Abstract JavaScript Object Notation or JSON is a ubiquitous data exchange format on the Web. Ingesting JSON documents can become a performance bottleneck due to the sheer volume of data. We are thus motivated to make JSON parsing as fast as possible.

Despite the maturity of the problem of JSON parsing, we show that substantial speedups are possible. We present the first standard-compliant JSON parser to process gigabytes of data per second on a single core, using commodity processors. We can use a quarter or fewer instructions than a state-of-the-art reference parser like RapidJSON. Unlike other validating parsers, our software (simdjson) makes extensive use of Single Instruction, Multiple Data (SIMD) instructions. To ensure reproducibility, simdjson is freely available as open-source software under a liberal license.

1 Introduction

JavaScript Object Notation (JSON) is a text format used to represent data [4]. It is commonly used for browser-server communication on the Web. It is supported by many database systems such as MySQL, PostgreSQL, IBM DB2, SQL Server, Oracle, and data-science frameworks such as Pandas. Many document-oriented databases are centered around JSON such as CouchDB or RethinkDB.

The JSON syntax can be viewed as a restricted form of JavaScript, but it is used in many programming languages. JSON has four primitive types or atoms (string, number, Boolean, null) that can be embedded within composed types (arrays and objects). An object takes the form of a series of key-value pairs between braces, where keys are strings (e.g., \{ "name" : "Jack", "age" : 22 \}). An array is a list of comma-separated values between brackets (e.g., \[ 1, "abc", null \]). Composed types can contain primitive types or arbitrarily deeply nested composed types as values. See Fig. 1 for an example. The JSON specification defines six structural characters (`, {, [): they serve to delimit the locations and structure of objects and arrays.

To access the data contained in a JSON document from software, it is typical to transform the JSON text into a tree-like logical representation, akin to the right-hand-side of Fig. 1, an operation we call JSON parsing. We refer to each value, object and array as a node in the parsed tree. After parsing, the programmer can access each node in turn and navigate to its siblings or its children without need for complicated and error-prone string parsing.

Parsing large JSON documents is a common task. Palkar et al. state that big-data applications can spend 80–90% of their time parsing JSON documents [25]. Boncz et al. identified the acceleration of JSON parsing as a topic of interest for speeding up database processing [2].

JSON parsing implies error checking: arrays must start and end with a bracket, objects must start and end with a brace, objects must be made of comma-separated pairs of values (separated by a colon) where all keys are strings. Numbers must follow the specification and fit within a valid range. Outside of string values, only a few ASCII characters are allowed. Within string values, several characters (like ASCII line endings) must
be escaped. The JSON specification requires that documents use a unicode character encoding (UTF-8, UTF-16, or UTF-32), with UTF-8 being the default. Thus we must validate the character encoding of all strings. JSON parsing is therefore more onerous than merely locating nodes. Our contention is that a parser that accepts erroneous JSON is both dangerous—in that it will silently accept malformed JSON whether this has been generated accidentally or maliciously—and poorly specified—it is difficult to anticipate or widely agree on what the semantics of malformed JSON files should be.

To accelerate processing, we should use our processors as efficiently as possible. Commodity processors (Intel, AMD, ARM, POWER) support single-instruction-multiple-data (SIMD) instructions. These SIMD instructions operate on several words at once unlike regular instructions. For example, starting with the Haswell microarchitecture (2013), Intel and AMD processors support the AVX2 instruction set and 256-bit vector registers. Hence, on recent x64 processors, we can compare two strings of 32 characters in a single instruction. It is thus straightforward to use SIMD instructions to locate significant characters (e.g., ‘‘’, ‘’’) using few instructions. We refer to the application of SIMD instructions as vectorization. Vectorized software tends to use fewer instructions than conventional software. Everything else being equal, code that generates fewer instructions is faster.

A closely related concept to vectorization is branchless processing; whenever the processor must choose between two code paths (a branch), there is a risk of incurring several cycles of penalty due to a mispredicted branch on current pipelined processors. In our experience, SIMD instructions are most likely to be beneficial in a branchless setting.

To our knowledge, publicly available JSON validating parsers make little use of SIMD instructions. Due to its complexity, the full JSON parsing problem may not appear immediately amenable to vectorization.

One of our core results is that SIMD instructions combined with minimal branching can lead to new speed records for JSON parsing—often processing gigabytes of data per second on a single core. We present several specific performance-oriented strategies that are of general interest.

– We detect quoted strings, using solely arithmetic and logical operations and a fixed number of instructions per input bytes, while omitting escaped quotes (§3.1.1).
– We differentiate between sets of code-point values using vectorized classification thus avoiding the burden of doing \(N\) comparisons to recognize that a value is part of a set of size \(N\) (§3.1.2).
– We validate UTF-8 strings using solely SIMD instructions (§3.1.5).

2 Related Work

A common strategy to accelerate JSON parsing in the literature is to parse selectively. Alagiannis et al. [1] presented NoDB, an approach where one queries the JSON data without first loading it in the database. It relies in part on selective parsing of the input. Bonetta and Brantner use speculative just-in-time (JIT) compilation and selective data access to speed up JSON processing [3]. They find repeated constant structures and generate code targeting these structures.

Li et al. present their fast parser, Mison which can jump directly to a queried field without parsing intermediate content [17]. Mison uses SIMD instructions to
quickly identify some structural characters but otherwise works by processing bit-vectors in general purpose registers with branch-heavy loops. Mison does not attempt to validate documents; it assumes that documents are pure ASCII as opposed to unicode (UTF-8). We summarize the most relevant component of the Mison architecture as follows:

1. In a first step, the input document is compared against each of the structural characters (’[’, ’{’, ’]’, ’}’, ’:’, ’,’) as well as the backslash (’\’). Each comparison uses a SIMD instruction, comparing 32 pairs of bytes at a time. The comparisons are then converted into bitmaps where the bit value 1 indicate the presence of the corresponding structural character. Mison omits the structural characters related to arrays (’[’, ’]’) when they are unneeded. Mison only uses SIMD instructions during this first step; it also appears to be the only step that is essentially branch-free.

