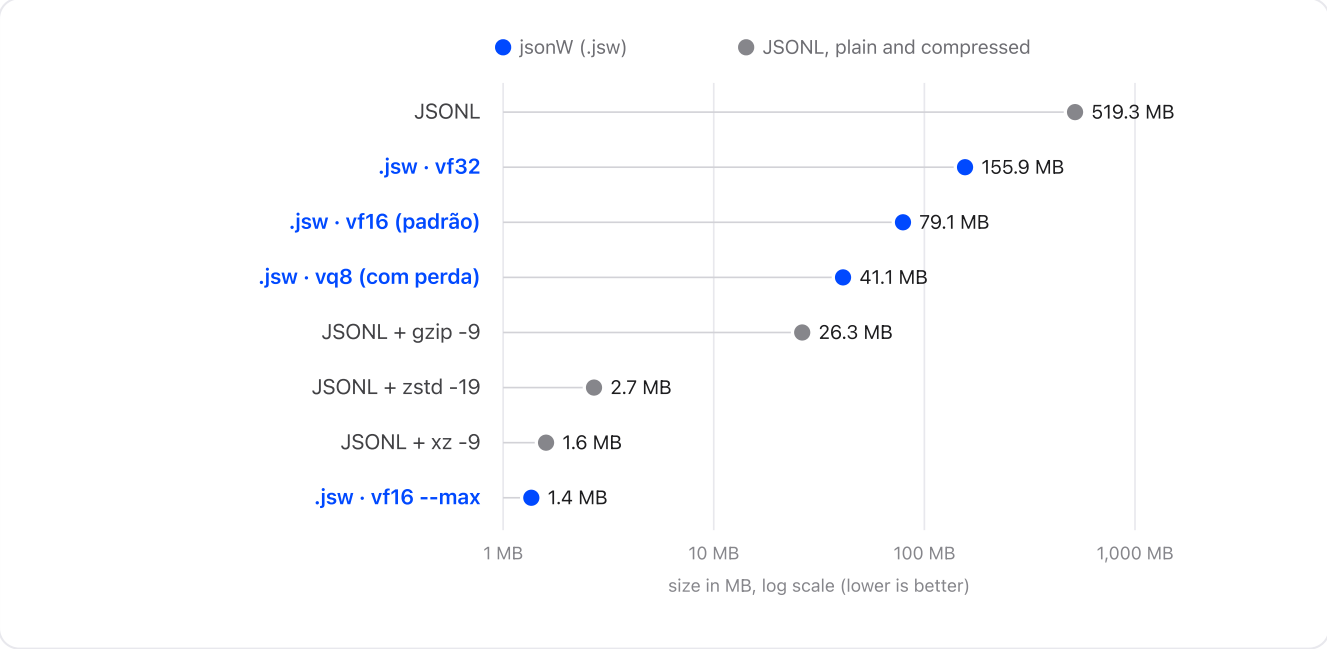


Figure 5. The with-vector sizes of Table 5, on a log scale.

Figure 5 shows the with-vector sizes. Against uncompressed JSONL, which is how JSONL is usually stored and exchanged, the default `.jsw` is 6.6 times smaller, and almost all of its size is the vector: 384 two-byte components in 100,000 rows are 76.8 MB. Against JSONL compressed with zstd or xz, the default `.jsw` is larger. At the maximum level, 1.3.11 came out smaller than both: 1.36 MB with the vector, against 2.67 MB for zstd and 1.56 MB for xz, and 17.5 kB without the vector, against 0.63 and 0.47 MB; in 1.2.1, the same level gave 3.75 and 2.19 MB. The gain comes from two new maximum-level codecs that store each cell and each vector as the difference from the previous row (the common prefix and suffix and the middle that changes), and it is large here because the test data is very repetitive: the texts repeat with a changed number and the vector follows an arithmetic pattern. The round trip is lossless: `para-texto` of the maximum-level file is byte-for-byte identical to that of the default-level file (same SHA-256, with and

without the vector). On real data the gain is much smaller: on the INPI base, the 1.3.11 maximum level came out 16.5% smaller than the 1.2.1 one (Table 5C). Real model embeddings also tend to compress little; we did not measure that here, but if so, the size of a `.jsw` with vectors stays close to the raw size of the vectors.

