RESEARCH ARTICLE

On-Demand JSON: A Better Way to Parse Documents?

John Keiser[1] | **Daniel Lemire**[1*]

[1]DOT-Lab Research Center, Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada

Correspondence
Daniel Lemire, DOT-Lab Research Center, Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada
Email: lemire@gmail.com

Funding information
Natural Sciences and Engineering Research Council of Canada, Grant Number: RGPIN-2017-03910

JSON is a popular standard for data interchange on the Internet. Ingesting JSON documents can be a performance bottleneck. A popular parsing strategy consists in converting the input text into a tree-based data structure—sometimes called a Document Object Model or DOM. We designed and implemented a novel JSON parsing interface—called On-Demand—that appears to the programmer like a conventional DOM-based approach. However, the underlying implementation is a pointer iterating through the content, only materializing the results (objects, arrays, strings, numbers) lazily. On recent commodity processors, an implementation of our approach provides superior performance in multiple benchmarks. To ensure reproducibility, our work is freely available as open source software. Several systems use On Demand: e.g., Apache Doris, the Node.js JavaScript runtime, Milvus, and Velox.

KEYWORDS
JSON, Semi-Structured Documents, Text Processing, SIMD Instructions, Performance

1 | INTRODUCTION

There are several text-based semi-structured document formats (e.g., HTML, XML, JSON). JSON is maybe the most popular format online for data interchange [1]. Several database systems such as CouchDB, RethinkDB, MongoDB, SimpleDB and JSON Tiles [2] use JSON as their primary exchange format.

A JSON document must be stored in a valid Unicode (UTF-8) string. The JSON syntax is nearly a strict subset

of the popular programming language JavaScript. It has four primitive types (string, number, Boolean, null) and two composed types (arrays and objects). An object takes the form of a series of key-value pairs between braces where keys are strings and values can be primitive or composed types (e.g., `{"name":"Jack","age":22}`). An array is a list of comma-separated values (either primitive of composed) between brackets (e.g., `[1,"abc",null]`). The JSON specification has six *structural characters* ('[', '{', ']', '}', ':', ',') to delimit the location and structure of objects and arrays.

Programmers rarely work directly on text-based semi-structured document formats: they prefer to work with a software interface—a *parser*—providing automated validation and text-to-data conversion. There are two popular general-purpose parsing strategies [3]:

- Many parsers process the input document (e.g., JSON, HTML, XML) immediately into an in-memory data structure. Most web browsers use such an approach under the name Document Object Model (DOM): web pages (HTML) are loaded in memory into an ordered tree, accessible as a data structure from a programming language such as JavaScript [4]. Fig. 1 illustrates how a JSON document might be viewed as an ordered tree. A DOM-like approach is convenient, but it requires that the original document be materialized in memory. If the programmer is only interested in a subset of the document, they must still construct the whole tree. Further, even when the programmer needs to ingest the whole document, they may need to copy the data into their own data structures and data types. Hence the materialization of the tree might be unnecessary and wasteful.
- There are also event-based or *streaming* strategies. The document is processed from the beginning, and each newly encountered component is an event: e.g., the beginning of a string, the beginning of an array, the end of an array, etc. In some instances, the programmer may provide functions corresponding to each event. E.g., a programmer might have a function that is triggered each time a string is encountered. One of the most popular families of such parsers might be Simple API for XML (SAX) [5]. Streaming approaches can be efficient: if the programmer only needs to capture part of the input document, they can ignore everything else. The programmer may also write the data to their own data structures directly, without the need to materialize a full tree in a temporary memory buffer. We may use these efficient strategies for specific tasks such as well-defined queries (e.g., JSONPath [6]). We may also apply them to the deserialisation of specific data structures: a popular framework that serves this purpose in Rust is called serde [7]. For general-purpose tasks, streaming strategies might be challenging to the programmer. For example, the programmer may need to write their own code to track their logical location within the document. Without help, the programmer may end up with streaming code that is not highly efficient.

It is possible to parse JSON at high speed with DOM-like approaches. The simdjson DOM parser [8] can construct an ordered tree at a speed of over 2 GiB s^{-1} on realistic inputs. However, we might be able to go even faster if we skip the in-memory construction of the ordered tree. The conventional approach to avoid the materialization of the document as an in-memory data structure is to adopt a streaming strategy. However, even when it would provide superior performance (e.g., 2× or 3× faster), we believe that many programmers would still prefer the DOM-like approach for convenience. To illustrate, consider the source code needed to load coordinates stored as JSON objects (e.g., {"x":1,"y":2,"z":3}) using the standard (DOM-like) interface of the popular JSON for Modern C++ library:[1]

```
auto root = nlohmann::json::parse(json.data(), json.data() + json.size());
for (auto point : root["coordinates"]) {
  result.emplace_back(json_benchmark::point{point["x"], point["y"], point["z"]});
}
```

[1] https://github.com/nlohmann/json

```
{
  "Width": 800,
  "Height": 600,
  "Title": "View from my room",
  "Url": "http://ex.com/img.png",
  "Private": false,
  "Thumbnail": {
    "Url": "http://ex.com/th.png",
    "Height": 125,
    "Width": 100
  },
  "array": [
    116,
    943,
    234
  ],
  "Owner": null
}
```

