Supporting Descendants in SIMD-Accelerated JSONPath

Mateusz Gienieczko
mat@gienieczko.com
Univeristy of Warsaw
Warsaw, Poland

Filip Murlak
fmurlak@mimuw.edu.pl
Univeristy of Warsaw
Warsaw, Poland

Charles Paperman
charles.paperman@univ-lille.fr
CRIStAL, Université de Lille, INRIA
Lille, France

ABSTRACT

Harnessing the power of SIMD can bring tremendous performance gains in data processing. In querying streamed JSON data, the state of the art leverages SIMD to fast forward significant portions of the document. However, it does not provide support for descendant, which excludes many real-life queries and makes formulating many others hard. In this work, we aim to change this: we consider the fragment of JSONPath that supports child, descendant, wildcard, and labels. We propose a modular approach based on novel depth-stack automata that process a stream of events produced by a state-driven classifier, allowing fast forwarding parts of the input document irrelevant at the current stage of the computation. We implement our solution in Rust and compare it with the state of the art, confirming that our approach allows supporting descendants without sacrificing performance, and that reformulating natural queries using descendants brings impressive performance gains in many cases.

KEYWORDS

json, jsonpath, simd, query language, data management

ACM Reference Format:
Mateusz Gienieczko, Filip Murlak, and Charles Paperman. XXXX. Supporting Descendants in SIMD-Accelerated JSONPath. In *Proceedings of (ASPLOS '24)*. ACM, New York, NY, USA, 22 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

JSON is the format of choice for both modern web communication and large datasets. Due to its prevalence, all modern programming frameworks provide some facility for JSON processing. While the technology is relatively mature, substantial performance gains can be still achieved by exploiting the Single Instruction, Multiple Data (SIMD) capabilities of modern commodity processors [24, 27, 28]. Langdale and Lemire harnessed SIMD instructions to parse JSON data, validate it and produce its tree representation (the DOM, Document Object Model), achieving impressive speed-ups over conventional parsers [26]. While DOM allows us to query the document efficiently, it is prohibitively costly for large datasets. Not only does it take up massive amounts of RAM, but constructing it would also take most of the query execution time – parsing can amount to up to 90% of time spent in such an application, while the actual query only touches a small portion of the input data [29]. When faced with terabytes of data to query, the only feasible solution is a streaming algorithm with minimal memory footprint.

1.1 State of the Art

General tools for querying streamed JSON data, such as JsonSurfer [34] and jq [6], are slow: the throughput of JsonSurfer oscillates around 200MB/s and jq is an order of magnitude slower. Meanwhile, Langdale and Lemire's simdjson can parse gigabytes of data per second, providing access via a SAX-like on-demand API [32]. Jiang and Zhao's recent JSONSki [25] shows that certain queries can be evaluated even faster, surpassing the throughput of simdjson thanks to clever SIMD-accelerated fast-forwarding through irrelevant fragments of the stream.

JSONSki supports a useful but limited subset of the popular JSONPath query language [21]. It has no support for descendant selectors, and their wildcard selector implements only a part of the JSONPath specification, stepping into every entry of an array, but not into every field of an object. Importantly, JSONSki relies on knowing whether a selector acts on objects or lists, which the authors identify as an obstacle for supporting descendant, and which is also incompatible with idiomatic wildcard.

1.2 Motivation

Full support for wildcards and descendants is a highly desirable feature. Idiomatic wildcards allow accessing elements without specifying the full path. Descendants allow reaching deep down the document and accessing elements at multiple depths with a single query. When they are supported, tasks like fetching all values associated with a given property name in the document become very easy, as they can be succinctly represented with a descendant query. For example, one could scrape all `url` property values from a document without knowing anything about the paths leading to them, whereas without descendants the user would need to know the depth of the property, and without full wildcard support also specify all labels on the path, leading to an explosion of possibilities. A more general example is the code-as-data application scenario (for instance, harvesting code for AI [8]), fueled by the proliferation of large code repositories and archives such as Software Heritage [30]. Tools such as `clang` output abstract syntax trees of programs as JSON documents, deep and highly irregular. Exploring such documents without wildcard and descendant is infeasible.

1.3 Our Contribution

We present an engine supporting JSONPath queries with labels, child, wildcard, and descendant selectors that does not rely on the advance knowledge of types of processed elements. We propose a novel modular approach based on *depth-stack automata* that process a stream of events produced by a *state-driven classifier* capable of fast-forwarding through currently irrelevant fragments of the stream using SIMD acceleration.

Abstract automaton execution. The classifier consumes the raw JSON stream and produces events associated with symbols meaningful for the query, such as structural symbols and relevant labels. Every step of the automaton is costly, but the classifier generates only the necessary events, allowing the automaton to skip over most of the raw input stream. This modular architecture allows one to compare different classifiers and different automata models. It also provides a general abstraction separating fast branchless SIMD-enhanced stream processing from the heavily branching sequential code implementing the query logic, potentially of use elsewhere.

Sparse stack representation. In the described model, queries combining child and descendant segments can be easily executed by a push-down automaton, but using the stack is potentially costly. Depth-register automata [1] are stackless, but they cannot handle all queries mixing child and descendant. Aiming to get the best of both worlds, we propose *depth-stack automata* which are equivalent to pushdown automata but offer a flexible sparse representation of the stack, allowing to keep its depth to a bare minimum.

State-driven classifier. Because the automaton cares about different events in different states, additional savings can be made by adjusting the set of recognized symbols depending on the current needs of the automaton. This way we reduce the workload of the automaton (fewer events to process), and allow the classifier to pick an algorithm optimized for a particular character set. The idea is implemented at two levels. Internally, our main classifier is capable of *toggling* specific symbols on the fly. Externally, our *multi-classifier pipeline* allows switching dynamically between the main classifier and additional specialized classifiers. In principle, there is an optimal classifier for each state of the automaton, but the cost of switching often exceeds the gain. This is why we do not switch whenever a state change occurs, but only when the expected benefits justify it. In the remaining cases we rely on the flexibility of the main classifier.

Implementation and Experiments. We implemented our solution in Rust. The resulting tool, dubbed rsonpath, is available online [13]. In a series of experiments we compare rsonpath with the state of the art, and confirm that our approach allows supporting descendant and idiomatic wildcard without sacrificing performance (actually, with 10-20% performance boost) and that by using descendants some natural queries can be sped up impressively (up to an order of magnitude).

1.4 Outline

In Section 2 provide the necessary background on JSON and JSONPath. Then we explain how the query logic is implemented using depth-stack automata (Section 3). Next, we describe the classifiers and how they are orchestrated to provide the automaton with suitable access to the stream (Section 4). Section 5 is devoted to the experimental evaluation. We conclude in Section 6.

2 BACKGROUND

JSON is a serialization format for JavaScript objects. A JSON document J is one of the following:

- an *atomic value*: a string, a number, or one of the special values `true`, `false`, `null`;
- an *array* of the form $\big[\, J_1,\, J_2,\, \ldots,\, J_n \,\big]$ where $J_1, J_2, \ldots, J_n$ are JSON documents;
- an *object* of the form $\big\{\, \ell_1 : J_1,\, \ell_2 : J_2,\, \ldots,\, \ell_n : J_n \,\big\}$ where here $J_1, J_2, \ldots, J_n$ are JSON documents and $\ell_1, \ell_2, \ldots, \ell_n$ are strings (called *property names* or *labels*).

We call $J_1, J_2, \ldots, J_n$ *direct subdocuments* (or *children*) of J. A string is a sequence of Unicode characters, wrapped in double quotes, using backslash escapes. For instance, `{"a":"{\"b\":2022}"}` is an object with a single property `a` whose value is the string `{"b":2022}`. The escaping mechanism poses additional challenges for rapid processing of JSON documents.

While JSONPointer [35] can point to a specific subdocument, JSONPath queries select sets of subdocuments accessible by paths specified using more flexible selectors. We consider a fragment of JSONPath using selectors of the forms $\$$, $.\ell$, $.*$, and $..\ell$; that is, path expressions are given by the grammar

$$e ::= \$ \;\mid\; e.\ell \;\mid\; e.* \;\mid\; e..\ell$$

where ℓ is a property name. These selectors allow navigating to, respectively, the document root, the property ℓ of the current element, any direct subdocument (object property or array entry) of the current element, the property ℓ of the current element or any of its subdocuments.

More precisely, we apply the *node semantics*, defined formally as follows. When evaluated over a JSON document J, an expression e returns a (possibly empty) set $e(J)$ of subdocuments of J (in the order in which they appear in J). The expression $\$$ returns J: $\$(J) = \{J\}$. Suppose

$$e(J) = \{J_1, J_2, \ldots, J_n\}$$

with $n \geq 0$. Then, for $s = .\ell$ or $s = .*$ or $s = ..\ell$,

$$es(J) = s(J_1) \cup s(J_2) \cup \ldots \cup s(J_n)\,,$$

where $s(J_i)$ is the set of subdocuments selected by the selector s in J_i: $.\ell(J_i)$ is the singleton of the value of property ℓ in J_i or the empty set if J_i is an array or does not have property ℓ; $.*(J_i)$ is the set of all direct subdocuments of J_i; and $..\ell(J_i)$ is the set of values of property ℓ in all objects in J_i, including J_i itself; that is,

$$..\ell(J_i) = .\ell(J_i) \cup ..\ell(J_i^1) \cup ..\ell(J_i^2) \cup \cdots \cup ..\ell(J_i^{n_i})\,,$$

where $J_i^1, J_i^2, \ldots, J_i^{n_i}$ are the direct subdocuments of J_i. For example, in the document `{a:[{b:{c:1}},{b:[2]}]}`, the query `$.a..b.*` returns 1 and 2.

There exists an alternative *path semantics*, in which it matters how a given subdocument is reached. For example, in the document `{a:{a:{a:{b:"Yay!"}}}}` the query `$..a..b` can select `Yay!` by matching the selector `..a` either to the child or to the grandchild or to the great grandchild of the root. Under the node semantics it does not matter. Under the path semantics this is reflected by returning `Yay!` three times: once for each way it can be reached. Path semantics can be defined formally by replacing sets with multisets in the definition above.

We posit that path semantics is undesirable. The reasons are twofold. First, cluttering results with duplicated values is usually not what the user wants. Second, path semantics is actually harder to compute. One might think it is easier, as it does not require eliminating duplicates, but this intuition is correct only for a naive recursive implementation. An efficient implementation will traverse the document only once, naturally producing no duplicates. Reproducing the multiplicities of the path semantics, however, would incur additional cost. By generalizing the above example we can see that the result set might even be exponentially large in the length of the query.

Using the JSONPath comparison project [3] we found that most existing implementations of JSONPath use path semantics: out of 44 tested implementations, 34 use path semantics (including the original implementation by Gössner [21]), while only 6 use node semantics (4 were errors). See Table9 for details. PostgreSQL's implementation of JSONPath also uses path semantics, which makes it possible to construct simple antagonistic queries against the database. From this point onward we consider only node semantics, as not only the more useful, but also easier to implement in our streaming model, which inherently demands a linear pass over the document.

3 QUERY EXECUTION

Our query execution algorithm has two phases. In the first phase, detailed in Section 3.1 below, the query is compiled into a deterministic *query automaton* recognizing access paths leading to subdocuments selected by the query. In the second phase, the query automaton is simulated over the streamed document. The simulation uses a concise stack representation (Section 3.2) as well as four skipping techniques (Section 3.3) allowing it to fast-forward through irrelevant fragments of the stream. The implementation (Section 3.4) abstracts away access to the stream as an iterator, which can be seen as a SIMD-enhanced lexer, capable of filtering out irrelevant tokens on demand. The iterator is the workhorse of rsonpath; we discuss it in detail in Section 4.

3.1 Constructing query automata

A JSONPath query can be naturally represented as a finite automaton that runs on a word formed by the sequence of labels on a path from the root to a node in the document tree (array entries are given an artificial label, different from property names).

For descendant-free queries the situation is simple. Consider a query of the form $\$.\ell_1.\ell_2 \cdots .\ell_n$, where each ℓ_i is either a label or a wildcard. An automaton for such a query has a very regular form, illustrated in Figure 1. It consists of a chain of states corresponding to the selectors of the query. From selector $.\ell_i$, the automaton moves to selector $.\ell_{i+1}$ when it sees a matching label. That is, if ℓ_i is a label it must see ℓ_i itself, and if ℓ_i is wildcard every letter is fine. From the last selector, the automaton moves to the accepting state (when it sees a matching label). Importantly, the automaton is a *deterministic finite automaton* (DFA), and as such it is ready for simulation.

