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
}
```

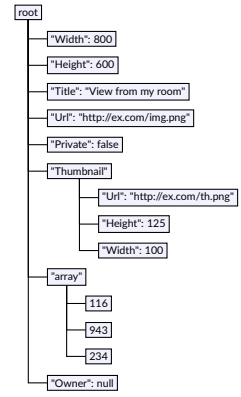

FIGURE 1 JSON example with corresponding tree form

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++