University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Mateusz Gienieczko
Student no. 394302

Fast execution of JSONPath queries

Master's thesis
in COMPUTER SCIENCE

Supervisor:
Dr. Filip Murlak
Faculty of Mathematics, Informatics and Mechanics

Co-supervisor:
Dr. Charles Paperman
Université de Lille & INRIA, France

Warsaw, July 2022

Abstract

JSON is the format of choice for both modern web communication and datasets. Yet, fast processing of JSON documents is still a major challenge. As shown recently by Langdale and Lemire [LL19], substantial speedups can be achieved by exploiting the Single Instruction, Multiple Data (SIMD) capabilities of modern commodity processors. In this work we move on from parsing to querying JSON data. We focus an XPath-like query language called JSONPath. Our objective is to evaluate JSONPath queries in the streaming model, without constructing and maintaining the costly DOM-like tree representation. While streaming processing of JSONPath queries in general requires a stack, multiple queries are amenable to stackless evaluation strategies [BMP21]. In this work we investigate possible speed-ups coming from SIMD processing and branchless evaluation strategies. As our main result we present `rsonpath` – an engine for a subset of the JSONPath query language capable of processing over a gigabyte of JSON data per second, while using merely kilobytes of memory. We also present a theoretical framework for representing JSONPath queries as small deterministic finite automata, and posit a circuit complexity conjecture that would provide insight into hardness of querying semi-structured data.

Keywords

JSON, Semi-structured Data, SIMD, JSONPath, Query Engine, Automata, Circuits

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

Computing methodologies~Vector / streaming algorithms
Information systems~Query languages

Tytuł pracy w języku polskim

Szybkie wykonywanie zapytań JSONPath

Contents

Introduction . . . 5

1. JSON and JSONPath . . . 7
- 1.1. JSON documents as trees . . . 7
- 1.2. JSONPath queries as node or path selectors . . . 9
  - 1.2.1. Node semantics . . . 9
  - 1.2.2. Path semantics . . . 9
  - 1.2.3. Semantics choice . . . 11
- 1.3. Syntax assumptions . . . 12
  - 1.3.1. Unicode escapes in labels . . . 13

2. SIMD model . . . 15
- 2.1. Preliminaries . . . 15
- 2.2. SIMD extensions . . . 16
  - 2.2.1. Basic SIMD operations . . . 17
- 2.3. Intrinsics . . . 18
- 2.4. x86 . . . 18
  - 2.4.1. Instruction sets . . . 19
  - 2.4.2. Lanes . . . 20
  - 2.4.3. CLMUL – carry-less multiplication . . . 20
  - 2.4.4. Shuffle – nibble lookup tables . . . 21
- 2.5. Padding and alignment . . . 21
  - 2.5.1. Padding . . . 22
  - 2.5.2. Alignment . . . 22
- 2.6. Example vectorial algorithm – discrepancy search . . . 23
  - 2.6.1. Performance . . . 24

3. Branchless streaming algorithms . . . 25
- 3.1. Find byte – `memchr` . . . 25
  - 3.1.1. Performance . . . 26
- 3.2. Find sequence – `memmem` . . . 27
  - 3.2.1. Code synthesis with Rust procedural macros . . . 28
  - 3.2.2. Performance . . . 28
- 3.3. Vectorised depth calculation . . . 31
  - 3.3.1. Eager implementation . . . 31
  - 3.3.2. Lazy implementation . . . 31
  - 3.3.3. Performance . . . 32

4. Vectorised classifier . . . 35
4.1. Structural lookup table . . . 35
4.1.1. Non-overlapping groups . . . 36
4.1.2. Few groups . . . 38
4.1.3. General case . . . 38
4.2. Handling escapes . . . 38
4.2.1. Block boundaries . . . 39
4.3. Recognising quoted sequences . . . 40
4.3.1. Block boundaries . . . 41
4.4. Structural iterator . . . 41
4.4.1. Performance . . . 41

5. Main query engine . . . 43
5.1. Stackless processing . . . 43
5.2. Query automata . . . 44
5.2.1. Descendant-only automaton . . . 44
5.2.2. Allowing child selectors . . . 45
5.3. Small stack model . . . 46
5.4. Full algorithm . . . 47
5.4.1. Automaton construction . . . 47
5.4.2. Execution . . . 47
5.5. Performance . . . 48
5.5.1. Analysis . . . 49
5.5.2. Existing implementations . . . 50

6. Benchmark methodology . . . 53
6.1. Datasets . . . 53
6.2. Machines . . . 54
6.3. Tooling . . . 54
6.4. Reproducibility . . . 54

Summary . . . 55

A. `aligners` crate . . . 57
A.1. Supported alignments . . . 57
A.2. Asserting alignment on a type level . . . 58
A.3. Comparison to existing solutions . . . 58

B. Circuit lower bound for Dyck . . . 61
B.1. Classes with a bounded number of special gates . . . 61
B.2. Dyck, Prefix-Dyck . . . 62
B.3. Neutral letters . . . 62
B.4. Open conjectures . . . 63

C. JSONPath implementations – node and path semantics . . . 65

Bibliography . . . 69

Introduction

Prevalence of JSON as the communication format of the web forces all modern programming frameworks to provide some facility for JSON parsing and processing. The problem of effectively parsing JSON data and producing its tree representation (the DOM, Document Object Model) was tackled by Langdale and Lemire in [LL19] with great success. Their usage of SIMD instructions found in modern CPUs allowed them to achieve impressive speed-ups over conventional parsers.

Parsing is one problem, but in today's world of big data querying massive datasets is also an important issue. Those datasets are often JSON documents, since that is what is commonly being produced and transferred between servers. When faced with terabytes of data to query, constructing a DOM would not only take up massive amounts of RAM, 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 [Pal+18]. In this paper we circumvent parsing by executing the query in a streaming setting, without a complex data structure serving as an intermediary step. Indeed, `rsonpath` usually requires less than a kilobyte of state to perform the entire query.

As preliminaries, we introduce JSON, JSONPath semantics, and a formal specification of what results are expected from a query in Chapter 1. We then introduce the concept of Single Instruction, Multiple Data processing in Chapter 2, available CPU instructions, and an example of a problem that lends itself well to SIMD parallelisation.

As secondary results, in Chapter 3 we present a number of stream search algorithm optimised with SIMD, along with relevant experiments. While tangential to the main result, these serve as an introduction on applications of SIMD instructions, while also being of potential use on their own. In particular, we point that our algorithms could be used to speed up a popular Rust string searching library, `memchr` (Section 3.2).

In Chapter 4 we show the SIMD core of the engine, the structural classifier. Based largely on work of Langdale and Lemire [LL19], we repeat their insights with respect to escape sequences and quoted blocks, while adding a few more details missing from their paper. A novel contribution is provided in Section 4.1 – a general algorithm for construction of SIMD classifier tables.

Our main contribution, the `rsonpath` application, is presented in Chapter 5. We show how to model tree path queries as nondeterministic finite automata, and how to effectively determinise and minimise them, using insights from [BMP21]. We then propose a model of execution based on such a minimal DFA that uses a compressed stack and minimal branching. The speed-ups achieved with this technique implemented in the Rust programming language are presented. Finally, in Chapter 6 we explain the machines and tools used to perform our experiments.

In the appendix we show a link between effective SIMD parallelisation and circuit complexity, providing a conjecture on circuit lowerbounds that would explain why parsing tree-shaped data is hard to parallelise (Appendix B). We also present a secondary contribution to the

Rust ecosystem dealing with proper memory alignment (Appendix A).

Chapter 1

JSON and JSONPath

JSONPath is a query language for JSON documents based on the well-known XPath query language for XML documents. XPath syntax and semantics is defined by the World Wide Web Consortium [CD99; RDS17], and is well studied to the point of having the formal semantics implemented in Coq for verification [GV04]. JSONPath, however, does not enjoy this level of formality.

JSONPath was first defined by Stefan Gössner in 2007 [Gös07], with a brief differential comparison with XPath and, crucially, an implementation in JavaScript and PHP. These implementations are the *de facto* standard semantics of JSONPath, as it often happens with programming languages where the answer to "what is the semantics?" is "whatever the main compiler implementation does". The semantics were adapted, extended, modified, and broken by many implementations since. The result of the lack of standardisation is seen in differences between JSONPath implementations. This fragmentation has been visualised in one project [Bur+19].

There is ongoing work to standardise in form of a draft specification [GNB22]. Our implementation conforms mostly to that draft in terms of syntax and semantics, with the notable exception of handling UTF-8 escapes (see Section 1.3.1). However, the draft leaves an important aspect of the semantics undefined. In this section we formally define the semantics of a JSONPath fragment using trees as a document model and tackle the difference between *path semantics* and *node semantics*.

1.1. JSON documents as trees

A JSON document's format is standardised by RFC8259 [Bra17]. This is an authoritative document for this entire paper – we define a valid JSON document as one conforming to the RFC8259 specification, and we only consider valid JSON documents. A JSON document is a recursive structure, defined as either an *object*, an *array*, or a *value*.

- An **object** is wrapped in curly braces and contains any number of members, which are key-document pairs. A *label* is the key of any such pair. A single object may have two members with the same label – RFC8259 specifies such a case as implementation defined, and our implementation allows such a case.
- An **array** is wrapped in square brackets and contains any number of documents separated by commas.
- A **value** is a primitive literal, either a string, a number, or one of the special values `true`, `false`, `null`.

As with XML documents, it is natural to consider JSON documents as trees. Every node is either an object, an array, or a value. An object has a number of labelled children nodes. An array has a number of children nodes that are naturally labelled by their index in the array, starting from `"[0]"`. A value is always a leaf.

```
{
  "person": {
    "name": "Gienieczko",
    "thesis": {
      "name": "SIMDPath",
      "advisors": [
        {
          "person": {
            "name": "Murlak"
          }
        },
        {
          "person": {
            "name": "Paperman"
          }
        }
      ]
    }
  }
}
```

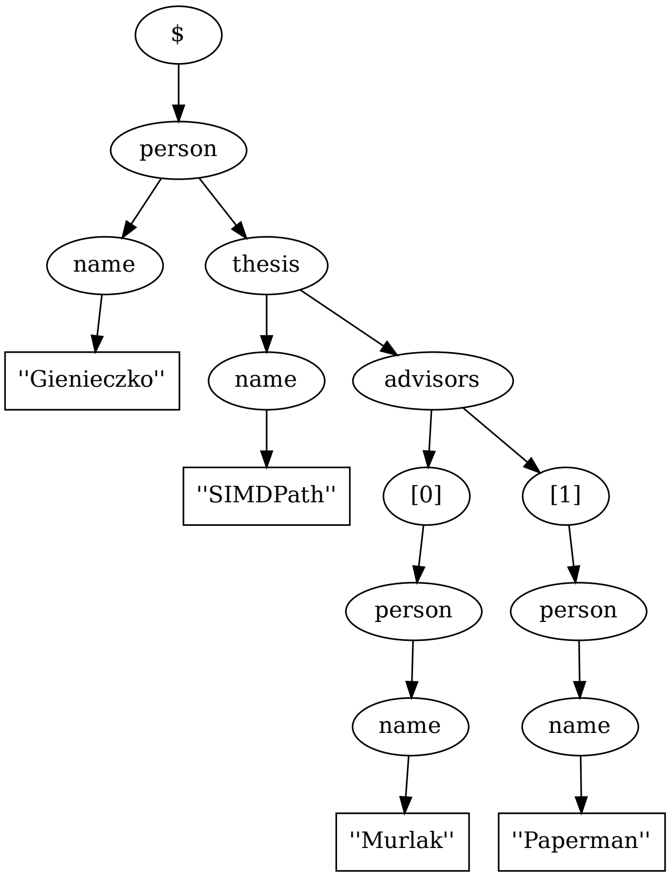


Figure 1.1: A JSON document and its tree representation.

Every node in the tree has a label – for objects it is the label of the member they are located in or the index at which they are located in an array, for values it is the value. The root gets a unique label that by convention we denote with `"$"`.

A JSON document is therefore described by a tree $T = \{V, p, \mathit{label}\}$, where V is a set of nodes, p is the parent function, and *label* is a function from nodes to their labels. The parent function for the root node v is defined as $p(v) = \perp$. From now on we only consider non-empty trees, i.e. $V \neq \emptyset$ – the result of any query on an empty tree is an empty set. Therefore we can define a function *root*, returning a root of the tree. We consider paths going in the reverse of the parent function and identify them with words over V, $v_1 v_2 \dots v_k$, where for each $1 \leq i < k$ we have $p(v_{i+1}) = v_i$. For such words we define $\mathit{first}(v_1 \dots v_k) = v_1$, $\mathit{last}(v_1 \dots v_k) = v_k$. The set of all such paths we denote as P_T. Note that they might not necessarily start at the root.

1.2. JSONPath queries as node or path selectors

A JSONPath query is a sequence of *selectors*, describing a path through a JSON tree. In this paper we consider a subset of the selectors:

- The root selector `"$"`.
- Dot selector `".label"`, selecting object members by label.
- Descendants selector `"..label"`, selecting members of an object or its descendants by label.

A valid query contains exactly one root selector at the beginning. The dot and descendants selectors can be arbitrarily chained.

1.2.1. Node semantics

Under node semantics, a JSONPath query selects a set of nodes of the JSON document. Formally, a JSON query $Q = \sigma_0\sigma_1 \dots \sigma_n$ applied to a tree $T = \{V, p, \mathit{label}\}$ produces a set $Q(T) \subseteq V$ defined recursively:

- If $n = 0$ then $\sigma_0 = \texttt{\$}$ and the result is equal to $\{\mathit{root}(T)\}$.
- If $n > 0$ then we take the result of $Q' = \sigma_0\sigma_1 \dots \sigma_{n-1}$, $S = Q'(T)$. Then:
  - If σ_n is `".x"` then the result is

    $$\{v \in V \mid \mathit{label}(v) = \texttt{"x"},\, \exists_{s \in S}\, p(v) = s\}.$$

  - If σ_n is `"..x"` then the result is the set of descendants of vertices of S with the correct label, i.e.

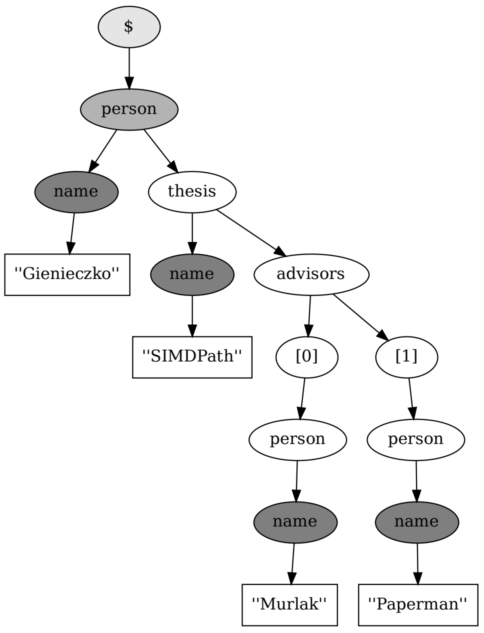
    $$\{v \in V \mid \mathit{label}(v) = \texttt{"x"},\, \exists_{k>1}\, \exists_{u_1 \dots u_k \in P_T}\, u_1 \in S,\, u_k = v\}.$$

A less formal, but more intuitive description is by defining evaluation step-by-step. The root selector selects the root node. Evaluation proceeds sequentially, in every step the next selector in the sequence is applied to the set of nodes from the previous step. A dot selector selects the children of each node with the correct label if any exist. A descendant selector selects every descendant of each node with the correct label if any exist. If a node has no matching children (respectively descendants), it is discarded.

As an example, the query `"$..person..name"` ran on the JSON from Figure 1.1 would select all four names, while the query `"$..person.name"` would restrict only to the three names nested directly in a `"person"` member. This is illustrated in Figure 1.2.

1.2.2. Path semantics

Under path semantics, 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 this marking – the paths without marking are the same, but a single path through the tree may appear twice as different results, because it is mapped to the query in a different way.



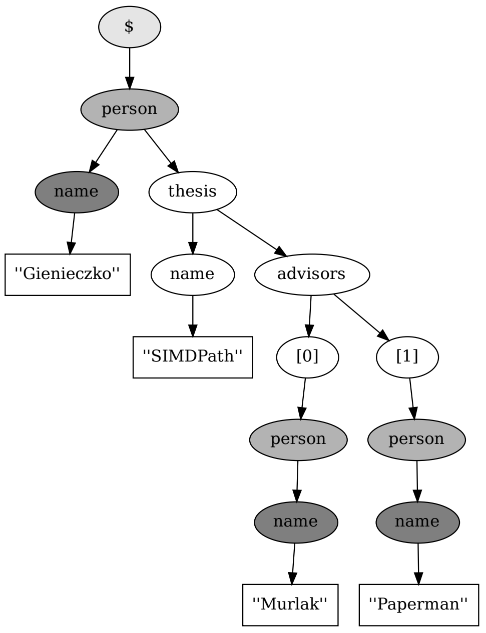


Figure 1.2: Results of running "`$..person..name`" (left) and "`$..person.name`" (right) on the tree from Figure 1.1 in node semantics. Light grey is the root node, selected by the first selector. Darkest nodes are the final result of the query. The second query first selects all "`person`" nodes, marked in darker grey, and only the direct children of these nodes are in the result set.