For queries with descendants, things get more complicated. One can still construct an automaton with states corresponding to selectors, but in states corresponding to descendants, the automaton can not only pass to the next state when it sees a matching label, but it can also loop regardless of the seen label; see Figure 2 (top). We call such states *recursive*. States corresponding to child selectors are called *direct*. As in recursive states the automaton has sometimes two choices for a single label, it is a *nondeterministic finite automaton* (NFA). To ease the simulation, we need to turn the NFA into a minimal DFA. One can, of course, use a general method to determinize and minimize an NFA, but it is useful to understand certain common structural properties of minimal DFAs obtained from JSONPath queries.

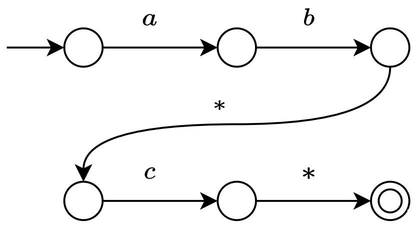


Figure 1: Minimal DFA recognising paths matching query `$.a.b.*.c.*` (trash state omitted).

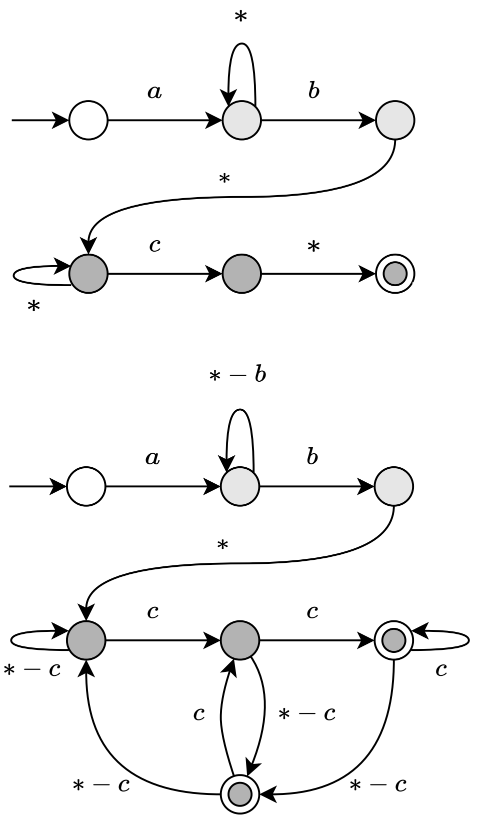


Figure 2: NFA (top) and minimal DFA (bottom) recognising paths matching query `$.a..b.*..c.*`. Different shades of grey represent segments in the NFA and corresponding components in the DFA.

Descendant selectors split the query into segments. Each segment begins with a descendant selector (or `$`) and ends just before the next descendant selector (or at the end of the query). For example, the query `$.a..b.*..c.*` consists of three segments: segment `$.a`, segment `..b.*`, and segment `..c.*`. The following *greedy match property* is key: when searching for query matches in the document, as soon as we find a match for a segment on some branch, we can ignore any further matches of the segment on this branch, and start looking for a match for the next segment, etc. Note that we crucially rely on node semantics here. Under the path semantics, we would have to consider each subsequent match too, as they would lead to different matches of the whole query even if the final selected node is the same. For example, for the query `$.a..b.*..c.*` and document `{a:{b:{b:{b:{c:[42]}}}}}`, we would have to consider matches of segment `..b.*` at all `b` nodes. Ultimately we would discover that the first two matches can be extended to the whole query, but the third one cannot, and we would report two matches in total (even though there is only one way to match the last segment). Under node semantics, we can assume that `..b.*` is matched at the first `b` and disregard subsequent matches.

The greedy match property implies that once we reach a given recursive state in the NFA, we can forget about all previous states. This breaks the automaton down into segments such that only one segment needs to be simulated at a time. Consequently, each segment can be determinized separately, usually—but not always—yielding a single strongly connected component of the DFA; see Figure 2 (bottom). Each state has some transitions over concrete labels (typically advancing the matching), and a single *fallback transition* over the remaining labels (typically leading to the *initial state of the component*, corresponding to a recursive state of the NFA). The segment of the NFA corresponding to selectors before the first descendant is always deterministic and is simply copied to the DFA. In this component, fallback transitions from non-wildcard states lead to an all-rejecting trash state (not shown in Figure 2). Other segments may easily yield exponentially large components. For example, the query $\$..a.*.*.\cdots.*$, selecting nodes whose ancestor n levels up has label a, reconstructs the classical example of exponential blowup when passing from an NFA to a DFA.

3.2 Simulating query automata on streamed trees

With the determinized query automaton at hand, executing a query boils down to simulating this automaton on all paths in the tree. The tree, however, is streamed. For now, let us take a high-level view and treat the streamed document as a sequence of tokens: structural characters '`{`', '`}`', '`[`', '`]`', '`:`', '`,`' , labels, and atomic values. In a pass over the streamed document we can simulate a DFA easily, as long as we use a stack:

- whenever an opening character ('`[`' or '`{`') is encountered, the state of the DFA is pushed to the stack;
- each encountered label (including the "non-label" of array entries) triggers a transition of the DFA and, if the new state is accepting, an answer is reported;
- a closing character ('`]`' or '`}`') pops the stack and restores the state to what it was before visiting the subtree.

In this work, however, we aim at minimising the use of the stack, the hypothesis being that such code will have fewer branches and thus should be more performant when paired with SIMD processing.

Querying streamed trees in a stackless manner was investigated in [1], where the authors characterise the kinds of queries that can be effectively executed on *depth-register automata*, which are finite automata with a constant number of depth registers and access to the current depth in the tree. The only operations allowed on registers are storing the current depth and comparing whether the stored value is less than, equal to, or greater than the current depth. It is easy to see that such automata can be effectively implemented, as the current depth can be tracked in a single integer variable that is incremented on every occurrence of a opening character and decremented on every occurrence of a closing character.

Automata resulting from descendant-only queries can be simulated stacklessly in the depth-register model. A stackless algorithm for the query $\$..\ell_1..\ell_2\ \ldots\ ..\ell_n$ uses depth registers $\delta_1, \ldots, \delta_{n-1}$ and states $1, 2, \ldots, n+1$. We start in state 1 and report an answer whenever in state $n+1$. When in state i, there are two kids of events that trigger a transition:

- if the current depth falls to the value in register δ_{i-1}, move to state $i-1$ (not applicable when $i = 1$);
- if label ℓ_i is found, set δ_i to the current depth and move to state $i+1$ (not applicable when $i = n+1$).

For automata resulting from queries mixing descendant and child selectors, the depth-register model is too weak [1]. Intuitively, this is due to the non-local nature of such queries: two children of the same node can be arbitrarily far away from each other in the input JSON string. For instance, for the query $\$..\ell_1.\ell_2$ the automaton would have to remember every occurrence of label ℓ_1 on the current path from the root in order to be able to check all its children, because the children of shallower ℓ_1 nodes might occur both before and after the children of deeper ℓ_1 nodes. While all DFAs can be simulated using a stack, this becomes costly when the stack gets large. As a remedy, we generalize the depth-register model by employing a *depth-stack*. Rather than just symbols from a fixed finite alphabet, a depth-stack stores *stack frames*, which consist of a symbol and a depth. By replacing the registers in a depth-register automaton with a depth-stack we obtain a *depth-stack automaton*. The automaton can pop a frame, push a frame with the current depth, and compare the depth in the top frame with the current depth.

The advantage of the depth-stack over the classical stack is conciseness. In the ordinary stack-based simulation the height of the stack is tied to the depth of the tree. In the depth-stack model we keep track of the depth using the counter and the stack is only used to record when the state of the simulated DFA changes:

- whenever a label is about to trigger a state change, the current state and depth are pushed to the depth-stack;
- whenever the current depth drops to the value in the topmost frame, the frame is popped and the current state is reverted to the one in the frame.

For a child-free query with n selectors this results in $O(n)$ memory usage, and the at most n frames on the stack correspond directly to the n registers from the stackless algorithm. For a query with child selectors the stack can grow up to the depth of the JSON tree, but for most real-life data this is rare: it requires documents where nodes with the same label are nested in itself, and the query asks for a child of a node with such a label (see query A1 in Section 5). In the implementation we represent the depth-stack using a special Rust structure `SmallVec`. This puts our depth-stack on the actual stack of the executing thread as long as it is relatively shallow (less than 128 elements, bounded by 512 bytes). In the rare cases when it grows larger than that, it is moved to the heap.

Apart from the depth-stack, the simulation algorithm stores the query automaton, whose size is negligible for practically useful queries. The overall memory footprint is linear in the depth of the document. Time complexity is linear in the size of the input data.

3.3 Skipping

While the algorithm described in Section 3.2 dutifully steps through every element of the document, the key insight from [25] is that one can *skip* fragments of the document that are known not to contain query matches. Of course, we cannot really jump over fragments of the stream, but we can fast-forward through them using SIMD processing. Below we discuss in the abstract the four types of skipping used in our query engine; their SIMD implementation is explained Section 4. In order to ensure correct semantics of wildcards, we skip less agressively than JSONSki, which assumes that wildcards match only array elements and so skips over objects if the next selector is a wildcard. Also, because automata for queries with wildcard and descendant are structurally non-trivial (as we have seen in Section 3.1), the criteria for skipping become more subtle. Finally, the last kind of skipping is entirely new: it is impossible for queries without descendant selectors.

Skipping leaves. We call a state *internal* if it has no transitions to accepting states. When the simulated DFA is in an internal state, it makes no sense to visit leaves (i.e. atomic values) in the tree because the automaton needs to descend at least two levels to accept. That is, when going through the children of some node v we would like to skip all leaves ahead of us and jump straight to the next child that has some descendants. For example, consider the query `$.a..b.*..c.*` from Figure 2 and the document

```
{b:"Long string with no matches for sure",
    c:[1,2,...,1000], a:{...}, ... }.
```

After reading the initial '`{`' we are in the initial state of the DFA in Figure 2 (bottom), which is an internal state. We would now like to skip the `b` child, because the DFA has nowhere to go below it. This can be done by jumping to the next opening character. However, we should also be mindful of closing characters because, if all remaining siblings are leaves, the next opening character will be already outside of the scope of the current subtree, which affects the simulation. Overall, when skipping leaves, we will be tracking structural characters '`{`', '`}`', '`[`', '`]`', fast-forwarding between their successive occurrences. In order to simulate the DFA, we will also need the label before each '`{`', but we can get it by backtracking. When not skipping leaves, we will have to pay attention also to the remaining structural characters: commas and colons.

Skipping children. When reading the label of the current node v moves the simulated DFA to the trash state, it makes no sense to visit the children of v. Instead, we would like to jump straight to the closing character marking the end of the whole subtree. Continuing the example above, after jumping to the first '`[`', we see a label c before it which would lead us to the trash state (not depicted in Figure 2). We would like to skip all the children of this node (i.e., all the entries in the array) and jump straight to the closing '`]`'. Importantly, since we have already seen the opening character, we know what the matching closing character will be, so we need to track only two characters: either '`{`', '`}`' or '`[`', '`]`'.

Skipping siblings. We call a state *unitary* if it has a single transition over a concrete label and its fallback transition leads to the trash state. (Such states correspond to non-wildcard selectors in the query before the first descendant selector.) Suppose that upon reading the label of the current node v the simulated DFA enters a unitary state with a transition over label ℓ. As soon as we discover a child v' of v with label ℓ, it makes no sense to visit the remaining siblings of v', because labels do not repeat among siblings. That is, after processing the subtree rooted at v, just like in the previous case, we would like to jump straight to the closing character marking the end of the subtree rooted at the parent of v, and we know that this character will be '`}`'. Moving on with our example, after reading the character '`}`' marking the end of the subdocument `a:{...}`, no matter if we have found some query matches there or not, we know that there can be no more properties with label `a` in the root object, so we can jump directly to the closing brace.