What a general-purpose compressed file does not give is access. To read one field of zstd-compressed JSONL, everything must be decompressed (0.13 s to 0.24 s to decompress `com-volta.jsonl.zst`, in 3 runs on the same machine, outside the measurement environment) and then the text parsed, while `.jsw` reads the field directly, skips blocks by zones, verifies custody and can be read remotely in parts.

5.4 Real data: CSV, XML, XLSX and Parquet

The measurements above use generated data. To compare against the most common spreadsheet and exchange formats, we used public data: the first one million rows of `MARCAS_DADOS_BIBLIOGRAFICOS.csv`, from the open trademark data of INPI, the Brazilian patent and trademark office (13 columns: application number, dates, presentation, nature, mark name, status). A C program with no libraries read the CSV following RFC 4180 (quotes, commas and line breaks inside a field) and wrote the same rows in four forms: the rewritten CSV; an XML with one element per row and one child per column; an XLSX assembled by the program itself (Office Open XML, inline string cells, compressed with `zip -6`); and the `.jsw`, with a hand-written schema (dates as text; presentation, nature and status as dictionary). Empty fields are absent in XML and jsonW. XLSX accepts at most 1,048,576 rows per sheet, hence one million.

The query was the same in all four: count the registrations in force (`codigo_situacao = IPAS023`). All four gave the same result, 352,248 records, and `wjsw conferir` verified the SHA-256 and CRC32C of every block of the `.jsw`. Each query ran 5 times with the file already cached; the lowest counts. Machine: 8-core server running Linux 7.3, jsonW 1.2.0, November 13, 2026.

Table 5B. One million real INPI rows in four formats. Size in MB (10^6 bytes); query = count registrations in force, in milliseconds.

Format	Size	With zstd -3	Query	How the query was run
XML	502.1	46.0	159	`grep -c` over the whole file
CSV	141.1	36.1	97	`grep -c` over the whole file
XLSX	74.3	n/a	9,364	`unzip -p` of the sheet + `grep`
`.jsw`	51.2	n/a	119	`wjsw calcular --onde`, that column only

In size, the `.jsw` was 2.8 times smaller than CSV, 1.5 times smaller than XLSX and 9.8 times smaller than XML. CSV and XML compressed with zstd are smaller than the `.jsw` (36.1 and 46.0 MB against 51.2 MB), but, as in Section 5.3, they must be fully decompressed before any read. In the query, the `.jsw` was 79 times faster than XLSX, which must be unzipped on every read. Against CSV and XML, `grep` was faster or similar: it only searches for a piece of text across the whole file, without splitting columns or types, and would give a wrong count if the same code appeared in another column. In this data set it does not, and the counts matched. A query with the same guarantee on CSV requires reading and splitting every row, which `awk` did in 1,597 ms in the first run of this measurement, without handling quotes.

What none of the three formats provides by itself: a typed schema (date, decimal, money, vector), reading one column without reading the others, term and meaning search, permission views, history of deleted rows and per-block custody. XLSX has cell types and multiple sheets, but caps the sheet size and must be

unzipped to be read.

The whole base. Table 5B uses one million rows of one file. To see the format with everything, we converted the seven files of the INPI open trademark data [38], downloaded on 12 November 2026, without cutting anything: 55,278,212 rows and 12.53 GB of CSV, with every column and the complete dispatch history of each application (24,978,976 rows in `MARCAS_DESPACHOS.csv` alone). Each file got a schema written by hand. Numbers, dates and codes kept their own type; text kept the bytes as they came. The converter, in C, checks each typed value on the spot: the number and the date must turn back into the same CSV text, or the conversion stops. This caught two real cases in the source, which were kept as they came: a priority date in the year 1018 (a typo in the original) and one row of the Nice classification without an application number, out of the 5,333,893 in that file.