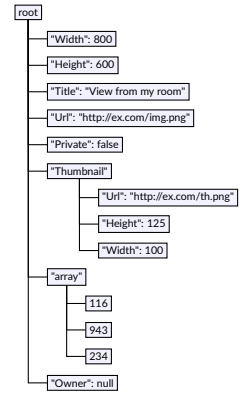


FIGURE 1 JSON example with corresponding tree form

We implemented the same solution using the streaming interface from the same library (JSON for Modern C++), see Fig. 2. For simplicity, we omitted boilerplate code which handles unexpected events. Even so, the streaming interface is relatively complicated, significantly longer (4 lines vs. 40), and requires much effort from the programmer compared to the standard imperative DOM approach. The code might be also more difficult to debug.

And this is a simple example: more complex processing would unavoidably require more code. It might be possible to improve our solution, but an event-based programming approach unavoidably requires the programmer to provide its own approach to track the position in the document and its current state.

For high-efficiency and convenience, we want to offer to the programmer a lazily evaluated tree [9]: if the programmer only needs one number out of a larger document, we allow the programmer to navigate to this one number using as little effort as possible, and to only materialize this one number. When the input document is an array (e.g., `[1,2,3]`), we propose to offer to the programmer what appears to be an array data structure, but is just an iterator over the values in the array. For objects (e.g., `{"one":1, "two":2, "three":3}` in JSON), we iterate over key-value pairs. The parsing of the values is delayed until it is needed: the programmer may choose to skip unneeded values. Furthermore, we seek to provide as much flexibility to the programmer as possible. A given value appearing in JSON as a number may be parsed by the programmer as an integer, a floating-point number, or a string; a string (e.g., `"3.1416"`) can be parsed as number; and so forth. We call this lazy parsing strategy *On-Demand*. We refer to On-Demand as a *front-end*: it is a programming interface for the programmer that serves to abstract much of the complexity necessary for correct and efficient parsing.

An open-source implementation of our proposal (simdjson On-Demand) provides high speed. A long-standing benchmark of several JSON parsers[2] (henceforth Kostya) starts with a 10 000-long array of coordinates in JSON (e.g., `{"coordinates":[{"x":2.0,"y":0.5,"z":0.25}, ...]}`) and requires that the parser sums all 'x' values, all 'y' values and all 'z' values. The On-Demand source code specific to this benchmark is provided in Fig. 3. The results are updated regularly, but the On-Demand approach is often ranked in first position among ≈ 75 competitors. We present some of the results in Table 1 for July 2022. The nearest competitor which does not sacrifice floating-point accuracy is Serde in the Rust programmming language. A highly efficient C++ approach (DAW JSON Link) is practically on par with On-Demand in performance and it offers a functionality similar to Serde, with other specialized compile-time constructions, although without exact number parsing [8, 10].

[2] `https://github.com/kostya/benchmarks`

```
struct Handler : json::json_sax_t {
  size_t k{0}; double buffer[3]; std::vector<point> &result;
  Handler(std::vector<point> &r) : result(r) {}

  bool key(string_t &val) override {
    switch (val[0]) {
    case 'x': k = 0; break;
    case 'y': k = 1; break;
    case 'z': k = 2; break;
    }
    return true;
  }
  bool number_float(number_float_t val, const string_t &s) override {
    buffer[k] = val;
    if (k == 2) {
      result.emplace_back(
          json_benchmark::point{buffer[0], buffer[1], buffer[2]});
      k = 0;
    }
    return true;
  }
  bool number_unsigned(number_unsigned_t val) override {
    buffer[k] = double(val);
    if (k == 2) {
      result.emplace_back(
          json_benchmark::point{buffer[0], buffer[1], buffer[2]});
      k = 0;
    }
    return true;
  }

  bool parse_error(std::size_t position, const std::string &last_token,
                   const json::exception &ex) override {
    return false;
  }
}; // Handler


// ...
Handler handler(result);
json::sax_parse(json.data(), &handler);
```

FIGURE 2 Example of a streaming approach to parse coordinate data in JSON using JSON for Modern C++

Library and Language	Time (s)	Energy Usage (J)
C++/g++ (simdjson On-Demand)	0.067	2.8
Rust (Serde)	0.11	5.6
C++/g++ (RapidJSON)	0.25	6.9
C++/g++ (Boost.JSON)	0.40	16
Go	0.86	36
C++/clang++ (Nlohmann)	1.3	53
Python	1.5	60

TABLE 1 Sample of the kostya JSON benchmark results (July 2022), after excluding techniques that fail to provide exact number parsing. The benchmark is reported to run on an Intel Xeon E-2324G processor (Rocket Lake), using a Debian distribution with GCC 12.1, Go 1.18.2, Rust 1.61 and Python 3.10.4. The test file is 115 GB, the timing excludes the time requires to load the file from disk. Memory usage is computed with from Intel performance counters.

```
auto doc = parser.iterate(json);
for (object point_object : doc["coordinates"]) {
  x += double(point_object["x"]);
  y += double(point_object["y"]);
  z += double(point_object["z"]);
}
```

FIGURE 3 On-Demand C++ source code used by the kostya JSON benchmark

We believe that the Kostya benchmark illustrates that an On-Demand front-end may achieve high performance with a software interface that is both expressive—our code is generic—and accessible: our users' code resembles that of a DOM interface.

