Supporting Descendants in SIMD-Accelerated JSONPath

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 put descendant queries in the focus: we consider the fragment of JSONPath that supports child, descendant, 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 by considering semantically equivalent queries formulated with and without descendant. The experiments confirm that our approach allows supporting descendant not only without sacrificing performance, but actually with impressive gains in many cases.

Do not forget to mention JSON Pointer

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 [18, 21, 22]. 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 [20]. 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 time of the query – 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 [23]. 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 [28] and jq [10], 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 [26]. Jiang and Zhao's recent JSONSki [19] 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 [15]. 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 means that there is no easy way to add support for either descendant or idiomatic wildcard.

1.2. Our Contribution

We present an engine supporting JSONPath queries with labels, child, and descendant, that does not rely on advance knowledge of types of values, and thus presents no fundamental barriers for implementing wildcards. These selectors allow reaching deep down the document without specifying the full path, and accessing elements at multiple depths with a single query. Tasks like fetching all values associated with a given field in the document become very easy, as they can be succinctly represented with a descendant query. As an 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 both know the depth of the property, and without full wildcard support also specify all labels on the path, leading to an explosion of possibilities.

We propose a modular approach to evaluating such queries based on novel depth-stack automata that process a stream of events produced by a state-driven classifier.

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 stream processing from the

heavily branching code implementing the logic.

Sparse stack representation. In the described model, child-descendent queries can be easily executed by a push-down automaton, but using the stack is potentially costly. Depth-register automata [5] 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 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 switching the classifier depending on the current needs of the automaton. This way we avoid classifying irrelevant symbols and generate fewer events. The idea is captured in our *multi-classifier pipeline* which allows switching dynamically between classifiers, based on the feedback from the automaton. 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.

Implementation and Experiments. We implemented our solution in Rust. The resulting tool, dubbed SIMD-Path (name changed due to anonymity requirements), is available online [4]. In a series of experiments we compare SIMDPath with the state of the art by considering semantically equivalent queries formulated with and without descendant. The experiments confirm that our approach allows supporting descendant not only without sacrificing performance, but actually with impressive gains in many cases.

1.3. Outline

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 $\left[J_1, J_2, \ldots, J_n\right]$ where $J_1, J_2, \ldots, J_n$ are JSON documents;
- an *object* of the form $\left\{\ell_1 : J_1, \ell_2 : J_2, \ldots, \ell_n : J_n\right\}$ 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* of J. A string is a sequence of zero or more 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 challanges for rapid processing of JSON documents.

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.

TODO: Can we use $.*$ everywhere instead of $[*]$?

We apply the *node semantics*, defined as follows. When evaluated over a JSON document J, an expression e returns a (possibly empty) sequence $e(J)$ of JSON documents. 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 $s = ..\ell$,

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

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

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

where $J_i^1, J_i^2, \ldots, J_i^{n_i}$ are the direct subdocuments of J_i.

There exists an alternative *path semantics*, in which a JSONPath query selects a set of marked paths in the tree. Intuitively, a marked path is a pair consisting of a path to a node that satisfies the query under node semantics, with an additional function that maps each selector in the query to a node on the path. The difference from node semantics comes from the marking: each selected node yields exactly one path, but it may yield multiple marked paths because the selectors in the query may be mapped to the path in multiple ways.

We posit that path semantics is undesirable. The reasons are two-fold. First, cluttering results with duplicated values is usually not what the user wants. Second, under path semantics the result set might grow exponentially large in the length of the query. The original implementation by Gössner [15] uses path semantics. It is unclear whether it was a conscious choice or simply a byproduct of the way the author implemented the engine at the time. Using the JSONPath comparison project [7] we found that most existing implementations of JSONPath use path semantics: out of 44 tested implementations, 34 of them use path semantics, while only 6 use node semantics (4 were errors). See 7 for details. PostgreSQL's implementation of JSONPath also uses path semantics, which makes it possible to construct simple antagonistic queries against the database. The current JSONPath specification draft [16] does not address this issue directly, but the semantics there is defined using node result sets. An implementation may still present output in a different way, so path semantics is not strictly disallowed. From this point onward we

Figure 1: An overview of `simdpath`

consider only node semantics, as not only the more useful and conforming to the specification draft, 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 working horse of `simdpath`; we discuss it in detail in Section 4.

3.1. Constructing query automata

A JSONPath query can be naturally represented as a nondeterministic finite automaton (NFA) 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). To ease the simulation, we turn it into a minimal deterministic finite automaton (DFA). Theoretically, the size of a DFA could be exponential in the size of the query. This indeed happens for some queries with wildcard, but for wildcard-free queries one can easily construct a (minimal) DFA of the same size as the original NFA.,

Descendant-only queries. We begin with the simpler case of queries with descendant selectors only. Let us consider a query $\$..\ell_1..\ell_2 \cdots ..\ell_n$. In plain English, this query asks for an ℓ_n node that is located in a subtree of an ℓ_{n-1} node that is located in a subtree ..., ultimately located in a subtree of an ℓ_1 node. An NFA for such a query has a very regular form, illustrated in Figure 2 (top). It consists of a chain of states corresponding to the selectors of the query. The automaton can always loop at the current selector $..\ell_i$, but when it sees the label ℓ_i, it may transition to the next selector $..\ell_{i+1}$. From the last selector $..\ell_n$ the automaton moves to the accepting state. In order to turn the NFA into a DFA we simply force the automaton to move on to $..\ell_{i+1}$ as soon as it encounters the label $..\ell_i$; see Figure 2 (bottom) for an illustration. Note that we crucially rely on the node semantics. The following *greedy match property* is key: once we find ℓ_i on a path in the tree, we can assume that the correspond-

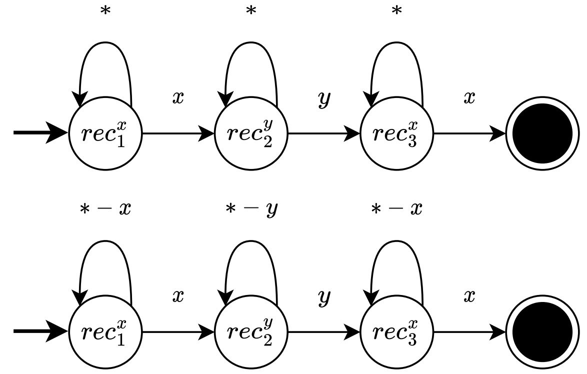


Figure 2: NFA (top) and minimal DFA (bottom) recognising paths matching query `$..x..y..x`.

ing selector $..\ell_i$ matches there, and start looking for ℓ_{i+1}. Under the path semantics, we would have to consider each subsequent ℓ_i too, as they would yield different markings for the resulting path. When looking at a query as an automaton, node semantics asks "Is this path accepted?", while path semantics asks "How many different accepting runs does this path induce?". Only the former allows us to effectively determinise and minimise the NFA.

