The VLDB Journal manuscript No.
(will be inserted by the editor)

Parsing Streams of JSON Documents at Over a Hundred Gigabytes per Second on CPUs

Jaël Champagne Gareau · Daniel Lemire

Abstract Large volumes of data are shipped as streams of JSON documents, one per line, and ingesting them is often a bottleneck for analytics. We present a parallel parser for such streams: workers claim byte ranges from one atomic counter and snap them to delimiters that cannot occur inside a value, so a document is never split and no lock is taken on the hot path. Because the parser is on-demand, it materializes only the fields a query names, which fuses query evaluation into the parse. We evaluate the design on five Intel and AMD servers against Pison, a state-of-the-art structural-index parser, and against the tree-building parsers in common use, verifying identical results throughout. With dynamic assignment the best rate per machine ranges from 105 GiB/s to 229 GiB/s (up to 246 GiB/s with static assignment), several times the throughput that graphics processing unit (GPU) parsers report on the same datasets. Against Pison we are 4.3 to 17.0 times faster end to end, because its index requires a padded copy that its published timings exclude. Against yyjson, the fastest tree-building parser, we retire fewer instructions per byte on every dataset. We also compare with three data systems—Apache Arrow, DuckDB and Polars—and with cuJSON on its own end-to-end workload. Our parser and benchmark are open source.

Keywords JSON · multicore parallelism · SIMD · performance counters

J. Champagne Gareau
Université du Québec (TELUQ), Montréal, Quebec, H2S 3L5, Canada
E-mail: champagne_gareau.jael@univ.teluq.ca

D. Lemire
Université du Québec (TELUQ), Montréal, Quebec, H2S 3L5, Canada
E-mail: daniel.lemire@teluq.ca
Corresponding author

1 Introduction

JSON [3] is a lightweight, text-based format that has become the standard for data exchange in web applications, APIs, and configuration files. Its syntax is effectively a subset of JavaScript. It contains four primitive types (string, number, Boolean, null) and two composite types (arrays and objects). An object is a series of key–value pairs enclosed in braces, where keys are strings (e.g. `{"name":"Jack","age":22}`); an array is a comma-separated list of values between square brackets (e.g. `[1,"abc",null]`).

Systems often exchange more than a single JSON document. Logging pipelines, high-throughput web analytics, machine-learning feature stores, and scientific data archives all process long *streams* of documents. The common encoding places one document per line, a format variously called Line-Delimited JSON (LD-JSON), Newline-Delimited JSON (NDJSON), or JSON Lines (JSONL). Variants exist: JSON Text Sequences [25] prefix each document with a record-separator character (`0x1E`), and some systems separate documents with commas. Separators are not strictly necessary—concatenating objects or arrays (`{"a":1}{"b":2}`) is unambiguous, although concatenated scalars are not—but a delimiter makes the stream splittable into self-contained documents.

Parsing such streams is embarrassingly parallel in principle: the documents are independent. In practice, however, two obstacles stand in the way. First, the parser must find document boundaries without parsing, or boundary-finding becomes the bottleneck. Second, the per-document work must not require shared mutable state. We present a parallel stream parser that resolves both issues, built on the simdjson library [13, 11]. Each worker claims a byte range from a shared atomic counter—an integer that any thread can incre-

Table 1 Best throughput reported for GPU JSON parsers on the Pison datasets (and, for cuJSON's one end-to-end test, on a GitHub Archive file), next to ours. The GPU figures are computed from the published times and dataset sizes and converted to GiB/s; ranges are over datasets. Our results are from Tables 7 and 11, with dynamic assignment except on the GitHub Archive file, where Table 11 uses static assignment

System	Hardware	Input	Timed	GiB/s
GpJSON	NVIDIA A100 (40 GB)	tweets as JSON lines, 0.8 to 12.8 GB; one filter query	index, query, results to host	≈ 4.3[a]
cuJSON	NVIDIA A100 (80 GB)	six datasets, each a single JSON document	parse, with both transfers; no query	8.5–15.5
cuJSON	NVIDIA A100 (80 GB)	six datasets as JSON lines, merged into one array	GPU kernels only; no transfers, no query	26–35
cuJSON	NVIDIA A100 (80 GB)	GitHub Archive events as JSON lines, 1.37 GB; one filter query	end to end: transfers, parse, and query	≈ 4.0[b]
ours	Intel Xeon Gold 6548N, 2 × 32 cores	six datasets as JSON lines, in memory	parse and query	65–105
ours	Intel Xeon Gold 6548N, 2 × 32 cores	GitHub Archive events as JSON lines, 1.37 GB, in memory; cuJSON's filter query	parse and query	67.8
ours	AMD EPYC 9R45 (Zen 5), 96 cores	six datasets as JSON lines, in memory	parse and query	142–229

[a]GpJSON's Figure 7. [b]323 ms for 1.37 GB in cuJSON's Figure 18.

ment without locking—and snaps both ends to a document boundary with at most two character searches (`memchr`), one per end. Each worker owns its own parser and output vector, so the hot path is lock-free. The underlying parser is *on-demand*—it materializes only the values the application asks for—so parsing and query evaluation are fused, and the fields that the query does not name are never decoded.

Graphics processors (GPUs) are a natural alternative for such a parallel task. Two recent GPU parsers, GpJSON [23] and cuJSON [6], are evaluated on the datasets we use, and both report gains over simdjson and Pison. Table 1 collects the throughput they report, converted to gibibytes per second, next to ours. The best GPU figure, cuJSON on an NVIDIA A100, is about 35 GiB/s counting GPU kernel time alone, and at most 15.5 GiB/s once the input is copied to the GPU and the result back; neither includes the query. In cuJSON, JSON lines are not parsed as separate documents: the lines are merged into a single array, which is parsed as one document. Its one end-to-end measurement, a filter query over a GitHub Archive file with every transfer included, runs at about 4 GiB/s. With the data in memory and the query included, our parser reaches 65 to 105 GiB/s on a dual-socket Intel server and up to 229 GiB/s on a 96-core AMD Zen 5 server. The comparison crosses hardware generations and is approximate, but the gap is several-fold. The CPU baselines in the GPU papers ran without document-level parallelism (Section 3); with it, a multicore server parses these streams faster than the GPUs did. On cuJSON's own end-to-end workload, our parser is 17 times faster than cuJSON (Section 5.13).

We evaluate against Pison [8], the closest CPU competitor and a common baseline [6, 23]. Pison builds a leveled structural index (bitmaps of the colon and comma positions at each nesting level) and can construct that index in parallel even for a single document (Section 5.12). We use Pison's six datasets and its published JSONPath queries (with an adapted two-column query for the NSPL dataset). Our contributions are:

- *A parallel stream parser* (Section 4) in which each worker claims a byte range from one atomic counter, snaps both ends to a delimiter with at most two `memchr` calls, and runs an on-demand parser that evaluates the query as it parses; there is no dispatcher, queue, or shared output. Along the way we contributed a fix to simdjson's buffer allocation, which rebuilt every buffer on slices shorter than the previous one.
- *An instruction-level analysis* showing that answering the query costs our parser fewer instructions per byte than Pison spends building its index alone, whereas Pison must then traverse that index a second time to answer the query; a newline-aware fast path lowers our cost further, quantified in Section 5.5.
- *An end-to-end comparison with Pison* that includes the pass converting a stream into the record table Pison requires, which its published timings exclude, with Pison's own loader and with a parallel one we wrote.

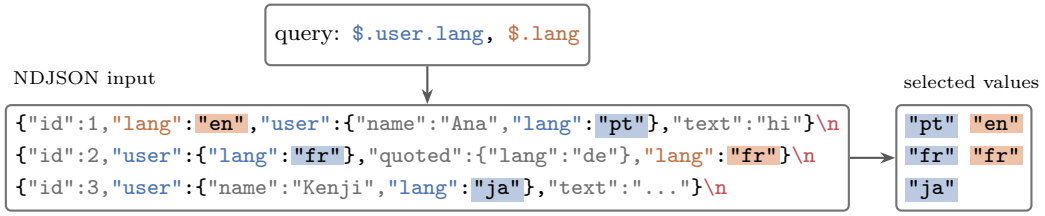


Fig. 1 Three JSON documents, one per line, with a query. Keys on the path `$.user.lang` are in blue and keys on `$.lang` in orange; the values they select are shaded in the same color

– *A comparison with the parsers in common use*: yyjson, RapidJSON, Boost.JSON, and nlohmann/json, each configured favorably and run under the same slicing.
– *A comparison with the GPU parser cuJSON* on its own end-to-end workload, a filter query over GitHub Archive events (Section 5.13).
– *A cross-vendor evaluation* on five contemporary x86-64 server processors—Intel Emerald Rapids, Granite Rapids, and Sapphire Rapids, and AMD Zen 4 and Zen 5—and an open benchmark, with a public corpus of long-record NDJSON drawn from OpenAlex (Section 5.11).

Section 2 defines the problem. Section 3 surveys related work. Section 4 describes the design. Section 5 presents the evaluation. We conclude in Section 6.

2 Background and Problem

We are given a buffer holding many valid JSON documents in one of two framings: NDJSON, where documents are separated by a line feed and no document contains a raw line feed, or RFC 7464, where each document is preceded by a record separator (`0x1E`). The documents may be objects, arrays, numbers, strings, Booleans, or nulls. The task is to parse every document and extract a set of values from each—in our experiments, the values are selected by an expression in JSONPath [7], a query language for selecting parts of a JSON document. Throughput is measured over the input bytes. Figure 1 gives an example.

Two properties of these formats make the stream splittable. First, the delimiter cannot occur inside a document: NDJSON forbids a raw line feed inside a document, and a record separator is not valid JSON text, so a delimiter appearing in the byte stream is necessarily structural, never part of a value. Second, documents are independent: nothing in document $i+1$ affects the parsing of document i. Taken together, these properties imply that a scan for the delimiter is sufficient to find a safe cut point, and that any two cut points delimit a self-contained unit of work.

Commas do *not* have this property. A comma may appear nested inside a document or inside a string, so locating top-level commas requires a structural scan of the whole input—precisely the serial bottleneck we are trying to avoid. We therefore support newline- and record-separator-delimited input, and we do not attempt to parallelize comma-separated input, which would first require that serial scan.

2.1 The simdjson baseline

Our work builds on simdjson [13,11], the first standard-compliant JSON parser to exceed a gigabyte per second on a single core. It parses in two stages. Stage 1 identifies, using SIMD (single instruction, multiple data) instructions, the positions of all structural characters (`[`, `{`, `]`, `}`, `:`, `,`) and the start of every primitive value, and validates UTF-8; the positions are written to an array of integer indices. Given `{"abc":2000}` the indices are $0, 1, 6, 7, 11$. Stage 2 walks through those indices to produce values.

For streams, simdjson offers `iterate_many`. The user gives a batch size, typically around a megabyte, that must exceed the longest document. Stage 1 runs over the current batch, building indices for all complete documents in it; a lightweight scan locates the end of the last complete document; an iterator then yields documents one at a time. When the batch is exhausted the next one is loaded starting at the end of the last document. Only one parser instance exists for the whole stream, so memory use is bounded by the batch size plus a small constant. The simdjson parser can already overlap stage 1 of the next batch with stage 2 of the current one on a second thread. Though it helps, it is limited to two threads.

2.2 On-demand parsing

The *On-Demand* interface of simdjson [11] is a *front-end*: to the programmer it looks like a document tree, but no tree is ever built. What looks like an object is an iterator over key–value pairs, and what looks like

an array is an iterator over values; the underlying state is a single pointer, internally a `json_iterator`, that advances through the stage-1 indices one entry at a time. The interface is a *lazily evaluated tree*.

Values are materialized only when asked for. Parsing of a value is delayed until the program requests it, so a program that names two fields of a large object never converts the numbers or unescapes the strings in the others—it advances the pointer past them. Escaped strings, when they are requested, are decoded into a scratch buffer owned by the parser instance rather than allocated per value. The parser instance is reusable: it owns the index and the scratch buffer and is designed to be carried from one document to the next.

A DOM (Document Object Model) parser must materialize the whole document before the program can look at any of it: if the program wants a subset, it pays for the rest anyway, and it may then copy the values into its own types. An event-based parser (SAX, Simple API for XML) can skip parts of the input, but it pushes the work of tracking position in the document onto the programmer, and the resulting code is long and easy to get wrong. On-Demand thus offers the DOM's interface at the streaming parser's cost.

3 Related Work

Fast JSON parsing has followed two broadly independent lines. One makes the parsing of a single stream of bytes cheaper, by classifying many characters at a time with SIMD instructions and by deferring or avoiding work the application never asks for. The other makes the parsing parallel, either across the cores of a CPU or on a GPU, and must first answer the question of how to divide a format whose meaning depends on what came before. Our contribution sits in the second line but depends on the first: the unit of division is a whole document, and what each worker runs is a fast serial parser. We survey both lines below, and give particular attention to Pison, which is our comparison point.