The Parquet [3] was written by DuckDB 2.0.0 [39] with the same types and zstd compression. The XML follows the layout of Table 5B. XLSX was left out: six of the seven files exceed the limit of 1,048,576 rows per sheet, and the dispatch file has almost 24 times that limit. The eight forms gave the same counts in the three queries, with one exception, stated below. `wjsw conferir` checked the SHA-256 and the CRC32C of every block of the seven `.jsw` files. Each query ran 5 times, with the file already cached, and the lowest counts. Writing the jsonW used one core. DuckDB, zstd and xz used all eight cores, and so did `wjsw calcular`. Machine: the same as Table 5B, 31 GB of memory, 13 November 2026.

Table 5C. The whole INPI trademark base (seven files, 55.3 million rows) in each format. Size in MB (10^6 bytes), summed over the seven files; write = time to produce the format from the CSV, in seconds.

Format	Size	Times smaller than CSV	Write	Reads one column without opening the rest?
XML	24,804.7	0.5	n/a	no
CSV (original)	12,527.4	1.0	n/a	no
`.jsw` , 1.2.1	2,804.8	4.5	176.7	yes
`.jsw` , 1.3.11	2,579.2	4.9	n/a	yes
`.jsw` , 1.2.1, `--max`	2,321.8	5.4	184.9	yes
XML + zstd -3	2,210.0	5.7	21.4	no, decompresses everything
`.jsw` , 1.3.11, `--max`	1,938.1	6.5	n/a	yes
CSV + zstd -3	1,908.9	6.6	13.2	no, decompresses everything
Parquet (zstd)	1,665.8	7.5	16.6	yes
CSV + xz -6	1,161.0	10.8	544.5	no, decompresses everything

Table 5D. Three column queries on the whole base, in milliseconds (lowest of 5), measured for the fifth time on 2026-11-13 at 06:21 UTC, with the `wjsw` 1.3.11 release and 1.2.1, at the same time. In force: count `codigo_situacao = IPAS023` in 6.7 million applications. History: all dispatches of one application (`numero_inpi = 818602856`) in 25.0 million. Class 35: count `classe_nice = 35` in 5.3 million. In parentheses, DuckDB with one core. XML comes from the first run, because the file was deleted after it (it took 24.8 GB).

Format and reader	In force	History	Class 35
Right answer	2,343,607	10	1,076,928
Parquet, DuckDB	54 (76)	129 (521)	41 (60)
`.jsw --max` 1.3.11, `wjsw calcular --onde`	43	207	142
`.jsw` 1.3.11, `wjsw calcular --onde`	44	254	152
`.jsw --max` 1.2.1, `wjsw calcular --onde`	49	310	145
`.jsw` 1.2.1, `wjsw calcular --onde`	54	353	198
CSV, DuckDB reading the CSV	375 (1,447)	1,205 (4,523)	592 (1,979)
CSV, `grep -c`	519	6,690	1,278 (wrong: 1,076,929)
CSV + zstd, `zstd -dc | grep -c`	798	3,129	2,786 (wrong)
CSV + xz, `xz -dc | grep -c`	1,050	3,025	2,105 (wrong)
CSV + xz, `xz -dc | grep -c`	971	2,628	1,950 (wrong)
XML, `grep -c` (first run)	3,017	9,925	3,430

What the whole base shows. In size, the 1.3.11 `.jsw` was 4.9 times smaller than the CSV at the default level and 6.5 times at the maximum, and 12.8 times smaller than the XML; 1.2.1 was 4.5 and 5.4 times smaller than the CSV, and 1.3.11 at the maximum came out 16.5% smaller than 1.2.1. In the queries, with eight cores on both sides, 1.3.11 counted the records in force in 43 ms at the maximum level and 44 ms at the default, ahead of the 54 ms of Parquet read by DuckDB; with a single core, DuckDB on Parquet took 76 ms on that count and 521 ms on the history. On all three queries, the `.jsw` was faster than DuckDB reading the CSV itself, than `grep` , than the compressed CSV and than the XML. On the history of one application and on class 35, the `.jsw` took 207 and 142 ms, and Parquet with eight cores, 129 and 41 ms: this is the next performance step in Section 8. In size, Parquet (1,665.8 MB) and the CSV compressed with xz (1,161.0 MB) take less than the `.jsw` at the maximum (1,938.1 MB), and the compressed CSV must be fully decompressed before any read; with zstd, the CSV came out practically equal to the `.jsw` at the maximum (1,908.9 MB). Writing the 1.2.1 `.jsw` , on one core, took 177 s; Parquet, with eight, 17 s. The 1.3.11 files were rewritten from the 1.2.1 `.jsw` (`wjsw regravar`), so Table 5C does not give their write time.