Allowing child selectors. Child selectors cause the query NFA to effectively have two types of states: *recursive*, which correspond to descendant selectors in the query, and *direct*, which correspond to child selectors; see Figure 3 (top). The greedy match property applies here in a generalised fashion: once we reach a given recursive state, we can forget about all states before it. This divides the automaton into *segments*, where only one segment needs to be simulated at a time. Moreover, determinisation of such an automaton causes no explosion of states – only transitions become more complicated. Every segment is translated to a same-size strongly connected component in the minimal DFA: essentially, the control table from the KMP pattern-matching algorithm for the pattern corresponding to the block of consecutive child selectors. This gives rise to two kinds of transitions: those that consume a label allowing to match a non-empty prefix of the block, and *fallback* transitions leading to the initial state of the component (corresponding to a recursive state of the NFA); see Figure 3 (bottom).

Allowing wildcard selectors. For queries with wildcards the NFA is still very simple: the only difference is that transitions from direct states corresponding to wildcard selectors are over an arbitrary label. See Figure 4 (top). The DFA, however, may be much more complicated; see Figure 4 (bottom). It is still a chain of strongly connected components corresponding to the segments of the NFA, and each state has a few transitions over concrete labels plus a single fallback transition over the remain-

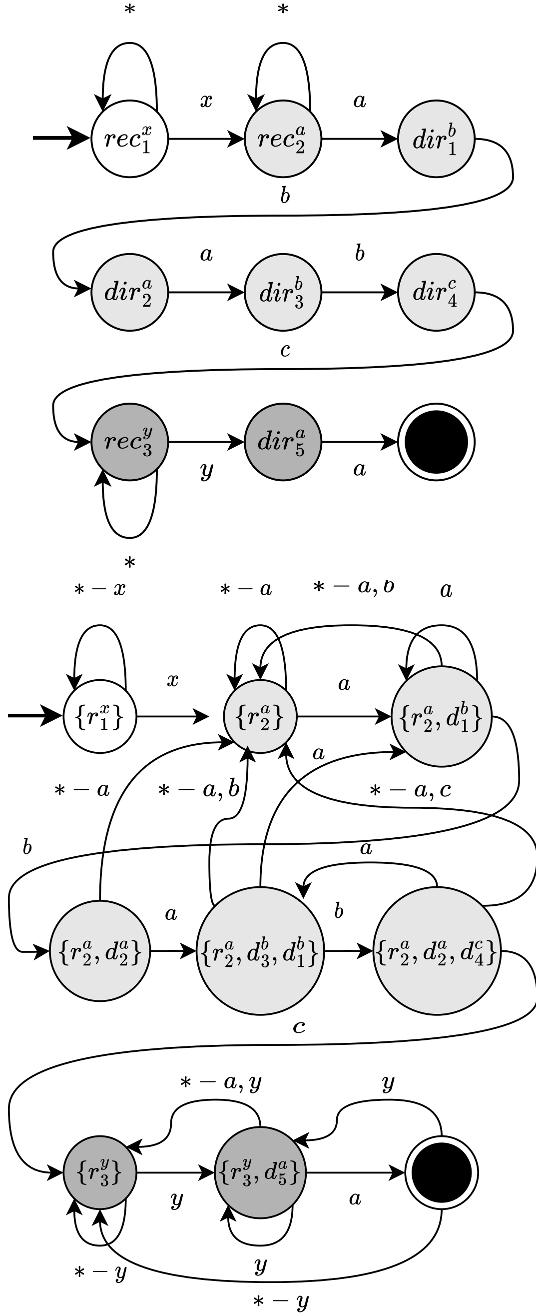


Figure 3: NFA (top) and minimal DFA (bottom) recognising paths matching query `$..x..a.b.a.b.c..y.a`. Different shades of grey represent segments in the NFA and corresponding strongly connected components in the DFA.

ing labels that loops or leads to the initial state of the component (except for the first component where it loops or leads to the trash state). The difference is that each strongly connected component may now be exponentially larger. Indeed, a classical example of exponential blowup can be now reconstructed: the query $..$a.*.*.\cdots.*$ selecting nodes whose ancestor n levels up has label a.

Figure 4: NFA (top) and minimal DFA (bottom) recognising paths matching query `....`

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 (Table 1), 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 structural character (‘`[`’ or ‘`{`’) is encountered, the state of the simulated automaton is pushed to the stack;
- each encountered label triggers a transition of the simulated automaton and, if the new state is accepting, an answer is reported;
- a closing structural 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 should be more performant when paired with SIMD processing.

Querying streamed trees in a stackless manner was investigated in [5], 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 descendent-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 [5]. 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 adventage 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 $\mathcal{O}(n)$ 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 obviously linear in the size of the input data.

3.3. Skipping

[TODO: say what's new and what's from JSONSki] While the algorithm described in Section 3.2 dutifully steps through every element of the document, the key insight from [19] 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.

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 of 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. This means simply that we are interested in the next opening character, except that we should also be mindful of closing characters, because maybe all the remaining siblings are leaves in which case the next opening character would be already outside of the scope of the current subtree. 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. Importantly, we know if we are skipping over an object or an array, so we know whether it is going to be '`}`' 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. (This kind of skipping is applicable only in objects.) As in the previous case, we would like to jump straight to the closing character marking the end of the whole subtree, and we know exactly which of the two closing characters it is going to be.