Consider a JSON query $Q = \sigma_0\sigma_1 \ldots \sigma_n$ applied to a tree $T = \{V, p, \mathit{label}\}$. A *marked path* is a pair consisting of a path $v_0v_1 \ldots v_k \in P_T$, and an injective function $f : [0..n] \to [0..k]$, satisfying the invariant that $f(n) = k$, i.e. the last selector always maps to the last node on the path. The query Q applied to T produces a set $Q_{path}(T)$ of marked paths, such that:

- If $n = 0$ then $\sigma_0 =$ `$` and the result is $\{\langle \mathit{root}(T), f\rangle\}$, where $f(0) = 0$.
- If $n > 0$ then we take the result of $Q' = \sigma_0\sigma_1 \ldots \sigma_{n-1}$, $Q'(T)$ and consider every $\langle \pi, f\rangle \in Q'(T)$.
  - If σ_n is "`.x`", then the result is

$$\bigcup_{\langle \pi, f\rangle \in Q'(T)} \{\pi v,\ f[n \mapsto |\pi| + 1] \mid v \in V,\ \mathit{label}(v) = \texttt{"x"},\ p(v) = \mathit{last}(\pi)\}.$$

  - If σ_n is "`..x`", then the result is

$$\bigcup_{\langle \pi, f\rangle \in Q'(T)} \{\pi\rho,\ f[n \mapsto |\pi\rho|] \mid \rho \in P_T,\ \mathit{label}(\mathit{last}(\rho)) = \texttt{"x"},\ p(\mathit{first}(\rho)) = \mathit{last}(\pi)\}.$$

As an example, the query "`$..person.name`" ran on the JSON from Figure 1.1 would select the three marked paths from the root to a "`name`" node nested directly in a "`person`" node, while the query "`$..person..name`" would select a total of six marked paths, because the

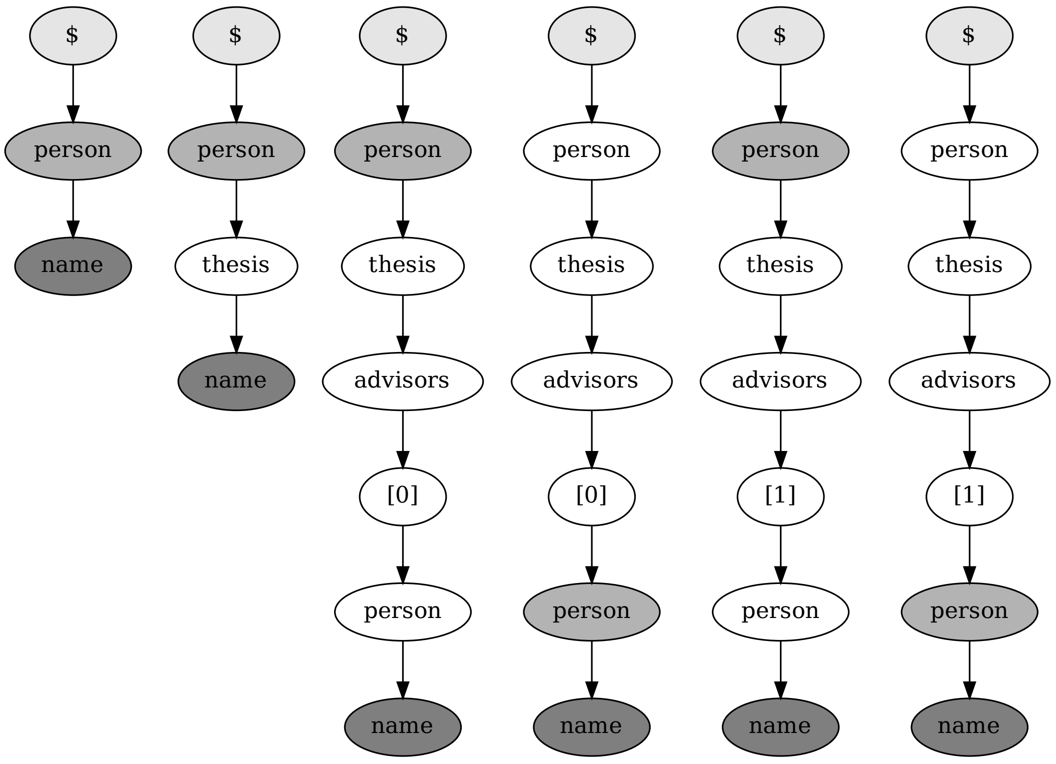


Figure 1.3: Results of running `"$..person.name"` on the tree from Figure 1.1 in path semantics. There are six marked paths matching the query, even though they describe only four nodes in the tree. Coloured nodes are the ones mapped to a selector. The root is always mapped to the first selector, and the end `"name"` node is always mapped to the `".name"` selector, but for the two longest paths there are two ways of mapping the `"..person"` selector.

descendant selector `"..person"` can be mapped to two different nodes on the two paths from root to `"name"`, as illustrated in Figure 1.3.

This is the key difference over node semantics. It appears only if there is a descendant selector `"..label"` in the query, and the document tree contains at least two nested nodes labelled `"label"`. It is obvious that if we project the results of a query under path semantics onto the first element of the pair we will get the same set of paths as we would when evaluating the query under node semantics.

1.2.3. Semantics choice

We posit that path semantics is undesirable. The difference is important for two reasons. One, cluttering results with duplicated values is usually not what the user wants. When running `"$..person..name"` on the JSON example it is more expected to get a result set of `["Gienieczko", "SIMDPath", "Murlak", "Paperman"]` (Figure 1.2) than `["Gienieczko", "SIMDPath", "Murlak", "Paperman", "Murlak", "Paperman"]` (Figure 1.3). Two, under path semantics the result set might grow very large – it is a simple exercise to construct a query and a document where the result set under node semantics has $\mathcal{O}(n)$ elements, while under path semantics it has $\mathcal{O}(n^k)$ elements, where k is the length of the query.

The original implementation by Gössner [Gös07] 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 `json-path-comparison` project [Bur+19] we identified that most currently known implementations use path semantics – out of 44 tested implementations, 34 of them use path semantics, while only 6 use node semantics (4 were errors). See Table C.1 for details. As an aside, PostgreSQL's implementation of JSONPath also uses path semantics, as discovered by C. Paperman [Pap21], which makes it possible to construct simple antagonistic queries against the database.

The current specification draft [GNB22] does not address this issue directly, but the semantics there are 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 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. This will be crucial for our constructions in Section 5.2.

1.3. Syntax assumptions

As mentioned before, only JSON documents valid w.r.t. RFC8259 [Bra17] are taken into account. The following assumptions are made and are required for our implementation to work correctly.

- The document is necessarily a valid UTF-8 encoded text.
- There are six structural characters.
  - Objects are delimited with *curly brackets*, '`{`' and '`}`'.
  - Arrays are delimited with *square brackets*, '`[`' and '`]`'.
  - Labels are separated from values in members by a *colon*, '`:`'.
  - Values in arrays and members within objects are separated by *commas*, '`,`'.
- All labels and string values are delimited with *double quotes*, '`"`'.
- Within labels and string values a character may be escaped with the *backslash* character '\'.
  - Note: since the backslash character can be escaped itself, this is equivalent to saying that a character is escaped iff it is preceded by an odd number of backslash characters. Therefore '\\\n' is interpreted as "backslash, newline", while '\\\\n' is "backslash, backslash, letter n".
- Structural characters may appear only in the predefined positions described above, or within labels or string values.
- Within labels and string values the double quotes character may only appear escaped.

All of these requirements are guaranteed by RFC8259.

The syntax of JSONPath queries is based on the specification draft [GNB22], with two distinctions.

1. The root selector is optional – since it must always appear exactly once at the beginning of the query we implicitly insert it if it is omitted.
2. We use the index selector in square brackets from the draft, but only for labels. In the draft this operator is overloaded – it accepts either a label, as in `"['name']"`, or an index to an array, as in `"[4]"`, where the former is applicable only to object nodes while the latter only to array nodes. Since our subset of selectors does not include indexing arrays we restrict it to labels. This operator allows asking for labels with special characters like dots and the dollar sign, which would otherwise be parsed as a selector. For example, selecting any node in the document with the label `"$"` can be done with `"$..['$']"`, while `"$..$"` would be an invalid query.

```
jsonpath             = [ root selector ],
                       { child selector | descendant selector } ;
root selector        = "$";
child selector       = dot selector | index selector ;
dot selector         = ".", label ;
descendant selector  = "..", ( label | index selector ) ;
label                = label first, { label character } ;
label first          = ALPHA | "_" | NONASCII ;
label character      = label first | DIGIT ;
index selector       = "[", quoted label, "]" ;
quoted label         = ( "'", single quoted label, "'" )
                     | ( '"', double quoted label, '"' ) ;
single quoted label  = { UNESCAPED | "\", ESCAPED
                       | '"' | "\'" } ;
double quoted label  = { UNESCAPED | "\", ESCAPED
                       | "'" | '\"' } ;

ALPHA                = ? [a-zA-Z] ? ;
DIGIT                = ? [0-9] ? ;
NONASCII             = ? [U+80-U+10FFFF] ? ;
                       (* non-ASCII Unicode characters *)
UNESCAPED            = ? [U+20-U+10FFFF] - ["'\] ? ;
                       (* all valid JSON label characters
                          without quotes and backslashes *)
ESCAPED              = ? [btnfru/\] ? ;
```

Figure 1.4: EBNF grammar of valid JSONPath queries in `rsonpath`.

1.3.1. Unicode escapes in labels

Valid JSON labels may contain arbitrary UTF-8 characters. Moreover, they can contain these characters escaped as a unicode point sequence, and UTF-16 characters can be encoded as two escaped sequences encoding the surrogate pair.

> ‘ *Any character may be escaped. If the character is in the Basic Multilingual Plane (U+0000 through U+FFFF), then it may be represented as a six-character sequence: a reverse solidus, followed by the lowercase letter u, followed by four hexadecimal digits that encode the character’s code point. The hexadecimal letters A through F can be uppercase or lowercase. So, for example, a string containing only a single reverse solidus character may be represented as "*`\u005C`*".*
>
> *To escape an extended character that is not in the Basic Multilingual Plane, the character is represented as a 12-character sequence, encoding the UTF-16 surrogate pair. So, for example, a string containing only the G clef character (U+1D11E) may be represented as "*`\uD834\uDD1E`*".* ’ ([Bra17], 7. Strings, pages 8-9)

This means that there are many ways of representing the same label. A label `"name"` might just as well appear in the document as `"\u0110\u0097\u0109\u0101"`. This is a major issue for our implementation, which needs to operate on a simple stream of bytes for maximum performance. The JSON specification explicitly notes this issue:

> ‘ *Software implementations are typically required to test names of object members for equality. Implementations that transform the textual representation into sequences of Unicode code units and then perform the comparison numerically, code unit by code unit, are interoperable in the sense that implementations will agree in all cases on equality or inequality of two strings. For example, implementations that compare strings with escaped characters unconverted may incorrectly find that "*`a\\b`*" and "*`a\u005Cb`*" are not equal.* ’ ([Bra17], 8.3. String Comparison, page 10)

This is indeed the case for our implementation. Properly dealing with this issue, i.e. converting the labels in the source document to a normalised representation every time a label needs to be compared, is prohibitively slow. Therefore, by default our implementation performs ordinal comparison – labels `"name"` and `"\u0110\u0097\u0109\u0101"` are not equal. if both of these labels were located in a document, then the query `"$..name"` would find only the former, while `"$..['\\u0110\\u0097\\u0109\\u0101']"` would find only the latter.

Chapter 2

SIMD model

Single Instruction, Multiple Data (SIMD, pronounced sim-dee) is an umbrella term for parallelisation techniques that do not involve additional processing units, instead utilising a single unit to perform an operation on many data points in a single step. The idea is extremely old, with simple vectorisation being presented by Lamport as long as ago as in 1975 [Lam75]. Today SIMD usually refers to the usage of special *SIMD instructions*, intrinsic to the CPU platform, that enable such parallelisation.

Before we talk about special instructions, consider a simple problem that can be made faster using SIMD with standard tools. We have two redundant sensors that provide us with a stream of measurements. The input is two arrays of 8-bit values, $a = a_0, a_1, \ldots, a_{N-1}$, $b = b_0, b_1, \ldots, b_{N-1}$, and the task is to find the first discrepancy in measurements, i.e. the least index i such that $a_i \neq b_i$, or a special `None` value if there is no such index.

A sequential algorithm would iterate from 0 to $N-1$ and compare the corresponding bytes. However, assuming that the data lies sequentially in memory for the respective sensors, we can be much more efficient by loading chunks of it into the CPU's registers. Assume we're working on an architecture with 64-bit registers. Then we can load a block of 8 bytes from a into one register, the corresponding block of 8 bytes from b into another, and XOR them together. If the resulting integer has any lit bits, then there was a discrepancy in the data at the position of that bit. Because loading a 64-bit register is about as fast as loading a single byte into a register, we can hope for a speed-up up to a factor of 8. Indeed, the benchmark described in Section 2.6 shows a 6 times speed-up between the functions listed in Figure 2.1.

The idea should be clear – we pack *multiple data* points into a single CPU register, so that a *single instruction* can yield results for all the data at once. The code of this optimisation (Figure 2.1) already exemplifies constraints on the data streams that need to be asserted for SIMD operations to be sound – we describe these in Section 2.5.

2.1. Preliminaries

In this entire section we assume that the target CPU architecture is little-endian. This matters when we interpret bit vectors as sequences of bytes – the first 8-bits of a vector are the first byte. Note that this is not a requirement of our solution, as it works on big-endian machines as well. A single integer is interpreted normally, however – the 8-bit number 11000110 is 198 in decimal and `0xc6` in hexadecimal.

We describe vectorial operations using two notations. The first is by specifying the raw bits of a vector, e.g. saying that a 16-bit vector is equal to 0011001111110000. The other is by interpreting the vector as individual numbers with given *granularity*. So the above vector

```
fn seq(a: &[u8], b: &[u8]) {
    let N = a.len();
    for i in 0..N {
        if a[i] != b[i] {
            return Some(i);
        }
    }
    return None;
}
```

```
fn simd(a: &[u8], b: &[u8]) {
    const SIZE = 8;
    let number_of_blocks = a.len() / SIZE;
    for i in 0..number_of_blocks {
        let start = i * SIZE;
        let end = (i + 1) * SIZE;
        let a_vec = a[start..end].as_u64();
        let b_vec = b[start..end].as_u64();
        let xor = a_vec ^ b_vec;
        if xor != 0 {
            let idx = start +
                (xor.trailing_zeros() / SIZE);
            return Some(idx);
        }
    }
    return None;
}
```

Figure 2.1: Discrepancy search algorithm, sequential (left) vs 64-bit vectorised (right). The Requires `a` and `b` to follow assumptions described in Section 2.5.

interpreted with 16-bit granularity is $[13296]_{16}$, under 8-bit granularity it is $[51, 240]_8$, and under 4-bit granularity it is $[3, 3, 15, 0]_4$. Note that real SIMD operations usually have lowest granularity of 8 bits, but to keep examples clear and concise we often use 4-bit granularity. We sometimes display bytes in hexadecimal when presenting the values of both nibbles (4-bit halves) is beneficial for clarity.

Code examples in this chapter are presented in Rust-like pseudocode – they omit most details not essential for the thesis, but turning them into correct Rust should be an easy exercise. We use some Rust-specific notation.

- Integer types are defined as the letter `i` or `u` followed by their size in bits, so `u8` is a byte, `i32` is a 32-bit signed integer, etc. The special `usize` type is an integer guaranteed to be of the same width as a pointer on the target architecture.
- The type `[u8; 64]` is an array of 64 bytes. Other sizes are also permitted, but must be constant.
- The type `&[u8]` is a non-mutable reference to an array of bytes. The ampersand symbol always signifies a reference to the type it precedes.
- An expression `&x[i..j]` for an array `x` represents the reference to a slice of the array from the i-th (inclusive) to j-th (exclusive) element.
- Most functions that are supposed to find something in a byte stream return an `Option` type, which is an algebraic union type of `None` and `Some(T)` for some type `T`. We construct a value of this type like in functional languages, using `None` or `Some(value)`.

2.2. SIMD extensions

Modern SIMD is achieved by additional CPU registers and special operations on those registers. The SIMD registers are larger than regular ones, starting at 128-bits. The operations can be roughly categorised into three kinds:

- **pack**, take existing data in regular registers or memory and load them into the SIMD registers;
- **transform**, manipulate the SIMD registers to produce new vectors;
- **extract**, convert a vector in a SIMD register into a regular value.

It is important to note that the transformations that can be done on SIMD are basically logical gates – they take some input vectors and produce output vectors[1]. Since the SIMD registers are separate from conventional registers, one must explicitly opt into the SIMD world with the **pack** operations; and then to do any of the regular CPU instructions like conditional jumps, one must explicitly opt out of that setting with the **extract** operations.

SIMD works best as a *pipeline*, where the SIMD operations can work on their vectors uninterrupted, without switching between the two worlds. Because the individual operations are a bit more costly, but also much more effective in terms of bytes processed per instruction, local parallelism and branch prediction are massively impactful. This means that for fast algorithms utilising SIMD we should strive to be *branchless*, i.e. avoid any conditional jumps that the CPU might mispredict. Even more important is avoiding function calls – SIMD registers need to be saved on the stack before a call, which defeats the pipeline. One can deduce from this that conceptually a SIMD algorithm proceeds in three phases:

- **prepare** the data in SIMD registers;
- **pipeline** as many operations as possible with the SIMD registers;
- **process** the results, extracting data from the end of the pipeline.

The phases naturally correspond to the types of operations we have. We first **prepare** the data by **packing** it into SIMD registers, **transform** the data in a **pipeline**, and then **extract** and **process** the results.

2.2.1. Basic SIMD operations

Different CPU architectures have different SIMD instruction sets. Not only that, but some architectures have many different SIMD instructions sets within themselves, e.g. x86 with SSE, SSE2, SSE3, SSSE3, AVX, AVX2, AVX512... For this thesis we focus on the x86 architecture, although it is important to note that only the classifier (Chapter 4) is architecture-specific, while the main query engine loop is architecture-agnostic. We briefly describe the available instructions, putting emphasis on AVX2, which is the most common instruction set and the one we support in `rsonpath`.

Virtually all SIMD architectures have standard bitwise operations like AND, OR, as well as common arithmetic, logical, and comparison operations. All of the latter operations can interpret the vectors as packed values of given bit length. Recall that, for example, the 32-bit vector $(00001111)^4$ interpreted with 8-bit granularity is a four-element sequence $[15, 15, 15, 15]_8$, while interpreted with 16-bit granularity is a two-element sequence $[3855, 3855]_{16}$.

This granularity is meaningful for most non-bitwise operations. Consider vectors 11101111 and 00000001. Under 8-bit granularity the result of addition on them is 11110000. Under 4-bit granularity the result is 11100000. This is because $[239]_8+[1]_8 = [240]_8$, but $[14, 15]_4+[0, 1]_4 =$

[1]This relation to logical circuits is further formalised in Appendix B.

$[14, 16]_4$. The 4-bit element overflows and wraps to 0, resulting in $[14, 0]_4$. For comparison operations this becomes even more meaningful. A comparison operation sets all the bits of the result for the chunk of the vector for which the comparison was true. Therefore, if we compare 11001100 with 11000011 under 8-bit granularity, we will get an all-zeroes vector, whereas under 4-bit granularity we get 11110000, because the first 4-bit elements were equal, and the second ones were not.