The runs and the fixes. Table 5D was measured five times, always with Parquet and CSV at the same time. With `wjsw` 1.2.0, the `.jsw` took 677, 1,494 and 1,114 ms on the three queries, 10 to 31 times the time of Parquet: the filter read every column of each row. Fixed to read only the column asked about, it took 82, 311 and 209 ms. In the third run, evaluation became per block, comparing the dictionary code and the integers directly in the column and building the row only when it passes: 48, 344 and 186 ms. In the fourth, with 1.3.11 still in preparation, the history went from 355 to 279 ms. In the fifth, with the release binary, which compares the dictionary in batches with vector instructions and evaluates "contains" per block, 1.3.11 took 44, 254 and 152 ms at the default level and 43, 207 and 142 ms at the maximum, against 54, 353 and 198 ms for 1.2.1 at the same time. The history looks up the application number, a code made only of numerals that 1.3.11 stores at fixed width (in DADOS_BIBLIOGRAFICOS, that column alone went from

53.0 to 25.7 MB). The machine was not isolated, and the same binary varied by up to 23% from one run to the next (Parquet, from 41 to 54 ms on the in-force count); so the comparison holds within the same run. A file written with the new codecs does not open in 1.2.x readers. The `grep` times on the CSV varied even more (from 2,102 to 6,690 ms on the history), because the 12.5 GB of CSV do not all fit in the cache alongside the other files, and in the fifth run the rewrites just before pushed the CSV out of the cache; the number from the same run is the one that counts.

The exception in the counts is class 35. `grep` on the CSV, plain or compressed, counted 1,076,929, one more than the right answer: the text `,35,` also appears in another column of one row. It is the risk described next to Table 5B, which did not happen in that set and happened here. Every reader that keeps the columns apart (jsonW, Parquet, DuckDB reading the CSV, and XML with the column name in the markup) gave 1,076,928.

Of this 1:1 conversion, the only feature beyond type and column that went in was custody: every block of the seven files has CRC32C and SHA-256, checked above. The conversion did not mark permissions or build search, to stay equal to the CSV. The format allows, in the same file, hiding the holders' CPF and CNPJ from whoever lacks the scope (Section 4.3), reading only part of it remotely, and searching by terms and by meaning (Section 3). Parquet has an optional per-page checksum [36] and per-column encryption [5], but not the hash chain of the whole file nor the scoped view; CSV and XML bring none of this.

5.5 Limitations of the measurement

- **The JSON reader is ours.** The JSON side was written and read by jsonW's own text code, which validates and converts types; it reads the JSONL at about 140 MB/s. State-of-the-art JSON readers such as simdjson [23] exceed 1 GB/s. With one of them, the "write" and "read all" rows of Table 4 would shrink considerably. As an estimate, not a measurement: at 2 GB/s, reading the 519 MB would take about 260 ms, and the "read one column" ratio would be in the hundreds, not the thousands.
- **Synthetic, repetitive data,** as stated in Section 5.3. The size ratio depends heavily on the data, and even more at the maximum level, which stores the difference between consecutive rows: measure with yours.
- **One machine, one system, memory instead of disk.** Absolute times change with the processor, storage and load; over the 10 runs, the largest ratio of each operation was 19% to 75% above the smallest; reading one column varies most, because jsonW's time there is 1.6 ms.
- **Parquet measured only with INPI data and one reader.** The performance comparison with Parquet is in Tables 5C and 5D, with DuckDB 2.0.0 as writer and reader. Another reader, another compression or another row group size change the numbers. Arrow, MessagePack, CBOR and Protobuf still have no performance measurement in this version of the text; Section 6 compares features, not speed.