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 descendent selectors $..\ell$. When the simulated DFA is in such a state, it would make sense to jump directly to the next descendent of the current node that has label ℓ. This is hard in general, as it requires monitoring the depth and the label, but it becomes straightforward 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 safely 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()`. 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 returns 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.

```
1  state = automaton.init_state()
2  stack = init_stack()
3  while event = iterator.next():
4    match event:
5      case Opening(c):
6        label = iterator.get_label()
7        target = state.transition(label)
8        if target.is_rejecting():
9          iterator.skip_children(c)
10         continue
11       if target != state:
12         stack.push(state, depth, c)
13         state = target
14       depth = depth + 1
15       if state.is_accepting():
16         report_match()
17       iterator.toggle(state, c)
18       if c == '[':
19         try_match_first_item()
20     case Closing(c):
21       depth = depth - 1
22       prev = stack.top()
23       if depth == prev.depth:
24         state = prev.state
25         stack.pop()
26         if state.is_unitary():
27           iterator.skip_siblings()
28           continue
29         iterator.toggle(state, prev.c)
30       else:
31         iterator.toggle(state, '{')
32     case Colon:
33       if iterator.peek() == Opening(_):
34         continue
35       label = iterator.get_label()
36       target = state.transition(label)
37       if target.is_accepting():
38         report_match()
39       if state.is_unitary():
40         iterator.skip_siblings()
41     case Comma:
42       if iterator.peek() != Opening(_):
43         report_match()
```

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.

Skipping children and siblings is handled by calling the respective methods, `skip_children()` and `skip_sibling()` of the iterator. Skipping to label is implemented outside of the above algorithm: if the query starts from a descendent selector $..\ell$, the engine finds the first occurrence of ℓ in the stream using the highly optimized `memmem` function from the `memchr` crate [13], 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, etc.

3.5. Automaton-classifier interaction [to cannibalize]

skip_children. When that happens, we use our classifier API to stop structural classification and trigger the depth classifier to fast-forward through the object or array. We then stop the depth classifier and resume the structural one, going one step backwards in the automaton to a non-rejecting state.

Skipping to label. This could be seen as an additional ultralight classifier, utilising SIMD instructions to match bytes of the label. This approach is correct, as for such queries we do not care about the depth of the first label, and we can consider each such subtree separately. Note that this is not true for child selectors, as we want the depth to be exactly 1, or for nested descendants, which have to be contained within the subtree selected by the preceding selectors. An integration between SIMD depth-checking and label-matching would prove invaluable for those cases.

4. Vectorised classification

The core of our engine is a SIMD pipeline (encapsulated in the iterator) that quickly locates relevant characters and fast forwards through irrelevant fragments of the stream.

The main ingredient of the pipeline is the *structural classifier* that recognizes JSON structural characters (see Table 1) and fast forwards through whitespaces, labels, and atomic values. Its task is not straightforward, because brackets, braces, colons, and commas located within strings do not play any structural role and should be ignored. Moreover, it is not enough to ignore characters located between matching double quotes, because not all double quotes delimit strings – some of them may be escaped with backslashes.

We we also use a lightweight *depth classifier*, that allows us to fast forward through irrelevant subtrees, known not to contain query matches.

In what follows we first present a general method for raw classification based on the `shuffle` instruction that can be of use for fast parsing of other document formats. We then describe an algorithm dealing with escaped and quoted sequences, largely analogous to Langdale and Lemire's solution in simdjson [20]. Next, we explain how block boundaries and multiple blocks are handled. Finally, we briefly discuss the depth classifier and explain how the two classifiers are orchestrated.

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* $f : \{\texttt{0x00}, \texttt{0x01}, \ldots, \texttt{0xff}\} \to \{0, 1, \ldots, k-1\}$. *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. When very few values are mapped to 1, a simple fast solution is to compare (vectorially) with each value separately, and aggregate the answers using bitwise OR. For **[TODO]** values this takes only **[TODO]** CPU cycles. When more values are mapped to 1, it pays off to do something smarter.

We will use the `shuffle` operation, 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]` at postion i, for all $0 \le i < 16$ (assuming that the upper nibbles of `b` are zeroed).[1] 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 a working solution for the general case. **[TODO: How does this compare to SIMDjson?]**

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 $\langle \texttt{3}, \texttt{a} \rangle$. We let $\mathit{Nib} := \{\texttt{0}, \texttt{1}, \ldots, \texttt{e}, \texttt{f}\}$.

Definition 1 (Acceptance Set)**.** *For a given nibble $u \in \mathit{Nib}$ its* acceptance set *is the set of all the nibbles l that cause $\langle u, l \rangle$ to be accepted, i.e.*

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

Define $\mathit{low}(u) : \mathit{Nib} \to \mathscr{P}(\mathit{Nib})$ as a function assigning to each nibble u its acceptance set.

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

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

Note that $|G| \le |\mathit{Nib}| = 16$.

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

[1] 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.

Table 1: JSON Structural Characters

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

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

$$low(\texttt{a}) = low(\texttt{b}) = \{1,2\}, \quad low(\texttt{c}) = \{2\},$$
$$G = \{\langle\{\texttt{a},\texttt{b}\},\{1,2\}\rangle, \langle\{\texttt{c}\},\{2\}\rangle\},$$

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

Depending on the properties of the group set G we can distinguish three cases of increasing complexity for the classification problem.

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

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

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

$$utab = [\texttt{0xfe},\texttt{0xfe},\texttt{0x02},\texttt{0x03},\texttt{0xfe},\texttt{0x01},\texttt{0xfe},\texttt{0x01},$$
$$\texttt{0xfe},\texttt{0xfe},\texttt{0xfe},\texttt{0xfe},\texttt{0xfe},\texttt{0xfe},\texttt{0xfe},\texttt{0xfe}],$$
$$ltab = [\texttt{0xff},\texttt{0xff},\texttt{0xff},\texttt{0xff},\texttt{0xff},\texttt{0xff},\texttt{0xff},\texttt{0xff},$$
$$\texttt{0xff},\texttt{0xff},\texttt{0x03},\texttt{0x01},\texttt{0x02},\texttt{0x01},\texttt{0xff},\texttt{0xff}].$$

NOTE: Is it worth to say how we toggle the commas and colons precisely, showing the masks used?