Skipping to a label. We call a state *waiting* if it has exactly one transition over a concrete label ℓ and its fallback transition is looping. Such a state corresponds to descendant selectors $..\ell$. When the simulated DFA is in such a state, it would make sense to jump directly to the next descendant of the current node that has label ℓ. This is hard in general, as it requires monitoring the depth and the label, but if the waiting state is the initial state of the automaton (this is the case for queries that begin with $\$..\ell$) then the current node is the root and we can jump to the next occurrence of label ℓ.

3.4 Main algorithm

The pseudocode of the main algorithm, shown below in a Python-like syntax for brevity, combines the depth-stack based algorithm with skipping leaves, children, and siblings. It uses an iterator that abstracts away all access to the stream. The iterator allows advancing to the next relevant structural character with method `next()` and peeking it without advancing with method `peek()`. Both these methods yield items of an algebraic sum type with four variants: `Closing` for '`{`' and '`[`', `Colon` for '`:`', `Comma` for '`,`', and `Opening` for '`}`' and '`]`'. Whenever an opening character is found, the method `get_label()` is used to get the label corresponding to the subdocument. The iterator does that simply by backtracking through whitespace characters (and possibly a colon) to the label and returning it. If instead it finds a comma or an opening character, it means that the encompassing element is an array and there is no label. In that case, an artificial label is returned, falling under the fallback transition of the simulated automaton.

By default, the iterator returns only opening and closing characters, which amounts to skipping leaves. The method `toggle()` checks if the automaton can accept in a single step from the current state in the current type of element (object or list). If so, it extends the set of structural characters to be iterated over: it adds commas if the current element is a list and colons if it is an object. (It would be possible to use commas in objects as well, but we found the current solution more efficient.) We make sure that only leaves are processed in cases `Colon` and `Comma` by checking if the next structural character is not an `Opening` character. If it is, the current node is not a leaf and it is handled in the main two cases. An additional corner case is the first item of an array, which will not be caught in the `Comma` case nor in the `Opening` case if it is leaf. We handle it using `try_match_first_item()` in the `Open('[')` case for the encompassing array, which reports an answer if: the array is non-empty, the next structural character is not opening, and the target of the fallback transition is accepting.

```
state = automaton.init_state()
stack = init_stack()
while event = iterator.next():
  match event:
    case Opening(c):
      label = iterator.get_label()
      target = state.transition(label)
      if target.is_rejecting():
        iterator.skip(c)
        continue
      if target != state:
        stack.push(state, depth, c)
        state = target
      depth = depth + 1
      if state.is_accepting():
        report_match()
      iterator.toggle(state, c)
      if c == '[':
        try_match_first_item()
    case Closing(c):
      depth = depth - 1
      prev = stack.top()
      if depth == prev.depth:
        state = prev.state
        stack.pop()
        if state.is_unitary():
          iterator.skip('{')
          continue
        iterator.toggle(state, prev.c)
      else:
        iterator.toggle(state, '{')
    case Colon:
      if iterator.peek() == Opening(_):
        continue
      label = iterator.get_label()
      target = state.transition(label)
      if target.is_accepting():
        report_match()
      if state.is_unitary():
        iterator.skip('{')
    case Comma:
      if iterator.peek() != Opening(_):
        report_match()
```

Table 1: JSON Structural Characters

Character	{	}	[	]	:	,
UTF	0x7b	0x7d	0x5b	0x5d	0x3a	0x2c

Skipping children and siblings is handled by calling the method `skip()` for the opening character that needs to be closed. Skipping to label is implemented outside of the above algorithm: if the query starts from a descendant selector $..\ell$, the engine finds the first occurrence of ℓ in the stream using the highly optimized SIMD-based `memmem` function from the `memchr` crate [9], and runs the described algorithm from there. When the whole subdocument associated with the first occurence of ℓ is processed, the external loop identifies the next occurrence of ℓ and runs the algorithm again, and so on.

4 VECTORISED CLASSIFICATION

The core of our engine is a state-driven SIMD pipeline that quickly locates relevant characters and fast-forwards through irrelevant fragments of the stream, allowing to implement the iterator used in the main algorithm from Section 3.4. The pipeline consists of: a *structural classifier* that recognizes structural characters shown in Table 1 (with the possibility to toggle commas and colons) and fast-forwards through whitespaces, labels, and atomic values; a *depth classifier* used to fast-forward through entire subdocuments; and a *quote classifier* that detects characters placed between quotes and handles escaping.

The structural classifier is obtained as a special case of a general method for *raw classification* (Section 4.1), which can be of use for fast parsing of other document formats. Along with the quote classifier (Section 4.2), it underlies the main functionality of the iterator (Section 4.3), whereas the depth classifier powers skipping (Section 4.4). Together, the classifiers build up a flexible and extensible multi-classifer pipeline (Section 4.5).

4.1 Raw classification

Our algorithm has to solve a special case of a more general *classification problem*, asking to classify an input vector of n bytes into k buckets. It can be stated formally as:

Problem 1 (Classification). *Fix a classification function* $f : \{\mathtt{0x00}, \mathtt{0x01}, \ldots, \mathtt{0xff}\} \longrightarrow \{0, 1, \ldots, k-1\}$. *Given a vector* v *of* n *bytes compute the vector* $[f(v_0), f(v_1), \ldots, f(v_{n-1})]$.

We will show that binary classification ($k = 2$) can be efficiently solved using few SIMD instructions, assuming some constant vectors are precomputed. A simple solution is to compare (vectorially) with each value mapped to 1 separately, and aggregate the answers using bitwise OR. This method is very fast when few values are mapped to 1, but becomes costly as their number grows (Table 2).[1] For 5 and more values one can do better.

Following [26], we use the `shuffle` instruction, which essentially allows one to compose functions represented as vectors, except that the values of the inner vector are trimmed to the lower *nibbles* (4-bit parts). For 128-bit vectors a and b, the result of `shuffle_epi8(a, b)` is a 128-bit vector with `a[b[i] & 0x0F]`

[1]Here and elsewhere we ignore the final `movemask` instruction needed in any SIMD solution.

Table 2: Naive classification on Intel Alderlake. Cycle values computed from throughput and latency specification [23] assuming maximum CPU pipelining.

Values	1	2	3-4	5-6	7	8	16	32	64	128
Cycles	1	2	4	6	7	8	14	28	54	108

at postion i, for all $0 \le i < 16$ (assuming that the upper nibbles of b are zeroed).[2] For instance, if $b = [15, 14, \ldots, 0]$, then `shuffle_epi8(a, b)` is the reverse of a.

While the original purpose of `shuffle` was to reorder entries of vector a based on vector b, as in the example above, one can also think of it as a way to perform parallel lookup in a table a at the positions indicated by vector b. We can use it to quickly classify a vector b of bytes into at most 16 buckets depending on their lower nibble. We can do the same for the upper nibbles (by simply shifting all bytes right by 4). However, not every classification function over bytes can be easily factorized into two classification functions over nibbles. In what follows we describe two cases when this is possible. In both cases we construct two lookup tables such that the results of the lookups can be easily combined to provide the final classification. We also describe an original working solution for the general case, readily applicable wherever fast lexing is needed.

Fix a binary classification function f. We identify each byte with a pair of an upper and lower nibble; e.g. `0x3a` is identified with $\langle 3, \mathtt{a}\rangle$. We let $Nib := \{\mathtt{0}, \mathtt{1}, \ldots, \mathtt{e}, \mathtt{f}\}$.

Definition 1. *The* acceptance set $low(u)$ *of a nibble u is the set of all nibbles l such that $\langle u, l\rangle$ is accepted, i.e.*

$$low(u) = \{l \in Nib \mid f(\langle u, l\rangle) = 1\}.$$

Definition 2. *An* acceptance group *is a defined by a maximal set of upper nibbles that have the same acceptance sets. For a nibble u let $U_u = \{u' \in Nib \mid low(u') = low(u)\}$. The set G of all acceptance groups is defined as*

$$G = \{\langle U_u, low(u)\rangle \mid u \in Nib\}.$$

Note that $|G| \le |Nib| = 16$.

Definition 3. *Acceptance groups $\langle U_1, L_1\rangle, \langle U_2, L_2\rangle \in G$ are* overlapping *if $U_1 \ne U_2$ and $L_1 \cap L_2 \ne \emptyset$.*

As an example, consider a classifying function that assigns 1 to bytes `0xa1`, `0xa2`, `0xb1`, `0xb2`, `0xc2` and 0 to the remaining bytes. Then:

$$low(\mathtt{a}) = low(\mathtt{b}) = \{\mathtt{1}, \mathtt{2}\}, \quad low(\mathtt{c}) = \{\mathtt{2}\},$$

$$G = \{\langle\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{1}, \mathtt{2}\}\rangle, \langle\{\mathtt{c}\}, \{\mathtt{2}\}\rangle\},$$

and the two groups in G are overlapping, since they share the element 2.

Depending on the properties of the set G of all acceptance groups we distinguish three cases of increasing complexity for the classification problem.

[2]Zeroing the upper nibbles is important because the exact semantics of `shuffle` make bits there meaningful – if the most significant bit is lit the result is mapped to zero. For all our purposes we want the upper nibbles to be zeroed.

Non-overlapping groups. If G contains no overlapping groups, then the simplest solution is to map lower and upper nibbles from a single group to a unique value and compare the results with `cmpeq`. Take an arbitrary enumeration $\langle U_1, L_1\rangle, \ldots, \langle U_{|G|}, L_{|G|}\rangle$ of groups in G. Then construct an upper table as a vector $utab$ such that $utab[x] = i$ if $x \in U_i$, and $utab[x] = 254$ if there is no such i. Analogously construct a lower table $ltab$ as $ltab[x] = i$ if $x \in L_i$, and $ltab[x] = 255$ if there is no such i. Naturally, $|G| < 254$. Then the required classification vector b is obtained with:

```
let usrc = shiftright_epi8(src, 4);
let llookup = shuffle_epi8(ltab, src);
let ulookup = shuffle_epi8(utab, usrc);
let b = cmpeq_epi8(llookup, ulookup);
```

x86 SIMD does not have the `shiftright_epi8` instruction, but it can be simulated with two instructions: first, 16-bit right shift by 4, then zero the upper nibbles with a precomputed mask. Both these instructions have latency 1. The total cost of the entire lookup is thus five SIMD operations, each of which has latency 1. The two shuffles can be effectively locally parallelised by the CPU, so the expected time of execution is four cycles.

As it happens, the non-overlapping case is sufficient for our JSON structural classifier. JSON structural characters are shown in Table 1, and the groups are

$$\{\langle\{\mathtt{5}, \mathtt{7}\}, \{\mathtt{b}, \mathtt{d}\}\rangle, \langle\{\mathtt{2}\}, \{\mathtt{c}\}\rangle, \langle\{\mathtt{3}\}, \{\mathtt{a}\}\rangle\},$$

and they are non-overlapping. Consequently, the lower and upper lookup table used in our classifier are

$$\begin{aligned} utab = [&\mathtt{0xfe}, \mathtt{0xfe}, \mathtt{0x02}, \mathtt{0x03}, \mathtt{0xfe}, \mathtt{0x01}, \mathtt{0xfe}, \mathtt{0x01}, \\ &\mathtt{0xfe}, \mathtt{0xfe}, \mathtt{0xfe}, \mathtt{0xfe}, \mathtt{0xfe}, \mathtt{0xfe}, \mathtt{0xfe}, \mathtt{0xfe}], \\ ltab = [&\mathtt{0xff}, \mathtt{0xff}, \mathtt{0xff}, \mathtt{0xff}, \mathtt{0xff}, \mathtt{0xff}, \mathtt{0xff}, \mathtt{0xff}, \\ &\mathtt{0xff}, \mathtt{0xff}, \mathtt{0x03}, \mathtt{0x01}, \mathtt{0x02}, \mathtt{0x01}, \mathtt{0xff}, \mathtt{0xff}]. \end{aligned}$$

Compared with the naive method (Table 2), we save two CPU cycles on each block. Simdjson [26] uses the same code with the same lookup tables, but does not abstract the general method as we do here.