5.6 Numbers published before

Until November 12, 2026, the jsonW site and repository cited "4.9 times smaller than JSONL" and "3,281 times faster to read a field", from the first measurement, on November 6, with version 1.0 on another machine. Text 1.0 of this paper, of November 12, redid the measurement with 1.2.1: 6.6 times smaller and about 2,500 times faster to read one column, with the maximum level at 3.75 MB with the vector and 2.19 MB without. Redone for this text with 1.3.11, on the same machine, it gives 6.6 times smaller and about 2,300 times faster to read one column (the difference is within the run-to-run variation), and the maximum level drops to 1.36 MB and 17.5 kB. The numbers of this text count, with the date, the machine and the version alongside.

5.7 How to reproduce

```
git clone https://github.com/developer-wortic/jsonw && cd jsonw
export JSONW_SEM_PING=1                      # no install notice
W=bin/linux-x86_64/wjsw                       # or bin/macos-arm64/wjsw
for i in $(seq 1 10); do $W medir /tmp 100000; done

# Table 5: the same set as JSONL, the .jsw and the compressors
curl -O https://jsonw.org/paper/gerar-cartoes.awk
LC_ALL=C awk -v N=100000 -v VET=1 -f gerar-cartoes.awk > com.jsonl
$W de-texto
"id:i64,tipo:dic,titulo:texto,corpo:texto,fonte:dic,valor:decimal[4],vetor:vf16[384],tags:lista<texto
>" com.jsonl com.jsw
$W para-texto com.jsw --json > volta.jsonl
gzip -9 -k volta.jsonl; zstd -19 volta.jsonl; xz -9 -k volta.jsonl; ls -l com.jsw volta.jsonl*
```

Listing 9. This text's measurements on your machine. `VET=0` and the schema without the `vetor` column give the "without vector" column.

6 Comparison with other formats

Table 6 compares what each format does by itself, without libraries on top. The mature columnar formats, Parquet and Arrow, already have much of the access advantage measured in Section 5, and several things jsonW does not have yet.

Table 6. What each format brings in the format itself. Sources in the references shown in each column.

	JSON · JSONL [1,2]	MessagePack · CBOR [7,8]	Protobuf [9]	Parquet [3,4,5]	Arrow [6]	jsonW [20]
Form	text; one line per record in JSONL	binary, schemaless	binary, compiled schema (`.proto`)	columnar file	in-memory columns; IPC file and stream	binary row, columnar file and text
Schema	no (JSON Schema separately)	no	yes, outside the message	yes, in the file	yes	yes, one line, in the file
Read one field without the rest	no	no	no	yes, by column	yes, direct access	yes, by row and field
Exact decimal	no	CBOR: yes (decimal fraction, tag 4)	not native	yes (DECIMAL)	yes (decimal128 and 256) [37]	yes
Money and measure	no	no	money through a common message (`google.type.Money`) [10]	no	no	yes, with currency and unit
Dense vector	list of numbers as text	list; CBOR has typed arrays [34]	repeated field	list or fixed-size bytes	fixed-size list	f32, f16, 8 and 4 bits, bits; tensor
Permissions	no	no	no	column encryption with different keys [5]	no	column, row and cell view, applied by whoever serves; no encryption
Integrity	no	no	no	optional per-page CRC [36]; authenticated encryption	not in the format	crc32c and hash per block, chained
Search and ranking in the format	no	no	no	row-group and page statistics to skip data	no	zones, cosine, sparse, grid, top-k, lexicon
Partial remote read	no	no	no	yes (footer and Range)	yes, in the file format	yes
Specification	open (RFC)	open (CBOR: RFC)	open	open (Apache)	open (Apache)	usage, text and ABI public; internal binary layout
Implementations	many, independent	many, independent	several	several, independent	several, independent	one (C core), with

	JSON · JSONL [1,2]	MessagePack · CBOR [7,8]	Protobuf [9]	Parquet [3,4,5]	Arrow [6]	jsonW [20]
						SDKs in 10 languages

Where each format fits. JSON is universal and readable anywhere. MessagePack, CBOR and Protobuf are mature and implemented in dozens of languages, and Protobuf brings mature schema evolution rules. Parquet and Arrow are open standards read by almost every analytics engine, and Parquet has column encryption for the file at rest. jsonW works with all of them: every JSON is already jsonW-text, `wcon` brings database tables in and back with the right types, and the file at rest can be encrypted by the disk or the storage.