2 | RELATED WORK

Our work builds on simdjson [8], specifically on its indexing stage. The simdjson parser was the first standard-compliant JSON parser to process gigabytes of data per second on a single core using commodity processors. Given an input JSON document, the simdjson parser identifies the starting location of all JSON nodes (e.g., numbers, strings, null, true, false, arrays, objects) as well as all JSON structural characters ('[', '{', ']', '}', ':', ','). The parser stores these locations as integer indexes in a separate array. For example, given the array `{"abc":2000}`, we might have the indexes 0, 1, 6, 7, 11. During this indexing stage, we must distinguish the characters that are between quotes, and thus inside a string value, from other characters. For example, the JSON document `"[1,2]\""` is a single string and we only need to locate the first quote character. During the indexing phase, the simdjson parser also checks the Unicode encoding using an efficient SIMD-based algorithm [11]. It also checks other rules such that we only find the ASCII space, the horizontal tab, the line feed, the carriage return and the JSON structural characters ('[', '{', ']', '}', ':', ',')

outside strings. The entirety of the indexing is done in a bit-parallel manner, and leveraging SIMD instructions when available. It reads blocks of 64 bytes one after the other. From the 64 bytes, it generates a 64-bit word which acts as a bitset: the location of each 1-bit indicates the potential location of a JSON structural character or the start of a value. From this 64-bit word, we extract index locations to an array of integers, each 64-byte block generating between zero to 32 index values. The software relies on runtime dispatching, first identifying the features supported by the CPU and then calling on the appropriate function. The simdjson library is used by popular database systems such as ClickHouse and StarRocks.

Though the simdjson parser provides state-of-the-art speed on single-core commodity processors, there are faster JSON parsers on more specialized hardware. Using field-programmable gate arrays (FPGA), Dann et al. [12] designed PipeJSON, the first standard-compliant JSON parser to process tens of gigabytes of data per second. It consists in a JSON parser prototype with a simdjson-compatible interface, and it can serve as a drop-in replacement for the simdjson parser. Hahn et al. [13] present a fast FPGA JSON filtering parser than can locate relevant components with high accuracy and high speed. Peltenburg et al. [14] convert JSON to the Arrow format at the rate of tens of gigabytes per second on an FPGA. Stehle and Jacobsen similarly achieve high speeds, using a graphical processing unit (GPU) [15]. It is also possible to accelerate JSON parsing with multicore parallelism [16, 17].

One of the advantages of our On-Demand approach is that the programmer can quickly skip irrelevant JSON data. Consuming JSON faster by skipping irrelevant sections is a common strategy. Alagiannis et al. [18] query JSON without loading it in their database—parsing only what is necessary. Bonetta and Brantner use just-in-time (JIT) compilation and selective data access [19]. Jiang and Zhao [3] implemented a JSON streaming framework (JSONSki) to fast-forward over different cases of irrelevant substructures. Li et al. present their fast parser, Mison which can jump directly to a queried field without parsing intermediate content [20]. Mison uses SIMD instructions to quickly identify some structural characters but otherwise works by processing bit-vectors in general purpose registers with branching loops. Mison preceded simdjson and it is broadly similar to simdjson's indexing. FishStore [21] parses JSON data and selects subsets of interest, storing the result in a fast key-value store [22]. Though built originally on Mison, FishStore has adopted simdjson[3]. Palkar et al. present Sparser: it filters quickly an unprocessed document to find mostly just the relevant information [23], and then relies on a parser.

3 | ON-DEMAND FRONT-END

Conceptually, parsing with On Demand begins with the creation of a parser instance (`ondemand::parser`). The parser is responsible for memory allocation and it can be reused from document to document. Given a parser instance, the user calls the `iterate` method on a string input which returns a document instance (`ondemand::document`). The method calls the simdjson indexing routine, and writes an index in the parser instance. Afterward, the front-end consists essentially in a pointer over the document which we call internally a `json_iterator`: the iterator resides within the document instance.

We typically traverse the document sequentially from the beginning to the end, always pointing at one pseudo-structural character at a time. The `json_iterator` has also a reference to a pre-allocated string buffer—owned by the parser instance—where we may decode JSON strings: strings in JSON may contain escaped characters (e.g., `\n` or `U0030`) and we provide the user with an unescaped version stored in our own buffer.

During the processing of a valid JSON document, we point at a structural character (‘`[`’, ‘`{`’, ‘`]`’, ‘`}`’, ‘`:`’, ‘`,`’) or at the beginning of a value:

[3] https://github.com/microsoft/FishStore

- A quote character (‘"’) marking the beginning of either a string value or an object key.
- A digit or minus character (‘-’, ‘0’, …, ‘9’) marking the beginning of number.
- The letters ‘n’, ‘t’ or ‘f’ for the start of a `null`, `true`, or `false` token.

However not all of these states are accessible to the programmer: in the normal course of operation, the code points at the beginning of an array (‘[’), at the beginning of an object (‘{’) or at the beginning of a value (number, string, Boolean, null).