An additional challenge in rsonpath is that sikipping leaves requires toggling commas and colons on request. This has an elegant solution. We can simultaneously disable all symbols with upper nibble b by zeroing `utab` at position b. Because `ltab` contains only non-zero values, `cmpq` will return 0 for all symbols with upper nibble b. As commas and colons do not share their upper nibble with any other accepted symbol, we can toggle them independently by XOR-ing `utab` with the following bitmasks:

$$\begin{aligned} toggle\_comma &= [\mathtt{0x00}, \mathtt{0x00}, \mathtt{0x02}, \mathtt{0x00}, \mathtt{0x00}, \ \ldots \ , \mathtt{0x00}], \\ toggle\_colon &= [\mathtt{0x00}, \mathtt{0x00}, \mathtt{0x00}, \mathtt{0x03}, \mathtt{0x00}, \ \ldots \ , \mathtt{0x00}]. \end{aligned}$$

When both commas and colons are disabled, our classifier has only 4 symbols to accept, but at 4 CPU cycles it is not slower than the naive method (see Table 2).

Few groups. Another case that can be efficiently solved is when $|G| \le 8$. The idea is to assign a unique index from 0 to 7 to each group and then let the upper nibble lookup zero the bit at the index corresponding to the unique group containing the nibble, and the lower nibble lookup set bits at all indices corresponding to groups whose acceptance set contains the nibble. That is, we take an arbitrary enumeration $\langle U_1, L_1\rangle, \ldots, \langle U_{|G|}, L_{|G|}\rangle$ of groups in G.

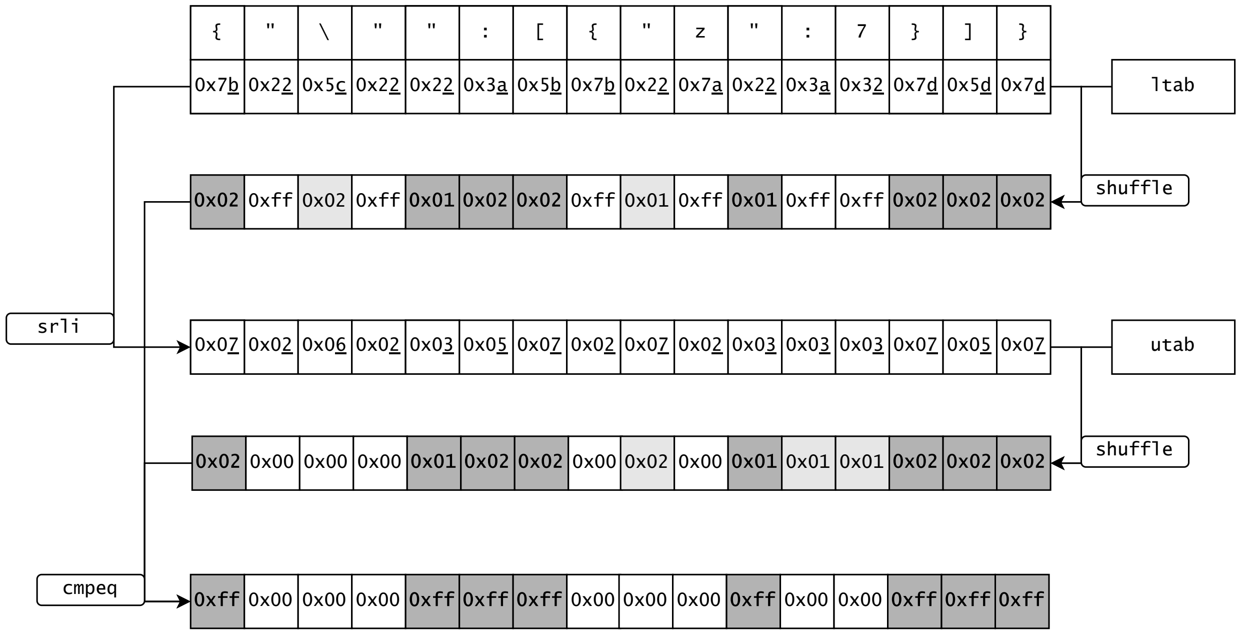

{ " \ " " : [{ " z " : 7 }] }
0x7b 0x22 0x5c 0x22 0x22 0x3a 0x5b 0x7b 0x22 0x7a 0x22 0x3a 0x32 0x7d 0x5d 0x7d
ltab
shuffle
0x02 0xff 0x02 0xff 0x01 0x02 0x02 0xff 0x01 0xff 0x01 0xff 0xff 0x02 0x02 0x02
srli
0x07 0x02 0x06 0x02 0x03 0x05 0x07 0x02 0x07 0x02 0x03 0x03 0x03 0x07 0x05 0x07
utab
shuffle
0x02 0x00 0x00 0x00 0x01 0x02 0x02 0x00 0x02 0x00 0x01 0x01 0x01 0x02 0x02 0x02
cmpeq
0xff 0x00 0x00 0x00 0xff 0xff 0xff 0x00 0x00 0x00 0xff 0x00 0x00 0xff 0xff 0xff

Figure 3: JSON document classified using the structural classifier's `ltab` and `utab` lookup tables.

We construct the upper table $utab$ such that $utab[x] = 2^8 - 1 - 2^{i-1}$ if $x \in U_i$, and $utab[x] = 0$ if no such i exists. The lower table is defined as

$$ltab[x] = 2^{i_1 - 1} + 2^{i_2 - 1} + \ldots + 2^{i_c - 1},$$

where $x \in L_{i_1}, x \in L_{i_2}, \ldots, x \in L_{i_c}$. Then, for every byte $b = \langle u, l \rangle$, $f(b) = 1$ if and only if the bitwise OR of $utab[u]$ and $ltab[l]$ is $2^8 - 1$. The classification vector b is obtained with just one more operation than in the non-overlapping case, increasing the expected time to five CPU cycles:

```
let usrc = srli_epi8(src, 4);
let llookup = shuffle_epi8(ltab, src);
let ulookup = shuffle_epi8(utab, usrc);
let lookup = or(llookup, ulookup);
let b = cmpeq_epi8(lookup, ALL_ONES);
```

This is faster than the naive method for 5 or more symbols.

General case. We have not found an elegant solution to the general case for $8 < |G| \leq 16$. A working approach is to apply the algorithm for the small case twice. First partition the set G into G_1, G_2 such that $|G_1| \leq 8$ and $|G_2| \leq 8$. Then classify the bytes according to G_1 and G_2, possibly with local parallelism. In the end, we take the OR of both classifications to obtain a classification for G.

```
let usrc = srli_epi8(src, 4);
let llookup1 = shuffle_epi8(ltab1, src);
let ulookup1 = shuffle_epi8(utab1, usrc);
let lookup1 = or(llookup1, ulookup1);
let llookup2 = shuffle_epi8(ltab2, src);
let ulookup2 = shuffle_epi8(utab2, usrc);
let lookup2 = or(llookup2, ulookup2);
let lookup = or(lookup1, lookup2);
let b = cmpeq_epi8(lookup, ALL_ONES);
```

Assuming maximal local parallelism this takes seven CPU cycles, since the two lookups are independent. This is always faster than the naive method (Table 2), because the number of accepted symbols is at least $|G| \geq 8$.

4.2 Recognising quoted sequences

The next step is to ignore characters recognised as structural by the lookup, but located inside JSON strings, that is, between unescaped double quotes. A double quote is escaped if and only if it is preceded by a sequence of backslashes of odd length: `"x\""` contains a single escaped double quote, while `"x\\"` contains a single escaped backslash and none of the double quotes are escaped.

We use the same solution as simdjson [26]. First, we mark all backslash and quote characters with a `cmpeq`. Then we mark starts of backslash sequences and use *add-carry propagation* to find their ends. Based on this we mark unescaped double quote characters. Finally, we compute the prefix-xor of the resulting bitmask using *carry-less multiplication* (the `clmul` instruction) by a vector with all bits lit, and the bits lit in the result indicate the positions between quotes.

We deal with block boundaries by maintaining a state storing two bits of information: whether the previous block's last character was an unescaped backslash and whether the last block ended while still within quotes.

4.3 Structural iterator

All the classification we have performed up to this point was on a single block of data. To feed information to the main algorithm we need an abstraction on top of a block classifier that will give us a classifier for the entire input stream. As explained in Section 3.4, the abstraction is provided by an iterator.

The iterator operates on classified blocks of structural characters, which contain a bitmask with all structural characters marked as described in the sections above, along with a reference to the original input block. The iterator begins by classifying the first block of input. Then, when asked for the next structural character, it examines the current classified block and its bitmask. If it is all-zeroes, then there are no structural characters in the current block and we need to classify the next one. If it is not, then we extract the position of the first lit bit by calculating trailing zeroes of the mask and check the original input block for the character located at that position. Additionally, we modify the stored bitmask by zeroing the lit bit we just processed.

While the SIMD pipeline calculating structural bitmasks is completely branchless, the outer loop of the iterator contains branching when we compare the mask to zero, and then when we branch on the input character to return a proper variant of the algebraic sum type.

The method `toggle()` enables and disables colons and commas by XOR-ing the upper lookup tables with precomputed bit masks, as explained in Section 4.1. When a character is enabled, the current block has to be reclassified. When a character is disabled, its occurrences in the reminder of the block are simply stepped over by the outer loop of the iterator.

4.4 Depth classifier

Skipping children and siblings is implemented by iterating over closing characters (of the suitable kind) and checking the relative depth at each time, until it drops to 0 for the first time. We use a specialised *depth classifier* here, because it has fewer characters to classify but additionally needs to maintain the depth.

We need to keep track of two characters only: either '`[`' and '`]`', or '`{`' and '`}`'. Hence, instead of using the complex method of the structural classifier (5 SIMD instructions), we can mark all opening and closing characters in the block using two `cmpeq` instructions. This results in two separate bitmasks, rather than a common bitmask for all relevant characters. As it happens, this suits us even better, because we can now find the next closing character by locating the first lit bit in the respective bitmask and we can update the relative depth by counting bits lit in the relevant chunk of the other bitmask.

We also use an additional heuristic that goes beyond JSONSki's methods. If the number of closing characters in the whole block is strictly smaller than the current relative depth, nowhere in the block the relative depth can drop to 0. We then jump directly to the end of the block instead of iterating through all closing characters and checking the relative depth for each of them. We update the relative depth based on the total numbers of opening and closing symbols in the block. This makes the algorithm much more efficient, especially on real-life JSON documents

4.5 Multi-classifier pipeline

As skipping children and siblings is performed on demand, we need to be able to trigger depth classification at any moment, pausing the full structural classifier for the duration of the fast-forward. After we are done skipping, we need to resume structural classification. To facilitate this we propose a robust multi-classifier pipeline ready for extension with different classifiers in the future.

Our pipeline consists of the core classifier, the *quote classifier*, which is responsible for creating the bitmasks marking characters that are within quotes and should be ignored. This classification is always required to maintain correctness. On top of that, we can run either the structural or the depth classifier. To allow quick switching between them, they both expose `stop` and `resume` methods. The `stop` returns an object that encapsulates the state of the underlying quote classifier – all of its internal structures and the last classified block along with the index to which classification was performed. The `resume` counterpart can take such an object, restore the quote classifier, and begin the top-level depth or structural classification. By leveraging Rust's ownership system we can easily pass the internal quote classifier structures between components without copying (which would be slow) or complicated memory management (which would be error-prone and hard to modify): Rust statically ensures that only one classifier has write access to the shared structures at a time.

While our implementation has only two classifiers in the pipeline, one can easily add further specialised classifiers by simply providing them with appropriate `stop` and `resume` methods. One can envision a progression of more and more refined classifiers. Our depth classifier allows to keep track of the depth to stay within an element. This can be extended to a classifier that allows to fast-forward to the next occurrence of a label within an object. Such a classifier could be leveraged to speed up the execution of nested descendant selectors. Finally, one could additionally demand staying at the same depth, which would be useful in processing child states.

We believe that a flexible pipeline is an important engineering milestone that can enable programmers to construct SIMD solutions that are naturally composable and easily modifiable. This increases the level of abstraction on which we operate when designing SIMD accelerations, while not sacrificing low-level control required to produce very performant code.

5 EXPERIMENTS

We performed a series of experiments comparing our engine with existing solutions. Our goals were three-fold: estimate the overall overhead incurred by supporting descendants and idiomatic wildcards; showcase the benefits from working directly with descendant queries rather than semantically equivalent descendant-free queries; and identify improvement opportunities. The benchmark code and all datasets are available online [13].

5.1 Setup

We use Rust Criterion [22] as the benchmarking harness. A benchmark is executed as follows. First, a warm-up is performed, ensuring that the low-level caches are filled for the actual measurements. Measurements are performed on a number of samples, each of which consists of many iterations of the benchmarked routine. The mean execution time of all iterations is taken as a single sample.