2. During a second step, Mison identifies the starting and ending point of each string in the document. It uses the quote and backslash bitmaps. For each quote character, Mison counts the number of preceding backslashes using a fast instruction (popcnt): quotes preceded by an odd number of backslashes are turned off and ignored.

3. During a third step, Mison identifies the string spans delimited by the quotes. It takes each word (e.g., 32 bits) from the bitmap produced during the second step. It iteratively turns pairs of quotes into a string mask (where a 1-bit indicates the content of a string); using a small number of arithmetic and logical operations during each iteration.

4. In a final step, Mison uses the string masks to turn off and ignore structural characters (e.g., {, }, :) contained inside strings. Mison stores all opening braces in a stack. It pops the stack with each new closing brace, starting from the left, thus finding pairs of matching braces. For each possible nesting depth, a bitmap indicating the location of the colons can be constructed by partially copying the input colon bitmap.

Starting from the colon locations extracted from the bitmaps, Mison can parse the keys by scanning backward and the values by scanning forward. It can select only the content at a given depth. In effect, the colon bitmap serves as an index to selectively parse the input document. In some instances, Mison can scan through JSON documents at a speed of over 2 GB/s for high selectivity queries on a 3.5 GHz Intel processor. It is faster than what is possible with a conventional validating parser like RapidJSON.

FishStore [29] parses JSON data and selects subsets of interest, storing the result in a fast key-value store [6]. While the original FishStore relied on Mison, the open-source version uses simdjson by default for fast parsing.

Pavlopoulou et al. [26] propose a parallelized JSON processor that supports advanced queries and rewrite rules. It avoids the need to first load the data.

Sparser filters quickly an unprocessed document to find mostly just the relevant information [25], and then relies on a parser. We could use simdjson with Sparser.

Systems based on selective parsing like Mison or Sparser might be beneficial when only a small subset of the data is of interest. However, if the data is accessed repeatedly, it might be preferable to load the data in a database engine using a standard parser. Non-validating parsers like Mison might be best with tightly integrated systems where invalid inputs are unlikely.

2.1 XML Parsing

Before JSON, there has been a lot of similar work done on parsing XML. Noga et al. [24] report that when fewer than 80% of the values need to be parsed, it is more economical to parse just the needed values. Marian et al. [19] propose to “project” XML documents, down to a smaller document before executing queries. Green et al. [14] show that we can parse XML quickly using a Deterministic Finite Automaton (DFA) where the states are computed lazily, during parsing. Farfán et al. [10] go further and skip entire sections of the XML document, using internal physical pointers. Takase et al. [28] accelerate XML parsing by avoiding syntactic analysis when subsets of text have been previously encountered. Kostoulas et al. designed a fast validating XML parser called Screamer: it achieves higher speed by reducing the number of distinct processing steps [15]. Cameron et al. show that we can parse XML faster using SIMD instructions [6], in their parser (called Parabix). Zhang et al. [31] show how we can parse XML documents in parallel by first indexing the document, and then separately parsing partitions of the document.

Mytkowicz et al. [22] show how to vectorize finite-state machines using SIMD instructions. They demonstrate good results with HTML tokenization, being more than twice as fast as a baseline.

2.2 CSV Parsing

Data also comes in the form of comma-separated values (CSV). Mühlbauer et al. optimize CSV parsing and

1 https://github.com/microsoft/FishStore
loading using SIMD instructions to locate delimiters and invalid characters [20]. Ge et al. use a two-pass approach where the first pass identifies the regions between delimiters while the second pass processes the records [12].

3 Parser Architecture and Implementation

In our experience, most JSON parsers proceed by top-down recursive descent [7] that makes a single pass through the input bytes, doing character-by-character decoding. We adopt a different strategy, using two distinct passes. We briefly describe the two stages before decoding. We adopt a different strategy, using two different approaches: one where the first pass identifies the regions between delimiters while the second pass processes the records [12].

1. In stage 1, we validate the character encoding and identify the starting location of all JSON nodes (e.g., numbers, strings, null, true, false, arrays, objects). We also need to locate the start of all strings (e.g., "[1,2]") defined in the JSON specification [4]. These locations are written as integer indexes in a separate array.

During this stage, it is necessary to distinguish the characters that are between quotes, and thus inside a string value, from other characters. For example, the JSON document "[1,2]" is a single string despite the appearance of brackets. That is, these brackets should not be identified as relevant structural characters. Because quotes can be escaped (e.g., "\") it is necessary to identify backslash characters as well. Outside of strings, only four specific white-space characters are allowed (space, tab, line feed, carriage return). Any other white-space character needs to be identified.

The first stage involves either SIMD processing over bytes or the manipulation of bitsets (arrays of bits) that have one bit corresponding to one byte of input. As such, it can be inefficient for some inputs—we can observe dozens of operations taking place to discover that there are in fact no odd-numbered sequences of backslashes or quotes in a given block of input. However, this inefficiency on such inputs is balanced by the fact that it costs no more to run this code over complex structured input, and the alternatives would generally involve running a number of unpredictable branches.

2. In stage 2, we process all of the nodes and structural characters. We distinguish the nodes based on their starting character. When a quote (""") is encountered, we parse a string; when a digit or a hyphen is found, we parse a number; when the letters 't', 'f', 'n' are found, we look for the values true, false and null.

Strings in JSON cannot contain some characters unescaped, i.e., ASCII characters with code points less than 0x20, and they may contain many sorts of escaped characters. It is thus necessary to normalize the strings: convert them to valid UTF-8 sequences. Encountered numbers must be converted to either integers or floating-point values. They can take many forms (e.g., 12, 3.1416, 1.2e+1). However, we must check many rules while parsing numbers. For example, the following strings are invalid numbers: 012, 1E+, and .1. The JSON specification is not specific regarding the range of numbers that we should accept: many parsers cannot represent integer values outside of the interval $[-2^{63}, 2^{63}]$. In contrast, we choose to accept all 64-bit integers in the interval $[-2^{63}, 2^{63}]$, while rejecting integers outside this interval. We also reject overly large floating-point numbers (e.g., 1e309 or -1e309).