SIMD parsing. Mison [15] introduced the idea of building a *structural index* for JSON—bitmaps recording the positions of structural characters—and of using it to jump directly to the fields a query needs, skipping the rest. Sparser [18] goes further and filters raw bytes before parsing, so that records a query cannot match are never parsed at all. JSONSki [9], from the authors of Pison, evaluates a path query in a single streaming pass and uses bit-parallel operations to fast-forward over the substructures the query does not need, without building a full index; it runs on one thread. The simdjson parser [13] made the index construction itself fast, using SIMD instructions to classify 64 bytes at a time and carry-less multiplication to resolve quote nesting, and paired it with a validating second stage; it was the first standard-compliant parser to exceed a gigabyte per second per core. Its on-demand interface [11] later removed the intermediate tree, decoding values only as the application requests them. Related data-parallel work has applied the same style to UTF-8 validation [10], base64 [17], and IP-address parsing [16].

Tree-building parsers. Many systems that ingest JSON use neither a structural index nor an on-demand front-end: they build a tree. RapidJSON and Boost.JSON parse into their own document types, yyjson parses into a compact immutable value tree, and nlohmann/json parses into a tagged union over the standard containers.

Parallel structural indexing. Pison [8] extends Mison's leveled bitmaps with a scheme for constructing them *in parallel for a single record*: the input is partitioned, each chunk is indexed speculatively under both possible in-string/out-of-string starting states, and the correct alternative is selected once the preceding chunk resolves.

Streams of documents. Schulze [21] extends, in his bachelor's thesis, the JSON parser of the Karbonit in-memory database [19] to NDJSON, parsing line by line with a thread pool. On Microsoft Academic Graph files he reports up to a 7× speedup with 19 threads, reaching 230 MB/s, two orders of magnitude below what we report here, which illustrates how much the per-document parser matters once the parallel decomposition is in place.

Data systems. Analytical systems that ingest NDJSON already split it at line feeds and parse the pieces on several threads. Apache Arrow's JSON reader [1] cuts its input into blocks (1 MiB by default) that end at a line feed and parses them in parallel with RapidJSON (as of pyarrow 26). DuckDB [20] reads newline-delimited files in parallel and parses them with yyjson [12]. Polars [24] divides the buffer into one range per thread, moves each end forward to a line feed that follows a closing brace, found with `memchr`, and parses each line into a value tree with simd-json, a Rust library derived from simdjson. Spark [2] reads JSON Lines through Hadoop's line-oriented input splits, in which each reader skips to the first line feed of its byte range, and parses each record with Jackson. The splitting rule is therefore not new. These systems convert every record into rows or columns of a table, with a schema given or inferred, through a tree-building or event-based parser. We keep

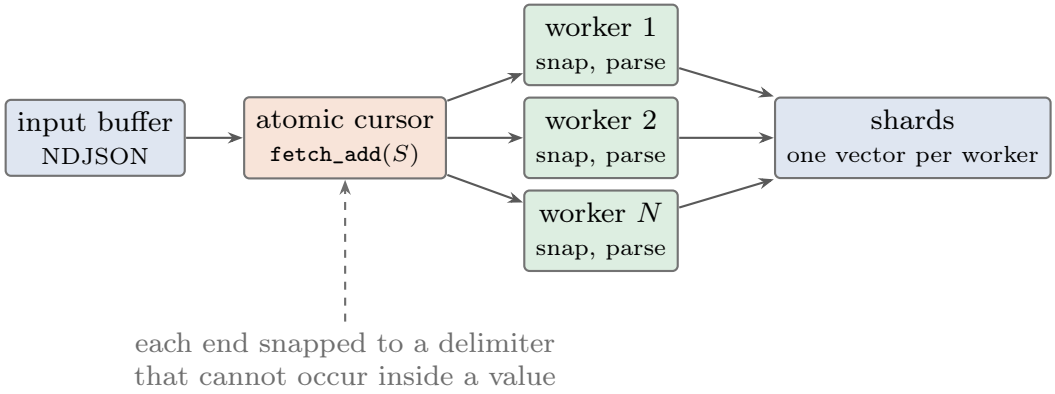


Fig. 2 The parallel stream parser. Each worker claims the next S bytes from one atomic counter and snaps both ends forward to a line feed (NDJSON) or a record separator (RFC 7464), so slices are exactly adjacent and a document is never split. There is no dispatcher thread and no queue: each worker owns a parser and an output vector, and the only shared mutable state is the counter

the rule and change what runs inside each slice: an on-demand parser that evaluates the query while parsing and never decodes the fields the query does not name. Section 5.7 measures what that choice is worth against tree-building parsers placed under the same slicing, and Appendix A measures Arrow, DuckDB and Polars.

GPU parsing. Stehle and Jacobsen [22] introduce ParPaRaw, a massively parallel algorithm for delimiter-separated data on GPUs, using bitmap indices for delimiters and control symbols followed by prefix sums to assign record and column offsets; they report up to 14 GB/s on an NVIDIA Titan X GPU. GpJSON [23] ports the multi-level bitmap index to CUDA for LD-JSON and JSONPath evaluation, reporting 2.9× to 4.5× over CPU parsers and 6–8× over NVIDIA RAPIDS cuDF on an A100. The cuJSON parser [6] redesigns the algorithm around minimal branching, offloading UTF-8 validation, tokenization, and nesting recognition to the GPU, and reports gains over simdjson, Pison, cuDF, and GpJSON. Table 1 gives the throughput both report.

Their SIMD CPU baselines, simdjson and Pison, are run without document-level parallelism: the Pison harnesses published with cuJSON use a single thread for most queries and simdjson its built-in two-thread pipeline, and GpJSON compares against them on a 12-core machine, noting that Pison would need line-level parallelism added by hand.

4 Design

We do the parsing itself using simdjson. To parallelize the work, we decompose the input into slices efficiently.

4.1 Slicing without parsing

We use a single shared atomic counter (Figure 2). For a target slice size S (see Section 5.9) a worker takes the next multiple of S with one `fetch_add`, then *snaps* both ends forward to the next delimiter with a character search (`memchr`): for newline-delimited streams just after a line feed, for JSON text sequences just before a record separator. Each resulting slice therefore contains a whole number of documents. A worker may claim the next slice from the counter (*dynamic*), or own a contiguous range of slices (*static*). The two differ in load balance, memory locality and counter traffic.

A worker that claims the raw offset r ends at snap($r+S$), and the worker that claims the offset $r+S$ begins at snap($r+S$), so the slices are exactly adjacent and never overlap. When both ends of a claimed range snap to the same boundary—a range that falls inside a document longer than the slice—that slice is empty; the document belongs to the preceding nonempty slice and is parsed exactly once. A worker detects this case without scanning to the end of the document: it searches for its start only within its own range, and if the range holds no delimiter, the slice is empty. Only the worker that owns a document searches past its range, to find the document's end. Each byte is thus searched about twice, whatever the document lengths.

To see how the rule plays out, consider a stream of fourteen documents whose end offsets—the first byte after each document—are 1, 3, 200, 210, 280, 300, 550, 555, 800, 810, 1900, 1910, 2200 and 2300 KiB, and, for illustration, a target slice size $S = 256$ KiB. Slices are half-open, so a document whose end offset is 280 occupies bytes up to 279. Table 2 lists the claims and the slices they produce.

Table 2 A worked example of the claiming rule with two workers ($S = 256\,\text{KiB}$). The order of claims is fixed by the atomic counter; which worker performs a given claim depends on availability and is illustrative

worker	slice (KiB)	#docs
A	$[\text{snap}(0), \text{snap}(256)) = [0, 280)$	1–5
B	$[\text{snap}(256), \text{snap}(512)) = [280, 550)$	6–7
A	$[\text{snap}(512), \text{snap}(768)) = [550, 800)$	8–9
B	$[\text{snap}(768), \text{snap}(1024)) = [800, 1900)$	10–11
A	$[\text{snap}(1024), \text{snap}(1280)) = [1900, 1900)$	empty
A	$[\text{snap}(1280), \text{snap}(1536)) = [1900, 1900)$	empty
A	$[\text{snap}(1536), \text{snap}(1792)) = [1900, 1900)$	empty
A	$[\text{snap}(1792), \text{snap}(2048)) = [1900, 2200)$	12–13
B	$[\text{snap}(2048), \text{snap}(2304)) = [2200, 2300)$	14

Each worker owns an `ondemand::parser` and its own output vector, so nothing is allocated or locked in common. Results are returned as one vector per worker ("shards"). Values keep their relative order within a shard. With dynamic assignment the shards are interleaved with respect to the input, and a caller that needs global order can record each value's position and sort by it, a cost we do not measure; with static assignment, concatenating the shards in worker order already restores the input order.

4.2 Query evaluation inside each slice

The worker applies a user-supplied extraction function to every document it parses (Listing 1), so the query is evaluated *during* the parsing, and the parallel decomposition needs no separate query pass. The values go straight into the worker's output vector, and only the values the query names are decoded.

4.3 Contrast: what Pison requires

A structural-index parser such as Pison consumes, for each document, a `Record` descriptor giving a pointer, an offset, and a length. A stream must therefore be converted into a table of N such descriptors before any indexing can start. That conversion is not merely a scan. Pison's index construction reads the record 32 bytes at a time and emits one 64-bit bitmap word per 64-byte block—one bit per byte—so a record of length ℓ is read in $\lfloor \ell/32 \rfloor$ steps but indexed with only $\lfloor \ell/64 \rfloor$ words. Its final $\ell \bmod 64$ bytes are therefore left out of the index, invisible to queries. Pison's own loader avoids this by padding each record with filler bytes up to a 64-byte multiple and concatenating the padded records into one fresh buffer (Figure 3).

Pison builds bitmaps only for the first `level_num` nesting levels, so `level_num` must cover the query's nesting depth, counting both objects and arrays. Figure 4 contrasts the data flows of our approach and of Pison's.

```
// query (JSONPath-style): $.user.lang, $.lang
error_code extract(auto doc, result &out) {
  ondemand::object root;
  if (auto err = doc.get_object().get(root)) {
    return err;
  }
  for (auto field : root) {
    std::string_view key;
    if (auto err = field.unescaped_key().get(key)) {
      return err;
    }
    if (key == "lang") {
      out.take(field.value());
    } else if (key == "user") {
      ondemand::object user;
      if (!field.value().get_object().get(user)) {
        ondemand::value lang;
        if (!user["lang"].get(lang)) {
          out.take(lang);
        }
      }
    }
  }
  return SUCCESS;
}
```

Listing 1 An extraction function. The parallel driver calls this once per document, on the worker thread that parsed it; `result::take` appends a selected value to that worker's output vector

5 Experiments

We compare our parser with Pison, on one thread and in parallel, then with the tree-building parsers in common use, and finally with the GPU parser cuJSON on its own workload.

5.1 Methodology

All measurements start from an in-memory buffer: file input/output is outside every timed region. Every Pison measurement uses our rewritten loaders, which take the

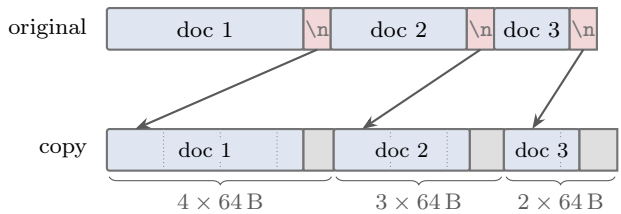


Fig. 3 Pison's loader. The loader copies each document into a fresh buffer and appends filler bytes (gray) until its length is a multiple of 64 bytes; dotted lines mark the 64-byte blocks

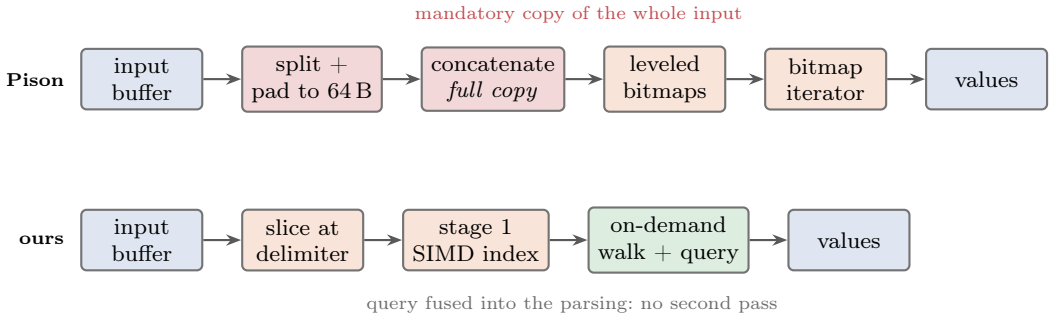


Fig. 4 Data flow. Pison's indexer covers only complete 64-byte blocks, so its loader pads each record to a 64-byte multiple in a contiguous buffer and the input is copied before indexing begins, and the query is a second traversal over the finished index. The on-demand parser slices the original buffer in place and decodes only the values the query asks for, so no copy and no separate query pass occur

buffer in memory. The exception is the row of Table 5 that times Pison's own loader unmodified: it accepts only a path, so that row includes a page-cache read. We report the minimum wall-clock time over ten repetitions; the instruction, cycle and CPU-time figures come from the fastest repetition. We measure the spread between repetitions as a percentage, $100\,(t_{\max} - t_{\min})/t_{\min}$. At the parallel peaks of Table 7 it has a median of 3 % and a maximum of 8 % for our parser, and a median of 6 % and a maximum of 54 % for Pison's index and query; reporting the minimum keeps that noise out of the comparison. The cuJSON comparison (Table 11) instead reports the mean of ten runs, as the cuJSON benchmarks do. Throughput is the number of input bytes divided by the wall-clock time. In the CPU comparisons the timed region covers parsing, query evaluation and the extraction of each selected value into an accumulator—a match count plus an order-independent hash of the extracted values—so that no engine retains the values it extracts and no library's allocation strategy is measured. Using hardware performance counters [14] we additionally report retired instructions per input byte (i/B), CPU cycles per input byte (c/B), and instructions per cycle (IPC).