Few groups. Another case that can be efficiently solved is when $|G| \leq 8$. 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. 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);
```

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 six CPU cycles, since the two lookups are independent.

4.2. Recognising quoted sequences

The next step is ignoring characters recognised as structural by the lookup, but located inside JSON strings. To that end, we mark all double quote characters with a simple `cmpeq`. However, we need to also account for escape sequences. A double quote is escaped if and only if it is preceded by a sequence of backslashes of odd length:

- JSON string `"x\""` contains a single escaped double quote;

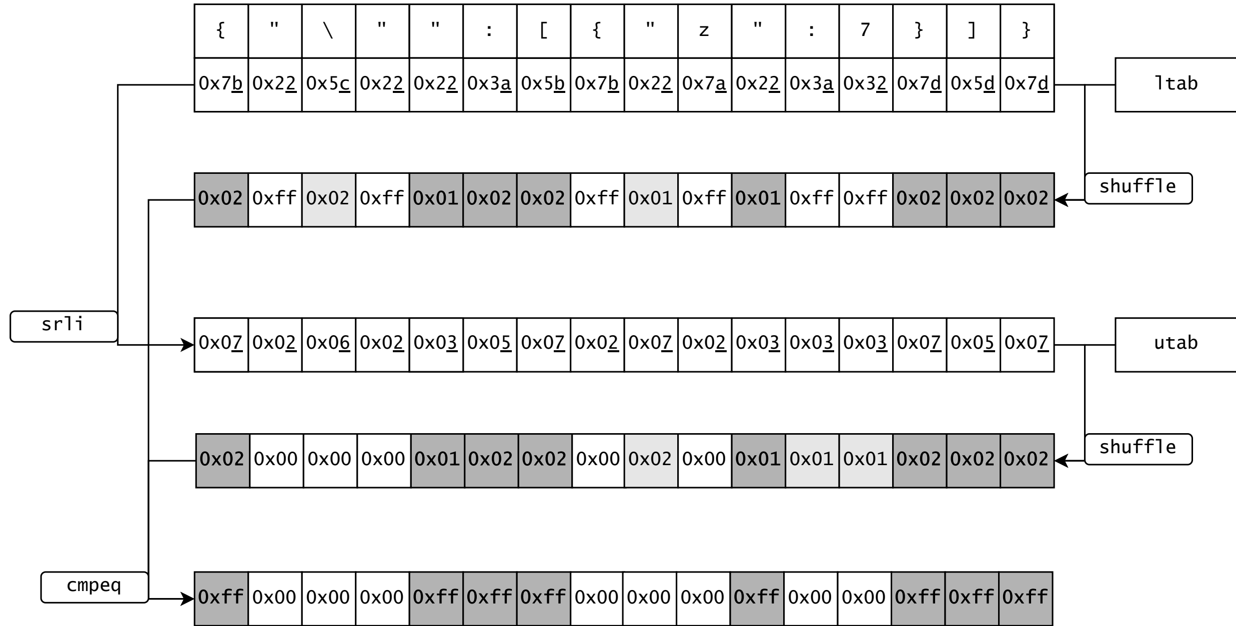


Figure 5: JSON document classified using the structural classifier's `ltab` and `utab` lookup tables.

- JSON string `"x\\"` contains a single escaped backslash; none of the double quotes are escaped.

We use the same solution as simdjson [20]. First, we mark all backslash and quote characters with a `cmpeq`. Next, we move out of the SIMD world and obtain two masks in regular registers, *quotes* and *slashes*. The key idea now is that we can mark starts of backslash sequences and use *add-carry propagation* to find their ends. To find starts, we ask for backslashes not preceded by other backslashes, which is *slashes* AND NOT $(slashes << 1)$. We partition the starts into those occurring at odd positions in the vector and those occurring at even positions in the vector, using a constant mask[2].

Then we arithmetically add *slashes* to each of these masks. The starting bit triggers a carry, which continues through the sequence of consecutive backslashes. The result is a single lit bit one place past the end of the sequence. We can now partition ends based on their positions as well. Escaped characters will be the ends of starts on even positions that occur at odd positions, and ends of starts on odd positions that occur at even positions. It remains to exclude those escaped characters from *quotes*.

Having recognised unescaped double quote characters we now want to exclude all structural characters that are quoted. Observe that if we take the vector where bits are lit at unescaped double quote characters, then a prefix-xor computed on it will mark with lit bits exactly those characters that are quoted. Prefix-xor can be computed efficiently as the result of carry-less multiplication by a vector with all bits lit [20]. Thus, it suffices to load the vector into a SIMD vector, perform the `clmul` operation, and then extract the information back to a mask.

[2] A 64-bit mask for even positions is `0x5555555555555555`, while for odd it is `0xAAAAAAAAAAAAAAAA`.

4.3. Block boundaries

There is an issue with the above algorithm stemming from the nature of block-by-block processing. If the boundary between two blocks falls between a pair of unescaped double quote characters, so that the opening double quote is in the first block while the closing is in the second, then we will misclassify all bytes in the second block. We might also get incorrect results if the block boundary falls in the middle of a sequence of consecutive backslashes.

There are two ways of dealing with this: introducing a state, or using overlapping windows. Overlapping windows would significantly degrade the classifier's performance, so we choose to carry a state. To handle escaping correctly, we need a single bit of information: whether the previous block's last character was an unescaped backslash. It is then used in two places: if the first character in a block is a backslash, but the bit is lit, then it is not a start of a sequence; and if the first character is not a backslash, but the bit is lit, then it is an escaped character. We avoid branching by converting the bit into a mask of length in bits equal to the block length in bytes and combining it with the rest of the information using bitwise operations:

```
let starts = slashes & (!slashes << 1)
                     & !prev_slash_mask;
...
let escaped = (ends_of_even_starts & ODD)
    | (ends_of_odd_starts & EVEN)
    | prev_slash_mask;
```

Another bit of state indicates whether the previous block ended while still within quotes, which is equivalent to the last bit extracted from the result of `clmul` being lit. We can efficiently store both bits of information in a single byte of storage.

4.4. 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.

We create a structure implementing the Rust's `Iterator` trait[3], yielding items of an algebraic sum type `Structural` with four variants:

- `Closing` – representing '`{`' or '`[`';
- `Colon` – representing '`:`';
- `Comma` – representing '`,`';
- `Opening` – representing '`}`' or '`]`'.

Additionally, each item stores the index at which the character occurred in the input.

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. This allows us to create the `Structural` item that is returned. Additionally, we modify the stored bitmask by zeroing the lit bit we just processed.

A crucial observation is that the `Colon` and `Comma` characters are usually not needed for proper query execution. They matter only at the end of the query, when atomic values can be added to the result set from a label or by extracting them from a list. Therefore, our structural classifier does not classify them by default. The main engine can ask the classifier to turn one of those characters on – this is done by XOR-ing the internal lookup table used for `shuffle` with a special precomputed mask targeting the nibbles of the colon (0x3a) or comma (0x2c) character.

We will later see that most of the time the query runner does not care about colons and commas. The main engine can ask the classifier to[...]

This provides the main implementation with a stream of the relevant structural elements, while all other characters are efficiently skipped over. The SIMD pipeline calculating structural bitmasks is completely branchless, while 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 `Structural` value.

[3] `Iterator` models a stream of elements with a single function `next` returning the next element in the stream, or stating that there are no more elements.

4.5. Depth classifier

The key insight from [19] is that objects (or lists) which are known to not contain query matches can be *skipped*. The JSONSki approach is to count braces (or brackets) to find the closing tag. Generalising, we can maintain the depth inside the subtree being skipped and ignore everything other than opening and closing tags while depth is above zero. We abstract it away as a separate *depth classfier* that supports two operations:

- query the relative depth at the current position;
- fast-forward to the next occurrence of a closing tag.

To implement the depth classifier, we mark all opening and closing tags using standard SIMD instructions and produce two bitmasks for them. Advancing to the next closing tag can be done with simple bitwise operations, and we update the relative depth by counting bits lit in the relevant chunk of the bitmask.

As an additional heuristic, we skip the entire block if we can see at a glance that the depth there cannot go to zero: this is the case when the number of closing tags in the block is lower than the current depth. This makes the algorithm much more efficient, especially on real-life JSON documents.

@Mat: 2 vs 5 SIMD operations, so much faster. It pays of to pay the price of switching the whole classifier.

General idea (intro? sec intro?): it makes sense to switch classifiers to skip more symbols and to make the classification itself cheaper. Sometimes the algorithm is similar, and can just implement the switch by replacing the lookup tables.

4.6. Multi-classifier pipeline

The nature of fast-forwarding is such that we need to be able to trigger depth classification on-demand, pausing the full structural classifier for the duration of the fast-forward. When the object or array is skipped, we need to resume structural classification. To facilitate this we propose a robust multi-classifier pipeline available for extension with different classifiers later on.

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 in-

ternal 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).

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 main goals were three-fold: estimate the overall overhead involved in supporting descendants, showcase the benefits from working directly with descendant queries, and identify improvement opportunities. The benchmark code and all data sets are available online [4].

This should be rewritten, the narrative changed. **Propositions** We performed a series of experiments comparing our engine with existing solutions. Our main goals were three-fold: estimate the benefits from the new classifiers methodology, showcase the benefits from working directly with descendant queries as opposed to semantically equivalent descendant-free query, and identify improvement opportunities. The benchmark code and all data sets are available online [4].

5.1. Competitors

For the baseline we chose JsonSurfer [28], 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 A0).

Table 2: 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)	2029.0	9	24.4
GoogleMap (G)	1136.1	10	36.9
NSLP (N)	1210.2	10	13.8
OpenFood (O)	0.9	9	19.8
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