We validate objects as sequences of strings, colons and values; we validate arrays as sequences of values separated by commas (','). We ensure that all objects started with an open brace ('{') are terminated with a closing brace ('}'). We ensure that all arrays started with an open square bracket ('[') are terminated with a closing square bracket (']'). The result is written in document order on a tape: an array of 64-bit words. The tape contains a word for each node value (string, number, true, false, null) and a word at the beginning and at the end of each object or array. To ensure fast navigation, the words on the tape corresponding to braces or brackets are annotated so that we can go from the word at the start of an object or array to the word at the end of the array without reading the content of the array or object. Specifically, the tape is constructed as follows.

- A null atom is represented as the 64-bit value ($'n'\times2^{56}$) where 'n' is the 8-bit code point values (in ASCII) corresponding to the letter ‘n’. A true atom is represented by $'t'\times2^{56}$, a false atom is given by $'f'\times2^{56}$.

- Numbers are represented using two 64-bit words. Integers are given by the word $'1'\times2^{56}$ followed by a 64-bit signed integer (in standard two’s complement form). Floating-point numbers are given by the word $'d'\times2^{56}$ followed by a 64-bit floating point value (in standard binary64).

- For an array, the first 64-bit tape element contains the value $'\times2^{56} + x$ where $x$ is one plus the index of the second 64-bit tape element on the tape. The second 64-bit tape element contains the value $'1'\times2^{56} + x$ where $x$ is the index of the first 64-bit tape element on the tape.
Our two-stage design is motivated by performance concerns. Stage 1 operates directly on the input bytes, processing the data in batches of 64 bytes. In this manner, we can make full use of the SIMD instructions that are key to our good performance. Stage 2 is easier because stage 1 identified the location of all atoms, objects and arrays: there is no need to finish parsing one atom to know the location of the next one. Thus when parsing \{'key1" : "val1", "key2" : "val2"\}', stage 2 receives the location of the tokens }, ',', ';', ',', ';', '}' corresponding to the start and end of the object, the two colons, and the beginning of each of the four strings. It can process the four strings without data dependence—we do not need to complete the parsing of the string "key1" to find the location of the column (:) before finding the location of the string "val1"! Except for unicode validation, we deliberately delay number and string validation to stage 2, as these tasks are comparatively expensive and difficult to perform unconditionally and cheaply over our entire input.

3.1 Stage 1: Structural and Pseudo-Structural Elements

The first stage of our processing must identify key points in our input: the structural characters of JSON (brace, bracket, colon and comma), the start and end of strings as delineated by double quote characters, other JSON atoms that are not distinguishable by simple characters (true, false, null and numbers), as well as discovering these characters and atoms in the presence of both quoting conventions and backslash escaping conventions.

In JSON, a first pass over the input can efficiently discover the significant characters that delineate syntactic elements (objects and arrays). Unfortunately, these characters may also appear between quotes, so we need to identify quotes. It is also necessary to identify the backslash character because JSON allows escaped characters: \"", \", \', \b', \f', \n', \r', \t', as well as escaped unicode characters (e.g. \uDD1E).

A point of reference is Mison [17], a fast parser in C++. Mison uses vector instructions to identify the colons, braces, quotes and backslashes. The detected quotes and backslashes are used to filter out the insignificant colons and braces. We follow the broad outline of the construction of a structural index as set forth in Mison; first, the discovery of odd-length sequences of colons, braces, quotes and backslashes. The detected structural characters (\{, \}, \[, \], \:\, \, \, \}, \, \, \, \} and numbers), as well as discovering these characters and atoms in the presence of both quoting conventions and backslash escaping conventions.

In JSON, a first pass over the input can efficiently discover the significant characters that delineate syntactic elements (objects and arrays). Unfortunately, these characters may also appear between quotes, so we need to identify quotes. It is also necessary to identify the backslash character because JSON allows escaped characters: \"", \", \', \b', \f', \n', \r', \t', as well as escaped unicode characters (e.g. \uDD1E).

A point of reference is Mison [17], a fast parser in C++. Mison uses vector instructions to identify the colons, braces, quotes and backslashes. The detected quotes and backslashes are used to filter out the insignificant colons and braces. We follow the broad outline of the construction of a structural index as set forth in Mison; first, the discovery of odd-length sequences of backslash characters—which will cause quote characters immediately following to be escaped and not serve their quoting role but instead be literal characters, second, the discovery of quote pairs—which cause structural characters within the quote pairs to also be merely...
literal characters and have no function as structural characters, then finally the discovery of structural characters not contained within the quote pairs. We depart from the Mison paper in method and overall design. The Mison authors loop over the results of their initial SIMD identification of characters, while we propose branchless sequences to accomplish similar tasks. For example, to locate escaped quote characters, they iterate over the repeated quote characters. Their Algorithm 1 identifies the location of the quoted characters by iterating through the unescaped quote characters. We have no such loops in our stage 1: it is essentially branchless, with a fixed cost per input bytes (except for character-encoding validation, §3.1.5). Furthermore, Mison’s vectorized processing is more limited by design as it does not identify the locations of the atoms, it does not process the white-space characters and it does not validate the character encoding.

3.1.1 Identification of the quoted substrings

Identifying escaped quotes is less trivial than it appears. While it is easy to recognize that the string “\” is made of an escaped quote since a quote character immediately preceded by a backslash, if a quote is preceded by an even number of backslashes (e.g., “\\”), then it is not escaped since \ is an escaped backslash. We distinguish sequences of backslash characters starting at an odd index location from sequences starting at even index location. A sequence of characters that starts at an odd (resp. even) index location and ends at an odd (resp. even) index location must have an even length, and it is therefore a sequence of escaped backslashes. Otherwise, the sequence contains an odd number of backslashes and any quote character following it must be considered escaped. We provide the code sequence with an example in Fig. 3 where two quote characters are escaped.

With the backslash and quote characters identified, we can locate the unescaped quote characters efficiently. We compute a shift followed by a bitwise ANDNOT, eliminating the escaped quote characters.

However, we are interested in finding the location between quotes (the strings), so we can find the actual structural characters. The desired bit pattern would be 1 if there are an odd-numbered number of unescaped quotes at or before our location and zero otherwise.