Counters are collected only for *single-threaded* configurations. The Linux `perf_event` interface, opened without inheritance, observes only the calling thread. For parallel configurations we report wall-clock throughput together with CPU seconds per gibibyte, taken from `getrusage`, which does aggregate all threads.

Every parallel engine—ours, Pison's index and query, our parallel loader and the DOM drivers—creates its worker threads at the start of each timed call and joins them at the end; none keeps a pool of threads alive between calls. Thread creation is therefore part of every parallel measurement, as it is for a program that parses one stream and exits.

Each dataset is read into memory once. In the same process we measure the single-threaded configuration first, then the thread sweep. We keep this order throughout the main benchmark: every corpus, both assignment policies, the slice-size and scheduling studies, and the OpenAlex scholarly-metadata case study (Section 5.11). The thread-sweep numbers therefore start from a buffer that the serial run has already traversed, with caches and allocator state warm.

Fairness. We adopted the following rules.

1. Every engine answers the *same* query and must agree. Agreement is checked on the number of matches and on an order-independent hash of the extracted values (a sum of per-value hashes, since our parallel parser returns values grouped by worker).
2. Each competing library is configured favorably (Section 5.7): we let it reuse memory so as not to measure `malloc`.
3. We measure Pison's record-table pass, which its published timings exclude, for four implementations (Section 5.4).
4. Pison is given the smallest number of bitmap levels that is *valid* for each dataset, which is the setting most favorable. The value is the query's nesting depth, counting both objects and arrays (Section 4.3): 2, 3, 8, 1, 2, and 5, in the order of Table 3. We verified empirically that, on every dataset whose minimum exceeds one, one level fewer crashes Pison, regardless of how deeply the documents themselves nest.
5. We report three workloads: *index* (build the structural index only), *query* (index plus JSONPath evaluation, extracting raw tokens exactly as Pison's iterator returns them), and *decode* (additionally unescape strings and convert numbers, which is what an application consumes). Since a structural index cannot decode, we wrote the decoding step for Pison our-

selves so that our parser and Pison can be compared on equal output.

5.2 Corpus

We use the six datasets of the Pison paper, also used by cuJSON [6]: tweets (TT), Best Buy products (BB), Google Maps directions (GMD), the UK National Statistics Postcode Lookup (NSPL), Walmart products (WM), and Wikipedia entities (WP). Each is published in two forms: one bulky JSON record, and a stream of the elements of its dominating array, one per line. The public collection ships all six bulky records but only two of the six JSON-lines files, so we regenerate the JSON-lines form from the bulky records by minifying each element of the dominating array onto its own line.

The regeneration is validated in two ways. For the two datasets published in both forms, the derived file has exactly the same record count as the published one. More strongly, five of the six queries reproduce the match count printed in Pison's paper *exactly* (Table 3). NSPL is the exception: the query Pison publishes reads a metadata header that exists only in the original file, not in the derived JSON-lines form, so we define our own two-column query over the records; its match count is therefore ours, not a reproduction of a published one.

Table 3 summarizes the six datasets; the Query column uses JSONPath-style notation, and each engine evaluates the same path directly.

5.3 Systems

Table 4 lists the 5 machines. All are x86-64 with AVX-512, and simdjson selects its `icelake` kernel on each. All but one are ephemeral bare-metal AWS instances, which are required for direct access to the performance-monitoring unit; the remaining machine, our reference system throughout, is a local dual-socket Intel Xeon Gold 6548N. Each compilation was done with GCC 14 in Release mode (`-O3`). The harness and the header-only DOM parsers additionally compile with `-march=native`; simdjson is built generically, so that its kernels are selected at runtime; and Pison is built with the SIMD flags of its own makefile.

5.4 The record table

Table 5 reports the pass that converts a stream into the descriptor table Pison consumes. Four implementations are measured. Pison's own `RecordLoader::loadRecords` reaches about 1.3 GiB/s, retiring 0.78 instructions per byte; it calls `strlen` five times per record, copies the payload three times (stdio buffer, then a growing `std::string`, then a fresh aligned buffer), and allocates one descriptor per line. Writing the same required layout carefully brings this to 3.9 GiB/s at 0.13 instructions per byte on the tweet stream—a 6-fold reduction in instruction count for identical output—and doing it in parallel reaches 23.7 to 29.5 GiB/s at its best worker count.

The last two rows isolate what the copy costs. Finding the record boundaries alone—one `memchr` per record, no copy, no padding—runs at 10 to 16 GiB/s on one thread and 109 to 124 GiB/s in parallel. The gap between that and the parallel loader is the padded copy, and it is not removable from Pison's current interface: Section 4.3 showed that its indexer covers only complete 64-byte blocks, so records must be padded to a 64-byte multiple and laid out contiguously.

5.5 Single-threaded cost

Table 6 and Figure 5 report the single-threaded cost of answering the same query, per input byte, excluding the record table. Two parser modes appear in the table. Both accept the same NDJSON bytes: the whitespace-delimited mode treats any JSON whitespace as a separator, whereas the newline-delimited mode relies on the guarantee that no document contains a raw line feed and can therefore skip an unread remainder. The index phase is the same stage-1 pass in both; only the boundary search differs. In whitespace mode, finding where a document ends requires walking all of its structural tokens, so the *index* phase is stage 1 plus a walk of every document; in newline mode it is stage 1 plus one `memchr` per document. On simdjson's own serial stream parser, whitespace framing is slower only where the query leaves most of each record unread: by 18 % on Best Buy and 6 % on NSPL, and by at most 2 % on the other datasets. Comma framing, which cannot be split without a structural scan (Section 2), is 1.3 to 2.8 times slower than newline framing in that parser, which is why we use NDJSON throughout.

This section is best read in four steps: (1) what the structural index costs on its own, (2) what the query adds to it, (3) what decoding adds beyond that, and (4) how little any of it depends on the machine.

The on-demand parser is cheaper even at index construction. Building the structural index costs Pison 3.44 to 5.93 instructions per byte across the six datasets; simdjson's stage 1 (the newline-mode index phase) costs 1.18 to 1.60, and is faster in terms of throughput on all six (4.33 to 5.80 GiB/s against 1.88 to 3.81 GiB/s). This

Table 3 Benchmark datasets, mean record sizes, queries, match counts, and offline-selected slice sizes

		Size (MiB)	Records	Mean B/record	Query	Matches	Slice (KiB)
TT	`twitter`	803	155 489	5 415	`$.user.lang, $.lang`	300 270	64
BB	`bestbuy`	996	230 089	4 538	`$.categoryPath[1:3].id`	459 332	64
GMD	`google_map`	1 036	4 439	244 790	`$.routes[*].legs[*].steps[*].distance.text`	1 716 752	1024
NSPL	`nspl`	1 065	1 754 882	637	`$[8], $[9]`	3 509 764	64
WM	`walmart`	949	272 499	3 651	`$.bestMarketplacePrice.price, $.name`	288 391	64
WP	`wiki`	1 048	137 877	7 969	`$.claims.P150[*].mainsnak.property`	15 603	64

Table 4 Benchmark systems (all x86-64 with AVX-512) and compiler versions; the 6548N is the reference machine

Processor	Cores/threads	Microarchitecture	Compiler
Intel Xeon Gold 6548N	64/128	Emerald Rapids (2024)	GCC 14.3.1
Intel Xeon 6975P-C	96/192	Granite Rapids (2025)	GCC 14.2.0
Intel Xeon Platinum 8488C	48/96	Sapphire Rapids (2023)	GCC 14.2.0
AMD EPYC 9R45	96/96	Zen 5 / Turin (2024)	GCC 14.2.0
AMD EPYC 9R14	192/192	Zen 4 / Genoa (2023)	GCC 14.2.0

Table 5 Record preparation throughput on 6548N, in GiB/s; parentheses give the single-threaded i/B (counters are not collected for multi-threaded runs), or, for the parallel rows, the number of workers with the highest measured throughput. The reference and parallel rows produce padded records; the boundary rows construct descriptors without copying or padding

Loader	TT	BB	GMD	NSPL	WM	WP
Pison, published (1 thread)	1.25 (0.78)	1.25 (0.80)	1.25 (0.60)	1.07 (2.08)	1.08 (0.91)	1.14 (0.72)
same layout, reference (1 thread)	3.87 (0.13)	3.73 (0.13)	4.00 (0.09)	3.27 (0.44)	3.72 (0.14)	3.92 (0.12)
same layout, parallel (best of sweep)	28.8 (32)	28.8 (32)	29.5 (32)	23.7 (64)	27.8 (32)	29.2 (32)
reference boundaries, 1 thread	11.0 (0.11)	14.5 (0.12)	16.0 (0.09)	10.3 (0.25)	10.9 (0.12)	10.9 (0.11)
reference boundaries, parallel (best of sweep)	119 (16)	124 (16)	124 (16)	109 (64)	124 (16)	122 (32)

is despite simdjson also validating UTF-8, which Pison does not.

The query is cheap for an on-demand parser, and expensive for an index. In whitespace mode, going from *index* to *index+query* costs simdjson at most 0.83 additional instructions per byte on the reference machine: answering the query costs barely more than walking every document without reading it, because the query is fused into that walk and the fields it does not name are skipped rather than decoded. Over stage 1 alone (newline mode), the query adds 0.4 to 1.5 instructions per byte. The same step costs Pison between 0.75 and 3.83 additional instructions per byte, because the index must first be built in full and then traversed a second time by the bitmap iterator. In either mode, answering the query costs our parser fewer instructions per byte than Pison spends building its index alone, on every dataset and machine. On NSPL, where the query picks two columns out of a 50-element array, Pison's instruction count rises by 74 % (5.15 to 8.98) while simdjson's in whitespace mode rises by 32 % (2.59 to 3.42).

Decoding is nearly free too. Unescaping strings and converting numbers on top of the query costs our parser and Pison little on these datasets (at most 0.78 instructions per byte for Pison, 0.06 for simdjson), because the queries select few fields. Overall, single-threaded, our parser answers the query 1.8 to 4.4 times faster than Pison across the six datasets and five machines, while retiring fewer instructions per byte in every one of the 30 combinations.

The counts vary less than throughput across machines. Figure 6 puts every machine's measurement of the same quantity side by side. In the index and query phases, retired instructions per byte vary by at most 1.1 % for simdjson and 9.9 % for Pison across three Intel generations and two AMD ones, while throughput varies by more than twofold. The difference is consistent with instruction-level parallelism: on the tweet stream our parser retires 2.60 instructions per byte on 6548N and 2.60 on the Zen 5 part—essentially the same—but executes them at 3.02 against 3.82 instructions per cycle; together with frequency, this explains the 3.76 GiB/s-versus-6.18 GiB/s gap. The counts therefore reflect the designs, not the machines.

5.6 Parallel scaling

Figure 7 shows throughput against thread count on the tweet stream, and Table 7 gives peak throughput for every dataset. Both parsers scale well up to 32

Table 6 Single-threaded index and query per-byte costs on 6548N. Index is Pison's leveled bitmaps, or simdjson's stage 1 plus the search for each document's end: one `memchr` in newline mode, a walk of every structural token in whitespace mode. Query evaluates the requested fields. Lower is better except for GiB/s and IPC

(a) tweets, Best Buy, Google Maps directions

		TT				BB				GMD			
Engine	Phase	GiB/s	i/B	c/B	IPC	GiB/s	i/B	c/B	IPC	GiB/s	i/B	c/B	IPC
Pison	index	3.07	3.75	1.05	3.56	2.92	4.69	1.11	4.23	2.62	4.19	1.24	3.39
	query	1.68	6.03	1.93	3.13	1.81	7.10	1.79	3.97	1.31	7.39	2.48	2.98
ours (newline)	index	5.13	1.27	0.63	2.00	5.80	1.24	0.56	2.21	4.33	1.58	0.75	2.11
	query	3.76	2.60	0.86	3.02	5.11	1.59	0.63	2.51	3.42	2.72	0.95	2.87
ours (whitespace)	index	4.23	2.32	0.77	3.03	4.41	2.56	0.73	3.48	3.75	2.48	0.86	2.86
	query	3.76	2.60	0.86	3.02	4.19	2.65	0.77	3.43	3.47	2.72	0.93	2.91

(b) NSPL, Walmart, Wikipedia

		NSPL				WM				WP			
Engine	Phase	GiB/s	i/B	c/B	IPC	GiB/s	i/B	c/B	IPC	GiB/s	i/B	c/B	IPC
Pison	index	2.55	5.15	1.27	4.05	3.81	3.44	0.85	4.04	1.88	5.93	1.73	3.44
	query	1.35	8.98	2.41	3.73	1.91	5.84	1.70	3.44	1.53	6.68	2.12	3.15
ours (newline)	index	4.81	1.60	0.67	2.37	5.35	1.18	0.61	1.95	4.82	1.26	0.67	1.87
	query	3.93	2.65	0.82	3.21	4.21	1.97	0.77	2.56	3.41	2.75	0.95	2.89
ours (whitespace)	index	4.21	2.59	0.77	3.36	5.44	1.36	0.60	2.29	3.68	2.98	0.88	3.38
	query	3.70	3.42	0.88	3.90	4.22	1.97	0.77	2.56	3.37	3.00	0.96	3.12