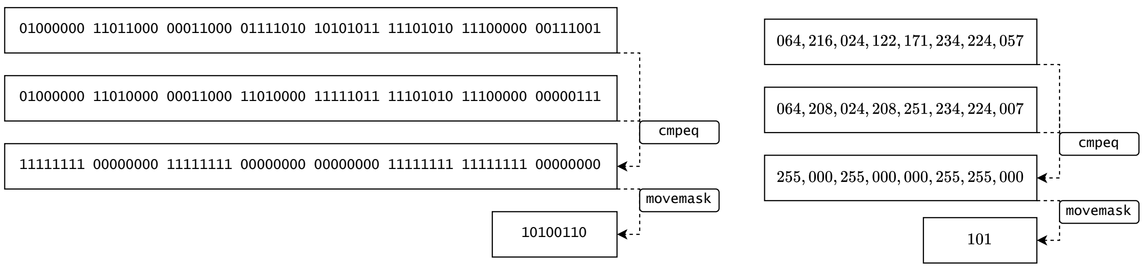


Figure 2.2: SIMD operation identifying positions at which two 64-bit vectors interpreted with 8-bit granularity agree. By examining the bits of the resulting 8-bit integer we can find the positions. Left side shows the bit vectors, whereas the right side their decimal interpretation under 8-bit granularity.

2.3. Intrinsics

SIMD is defined in terms of CPU instructions and registers. That is a fine interface for compilers attempting to optimise user code written in a high-level language, but coding in assembly would be unwieldy for regular developers.

Therefore, SIMD operations come with a set of library functions called *intrinsics*. These are functions provided by authors of the SIMD instruction sets that are well-known to the compiler and implemented in assembly using the SIMD instructions. The compiler encountering such a function call inlines the SIMD instructions at the call site. This allows us to write code in a higher-level language like Rust while still getting full performance benefits of using the low-level SIMD instructions.

When describing our algorithms we refer to intrinsics names instead of the actual CPU opcodes. Note that intrinsics introduce only the thinnest level of abstraction – an intrinsic is tightly bound to the CPU instructions it produces, and an x86 AVX2 intrinsic will produce code that is valid only on an x86 architecture CPU supporting AVX2.

2.4. x86

The x86 defines many SIMD instruction sets developed by Intel, with vector sizes spanning from 128-bit to 512-bit. The intrinsics names remain similar throughout the extensions, differing only in the register width – e.g. AVX has `_mm256_cmpeq_epi8`, while AVX512 has `_mm512_cmpeq_epi8`. An important operation available on all x86 SIMD implementations is the `movemask` operation, which is an **extract** operation that allows smooth transitions from the SIMD world to the sequential world.

The operation collapses a SIMD vector into a regular-sized value, turning chunks of the SIMD vector of some granularity into the bits of the regular value by examining the source's most significant bits. In other words, applying `movemask` of 8-bit granularity to a vector of 64 bytes results in a 64-bit number in which the i-th bit is lit if and only if the most significant bit of the i-th byte in the vector was lit. For example, deciding whether two vectors are exactly equal can be done by vectorial `cmpeq` comparison and then collapsing the result with `movemask` – the result is all-ones if and only if the two vectors were equal. This is a core operation that allows us to quickly examine results of a SIMD pipeline. For an application see the vectorised discrepancy algorithm in Section 2.6.

2.4.1. Instruction sets

SSE2

Streaming SIMD Extensions 2 were introduced in 2000 by Intel in Pentium 4. It is the oldest widely used SIMD instruction set with support for integer operations (SSE supported only floating-point operations). It utilises 128-bit wide registers labelled `xmm0`, `xmm1`, ..., `xmm15` (some old configurations have only eight registers, most have sixteen). It gives us the `__m128i` type representing a single SIMD vector and most typical operations on such vectors:

- loads: `_mm_set1_epi8`, `_mm_load_si128`, ...
- arithmetic operations: `_mm_add_epi8`, `_mm_sub_epi8`, ...
- logical operations: `_mm_and_si128`, `_mm_xor_si128`, ...
- comparisons: `_mm_cmpeq_epi8`, `_mm_cmpge_epi8`, ...
- movemask: `_mm_movemask_epi8`.

The `set1` operation allows us to set all chunks of a vector to the same value of given size. So `_mm_set1_epi8(byte)` replicates the `byte` 16 times into the vector. The `_mm_load_si128` simply loads 16 bytes from a given pointer into a SIMD register. For SSE2 it is extremely important to align the data to the 16-byte boundary, as loading unaligned data (with a special `_mm_loadu_si128` intrinsic) is reported in folklore to incur massive performance penalties (see Section 2.5).

AVX2

Advanced Vector Extensions 2 were introduced in 2013 by Intel with the Haswell architecture (4th generation Intel Core), improving on the previous AVX extension. It is the most mainstream SIMD instruction set, available in any desktop CPU shipped in the last decade. It utilises 256-bit wide registers labelled `ymm0`, `ymm1`, ..., `ymm15`, extending the `xmm` registers from SSE. The type representing an AVX2 vector in intrinsics is `__m256i`. In accordance with the x86 SIMD naming scheme, the intrinsics for AVX2 are, among others:

- loads: `_mm256_set1_epi8`, `_mm256_load_si256`, ...
- arithmetic operations: `_mm256_add_epi8`, `_mm256_sub_epi8`, ...
- logical operations: `_mm256_and_si256`, `_mm256_xor_si256`, ...
- comparisons: `_mm256_cmpeq_epi8`, `_mm256_cmpge_epi8`, ...
- movemask: `_mm256_movemask_epi8`.

AVX512

Extensions introduced by Intel in 2013 for workstations and in 2017 for consumer CPUs with Skylake-X. It is the cutting-edge SIMD instruction set for x86, utilising 512-bit wide registers `zmm0`, `zmm1`, ..., `zmm35` (number of SIMD registers increased to 36). Being the newest extensions, these are not as widespread as AVX2, and support for their intrinsics is still experimental in Rust. Additionally, heavy use of 512-bit instructions may lead to performance *degradation*, due to power consumption and heat issues. In our implementation we focus on AVX2 as the main target.

2.4.2. Lanes

Even though the SIMD registers are larger than 128 bits in AVX and above, they are actually divided physically into *lanes* of size 128. Most operations are unable to move any data between the lanes [ML17]. This is usually unnoticeable, as most of the time the operations performed on separate chunks of the vector are independent of each other. Sometimes, however, one would like to perform an operation on the entire vector at once, treating it with granularity larger than 128.

An example is the bitwise shift operation. We move each bit of the vector a fixed number of positions to the left or to the right, shifting in zeroes in missing spots. One would expect that a left-shift by one position the 256-bit vector 1^{256} would result in $1^{255}0$. That is not the case – the lanes are separate and are shifted separately, causing the result to be $1^{127}01^{127}0$.

This is an issue if we want to perform operations across lane boundaries. The physical separation carries further implications, as some operations are available in cross-boundary variants, but their performance is noticeably lower than of the basic variant instruction. It is best to avoid cross-boundary operations if possible.

2.4.3. CLMUL – carry-less multiplication

The carry-less multiplication extension was introduced in 2010 as an improvement for block cipher algorithms working in Galois/Counter mode (for example AES-GCM). It is defined as multiplication in the $GF(2)$ field, where a bit string $a_0a_1\ldots a_{63}$ represents a polynomial $a_0 + a_1X + a_2X^2 + \ldots + a_{63}X^{63}$. The formal definition of the operation is as follows: given two bit strings a, b of length 64 define the *carry-less product* of a and b as:

$$c_i = \bigoplus_{j=0}^{i} a_j b_{i-j},$$

i.e. an exclusive alternative of products of bits of a and b.

At first glance it is not at all clear how this operation could be useful for the purpose of querying JSON documents, but it turns out to be a crucial optimisation for determining escape sequences [LL19]. The key observation is that by setting $b = 1^{64}$ we obtain the following formula:

$$c_i = \bigoplus_{j=0}^{i} a_j,$$

which is simply a prefix-XOR operation. The application of this is described in detail in Section 4.2. This operation is exposed via the `_mm_clmulepi64_si128` intrinsic, and it is important to note that it operates only on 64-bits – one needs to select which halves of a and b will get multiplied. So this is in fact a 64-bit-width SIMD operation.

2.4.4. Shuffle – nibble lookup tables

The shuffle operation, available on x86 via `_mm_shuffle_epi8`, `_mm256_shuffle_epi8`, or `_mm512_shuffle_epi8`, is arguably the most important optimisation for input character classification, and a key idea of [ML17] and [LL19]. In short, it allows us to use the lower nibbles (4-bit parts) of a byte as indices to a lookup table. This operation is straightforward for a 128-bit vector, as they contain 16 bytes, so we can express all valid indices with a 4-bit number.

As a simple example, take the 128-bit vector *index* that represents $[15, 14, \ldots, 0]_8$ when taken with 8-bit granularity. Then we can reverse the bytes in any SIMD vector *source* with `_mm_shuffle_epi8(source, index)`. The operation can be simply defined, for all $0 \leq i < 16$:

```
shuffle_epi8(source, index)[i] = source[index[i] & 0x0F]
```

For a more complicated example see Figure 2.3.

For larger vectors indices are no longer between 0 and $2^4 - 1$. Additionally, because of the physical lane separation (Section 2.4.2), the operations disallow shuffling across lane boundaries. Instead, multiple shuffle operations are performed in parallel on independent lanes. The nibbles in the first 16 elements of the index vectors are used to index the first 16 elements of the source vector, next 16 elements are used to index between 16 and 31, etc. To make this precise, here is the pseudocode representing this operation for a 512-bit width SIMD vector:

```
fn _mm512_shuffle_epi8(a: [u8; 64], b: [u8; 64]) {
    let mut result = [0; 64];
    for j in 0..64 {
        let lower_nibble = b[j] & 0x0F;
        // Integral division. If 0 <= j < 16, this is 0.
        // If 16 <= j < 32, this is 16, etc.
        let offset = (j / 16) * 16;
        let index = lower_nibble + offset;
        result[j] = a[index];
    }
    return result;
}
```

The key observation is that we can consider this operation as a lookup instead of a shuffle. We can use it to quickly classify a vector of bytes into at most 16 buckets depending on their lower nibble. By performing a second lookup on the upper nibble (by simply shifting all bytes right by 4) and combining the results we get a generic lookup operation. This is applied in the core part of the branchless classifier, described in Section 4.1.

2.5. Padding and alignment

For the algorithm in Figure 2.1 to be sound, two conditions must be met:

1. the length of the array we wish to split into blocks has to be divisible by the block size of k bytes; and

2. each block must be k-byte aligned.

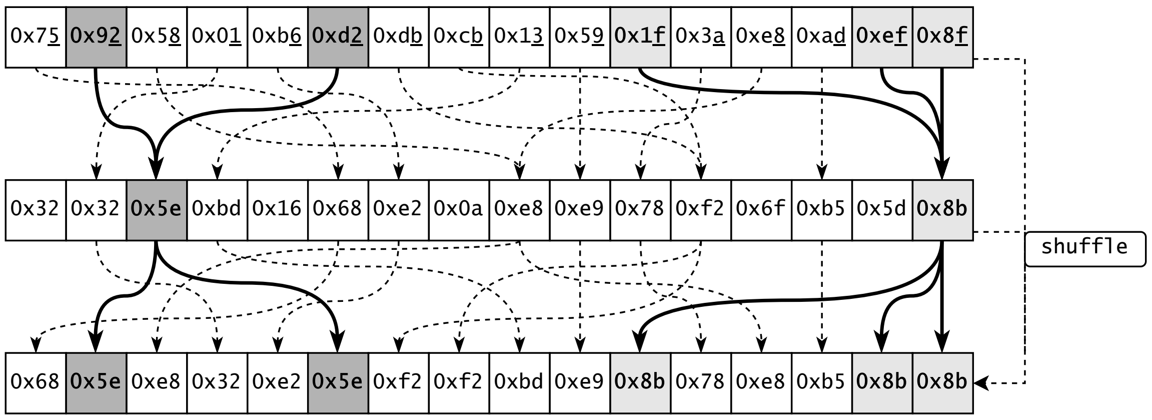


Figure 2.3: Example shuffle operation where a source vector (middle) gets shuffled with the index vector (top). The lower nibble of each index byte (underlined) is used as an index into the source vector. For example, the bytes with lower nibble 2 get mapped to `0x5e` located at index 2 of the source vector (dark grey), while bytes with lower nibble `f` get mapped to the last source value (light grey). Some source values are never used, as there are no bytes with lower nibble 0 in the index vector; these are indices 0, 4, 7, 12, and 14.

2.5.1. Padding

The first condition is rather obvious – we would go outside the bounds of input otherwise. There are two natural ways of enforcing this precondition.

- One may limit the vectorised algorithm to the prefix of the input that is divisible by the block size, and then perform the final phase of the algorithm sequentially on the remaining, small portion of data. The disadvantage of this approach is that it essentially requires us to write two separate algorithms. It might also be not obvious how to flow carry information (as outlined in Section 4.2.1 and Section 4.3.1) from the SIMD phase to the sequential phase.
- In case of many algorithms it is possible to *pad*, artificially increasing the length to be divisible by k. For our discrepancy example, one can simply pad the arrays with zeroes without changing the output of the algorithm. In case of JSON processing this approach is also viable – we pad the input document with a number of space characters (`U+0020`) that will necessarily be ignored. Somewhat equivalently, one can pad the input string with NULL terminators.

The latter approach is easier to employ, since it needs to be done only once before feeding the input to the algorithm and is separated from the implementation itself. In the rest of the paper we assume that the input size is divisible by the block size.

2.5.2. Alignment

A somewhat subtler issue is alignment. All types of data have their alignment, which indicates the correct memory addresses at which a value of the type can be stored. Alignment is given in bytes – 32-bit integers are 4-*aligned*, 64-bit integers are 8-*aligned*, etc. This alignment must be preserved – on many platforms loading a misaligned value incurs a hefty performance penalty at best, or results in an incorrect value read at worst. Versions of ARM before ARMv6 were permitted to throw an MMU exception on unaligned access, or just silently coerce the access into aligned, yielding incorrect results.[2]

Because of subtle differences in how unaligned reads are treated on different architectures, *dereferencing a misaligned pointer in Rust is undefined behaviour.* Therefore, the code in Figure 2.1 implemented in Rust would be incorrect, because the conversion between a block of input bytes to `u64` is undefined behaviour, unless the input is aligned. Moreover, AVX2 aligned load instructions may cause a segmentation fault if the input data is not aligned. Our implementation needs to take special care to always use data aligned to an expected boundary. Since this would devolve into a quagmire of hard to track bugs if done on raw pointers, our solution is to solve alignment on a type level (see Appendix A).

A byproduct of loading misaligned data into large SIMD vectors is that the loads will always happen across a CPU's cache line, which is more costly. For example, in AVX512 on an Intel CPU, the size of the vector is exactly the same as of the cache line – 64 bytes. Hence *every* load of misaligned data on AVX512 will be across cache lines. Therefore, while aligning data does not necessarily improve performance, misaligned loads may degrade it.

2.6. Example vectorial algorithm – discrepancy search

As a motivating example we can apply x86 SIMD extensions to the discrepancy search problem described in the introduction of this chapter, and vectorised with 64-bit registers in Figure 2.1. The algorithm is similar to the 64-bit case. We will load a chunk of data into SIMD registers, use the `cmpeq_epi8` intrinsic to compare the two vectors, extract that data with a `movemask_epi8`, and check if it is all-ones. If there are any zeroes, we report the first one's position. The pseudocode for AVX2 is presented below, with SSE2 and AVX512 solutions being very similar.

```
fn discrepancy_size256_impl(a, b) {
    const SIZE: usize = 32;

    // Produce a sequence of pairs of 32-byte blocks from both inputs
    // and mark them with their index starting from zero.
    let blocks = a.iter_blocks().zip(b.iter_blocks()).enumerate();

    for (i, (a_block, b_block)) in blocks {
        let a_vec = _load_si256(a_block);
        let b_vec = _load_si256(b_block);
        let xor = _cmpeq_epi8(a_vec, b_vec);
        let mask = _movemask_epi8(xor) as u32;

        if mask != 0xFFFFFFFF {
            let idx = i * SIZE + mask.trailing_ones();
            return Some(idx);
        }
    }
    return None;
}
```

[2]This appears to be easily verifiable folklore, usually in form of confused developers reporting errors when accessing 32-bit values not aligned to the 4-byte boundary.

In case of AVX512 there is no separate `cmpeq` function, only a fused version that returns the mask result: `_mm512_cmpeq_epi8_mask`. Regardless, the reported latency and throughput is not worse than of the two separate instructions in AVX2 – `cmpeq` has latency 1 and `movemask` 2, while AVX512's `cmpeq_mask` has latency 3 [Int22]. See Table 2.1 for full results.

The construction of this algorithm follows the outline we described above. We first **prepare** the data into two SIMD registers, **transform** it with `cmpeq`, and then **extract** it into a bitmask. Then the conditional check on the mask must occur, as well as a jump to the next iteration of the loop.

Implementation	**Mean time (μs)**	**Throughput (GB/s)**	**Ratio**
Sequential (8-bits/iteration)	2381.0	1.76	1.0000
Machine word (64-bits/iteration)	572.2	7.33	0.2403
SSE2 (128-bits/iteration)	375.8	11.16	0.1578
AVX2 (256-bits/iteration)	313.7	13.37	0.1317
AVX512 (512-bits/iteration)	311.4	13.47	0.1308

Table 2.1: Benchmark results of discrepancy-search.

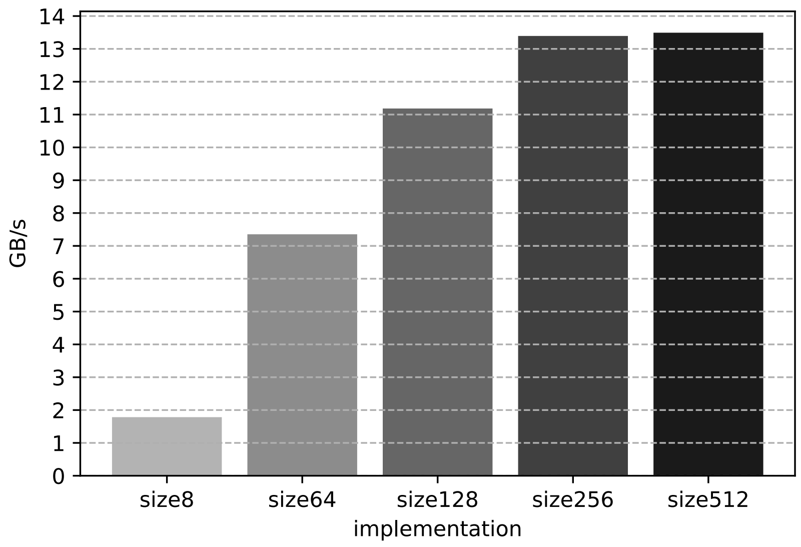


Figure 2.4: Throughput plot of Table 2.1

2.6.1. Performance

The data is 4 megabytes of random bytes, equal between both vectors except for the final byte. We benchmarked the sequential and machine-word vectorised solutions (see Figure 2.1) against three different SIMD solutions: SSE2, AVX2, and AVX512. Unsurprisingly, largest SIMD vector size has the best performance, as shown in Table 2.1 and Figure 2.4. It is pretty close to the ideal throughput of 17 GB/s (see Section 3.1.1). A perhaps surprising characteristic is that AVX512 is not much faster than 256-bit AVX2. Our hypothesis is that this is the AVX512 throttling in play – the power consumption and heat generation on the CPU are higher than the gains obtained from a larger SIMD vector.