For example, given the word 0b100010000 representing quote locations (with 1-bit), we wish to compute 0b0111110000. We can achieve this result using the prefix sum of the XOR operation over our bit vector representing unescaped quotes. That is, the resulting bit value at index i is the XOR of all bit values up to and including the bit value at index i in the input. We can compute such a prefix sum in C++ with a loop that repeatedly apply the bitwise XOR on a left-shifted word: for (i=0;i<64;i++) mask = mask xor (mask << 1). This prefix sum can be more efficiently implemented as one instruction by using the carry-less multiplication (implemented with the pclmulqdq instruction) of our unescaped quote bit vector by another 64-bit word made entirely of ones. The carry-less multiplication works like the regular integer multiplication, but, as the name suggests, without a carry because it relies on the XOR operation instead of the addition. Let us use the convention that given a 64-bit integer a, a_i is the value of the i-th bit so that a = \sum_{i=0}^{63} a_i \cdot 2^i. The regular product between two 64-bit integers a, b is given by \sum_{i=0}^{63} a_i b_i^2 where a_i b_i is zero when a_i is zero, and otherwise it is b left righted by i bits. With these conventions, the carry-less product is given by \sum_{i=0}^{63} a_i b_i^2; that is, we replace the sum (\sum) by a series of XOR (symbolized by \oplus). Thus we see that when a_i = 1 for all indexes i, we get \sum_{i=0}^{63} b_i^2 which is the prefix sum of the XOR operation. The carry-less multiplication is broadly supported and fast on recent processors due to its applications in cryptography. On Skylake processors, the carry-less multiplication (pclmulqdq) has a latency of 7 cycles and one can be issued per cycle.

3.1.2 Vectorized Classification

Mison does one SIMD comparison per character (‘;’, ‘;’, ‘;’, ‘;’, ‘;’). We proceed similarly to identify the quotes and the backslash characters. However, there are six structural characters, and, for purposes of further analysis, we also need to discover the four permissible white-space characters. Doing ten comparisons and accompanying bitwise OR operations would be expensive. Instead of a comparison, we use the AVX2 vpshufb instruction to acts as a vectorized table lookup to do a vectorized classification. The vpshufb instruction uses the least significant 4 bits of each byte (low nibble) as an index into a 16-byte table. Other processor architectures (ARM and POWER) have similar SIMD instructions.

By doing one lookup, followed by a 4-bit right shift and a second lookup (using a different table), we can
To see how this can be used to identify sets of characters, suppose that we want to identify the byte values 0x9, 0xa, and 0xd. The low nibbles are 9, a, and d, and the high nibbles are all zeros. In the first 16-byte lookup table, we set the fourth least significant bit to 1 for the values corresponding to indexes 9, a, and d, and only for these values. In the second 16-byte lookup table, set the fourth least significant bit to 1 for the value at index 0, and only for this value. Then we have that whenever the input values 0x9, 0xa, and 0xd are encountered, and only for these values, the fourth least significant bit of the result of the bitwise AND is 1. Hence, using two `vpshufb` instructions, a shift and a few bitwise logical operations, we can identify a set of characters. If we could only identify one set of characters, this approach would not be necessarily advantageous, but we can identify many different sets with the same two `vpshufb` instructions. We can repeat the same strategy with new sets of input values, always making them match a given bit index (the fourth in our example). To avoid misclassifications, we need to ensure that each set of input values corresponding to a bit index is uniquely characterized by a set of low nibbles and high nibbles. The set `{0x9, 0xa, 0xd}` works since it is the set of all values with low nibbles 9, a, d, and high nibble 0. The set `{0x5b, 0x5d, 0x7b, 0x7d}` also works since it is the set of all values with the low nibbles b and d and the high nibbles 5 and 7. The set `{0x21, 0x33}` would not work since 0x31 and 0x23 also share the same high nibbles and low nibbles: we would need to break the set `{0x21, 0x33}` into two subsets ({{0x21}, {0x33}}). We can support as many sets as we have bit indexes (8).

We break the set of code-point values corresponding to structural characters into three sets: `{0x2c}, {0x3a}, {0x5b,0x5d, 0x7b, 0x7d}`. We match them to the first three bit indexes. We break the set of code-point values corresponding to white-space characters into two sets `{0x9, 0xa, 0xd}, and `{0x20}`. We match them to the fourth and fifth bit indexes. These sets are all uniquely characterized by their low and high nibbles. Our solution is not unique: e.g., we could have broken `{0x5b,0x5d, 0x7b, 0x7d}` into two sets ( {{0x5b,0x5d}, {0x7b, 0x7d}}).
Table 1: Table describing the vectorized classification of the code points. The first column and first row are indexes corresponding to the high and low nibbles. The second column and the second row are the looked up table values. The main table values are the bitwise AND result of the two table values (e.g., 10 AND 8 is 8). The omitted values are zeroes. On the right, we give the desired classification.

| low nibble | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|------------|---|---|---|---|---|---|---|---|
| high nibble| 0 | 8 | 1 | 0 | 2 | 17| 3 | 2 |
|            | 4 | 0 | 5 | 4 | 6 | 0 | 7 | 4 |
| code points|   |   |   |   |   |   |   |   |
| desired value| 1 | 2 | 4 | 8 | 0 | 16|

See Table 1. The table for the low nibbles is 16, 0, 0, 0, 0, 0, 0, 0, 0, 8, 10, 4, 1, 12, 0, 0; and the table for the high nibbles is 8, 0, 17, 2, 0, 4, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0. Applying our algorithm, we get the following results:

- For the comma (code-point value 0x2c), we get 1, as the bitwise AND between 17 and 1.
- For the colon (code-point value 0x3a), we get 0b10 (2 in binary).
- For the other structural characters ‘[’, ‘]’, ‘{’, ‘}’ (code-point values 0x5b,0x5d, 0x7b, 0x7d), we get 0b100 (4 in binary).
- For the first three white-space characters (with code-point values 0x9, 0xa and 0xd), we get the value 0b1000 (8 in binary).
- For the space character (code-point value 0x20), we get 0b10000 (16 in binary).
- All other character inputs will yield zero.