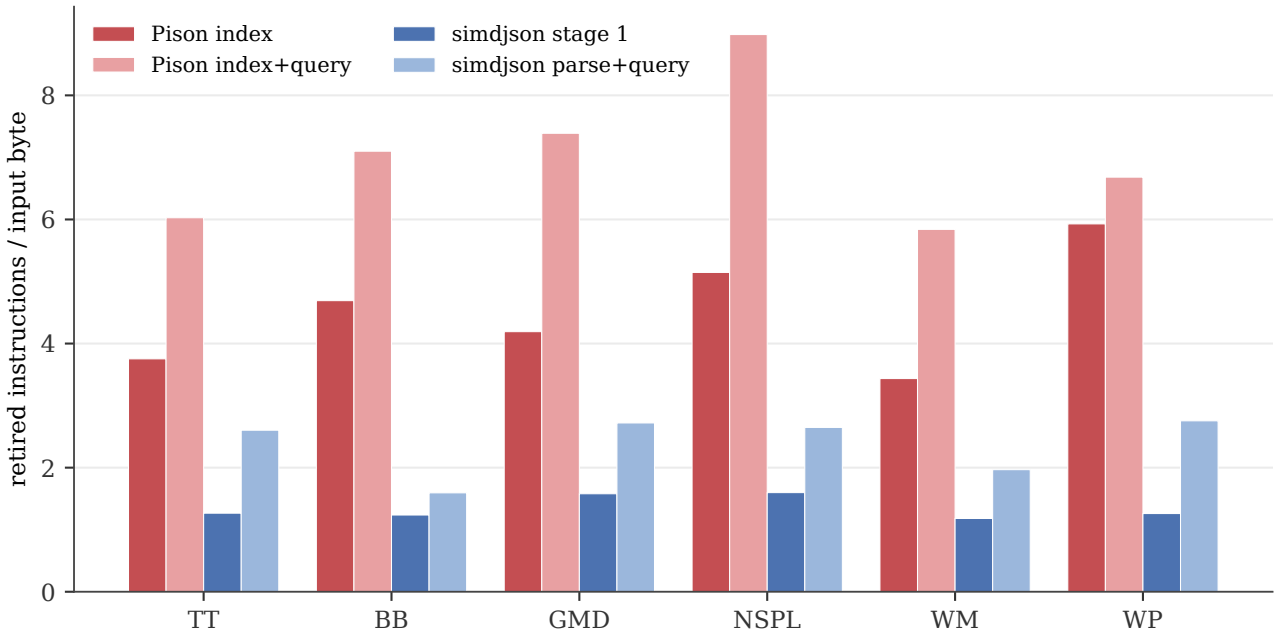


Fig. 5 Retired instructions per input byte, single-threaded, for the newline-aware path, whose *index* phase is stage 1 plus one `memchr` per document. The on-demand parser retires fewer instructions than the structural index for both *index* and *index+query* on every dataset. Table 6 also gives whitespace mode, whose index phase walks every document to find its end; adding the query to that walk costs the on-demand parser at most 0.83 instructions per byte

threads: ours reaches 23 times its one-worker rate and Pison's index and query 29 times, starting from a single-threaded rate about half of ours. On 6548N our parser goes 3.51 GiB/s, 6.84 GiB/s, 13.35 GiB/s, 24.95 GiB/s, 44.51 GiB/s and 79.35 GiB/s at 1, 2, 4, 8, 16 and 32 threads, and peaks at 91.1 GiB/s at 64 threads—the physical core count of this dual-socket machine—before falling back to 74.6 GiB/s at 128 threads (Section 5.8 examines why). Pison's index-plus-query, which still excludes its record table and so is not yet a comparable

Table 7 Peak in-memory query throughput (GiB/s) over the worker sweep; parentheses give the number of workers with the highest measured throughput; buffer in, query results consumed. The loader rows include Pison's record-table pass; the last row of each machine excludes it and gives Pison's index and query alone. Our parser appears twice: workers claim slices from a shared counter (dynamic) or own contiguous ranges (static)

Machine	Engine	TT	BB	GMD	NSPL	WM	WP
6548N	simdjson (dynamic)	91.1 (64)	105 (64)	76.5 (32)	64.5 (64)	92.7 (64)	86.7 (64)
	simdjson (static)	93.7 (64)	110 (64)	71.9 (32)	69.0 (64)	100 (64)	87.1 (64)
	Pison (parallel loader)	18.0 (64)	18.4 (32)	14.9 (128)	15.0 (64)	18.8 (64)	18.2 (64)
	Pison (reference loader)	3.37 (128)	3.45 (64)	3.41 (128)	2.89 (128)	3.37 (128)	3.43 (128)
	Pison (no record table)	59.5 (128)	62.1 (128)	37.7 (128)	40.6 (64)	64.8 (128)	63.1 (128)
6975P-C	simdjson (dynamic)	104 (64)	120 (96)	108 (96)	75.1 (64)	115 (96)	111 (96)
	simdjson (static)	114 (96)	128 (64)	104 (96)	89.8 (128)	121 (96)	111 (96)
	Pison (parallel loader)	8.95 (96)	12.8 (64)	9.14 (128)	10.5 (128)	12.9 (64)	9.76 (128)
	Pison (reference loader)	1.92 (32)	1.98 (96)	1.99 (128)	1.99 (192)	1.98 (64)	2.00 (128)
	Pison (no record table)	59.5 (64)	110 (128)	46.7 (192)	62.4 (192)	106 (128)	103 (128)
8488C	simdjson (dynamic)	105 (64)	121 (64)	103 (48)	67.4 (96)	111 (64)	103 (96)
	simdjson (static)	111 (96)	129 (96)	98.0 (48)	75.5 (96)	123 (96)	111 (96)
	Pison (parallel loader)	10.2 (96)	10.9 (96)	9.71 (96)	9.08 (96)	10.7 (96)	10.3 (96)
	Pison (reference loader)	1.64 (96)	1.65 (96)	1.63 (96)	1.69 (96)	1.65 (96)	1.64 (32)
	Pison (no record table)	71.5 (96)	71.9 (96)	40.4 (96)	46.1 (96)	78.1 (96)	70.3 (96)
9R45	simdjson (dynamic)	184 (96)	229 (64)	192 (64)	142 (96)	204 (96)	175 (96)
	simdjson (static)	196 (96)	246 (64)	190 (64)	138 (96)	215 (96)	177 (96)
	Pison (parallel loader)	14.7 (96)	16.1 (96)	15.4 (96)	14.3 (96)	16.3 (96)	16.3 (96)
	Pison (reference loader)	3.29 (32)	3.31 (96)	3.64 (64)	3.07 (96)	3.27 (32)	3.47 (64)
	Pison (no record table)	126 (96)	133 (96)	66.6 (96)	87.7 (96)	132 (96)	127 (96)
9R14	simdjson (dynamic)	112 (64)	154 (64)	134 (64)	79.8 (128)	121 (64)	111 (128)
	simdjson (static)	106 (64)	130 (64)	129 (64)	74.5 (128)	118 (64)	110 (128)
	Pison (parallel loader)	7.47 (128)	9.07 (128)	8.25 (64)	7.51 (128)	7.72 (128)	9.05 (128)
	Pison (reference loader)	1.53 (128)	1.55 (128)	1.55 (128)	1.47 (128)	1.51 (192)	1.55 (128)
	Pison (no record table)	94.5 (128)	88.0 (128)	50.0 (192)	79.2 (128)	104 (128)	84.4 (192)

quantity, peaks at 59.5 GiB/s; Table 7 reports it in a separate row, below the comparable end-to-end ones.

The same shape holds on the other four machines, at higher absolute rates. With dynamic assignment, the policy of the main results, the fastest configuration is 229 GiB/s, on the Zen 5 part and the Best Buy stream; on that same machine the tweet stream reaches 184 GiB/s and NSPL 142 GiB/s, so the spread across datasets spans 142 to 229 GiB/s and no single number characterizes the machine. With static assignment the fastest configuration is the same stream on the same machine, at 246 GiB/s on 64 threads (Section 5.10). Every machine peaks between 24 and 34 times the serial parser's rate on the same machine, on the tweet stream.

Two patterns are worth noting. First, for the tweet stream, throughput saturates in the 64 to 96 worker range on all five machines regardless of how many cores they have, and declines past it where more workers were measured. Extra threads beyond the physical cores help mainly on the 48-core 8488C, where 5 of the six datasets peak above its core count; elsewhere the gain, if any, is within run-to-run variation. The 192-core Zen 4 part peaks at 64 threads on the tweet stream, a third of its cores. Relatedly, core count alone predicts little—the 96-core Zen 5 part reaches 1.64 times the tweet-stream peak of the 192-core Zen 4 part. Per-core speed accounts for part of that gap: single-threaded, Zen 5 runs the same query 1.39 times faster (Zen 5 executes AVX-512 on native 512-bit datapaths, while Zen 4 double-pumps its 256-bit execution units). The rest comes from spanning two sockets: on one socket the Zen 4 part reaches 128 GiB/s, and 188 GiB/s against that is a factor of 1.47, close to the single-threaded ratio (Section 5.8).

The end-to-end picture is different. When including the record table, which we need to do for a meaningful comparison, Pison reaches 18.0 GiB/s on 6548N with our parallel loader and 3.37 GiB/s with the serial reference loader. Across all six datasets and all five machines, the end-to-end advantage of our parser over the strengthened Pison configuration is between 4.3 and 17.0 times (Figure 8 and Table 7); it is smallest on NSPL on the 6548N and largest on Best Buy on the 9R14, the ma-

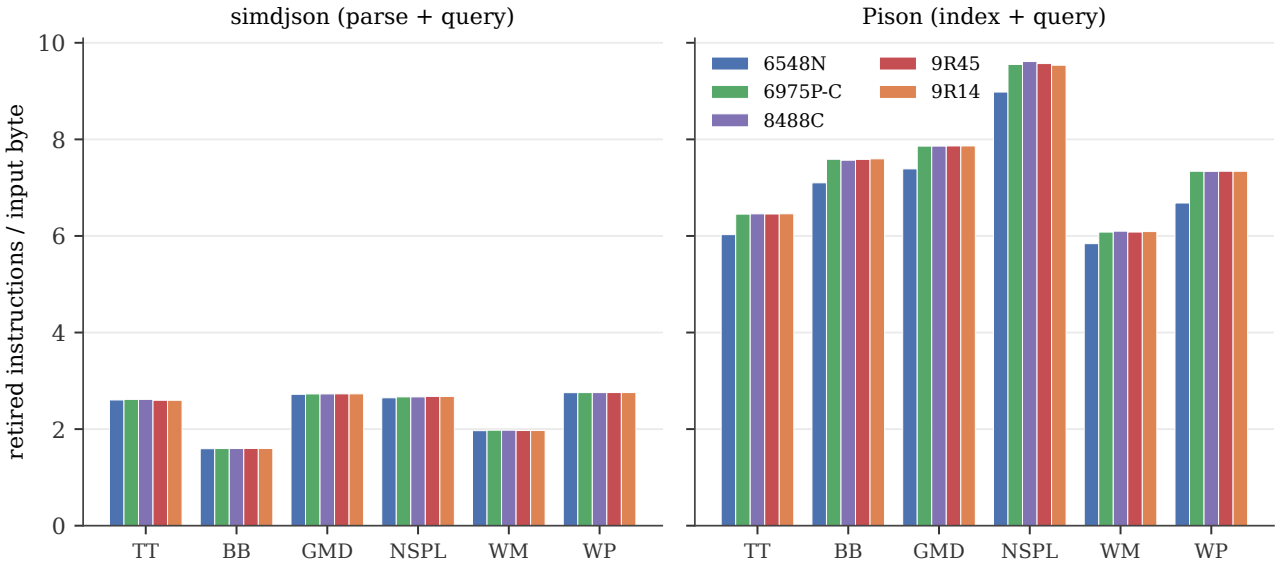


Fig. 6 Retired instructions per input byte for the query workload, one bar per machine. Counts vary less than throughput across vendors and generations—at most 1.1 % for simdjson and 9.9 % for Pison across machines, for every dataset and phase—so instruction count is largely a property of the algorithm and compiler, not the microarchitecture

chine with the most cores, because Pison's mandatory copy scales worse than the parsing it precedes.

Excluding Pison's record table isolates the parsers themselves (last row of each machine in Table 7). On that comparison, on the reference machine, the gap is narrowest on Wikipedia—63.1 GiB/s for Pison against 86.7 GiB/s for ours on 6548N, a factor of 1.37, and between 1.08 and 1.47 across the 5 machines—so on that dataset most of the end-to-end difference is the padded copy. It is widest on Google Maps directions (GMD), between 2.0 and 2.9 times on every machine. GMD has by far the largest documents (239 KiB on average, against a few kilobytes elsewhere) and the heaviest query, which descends through three nested arrays and returns 1.7 million matches. Our parser reaches its peak at fewer workers there (32 on 6548N), because slicing granularity is bounded below by the document size, yet still reaches 76.5 to 192 GiB/s; Pison's index and query peak at 37.7 to 66.6 GiB/s, its lowest of the six datasets on every machine.