The On-Demand front-end validates the content that it parses. We may choose to skip a value (e.g., a string or a number) and in which case the content of the value is not fully validated. For example, the JSON string `[1,1b]` is invalid—because `1b` is not a valid number string—but if it is not accessed (i.e., it is skipped), then no error may be reported.

To help us keep track of the structure of the JSON document, our pointer instance (`json_iterator`) contains a depth counter. When we enter an array or an object, the depth is incremented by one, and decremented by one when we exit the object or the array. In case of an error or when we have consumed the whole document, the depth is set to zero. Table 2 illustrates the depth model for the document from Fig. 1.

From the point of view of the programmer, we arx create arrays and objects (`ondemand::array` and `ondemand::object`). Yet these instances are mere thin wrappers and the creation of the instance translates into a check for the presence of the right structural characters (‘[’, ‘{’, ‘]’, ‘}’). They record their starting position and their document depth. They provide access to corresponding iterators over the elements of the arrays or the fields of the object.

When reading an object’s key-value pair, the key’s location is accessed and the pointer is moved to the associated value. We return to the user a `field` structure containing a pointer to the key that was just read, and a reference to the value. We may also seek a key within an object: when doing so, we scan for the presence of string instances (recognized by a quote character ‘"’) preceded by a colon character (‘:’) at the depth of the current object. The programmer might seek different key values as in Fig. 3: each time we advance through the content, with a possible restart from the beginning if we arrive at the end of the object. We know that a key cannot be found if we have scanned through the object entirely, with a possible restart from the beginning, arriving back to our starting point. We always scan for keys from our current position, instead of systematically starting from the beginning. If the key is not found then we are left pointing at the either the start of the object or at a comma. We are at the depth of the object itself. When searching for a key, if the key is found then we are left pointing at the colon, and we are one depth below the object itself. We then immediately advance to the value.

As much as possible, we rely on `std::string_view` instances. Unlike the conventional C++ `std::string` which allocate their own memory, the `std::string_view` instances are thin wrappers around a character pointer. Thus when we return a string value, we do not need to allocate a new buffer. Given a JSON value (string, number, Boolean or null), the `raw_json_token()` method provides a direct (unprocessed) `std::string_view` instance mapped to the original JSON string. Because the document is indexed, such a method can be implemented using few instructions. Given an array or an object, a programmer can have direct access to the corresponding memory region in the original document: the `raw_json()` method returns a `std::string_view` instance representing the unprocessed string representation of the array or object.

Most functions may return an error. To provide flexible error handling, our functions often return a value of a type corresponding to the template `simdjson_result`. This template contains a pair of values: an error code and the actual value. Hence, a `get_double()` method might return a structure containing an error code and a 64-bit floating-point value of the type `double`. A `simdjson_result` instance throws an exception when we attempt to cast it to its value type (e.g., `double`) if the error condition indicates an error. For users who prefer to avoid exception handling, we make it possible to recover the value from a `simdjson_result` instance after checking the error condition. Most

data type	pointed text (prefix)	depth
+document	{ "Width": 800, "Height": 60	1
+object	{ "Width": 800, "Height": 60	1
double	800, "Height": 600, "Title":	2
double	600, "Title": "View from my r	2
string	"View from my room", "Url": "	2
string	"http://ex.com/img.png", "Pri	2
bool	false, "Thumbnail": { "Url"	2
bool	false, "Thumbnail": { "Url"	2
+object	{ "Url": "http://ex.com/th.p	2
string	"http://ex.com/th.png", "Hei	3
double	125, "Width": 100 }, "arra	3
double	100 }, "array": [116, 9	3
-object	}, "array": [116, 943,	2
+array	[116, 943, 234], "Ow	2
double	116, 943, 234], "Owner"	3
double	943, 234], "Owner": null	3
double	234], "Owner": null }	3
-array	], "Owner": null }	2
skip	null }	2
-object	}	1

TABLE 2 Representation of the state of our On-Demand pointer as we move through the document. The '+' and '-' prefix on the data types indicate the beginning and end of arrays and objects respectively.

of our methods (`get_array()`, `get_object()`, `get_bool()`, ...) return `simdjson_result` instances. For convenience, we can cast values directly to a given type (e.g., `double`) without explicitly creating a `simdjson_result` instance.

Fig. 4 illustrates how the On Demand front-end might work with a non-trivial example. Given a JSON document made of an array of objects, we initiate a loop, accessing each element of the array as an object (`car`). Given such an object, we may query some fields (e.g., `model`) but not others (e.g., `make`): unqueried fields are skipped as much as possible. We may cast a value to a string, to an integer or to a floating-point value. In this particular example, if the value does not match the desired type, then a C++ exception would be thrown. We also support arbitrary schema through the `type()` method: it indicates whether the current node is potentially a number, a string, a Boolean, an array, an object, etc.