Chapter 3

Branchless streaming algorithms

In this chapter we present a number of experiments with SIMD acceleration that are related to the query engine, but ultimately did not find a place in the engine implementations. We feel their inclusion is beneficial for better understanding of the final classifier implementation, described in Chapter 4.

3.1. Find byte – `memchr`

The simplest query operation we can perform on a stream on bytes is to find the first occurrence of a specific byte. The core implementation of such an algorithm is the `memchr` function from C standard library. It is a very well optimised routine, actually using SIMD operations on x86 under the hood, albeit in handwritten assembly instead of C intrinsics. As the experiment we compared our implementations to the standard `memchr` implementation to see if we could achieve its level of throughput with a SIMD pipeline.

A sequential algorithm for this problem is trivial. For a SIMD approach with vector size of k-bytes we first load the byte we want to find, b, into a SIMD register replicated k times. Then we iterate over blocks of k bytes and use `cmpeq_epi8` to compare each of the k bytes to b. Then we can `movemask` and compare the resulting mask to 0 to see if b was found.

```
fn find_byte(byte, bytes) {
    const SIZE: usize = 32;
    let byte_mask = _set1_epi8(byte as i8);
    // Sequence of 32-byte blocks marked with their index.
    let blocks = bytes.iter_blocks().enumerate();

    for (i, block) in blocks {
        let vec = _load_si256(block);
        let cmp_vector = _cmpeq_epi8(vec, byte_mask);
        let cmp_packed = _movemask_epi8(cmp_vector);

        if cmp_packed != 0 {
            let idx = i * SIZE + cmp_packed.trailing_zeros();
            return Some(idx);
        }
    }
    return None;
}
```

Using that code we can achieve throughput comparable, yet inferior, to the out-of-the-box `memchr` implementation.

Additionally, we compare an idiomatic Rust one-line implementation of this search using iterators. Examining the assembly output of such a solution reveals that the compiler vectorises the search by loading 8-byte fragments into CPU registers and comparing them, similarly to the Machine word implementation from Table 2.1.

```
fn find_byte_rust_optimised(byte, slice) {
    slice.iter().position(|&x| x == byte)
}
```

Implementation	**Mean time (ms)**	**Throughput (GB/s)**	**Ratio**
Sequential (8-bits/iteration)	19.361	1.733	1.0000
Rust-optimised (64-bits/iteration)	9.324	3.598	0.4816
SSE2 (128-bits/iteration)	2.941	11.407	0.1519
AVX2 (256-bits/iteration)	2.604	12.888	0.1345
AVX512 (512-bits/iteration)	2.250	14.911	0.1162
`memchr`	1.944	17.268	0.1004

Table 3.1: Benchmark of find-byte implementations.

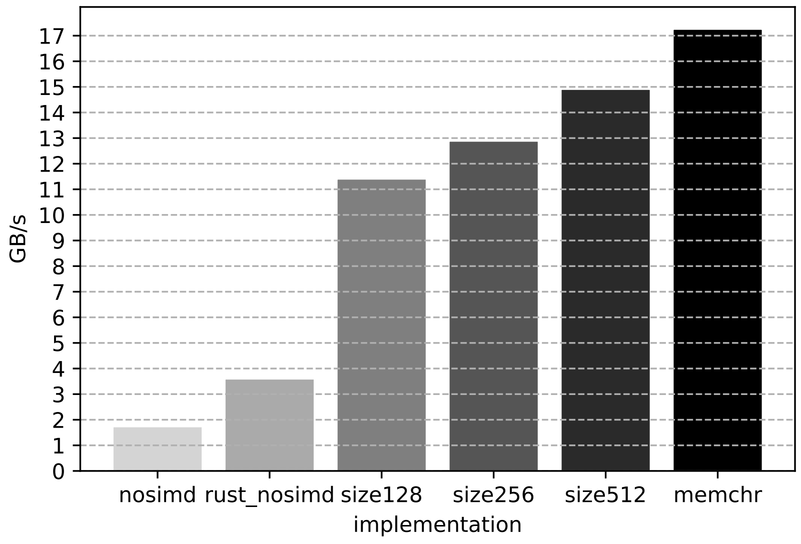


Figure 3.1: Throughput plot of Table 3.1

3.1.1. Performance

The data is constructed by repeating all lowercase ASCII letters until 32 MiB are generated. Then a single uppercase 'X' is appended. The implementations look for the uppercase 'X', thus they have to read the entire input. We benchmarked the sequential and Rust-idiomatic solution (using a standard library `position` function) against four different SIMD solutions:

SSE2, AVX2, AVX512, and `memchr` from glibc. The `memchr` implementation can be considered as state-of-the-art and the absolute limit on how performant any nontrivial algorithm on a byte stream can be – finding a single byte is the most basic problem one could define. Therefore, around 17 GB/s is what we consider a performance ceiling.

A naive sequential implementation is slower than Rust's well-optimised library method, which is not surprising. SIMD implementations outclass all others, with larger vector sizes being more performant. The `memchr` implementation on the architecture of the benchmark machine (x86_64) appears to be handwritten assembly code, that combines SIMD vectorisation with loop unrolling – 4 blocks are processed in a single loop iteration.

3.2. Find sequence – `memmem`

An extension of the previous algorithm is to find a *sequence* of m bytes $b_1 b_2 \dots b_m$ in the input. This could be useful for locating labels in the input JSON. To arrive at a general algorithm, first consider the problem for $m = 2$. In such a case we can replicate b_1 into one vector, b_2 into another, and then compare batches of k bytes against both with two `cmpeq_epi8` instructions.

If we have a vector where marked positions signify "b_2 is at this position in the input", then by bitwise shifting that vector to the right[1] by one we get a vector where marked positions signify "b_2 is at the next position in the input". By combining the vectors "b_1 is at this position" with "b_2 is at the next position" with an AND operation we get "the sequence $b_1 b_2$ is at this position".

One approach would be to perform this transformation on each block of k bytes and with that find all occurrences of $b_1 b_2$. This does not work. There is an edge case where the block segmentation causes b_1 to occur at the end of a block and b_2 at the start of the next block. Such sequences will not be detected by this naive algorithm. There are two possible solutions to this.

1. **Pipeline state** – introduce state to the pipeline that will be carried from block to block; in this case, a single bit indicating whether the previous block ended with b_1. In practice, to avoid branching, we store a full bitmask where the first bit is lit if and only if the previous block ended with b_1. Then that mask can combined with the main algorithm with a simple OR.

2. **Overlapping windows** – run the algorithm on two blocks at a time, *current* and *next*. This way, a sequence spanning two blocks will always be caught in some iteration of the algorithm. Naively, this would cause almost all blocks to be processed twice. However, when we calculate transformations for a *next* block, then the results can be used when it becomes the *current* block in the next iteration.

The overlapping windows solution causes some work to be done twice on the same block, but it scales better than the state solution, since for a sequence of length m we would need to hold $m - 1$ masks.

The general algorithm will work for any $m \leq k$. We compute the m masks for each of the bytes, shift the i-th by $i-1$ to the right, and AND them together. Care has to be taken during the final step to enable local parallelism for the executing CPU. To see this on an example,

[1]Due to endianness, this might be counterintuitive. While we might think about the input stream being read left-to-right, in a bitmask shifting left actually means moving the stream forward, while shifting right moves it backwards.

assume that $m = 8$ and we have computed masks `mask1`, `mask2`, . . . , `mask8`. Consider the two ways of computing the result mask `and7`:

```
let and1 = mask1 & mask2;         let and1 = mask1 & mask2;
let and2 = and1 & mask3;          let and2 = mask3 & mask4;
let and3 = and2 & mask4;          let and3 = mask5 & mask6;
let and4 = and3 & mask5;          let and4 = mask7 & mask8;
let and5 = and4 & mask6;          let and5 = and1 & and2;
let and6 = and5 & mask7;          let and6 = and3 & and4;
let and7 = and6 & mask8;          let and7 = and5 & and6;
```

The left implementation is serial – each result depends on the previous result. Because of this data dependence, the CPU is forced to execute those instructions one-by-one, waiting for `and`i to finish before computing `and`$(i + 1)$. On the right, data independence is maximised – the values of `and1`, `and2`, `and3`, `and4` can all be computed in parallel, same is true for `and5` and `and6`. When data flow is drawn out, the code on the left forms a linear path, while the code on the right forms a binary tree. For each m the optimal shape of a tree needs to be considered and proper instructions emitted.

For $m > k$ we employ a heuristic – first try to match the prefix of the sequence of length k in vectorised fashion. If it matches, compare the remaining suffix sequentially.

3.2.1. Code synthesis with Rust procedural macros

Since each value of m requires different code – more instructions, different shape of the final AND tree – and it has to be branchless, there are m different functions computing results for each of them. For a maximal value of $m = 32$ this results in over five thousand lines of Rust code just for AVX2. It would be asinine to write that code manually, therefore we use code generation enabled by the Rust ecosystem.

Rust allows parsing and synthesising more Rust code procedurally, with a combination of build scripts [Rus15], types representing Rust abstract syntax trees [Tol17] and quasiquotation [Tol16]. Quasiquotation originated in Lisp [Baw99] and is a well-known technique in languages like Haskell [Mai07]. It allows a programmer to write down code in a formal language and then manipulate its structure procedurally like any other data type. Thanks to it, we can write relatively short code describing the shape of the solution for any m (around 300 lines for AVX2), and then manipulate it to produce all 32 versions of the function during compilation.

3.2.2. Performance

The data consists of 32 MiB of all lowercase ASCII letters sorted lexicographically, and then a shuffled sequence of letters. The implementations look for prefixes of this sequence. We compared an idiomatic Rust implementation[2] and a widely used crate `memchr` [Gal15] with its `memmem::find` function to two SIMD implementations: SSE2 and AVX2. We did not implement an AVX512 version.

Interestingly enough, sequential implementation works significantly faster on sequence lengths being powers of two. This is because of Rust's compiler – if we examine the produced assembly we can see automatic vectorisation. The code produced for length 4, for example, automatically packs 4 bytes at a time into a single register, just like we did in our first attempt

[2]Using the `windows` function, which creates a stream of all slices of given length, and then using `position` to find the occurrence of the sequence.

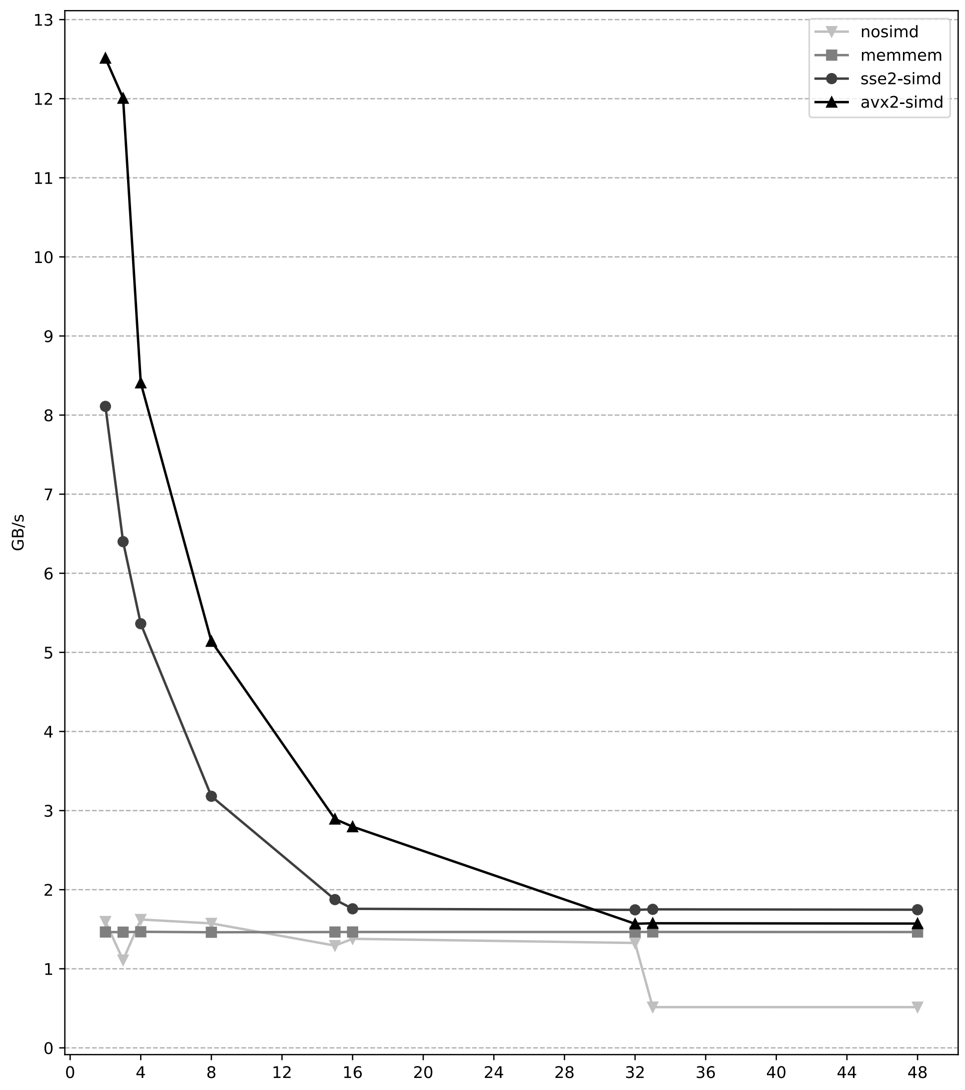


Figure 3.2: Throughput plot of Table 3.2.

at optimising the Discrepancy problem (Figure 2.1). For a sequence of 3 bytes it cannot apply that trick, so the performance is worse.

Between AVX2 and SSE2 the longer SIMD vector is clearly the winner. However, for long sequences, it appears that it might be more beneficial to run the heuristic on shorter sequences to quickly discard false matches, since SSE2 performs slightly better in that regard.

The `memmem` implementation has stable performance that outclasses sequential implementations for most sequences, but performs significantly worse than SIMD versions. It is important to note that our benchmark is ran on only a single input that might not be representative of real-world use cases, and we only consider throughput, while the `memchr` crate puts some emphasis on latency. However, it appears like our SIMD implementations could be a promising avenue of improvement for the crate.

Implementation	**Length**	**Mean time (ms)**	**Throughput (GB/s)**	**Ratio**
Sequential	2	21.004	1.598	1.00
`memmem`	2	22.910	1.465	1.09
SSE2 (128-bit)	2	4.137	8.110	0.20
AVX2 (256-bit)	2	2.682	12.513	0.13
Sequential	3	30.223	1.110	1.00
`memmem`	3	22.937	1.463	0.76
SSE2 (128-bit)	3	5.243	6.400	0.17
AVX2 (256-bit)	3	2.795	12.006	0.09
Sequential	4	20.673	1.623	1.00
`memmem`	4	22.866	1.467	1.11
SSE2 (128-bit)	4	6.257	5.362	0.30
AVX2 (256-bit)	4	3.991	8.408	0.19
Sequential	8	21.320	1.574	1.00
`memmem`	8	22.955	1.462	1.08
SSE2 (128-bit)	8	10.549	3.181	0.49
AVX2 (256-bit)	8	6.5275	5.140	0.31
Sequential	15	25.950	1.293	1.00
`memmem`	15	22.911	1.465	0.88
SSE2 (128-bit)	15	17.898	1.875	0.69
AVX2 (256-bit)	15	11.598	2.893	0.45
Sequential	16	24.339	1.379	1.00
`memmem`	16	22.900	1.465	0.94
SSE2 (128-bit)	16	19.084	1.758	0.78
AVX2 (256-bit)	16	11.999	2.796	0.49
Sequential	32	25.299	1.326	1.00
`memmem`	32	22.898	1.465	0.91
SSE2 (128-bit)	32	19.229	1.745	0.76
AVX2 (256-bit)	32	21.388	1.569	0.85
Sequential	33	65.112	0.515	1.00
`memmem`	33	22.897	1.465	0.35
SSE2 (128-bit)	33	19.159	1.751	0.29
AVX2 (256-bit)	33	21.298	1.575	0.33
Sequential	48	65.159	0.515	1.00
`memmem`	48	22.913	1.464	0.35
SSE2 (128-bit)	48	19.210	1.747	0.29
AVX2 (256-bit)	48	21.351	1.572	0.33

Table 3.2: Benchmark of the various find-byte-sequence implementations for different sequence lengths.

3.3. Vectorised depth calculation

The first version of our engine vectorised depth calculation instead of character classification. While we discarded this idea, since the classifier had better performance characteristics, the algorithm is interesting on its own.

The idea is to provide an interface that would allow to query document depth at any point in a JSON tree. The exact interface we want to fulfil is an algorithm going block-by-block that will allow us to answer whether the depth at a given spot in the block is greater or equal to some value (which we will call a *query*), and compute the depth at the end of the block. Sequentially, the algorithm is very simple – every occurrence of '`{`' or '`[`' (called *opening characters*) increases the depth by one, while every occurrence of '`}`' or '`]`' (called *closing characters*) decreases depth by one.

The user goes block by block and keeps the current accumulated depth. At the end of each block the depth change from the block is added to the accumulator. When depth at a given point needs to be known exactly, e.g. on a closing character, the user asks if the depth withing the block is greater or equal to some value. Vectorially, we have two approaches for providing such a structure – eager and lazy.

3.3.1. Eager implementation

The key observation is that depth at a given point is equal to the prefix sum of the input, where opening characters are interpreted as 1, closing characters as -1, and all other characters as 0. If we can calculate a prefix sum using SIMD, then we get a vector of depths.