5.7 Conventional parsers

Commonly used alternatives are tree-building libraries: yyjson, RapidJSON, Boost.JSON, and nlohmann/json. We therefore measured all four on the same streams, with the same queries, on the reference machine, and verified they return the same values. Figure 9 compares them, and Table 8 gives the numbers. We repeated the comparison on the Zen 5 machine (Figure 10). The versions are pinned by commit in the benchmark: yyjson 0.10.0, RapidJSON at commit `24b5e7a`, Boost.JSON 1.87.0, and nlohmann/json 3.11.3.

Each library is configured in the way most favorable to it. Three of them reuse an arena across documents—yyjson a pool allocator, RapidJSON a `MemoryPoolAllocator` over a user buffer, and Boost.JSON a `monotonic_resource`—so that their steady state performs no allocation.

One thread: the instruction counts. Our parser retires 1.6 to 2.8 instructions per input byte across the six datasets. The yyjson library retires 4.8 to 8.1, Boost.JSON 9.3 to 28.7, RapidJSON 30.9 to 34.6, and nlohmann/json 69.9 to 113. In throughput the same ordering gives us 2.2 to 2.9 times yyjson, 3.2 to 8.8 times Boost.JSON, 6.6 to 9.2 times RapidJSON, and 24 to 53 times nlohmann/json.

Many threads: slower parsers scale further. We ran each library under the same atomic-cursor decomposition, each worker with its own parser and arena, and checked that the result still matches on the match count and the value hash, a consistency check on the slicing. All four libraries scale with the thread count: on the tweet stream, from one thread to their best, yyjson gains 38 times, Boost.JSON 46, RapidJSON 51, and nlohmann/json 67, against 24 for ours. The ordering there is inverted, which is consistent with a parser that is slower per thread reaching the memory system's limit later, so it has more room to scale.

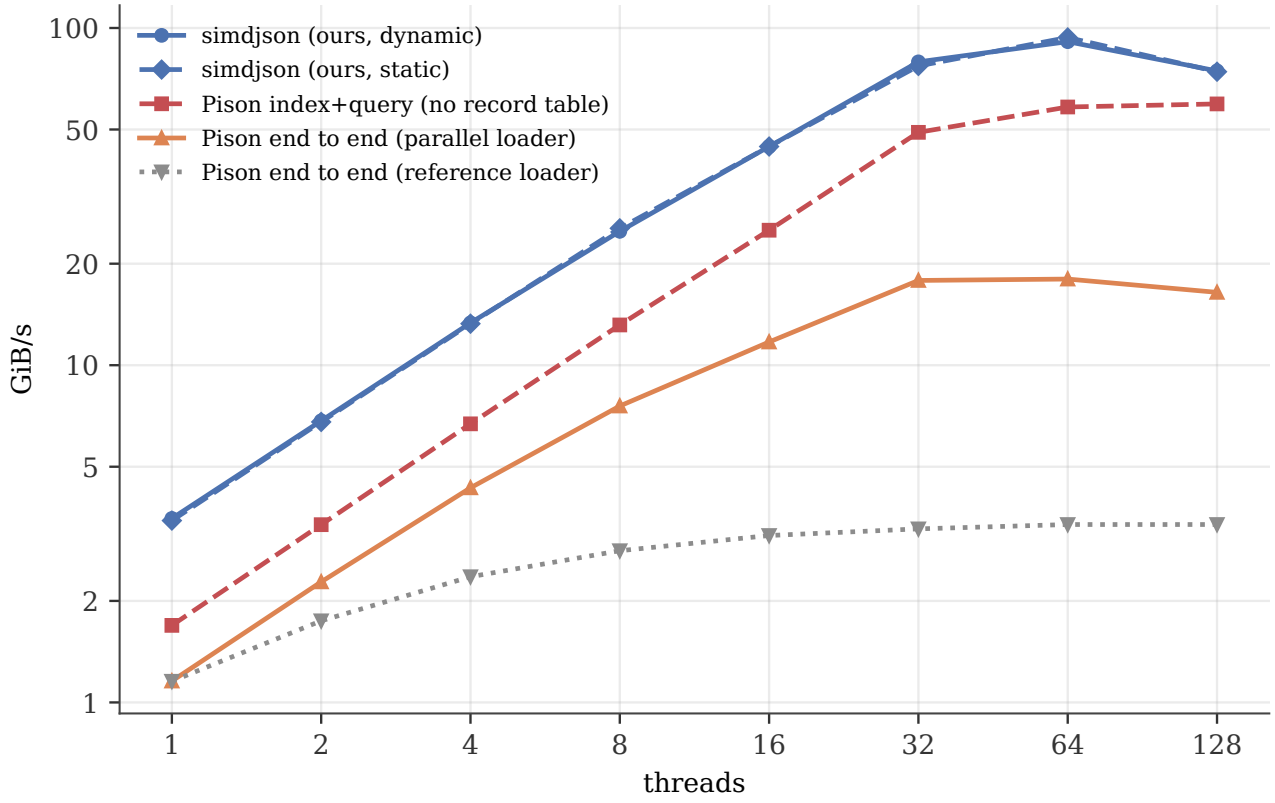


Fig. 7 Throughput against thread count on the tweet stream on 6548N (0.8 GiB, 155,489 documents), log–log. The two upper curves are our parser, solid with dynamic assignment and dashed with static assignment (Section 5.10); the next is Pison's index and query without its record table; the two lower curves are Pison measured end to end, with our parallel loader and with its serial reference loader

At their peaks the throughput gap narrows—we lead yyjson by 1.30 to 2.27 times. At each engine's own best configuration we spend 0.40 to 0.71 CPU seconds per gibibyte; yyjson spends 1.00 to 1.55, Boost.JSON 1.44 to 4.34, RapidJSON 2.87 to 3.62, and nlohmann/json 10.1 to 20.2. That is between 1.8 and 3.5 times our cost per gibibyte for yyjson, and up to 39 times for nlohmann/json.

Because the DOM parsers are slower per thread, they peak at 128 threads (except yyjson on Google Maps and Wikipedia, at 64) where we peak at 64 (except Google Maps, at 32), which invites a comparison at 128 threads alone. It does not change the conclusion: on all six datasets our *degraded* 128-thread throughput still exceeds yyjson's best at any thread count (tweets 75.0 GiB/s against 55.9 GiB/s). The margin is narrowest on NSPL, 54.0 GiB/s against 49.6 GiB/s, where yyjson at 128 threads spends 1.42 CPU seconds per gibibyte against our 0.86. Past the physical core count our parser and yyjson waste work, for reasons explained in Section 5.8.

On the 96-core AMD Zen 5 server, which has one thread per core, Figure 10 shows the same picture. On one thread we are 2.4 to 3.9 times faster than yyjson, against 2.2 to 2.9 on 6548N; at each parser's best thread count, 1.5 to 2.0 times, against 1.30 to 2.27. Our parser peaks at 139 to 227 GiB/s, yyjson at 85 to 136 GiB/s. With no simultaneous multithreading to exploit, the four libraries peak at the full 96 threads on every dataset; ours peaks at 64 or 96 threads.

5.8 Cost per unit of work

Raw throughput rewards using more cores. Figure 11 therefore plots CPU seconds per gibibyte, which is flat for a perfectly scalable system and rises when parallelism buys speed by wasting work.

Our parser spends 0.28 to 0.37 CPU seconds per gibibyte from 1 to 32 threads and Pison's index and query 0.59 to 0.62, so the speedup in that range is genuine. Pison end to end, which includes the record table, rises from 0.86 to 1.59. Beyond 32 threads every curve rises, even where throughput still improves.

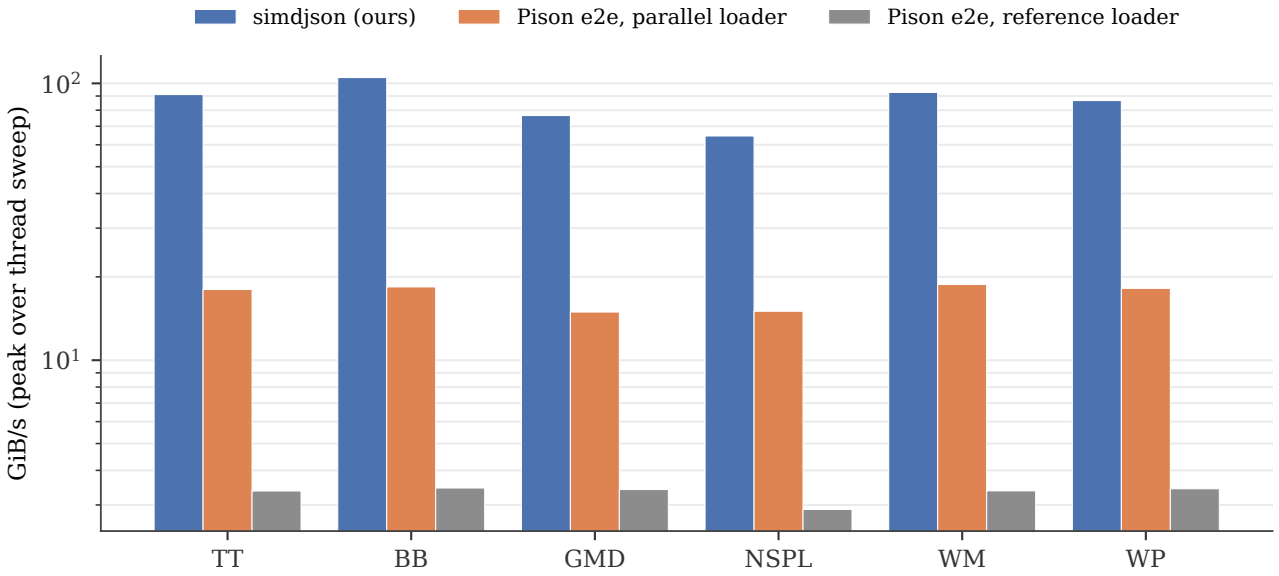


Fig. 8 Peak end-to-end throughput per dataset on 6548N, in-memory buffer in and query results out; note the logarithmic axis. The middle bar gives Pison our parallel, carefully written implementation of the input pass its interface requires

The throughput optimum rarely exceeds the physical core count (5 of 30 machine–dataset peaks do) and can sit well below it (64 of 192 cores on the 9R14 for the tweet stream). The implementation's default, `hardware_concurrency()` workers, is therefore too many on most machines.

An earlier round showed Pison's index-and-query curve dipping below ours at 128 threads. The cause was a bug: `parser::allocate` in simdjson returned early only when the requested capacity *equaled* the current one, so every slice shorter than its predecessor rebuilt every buffer. We contributed the one-line fix upstream. With it, our cost per gibibyte is below Pison's index and query in all but 2 of the 264 configurations (machines, datasets, thread counts), both Google Maps at the full hardware-thread count of a machine with SMT.

Why throughput falls. The main results neither pin threads nor place memory. To separate the causes, we reran the tweet stream on every machine confined to one NUMA node, with memory interleaved over all nodes, and, at the physical core count, with one hardware thread per core against both hardware threads on half the cores. These runs are a separate session that samples more thread counts (adding 96 and 144 on the 9R14), so their peaks differ from Table 7 by up to 8 %. Machines with one memory node barely degrade: the 9R45 scales monotonically to 188 GiB/s, and the 8488C holds 105 GiB/s on all its hardware threads. SMT is minor: packing workers onto half the cores changes throughput by at most a tenth (−10 %, 5 % and −10 % on the 8488C, 6975P-C and 6548N).

Memory topology matters more. The two-socket 9R14 peaks at 114 GiB/s on 96 threads and falls to 75.8 GiB/s on 192, yet one socket alone reaches 128 GiB/s, more than the whole machine at any thread count. Interleaving memory does not help (115 GiB/s at best), and neither does avoiding the shared slice counter: static assignment falls 32 % from its peak, against 34 % for dynamic. Tracing each worker shows the main cause: a call creates its workers one after another, about 44 µs apart, so with 192 workers the last one starts 8.4 ms in, after the parsing is over, and 86 of them never claim a slice. Past the peak, each added worker costs more creation time than it saves. The single socket is also the fair comparison for the Zen 5 part: 188 GiB/s against 128 GiB/s is a factor of 1.47, close to the 1.39 of the single-threaded rates. The dual-socket 6548N behaves similarly: its whole-machine peak (93.2 GiB/s) matches one socket alone (92.9 GiB/s).

All engines run unpinned and create their threads the same way, so the falloff is a property of the machines and of how these implementations create their threads. Pison's index-plus-query workload edges ahead of ours in throughput in only 4 machine–dataset pairs (Walmart on the 6975P-C; tweets, Best Buy and Walmart on the 9R14), each at the full thread count, by at most 4 %, while still costing more CPU seconds per gibibyte.