We can recognize the structural characters (‘,’ ‘;’ ‘[‘ ]’ ‘{‘ ‘}’ ‘
\r’ ‘\n’ ‘\t’ ‘ ’) and others. No branching is required.

### 3.1.3 Identification of White-Space and Pseudo-Structural Characters

We also make use of our ability to quickly detect white space in this early stage. We can use another bitset-based transformation to discover locations in our data that follow a structural character or quote followed by zero or more characters of white space; excluding locations within strings, and the structural characters we have already discovered, these locations are the only place that we can expect to see the starts of the JSON atoms (such as numbers whether or not starting with a minus sign, null, true, and false). These locations are thus treated as structural and we term them pseudo-structural characters. Formally, we define pseudo-structural characters as non-white-space characters that are

1. outside quotes and
2. have a predecessor that is either a white-space character or a structural character.

We use a feature of JSON: the legal atoms can all be distinguished from each other by their first character: ‘t’ for true, ‘f’ for false, ‘n’ for null and the character class [0-9-] for numerical values.

As a side-effect, identifying pseudo-structural characters helps validate documents. For example, only some ASCII white-space characters are allowed unescaped outside a quoted range in JSON. An isolated disallowed character would be flagged as a pseudo-structural character and subsequently rejected in stage 2. Furthermore, dangling atoms are automatically identified (as the a in [12 a]) and will be similarly rejected. The key insight is that stage 1 need not discover whether such out-of-place characters are legal JSON—it only needs to expose them in the stream of structural and pseudo-structural characters that will be parsed in stage 2.

Fig. 5 illustrates a final sequence in stage 1 where given the (unescaped) quotes and the quoted ranges, as well as the structural and white-space characters, we identify the pseudo-structural characters.

### 3.1.4 Index Extraction

During stage 1, we process blocks of 64 input bytes. The end product is a 64-bit bitset with the bits corresponding to a structural or pseudo-structural characters set
Our structural and pseudo-structural characters are relatively rare and can sometimes, but not always, be infrequent. E.g., we can construct plausible JSON inputs that have such a character once over 40 characters or once every 4 characters. As such, continuing to process the structural characters as bitsets involves manipulating data structures that are unpredictably spaced. We choose to transform these bitsets into indexes. That is, we seek a list of the locations of the 1-bits. Once we are done with the extraction of the indexes, we can discard the bitset. In contrast, Mison does not have such an extraction step and iterates directly over the 1-bits.

Our implementation involves a transformation of bitsets to indexes by use of the count trailing zeroes operation (via the tzcnt instruction) and an operation to clear the lowest set bit: \( s = s \land (s - 1) \) in C which compiles to a single instruction (blsr). This strategy introduces an unpredictable branch; unless there is a regular pattern in our bitsets, we would expect to have at least one branch miss for each word. However, we employ a technique whereby we extract 8 indexes from our bitset unconditionally, then ignore any indexes that were extracted excessively by means of overwriting those indexes with the next iteration of the index extraction loop. See Fig. 6. This means that as long as the frequency of our set bits is below 8 bits out of 64 we expect few unpredictable branches. The choice of the number 8 is a heuristic based on our experience with JSON documents; a larger unconditional extraction procedure would be more expensive due to having to use more operations, but even less likely to cause a branch miss as a wider range of bit densities could be handled by extracting, say, 8 indexes from our bitset.

\[
\text{we decode the set bits from 's' to array 'b'}
\]

```c
uint64_t s = ...;
uint32_t * b = ...;
// next line => popcnt instruction
uint32_t cnt = __builtin_popcount(s);
uint32_t next_base = b + cnt;
while (s) {
    // next line => tzcnt instruction
    *b++ = idx + trailingzeroes(s);
    // next line => blsr instruction
    s = s & (s - 1);
    *b++ = idx + trailingzeroes(s);
    s = s & (s - 1);
    *b++ = idx + trailingzeroes(s);
    s = s & (s - 1);
    *b++ = idx + trailingzeroes(s);
    s = s & (s - 1);
    *b++ = idx + trailingzeroes(s);
    s = s & (s - 1);
    *b++ = idx + trailingzeroes(s);
    s = s & (s - 1);
}
b = new_base;
```

Fig. 6: Code sequence to extract set bits out of a bitset

### 3.1.5 Character-Encoding Validation

In our experience, JSON documents are served using the unicode format UTF-8. Some programming languages like Java use UTF-16 for in-memory strings. Yet if we consider the popular JSON Java library Jackson, then the common way to serialize an object to JSON...
is to call the function `ObjectMapper.writeValue`, and the result is in UTF-8. Indeed, the JSON specification indicates that many implementations do not support encodings other than UTF-8. Parsers like Mison assume that the character encoding is ASCII [17]. Though it is reasonable, a safer assumption is that unicode (UTF-8) is used. Not all sequences of bytes are valid UTF-8 and thus a validating parser needs to ensure that the character encoding is correct. We assume that the incoming data is meant to follow UTF-8, and that the parser should produce UTF-8 strings.

UTF-8 is an ASCII superset. The ASCII characters can be represented using a single byte, as a numerical value called code point between 0 and 127 inclusively. That is, ASCII code points are an 8-bit integer with the most significant bit set to zero. UTF-8 extends these 128 code points to a total of 1,114,112 code points. Non-ASCII code points are represented using from two to four bytes, each with the most significant bit set to one. Non-ASCII code points cannot contain ASCII characters: we can therefore remove from an UTF-8 stream of bytes any number of ASCII characters without affecting its validation.

Outside of strings in JSON, all characters must be ASCII. Only the strings require potentially expensive validation. However, there may be many small strings in a document, so it is unclear whether vectorized unicode validation would be beneficial at the individual string level. Thus we validate the input bytes as a whole.

We first test if a block of 64 bytes is made entirely of ASCII characters. It suffices to verify that the most significant bit of all bytes is zero. This optimization might trigger some unpredictable branches, but given how frequently JSON documents might be almost entirely composed of ASCII characters, it is a necessary risk.