SIMD processing of prefix sums has been studied and benchmarked in a recent paper [ZWR20]. Based on those results we chose the Horizontal SIMD algorithm, which scores best in throughput benchmarks despite not being work-efficient – it performs $\mathcal{O}(n \log n)$ SIMD operations for input length n. In step i it computes the prefix sums of all fragments of length 2^i of the input. Conceptually, we construct a binary tree over the data of logarithmic depth.

An issue arises with vectors comprised of many lanes (Section 2.4.2). Because of physical isolation shifts occur within lanes, so shifting by 128-bits always results in all-zeroes. To work around that, as the last level of the tree an AVX2 implementation needs to manually extract the sum of elements in the first lane, broadcast it over the second lane, and then add. For pseudocode see Figure 3.3.

3.3.2. Lazy implementation

In an eager implementation a block is loaded, the prefix sum computed, and then queries answered instantly. However, in a typical JSON document the depth does not change too sharply within a block – opening characters and closing characters are not too densely packed – making locations where a query is required few and far between. Therefore, instead of calculating the entire depth table up front, we can just produce bitmasks of where opening characters and closing characters are, and report the total depth change within the block. When we are asked for depth at a given point, we can calculate it from the masks with a simple shift and popcount.

As an additional heuristic, if we want to ask whether depth decreased by k within a block and we know that there are strictly less closing characters in the block than k, then we can immediately say no without calculating the actual depth. This makes the algorithm much more efficient, especially on real-life JSON documents.

```
let vector1 = _sub_epi8(closing_vector, opening_vector);
let vector2 = _add_epi8(vector1, _slli_si256(vector1, 1));
let vector4 = _add_epi8(vector2, _slli_si256(vector2, 2));
let vector8 = _add_epi8(vector4, _slli_si256(vector4, 4));
let vector16 = _add_epi8(vector8, _slli_si256(vector8, 8));

let halfway = _extract_epi8(vector16, 15);
let halfway_vector = _set1_epi8(halfway);
// SECOND_LANE is a constant mask zeroing the first lane.
let halfway_vector_second_lane_only = _and(halfway_vector, SECOND_LANE);

let vector32 = _add_epi8(vector16, halfway_vector_second_lane_only);

let array = [0; 32];
_storeu_si256(array.as_ptr(), vector32);
return array
```

Figure 3.3: Eager depth calculation for 256-bit SIMD with 128-bit lanes.

3.3.3. Performance

The data used in the experiment is the `wikidata-combined.json` totalling 71.42 MB, which is also the principal dataset for our main benchmarks (see Chapter 6 for more details). The implementations process this data, accumulating the depth and querying whether it is greater or equal to 5 at every step. We compared the sequential version to eager SSE2 and AVX2 implementations, as well as lazy SSE2, AVX2, and AVX512 implementations.

Lazy implementations consistently perform better than all others, with larger vector sizes providing reliable performance gains. We conclude that for our specific version of the streaming prefix sum problem, where all values are in the set $\{-1, 0, 1\}$ and most are 0, the lazy SIMD approach is the most efficient.

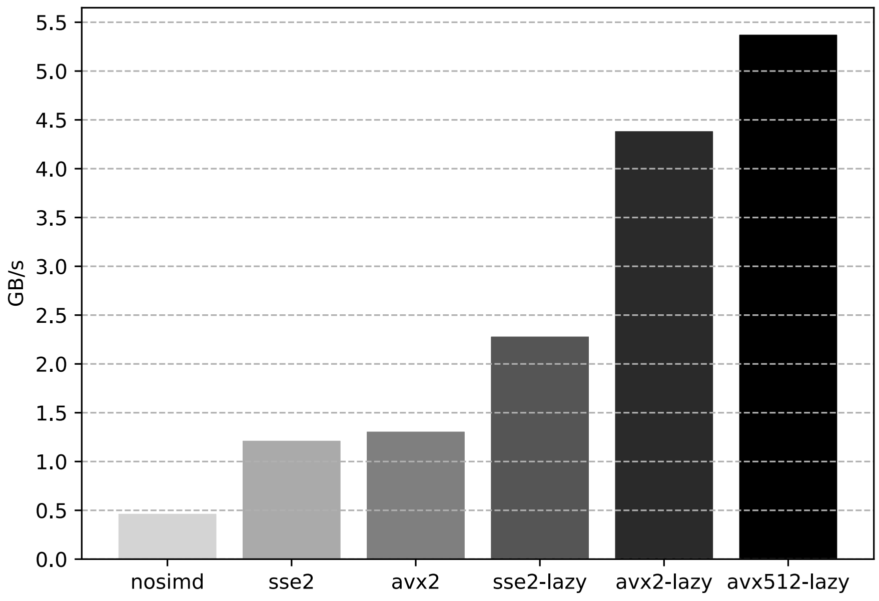


Figure 3.4: Throughput plot of Table 3.3.

Implementation	**Mean time (ms)**	**Throughput (GB/s)**	**Ratio**
Sequential	151.470	0.472	1.00
SSE2	58.535	1.220	0.39
AVX2	54.372	1.314	0.36
Lazy SSE2	31.223	2.287	0.21
Lazy AVX2	16.264	4.391	0.11
Lazy AVX512	13.275	5.380	0.09

Table 3.3: Benchmark of the depth calculator implementations.

Chapter 4

Vectorised classifier

The core of our algorithm is a SIMD pipeline that quickly recognises JSON structural tokens (see Section 1.3). This vastly decreases the amount of data that the main loop needs to process, discarding all irrelevant fragments like whitespace and atomic values. Its speed depends largely on the `shuffle` operation (see Section 2.4.4) and a specialised lookup table.

The problem we need to solve is marking braces and colons with SIMD. This is not straightforward, as those characters can be located within strings. In other words, characters between pairs of matching double quotes need to be ignored. This poses another issue, as not all double quotes delimit strings – some of them may be escaped with backslashes. For example:

- JSON string `"x{}[]:"` contains brackets and a colon that are not considered structural;
- JSON string `"x\""` contains a single escaped double quote;
- JSON string `"x\\"` contains a single escaped backslash; none of the double quotes are escaped, demonstrating that it is not sufficient to just ignore double quotes preceded by backslashes.

We present a general way of creating efficient SIMD lookup tables for the `shuffle` instruction that can be of use for quick 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` [LL19].

4.1. Structural lookup table

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 4.1 (Classification). *Fix a classification function* $f : \{\texttt{0x00}, \texttt{0x01}, \dots, \texttt{0xff}\} \to \{0, 1, \dots, k-1\}$. *Given a vector* a *of* n *bytes compute the vector* $b = [f(a_0), f(a_1), \dots, f(a_{n-1})]_8$.

As usual, a sequential solution is trivial. However, we claim that for binary classification ($k = 2$) it can also be efficiently solved using few SIMD instructions, assuming some constant vectors are precomputed. First we define auxiliary terms. Assume $k = 2$ and fix the classification function f. We identify each byte with a pair of an upper and lower nibble, e.g. `0x3a` is isomorphic to $\langle 3, a \rangle$. Define $\mathit{Nib} := \{0, 1, \dots, e, f\}$.

Definition 4.1 (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 $\mathit{low}(u) : \mathit{Nib} \to \mathcal{P}(\mathit{Nib})$ as a function assigning to each nibble u its acceptance set.

Definition 4.2 (Accepted Group). *An* accepted group *is defined by a set of upper nibbles that have the same acceptance sets. Formally, consider the equivalence relation $\equiv$ given by the kernel of low and let $\mathit{Nib}_{/\equiv}$ be its quotient set. Then the set G of all accepted groups is defined as:*

$$G = \{\langle [u]_{\equiv}, f(u) \rangle \mid u \in \mathit{Nib}\}.$$

It is easy to see that it contains exactly $|\mathit{Nib}_{/\equiv}|$ groups.

Definition 4.3 (Overlapping groups). *Two groups $\langle U_1, l_1 \rangle, \langle U_2, l_2 \rangle \in G$ are* overlapping *if $U_1 \neq U_2$ and $l_1 \cap l_2 \neq \emptyset$.*

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

$$\begin{aligned}
\mathit{low}(a) &= \{1, 2\},\\
\mathit{low}(b) &= \{1, 2\},\\
\mathit{low}(c) &= \{2\},\\
G &= \{\langle \{a, b\}, \{1, 2\} \rangle, \langle \{c\}, \{2\} \rangle\},
\end{aligned}$$

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. Note that $|G| \leq 16$, since there are only 16 possible nibbles.

4.1.1. 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 ordering of groups in G, $\langle u_1, l_1 \rangle, \ldots, \langle u_{|G|}, l_{|G|} \rangle$. Then construct an upper table as a vector *utab* such that $\mathit{utab}[x] = i$ if $x \in u_i$, and $\mathit{utab}[x] = 0$ otherwise. Analogously construct a lower table *ltab*, $\mathit{ltab}[x] = i \iff x \in l_i$, and $\mathit{ltab}[x] = 255$ otherwise. Naturally, $|G| < 255$. Then the required classification vector b is obtained as:

```
let upper_source = shiftright_epi8(source, 4);
let ltab_lookup = shuffle_epi8(ltab, source);
let utab_lookup = shuffle_epi8(utab, upper_source);
let b = cmpeq_epi8(ltab_lookup, utab_lookup);
```

There is one issue with this – x86 SIMD does not have a `shiftright_epi8` instruction. In fact, it has no 8-bit shift instructions. The exact reasons for that are unknown. However, it can be simulated with two instructions – first, 16-bit right shift by 4, then zero the upper nibbles with a precomputed mask[1]. Both these instructions have latency 1. The total cost of

[1]Zeroing the upper nibbles is important, because the exact semantics of `shuffle` make bits there meaningful. We omit these details as completely irrelevant – for all our purposes we want the upper nibbles to be zero.

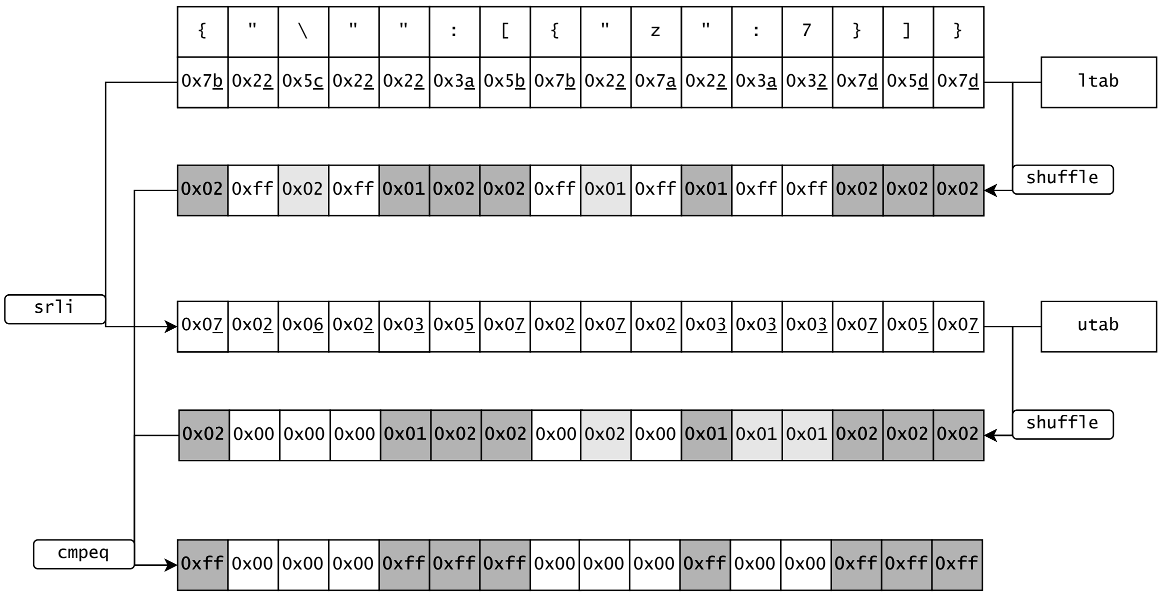

{ " \ " " : [{ " z " : 7 }] }
0x7b 0x22 0x5c 0x22 0x22 0x3a 0x5b 0x7b 0x22 0x7a 0x22 0x3a 0x32 0x7d 0x5d 0x7d
ltab
shuffle
0x02 0xff 0x02 0xff 0x01 0x02 0x02 0xff 0x01 0xff 0x01 0xff 0xff 0x02 0x02 0x02
srli
0x07 0x02 0x06 0x02 0x03 0x05 0x07 0x02 0x07 0x02 0x03 0x03 0x03 0x07 0x05 0x07
utab
shuffle
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 4.1: JSON document classified using the structural classifier's `ltab` and `utab` lookup tables.

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 the one sufficient for our JSON structural classifier. The structural characters are:

- '`{`' and '`}`' – `0x7b`, `0x7d`,
- '`[`' and '`]`' – `0x5b`, `0x5d`,
- '`:`' – `0x3a`,
- '`,`' – `0x2c`.

We ignore commas in our implementation[2], so the groups are:

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

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

$$\begin{aligned} utab = [&\texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x02}, \texttt{0x00}, \texttt{0x01}, \texttt{0x00}, \texttt{0x01}, \\ &\texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}] \\ ltab = [&\texttt{0xff}, \texttt{0xff}, \texttt{0xff}, \texttt{0xff}, \texttt{0xff}, \texttt{0xff}, \texttt{0xff}, \texttt{0xff}, \\ &\texttt{0xff}, \texttt{0xff}, \texttt{0x01}, \texttt{0x02}, \texttt{0xff}, \texttt{0x02}, \texttt{0xff}, \texttt{0xff}] \end{aligned}$$

[2]Although including them would not introduce an overlapping group, so extending the classifier with commas is straightforward.

4.1.2. 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 make the lower nibble lookup set the bits at indices of groups for which they are part of the acceptance set.

More precisely, we take an arbitrary ordering $\langle u_1, l_1 \rangle, \ldots, \langle u_{|G|}, l_{|G|} \rangle$. We construct the *utab* such that $utab[x] = 2^8 - 1 - 2^{i-1}$ if $x \in u_i$, and $utab[x] = 0$ otherwise. Then the lower table is defined as:

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

It should be clear from this that if we take a byte $b = \langle u, l \rangle$ such that $f(b) = 1$ then the bitwise OR of $utab[u]$ and $ltab[l]$ is $2^8 - 1$. Indeed, there must be a group $\langle u_i, l_i \rangle$ such that $u \in u_i$ and $l \in l_i$, and then $utab[u] = 2^8 - 1 - 2^{i-1}$. Then $ltab[l]$ must have the bit $i - 1$ lit by definition. For an example of this, see Figure 4.2.

The classification vector b is obtained with just one more operation than in the non-overlapping case:

```
let upper_source = srli_epi8(source, 4);
let ltab_lookup = shuffle_epi8(ltab, source);
let utab_lookup = shuffle_epi8(utab, upper_source);
let lookup = or(ltab_lookup, utab_lookup);
let b = cmpeq_epi8(lookup, ALL_ONES);
```

This increases the expected time to five CPU cycles.

4.1.3. General case

Unfortunately, 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 upper_source = srli_epi8(source, 4);
let ltab1_lookup = shuffle_epi8(ltab1, source);
let utab1_lookup = shuffle_epi8(utab1, upper_source);
let lookup1 = or(ltab1_lookup, utab1_lookup);
let ltab2_lookup = shuffle_epi8(ltab2, source);
let utab2_lookup = shuffle_epi8(utab2, upper_source);
let lookup2 = or(ltab2_lookup, utab2_lookup);
let final_lookup = or(lookup1, lookup2);
let b = cmpeq_epi8(final_lookup, ALL_ONES);
```

Assuming maximal local parallelism this takes six CPU cycles, since the two lookups are independent.

4.2. Handling escapes

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`.

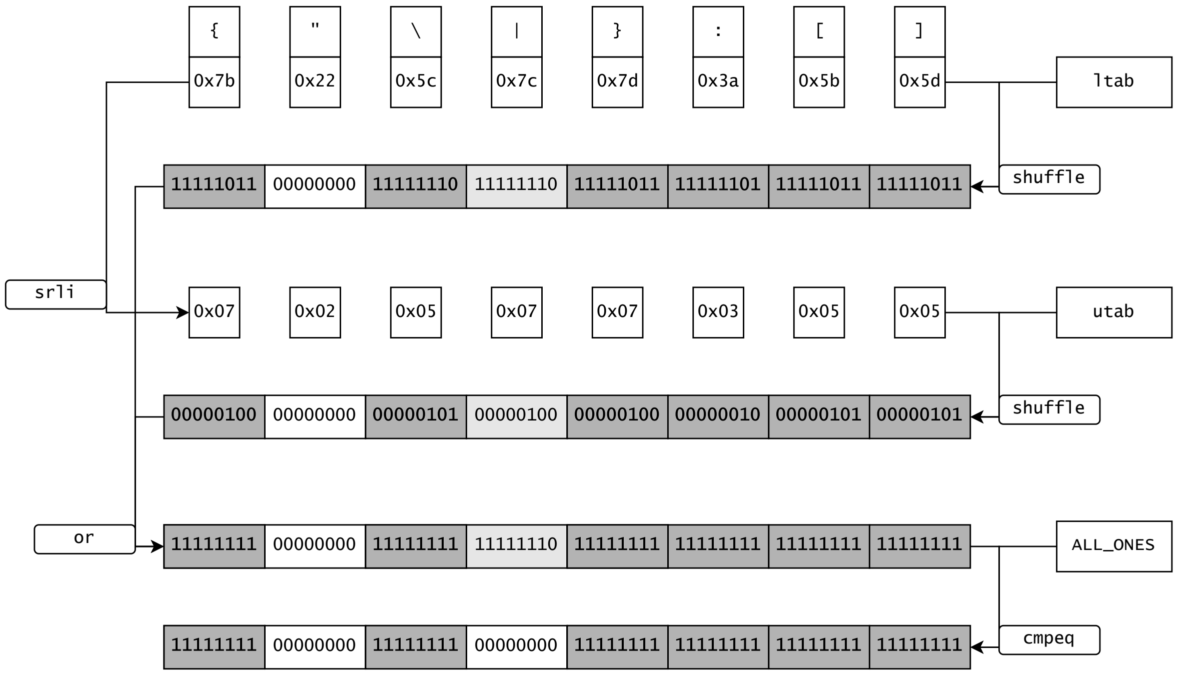


Figure 4.2: Classifying the bytes `0x3a`, `0x5b`, `0x5c`, `0x5d`, `0x7b`, `0x7d`, which have overlapping groups $\langle\{5\},\{b,c,d\}\rangle,\langle\{7\},\{b,d\}\rangle$.

However, we need to also account for escape sequences. Recall that a double quote character is escaped if and only if it is preceded by a sequence of backslashes of odd length.

We use the same solution as simdjson [LL19]. First, mark all backslash and quote characters with a `cmpeq`. Now 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 between those occurring at odd positions in the vector and those occurring at even positions using a constant mask[3].

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*.

4.2.1. 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 in the middle of a sequence of backslashes, then we might get incorrect results. Consider a label `"x\\"`. If the block boundary happens to be between the backslashes so that the first backslash is at the end of the first block and the other at the start of the second block, then the quote they precede would be incorrectly classified as escaped.

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

`-------------------v-------------------------------v--------v---`	
`{ "x{}[]:": 42, "x\"": 17, "x\\": 37, "x\\\\": "\\\"{}[]:\\\"" }`	`input`
`0000000000000000001000000000011000000000111100001110000001110000`	`B`
`0000000000000000001000000000010000000000100000001000000001000000`	`Starts = B & !(B << 1)`
`1010101010101010101010101010101010101010101010101010101010101010`	`ODD`
`0000000000000000001000000000000000000000100000001000000000000000`	`OStart = Start & ODD`
`0000000000000000000100000000011000000000000010000001000001110000`	`OCarry = OStart + B`
`0000000000000000000100000000000000000000000010000001000000000000`	`EOS = OCarry & !B`
`0000000000000000000100000000000000000000000000000001000000000000`	`EEOS = EOS & EVEN`
`0101010101010101010101010101010101010101010101010101010101010101`	`EVEN`
`0000000000000000000000000000010000000000000000000000000001000000`	`EStart = Start & EVEN`
`0000000000000000001000000000000100000000111100001110000000001000`	`ECarry = EStart + B`
`0000000000000000000000000000000100000000000000000000000000001000`	`EES = ECarry & !B`
`0000000000000000000000000000000000000000000000000000000000001000`	`OEES = EES & ODD`
`0000000000000000000100000000000000000000000000000001000000001000`	`escaped = EEOS | OOES`

Figure 4.3: Finding escaped characters in an input stream. The *slashes* vector is renamed as B.

As described in Section 3.2, there are two ways of dealing with this – introducing pipeline state, or switching to overlapping windows. Overlapping windows would significantly degrade the classifier's performance, so we choose to carry a state. In this case it is conceptually a single bit of information – whether the previous block's last character was a backslash and not an end of a backslash sequence. This 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;
```

Figure 4.4: Applying the backslash mask from the previous block's state to accurately compute starts of escape sequences and escaped characters.