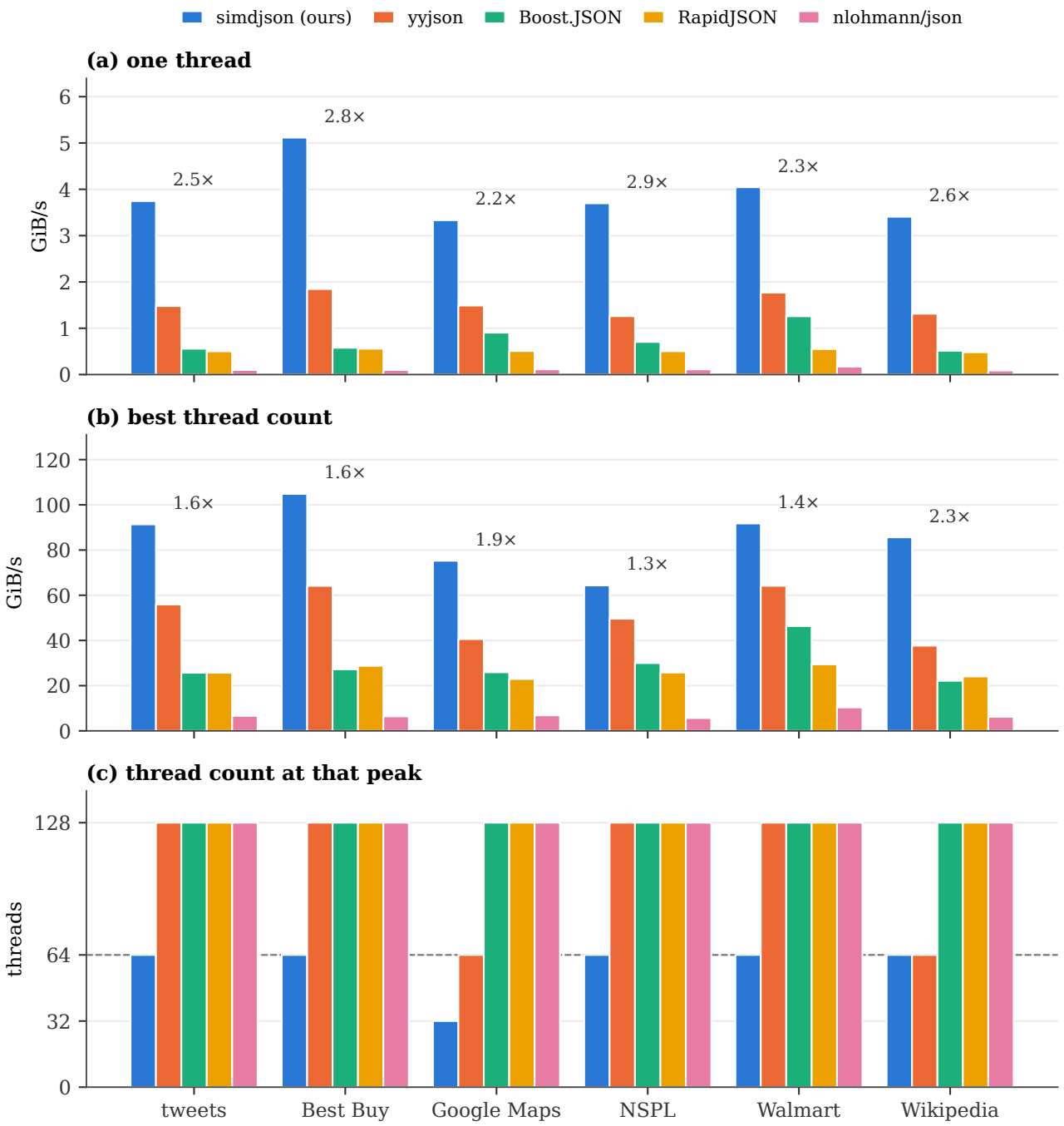


Fig. 9 Our parser against the tree-building parsers on 6548N, decode workload: (a) throughput on one thread, (b) throughput at each parser's best thread count, and (c) that best thread count (the dashed line marks the physical cores). The factor above each throughput group is our advantage over the best of the other four parsers on that dataset; Table 8 gives the CPU cost

5.9 Slice size

The slice size trades per-slice overhead against load balance. Figure 12 shows the tradeoff on the reference machine.

Peak query throughput over the thread sweep is 91.4 GiB/s, 92.3 GiB/s, 90.0 GiB/s, 80.4 GiB/s, 73.1 GiB/s, 34.1 GiB/s and 27.3 GiB/s for slices of 64 KiB, 128 KiB, 256 KiB, 512 KiB, 1,024 KiB, 4,096 KiB and 16,384 KiB. Peak throughput is approximately flat from 64 KiB to 256 KiB (within 2.5 %), so this sweep does not identify a unique optimum; beyond 256 KiB throughput falls, and at 16 MiB a 0.8 GiB input yields only about 51 slices, so most of the

Table 8 Decode workloads: serial throughput and instructions per byte, then peak parallel throughput (worker count in parentheses) with CPU cost, on 6548N

(a) tweets, Best Buy, Google Maps directions

	TT		BB		GMD	
Parser	GiB/s	i/B	GiB/s	i/B	GiB/s	i/B
ours (On-Demand)	3.75	2.6	5.12	1.6	3.33	2.7
yyjson	1.48	6.2	1.85	6.2	1.49	6.3
Boost.JSON	0.56	21.6	0.58	22.0	0.90	16.1
RapidJSON	0.50	31.9	0.56	32.0	0.51	33.3
nlohmann/json	0.10	98.1	0.10	106	0.11	102
	GiB/s (thr)	CPU-s/GiB	GiB/s (thr)	CPU-s/GiB	GiB/s (thr)	CPU-s/GiB
ours (On-Demand)	91.4 (64)	0.57	105 (64)	0.49	75.3 (32)	0.40
yyjson	55.9 (128)	1.18	64.2 (128)	1.00	40.6 (64)	1.38
Boost.JSON	25.7 (128)	3.07	27.2 (128)	2.94	25.9 (128)	2.94
RapidJSON	25.7 (128)	3.13	28.8 (128)	3.17	23.0 (128)	3.39
nlohmann/json	6.63 (128)	16.3	6.42 (128)	17.5	6.86 (128)	15.6

(b) NSPL, Walmart, Wikipedia

	NSPL		WM		WP	
Parser	GiB/s	i/B	GiB/s	i/B	GiB/s	i/B
ours (On-Demand)	3.70	2.7	4.04	2.0	3.41	2.8
yyjson	1.26	8.1	1.77	4.8	1.31	7.7
Boost.JSON	0.70	22.9	1.26	9.3	0.51	28.7
RapidJSON	0.50	34.6	0.55	30.9	0.48	34.0
nlohmann/json	0.11	103	0.17	69.9	0.09	113
	GiB/s (thr)	CPU-s/GiB	GiB/s (thr)	CPU-s/GiB	GiB/s (thr)	CPU-s/GiB
ours (On-Demand)	64.3 (64)	0.71	91.8 (64)	0.57	85.6 (64)	0.63
yyjson	49.6 (128)	1.42	64.2 (128)	1.02	37.7 (64)	1.55
Boost.JSON	30.0 (128)	2.68	46.4 (128)	1.44	22.1 (128)	4.34
RapidJSON	25.8 (128)	3.43	29.4 (128)	2.87	24.1 (128)	3.62
nlohmann/json	5.68 (128)	20.2	10.4 (128)	10.1	6.20 (128)	18.8

workers receive one slice or none and the straggler sets the total time. With a single worker, slicing changes the rate in either direction, depending on the machine: with 64 KiB slices, one worker runs between 17 % slower and 18 % faster than the serial parser on the tweet stream, and with 1 MiB slices between 3 % slower and 22 % faster.

Different record-size distributions motivate a corpus-dependent choice. What matters is how many bytes lie in records longer than the slice. Such a record cannot be split: the slice where it starts takes all of it, and every claim that lands inside it comes back empty. The more bytes lie in such records, the less the slice size controls the work per claim. We therefore select the smallest candidate $S \in \{64, 128, 256, 512, 1024\}$ KiB such that at most 1 % of the corpus bytes lie in records longer than S, falling back to 1,024 KiB if no candidate qualifies. This is a practical heuristic, not a proven optimum; it requires one scan of the corpus, performed before any timed region. It selects 64 KiB for five of the six Pison corpora and 1 MiB for Google Maps, whose records average 239 KiB and where 5.2 % of the bytes lie in records longer than 512 KiB, so only the 1 MiB candidate meets the 1 % threshold (Section 5.2). The results for the six datasets use these derived sizes under both assignments.

5.10 Static versus dynamic assignment

The design leaves the assignment of slices to workers open (Section 4.1): a worker may claim the next slice from a shared counter (*dynamic*) or own a contiguous range (*static*). We compared the two at equal slice size and equal worker count on every machine and corpus, at three thread counts per machine, so that only the assignment differs: 180 pairs at 64 KiB and 256 KiB across the six corpora, plus 15 for Google Maps at 1 MiB, 195 in all (Table 9). The sizes are targeted rather than a full grid: 64 KiB is what the heuristic selects for five of the corpora and 256 KiB is a larger comparison point, while Google Maps also gets 1 MiB because that is the size the heuristic selects for it.

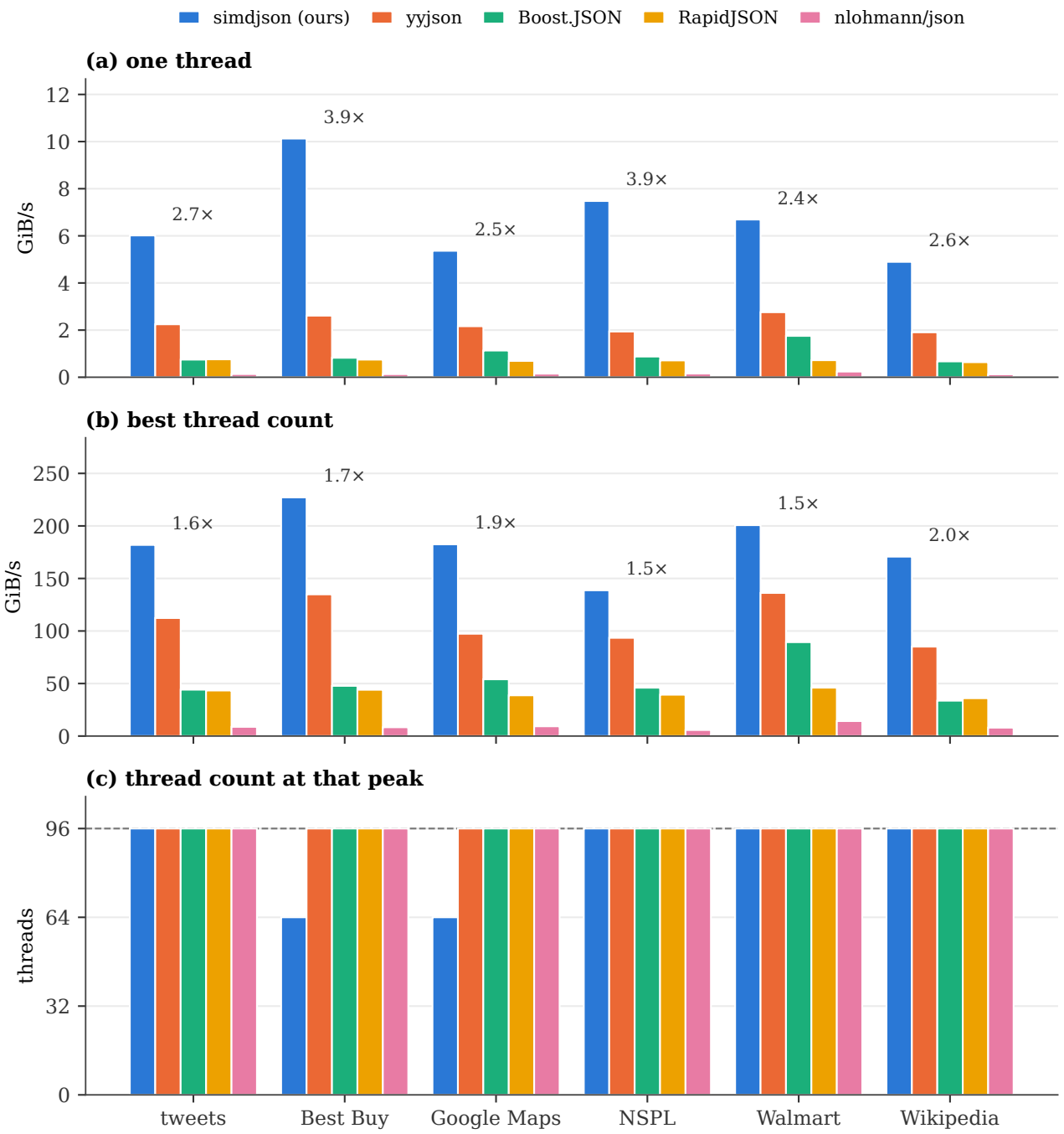


Fig. 10 As Figure 9, on the 96-core AMD EPYC 9R45 (Zen 5), which has one thread per core: (a) throughput on one thread, (b) throughput at each parser's best thread count, and (c) that best thread count (the dashed line marks the physical cores)

Outside Google Maps the two policies are close: over 150 pairs the dynamic/static throughput ratio spans 0.79–1.25, with a median of 1.01, and 132 of the 150 lie within 10 % of each other. Neither dominates. Dynamic is faster in 88 pairs and static in 62; the median ratio is 0.99 at 64 KiB and 1.03 at 256 KiB.