What jsonW brings together. jsonW's difference is not one isolated feature, but having, in the same model, the message form, the columnar file form and a JSON-compatible text form; business types checked on write; column, row and cell permissions applied on read; and ranking, search and classification operations inside the file. The cost of that choice is being, for now, a single implementation with a closed core (Section 8).

7 Uses

7.1 AI models and semantic search

A knowledge base fits in one `.jsw` with the text, metadata and vector of each passage; semantic search is an `avaliar … cosseno` that reads only the vector column, and the lexicon and sparse vectors cover term search. With top k and the lexicon, the same file serves as a router that picks the expert: the request text is classified by the lexicon, and the ranking chooses.

7.2 Company data

Registries such as Brazil's company (CNPJ) and postal code (CEP) records are where JSON loses the most: the leading zero disappears when the code becomes a number, share capital becomes a double and the date loses its time zone. In jsonW, each thing has its type:

```
cnpj:codigo,razao_social:texto,uf:dic,municipio:dic,cep:codigo,capital_social:dinheiro,abertura:tempo
,cnae:dic,tenant:i64
```

Listing 10. A company registry schema: byte-exact codes, repeated text in a dictionary, money with currency and a tenant to separate customers.

7.3 Databases

`wcon`, the package's database connector, takes data from PostgreSQL, MySQL and MariaDB, SQLite, Redis and Valkey and MongoDB to a `.jsw` and back, with the schema taken from each column's type: `numeric` becomes an exact decimal, `bigint` stays a 64-bit integer and pgvector's `vector` and `halfvec` [30] become `vf32` and `vf16`. Values always travel as data, length-prefixed, never spliced into a command; the return trip runs in a single transaction. The round trip was checked with PostgreSQL 14, 16 and 18 (same md5 on both sides), MariaDB 11.8 (hex of each column), SQLite, Redis 7 and MongoDB 7, under hostile load: injection, quotes, `\0`, accents, emoji and right-to-left text [20].

7.4 Web servers

With `wjsw-cgi` (Section 4.6), a site serves a `.jsw` as JSON, NDJSON or the file itself, with filtering, field selection and limits in the address, always with the session's view.

7.5 Storage on R2 and S3

A `.jsw` can live in a Cloudflare R2 bucket or any S3 and be queried there, without downloading the whole file (Figure 4). Since R2 does not charge for egress [26], querying a piece of a large file costs the piece read plus the operations, which R2 charges per request (Listing 6 makes 5 requests).

8 Limitations and next steps

- **One implementation, closed core.** The binary layout is not published; reading and writing `.jsw` depends on `libjsonw`. The stable ABI and the SDKs reduce the risk, but do not replace an open specification with independent implementations.
- **No independent audit.** The evidence cited (fuzzing, hostile loads, byte-for-byte comparison across SDKs) is ours.
- **Platforms.** There are binaries for macOS arm64, Linux x86_64 (glibc 2.34 or newer) and Windows x86_64. There are none yet for macOS Intel, Linux arm64 or Linux with musl. On Windows, `wcon` and `wremoto` run through WSL.
- **Features without a public command.** Key index, live writing, the "current only" view and compaction exist in the core, but have no command in `wjsw` and no function in ABI 1.3.11; the grid table format is not documented.
- **Wire not in the package.** The post-quantum wire protocol does not ship in the public 1.3.11 package.
- **Portuguese words.** Types and commands use Portuguese words (`texto` , `dinheiro` , `medida` , `ver` , `conferir`), which calls for a glossary for readers who do not read Portuguese.
- **Databases.** MySQL 8 works only with `mysql_native_password` ; the full first authentication of `caching_sha2_password` does not exist yet.
- **Measurements.** The limitations in Section 5.5.
- **Column queries: the next step.** On the whole INPI base (Table 5D), `wjsw calcular` 1.3.11 counts the records in force in 43 ms, ahead of DuckDB reading Parquet, and since the first run (1.2.0) that count went from 677 to 43 ms. The next step is to bring the same block-wise, batched reading to the history of one application and to class 35, where the `.jsw` took 207 and 142 ms and DuckDB with eight cores 129 and 41 ms. In size, the next codecs (delta and bit packing for integers and dates) target the 1,938.1 MB of the maximum level.