4.3. Recognising quoted sequences

Having recognised unescaped double quote characters we now want to exclude all structural characters that are quoted. We use a solution from [LL19], utilising the `clmul` instruction described in Section 2.4.3. 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. Thus, it suffices to load the vector into a SIMD vector, perform the `clmul` operation, and then extract the information back to a mask.

4.3.1. Block boundaries

The same issue as with backslashes also occurs for quotes. If the block boundary falls between a pair of 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. The solution is the same – we introduce a second bit of state that signifies 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 `Iterator` trait[4], yielding items of an algebraic sum type `Structural` with three variants:

- `Closing` – representing '`{`' or '`[`';
- `Colon` – 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.

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 contains branching when we compare the mask to zero, and then when we branch on the input character to return a proper `Structural` value.

4.4.1. Performance

The data used in our classifier experiments is the `wikidata-combined.json` totalling 71.42 MB, which is also the principal dataset for our main benchmarks (see Chapter 6 for more details), and a prettified version (154.19 MB). We benchmark the main classifier using AVX2 SIMD against a sequential implementation. The benchmark consumes the produced stream of structural characters to avoid the compiler optimising work away. Effectively, we measure exactly the time it takes to classify the input.

[4]`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.

The SIMD classifier gives over three times speedup on compressed data, and almost five times speedup on prettified JSON. The sequential classifier suffers on prettified data, since it cannot simply skip all the whitespace characters like the SIMD implementation can. It is important to note that the throughput values for this benchmark are a limit for the main result as well – surely, a query engine doing work on the structural stream cannot be faster than the stream itself.

Data	**Implementation**	**Mean time (ms)**	**Throughput (GB/s)**	**Ratio**
Compressed	Sequential	109.24	0.6538	1.00
Compressed	Main (AVX2 SIMD)	32.32	2.2096	0.30
Prettified	Sequential	252.74	0.6101	1.00
Prettified	Main (AVX2 SIMD)	48.82	3.1584	0.19

Table 4.1: Benchmark of the classifiers on compressed (71.42 MB) and prettified (154.19 MB) datasets.

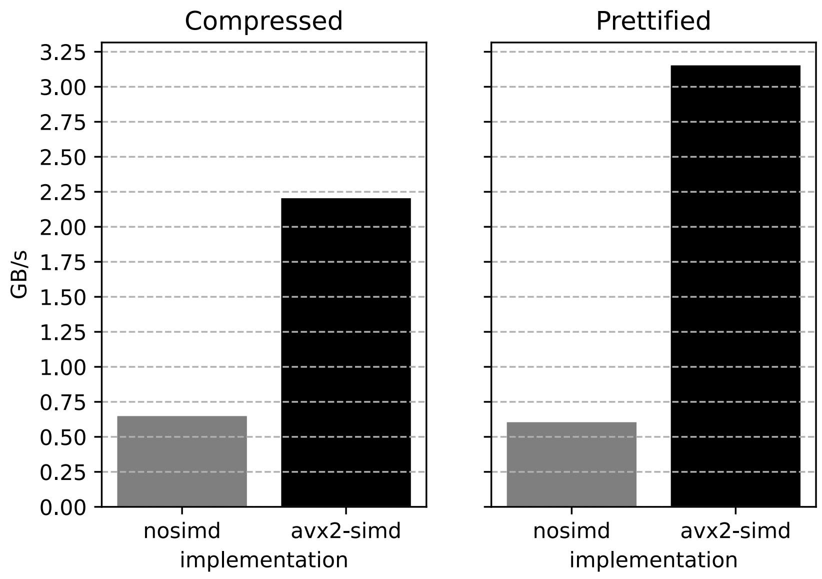


Figure 4.5: Throughput plot of Table 4.1

Chapter 5

Main query engine

The classifier is decoupled from the actual query engine, which allows us to measure the impact that the query logic itself has on performance. Our main implementation, based on [BMP21] tries to minimise branching, with the hypothesis being that such code should be much more performant with a SIMD structural classifier. To compare it, we have a reference implementation we call *recursive*, which is a run-of-the-mill engine based on mutually recursive functions. We focus our description on the main implementation.

5.1. Stackless processing

A useful theoretical framework for SIMD parallelism was provided in [MPP16], called *streaming circuits*. An algorithm that processes blocks of k-bits can be interpreted as a circuit over the input size k with an additional constant-size state. There is a direct link between the complexity of the circuit and effective parallelisation.

This gives motivation to construct algorithms that are effectively finite automata over the input stream of structural characters. In particular, parsing the input JSON tree is a no-go – such a parser would necessarily have to recognise the language of correct bracketings (the Dyck language), which is known to be TC^0-complete. For effective parallelisation we should be aiming at AC^0 circuits[1]. Since the Dyck language is in a sense complete for languages recognised by pushdown automata[2], it appears natural that using a stack for query processing would severely limit parallelism.

Querying streaming trees in a stackless manner was examined in [BMP21]. The authors characterise the kinds of queries that can be effectively executed on *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 the `Opening` structural character and decremented on every occurrence of the `Closing` structural character.

Unfortunately, it was proven that in general queries mixing descendant and child selectors cannot be implemented without the stack. Intuitively, the algorithm would have to remember every occurrence of the label that is recursively searched for to check its children. We first

[1]For a more in-depth analysis of the relation between SIMD and circuits consult Appendix B.

[2]It is easy to prove that every context-free language is a homomorphic image of Dyck.

consider queries with descendant selectors only, which can be effectively implemented, and then generalise to mixed selectors with what we call a *small stack model*.

5.2. Query automata

A query can be represented as a nondeterministic finite automaton running on a word comprised of the sequence of labels on a path from root to a node in the tree. Executing a query then boils down to simulating the runs of this automaton on all paths in the tree.

A pass over the stream of structural characters gives us enough information to simulate a *deterministic* automaton, as long as we use a stack – whenever an `Opening` character is encountered the state of the automaton is preserved on the stack, while each label triggers a transition; a `Closing` character pops the stack and restores the state to what it was before visiting the subtree. Theoretically, since the query gives us an NFA, the size of a DFA obtained from it could be exponential. However, if we examine the specific structure NFAs for our JSONPath queries have, it turns out that we can have an effective algorithm for finding a minimal deterministic automaton of the same size as the original NFA.

5.2.1. Descendant-only automaton

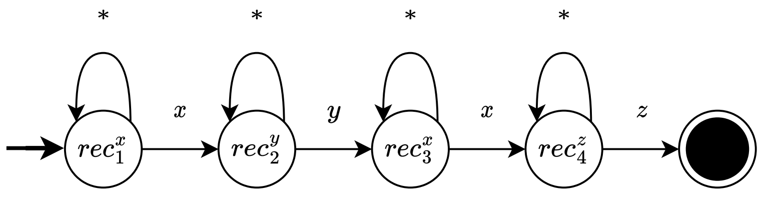


Figure 5.1: NFA accepting paths matching the query `$..x..y..x..z`.

For now, consider queries with descendant selectors only, as they can be implemented without using the stack at all. Let us consider a query $..$\mathtt{label_1}$..$\mathtt{label_2}$ . . . ..$\mathtt{label_n}$. In plain English, this query asks for a $\mathtt{label_n}$ node that is located in a subtree of a $\mathtt{label_{n-1}}$ node that is located in a subtree, *et cetera*, ultimately located in a subtree of a $\mathtt{label_1}$ node.

An NFA for such a query has a very regular form, as illustrated in Figure 5.1. There is a natural ordering of the states of the automaton based on the order of selectors in the JSONPath query. Assume the order is $q_1, \ldots, q_{n+1}$. Consider a DFA constructed from the NFA with the standard powerset construction. Then it is obvious that any state Q in the DFA is equivalent to $\{\max Q\}$. Therefore, the state of the DFA can be unambiguously represented with a single number from 1 to $n + 1$.

A stackless algorithm for this query uses n depth registers $\delta_1, \ldots, \delta_{n-1}$. We start in state 1 and accept in $n + 1$. When in state i, there are two transitions that can be taken:

- if current depth falls to δ_{i-1}, transition to i (not applicable to $i = 1$);
- if $\mathtt{label_i}$ is found, set δ_i to current depth and transition to $i + 1$ (not applicable to $i = n + 1$).

Correctness follows from a simple inductive argument.

Note that we crucially rely on node semantics of the query (defined in Section 1.2.1) – the key property is that once we enter a subtree of $\mathtt{label_i}$ we can ignore further nested $\mathtt{label_i}$ nodes, simply because any deeper subtree of $\mathtt{label_i}$ will be contained within the shallower subtree we encountered first. In path semantics (defined in Section 1.2.2) we would have to consider every such nested subtree individually, as they would provide different markings for the resulting path.

When looking at a query as an automaton, node semantics corresponds to us asking "is this path accepted", while path semantics would ask "how many different accepting runs this path induces". Only the former allows us to effectively determinise and minimise the NFA.

5.2.2. Allowing child selectors

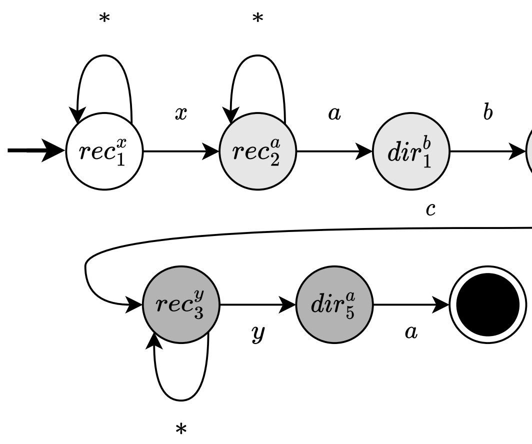


Figure 5.2: NFA accepting paths matching the query `$..x..a.b.a.b.c..y.a`. Segments coloured in different shades of grey become separate strongly connected components after minimisation.

Child selectors cause the automaton to effectively have two types of states – *recursive*, which correspond to descendant selectors in the query, and *direct*, which correspond to the child selectors. Again, we have a natural ordering on the states. The intuition about nested subtrees from the descendant-only case applies here in a more 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, minimisation of such an automaton causes no explosion of states – only transitions become more complicated. Every segment is translated to a strongly connected component in a minimised DFA. Crucially, this again allows us to represent the state as a single number from 1 to $n+1$ for a query of n child and descendant selectors.

Lemma 5.1. *For any query of n selectors the minimal DFA has exactly $n+1$ states and can be constructed in $\mathcal{O}(n^2)$.*

Proof (sketch). We proceed by induction on the number of descendant selectors in the query. If there are none then the claim is trivial, since clearly an NFA for such a query is already deterministic and minimal.

Assume the state i is the last recursive state (corresponding to the last descendant selector). Consider the DFA obtained with a standard powerset construction on the original NFA. It is clear that for any set of states Q where $\forall_{q \in Q}\, q \leq i$ is equivalent to $\{i\}$. Take the unique word of length $n-i$ that is accepted from i, $a_1 a_2 \ldots a_{n-i}$. By simple induction on its length we can show that the minimal DFA for the segment consisting of states i through $n+1$ will have $n-i+1$ states corresponding to $\{i\}$, $\Delta(\{i\}, a_1)$, $\Delta(\Delta(\{i\}, a_1), a_2)$, etc, where

$$\Delta(Q, a) = \{p \mid q \in Q, \langle q, a, p\rangle \in \delta\}$$

We conclude by saying that the DFA for states 1 through $i-1$ can be linked with the one for i through $n+1$ to obtain a minimal DFA for the whole query. $\square$

We note that the running time in Lemma 5.1 can be improved to $\mathcal{O}(n)$, since in essence the automaton represents the control table from the preprocessing step of the KMP algorithm, which can be constructed in linear time. This is irrelevant for our engine, as all real-life queries compile very fast, and we feel like the construction from our proof gives better insight into the nature of the query automata.

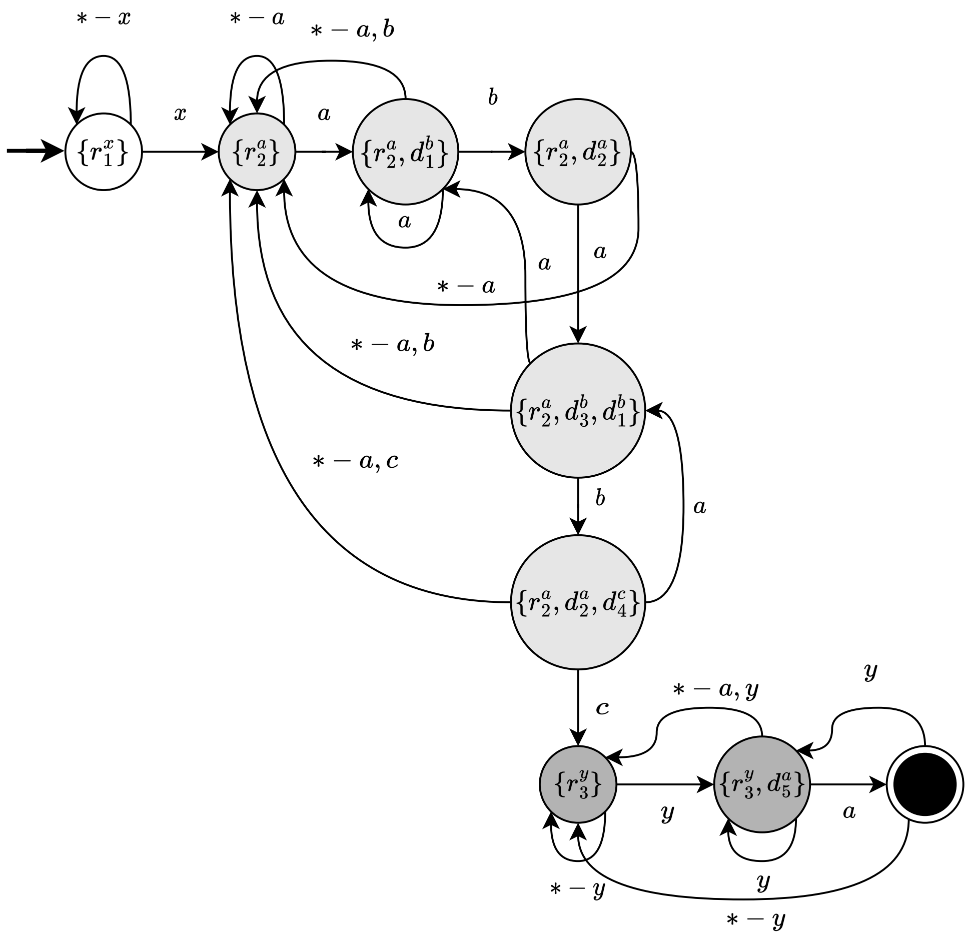


Figure 5.3: Automaton from Figure 5.2 after minimisation, clearly separated into three strongly connected components.

5.3. Small stack model

As noted before, it is impossible to run a query containing both descendant and child selectors in memory independent of the input document [BMP21]. Intuitively, this is due to the non-local nature of such a query – two children of the same node can be arbitrarily far away from each other in the input JSON string. In our approach, we try to get the best of both worlds by employing a *small stack*.

The stack alphabet consists of pairs of depth and state, called *stack frames*. The stack is small, in the sense that it only records when the simulated automaton's state changes –

all other information is redundant. It is easy to see that for a descendant-only query with n labels this will result in $\mathcal{O}(n)$ memory usage, and the at most n elements on the stack will correspond directly to the n registers of a register automaton.

For a query with child selectors the stack can grow up to the depth of the JSON tree. 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.

Implementation-wise, a special Rust structure is used – `SmallVec`. This puts our small stack on the actual stack of the executing thread as long as it is relatively shallow (up to 256 bytes). In the rare cases where it grows larger than that it is moved to the heap.

5.4. Full algorithm

The full algorithm has two phases. First, the query automaton is constructed and minimised. Then it is simulated in the small stack model on the stream of structural characters obtained from the classifier.

5.4.1. Automaton construction

The construction of a query NFA is straightforward, child selectors are translated into a direct state with one outgoing transition, while descendant selectors are translated into a recursive state, with an outgoing transition and a catch-all loop.