Our main point of reference is JSONSki [19], to the best of our knowledge the only JSONPath engine using SIMD optimization. We do not include simdjson [20] 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 [19].

5.2. Datasets

In our experiments we use dataset from JSONSki as a baseline to compare our implementation on their query set. We excluse two queries of the dataset becaue (...). We add two more diverse datasets, representing characteristic usecase. purge twitter.json and include the one of JSONSKi We integrate other dataset and queries to showcase the performance improvement of The first one is the `twitter.json` file extracted from simdjson's quick-start tutorial in the project's repository [27]. It is a typical file obtained by querying an API, small but irregular. Our second dataset is an arbitrarily chosen fragment of the datadump [9] of the Crossref service [8] collecting metadata of several hundred millions of scientific publications. We restrict it to a smaller chunk for the sake of the experiment. The document is highly regular: it contains collections of sub-documents of very similar shape. Finally, our last 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 (e.g. harvesting code for AI [12]), gaining importance with the proliferation of large code repositories and archives such as Software Heritage [24]. Detailed characteristics of the datasets are shown in Table 2; by verbosity we mean the ratio of the size of the document to the number of nodes in the underlying tree. All three datasets are accessible through Zenodo [11]: AST [3], Crossref [1], Twitter [2].

5.3. Queries

The queries used in the experiments, shown in Table 3, differ in selectivity and display diverse combinations of

Table 3: Queries from JsonSki

ID	Query	Matches
B1	$.products[*].categoryPath[*].id	
B2	$.products[*].videoChapters[*].chapter	
B3	$.products[*].videoChapters	
G1	$[*].routes[*].legs[*].steps[*].distance.text	
G2	$[*].available_travel_modes	
N1	$.meta.view.columns[*].name	
N2	$.data[*][*][*]	
T1	$[*].entities.urls[*].url	
T2	$[*].text	
Wa1	$.items[*].bestMarketplacePrice.price	
Wa2	$.items[*].name	
Wi	$[*].claims.P150[*].mainsnak.property	

Table 4: Rewriten query

ID	Query
B1^r	$..categoryPath..id
B2^r	$..videoChapters..chapter
B3^r	$..videoChapters
G2^r	$..available_travel_modes
Wa1^r	$..bestMarketplacePrice.price
Wa2^r	$..name
Wi^r	$..P150..mainsnak.property

child and descendant selectors. We run these queries directly on JsonSurfer and `simdpath`, but for JSONSki we need to reformulate them without descendant as it is not supported (see Table **??**). This is possible only for some queries: A0 and A1 cannot be expressed without descendant due to the highly irregular structure of the AST dataset; C2 and T6 cannot be reformulated because the matched nodes are at different depths. The reformulation for T9 and T10 is simply T8. Let us stress that the reformulations are not equivalent to the original queries in general, they only return the same matches on the specific datasets.

5.4. Setup

We use Rust Criterion [17] 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.

We ran JsonSurfer without any modifications via JNI[4] [25]. For JSONSki we introduced a few technical tweaks to integrate it with our Rust-based benchmark harness.

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

Table 5: Other queries of interest

ID	Query
A1	$..decl.name
A2	$..loc.includedFrom.file
A3	$..inner..inner..type.qualType
C1	$..DOI
C2	$.items[*].title
C2^r	$..title
C3	$.items[*].author[*].affiliation[*].name
C3^r	$..author..affiliation..name
C4	$.items[*].editor[*].affiliation[*].name
C4^r	$..editor..affiliation..name
C5	$.items[*].author[*].ORCID
C5^r	$..author..ORCID
O1	$.products[*].vitamins_tags
O1^r	$..vitamins_tags
O2	$.products[*].added_countries_tags
O2^r	$..added_countries_tags
O3	$.products[*].specific_ingredients[*].ingredient
O3^r	$..specific_ingredients..ingredient
Ts1	$..hashtags..text
Ts2	$.statuses[*].retweeted_status.entities.hashtags[*].text
Ts2^r	$..retweeted_status..hashtags..text
Ts3	$.search_metadata.count
Ts4	$..search_metadata.count
Ts5	$..count

First, we removed `std::vector`-based results gathering in favour of a simple match counter. This slightly increases the performance of JSONSki as compared to the original [19]. 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.

5.5. Hardware

Experiments were run in a stable environment, on machines hosted by Grid'5000, a federated testbed setup providing a high diversity of architectures [14]. 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 A.

5.6. Results

Figure 6 shows the throughput comparison of the three tools. JsonSurfer consistently performs an order of magnitude slower than both JSONSki and SIMDPath. (Let us point out however, that it is the only of the three tools that supports full JSONPath.) In the right-hand plot one can observe several phenomena.

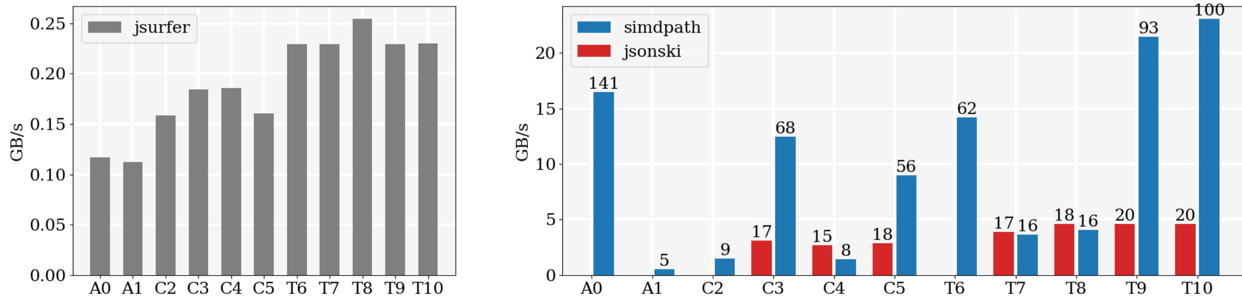


Figure 6: Throughput comparison. Numbers on the bars indicate speedup with respect to JsonSurfer.

Queries A0, A1, C2, and T6 cannot be expressed without descendant because of relevant labels appearing at different levels, thus we do not run JSONSki on them. A0 illustrates how fast one can go for highly selective queries using the `memmem` acceleration described in Section 3.5 (processing subdocuments rooted at `decl` labels is fast because they are small). On A1 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 **??**. No good implementation for such queries is known, even theoretically. The purpose of C2 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. T6 is similar to A0 but it uses two descendant selectors; SIMDPath performs well for similar reasons.