If there are non-ASCII characters, we apply a vectorized UTF-8 validation algorithm. We need to check that sequences of bytes are made of valid UTF-8 code points (see Tables 2). It involves several steps, but each one is efficient. We work exclusively with SIMD instructions.

- The byte values 0xC0 and 0xC1 are forbidden. When the byte value is 0xF0, the next byte value is larger than 0xA0. When the byte value is 0xF0, the next byte value is at least 0x90.
- When a byte value is outside the range of ASCII values, it belongs to one out of four classes, depending on the value of its high nibble:
  - If the high nibble is 8, 9, a or b (in hexadecimal) then the byte is the second, third of fourth byte in a code point.
  - If the high nibble is c or d then the byte must be the first of a sequence of two bytes forming a code point.
  - If the high nibble is e then the byte is the first out of a code point made of three bytes.
  - Finally, if the high nibble is f, then the byte is first in a sequence of four bytes.

We use the `vpshufb` instruction to quickly map bytes to one of these categories using values 0, 2, 3, and 4. We map ASCII characters to the value 1. If the value 4 is found (corresponding to a nibble value of f), it should be followed by three values 0. Given such a vector of integers, we can check that it matches a valid sequence of code points in the following manner. Shift values by 1 and subtract 1 using saturated subtraction, add the result to the original vector. Repeat the same process with a factor of two: shift values by 2 and subtract 2, add the result to the original vector. Starting with the sequence 4 0 0 0 2 0 1 1 3 0 0, you first get 4 3 0 0 2 1 1 1 3 2 0 and then 4 3 2 1 2 1 1 1 3 2 1. If the sequence came from valid UTF-8, all final values should be greater than zero, and be no larger than the original vector.

All these checks are done using SIMD registers solely, without branching. At the beginning of the processing, we initialize an error variable (as a 32-byte vector) with zeroes. We compute in-place the bitwise OR of the result of each check with our error variable. Should any check fail, the error variable will become non-zero. We only check at the end of the processing (once) that the variable is zero. If a diagnosis is required to determine where the error occurs, we can do a second pass over the input.

3.2 Stage 2: Building the Tape

In the final stage, we iterate through the indexes found in the first stage. To handle objects and arrays that can be nested, we use a goto-based state machine. Our state is recorded as a stack indicating whether we are in an array or an object, we append our new state to the stack whenever we encounter an embedded array.
Table 2: Unicode code points and their representation into sequences of up to four bytes under UTF-8.

| code points | 1st byte | 2nd byte | 3rd byte | 4th byte |
|-------------|----------|----------|----------|----------|
| 0x000000...0x00007F | 00...7F |           |          |          |
| 0x000080...0x00007FF | C2...DF | 80...BF  |          |          |
| 0x000080...0x0000FFFF | E0...A0 |          | 80...BF  |          |
| 0x000080...0x0007FF | E1...EC | 80...BF  |          |          |
| 0x000080...0x00007FF | ED       | 80...9F  |          |          |
| 0x000080...0x00007FF | EE...EF | 80...BF  |          |          |
| 0x000080...0x00007FF | F0       | 90...BF  |          |          |
| 0x000080...0x00007FF | F1...F3 | 80...BF  |          |          |
| 0x000080...0x00007FF | F4       | 80...8F  |          |          |

When encountering a quote character, we always read 32 bytes in a vector register, then look for the quote and the escape characters. If an escape character is found before the first quote character, we use a conventional code path to process the escaped character, otherwise we just write the 32-byte register to our string buffer. Some unicode characters may be represented using one or two 4-character hexadecimal code (e.g., \uD834\uDD1E); we convert them to valid UTF-8 bytes.

Our string buffer is made of a 32-bit integer indicating the length of the string followed by the string content in UTF-8. The JSON specification allows strings containing null characters, hence the need for a integer specifying the length of the string. As part of the string validation, we must check that no code-point value less than 0x20 is found: we use vectorized comparison.

3.2.1 Number Parsing

It is difficult to do number parsing and validation without proceeding in a standard character-by-character manner. We must check for all of the rules of the specification. Thus we proceed as do most parsers. However, we found it useful to test for the common case where there are at least eight digits as part of the fractional portion of the number. Given the eight characters interpreted as a 64-bit integer val, we can check whether it is made of eight digits with an expensive comparison:

\[
(((val \& 0xF0F0F0F0F0F0F0F0) \\
| (((val + 0x0606060606060606) \\
& 0x0F0F0F0F0F0F0F0F)) >> 4)) \\
== 0x3333333333333333)
\]

When this check is successful, we invoke a fast vectorized function to compute the equivalent integer value (see Fig. 7). This fast function begins by subtracting from all character values the code point of the character ‘0’, using the _mm_sub_epi8 intrinsic (using the _mm_add_epi16 intrinsic). Because the digits have consecutive code points in ASCII, this ensures that digit characters are mapped to their values: ‘0’ becomes 0, ‘1’ becomes 1 and so forth. We then invoke the _mm_madd_epi16 intrinsic (i.e., the _mm_add_epi16 intrinsic) to multiply every other digit by 10 and add the result to the previous digit, as a 16-bit sum. We repeat a similar process with the _mm_madd_epi16 intrinsic (i.e., the _mm_add_epi16 intrinsic), this time multiplying every other value by 100 and adding it to the previous value as a 32-bit sum.

The maximal value of these sums is 9999 which fits in a 16-bit integer. We apply the _mm_packus_epi32 intrinsic (i.e., the _mm_add_epi16 intrinsic) to pack the four 32-bit integers into four 16-bit integers. Finally, we call the _mm_madd_epi16 intrinsic again to multiply every other 16-bit value by 10000 and add it to the preceding value, generating a 32-bit sum. We only return one 32-bit value corresponding to 8 digits even though, technically, our function converts two series of eight digits to 32-bit integers. On a Skylake processor, the instructions have a latency of 1 cycle (psubb, _mm_add_epi16) or 5 cycles (pmaddubsw and pmaddwd). Because the pmaddwd instruction is needed twice, we can estimate that this function has a latency of at least 17 cycles, not counting store and load instructions. However, only about seven instructions are needed (with load/store instructions) which might be less than half what a naive approach might require. Indeed, merely loading eight digits may require eight distinct load instructions.