Finally, collected samples are analysed to find the statistical distribution, outliers are detected, and a mean time is reported. We calculate throughput based on that.

All experiments were run in a stable environment, on machines hosted by Grid'5000, a federated testbed setup providing a high diversity of architectures [20]. We show the results for the *Chetemi* nodes which offer *Intel Xeon E5-2630 v4* CPUs (Intel Broadwell) with 256 GiB of (8x16GiB) of 24000 MHz RAM. Plots for further architectures (Intel Skylake, Intel Cascade-Lake, AMD Zen 2) can be found in the Appendix B.

5.2 Competitors

For the baseline we chose JsonSurfer [34], which works in the streaming model and supports full JSONPath, but does not apply any SIMD optimizations. We exclude the very popular jq, which supports a much more expressive variant of JSONPath, because it is much slower than JsonSurfer (one order of magnitude slower on A1).

Our main point of reference is JSONSki [25], to the best of our knowledge the only JSONPath engine using SIMD optimization. We do not include simdjson [26] in the comparison, because it does not support JSONPath queries directly; instead, queries have to be implemented by hand using an on-demand API. From previous studies we know that JSONSki performs better [25].

We ran JsonSurfer without any modifications via JNI[3] [31]. For JSONSki we introduced a few technical tweaks to integrate it with our Rust-based benchmark harness. First, we removed gathering results with `std::vector` in favour of a simple match counter. This slightly increases the performance of JSONSki as compared to the original [25]. Second, we fixed a few memory-management issues that become apparent when running the engine many subsequent times within the same process. These were identified by running the executable under Valgrind and are minor changes in constructor and destructor code, with virtually no effect on the actual query performance.

5.3 Datasets

In our experiments we use the 6 datasets from the JSONSki benchmark [14–19] as well as 3 additional datasets representing characteristic usecases. The first one is the `twitter.json` file extracted from simdjson's quick-start tutorial in the project's repository [33]. It is a typical file obtained by querying an API, small but irregular. The second dataset is an arbitrarily chosen fragment of the datadump [5] of the Crossref service [4] collecting metadata of several hundred millions of scientific publications. The document is highly regular: it contains collections of sub-documents of very similar shape. The third dataset is a deep and highly irregular file: it is the abstract syntax tree of a single large C file (23K lines of code) obtained with `clang`. It falls within the code-as-data application scenario mentioned in Section 1.2. Detailed characteristics of the datasets are shown in Table 3; the asterisk indicates added datasets. By *verbosity* we mean the ratio of the size of the document to the number of nodes in the underlying tree: the lower the verbosity, the harder it is to achieve high throughput. The added datasets are accessible through Zenodo [7]: AST [12], Crossref [10], Twitter [11].

[3]The overhead introduced due to Rust-Java interop amounts to nanoseconds per call, which is negligible considering the query execution times are in the milliseconds range

Table 3: Datasets used in experiments

Name	Size [MB]	Depth	Verbosity
* AST (A)	25.6	102	14.3
BestBuy (B)	1044.6	8	24.5
* Crossref (C)	550.5	9	27.0
GoogleMap (G)	1136.1	10	36.9
NSLP (N)	1210.2	10	13.8
Twitter (T)	842.5	12	29.0
* Twitter small (Ts)	0.7	11	50.6
Walmart (Wa)	995.4	5	96.9
Wikimedia (Wi)	1099.0	13	18.7

5.4 Experiment A: Overhead

Here we aimed to estimate how the added support for descendants and idiomatic wildcards influences the performance on descendant-free queries. We compared the performance of jsurfer, JSONSki, and rsonpath on the JSONSki benchmark (Table 4). We modified queries B1 and N2 by replacing slices (which we do not support) with wildcards, adding more equivalent work for all implementations, and dropped the other Wikipedia query because it coincides with Wi when slice is replaced with wildcard. We also added one query, indicated with *.

The results (Figure 4) show that the multi-classifier pipeline not only incurs no overhead, but actually boosts the performance by 10-20%. With no SIMD acceleration to muster, jsurfer consistently performs an order of magnitude worse (missing bar indicates that jsurfer was unable to load the dataset into memory due to JVM limitations).

We added query B3 to explore the effect of JSONSki's assumption that wildcard only traverses array elements. B3 was obtained from B2 by removing the last two selectors. It returns much fewer answers but, because of the missing wildcard selector, JSONSki cannot conclude that `.videoChapters` must match an array and needs to check objects and atomic values as well. In consequence, JSONSki is around three times faster on B2 than on B3. While JSONSki exploits this assumption for each wildcard selector, it does not give it an edge one might expect. Interestingly, rsonpath is also slower on B3, because it cannot apply leaf skipping any more.

On query N1 (not plotted), after processing the `meta` element contained in a small initial fragment of the document, JSONSki exits immediately instead of skipping the remaining children of the root in the usual way by fast-forwarding to the corresponding closing character. Because the overall size of the document is large, this gives terabytes per second of throughput, but is not a very meaningful statistic. We have not implemented this optimization in rsonpath, but it is easy to add.

5.5 Experiment B: Benefits from descendants

Here we explore how the performance changes when queries are reformulated using descendants (without altering the semantics). Whenever such rewritings exist, they are simpler than the originals (Table 5). The results, shown in Figure 5, confirm that descendants

Table 4: JSONSki queries

ID	Query	Matches
B1	$.products.*.categoryPath.*.id	697440
B2	$.products.*.videoChapters.*.chapter	8857
*B3	$.products.*.videoChapters	769
G1	$.*.routes.*.legs.*.steps.*.distance.text	1716752
G2	$.*.available_travel_modes	90
N1	$.meta.view.columns.*.name	44
N2	$.data.*.*	8774410
T1	$.*.entities.urls.*.url	88881
T2	$.*.text	150135
W1	$.items.*.bestMarketplacePrice.price	15892
W2	$.items.*.name	272499
Wi	$.*.claims.P150.*.mainsnak.property	15603

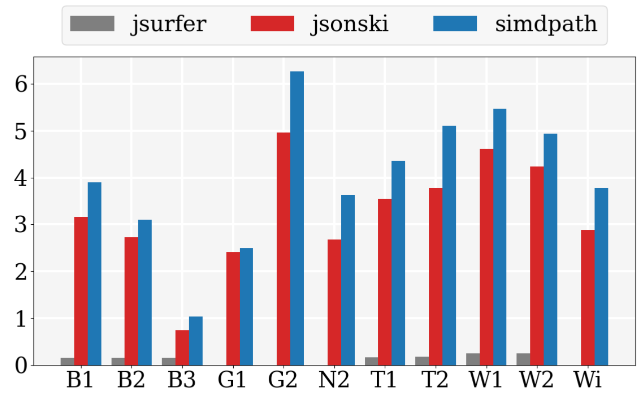


Figure 4: Throughput [GB/s] for descendant-free queries.

Table 5: JSONSki queries rewritten with descendants

ID	Query	
$B1^r$	$..categoryPath..id	697440
$B2^r$	$..videoChapters..chapter	8857
$*B3^r$	$..videoChapters	769
$G2^r$	$..available_travel_modes	90
$W1^r$	$..bestMarketplacePrice.price	15892
$W2^r$	$..name	272499
Wi^r	$..P150..mainsnak.property	15603

not only simplify formulating queries, but also significantly boost performance, reaching throughput above 10GBs for most tested queries. This is largely due to the memmem-based skipping to a label: rsonpath essentially jumps from one shallowest occurrence of the first label in the query to another. The boost is particularly impressive for the added query B3, whose original version is particularly hard for both JSONSki and rsonpath. The two queries with throughput below 10GB/s have an order of magnitude more matches than others, and the sheer cost of label comparison and the branching code to accumulate the results cannot be compensated by SIMD. Jsurfer performs orders of magnitude worse, essentially the same on the original and rewritten queries.

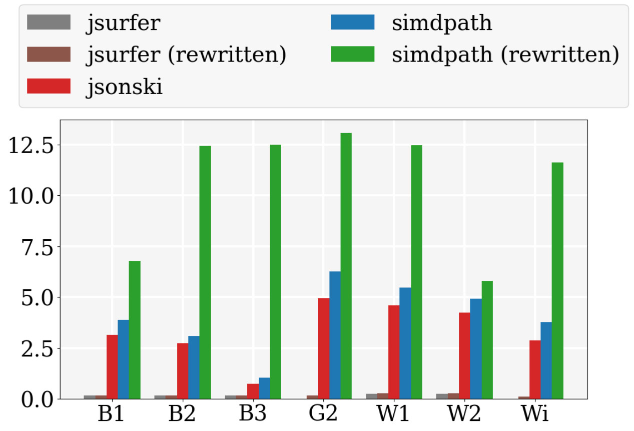


Figure 5: Throughput [GB/s] for descendant-free queries and their rewritings using descendants.

5.6 Experiment C: Limitations and Opportunities

In this experiment we aim to learn more about the behaviour of rsonpath in diverse scenarios. We use the new datasets and queries shown in Table 6; the results are shown in Figure 6.

Queries A1, A2, and C1 cannot be expressed without descendant because of relevant labels appearing at different levels. We plot them in green to put them visually in the same group with rewritings, as they are all descendant queries. Our target is the memmem-based skipping to a label. A1 illustrates how fast one can go for highly selective queries. On A2 the performance is very low (only x4 speedup over JsonSurfer). This is because `inner` labels are nested and the query is highly ambiguous, which makes the depth-stack grow deep, as described in Section 3.2. No good implementation for such queries is known, even theoretically. The purpose of C1 is to test the memmem acceleration in a very low selectivity situation. As expected, performance is mediocre because repeated calls to memmem result in numerous short fast-forwards, rather a few long ones. Yet, low selectivity (C1) does not slow rsonpath down as much as nested labels (A2).

Queries C2 and C3 illustrate how the gain from rewriting with descendants may depend on the structure of the document. The queries have the same structure, yet the gain for C2 is negligible. Indeed, the rewriting of C2 is hard for rsonpath. JSONSki only examines `author` nodes at depth 3 while rsonpath goes through all `author` nodes (12x more), and each time looks through the whole subdocument searching for affiliation names (and finds no additional ones). We would improve our performance on this query if we could fast-forward to a given label within an element (see Section 4.5): this would allow us to quickly discard authors without affiliations. In contrast, the rewriting of C3 is easy for rsonpath, because all editors are at the same level, so there are no additional subdocuments to process.

The query Ts^p is a partial rewriting of Ts: it keeps the label `search_metadata` even though it is semantically redundant. Thus, the query Ts, its rewriting (Ts^r), and partial rewriting (Ts^p), all return the same results, but specify the access path with varying

Table 6: Additional queries of interest

ID	Query	Matches
A1	\$..decl.name	35
A2	\$..inner..inner..type.qualType	78129
C1	\$..DOI	1073589
C2	\$.items.*.author.*.affiliation.*.name	64495
$C2^r$	\$..author..affiliation..name	64495
C3	\$.items.*.editor.*.affiliation.*.name	39
$C3^r$	\$..editor..affiliation..name	39
Ts	\$.search_metadata.count	1
Ts^r	\$..count	1
Ts^p	\$..search_metadata.count	1

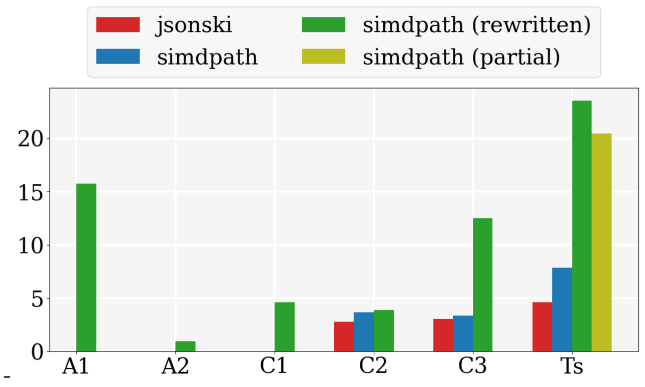


Figure 6: Throughput [GB/s] for additional queries and their rewritings.

precision. They show that in our approach the less specified the path, the better. On Ts rsonpath and JSONSki display similar performance. If we knew how to find the next label at the same depth (a sibling) we could do better, possibly closing the gap to the rewritten variants.