The simdjson library offers both a conventional DOM implementation and an On Demand approach. The On Demand approach requires less memory in general. While both strategies materialize the index, which uses one word (four bytes) per pseudo-structural character, the On Demand approach may not require any other intermediate memory.

```
ondemand::parser parser;
auto cars_json = R"([
{"make":"Toyota","model":"Camry","year":2018,
"tire_pressure":[40.1,39.9,37.7,40.4]},
{"make":"Kia","model":"Soul","year":2012,
"tire_pressure":[30.1,31.0,28.6,28.7]},
{"make":"Toyota","model":"Tercel","year":1999,
"tire_pressure":[29.8,30.0,30.2,30.5]}
])"_padded;
// Iterating through an array of objects
for (ondemand::object car : parser.iterate(cars_json)) {
  // Accessing a field by name
  cout << "Model: " << std::string_view(car["model"]) << endl;
  // Casting a JSON element to an integer
  uint64_t year = car["year"];
  cout << "- This car is " << 2020 - year << "years old." << endl;
  // Iterating through an array of numbers
  double total_tire_pressure = 0;
  for (double tire_pressure : car["tire_pressure"]) {
    total_tire_pressure += tire_pressure;
  }
  cout << "- Average tire pressure: " << (total_tire_pressure / 4) << endl;
}
```

FIGURE 4 On-Demand C++ example: it illustrates how we access key-value elements, handle integers and strings.

In the DOM implementation of simdjson, a tree-like data structure is written during parsing, which is later accessed by the programmer. This data structure simply does not exist in the On Demand approach.

We designed the simdjson library so that memory usage is owned by the parser instance. We expect users facing memory-intensive conditions to reuse the same parser repeatedly, which may eliminate the need to allocate new memory as long as the size of the JSON documents is bounded.

The On Demand design has some known limitations. On Demand operates with a single pointer within the document. This means that if a user needs to process multiple components of the documents simultaneously, they must either materialize the data into their own data structure or rewind the iterator to earlier sections of the document. Rewinding the iterator may lead to reparsing the same JSON data multiple times, resulting in a performance penalty.

To illustrate this effect, consider a user who acquires an object instance and repeatedly queries a key value (e.g., `object["mykey"]`). Doing so might require scanning the JSON data of the object multiple times for the same key. In this scenario, the programmer should attempt to materialize the value once because On Demand does not currently support caching.

An anti-pattern for On Demand would be this code sequence:

```
std::string_view make = o["data"]["make"];
```

```
std::string_view model = o["data"]["model"];
std::string_view year = o["data"]["year"];
```

A more performant solution would search for the key `data` just once:

```
ondemand::object data = o["data"];
std::string_view model = data["model"];
std::string_view year = data["year"];
std::string_view rating = data["rating"];
```

In general, On Demand provide an interface that appears like a DOM, but it has different performance characteristics and it requires some care to provide optimal performance. We therefore encourage the programmers to run their performance-sensitive code with logging enabled, so that the programmer can see the various tasks that the front-end has to carry.

The On-Demand provides complete error handling, and it is even possible to know where (in the JSON document) the error occurred, but it can be surprising for some users that an error is triggered after they have seemingly loaded the JSON document. Indeed, the On Demand front-end generally accepts invalid documents, and it only validates the components that the user parses, as they are being parsed. It requires slightly more care on the part of the programmer because the errors can happen at more locations in their code.

4 | EXPERIMENTS

We benchmark our On-Demand front-end on a recent Intel architecture. We use a server configured with Rocky Linux 9. The server has two 32-core Intel Xeon Gold 6338 (Ice Lake) processors having 48 MiB of L3 memory: the 64 cores have 48 kB of L1 data cache memory and 1.24 MiB of L2 cache memory. They are rated with a base clock frequency of 2.0 GHz and a maximal frequency of 3.2 GHz. The server has 376 GiB of main memory (DDR4, 3200 MHz). The benchmarks are single-threaded and we exclude disk and network accesses from our tests.

The software is written in C++ and compiled with GCC 12 using the default CMake setting for a release build: `-O3 -DNDEBUG`. We do not compile for a specific processor architecture. The benchmarks are written using the Google Benchmark (v1.6.0) framework.[4] Our benchmarking code is *instrumented*: we use the performance counters of the processors to record the number instructions retired, the number of cycles and the number of mispredicted branches. It allows us to compute the actual average processor frequency: we verify that it can reach 3.2 GHz as expected. The number of instructions *retired* includes only the instructions that have been fully executed and excludes instructions issued solely due to *speculative execution* [24, 25, 26].

To benchmark and validate our work, we need various tasks. Our goal is to compare On-Demand with existing generic JSON parsers. We choose to compare against the following state-of-the-art C++ standard-compliant parsers.

- The yyjson parser is a C library that was released shortly after the first release of the simdjson library. According to the author (private communication), the goal of the yyjson was to get as close as possible to simdjson, while using only generic C code. We use the version 0.5.1, released on June 17, 2022.
- In earlier work, Langdale and Lemire selected RapidJSON and sajson as references. In 2018, RapidJSON was described as the fastest traditional state-machine-based parser available [23]. Though they both have similar performance, sajson only does partial UTF-8 validation. Given the introduction of yyjson, we keep only RapidJSON (version 1.1.0, released in September 2018).

[4] https://github.com/google/benchmark

- We also include JSON for Modern C++ (version 3.10.5, from January 2022)—it is also known as Nlohmann/json. It is a popular parser.