3.2.2 String Validation and Normalization

When encountering a quote character, we always read 32 bytes in a vector register, then look for the quote and the escape characters. If an escape character is found before the first quote character, we use a conventional code path to process the escaped character, otherwise we just write the 32-byte register to our string buffer. Some unicode characters may be represented using one or two 4-character hexadecimal code (e.g., \uD834\uDD1E); we convert them to valid UTF-8 bytes.

Our string buffer is made of a 32-bit integer indicating the length of the string followed by the string content in UTF-8. The JSON specification allows strings containing null characters, hence the need for a integer specifying the length of the string. As part of the string validation, we must check that no code-point value less than 0x20 is found: we use vectorized comparison.
4 Experiments

We validate our results through a set of reproducible experiments over varied data. In §4.3, we report that the running time during parsing is split evenly between our two stages. In §4.4, we show that we use half as many instructions during parsing as our best competitor. In §4.5, we show that this reduced instruction count translates into a comparable runtime advantage.

4.1 Hardware and Software

Most recent Intel processors are based on the Skylake microarchitecture. We also include a computer with the more recent Cannon Lake microarchitecture in our tests. We summarize the characteristics of our hardware platforms in Table 3. We verified the CPU characteristics experimentally with avx-turbo.

Our software was written using C++17. We tested it under several recent compilers (LLVM’s clang, GNU GCC, Microsoft Visual Studio). Our library is well tested and included in actual systems such as Yandex Clickhouse. We use the GNU GCC 9.1 compiler with the -O3 and -march=native flags under Linux. We compile the code as is, without profile-guided optimization. All code is single-threaded. We disable hyper-threading.

Our experiments assume that the JSON document is in memory; we omit disk and network accesses. Popular disks (e.g., NVMe) have a bandwidth of 3 GB/s and more. In practice, JSON documents are frequently ingested from the network. Yet current networking standards allow for speeds exceeding 10 GB/s and modern networking hardware can allow network data to be read directly into a cache line. A high performance implementation of JSON scanning is desirable even for data coming from a fast disk or from the network. While we focus on speed, we also expect that more efficient parsers reduce energy consumption.

After reviewing several parsers, we selected RapidJSON and sajson, two open-source C++ parsers, as references (see Table 4). Palkar et al. describe RapidJSON as the fastest traditional state-machine-based parser available. In practice, we find that another C++ parser, sajson, is faster. They are both mature and highly optimized: they were created in 2011 and 2012 respectively. The sajson parser can be used with either static or dynamic memory allocations; the static version is faster, so we adopt it.

Counting our own parser (simdjson), all three parsers can parse 64-bit floating-point numbers as well as integers. However, sajson only supports 32-bit integers whereas both RapidJSON and simdjson support 64-bit integers. RapidJSON represents overly large integers as 64-bit floating-point numbers (in a lossy manner) whereas both our parser (simdjson) and sajson reject documents with integers that they cannot exactly represent.

RapidJSON can either normalize strings in a new buffer or within the input bytes (insitu). We find that the parsing speed is greater in insitu mode, so we present these better numbers. In contrast, sajson only supports insitu parsing. Our own parser does not modify the input bytes: it has no insitu mode. All three parsers do UTF-8 validation of the input. However, the sajson parser does partial UTF-8 validation, accepting invalid code point sequences such as 0xb1 0x87. The number parsing accuracy of all three parsers is within one unit in the last place.

We consider other open-source parsers but we find that they are either slower than RapidJSON, or that

```c
uint32_t parse_eight_digits_unrolled(char * chars) {
  __m128i ascii0 = _mm_set1_epi8('0');
  __m128i mul_1_10 =
      _mm_setr_epi8(10, 1, 10, 1, 10, 1, 10, 1, 10, 1, 10, 1, 10, 1, 10, 1);
  __m128i mul_1_100 =
      _mm_setr_epi16(100, 1, 100, 1, 100, 1, 100, 1, 100, 1, 100, 1, 100, 1, 100, 1);
  __m128i mul_1_10000 =
      _mm_setr_epi16(10000, 1, 10000, 1, 10000, 1, 10000, 1, 10000, 1);
  __m128i in = _mm_loadu_si128((__m128i*)chars);
  __m128i t1 = _mm_maddubs_epi16(in, ascii0);
  __m128i t2 = _mm_madd_epi16(t1, mul_1_10);
  __m128i t3 = _mm_packus_epi32(t2, t2);
  __m128i t4 = _mm_madd_epi16(t3, mul_1_10000);
  return _mm_cvtsi128_si32(t4);
}
```

Fig. 7: Code sequence using Intel intrinsics to convert eight digits to their integer value.

Table 3: Hardware

| Processor       | Base Frequency | Max. Frequency | Microarchitecture | Memory          | Compiler   |
|-----------------|----------------|----------------|-------------------|-----------------|------------|
| Intel i7-6700   | 3.4 GHz        | 3.7 GHz        | Skylake (x86, 2015)| DDR4 (2133 MT/s) | GCC 9.1    |
| Intel i3-8121U  | 2.2 GHz        | 3.2 GHz        | Cannon Lake (x64, 2018)| LPDDR4 (3200 MT/s) | GCC 9.1    |

they failed to abide by the JSON specification (see §4.5). For example, parsers like gason, jsnu and ultrajson accept [0e+] as valid JSON. Parsers like fastjson and ultrajson accept unescaped line breaks in strings. Other parsers are tightly integrated into larger frameworks, making it difficult to benchmark them fairly. For methodological simplicity, we also do not consider parsers written in Java or other languages.

RapidJSON has compile-time options to enable optimized code paths making use of SIMD optimizations: these optimizations skip spaces between values or structural characters. However, we found both of these compile-time macros (RAPIDJSON_SSE2 and RAPIDJSON_SSE42) to be systematically detrimental to performance in our tests. Moreover, they are disabled by default in the library. Thus we do not make use of these optimizations.