5.7 Experimend D: Scalability

To test the scalability of rsonpath on various architectures (see Appendix B), we ran the query `$..affiliation..name` on fragments of the Crossref dataset of increasing size (Table 7). As expected, we observed no significant variation.

Table 7: Throughput [GB/s] on the Crossref dataset

Size [GB]	0.3	0.5	1.1	2
Broadwell	7.9	7.8	7.9	7.8
Skylake	8.2	8.2	8.1	8.1
Cascade-Lake	8.3	8.1	8.1	8.1
Zen 2	9.8	10.0	10.1	9.8

6 LOOKING AHEAD

We have shown that it is possible to support descendants and idiomatic wildcards in SIMD-accelerated JSONPath processing without paying a penalty and that natural queries can be sped up significantly by using descendants. Supporting array indexing is compatible with our approach and we plan to implement it in the near future. Filters, on the other hand, are highly problematic because multiple potential matches must be stored until all filters are evaluated and it transpires which matches should be returned. This requires unbounded working memory and departs from the streaming paradigm. Another challenge is compositionality: processing queries in succession, with the output of one query fed directly to another.

The key lesson from our work is that it pays off to split complex SIMD optimization tasks into a simple lower layer with high potential for SIMD acceleration, and a stateful sequential upper layer. This facilitates optimizations that would otherwise be impossible to design. One can apply it to any stateful SIMD processing task: regexp matching, DNA processing, XML parsing and querying, etc. Moreover, the separation into a hardware-dependent branchless layer and a higher-level sequential layer is relevant not only for SIMD but also for FPGAs and ASICs, except that acceleration options are richer. Ideally, one would want to use the acceleration to compute matching parentheses in the stream, but even theoretically efficient algorithms for this task are not known.

REFERENCES

[1] Corentin Barloy, Filip Murlak, and Charles Paperman. Stackless processing of streamed trees. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS'21, page 109–125, New York, NY, USA, 2021. Association for Computing Machinery.

[2] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, yvon Jégou, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quétier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, 2006.

[3] Christoph Burgmer et al. json-path-comparison, 2019. https://github.com/cburgmer/json-path-comparison/commit/c0a5122a7c6ae8923550e7208d6443be79bc94d0.

[4] Crossref. Crossref. https:cd//www.crossref.org/.

[5] Crossref. Crossref datadump, 2022. https://www.crossref.org/blog/2022-public-data-file-of-more-than-134-million-metadata-records-now-available/.

[6] Stephen Dolan. jq. https://stedolan.github.io/jq/.

[7] European Organization For Nuclear Research and OpenAIRE. Zenodo, 2013. https://www.zenodo.org/.

[8] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. The robots are coming: Exploring the implications of OpenAI codex on introductory programming. In *Australasian Computing Education Conference*. ACM, February 2022.

[9] Andrew Gallant. `memchr`, 2015. https://crates.io/crates/memchr.

[10] Mateusz Gienieczko, Filip Murlak, and Charles Paperman. Extraction of Crossref datadump for benchmarking purpose, 2022. https://zenodo.org/record/7683117.

[11] Mateusz Gienieczko, Filip Murlak, and Charles Paperman. JSON file from Twitter API used for benchmarking JSONPath, 2022. https://zenodo.org/record/7229287.

[12] Mateusz Gienieczko, Filip Murlak, and Charles Paperman. JSON file of the AST of a C program, 2022. https://zenodo.org/record/7229268.

[13] Mateusz Gienieczko, Filip Murlak, and Charles Paperman. rsonpath, 2022. https://zenodo.org/record/7689885.

[14] Mateusz Gienieczko, Filip Murlak, and Charles Paperman. Bestbuy json dataset, January 2023. https://zenodo.org/record/7607865.

[15] Mateusz Gienieczko, Filip Murlak, and Charles Paperman. Google map json data file, January 2023. https://zenodo.org/record/7607889.

[16] Mateusz Gienieczko, Filip Murlak, and Charles Paperman. National Statistics Post-code Lookup: Json file, February 2023. https://zenodo.org/record/7607878.

[17] Mateusz Gienieczko, Filip Murlak, and Charles Paperman. tweets from twitter developer api, February 2023. https://zenodo.org/record/7607891.

[18] Mateusz Gienieczko, Filip Murlak, and Charles Paperman. Walmart product dataset, February 2023. https://zenodo.org/record/7607882.

[19] Mateusz Gienieczko, Filip Murlak, and Charles Paperman. Wikipedia entity dataset, February 2023. https://zenodo.org/record/7607884.

[20] Groupement d'Intérêt Scientifique. Grid'5000 hardware, 2018.

[21] Stefan Gössner. JSONPath, 2007. https://goessner.net/articles/JsonPath/.

[22] Brook Heisler. `criterion-rs`, 2017. https://crates.io/crates/criterion-rs.

[23] Intel Corporation. Intel intrinsics guide, 2022. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[24] Lin Jiang, Junqiao Qiu, and Zhijia Zhao. Scalable structural index construction for JSON analytics. *Proceedings of the VLDB Endowment*, 14(4):694–707, December 2020.

[25] Lin Jiang and Zhijia Zhao. JSONSki: Streaming semi-structured data with bit-parallel fast-forwarding. In *ASPLOS*, pages 200–211. ACM, 2022.

[26] Geoff Langdale and Daniel Lemire. Parsing gigabytes of JSON per second. *VLDB J.*, 28(6):941–960, 2019.

[27] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison. *Proceedings of the VLDB Endowment*, 10(10):1118–1129, June 2017.

[28] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Instant loading for main memory databases. *Proceedings of the VLDB Endowment*, 6(14):1702–1713, September 2013.

[29] Shoumik Palkar, Firas Abuzaid, Peter D. Bailis, and Matei A. Zaharia. Filter before you parse: Faster analytics on raw data with sparser. *Proc. VLDB Endow.*, 11:1576–1589, 2018.

[30] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. The software heritage graph dataset: Large-scale analysis of public software development history. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 1–5, 2020.

[31] Prevoty, Inc. and jni-rs contributors. jni. https://crates.io/crates/jni.

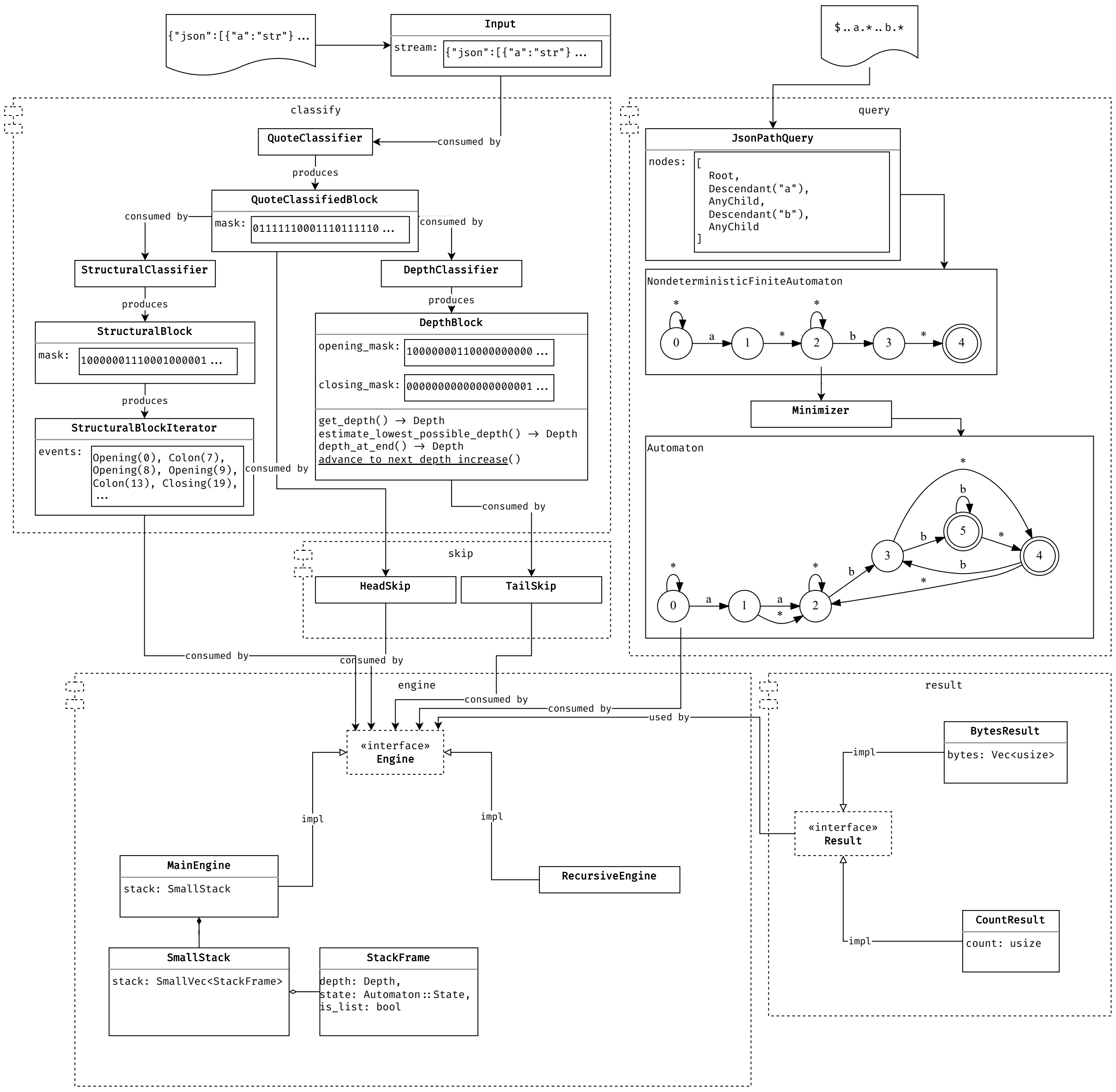
[32] simdjson. On-demand API. https://simdjson.org/api/0.6.0/md_doc_ondemand.html.

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

[34] Leo Wang. A streaming JSONPath processor in Java. https://github.com/jsurfer/JsonSurfer/.

[35] K Zyp. Rfc 6901: Javascript object notation (json) pointer, 2013.

A ARCHITECTURE OVERVIEW



$$\begin{tikzcd}
0 \arrow[loop above, "*"] \arrow[r, "a"] & 1 \arrow[r, "*"] & 2 \arrow[loop above, "*"] \arrow[r, "b"] & 3 \arrow[r, "*"] & 4
\end{tikzcd}$$

$$\begin{tikzcd}
0 \arrow[loop above, "*"] \arrow[r, "a"] & 1 \arrow[r, bend left, "a"] \arrow[r, bend right, "*"'] & 2 \arrow[loop above, "*"] \arrow[r, "b"] & 3 \arrow[r, "b"] \arrow[rr, bend left, "*"] & 5 \arrow[loop above, "b"] \arrow[r, "*"] & 4 \arrow[ll, bend left, "b"] \arrow[lll, bend left, "*"]
\end{tikzcd}$$

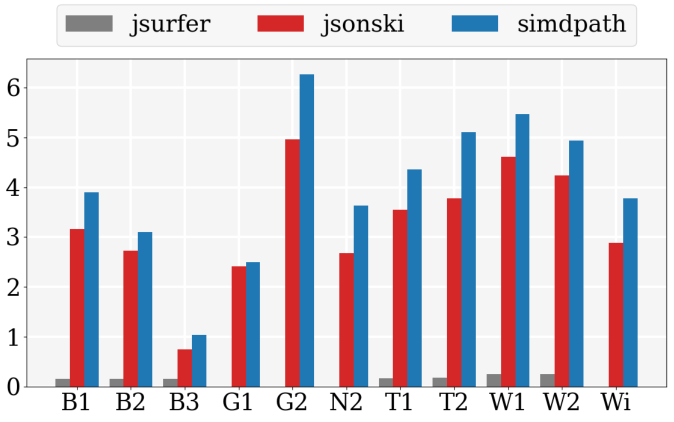
Figure 7: An overview of rsonpath's architecture