Though there are many other libraries, in different programming languages, we believe that these three choices represent a good sample of competitive solutions. To test On-Demand, we use the simdjson library version 3.2.3 (released in August 2023[5]).

We use the JSON input as a database, and we seek to resolve some queries. We load all JSON documents in memory prior to running the benchmark; we omit disk and network accesses. Our main test document is the `twitter.json` file [8]: it is the result of a search for the character one in Japanese and Chinese using the Twitter API. It contains 631 515 bytes, 2108 integers, 18 099 strings, 1264 objects and 1050 arrays. We picked the following tasks.

- *json2msgpack*: We must validate and convert the `twitter.json` file into a binary MessagePack (MsgPack for short) equivalent. MessagePack is a binary format comparable to JSON [27]. We use a simple conversion: all numbers are mapped to double types, all strings are prefixed by a 32-bit counter, and so forth. All our implementations rely on a recursive function call where the type of the current document node is queried and used as part of a switch/case query. When an array or an object is encountered, its content is processed recursively. It is an instance where the entire content of the file is consumed.
- *partial tweets*: The `twitter.json` file consists of an object containing a field with the key `statuses` having as a value an array. Each element of the array is a tweet as a JSON object. We seek to extract from each tweet the fields `created_at`, `id`, `text`, `in_reply_to_status_id`, `retweet_count`, `favorite_count` and from a subobject associated with the key `user`, we want to acquire the user's `id` and the user's `screen_name`. We expect that such partial extraction of the content of a JSON document is common in practice.
- *distinct user*: In the `twitter.json` file, we seek to find all user identifiers (`id`) as 64-bit identifiers and store them in an array implemented as a `std::vector<uint64_t>` instance. They are found as part of tweets (`user/id`) or within retweet metadata (keys `retweeted_status/user/id`).
- *find tweet*: We seek the textual content (`text`) of the tweet having identifier `505874901689851904`. There is only one such tweet.
- *top tweet*: We scan all tweets and report the most retweeted one (according to `retweet_count`), returning the `screen_name` of the author and the textual content (`text`).

We also extended our benchmarks with two synthetic data sources:

- *kostya*: We reproduce the Kostya benchmark (see § 1 and Fig. 3), with less overhead and in a more controlled setting (Google Benchmark). We process 524 288 triples stored in memory. Whereas the original Kostya benchmark computes the sums of the `x`, `y`, and `z` values, we store the parsed triples in a dynamic array (`std::vector`).
- *large random*: We proceed with a task similar to the Kostya benchmark except that instead of summing up the values, we construction three-value vectors and we append them to an array. We create, in memory, a temporary JSON document made of 1 000 000 triples of floating-point values within objects (e.g., `"x":0.14323412321, "y":0.931321111, "z":0.0111141232312`).

On a recent Linux system with the prerequisite (CMake version 3.15 or better and GCC 12), we expect our benchmarks to be reproducible using a few command lines, e.g.:

```
git clone https://github.com/simdjson/simdjson.git
cd simdjson
cmake -B build -DSIMDJSON_DEVELOPER_MODE=ON
```

[5] https://github.com/simdjson/simdjson

TABLE 3 Best and average processing speed in GiB s^{-1} for various tasks and implementations.

task	simdjson (On Demand)	simdjson (DOM)	yyjson	RapidJSON	JSON for Modern C++
json2msgpack	2.3–2.5	1.7–1.8	0.75–1.3	0.38–0.43	0.023–0.025
partial tweets	4.8–5.2	2.9–3.1	0.96–2.0	0.38–0.48	0.090–0.10
distinct user	5.0–5.4	2.9–3.2	0.96–2.1	0.38–0.50	0.098–0.11
find tweet	8.0–8.7	3.1–3.3	0.97–2.1	0.38–0.49	0.12–0.13
top tweet	4.9–5.3	3.0–3.2	0.97–2.0	0.38–0.55	0.080–0.086
kostya	2.5–2.7	1.5–1.6	0.97–1.0	0.54–0.57	0.094–0.099
large random	0.87–0.91	0.48–0.51	0.45–0.47	0.24–0.25	0.043–0.045
geometric mean	3.3–3.6	1.9–2.1	0.84–1.4	0.37–0.46	0.069–0.075

```
cmake --build build -j
./build/benchmark/bench_ondemand
```