Next steps for this text and the package: binaries for macOS Intel, Linux arm64 and musl; a query that checks only each block's CRC32C, leaving SHA-256 to `conferir` (Parquet checks no hash on read, and part of the remaining gap comes from there); the windowed dictionary codec, for columns such as the Nice class; delta encoding with bit packing for integers and dates, measured again against Tables 5C and 5D; and a performance comparison with Arrow on the same machine.

9 Availability and license

The tool, library, ABI, SDKs, web server configurations and documentation are at github.com/developer-wortic/jsonw [20], and the Node.js SDK is on npm as `@worticteCnologia/jsonw`. Use is governed by the jsonW Sustainable Use License (in Portuguese), which prevails over this summary: jsonW is free to use on as many servers and apps as needed, including commercial products, as long as it is a component and not the service being sold; without a written agreement with Wortic, nobody may sell, host or offer to third parties jsonW or a service whose main function is jsonW's (storing, converting, serving or transporting data in jsonW), paid or free; the libraries and tools distributed as binaries may not be reverse engineered, except as the law allows; and the internal file layout, the codecs, the wire protocol and the core belong to Wortic, so an independent implementation of them also requires a written agreement. The public documentation may be read, quoted and freely used to build programs that use jsonW.

For transparency: the first time it runs on each machine, the `wjsw` tool sends only the version and platform (for example `v=1.3.11&so=linux-x86_64`) so we can count installations, and does not repeat afterwards. The IP reaches the server with the connection but is not kept, nor any name, path or data: the server records only the day, platform and version. `JSONW_SEM_PING=1` or the file `~/.jsonw/sem-ping` turn the notice off before the first run. The library and the SDKs never access the network.

References

[1] T. Bray (ed.). *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259, IETF, 2017. rfc-editor.org/rfc/rfc8259

[2] *JSON Lines*. jsonlines.org

[3] Apache Software Foundation. *Apache Parquet: File Format*. parquet.apache.org/docs/file-format

[4] Apache Software Foundation. *Parquet Logical Type Definitions*. github.com/apache/parquet-format/blob/master/LogicalTypes.md

[5] Apache Software Foundation. *Parquet Modular Encryption*. parquet.apache.org/docs/file-format/data-pages/encryption

[6] Apache Software Foundation. *Arrow Columnar Format*. arrow.apache.org/docs/format/Columnar.html

[7] *MessagePack specification*. github.com/msgpack/msgpack/blob/master/spec.md

[8] C. Bormann, P. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 8949, IETF, 2020. rfc-editor.org/rfc/rfc8949

[9] Google. *Protocol Buffers: Encoding*. protobuf.dev/programming-guides/encoding

[10] Google. *google/type/money.proto*. github.com/googleapis/googleapis/blob/master/google/type/money.proto

[11] J. Satran et al. *Internet Small Computer Systems Interface (iSCSI)*, which defines CRC32C (Castagnoli). RFC 3720, IETF, 2004. rfc-editor.org/rfc/rfc3720

[12] NIST. *Secure Hash Standard (SHS)*. FIPS 180-4, 2015. csrc.nist.gov/pubs/fips/180-4/upd1/final

[13] J. O'Connor, J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn. *BLAKE3: one function, fast everywhere*. 2020. github.com/BLAKE3-team/BLAKE3-specs

[14] NIST. *Module-Lattice-Based Key-Encapsulation Mechanism Standard*. FIPS 203, 2024. csrc.nist.gov/pubs/fips/203/final

[15] J. Iyengar, M. Thomson (eds.). *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000, IETF, 2021. rfc-editor.org/rfc/rfc9000

[16] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446, IETF, 2018. rfc-editor.org/rfc/rfc8446

[17] K. Kwiatkowski, P. Kampanakis, B. E. Westerbaan, D. Stebila. *Post-Quantum Traditional (PQ/T) Hybrid Key Agreement Mechanisms for TLS 1.3*. RFC 10024, IETF, 2026. rfc-editor.org/rfc/rfc10024

[18] H. Krawczyk, M. Bellare, R. Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104, IETF, 1997. rfc-editor.org/rfc/rfc2104