On queries C3, C5 SIMDPath performs 3x times better than JSONSki. Both C3 (high selectivity) and C5 (medium selectivity) can be expressed without descendant, but less conveniently. Formulated with descendant they run much faster on our engine.

Queries T8, T9, and T10 all return the same results, but specify the access path less and less precisely. They show that in our approach it is better to have the path underspecified.

On T7 and T8 SIMDPath and JSONSki display similar performance. T7 is similar to T6, restricted to one level so that we can compare to JSONSki. While having the whole path specifed could have helped JSONSki a lot, the descendant approach is competitive, and more convenient to the user. On T8 SIMDPath is competitive as well. If we new how to find the next label at the same depth (a sibling) we could do better, possibly closing the gap to T9 and T10. Note that all of those queries return the same results on this dataset.

Query C4 is hard for SIMDPath. JSONSki only examines `author` nodes at depth 3 while SIMDPath goes through all `author` nodes (12x more), none of which have ORCID, and each time look through the whole subdocument searching for ORCID. We could improve our performance on this query if we could fast-forward to a given label within an element (see Section 4.6): this would allow to quickly discard authors without ORCID.

Table 6: Throughput [GB/s] on the Crossref dataset

Size [GB]	1	4	8	16
Broadwell	9.4	9.4	9.3	9.4
Skylake	9.6	9.4	9.3	10.0
Cascade-Lake	9.1	9.1	9.1	9.1
Zen 2	11.3	11.5	11.1	9.5

To test the scalability of SIMDPath on various architectures (see Appendix A), we ran the query `$..affiliation..name` on fragments of the Crossref dataset of increasing size (Table 6). We observed no significant variation.

6. Looking ahead

We have demonstrated that it is possible to support descendent queries in SIMD-accelerated JSONPath processing without paying a penalty. Supporting wildcard and arrays is compatible with our approach and we plan to implement it in near future. A grand challenge is supporting filters. They are known to be highly incompatible with the streaming model, not only drastically affecting memory usage, but also making returning matches problematic [5]. Another challenge is compositionality: processing queries in succession, with the output of one query being fed directly to another one.

References

[1] Author removed for double-blind review. Extraction of Crossref datadump for benchmarking purpose, 2022. https://zenodo.org/record/7231920.

[2] Author removed for double-blind review. JSON file from Twitter API used for benchmarking JSONPath, 2022. https://zenodo.org/record/7229287.

[3] Author removed for double-blind review. JSON file of the AST of a C program, 2022. https://zenodo.org/record/7229268.

[4] Author removed for double-blind review. simdpath, 2022. https://zenodo.org/record/7231970.

[5] 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.

[6] 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.

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

[8] Crossref. Crossref. https://www.crossref.org/.

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

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

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

[12] 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.

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

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

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

[16] Stefan Gössner, Glyn Normington, and Carsten Bormann. JSONPath: Query expressions for JSON. Internet-Draft draft-ietf-jsonpath-base-05, Internet Engineering Task Force, April 2022. Work in Progress.

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

[18] 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.

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

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

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

[22] 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.

[23] 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.

[24] 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.

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

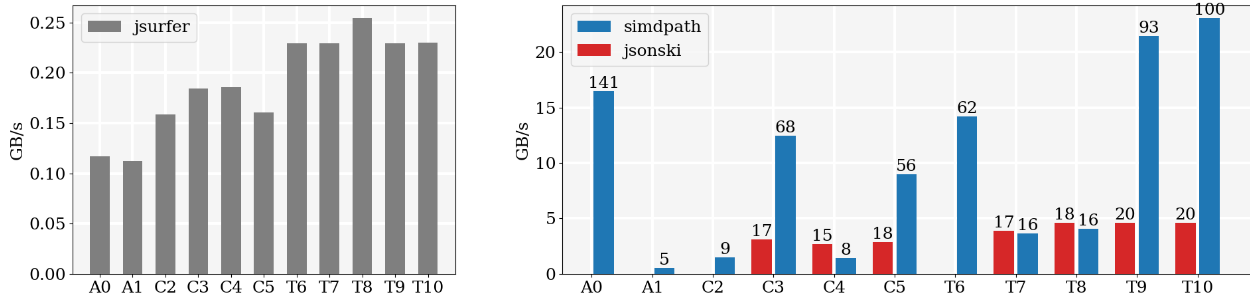
[26] simdjson. On-demand API. https://simdjson.org/api/0.6.0/md_doc_ondemand.html.

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

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

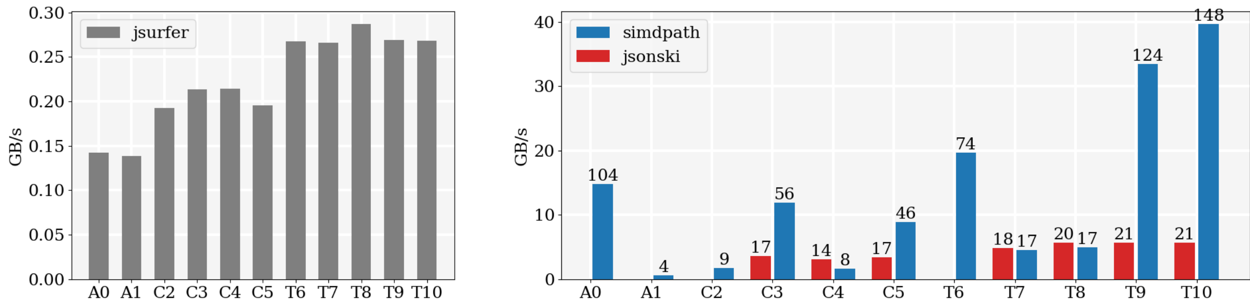
A. Additional experiments

The CPU information is available from the G5000 website [6]. The RAM information is extracted by the output of the `lshw` command .



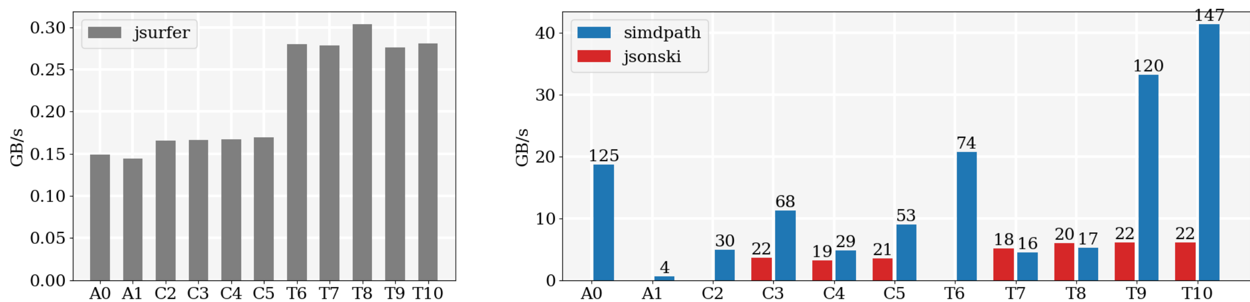
- CPU: Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz (Broadwell)
- RAM: 8x16GiB DIMM DDR4 Synchronous Registered (Buffered) 2400 MHz

Figure 7: Broadwell (Chetemi): Throughput comparison



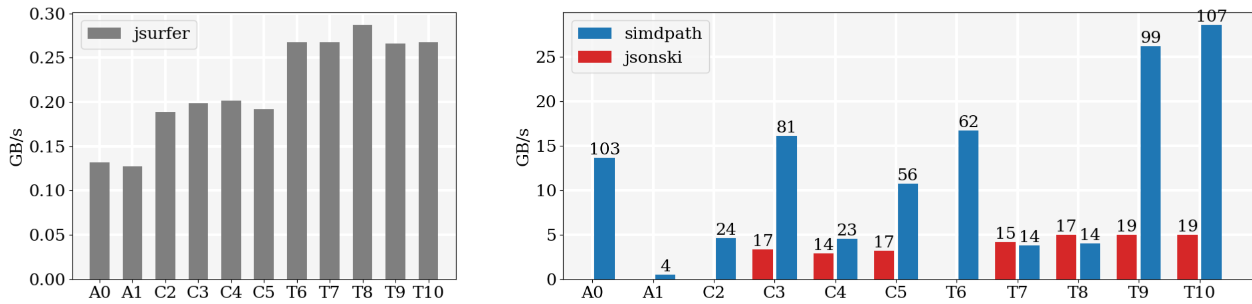
- CPU: Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz (Skylake)
- RAM: 12x16GiB DIMM DDR4 Synchronous Registered (Buffered) 2666 MHz

Figure 8: Skylake (Chifflot): 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

Figure 9: Cascade-Lake (Troll): Throughput comparison



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

Figure 10: Zen 2 (Servan): Throughput comparison

B. JSONPath implementations – node and path semantics

Using the `json-path-comparison` project [7] 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';
```