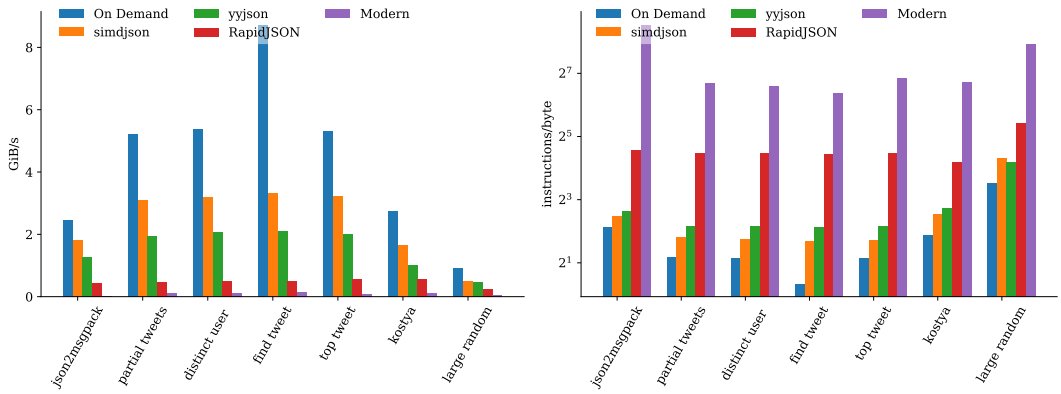
Table 3 presents the processing speed for various tasks. We define the speed as the size of the input divided by the estimated elapsed time. Performance timings are heavily skewed toward the minimum; they do not follow a normal distribution [28]. Accordingly, we present the best processing speed (out of all runs) and the average processing speed. Except for the *json2msgpack* task, the On-Demand front-end is more than twice as fast as any competitor, except maybe for the DOM version of simdjson. Considering the geometric mean of the best results, we find that On Demand is 70% faster than the conventional (DOM-based) simdjson, it is over 2.5 times faster than yyjson, and nearly eight times faster than RapidJSON. On Demand is nearly 50 times faster than JSON for Modern C++.

Table 4 presents the number of CPU instructions retired by input byte. The number of instructions required is stable in our tests (within 1%). We find that the On-Demand front-end requires far fewer instructions than the competitors. It is indicative of a low number of mispredicted branches and of limited data dependencies. When considering the geometric mean, On Demand uses 60% of the instructions of the conventional (DOM-based) simdjson, half of yyjson's instructions, and nearly eight times fewer instructions than RapidJSON. Compared to JSON for Modern C++, On Demand requires over 40 times fewer instructions. In general, On Demand is even faster than the numbers suggest, since it typically achieves a high number of instructions retired per unit of time. This is indicative of a low number of mispredicted branches and limited data dependencies.

Fig. 5 presents the speed and number of instructions in graphical form. JSON for Modern C++ is much slower than the alternatives and it is difficult to represent it graphically on the same scale: we omit its instruction counts in the plot, and its best speed is barely visible. We see graphically that the benefits of On Demand vary significantly depending on the task: they are modest for *json2msgpack* and much more significant for *find tweet*. We explain this difference by the fact that in *json2msgpack*, the entire document must be parsed into a data structure, and On Demand is thus unable to make gains due to its ability to skip some content. On the contrary, in the *find tweet* task, much of the JSON processing can be skipped without adverse consequence. On Demand still provides a benefit with the *json2msgpack* task because the data is written directly to the desired output without an intermediate storage.

TABLE 4 Instructions per input byte for various task and JSON implementations.

task	simdjson (On Demand)	simdjson (DOM)	yyjson	RapidJSON	JSON for Modern C++
json2msgpack	4.4	5.6	6.3	24.0	374.0
partial tweets	2.3	3.5	4.5	22.2	104.
distinct user	2.2	3.4	4.5	22.1	97.1
find tweet	1.3	3.3	4.4	22.0	82.3
top tweet	2.2	3.3	4.4	22.1	115.0
kostya	3.7	5.8	6.7	18.2	105.0
large random	11.4	19.9	18.4	43.3	244.0
geometric mean	3.1	5.1	6.1	23.9	137.0



(a) Best speed (gigabytes per second)

(b) instructions per byte

FIGURE 5 Speed and number of instructions for various benchmarks.

5 | CONCLUSION

Our work suggests that On-Demand might be a better approach for parsing semi-structured text documents when high efficiency and high expressivity are important. To ensure reproducibility, our work is freely available as open source software.

Future work should assess the generality of On-Demand. We should write On-Demand front-end in other programming languages (e.g., Java, Rust, Go). We should apply it to other data sources (e.g., XML, CSV).

There are implementation design strategies that warrant further investigation. For example, our implementation relies on a precomputed index, giving us the location of structural characters: it makes our implementation convenient and the construction of the index is fast, but an On-Demand front-end may not require an index. Yet an even richer document index might prove more beneficial in some cases: instead of merely indexing the location of the objects, arrays and values, the index could be built with more information about the schema: e.g., the index could inform us that we have encountered an array made of n integers. Future work should consider heterogeneous computing [29]

with On Demand: the indexing could be computed by a GPU [15] or on an FPGA [12, 13, 14] and passed on to the On Demand front-end executed on general-purpose processor.

Acknowledgements

We thank N. Boyer for his help writing benchmarks and tests.

references

[1] Bray T, The JavaScript Object Notation (JSON) Data Interchange Format; 2017. Internet Engineering Task Force, Request for Comments: 8259. https://tools.ietf.org/html/rfc8259.

[2] Durner D, Leis V, Neumann T. JSON Tiles: Fast Analytics on Semi-Structured Data. In: Proceedings of the 2021 International Conference on Management of Data SIGMOD '21, New York, NY, USA: Association for Computing Machinery; 2021. p. 445–458. https://doi.org/10.1145/3448016.3452809.

[3] Jiang L, Zhao Z. In: JSONSki: Streaming Semi-Structured Data with Bit-Parallel Fast-Forwarding New York, NY, USA: Association for Computing Machinery; 2022. p. 200–211. https://doi.org/10.1145/3503222.3507719.