[19] L. K. Grover. *A fast quantum mechanical algorithm for database search*. STOC 1996. arxiv.org/abs/quant-ph/9605043

[20] Wortic Tecnologia. *jsonW (json Wondrous) 1.3.11: public repository, documentation (docs/JSONW.md), ABI (sdk/jsonw.h) and binaries*. github.com/developer-wortic/jsonw

[21] A. V. Aho, M. J. Corasick. *Efficient string matching: an aid to bibliographic search*. Communications of the ACM 18(6), 1975. doi.org/10.1145/360825.360855

[22] K. Spärck Jones. *A statistical interpretation of term specificity and its application in retrieval*. Journal of Documentation 28(1), 1972. doi.org/10.1108/eb026526

[23] G. Langdale, D. Lemire. *Parsing Gigabytes of JSON per Second*. The VLDB Journal 28, 2019. arxiv.org/abs/1902.08318

[24] R. Fielding, M. Nottingham, J. Reschke (eds.). *HTTP Semantics*, section 14 (Range Requests). RFC 9110, IETF, 2022. rfc-editor.org/rfc/rfc9110

[25] Amazon Web Services. *AWS Signature Version 4 for API requests*. docs.aws.amazon.com/IAM/latest/UserGuide/reference_sigv.html

[26] Cloudflare. *R2 pricing*. developers.cloudflare.com/r2/pricing

[27] ISO. *ISO 4217: Currency codes*. iso.org/iso-4217-currency-codes.html

[28] IEEE. *IEEE Standard for Floating-Point Arithmetic*. IEEE 754-2019. doi.org/10.1109/IEEESTD.2019.8766229

[29] Unicode Consortium. *Unicode Normalization Forms*. UAX #15. unicode.org/reports/tr15

[30] *pgvector: open-source vector similarity search for Postgres*. github.com/pgvector/pgvector

[31] Y. Collet, M. Kucherawy (eds.). *Zstandard Compression and the 'application/zstd' Media Type*. RFC 8878, IETF, 2021. rfc-editor.org/rfc/rfc8878

[32] P. Deutsch. *GZIP file format specification version 4.3*. RFC 1952, IETF, 1996. rfc-editor.org/rfc/rfc1952

[33] *XZ Utils*. tukaani.org/xz

[34] C. Bormann (ed.). *Concise Binary Object Representation (CBOR) Tags for Typed Arrays*. RFC 8746, IETF, 2020. rfc-editor.org/rfc/rfc8746

[35] G. Brassard, P. Høyer, A. Tapp. *Quantum Algorithm for the Collision Problem*. 1997. arxiv.org/abs/quant-ph/9705002

[36] Apache Software Foundation. *parquet.thrift* (the `crc` field of `PageHeader`). github.com/apache/parquet-format/blob/master/src/main/thrift/parquet.thrift

[37] Apache Software Foundation. *Arrow Schema.fbs* (Decimal type, 128- and 256-bit widths). github.com/apache/arrow/blob/main/format/Schema.fbs

[38] Instituto Nacional da Propriedade Industrial (INPI, Brazil). *Open trademark data* (`marcas.zip`, seven CSV files), downloaded on 2026-11-12. dadosabertos.inpi.gov.br/download/marcas/marcas.zip

[39] DuckDB Foundation. *DuckDB 2.0.0*, CSV and Parquet reading. duckdb.org/docs

How to cite

Wortic Tecnologia. *jsonW: one data model, three lossless forms and permissions inside the format*. Text 1.1, on jsonW 1.3.11. November 13, 2026. Available at: https://jsonw.org/paper/en.

```
@techreport{jsonw2026en,
  title       = {jsonW: one data model, three lossless forms
                 and permissions inside the format},
  author      = {{Wortic Tecnologia}},
  institution = {Wortic Tecnologia},
  year        = {2026},
  month       = nov,
  note        = {Text 1.1, on jsonW 1.3.11},
  url         = {https://jsonw.org/paper/en}
}
```

Text 1.1, published 2026-11-13, on jsonW 1.3.11 (text 1.0, of 2026-11-12, covered 1.2.1). jsonW is json Wondrous; the C core is maintained by Wortic Tecnologia. Corrections and third-party measurements are welcome through the public repository.