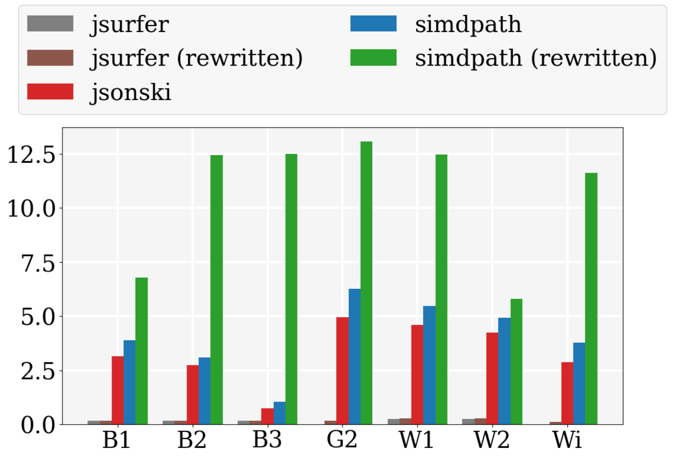
B ADDITIONAL EXPRIMENTS

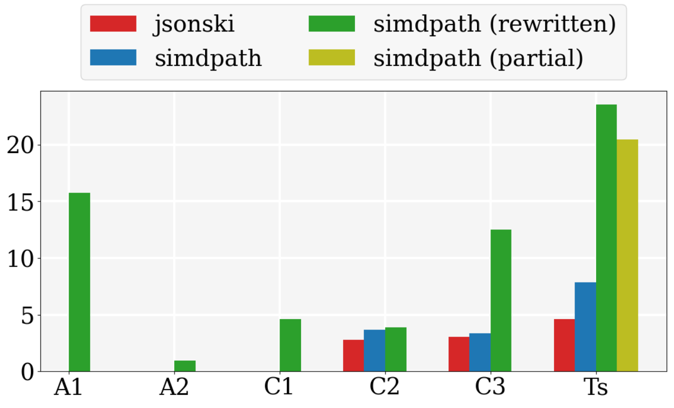
The CPU information is available from the G5000 website [2]. The RAM information is extracted by the output of the `lshw` command. All value are in GB/s

B.1 Broadwell (Chetemi): Throughput comparison

- CPU: Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz (Broadwell)
- RAM: 8x16GiB DIMM DDR4 Synchronous Registered (Buffered) 2400 MHz

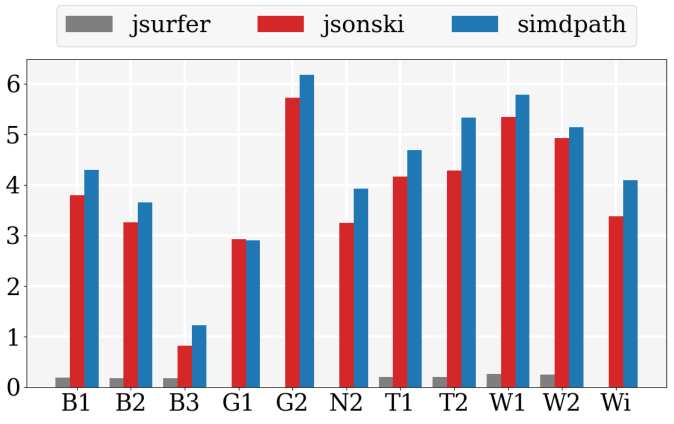


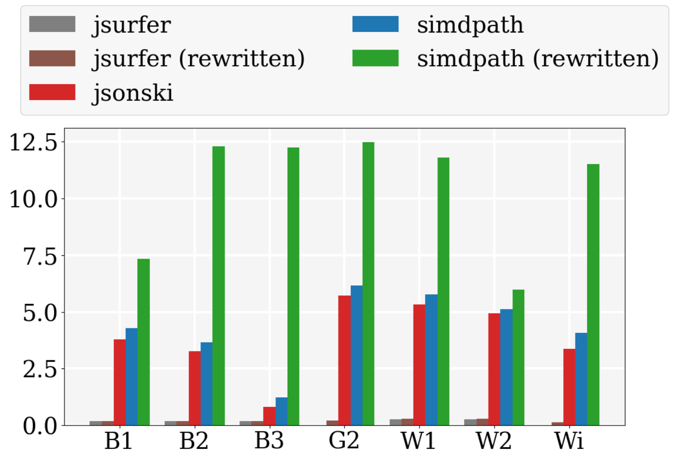


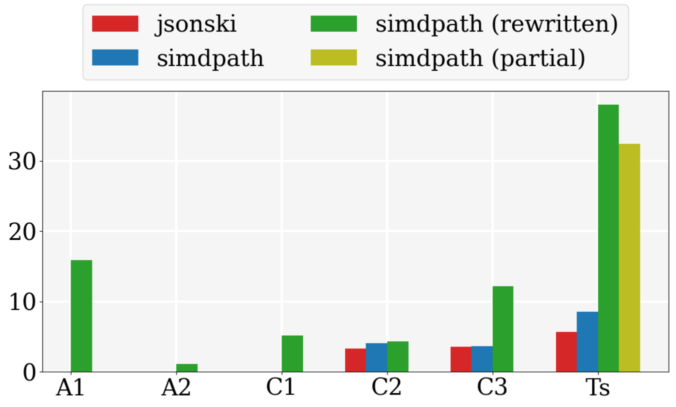


B.2 Skylake (Chifflot): Throughput comparison

- CPU: Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz (Skylake)
- RAM: 12x16GiB DIMM DDR4 Synchronous Registered (Buffered) 2666 MHz

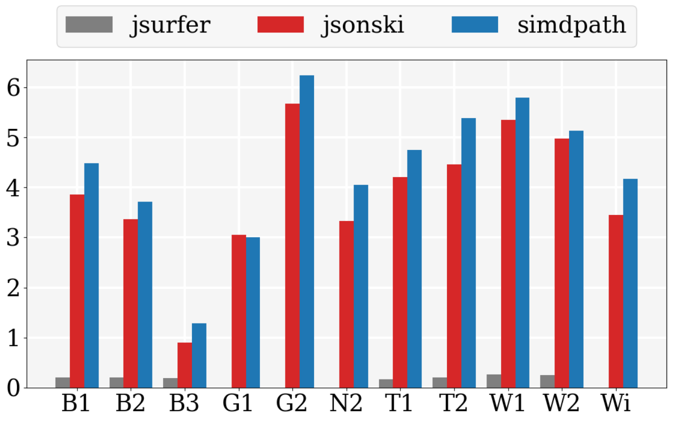


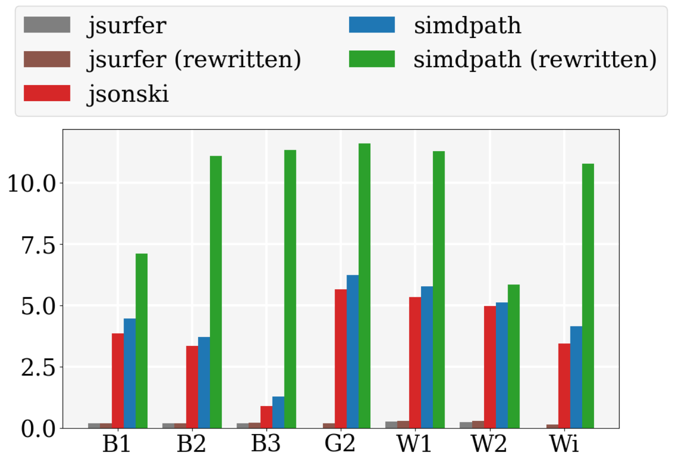


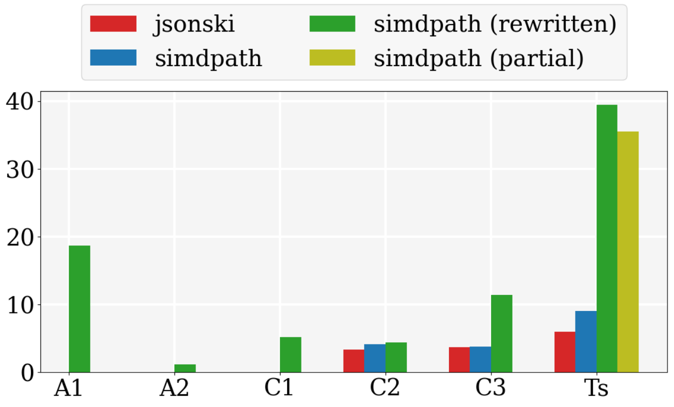


B.3 Cascade-Lake (Troll): Throughput comparison

- CPU: Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz (Cascade-Lake)
- RAM:
  - 12x32GiB DIMM DDR4 Synchronous Registered (Buffered) 2933 MHz
  - 12x128GiB DIMM DDR4 Synchronous Non-volatile LRDIMM 2666 MHz

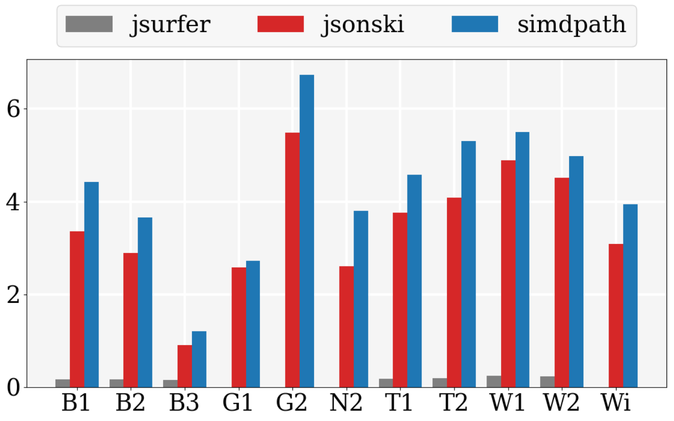


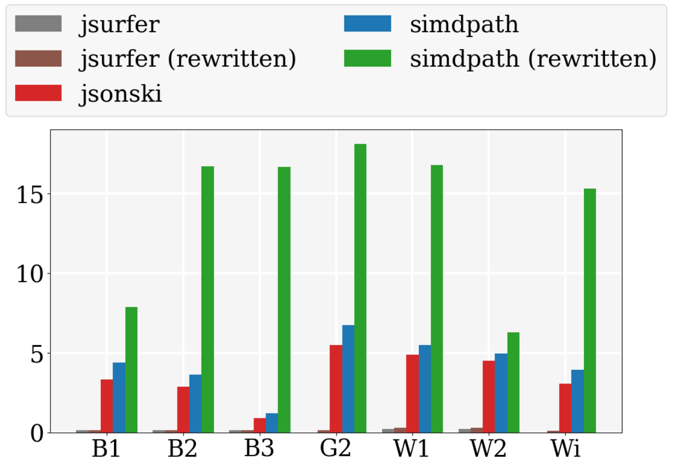


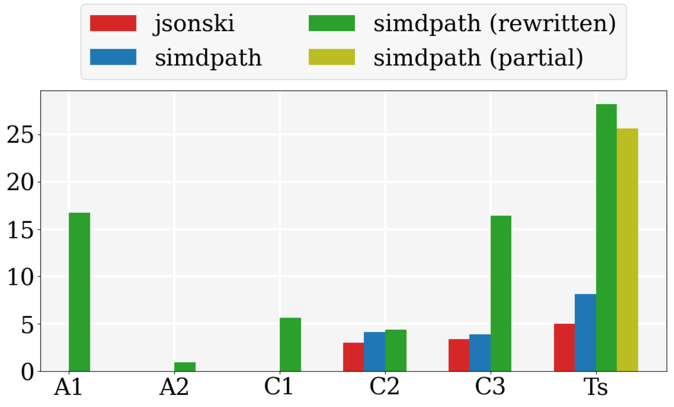


B.4 Zen 2 (Servan): Throughput comparison

- CPU: AMD EPYC 7352 (Zen 2)
- RAM: 8x16GiB DIMM DDR4 Synchronous Registered (Buffered) 3200 MHz