Google Maps is the exception: the effect is most pronounced at the smallest slice. Its records average 239 KiB, so a 64 KiB slice usually falls inside a single record. There the ratio spans 0.24–1.11 (median 0.84), static is faster in 14 of the 15 pairs, and on the 6975P-C machine it is up to 4.1 times faster (at 128 threads). The gap shrinks as the slice approaches the record size—0.45–

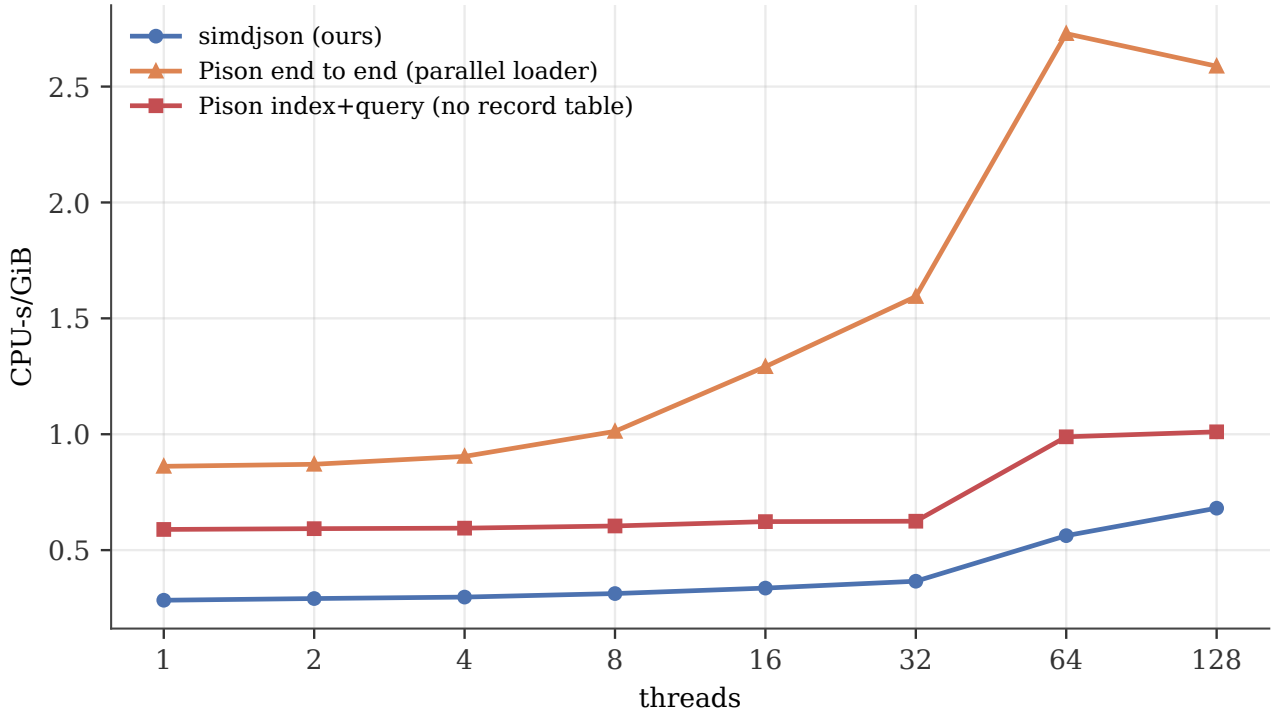


Fig. 11 CPU seconds per gibibyte against thread count on 6548N (tweet stream); lower is better. The two lower curves are nearly flat to 32 threads, so the speedup to that point is real rather than bought; the upper curve includes Pison's record table and rises over the same range. The middle curve is Pison *without* the record table it cannot skip; the upper one is the like-for-like comparison with ours

Table 9 Static and dynamic assignment at equal slice size and worker count. Entries are ranges of dynamic/static throughput ratios (above 1 favors dynamic), taken over the corpora and thread counts; Google Maps is shown separately because its mean record size (239 KiB) is large relative to the smaller slices. Best of ten runs

	Other corpora		Google Maps		
Machine	64 KiB	256 KiB	64 KiB	256 KiB	1 MiB
6548N	0.79–1.06	0.94–1.16	0.77–0.87	0.98–1.00	0.96–1.06
6975P-C	0.86–1.03	0.92–1.17	0.24–0.47	0.45–0.93	1.01–1.08
8488C	0.93–1.12	0.99–1.14	0.83–1.11	0.96–1.15	0.96–1.17
9R45	0.94–1.06	0.96–1.10	0.77–0.90	0.91–0.99	0.99–1.08
9R14	1.00–1.10	1.05–1.25	0.86–0.87	0.95–0.99	1.10–1.14

1.15 at 256 KiB—and is smallest at 1 MiB (0.96–1.17). The pattern is consistent with the cost of claiming: when records are longer than the slice, most claims come back empty, so a dynamic worker spends much of its time contending for the shared counter only to find no work, whereas a static worker passes over the empty slices in its range without coordinating with anyone.

There is therefore no universal winner at a fixed slice size: the ranking depends on how the slice size compares with the record size. For Google Maps the heuristic selects 1 MiB, where the two policies are within about 17 % of each other, and the small-slice corner where static wins is not a size it chooses.

The full sweep, every thread count at the derived slice sizes, shows the same balance with a machine-dependent tilt (Table 7 and Figure 7). Over 30 machine–dataset pairs on 5 machines, the ratio of the two peaks, dynamic over static, spans 0.84 to 1.19: dynamic is ahead in 12 pairs and static in 18. Only the dual-socket 9R14 favors dynamic on every dataset (1.01 to 1.19). Static is mostly ahead on the 6975P-C (0.84 to 1.04) and the 8488C (0.89 to 1.05), and slightly on the 9R45 (0.93 to 1.02) and the 6548N (0.93 to 1.06).

Thread creation explains the 9R14 (Section 5.8). Its 96 workers start over the first 4.2 ms of the call. Under static assignment every worker has the same share of the input however late it starts, so each finishes in

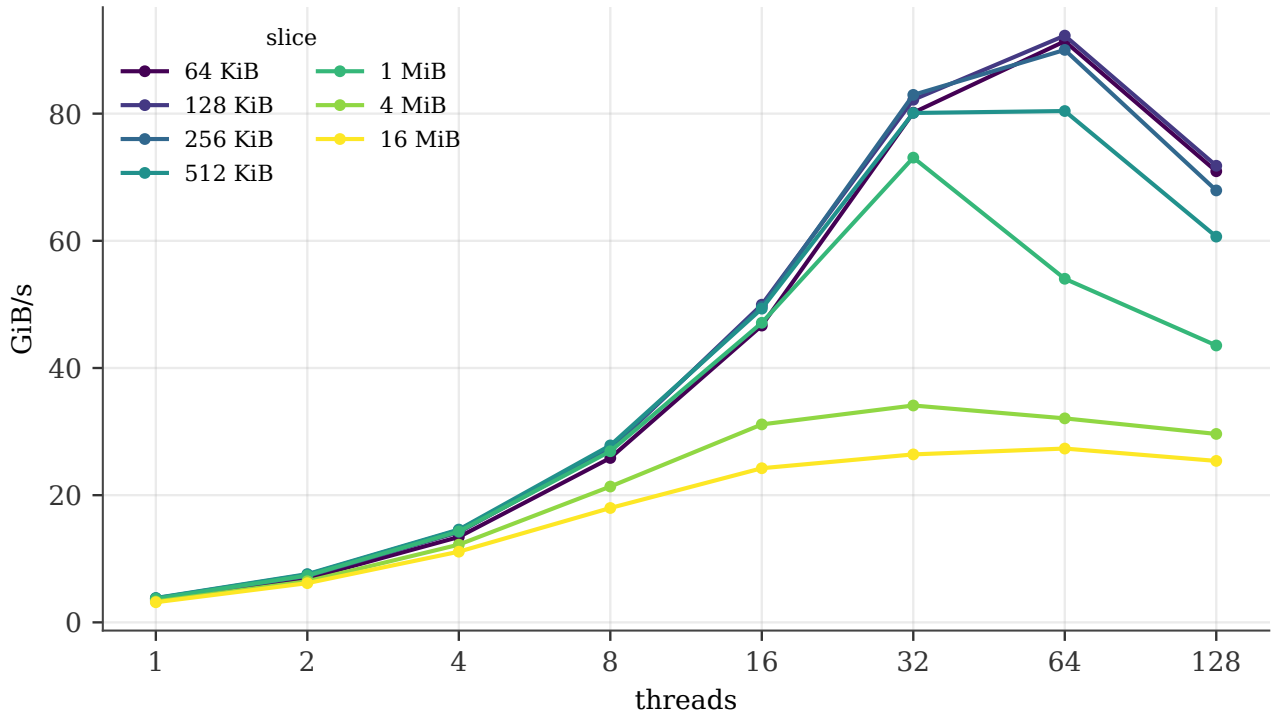


Fig. 12 Query throughput against thread count for several slice sizes on the reference machine (tweet stream). Throughput is flat from 64 KiB to 256 KiB and then falls as slices grow; at 16 MiB there are too few slices to occupy the workers

the order it started (the correlation of start and finish times is 0.99), and the workers spend 33 % of their time waiting for the last one: the call takes 7.4 ms. Under dynamic assignment the workers that start early claim more slices, idle time falls to 2 %, and the call takes 6.5 ms; at 256 KiB slices that is 120 GiB/s against 106 GiB/s. Contiguity is not the difference: dynamic workers that claim 16 adjacent slices at a time reach 118 GiB/s. Dynamic assignment is the more robust policy when workers start at different times. We traced only the 9R14; on the other machines the two policies are close. On the tweet stream, static assignment peaks at 196 GiB/s on the 9R45 against 184 GiB/s for dynamic, and at 114 GiB/s against 104 GiB/s on the 6975P-C; on the 9R14 it is the other way round (106 GiB/s against 112 GiB/s). Google Maps, at 1 MiB slices, is within a few percent everywhere (1.01 to 1.06). A system that must pick one policy without measuring loses at most about a fifth either way; one that can measure should try both on its own hardware.

5.11 A case study on OpenAlex

The six corpora above all come from Pison's benchmark, and their records are short: the longest, a Wikipedia entity, reaches 680 KiB, still under the 1 MiB batch of the serial path. To exercise the slicing rule on records that exceed that batch, we add a partition of the OpenAlex authors snapshot [5]: the 232,330 authors updated on 2026-03-30, stored as NDJSON and copied verbatim (6.0 GiB, mean record 27 KiB, longest 1.3 MiB), queried for each author's display name and works count.

Its long records expose the batch-size constraint directly. The batch must exceed the longest document (Section 2), so the harness raises the serial baseline's 1 MiB batch to just above the longest record, 1.3 MiB here. OpenAlex is the only corpus compared with Pison that needs this: its longest record exceeds the 680 KiB maximum of the other six. The slice parser needs no such tuning, because a slice always ends at a document boundary (Section 4.1).

The long tail also changes the slice choice. The pre-timing scan finds that 5.3 % of the bytes lie in records longer than 128 KiB, but only 0.7 % in records longer than 256 KiB, so the rule selects 256 KiB rather than the 64 KiB selected for five of the six Pison corpora. We sweep the size under static assignment, which isolates it from counter traffic and from which worker claims which slice, and run dynamic, the policy of the main results, at the selected size. That choice matters: at 64 threads on the reference machine (Table 10) the sweep rises from 90.2 GiB/s at 64 KiB to a peak of 104 GiB/s at 256 KiB—

Table 10 OpenAlex authors (232,330 records, 6.0 GiB): simdjson-parallel slice query throughput on 6548N; static sweep and dynamic assignment at 256 KiB; best of ten runs

Assignment	Slice (KiB)	GiB/s	
		64 threads	128 threads
static	64	90.2	71.1
static	128	99.9	75.9
static	256	104	67.4
static	512	97.7	64.1
static	1024	67.5	53.1
dynamic	256	108	66.2

the size the rule selects, and 15 % above the fixed 64 KiB baseline—and falls to 67.5 GiB/s at 1 MiB. Dynamic assignment reaches 108 GiB/s at 256 KiB, 3.8 % above static, with spreads of up to 5.4 % between repetitions, so the two policies are close here. A run that is given no explicit size derives the same 256 KiB and agrees with the explicit run to within 2.9 %.

5.12 Where the structural index wins

We use the bulky form of the tweet dataset (0.79 GiB as one JSON record), slightly prepared so that Pison can read it, with Pison's own bulky query: `$[*].user.lang` and `$[*].lang`.[1] Both engines return the same 300,270 values; the simdjson parser reads the array as one document.

Pison's parallel index construction runs at 2.91 GiB/s on one thread and 26.1 GiB/s on 64, a 9.0× speedup that a serial parser cannot match: simdjson's stage 1 over the same document runs at 4.49 GiB/s. Answering the query is another matter. Pison's bitmap iterator walks the finished index on one thread, so its index and query together run at 1.59 GiB/s on one thread and only 2.94 GiB/s on 64, a 1.8× speedup, against 2.55 GiB/s for the single-threaded on-demand parser. Pison needs 16 threads to answer this query faster than one simdjson thread.

5.13 Against a GPU parser

The cuJSON paper [6] reports one end-to-end measurement (its Section 8.7): from a day and a half of GitHub Archive events (1.37 GB as JSON lines), return the repository name of every push, `$[?(@.type=="PushEvent")].repo.name`. On an NVIDIA A100 it takes 323 ms, copying the input to the GPU and the results back included. We ran the same query on the same file, taken from cuJSON's dataset folder, on 6548N with the input in memory, as everywhere in this paper. Every configuration extracts the same 289,117 names—the count cuJSON reports—and agrees on their hash. Table 11 gives the results.