[4] Wood L, Le Hors A, Apparao V, Byrne S, Champion M, Isaacs S, et al. Document Object Model (DOM) level 1 specification. W3C; 1998.

[5] Means SS. The Book of SAX. USA: No Starch Press; 2002.

[6] Jiang L, Sun X, Farooq U, Zhao Z. Scalable Processing of Contemporary Semi-Structured Data on Commodity Parallel Processors - A Compilation-Based Approach. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS '19, New York, NY, USA: Association for Computing Machinery; 2019. p. 79–92. https://doi.org/10.1145/3297858.3304008.

[7] Scott NW, Hodson DD, Dill R, Grimaila MR. Using Serde to Serialize and Deserialize DIS PDUs. In: 2020 International Conference on Computational Science and Computational Intelligence (CSCI) IEEE; 2020. p. 1425–1428.

[8] Langdale G, Lemire D. Parsing gigabytes of JSON per second. The VLDB Journal 2019;28(6):941–960.

[9] Henderson P, Morris Jr JH. A lazy evaluator. In: Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages; 1976. p. 95–103.

[10] Lemire D. Number parsing at a gigabyte per second. Software: Practice and Experience 2021;51(8):1700–1727.

[11] Keiser J, Lemire D. Validating UTF-8 in less than one instruction per byte. Software: Practice and Experience 2021;51(5):950–964.

[12] Dann J, Wagner R, Ritter D, Faerber C, Froening H. PipeJSON: Parsing JSON at Line Speed on FPGAs. In: Data Management on New Hardware DaMoN'22, New York, NY, USA: Association for Computing Machinery; 2022. https://doi.org/10.1145/3533737.3535094.

[13] Hahn T, Becher A, Wildermann S, Teich J. Raw filtering of JSON data on FPGAs. In: 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE) IEEE; 2022. p. 250–255.

[14] Peltenburg J, Hadnagy Á, Brobbel M, Morrow R, Al-Ars Z. Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators. In: 2021 International Conference on Field-Programmable Technology (ICFPT) IEEE; 2021. p. 1–9.

[15] Stehle E, Jacobsen HA. ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data. Proc VLDB Endow 2020 jan;13(5):616–628. https://doi.org/10.14778/3377369.3377372.

[16] Jiang L, Qiu J, Zhao Z. Scalable Structural Index Construction for JSON Analytics. Proc VLDB Endow 2020 dec;14(4):694–707. https://doi.org/10.14778/3436905.3436926.

[17] Pavlopoulou C, Carman Jr EP, Westmann T, Carey MJ, Tsotras VJ. A Parallel and Scalable Processor for JSON Data. In: EDBT'18; 2018. .

[18] Alagiannis I, Borovica R, Branco M, Idreos S, Ailamaki A. NoDB in Action: Adaptive Query Processing on Raw Data. Proc VLDB Endow 2012 Aug;5(12):1942–1945.

[19] Bonetta D, Brantner M. FAD.Js: Fast JSON Data Access Using JIT-based Speculative Optimizations. Proc VLDB Endow 2017 Aug;10(12):1778–1789.

[20] Li Y, Katsipoulakis NR, Chandramouli B, Goldstein J, Kossmann D. Mison: A Fast JSON Parser for Data Analytics. Proc VLDB Endow 2017 Jun;10(10):1118–1129.

[21] Xie D, Chandramouli B, Li Y, Kossmann D. FishStore: Faster Ingestion with Subset Hashing. In: Proceedings of the 2019 International Conference on Management of Data SIGMOD '19, New York, NY, USA: ACM; 2019. p. 1711–1728.

[22] Chandramouli B, Prasaad G, Kossmann D, Levandoski J, Hunter J, Barnett M. FASTER: A Concurrent Key-Value Store with In-Place Updates. In: Proceedings of the 2018 International Conference on Management of Data SIGMOD '18, New York, NY, USA: ACM; 2018. p. 275–290.

[23] Palkar S, Abuzaid F, Bailis P, Zaharia M. Filter before you parse: faster analytics on raw data with Sparser. Proceedings of the VLDB Endowment 2018;11(11):1576–1589.

[24] Intel, Intel 64 and ia-32 architectures software developer's manual, Volume 3B: System programming Guide, Part 2; 2016. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html.

[25] Marques D, Ilic A, Sousa L. Mansard roofline model: Reinforcing the accuracy of the roofs. ACM Transactions on Modeling and Performance Evaluation of Computing Systems 2021;6(2):1–23.

[26] Gravelle BJ, Nystrom WD, Norris B. Performance Analysis with Unified Hardware Counter Metrics. In: 2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) IEEE; 2022. p. 60–70.

[27] Ching T, Eddelbuettel D. RcppMsgPack: MessagePack Headers and Interface Functions for R. R Journal 2018;10(2).

[28] Hoefler T, Belli R. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In: Proceedings of the international conference for high performance computing, networking, storage and analysis; 2015. p. 1–12.

[29] Khokhar AA, Prasanna VK, Shaaban ME, Wang CL. Heterogeneous computing: Challenges and opportunities. Computer 1993;26(6):18–27.