C. Result tabular format

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 7: 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.

	dataset	ID bench	dataset	file	query	result	rsonpath	jsonski
A1	ast	decl_name	ast	ast/ast.json	$..decl.name	35	16.219	None
A2	ast	included_from	ast	ast/ast.json	$..loc.includedFrom.file	482	13.779	None
A3	ast	nested_inner	ast	ast/ast.json	$..inner..inner..type.qualType	78129	0.733	None
B1	bestbuy_large_record	BB1_products_category	bestbuy_large	pison/bestbuy_large_record.json	$.products[*].categoryPath[*].id	697440	2.797	3.044
B1r	bestbuy_large_record	BB1'_products_category	bestbuy_large	pison/bestbuy_large_record.json	$..categoryPath..id	697440	5.730	3.164
B2	bestbuy_large_record	BB2_products_video	bestbuy_large	pison/bestbuy_large_record.json	$.products[*].videoChapters[*].chapter	8857	2.438	2.630
B2r	bestbuy_large_record	BB2'_products_video	bestbuy_large	pison/bestbuy_large_record.json	$..videoChapters..chapter	8857	12.520	2.745
B3	bestbuy_large_record	BB3_products_video_only	bestbuy_large	pison/bestbuy_large_record.json	$.products[*].videoChapters	769	0.919	0.721
B3r	bestbuy_large_record	BB3'_products_video_only	bestbuy_large	pison/bestbuy_large_record.json	$..videoChapters	769	12.501	0.732
	crossref0	scalability_affiliation0					8.644	None
	crossref1	scalability_affiliation1					8.605	None
C3	crossref2	author_affiliation	crossref	crossref/crossref1.json	$.items[*].author[*].affiliation[*].name	64495	2.780	2.818
C3r	crossref2	author_affiliation_descendant	crossref	crossref/crossref1.json	$..author..affiliation..name	64495	2.957	None
C1	crossref2	DOI	crossref	crossref/crossref1.json	$..DOI	1073589	3.737	None
C4	crossref2	editor	crossref	crossref/crossref1.json	$.items[*].editor[*].affiliation[*].name	39	2.664	3.055
C4r	crossref2	editor_descendant	crossref	crossref/crossref1.json	$..editor..affiliation..name	39	12.451	None
C5	crossref2	orcid	crossref	crossref/crossref1.json	$.items[*].author[*].ORCID	18401	2.643	2.680
C5r	crossref2	orcid_descendant	crossref	crossref/crossref1.json	$..author..ORCID	18401	2.627	None
	crossref2	scalability_affiliation2				93407	6.763	None
C2	crossref2	title	crossref	crossref/crossref1.json	$.items[*].title	93407	2.569	2.906
C2r	crossref2	title_descendant	crossref	crossref/crossref1.json	$..title	1716752	8.652	None
	crossref4	scalability_affiliation4				90	8.599	None
G1	google_map_large_record	GMD1_routes	google_map	pison/google_map_large_record.json	$[*].routes[*].legs[*].steps[*].distance.text	90	1.865	189582.471
G2	google_map_large_record	GMD2_travel_modes	google_map	pison/google_map_large_record.json	$[*].available_travel_modes	44	4.426	200797.250
G2r	google_map_large_record	GMD2'_travel_modes	google_map	pison/google_map_large_record.json	$..available_travel_modes	8774410	12.949	202334.950
N1	nspl_large_record	NSPL1_meta_columns	nspl	pison/nspl_large_record.json	$.meta.view.columns[*].name	24	4.852	48342.693
N2	nspl_large_record	NSPL2_data	nspl	pison/nspl_large_record.json	$.data[*][*][*]	24	2.653	2.599
O2	openfood	added_counties_tags	openfood	openfood/openfood.json	$.products[*].added_countries_tags	5	5.114	3.812
O2r	openfood	added_countries_tags_descendant	openfood	openfood/openfood.json	$..added_countries_tags	5	16.362	None
O3	openfood	specific_ingredients	openfood	openfood/openfood.json	$.products[*].specific_ingredients[*].ingredient	24	2.913	2.569
O3r	openfood	specific_ingredients_descendant	openfood	openfood/openfood.json	$..specific_ingredients..ingredient	24	14.999	None
O1	openfood	vitamins_tags	openfood	openfood/openfood.json	$.products[*].vitamins_tags	10	2.149	1.782
O1r	openfood	vitamins_tags_descendant	openfood	openfood/openfood.json	$..vitamins_tags	2	19.498	None
Ts1	twitter	all_hashtags	twitter	twitter/twitter.json	$..hashtags..text	1	12.862	None
Ts2	twitter	hashtags_of_retweets	twitter	twitter/twitter.json	$..retweeted_status..hashtags..text	1	6.141	3.803
Ts3	twitter	metadata_1	twitter	twitter/twitter.json	$.search_metadata.count	1	4.943	4.448
Ts4	twitter	metadata_2	twitter	twitter/twitter.json	$..search_metadata.count	88881	19.977	4.444
Ts5	twitter	metadata_3	twitter	twitter/twitter.json	$..count	150135	23.564	4.446
T1	twitter_large_record	TT1_entities_urls	twitter	pison/twitter_large_record.json	$[*].entities.urls[*].url	15892	3.126	67875.876
T2	twitter_large_record	TT2_text	twitter	pison/twitter_large_record.json	$[*].text	15892	3.620	68213.286
Wa1r	walmart_large_record	WM1_items_price	walmart	pison/walmart_large_record.json	$.items[*].bestMarketplacePrice.price	272499	4.109	4.362
Wa1r	walmart_large_record	WM1'_items_price	walmart	pison/walmart_large_record.json	$..bestMarketplacePrice.price	272499	12.333	4.614
Wa2	walmart_large_record	WM2_items_name	walmart	pison/walmart_large_record.json	$.items[*].name	15603	3.471	4.127
Wa2r	walmart_large_record	WM2'_items_name	walmart	pison/walmart_large_record.json	$..name	15603	3.821	4.249
Wi	wiki_large_record	WP1_claims_p150	wiki	pison/wiki_large_record.json	$[*].claims.P150[*].mainsnak.property		2.889	189251.257
Wir	wiki_large_record	WP1'_claims_p150	wiki	pison/wiki_large_record.json	$..P150..mainsnak.property		11.358	196451.681