C RESULT TABULAR FORMAT

id	rsonpath_id	dataset	query	count
A1	decl_name	ast	$..decl.name	35
A2	nested_inner	ast	$..inner..inner..type.qualType	78129
A3	included_from	ast	$..loc.includedFrom.file	482
B1	BB1_products_category	bestbuy_large_record	$.products[*].categoryPath[*].id	697440
B1r	BB1'_products_category	bestbuy_large_record	$..categoryPath..id	697440
B2	BB2_products_video	bestbuy_large_record	$.products[*].videoChapters[*].chapter	8857
B2r	BB2'_products_video	bestbuy_large_record	$..videoChapters..chapter	8857
B3	BB3_products_video_only	bestbuy_large_record	$.products[*].videoChapters	769
B3r	BB3'_products_video_only	bestbuy_large_record	$..videoChapters	769
C1	DOI	crossref2	$..DOI	1073589
C2	author_affiliation	crossref2	$.items[*].author[*].affiliation[*].name	64495
C2r	author_affiliation_descendant	crossref2	$..author..affiliation..name	64495
C3	editor	crossref2	$.items[*].editor[*].affiliation[*].name	39
C3r	editor_descendant	crossref2	$..editor..affiliation..name	39
C4	title	crossref2	$.items[*].title	93407
C4r	title_descendant	crossref2	$..title	93407
C5	orcid	crossref2	$.items[*].author[*].ORCID	18401
C5r	orcid_descendant	crossref2	$..author..ORCID	18401
G1	GMD1_routes	google_map_large_record	$[*].routes[*].legs[*].steps[*].distance.text	1716752
G2	GMD2_travel_modes	google_map_large_record	$[*].available_travel_modes	90
G2r	GMD2'_travel_modes	google_map_large_record	$..available_travel_modes	90
N1	NSPL1_meta_columns	nspl_large_record	$.meta.view.columns[*].name	44
N2	NSPL2_data	nspl_large_record	$.data[*][*][*]	8774410
O1	vitamins_tags	openfood	$.products[*].vitamins_tags	24
O1r	vitamins_tags_descendant	openfood	$..vitamins_tags	24
O2	added_counties_tags	openfood	$.products[*].added_countries_tags	24
O2r	added_countries_tags_descendant	openfood	$..added_countries_tags	24
O3	specific_ingredients	openfood	$.products[*].specific_ingredients[*].ingredient	5
O3r	specific_ingredients_descendant	openfood	$..specific_ingredients..ingredient	5
S0	scalability_affiliation0	crossref0	$..affiliation..name	38352
S1	scalability_affiliation1	crossref1	$..affiliation..name	64535
S2	scalability_affiliation2	crossref2	$..affiliation..name	116187
S4	scalability_affiliation4	crossref4	$..affiliation..name	221443
T1	TT1_entities_urls	twitter_large_record	$[*].entities.urls[*].url	88881
T2	TT2_text	twitter_large_record	$[*].text	150135
Ts1	metadata_1	twitter	$.search_metadata.count	10
Ts2	metadata_2	twitter	$..search_metadata.count	2
Ts3	metadata_3	twitter	$..count	1
Ts4	all_hashtags	twitter	$..hashtags..text	1
Ts5	hashtags_of_retweets	twitter	$..retweeted_status..hashtags..text	1
W1	WM1_items_price	walmart_large_record	$.items[*].bestMarketplacePrice.price	15892
W1r	WM1'_items_price	walmart_large_record	$..bestMarketplacePrice.price	15892
W2	WM2_items_name	walmart_large_record	$.items[*].name	272499
W2r	WM2'_items_name	walmart_large_record	$..name	272499
Wi	WP1_claims_p150	wiki_large_record	$[*].claims.P150[*].mainsnak.property	15603
Wir	WP1'_claims_p150	wiki_large_record	$..P150..mainsnak.property	15603

machine	chetemi			chifflot			servan			troll		
engine	rsonpath	jsonski	jsurfer	rsonpath	jsonski	jsurfer	rsonpath	jsonski	jsurfer	rsonpath	jsonski	jsurfer
id												
A1	15.8		0.1	14.3		0.1	16.7		0.1	18.7		0.1
A2	1.0		0.1	1.1		0.1	1.0		0.1	1.2		0.1
A3	12.9		0.1	13.0		0.1	12.0		0.1	14.8		0.1
B1	3.9	3.2	0.2	4.4	3.9	0.2	4.4	3.4	0.2	4.5	3.9	0.2
B1r	6.8		0.2	7.3		0.2	7.9		0.2	7.1		0.2
B2	3.1	2.7	0.2	3.6	3.3	0.2	3.7	2.9	0.2	3.7	3.4	0.2
B2r	12.4		0.2	12.4		0.2	16.7		0.2	11.1		0.2
B3	1.0	0.8	0.2	1.2	0.8	0.2	1.2	0.9	0.2	1.3	0.9	0.2
B3r	12.5		0.2	12.4		0.2	16.7		0.2	11.3		0.2
C1	4.6		0.2	5.2		0.2	5.7		0.2	5.2		0.2
C2	3.7	2.8	0.2	4.0	3.3	0.2	4.2	3.0	0.2	4.2	3.4	0.2
C2r	3.9		0.2	4.3		0.2	4.4		0.2	4.4		0.2
C3	3.3	3.1	0.2	3.7	3.6	0.2	3.9	3.4	0.2	3.8	3.7	0.2
C3r	12.5		0.2	12.3		0.2	16.4		0.2	11.4		0.2
C4	3.3	2.9	0.2	3.6	3.4	0.2	3.7	3.1	0.2	3.7	3.5	0.2
C4r	9.2		0.2	9.4		0.2	11.9		0.2	9.2		0.1
C5	3.4	2.7	0.2	3.8	3.1	0.2	4.0	2.9	0.2	3.9	3.2	0.2
C5r	3.4		0.2	3.8		0.2	3.9		0.2	3.9		0.2
G1	2.5	2.4		2.9	2.9		2.7	2.6		3.0	3.1	
G2	6.3	5.0		6.4	5.9		6.7	5.5		6.2	5.7	
G2r	13.1		0.2	12.5		0.2	18.1		0.2	11.6		0.2
N1	6.9	∞		6.9	∞		7.6	∞		6.7	∞	
N2	3.6	2.7		4.1	3.3		3.8	2.6		4.0	3.3	
O1	2.6	1.8	0.1	3.1	2.0	0.2	3.0	2.0	0.2	3.3	2.2	0.2
O1r	20.2		0.1	28.8		0.2	34.6		0.1	30.9		0.2
O2	7.8	3.9	0.1	8.7	4.5	0.2	8.1	4.3	0.2	9.1	4.9	0.2
O2r	17.6		0.1	24.5		0.2	27.4		0.2	25.4		0.2
O3	3.6	2.6	0.1	4.3	3.0	0.2	4.2	2.9	0.2	4.5	3.2	0.2
O3r	16.5		0.1	22.6		0.2	26.0		0.1	23.4		0.2
S0	7.9			8.3			9.8			8.3		
S1	7.8			8.2			10.0			8.1		
S2	7.9			8.2			10.1			8.1		
S4	7.8			8.1			9.8			8.1		
T1	4.4	3.5	0.2	4.7	4.2	0.2	4.6	3.8	0.2	4.7	4.2	0.2
T2	5.1	3.8	0.2	5.4	4.4	0.2	5.3	4.1	0.2	5.4	4.5	0.2
Ts	7.9	4.6	0.2	8.6	5.6	0.3	8.2	5.0	0.3	9.0	6.0	0.3
Ts4	14.6		0.2	20.4		0.3	18.1		0.3	21.4		0.3
Ts5	8.2	3.9	0.2	10.3	4.7	0.3	8.5	4.1	0.3	10.6	5.1	0.3
Tsp	20.5	4.6	0.2	33.8	5.6	0.3	25.6	5.0	0.3	35.6	5.9	0.3
Tsr	23.5	4.6	0.2	40.1	5.6	0.3	28.2	5.0	0.3	39.5	5.9	0.3
W1	5.5	4.6	0.3	5.9	5.5	0.3	5.5	4.9	0.2	5.8	5.3	0.3
W1r	12.5		0.3	12.2		0.3	16.8		0.3	11.3		0.3
W2	4.9	4.2	0.3	5.2	5.0	0.3	5.0	4.5	0.2	5.1	5.0	0.3
W2r	5.8		0.3	6.1		0.3	6.3		0.3	5.9		0.3
Wi	3.8	2.9		4.1	3.4		3.9	3.1		4.2	3.5	
Wir	11.6		0.1	11.9		0.2	15.3		0.1	10.8		0.2

D JSONPATH IMPLEMENTATIONS – NODE AND PATH SEMANTICS

Using the `json-path-comparison` project [3] we ran a comparison of existing implementations of JSONPath w.r.t. node and path semantics, as described in 2. We used the example JSON from that section, with values shortened for brevity:

```
{
  "person": {
    "name": "A",
    "thesis": {
      "name": "B",
      "advisors": [
        {
          "person": {
            "name": "C"
          }
        },
        {
          "person": {
            "name": "D"
          }
        }
      ]
    }
  }
}
```

The query tested is `$..person..name`, which witnesses the difference between the semantics. Ignoring ordering, the expected results are:

- `["A", "B", "C", "D"]`, for node semantics; or
- `["A", "B", "C", "D", "C", "D"]`, for path semantics.

The experiment is exactly reproducible – put the JSON document into a `source.json` file and execute, from the root of `json-path-comparison`:

```
cat source.json | ./src/with_docker.sh ./src/one_off.sh '$..person..name';
```

Implementation	Output	Classification
Bash JSONPath.sh	["A", "B", "C", "D"]	node
C json-glib	["A", "B", "C", "D"]	node
Clojure json-path	["A", "B", "C", "D", "C", "D"]	path
C++ jsoncons	["A", "B", "C", "D", "C", "D"]	path
Dart json_path	["A", "B", "C", "D", "C", "D"]	path
Elixir ExJsonPath	["A", "B", "C", "D", "C", "D"]	path
Elixir jaxon	["A"]	error
Elixir warpath	["A", "B", "C", "D", "C", "D"]	path
Erlang ejsonpath	["A", "B", "C", "D", "C", "D"]	path
Go PaesslerAG/jsonpath	["A", "B", "C", "D", "C", "D"]	path
Go jsonslice	[["A, B, C, D"], ["C"], ["D"]]	path [a]
Go ojg	["C", "D", "C", "D", "A", "B"]	path
Go oliveagle/jsonpath	not supported	error
Go ajson	["A", "C", "D", "B", "C", "D"]	path
Go yaml-jsonpath	["A", "B", "C", "D", "C", "D"]	path
Haskell jsonpath	["A", "B", "C", "D", "C", "D"]	path
JavaScript Goessner	["A", "B", "C", "D", "C", "D"]	path
JavaScript brunerd	["A", "B", "C", "D", "C", "D"]	path
JavaScript jsonpath	["A", "B", "C", "D", "C", "D"]	path
JavaScript jsonpath-plus	["A", "B", "C", "D", "C", "D"]	path
Java jsurfer	["A", "B", "C", "D"]	node
Java jsonpath	["A", "B", "C", "D", "C", "D"]	path
Kotlin jsonpathkt	["A", "B", "C", "D", "C", "D"]	path
Objective-C SMJJSONPath	["A", "B", "C", "D", "C", "D"]	path
PHP Goessner	["A", "B", "C", "D", "C", "D"]	path
PHP galbar/jsonpath	["A", "B", "C", "D"]	node
PHP remorhaz/jsonpath	["A", "B", "C", "D"]	node
PHP softcreatr/jsonpath	["A", "B", "C", "D", "C", "D"]	path
Perl JSON-Path	["A", "B", "C", "D", "C", "D"]	path
Python jsonpath	["A", "B", "C", "D", "C", "D"]	path
Python jsonpath-ng	["A", "B", "C", "D", "C", "D"]	path
Python jsonpath-rw	["A", "B", "C", "D", "C", "D"]	path
Python jsonpath2	["A", "B", "C", "D", "C", "D"]	path
Raku JSON-Path	["C", "D"]	error
Ruby jsonpath	["A", "B", "C", "D", "C", "D"]	path
Rust jsonpath	not supported	error
Rust jsonpath_lib	["A", "B", "C", "D", "C", "D"]	path
Rust jsonpath_plus	["A", "B", "C", "D", "C", "D"]	path
Scala jsonpath	["A", "B", "C", "D"]	node
Swift Sextant	["A", "B", "C", "D", "C", "D"]	path
.NET Json.NET	["A", "B", "C", "D", "C", "D"]	path
.NET JsonCons.JsonPath	["A", "B", "C", "D", "C", "D"]	path
.NET JsonPath.Net	["A", "B", "C", "D", "C", "D"]	path
.NET JsonPathLib	["A", "B", "C", "D", "C", "D"]	path
.NET Manatee.Json	["A", "B", "C", "D", "C", "D"]	path

Table 9: Semantics chosen by known JSONPath implementations. Node semantics is highlighted in dark grey. Light grey indicates errors.

[a]Different output presentation, but clearly path semantics.