Our parser answers the query in 18.8 ms at 32 threads—67.8 GiB/s, 17 times faster than cuJSON's 323 ms (3.9 GiB/s). A single thread nearly matches the GPU: simdjson's own stream parser takes 342 ms, and our parser on one thread 334 ms. With its second thread, which runs stage 1 of the next batch, the stream parser takes 196 ms, faster than the GPU. The comparison is not exact: cuJSON's time includes the transfers to and from the GPU, which its authors identify as its costliest stage, whereas our input is already in memory. Without them the gap narrows, but it does not close: cuJSON's GPU kernels alone take 24 to 39 ms to parse the JSON-lines forms of the datasets of Table 3, each about 1 GB (its Figure 11), before any query is answered—more than our end-to-end time on a larger file. The table uses static assignment; dynamic assignment reaches 23.4 ms at 64 threads.

6 Conclusion

NDJSON and RFC 7464 streams are parallel by construction, and their delimiter cannot appear inside a value. A parser that exploits both—cutting the input with two `memchr` calls per slice, giving each worker its own parser and output buffer, and decoding only the values a query names—reaches 142 to 229 GiB/s on a 96-core AMD Zen 5 server with dynamic assignment, depending on the dataset, and between 24 and 34 times the serial parser's rate on the tweet stream at each machine's best worker count (64 or 96), with CPU cost per byte that stays flat up to a few dozen threads. That is several times the throughput reported for GPU JSON parsers on the same datasets, even counting only their GPU kernel time (Table 1).

Against Pison, the state of the art in parallel structural indexing on CPUs, measured on Pison's corpus and its published queries (with an adapted query for NSPL), verified to produce the same match counts and value hashes on every machine, the on-demand parser is 1.8 to 4.4 times faster single-threaded and retires fewer instructions per byte in every dataset and machine combination. The gap widens end to end, to between 4.3 and 17.0 times: a leveled bitmap index must be built over

[1] The published record wraps its 155,489 tweets in an object and separates them with carriage returns and line feeds. Below the top level, Pison's iterator has room for only 125,000 element positions, and before a value it skips only spaces and line feeds. Both engines are therefore given the tweet array itself, with each carriage return replaced by a space, which leaves the document's meaning unchanged. The preparation is outside the timed region.

Table 11 cuJSON's end-to-end query, `$[?(@.type=="PushEvent")].repo.name`, over its GitHub Archive file (1.37 GB): cuJSON's published time on an NVIDIA A100, transfers included, against simdjson on 6548N with the input in memory. Mean of 10 runs; every configuration extracts the same 289,117 names and agrees on their hash. The last column is cuJSON's time divided by ours

	Threads	Time (ms)	GiB/s	Speedup
cuJSON (A100, published)	–	323	3.94	1.0
simdjson, one JSON document	1	354	3.60	0.9
simdjson, stream (`iterate_many`)	1	342	3.72	0.9
simdjson, stream, threaded	2	196	6.48	1.6
ours, sliced	1	334	3.81	1.0
ours, sliced	8	47.2	27.0	6.8
ours, sliced	32	18.8	67.8	17.2

the input, and Pison's indexer covers only complete 64-byte blocks, so its loader pads each record to a 64-byte multiple in a contiguous buffer and the input must be copied before indexing can begin. Pison's parallel index construction remains the better tool for a single enormous document, though even there it needs 16 threads to answer its own query faster than one simdjson thread (Section 5.12).

Against the tree-building parsers in common use, our parser retires 1.6 to 2.8 instructions per input byte where yyjson retires 4.8 to 8.1 and nlohmann/json 69.9 to 113, because a DOM must be materialized in full before the program can read any of it. Placed under our slicing framework, they remain slower: at each library's best thread count we are 1.30 to 2.27 times faster than yyjson, the fastest of them. Our CPU time per gibibyte is 1.8 to 3.5 times lower than yyjson's, and up to 39 times lower than nlohmann/json's.

These rates are on inputs of about a gibibyte, with the worker threads created inside each timed call (Section 5.1). When the parsing takes only a few milliseconds, thread creation dominates (Section 5.8).

Statements and Declarations

Competing interests

The authors declare no competing interests.

Funding

This work was supported by the Natural Sciences and Engineering Research Council of Canada, Grant Number: RGPIN-2024-03787. The first author is supported by a postdoctoral grant from Fonds de recherche du Québec, `https://doi.org/10.69777/361128`.

Data availability

The benchmark is available at `https://github.com/simdjson/jsonstreambench` [4]. It pins simdjson 5.0.6 (`98464972`), Pison, and the counter library at configure time. The main results were produced at revision `e75c7e3` (on the reference machine) and its parent `154de3d` (on the AWS machines), which differ only in the default simdjson pin. The scheduling study (Section 5.10) and the OpenAlex study were produced at revision `40677e3`; since these studies set the slice size and the assignment explicitly, the only change since the main revisions that affects them is how a slice that starts inside a long document finds out that it is empty. The comparison with cuJSON (Section 5.13) uses its `cujson_baseline` directory at revision `9a41769`. The measurements of Appendix A come from `https://github.com/lemire/ndjson_bench` at revision `074a2f4`. The OpenAlex authors partition is archived at Zenodo under CC0 (`https://doi.org/10.5281/zenodo.21813521`).

Author contributions

Jaël Champagne Gareau: conceptualization; investigation; software; experimentation; writing—review and editing. Daniel Lemire: conceptualization; software; validation; experimentation; data analysis; writing—original draft; writing—review and editing.

A Data systems

Apache Arrow, DuckDB and Polars already split NDJSON at line feeds and parse the pieces on several threads (Section 3). We ran the six queries with all three on 6548N, through their own readers: Arrow's `pyarrow.json.read_json`, DuckDB's `read_json` and Polars' `read_ndjson`, called from Python. Each is given an explicit schema naming only the queried fields, which avoids schema inference. The timed region goes from the input to the extracted values in memory. Arrow and Polars read from an in-memory buffer; DuckDB reads from a path,

Table 12 The six queries with three data systems that read NDJSON in parallel, on 6548N: DuckDB 2.0.0, Polars 2.1.0 and Apache Arrow (pyarrow 26.0.0), each given an explicit schema naming only the queried fields, against our parser's decode workload (Table 8). Throughput in GiB/s on one thread, then at the best thread count (in parentheses); best of ten runs. Arrow cannot read the positional rows of NSPL

	TT	BB	GMD	NSPL	WM	WP
one thread						
ours	3.75	5.12	3.33	3.70	4.04	3.41
DuckDB	1.07	1.12	0.90	0.36	1.45	1.00
Polars	0.45	0.36	0.26	0.13	0.87	0.28
Arrow	0.42	0.47	0.38	–	0.55	0.39
best thread count						
ours	91.4 (64)	105 (64)	75.3 (32)	64.3 (64)	91.8 (64)	85.6 (64)
DuckDB	11.6 (32)	11.7 (32)	11.3 (64)	5.35 (32)	16.9 (32)	12.2 (32)
Polars	3.51 (16)	3.67 (16)	2.17 (64)	5.32 (128)	3.15 (8)	3.24 (128)
Arrow	24.6 (128)	16.8 (128)	3.23 (16)	–	29.1 (128)	20.7 (128)

so its files are in a memory-backed file system (`/dev/shm`). The NSPL records are positional arrays, which the NDJSON readers of Arrow and Polars do not accept. For NSPL, DuckDB therefore reads each line as a JSON value and extracts the two cells with `json_extract_string`, Polars reads each line as a string and applies its JSONPath function, and Arrow is left out. Every system returns the match counts of Table 3, and the three agree on a checksum of the values.

Each reader parses with a different JSON library, as we confirmed in the binaries. Arrow (pyarrow 26.0.0) cuts each block at a line feed and parses it with the event-based (SAX) reader of RapidJSON, whose events go straight to Arrow's column builders. DuckDB (2.0.0) parses each record into a document tree with yyjson and converts the requested fields into its vectors; `json_extract_string`, used for NSPL, also parses each line in full with yyjson. Polars (2.1.0) parses each line into a value tree with simd-json, a Rust library derived from simdjson, and fills its columns from that tree; for NSPL, its JSONPath function evaluates the path with the jsonpath_lib crate over serde_json values.

Table 12 compares them with our decode workload, which, like them, unescapes the strings and converts the numbers it returns. On one thread we are 2.8 to 10.2 times faster than the fastest of the three on each dataset. At each system's best thread count the gap is 3.2 to 12.0 times: Arrow reaches at most 29.1 GiB/s, DuckDB 16.9 GiB/s and Polars 5.3 GiB/s. All three split the input as we do, so the difference lies in what runs on each piece: they parse every record with a tree-building or event-based parser (RapidJSON, yyjson, simd-json) and then build table columns from the queried fields. These numbers measure what a user of these systems pays for the query, not their parsers alone.

References

1. Apache Arrow: Reading JSON files. `https://arrow.apache.org/docs/cpp/json.html` (2025)
2. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., Zaharia, M.: Spark SQL: Relational data processing in Spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15, pp. 1383–1394. Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2723372.2742797
3. Bray, T.: The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, RFC Editor (2017). https://doi.org/10.17487/RFC8259
4. Champagne Gareau, J., Lemire, D.: jsonstreambench: a fair benchmark for parsing streams of JSON documents. `https://github.com/simdjson/jsonstreambench` (2026)
5. Champagne Gareau, J., Lemire, D.: OpenAlex authors snapshot partition (updated_date=2026-03-30), NDJSON benchmark corpus. `https://doi.org/10.5281/zenodo.21813521` (2026). CC0
6. Gargary, A.V., Loaliyan, S.S., Zhao, Z.: cuJSON: A Highly Parallel JSON Parser for GPUs. In: Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS '26, p. 85–100. Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3760250.3762222
7. Gössner, S., Normington, G., Bormann, C.: JSONPath: Query expressions for JSON. RFC 9535, RFC Editor (2024). https://doi.org/10.17487/RFC9535
8. Jiang, L., Qiu, J., Zhao, Z.: Scalable Structural Index Construction for JSON Analytics. Proceedings of the VLDB Endowment **14**(4), 694–707 (2020). https://doi.org/10.14778/3436905.3436926
9. Jiang, L., Zhao, Z.: JSONSki: Streaming semi-structured data with bit-parallel fast-forwarding. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, p. 200–211. Association for Computing Machinery, New York, NY, USA (2022). URL `https://doi.org/10.1145/3503222.3507719`
10. Keiser, J., Lemire, D.: Validating UTF-8 in less than one instruction per byte. Software: Practice and Experience **51**(5), 950–964 (2021). https://doi.org/10.1002/spe.2920
11. Keiser, J., Lemire, D.: On-demand JSON: A better way to parse documents? Software: Practice and Experience **54**(6), 1074–1086 (2024). https://doi.org/10.1002/spe.3313
12. Kuiper, L.: Shredding deeply nested JSON, one vector at a time. `https://duckdb.org/2023/03/03/json.html` (2023)
13. Langdale, G., Lemire, D.: Parsing gigabytes of JSON per second. The VLDB Journal **28**(6), 941–960 (2019). https://doi.org/10.1007/s00778-019-00578-5
14. Lemire, D.: counters: a C++ library to access hardware performance counters. `https://github.com/lemire/counters` (2024)

15. Li, Y., Katsipoulakis, N.R., Chandramouli, B., Goldstein, J., Kossmann, D.: Mison: A fast JSON parser for data analytics. Proceedings of the VLDB Endowment **10**(10), 1118–1129 (2017). https://doi.org/10.14778/3115404.3115416
16. Muła, W.: SIMD-ized faster parse of IPv4 addresses. `http://0x80.pl/notesen/2023-04-09-faster-parse-ipv4.html` (2023). Accessed 2026-08-02
17. Muła, W., Lemire, D.: Faster base64 encoding and decoding using AVX2 instructions. ACM Transactions on the Web **12**(3), 1–26 (2018). https://doi.org/10.1145/3132709
18. Palkar, S., Abuzaid, F., Bailis, P., Zaharia, M.: Filter before you parse: faster analytics on raw data with Sparser. Proceedings of the VLDB Endowment **11**(11), 1576–1589 (2018). https://doi.org/10.14778/3236187.3236207
19. Pinnecke, M.: Karbonit: Home of the karbonit key-value store. `https://github.com/pinnecke/karbonit` (2019)
20. Raasveldt, M., Mühleisen, H.: DuckDB: an embeddable analytical database. In: Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19, pp. 1981–1984. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3299869.3320212
21. Schulze, S.: Fast parsing of NDJSON files in main memory databases. Bachelor's thesis, University of Magdeburg (2020)
22. Stehle, E., Jacobsen, H.A.: ParPaRaw: Massively parallel parsing of delimiter-separated raw data. Proceedings of the VLDB Endowment **13**(5), 616–628 (2020). https://doi.org/10.14778/3377369.3377372
23. Talluri, S., Di Donato, G.W., Danelutti, L., Vlaswinkel, K.R., Arnaboldi, M., Delamare, A., Santambrogio, M.D., Bonetta, D.: GpJSON: High-performance JSON data processing on GPUs. Proceedings of the VLDB Endowment **18**(9), 3216–3229 (2025). https://doi.org/10.14778/3746405.3746439
24. Vink, R., et al.: Polars: dataframes powered by a multithreaded, vectorized query engine. `https://github.com/pola-rs/polars` (2025)
25. Williams, N.: JavaScript object notation (JSON) text sequences. RFC 7464, RFC Editor (2015). https://doi.org/10.17487/RFC7464