We then minimise the automaton step by step, constructing transition tables for each state. They will contain a number of transitions taken when a specific label is matched, and a single *fallback* transition taken when none of the labels matched. Then for every segment we simulate the NFA with a powerset construction. Starting with a set containing the initial recursive state, all possible transitions are examined to see how the states change, and an appropriate transition is created. We then follow the “forward” transition to the next set of states.

In the end, we will have transition tables constructed between sets of states from the original NFA. It remains to map every such set to a unique integer, creating the final minimised DFA.

5.4.2. Execution

We keep the current state, depth, and the small stack while iterating over the stream of structural characters and simulating the minimised DFA.

- If the character is `Closing`, depth decreases and we examine the stack. If the top stack frame’s depth is equal to the new depth, then the frame is popped and we transition to the state from the frame.
- If the character is `Colon`, we peek ahead in the stream, and then consider every transition from the current state. There are two cases:
  - the peeked character is `Opening` – we check if the label matches, and transition to the target state if it does, effectively recursing into the subtree; if the target is the accepting state, a match is reported;
  - the peeked character is something else – we check the label only if the transition target is the accepting state, and if that matches, we report a query match; there is no subtree to recurse into in this case.

- If the character is `Opening`, depth increases. If we have not followed any transition in the preceding colon, we follow the fallback transition; otherwise, we do nothing, since a transition was already triggered when we examined the label.

Additionally, every time we follow a transition to a different state we remember it on the stack by recording the depth and the state from which the transition was triggered.

Checking whether a label matches is done in the traditional manner – we compare the bytes in the input of expected length to the label. Additionally, we need to make sure that the label is enclosed in double quotes *and* that the opening double quote is not escaped. It is important to note that in a valid JSON document the colon structural character is not necessarily immediately preceded by the closing double quote of a label – there might be whitespace characters in between. In the worst case, we would have to backtrack from the colon through all the whitespace to the closing quote. However, finding real-world data with such pathological formatting is near impossible. The case is handled, but assuming sensibly formatted input will never be hit and will not impose a performance penalty.

5.5. Performance

We benchmark four solutions. There are two classifier implementations, SIMD and no SIMD, as well as two engine implementations: recursive, serving as the baseline, and main, based on a small stack.

The results are largely indicative of our intuitions. SIMD classifier provides large gains for both solutions, but the small stack model benefits more due to limited branching. Prettified documents benefit even more from SIMD processing, achieving performance closer to the classifier's.

The benchmark measures time that it takes for the engine to report all matches – the time to load the dataset into memory and compile the query is *not* measured. We note that query compilation times are negligible, in the order of a few microseconds. Only the count of matches is reported – this makes sure we measure only the main loop of the engine itself, without the noise that would be introduced from gathering large result sets in memory or reporting them to standard output.

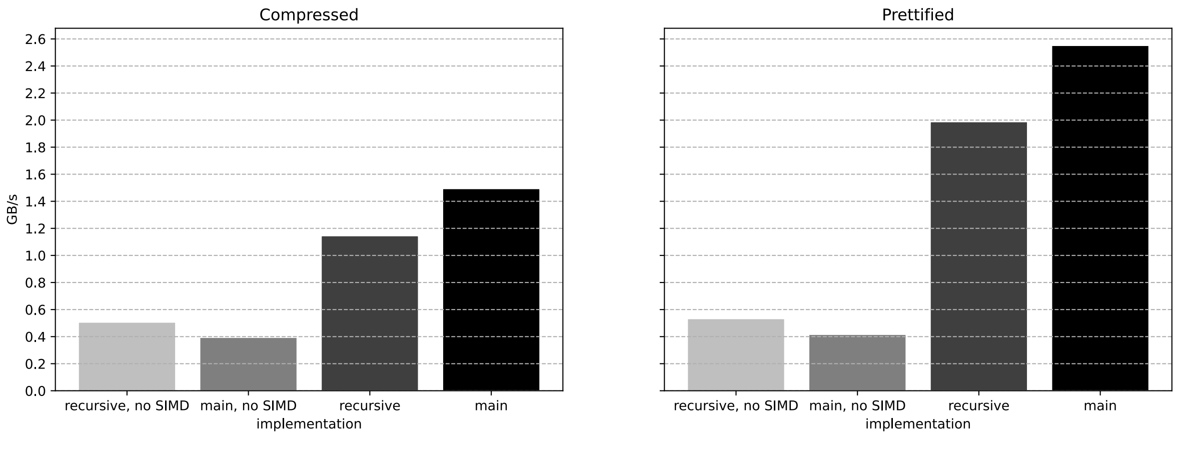


Figure 5.4: Throughput plot of Table 5.1.

Dataset	Implementation	Mean time (ms)	Throughput (GB/s)	Ratio
`combined`	Recursive, no SIMD	141.09	0.5062	1.00
`combined`	Main, no SIMD	181.51	0.3935	1.29
`combined`	Recursive	62.38	1.1449	0.44
`combined`	Main	47.83	1.4932	0.34
`prettified`	Recursive, no SIMD	289.69	0.5322	1.00
`prettified`	Main, no SIMD	370.80	0.4158	1.28
`prettified`	Recursive	77.58	1.9876	0.27
`prettified`	Main	60.44	2.5511	0.21

Table 5.1: Query `$..claims..references..hash` executed on the main dataset. The number of matches for this query is 57 117.

5.5.1. Analysis

Based on the data in Table 5.1, Table 5.2, and Table 5.3, as well as dataset characteristics described in Section 6.1, we make the following observations.

- The main solution with SIMD vastly outperforms other implementations on all data.
- SIMD solutions' performance is closely correlated with label density in the document. Indeed, largest throughput is obtained on the `prettified` dataset, with lowest label density, while the lowest throughput is on the densest dataset, `professions`. This is best visualised in Figure 5.5. This corresponds to our intuitions, as our label comparisons are strictly sequential and bound to take the bulk of processing time, as well as inhibit SIMD pipelining.
- Unsurprisingly, prettified datasets benefit most from SIMD processing, due to the ability to skip all irrelevant whitespace quickly.
- There do not appear to be large differences between execution times of queries with descendant selectors versus queries with child selectors. Small difference visible in Figure 5.6 can be explained by the difference in number of results and measurement noise.

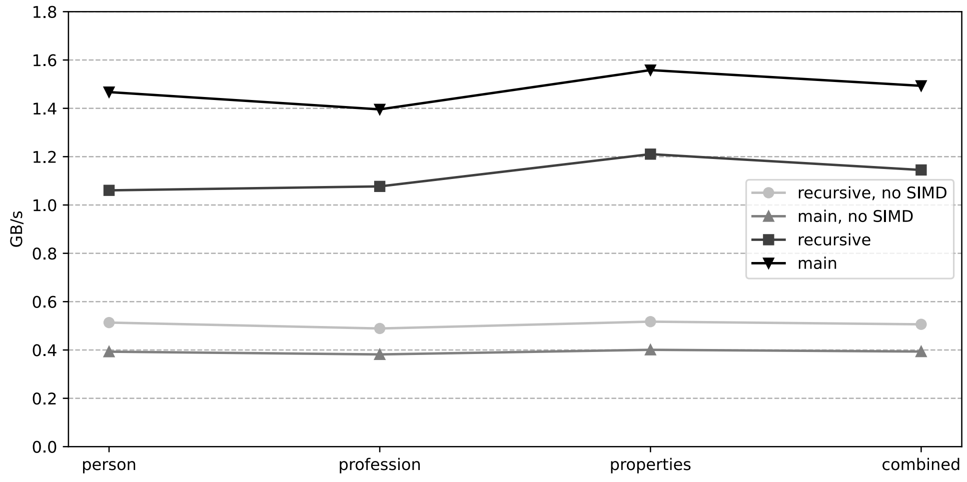


Figure 5.5: Throughput plot of Table 5.2.

5.5.2. Existing implementations

Lack of comparison with other existing implementations might be worrying at first. As shown in Appendix C, only a few implementations have the correct semantics. Out of those only two are sensible competitiors – `json-glib`, written in C, and `jsurfer`, written in Java[3]. The glib version is not a good comparison, because it builds the DOM, putting it at an immediate disadvantage. We note that a rough benchmark on the `combined` dataset puts its throughput at around 30 MB/s, orders of magnitude slower than `rsonpath`. Even when allowed to parse the DOM outside of measurements and tested only for the time it takes to execute the query on a tree in memory, the throughput reaches upwards of 220 MB/s[4].

The `jsurfer` is a valid target for comparison. However, we were unable to produce a reliable benchmark environment that could run both Rust and Java code in due time. We note only that a rough benchmark shows the Java solution to achieve a throughput of around 200MB/s[5].

These are much less rigorous benchmark than our other experiments, therefore we do not make sweeping statements based on it. It is clear, however, that our main solution is significantly faster.

Dataset	Implementation	Mean time (ms)	Throughput (GB/s)	Ratio
`person`	Recursive, no SIMD	38.857	0.5134	1.00
`person`	Main, no SIMD	48.151	0.3930	1.31
`person`	Recursive	17.840	1.0606	0.48
`person`	Main	12.940	1.4622	0.35
`profession`	Recursive, no SIMD	68.103	0.4890	1.00
`profession`	Main, no SIMD	87.256	0.3817	1.28
`profession`	Recursive	30.922	1.0770	0.45
`profession`	Main	23.860	1.3957	0.35
`properties`	Recursive, no SIMD	37.109	0.5173	1.00
`properties`	Main, no SIMD	47.901	0.4007	1.29
`properties`	Recursive	15.857	1.2105	0.43
`properties`	Main	12.318	1.5583	0.33

Table 5.2: Execution results on individual datasets constituting `combined`, all of them compressed. For `person` and `profession` the query is `$..claims..references..hash`, while for `properties` it is `$..qualifiers..datavalue..id`. Number of results in each dataset is 37 736, 14 142, and 18 219, respectively.

[3]The bash and PHP versions are extremely slow, and Scala's `jsonpath` appears to be abandoned.

[4]Measured on the Lille chifflot cluster on the `combined` dataset with the same query as our solutions. Our code parses the tree and compiles the query outside of measurement. The benchmark measures the execution time of the `json_path_match` method by running it 100 times and taking the execution time divided by 100 as the sample.

[5]Measured on the Lille chifflot cluster on the `combined` dataset with the same query as our solutions. Our code compiles the query and configures a listener that increments a result count integer variable. The benchmark measures the execution time of the `surf` method by running it 100 times and taking the execution time divided by 100 as the sample.

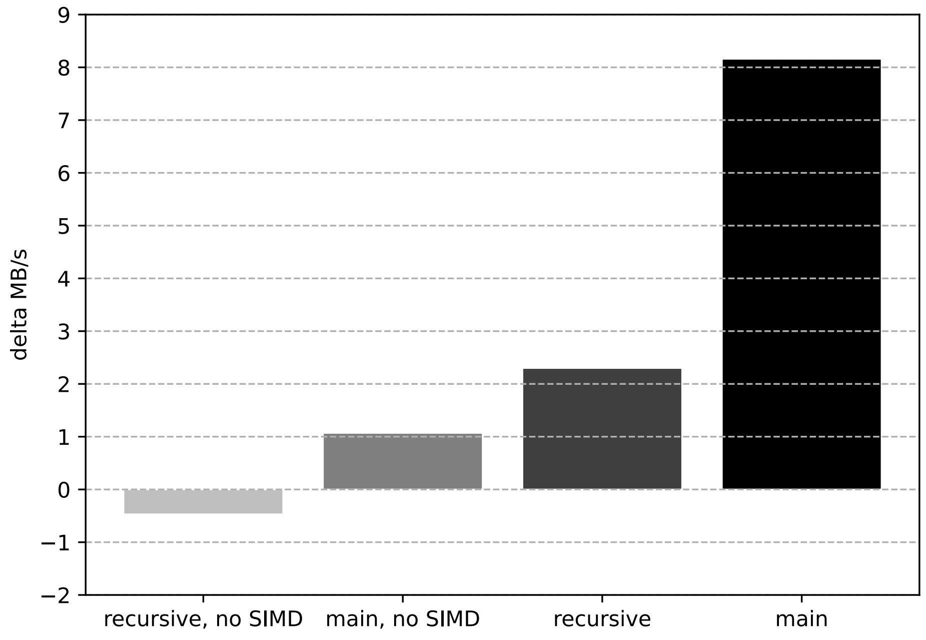


Figure 5.6: Plot of difference in throughput between the recursive and direct queries from Table 5.3. The recursive query performs negligibly worse, although the difference is statistically insignificant in all cases.

Query	**Implementation**	**Mean (ms)**	**Throughput (GB/s)**	**Ratio**
`$..en..value`	Recursive, no SIMD	36.071	0.5246	1.00
`$..en..value`	Main, no SIMD	46.966	0.4029	1.30
`$..en..value`	Recursive	16.209	1.1673	0.45
`$..en..value`	Main	12.245	1.5452	0.34
`$..en.value`	Recursive, no SIMD	36.039	0.5250	1.00
`$..en.value`	Main, no SIMD	47.091	1.1650	1.31
`$..en.value`	Recursive	16.241	0.4018	0.45
`$..en.value`	Main	12.310	1.5371	0.34

Table 5.3: Two queries differing in selector choice executed on the `person` dataset. The number of matches for the recursive query is 2 360, while for the direct-child query it is 1 753.

Chapter 6

Benchmark methodology

All benchmarks shown in this paper are performed on a stable server environment, using Rust Criterion [Hei17] as the benchmarking harness, on publicly available JSON documents describing Wikipedia pages.

6.1. Datasets

We used the open-source database dumps of Wikidata in a JSON format [Wik22]. The three core datasets are `person`, `profession`, and `properties`, which are subdocuments of the main JSON dump. These three documents were combined into a single large document, which provides the main benchmark – `combined`.

The datasets come in two types – compressed and prettified. Compressed are the JSON documents with all meaningless whitespace characters removed, resulting in a single line of condensed characters. Prettified data is the opposite, being multiline and indented with four spaces per level to be human-readable. These represent the two common formats in which JSON data is generally stored and passed around. We focus on compressed data, as it is the harder task for our engine, thus we consider only one non-compressed `prettified`, which is structurally the same as `combined`.

Relevant characteristics of the datasets can be found in Table 6.1.

Dataset	**Size (MB)**	**Labels**	**Labels/MB**	**Max depth**	**Average depth**
`person`	18.92	870 769	46 024	13	8.15
`profession`	33.30	1 574 975	47 297	13	6.35
`properties`	19.19	822 899	42 882	13	7.23
`combined`	71.42	3 268 644	45 766	15	9.05
`prettified`	154.19	3 268 644	21 199	15	9.05

Table 6.1: Metadata of utilised datasets. Note that the data in `wikidata_combined` is artificially nested within a root object with a single member, whose value is a list containing the three nested documents. Therefore, it contains one more label than the sum of the three nested documents, and an max and average depth inflated by 2.

6.2. Machines

Our benchmark results are ran on Grid 5000 [Bol+06] – a French nation-wide grid of compute platforms for experiment-driven research in computer science. The hardware available in the grid is listed in [Gro18]. For all our benches we used the *chifflot* cluster in the Lille site, whose specification is listed in Table 6.2. The site was chosen due to its CPUs built on Skylake, which is the first architecture with AVX512 support, and high single-core performance.

Category	**Description**
Kernel	Linux 5.10.0-15-amd64 x86_64
Architecture	x86_64, Skylake
CPUs	2× Intel Xeon Gold 6126, 12 cores, 24 threads 2.6 GHz (3.7 GHz Boost)
RAM	192 GiB
Rust	1.61.0 stable

Table 6.2: Specification of Lille chifflot cluster machines.

6.3. Tooling

We use the Criterion Rust benchmark framework, based on a framework by the same name for Haskell. Criterion handles the harness code and performs statistical analysis on the results.

A benchmark is executed as follows. First, a warm-up phase is performed, making sure 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.

The mean time of execution and throughput calculated based on that are generally the most important statistics. Note that our methodology specifically and deliberately targets bandwidth only – latency is not an issue for this paper.

6.4. Reproducibility

The benchmark code and all data sets are available as part of the project's GitHub repository [Gie22b]. Up-to-date information on executing the benchmarks is also available. Benchmarks for results given in Chapter 3 are located in a separate package, `simd-benchmarks`. Benchmarks for main results – the classifier and the engine – are in the main `rsonpath` package.

Summary

We have presented a working JSONPath query engine for a subset of available query selectors based on a SIMD structural classifier, as well as a theoretical framework based on finite automata for designing JSONPath query execution engines. Our novel small stack model allows us to efficiently execute such automata with minimal branching and stack usage, allowing more efficient SIMD pipelining. The general algorithm for constructing SIMD lookup tables presented in Section 4.1 is of potential use for designing similar vectorised classifiers for different domains.

The solution is open-sourced and is not a mere prototype – it is well-engineered and ready for extensions. The benchmark harness we have constructed should allow us to further improve the performance of the engine and extend it with new selectors, with relevant experiments to back that up.

There is much room for extension in `rsonpath`. Firstly, we would like to support more selectors. Wildcard selectors, matching any label, appear to be a straightforward addition. Accessing elements of lists at given indices should also be simple. A hard problem to crack are filters, which add non-locality to the queries, making them impossible to simulate in a linear pass through a stream. One would need a theoretical classification of feasible filtering queries that could be integrated into the engine.

We were unable to provide rigorous benchmarks against other existing JSONPath query engines. It is clear from preliminary runs that our main implementation is faster, but one would need to integrate solutions in potentially wildly different programming languages into our benchmark harness to produce reliable results. We believe this to be the only missing piece between turning this thesis into a publication at a systems conference.

Finally, the related circuit complexity conjecture described in Appendix B remains an open problem. Solving it would shed light on the nature of SIMD parallelisation and characterise the gap between AC^0 and TC^0 circuit classes.

Appendix A

aligners crate

Keeping alignment of bytes that our SIMD algorithms run on is crucial, as described in Section 2.5. Unfortunately, aligned and misaligned byte slices are indistinguishable during static analysis. There is no syntactical tool to assert that a slice is aligned, there is no check that would make sure that taking a subslice does not break alignment.

This is especially frustrating when decoupling modules – there is no way to express syntactically that a given public function accepts only aligned bytes. Moreover, there is no cross-platform way of checking whether a pointer is aligned to a k-byte boundary. A naive way of casting the pointer to an integer and checking its remainder modulo k might work on most conventional architectures, but there is no guarantee on that. The Rust standard library exposes a function checking alignment of a pointer, `pointer::align_offset`, but it is explicitly stated that one cannot depend on its output for correctness, as it can return a sentinel "unknown" value if it cannot check the alignment [Rus19].

To solve this problem we devised a library of types to control alignment statically. This crate is available on the Rust community library repository, crates.io [Gie22a]. Such crates existed before in the Rust ecosystem, but they were unsatisfactory for our purposes. We assume basic knowledge of the Rust programming language in the following description.

A.1. Supported alignments

The information about alignment is provided by implementers of the `Alignment` trait. It has a very short definition:

```rust
pub unsafe trait Alignment {
    fn size() -> usize;
}
```

It is unsafe, because the `size` function must satisfy two invariants:

1. its return value must be a power of 2;
2. it must be pure, i.e. its result must always be the same every time it is called.

Condition 1. is required to make alignment sound – heap allocators do not support non-power-of-two alignments. Condition 2. must be upheld for the guarantee to make sense – "this pointer is aligned to `A::size()`" would not be well-defined if `A::size()` could return different values between calls.

The crate provides four alignment types supported out-of-the-box:

- `TwoTo<const N: u32>` – alignment to 2^N. This type can obviously cover all valid alignments under the conditions mentioned above.
- `SimdBlock` – guaranteed to be the alignment of a single SIMD block under the target architecture, decided at compile time. For example, for AVX2 this is 32-byte alignment, while for AVX512 it is 64-byte.
- `TwoSimdBlocks` – guaranteed to be exactly twice the alignment of `SimdBlock`. This is useful for processing blocks in pairs, either to implement overlapping windows, or to make algorithms more efficient, for example by performing operations on two AVX2 vectors and packing them into a single `u64` instead of two `u32`s.
- `Page` – alignment to the OS memory page size, for better cache performance.

For purposes of this paper, `Page` alignment is useful to align the entire input on page boundary, while `SimdBlock` and `TwoSimdBlocks` are essential for statically enforcing that calls to SIMD load instructions are sound.

As syntactic sugar some aliases are defined, like `Eight` and `SixtyFour` equivalent to `TwoTo<3>` and `TwoTo<6>`, respectively.

A.2. Asserting alignment on a type level

The core type provided by the crate is `AlignedBytes<A: Alignment>`, which is a heap-allocated byte array guaranteed to be aligned w.r.t. `A`.

Taking a reference to the bytes gives a `&AlignedSlice<A>` (or `&mut AlignedSlice<A>`), which can be treated the same as `&[u8]` (or `&mut [u8]`) thanks to Rust's `Deref` trait (or `DerefMut`). Most importantly, one can flow the alignment information through the type system in two ways.

1. Using `offset(&self, count: isize)`, which returns the slice offset by `count` blocks of size `A::size()`. The result is still aligned, so the result preserves this information.
2. Using `iter_blocks(&self)`, which returns an iterator over blocks of size `A::size()` of the bytes. Each of these blocks is naturally guaranteed to be aligned.

This allows algorithms to effectively run on a reference to `AlignedSlice<A>` by only ever considering aligned blocks of the input. For example, the algorithm shown on Figure 2.1 can now be safely implemented by taking in `&AlignedSlice<Eight>` and iterating over blocks of 8 bytes (see Figure A.1).

A.3. Comparison to existing solutions

Crates related to memory alignment already exist, but none of them were satisfactory for us.

By far the most popular crate for memory alignment, still actively maintained, is `aligned`. It provides a thin wrapper over any type and forces it to be aligned to the specified boundary. However, it provides only a handful of alignments (2, 4, 8, 16, 32, 64), and it has no facilities for safe slicing of aligned bytes or iterating through aligned blocks. Therefore, it does not allow us to express the contract of a function using SIMD requiring an aligned pointer to a block of k bytes, so it would be useless for us.

Other crates like `aligned-array`, `align-data` or `maligned` are either also lacklustre in their features, or no longer maintained.

```rust
fn simd_align(a: &AlignedSlice<Eight>, b: &AlignedSlice<Eight>) {
    const SIZE = 8;
    let a_blocks = a.iter_blocks();
    let b_blocks = b.iter_blocks();
    for (i, (a, b)) in a_blocks.zip(b_blocks).enumerate() {
        let a_vec: u64 = a.as_u64();
        let b_vec: u64 = b.as_u64();
        let xor = a_vec ^ b_vec;
        if xor != 0 {
            let idx = i * SIZE + (xor.trailing_zeros() / SIZE);
            return Some(idx);
        }
    }
    return None;
}
```

Figure A.1: Vectorised algorithm from Figure 2.1, but guaranteeing the assumptions described in Section 2.5 statically at compile time.

Appendix B

Circuit lower bound for Dyck

As introduced in [MPP16], an algorithm that processes blocks of n-bits can be interpreted as a circuit over the input size n with an additional constant-size state – a *streaming circuit*. There is a direct link between the complexity of the circuit and effective parallelisation. To justify our approach of circumventing parsing, we would want to show that the fundamental problem of matching bracket pairs is not achievable with simple circuits. It is natural that any parser for JSON, or indeed any tree-shaped data format, needs to contain a solution to this problem. Formally, we are trying to recognise the language of correct bracketings, called the *Dyck language*.

The Dyck language is known to be complete for the class TC^0 of polynomial-size, constant depth circuits with unbounded fan-in, equipped with majority (or equivalently threshold) gates [MC89]. It is clear that SIMD programs contain TC^0-complete operations, for example one can count the number of lit bits, which is clearly stronger than majority. This separates them from AC^0, as majority is not in AC^0 (by virtue of parity not being in AC^0). However, SIMD programs are limited by their length. Ideally, to model SIMD parallelism we want programs of constant lengths that use building blocks working on vectors of some size n. In other words, we want circuits of constant depth and a constant number of special operations that require involved instructions.

We conjecture that recognising the Dyck language cannot be expressed in a constant-size SIMD program – intuitively, it appears that a solution to Dyck must utilise linearly many majority gates. On the other hand, it is clear that SIMD programs are stronger than AC^0 circuits, as operations like counting the number of lit bits are also TC^0-complete. The nuance here is that SIMD programs cannot perform a non-constant number of such operations, so while they exceed the expressiveness of AC^0 it is unlikely that they fully capture TC^0. Even though SIMD programs might be TC^0-complete with respect to AC^0 reductions, this is irrelevant for practical purposes, as such reductions could blow-up the number of used operations from constant to polynomial. It remains an open question what class of circuits would capture all of SIMD operations. As a stepping stone between AC^0 and TC^0 we consider a class $\mathrm{AC}^0[\mathit{Maj}_{\mathcal{O}(1)}]$.

B.1. Classes with a bounded number of special gates

Consider any class of circuit families $\mathcal{C}$. Let G be any logic gate. For $k \in \mathbb{N}$ we denote as $\mathcal{C}[G_k]$ the class of all circuit families that follow the restrictions of the class $\mathcal{C}$ except that each circuit can also use at most k gates of type G of unbounded fan-in.

Additionally, for any function $f : \mathbb{N} \to \mathbb{N}$ we define $\mathcal{C}[G_{f(n)}]$ to be the class of circuit families from $\mathcal{C}$, such that each circuit C_n for inputs of size n can use at most $f(n)$ gates of type G of unbounded fan-in. As an additional shorthand define $\mathcal{C}[G_{\mathcal{O}(f(n))}]$ as the class $\bigcup_{c\in\mathbb{N}} \mathcal{C}[G_{c\cdot f(n)}]$.

B.2. Dyck, Prefix-Dyck

The Dyck language represents correct bracketings, which is a prefix property, i.e. a word is a correct bracketing if and only if all of its prefixes contain no more closing brackets than matching opening brackets. A natural valuation for a word is the difference between opening and closing bracket characters in it. We want that difference to be non-negative in all proper prefixes, and to be zero for the entire word. This is formalised below. Consider words over the alphabet $\Sigma = [1..k] \cup [\bar{1}..\bar{k}]$. A circuit with inputs over this alphabet can be modelled as receiving the characteristic function for each input character and each alphabet letter. In other words, every character is represented as $|\Sigma|$ bits, out of which exactly one is lit.

Definition B.1 (prefix Dyck value)**.** *For all words we define its* prefix Dyck value *for* $1 \leq i \leq k$, $\mathit{pdyck}_i : \Sigma^* \to \mathbb{N}$, *as:*

$$\mathit{pdyck}_i(w) = \#_i(w) - \#_{\bar{i}}(w).$$

Definition B.2 (PDyck_k language)**.** *The language PDyck_k (Prefix-Dyck) is defined inductively as:*

$$\mathit{PDyck}_k = \{\epsilon\} \cup \{wa \mid w \in \mathit{PDyck}_k, a \in \Sigma, \forall_{1\leq i\leq k}\, \mathit{pdyck}_i(wa) \geq 0\}.$$

Definition B.3 (Dyck_k language)**.** *The Dyck_k language is a subset of PDyck_k:*

$$\{w \in \mathit{PDyck}_k \mid \forall_{1\leq i\leq k}\, \mathit{pdyck}_i(w) = 0\}.$$

When we go through a JSON document we are actually interested in computing pdyck_2 for every input prefix. This is due to the nature of the streaming setting – we need to have this information for the portion of data we have read, and then we receive another block and process it using that information. It is obvious that if we could compute pdyck_k for each prefix then we would also be able to decide PDyck_k. It is also obvious that pdyck_{k+1} is harder to compute than pdyck_k. Moreover, PDyck_k is at least as hard as Dyck_k, by the following lemma.

Lemma B.1. *Recognising Dyck_k reduces to recognising PDyck_k with an $\mathrm{NC}^0[\mathit{Maj}_{2k}]$ conversion.*

Proof. Having a circuit that recognises PDyck_k for input of any length, it remains to connect, for all $i \in [1..k]$, all inputs representing i to a majority gate, its negated (with negation defined as $\lambda i.\bar{i}$) inputs to another one, and assert that neither has the majority. $\square$

B.3. Neutral letters

This notion corresponds more closely to our JSON query problem. When we want to match brackets in a JSON we do not get a pure stream of the structural brackets only, we are working on the entire document containing labels, values, colons, commas, etc., which are irrelevant to the bracketing problem. This means that we want to compute a variation of PDyck_2, where we collapse all irrelevant characters to a special e symbol that does not influence membership to the language in any way. We formalise this idea.

Definition B.4. *Any language L over an alphabet Σ can be extended with a* neutral letter e *into a language $neut(L, e)$ over $\Sigma \cup \{e\}$, with the following transformation:*

$$neut(L, e) = \{e^* a_1 e^* a_2 e^* \dots e^* a_n e^* \mid a_1 a_2 \dots a_n \in L\}$$

In conclusion, for JSON querying we are interested in computing $neut(PDyck_2, e)$. It is trivial to see that Lemma B.1 extends to a reduction of $neut(Dyck_k, e)$ to $neut(PDyck_k, e)$.

B.4. Open conjectures

The following conjecture is the end goal of our investigation into circuit complexity of $Dyck_k$:

Conjecture B.1. $\mathrm{AC}^0[Maj_{\mathcal{O}(1)}]$ *cannot recognise $neut(Dyck_1, e)$.*

This would in some sense prove that there does not exist an efficient SIMD-parallel algorithm for computing $Dyck_1$, which would justify why an efficient parser/query engine implementation for JSON cannot simply attempt to match the structural brackets.

A weaker version of this conjecture is also an open problem.

Conjecture B.2. *A circuit in $\mathrm{AC}^0[Maj_1]$ of depth 2 cannot recognise $neut(Dyck_1, e)$.*

We note that this is a very regular setting – all circuits here correspond to a majority of disjunctions, i.e. we take an OR of a number of inputs or their negations, and test whether the majority of such clauses are satisfied. Even this separation, however, remains elusive.

In the end, we believe that $Dyck_1$ itself is inexpressible in $\mathrm{AC}^0[Maj_1]$, which would trivially imply both of the above conjectures. However, proving separation for $neut(Dyck_1, e)$ should be easier than for $Dyck_1$.

Appendix C

JSONPath implementations – node and path semantics

Using the `json-path-comparison` project [Bur+19] we ran a comparison of existing implementations of JSONPath w.r.t. node and path semantics, as described in Section 1.2. 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 (see Figure 1.2 and Figure 1.3). 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 C.1: 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.

Bibliography

[Baw99] Alan Bawden. 'Quasiquotation in Lisp'. In: *PEPM*. 1999.

[BMP21] 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. Virtual Event, China: Association for Computing Machinery, 2021, pp. 109–125. ISBN: 9781450383813. DOI: `10.1145/3452021.3458320`. URL: `https://doi.org/10.1145/3452021.3458320`.

[Bol+06] Raphaël Bolze et al. 'Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed'. In: *International Journal of High Performance Computing Applications* 20.4 (2006), pp. 481–494. DOI: `10.1177/1094342006070078`. URL: `https://hal.inria.fr/hal-00684943`.

[Bra17] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. Tech. rep. 8259. Dec. 2017. 16 pp. DOI: `10.17487/RFC8259`. URL: `https://www.rfc-editor.org/info/rfc8259`.

[Bur+19] Christoph Burgmer et al. *json-path-comparison*. 2019. URL: `https://github.com/cburgmer/json-path-comparison/commit/c0a5122a7c6ae8923550e7208d6443be79bc94d0` (visited on 01/05/2022).

[CD99] James Clark and Steven DeRose. *XML Path Language (XPath) Version 1.0*. Recommendation. `https://www.w3.org/TR/1999/REC-xpath-19991116/`. Latest version available at `https://www.w3.org/TR/xpath/`. W3C, Nov. 1999.

[Gal15] Andrew Gallant. *`memchr`*. 2015. URL: `https://crates.io/crates/memchr`.

[Gie22a] Mateusz Gienieczko. *`aligners` crate*. 2022. URL: `https://crates.io/crates/aligners`.

[Gie22b] Mateusz Gienieczko. *`rsonpath`*. 2022. URL: `https://github.com/V0ldek/rsonpath`.

[GNB22] Stefan Gössner, Glyn Normington and Carsten Bormann. *JSONPath: Query expressions for JSON*. Internet-Draft draft-ietf-jsonpath-base-05. Work in Progress. Internet Engineering Task Force, Apr. 2022. 45 pp. URL: `https://datatracker.ietf.org/doc/html/draft-ietf-jsonpath-base-05`.

[Gös07] Stefan Gössner. *JsonPath*. 2007. URL: `https://goessner.net/articles/JsonPath/` (visited on 01/05/2022).

[Gro18] Groupement d'Intérêt Scientifique. *Grid'5000 hardware*. 2018. URL: `https://www.grid5000.fr/w/Hardware` (visited on 30/06/2022).

[GV04] Pierre Genevès and Jean-Yves Vion-Dury. 'XPath Formal Semantics and Beyond: a Coq based approach'. In: (Aug. 2004).

[Hei17] Brook Heisler. *`criterion-rs`*. 2017. URL: `https://crates.io/crates/criterion-rs`.

[Int22] Intel Corporation. *Intel Intrinsics Guide.* Version 3.6.2. 2022. URL: https://www.intel.com/content/www/us/en/docs/intrinsics-guide (visited on 29/05/2022).

[Lam75] Leslie Lamport. 'Multiple Byte Processing with Full-Word Instructions'. In: *Communications of the ACM* 18.8 (Aug. 1975), pp. 471–475. URL: https://www.microsoft.com/en-us/research/publication/multiple-byte-processing-full-word-instructions/.

[LL19] Geoff Langdale and Daniel Lemire. 'Parsing Gigabytes of JSON per Second'. In: *CoRR* abs/1902.08318 (2019). arXiv: 1902.08318. URL: http://arxiv.org/abs/1902.08318.

[Mai07] Geoffrey Mainland. 'Why It's Nice to Be Quoted: Quasiquoting for Haskell'. In: *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop.* Haskell '07. Freiburg, Germany: Association for Computing Machinery, 2007, pp. 73–82. ISBN: 9781595936745. DOI: 10.1145/1291201.1291211. URL: https://doi.org/10.1145/1291201.1291211.

[MC89] David A. Mix Barrington and James Corbett. 'On the relative complexity of some languages in NC1'. In: *Information Processing Letters* 32.5 (1989), pp. 251–256. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(89)90052-5. URL: https://www.sciencedirect.com/science/article/pii/0020019089900525.

[ML17] Wojciech Mula and Daniel Lemire. 'Faster Base64 Encoding and Decoding Using AVX2 Instructions'. In: *CoRR* abs/1704.00605 (2017). arXiv: 1704.00605. URL: http://arxiv.org/abs/1704.00605.

[MPP16] Filip Murlak, Charles Paperman and Michał Pilipczuk. 'Schema Validation via Streaming Circuits'. In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems.* PODS '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 237–249. ISBN: 9781450341912. DOI: 10.1145/2902251.2902299. URL: https://doi.org/10.1145/2902251.2902299.

[Pal+18] Shoumik Palkar et al. 'Filter Before You Parse: Faster Analytics on Raw Data with Sparser'. In: *Proc. VLDB Endow.* 11 (2018), pp. 1576–1589.

[Pap21] Charles Paperman. *jsonpath duplication result.* PostgreSQL mailing list exchange. 2021. URL: https://www.postgresql.org/message-id/YTDMTNTZcFACww2V%40paperman.name.

[RDS17] Jonathan Robie, Michael Dyck and Josh Spiegel. *XML Path Language (XPath) 3.1.* Recommendation. https://www.w3.org/TR/2017/REC-xpath-31-20170321/. Latest version available at https://www.w3.org/TR/xpath-31/. W3C, Mar. 2017.

[Rus15] Rust Project. *The Cargo Book – Build Scripts.* Version 0.62.0. 2015. URL: https://doc.rust-lang.org/cargo/reference/build-scripts.html#build-scripts (visited on 11/06/2022).

[Rus19] Rust Project. *Rust Language Documentation – `pointer::align_offset`.* 2019. URL: https://doc.rust-lang.org/std/primitive.pointer.html#method.align_offset (visited on 19/05/2022).

[Tol16] David Tolnay. *`quote` crate.* 2016. URL: https://crates.io/crates/quote.

[Tol17] David Tolnay. *`proc-macro2` crate*. 2017. URL: `https://crates.io/crates/proc-macro2`.

[Wik22] Wikimedia Foundation. *Wikidata Database*. 2022. URL: `https://www.wikidata.org/wiki/Wikidata:Database%5C_download%5C#JSON%5C_dumps%5C_(recommended)`.

[ZWR20] Wangda Zhang, Yanbin Wang and Kenneth Ross. 'Parallel Prefix Sum with SIMD'. In: *ADMS@VLDB*. 2020.