Other than RapidJSON, we find that none of the libraries under consideration make deliberate use of SIMD instructions. However, we expect that all libraries benefit of SIMD instructions in our tests: many functions from the standard libraries are vectorized, and the compiler translates some conventional code to SIMD instructions (e.g., via autovectorization [23]).

We cannot directly compare with Mison since their software is not available publicly [17]. However, the authors of Mison reports speeds up to slightly over 2 GB/s on an Intel Xeon Broadwell-EP (E5-1620 v3) with a base frequency of 3.5 GHz: e.g., while parsing partially Twitter data. Mison does not attempt to validate the documents nor to parse them entirely but it also does some additional processing to answer a specific query.

4.2 Datasets

Parsing speed is necessarily dependent on the content of the JSON document. For a fair assessment, we chose a wide range of documents. See Table 5 for detailed statistics concerning the chosen files. In Table 6 we present the number of bytes of both the original document and the version without extraneous white-space characters outside strings.

From the author of RapidJSON [5], we acquired three data files. We have canada.json which is a description of the Canadian contour in GeoJSON: it contains many numbers. We have citm_catalog.json which is commonly used benchmark file. Finally, we have twitter.json which is the result of a search for the character one in Japanese and Chinese using the Twitter API: it contains many non-ASCII characters. From the author of sajson [6] we retrieved several more files: apache_builds.json, github_events.json, instruments.json, mesh.json, mesh.pretty.json, update-center.json.

We also generated number.json as a large array of random floating-point numbers. We also created twitterescaped.json which is a minified version of the twitter.json where all non-ASCII characters have been escaped.

Many of these documents require much number parsing or much string normalization. We deliberately did not consider small documents (smaller than 50 kB). The task of parsing many tiny documents is outside our scope.

4.3 Running Time Distribution

In Fig. 8 we present the distribution of cycles per stage, for each test file:

- The label 1: no utf8 refers to the time spent in stage 1, except for UTF-8 validation. This time is similar over all files, between 0.5 and 1 cycles per input byte. The time per input byte is higher for documents with many structural and pseudostructural characters per input byte such as canada, marine_ik, mesh and random (see Table 5). The difference can be explained by the cost of index extraction which is higher on a per byte basis in documents with many structural characters (see §3.1.4).
- The label 1: just utf8 refers to the time spent doing UTF-8 validation. It is negligible for all but the random and twitter documents. In the random file, UTF-8 validation is a significant cost. This file has a relatively high fraction of non-ASCII characters (20%). In comparison, the twitter file has only 3% of non-ASCII characters.
- The label 2: core is for the time spent in stage 2 except for string and number parsing. It is higher

[5] https://github.com/miloyip/nativejson-benchmark
[6] https://github.com/chadaustin/sajson/tree/master/testdata
Table 4: Competitive parsers

| Processor     | snapshot        | link                                      |
|---------------|-----------------|-------------------------------------------|
| simdjson      | June 12th 2019  | [https://github.com/lemire/simdjson](https://github.com/lemire/simdjson) |
| RapidJSON     | version 1.1.0   | [https://github.com/Tencent/rapidjson](https://github.com/Tencent/rapidjson) |
| sajson        | September 20th 2018 | [https://github.com/chadaustin/sajson](https://github.com/chadaustin/sajson) |

Table 5: Datasets statistics. The second last column (struct.) is the number of structural and pseudo-structural characters. The last column is the ratio of the total number of bytes over the number of structural and pseudo-structural characters.

| file            | integer | float | string | non-ascii | object | array | null | true | false | struct. | byte/struc. |
|-----------------|---------|-------|--------|-----------|--------|-------|------|------|-------|---------|-------------|
| apache_builds   | 2       | 0     | 5289   | 0         | 884    | 3     | 0    | 2    | 1     | 12365   | 10.3        |
| canada          | 46      | 111080| 12     | 0         | 4      | 56045 | 0    | 0    | 0     | 334374  | 6.7         |
| citm_catalog    | 14392   | 0     | 26604  | 348       | 10937  | 10541 | 1263 | 0    | 0     | 135991  | 12.7        |
| github_events   | 149     | 0     | 1891   | 4         | 180    | 19    | 24   | 57   | 7     | 4657    | 14.0        |
| geoc-2018       | 0       | 34128 | 0      | 3793      | 0      | 0     | 0    | 0    | 0     | 75842   | 43.9        |
| instruments     | 4935    | 0     | 6889   | 0         | 1012   | 194   | 431  | 17   | 109   | 27174   | 8.1         |
| marine_jk       | 130225  | 114950| 38268  | 0         | 9680   | 28377 | 0    | 6    | 0     | 643013  | 4.6         |
| mesh            | 40613   | 32400 | 11     | 0         | 3      | 3610  | 0    | 0    | 0     | 153275  | 4.7         |
| mesh.pretty     | 40613   | 32400 | 11     | 0         | 3      | 3610  | 0    | 0    | 0     | 153275  | 10.3        |
| numbers         | 0       | 10001 | 0      | 0         | 0      | 1     | 0    | 0    | 0     | 20004   | 7.5         |
| random          | 5002    | 0     | 33005  | 103482    | 4001   | 1001  | 0    | 495  | 505   | 88018   | 5.8         |
| twitterescaped  | 2108    | 1     | 18099  | 0         | 1264   | 1050  | 1946 | 345  | 2446  | 55264   | 10.1        |
| twitter         | 2108    | 1     | 18099  | 95406     | 1264   | 1050  | 1946 | 345  | 2446  | 55264   | 11.4        |
| update-center   | 0       | 0     | 27229  | 49        | 1896   | 1937  | 0    | 134  | 252   | 63420   | 8.4         |

Table 6: Datasets sizes: minified size omits white-space characters outside quotes.

| file            | bytes (minified) | bytes (original) | ratio |
|-----------------|------------------|------------------|-------|
| apache_builds   | 94653            | 127275           | 74%   |
| canada          | 2251027          | 2251027          | 100%  |
| citm_catalog    | 500299           | 1727204          | 29%   |
| github_events   | 53329            | 65132            | 82%   |
| geoc-2018       | 3073766          | 3327831          | 92%   |
| instruments     | 1083131          | 220346           | 49%   |
| marine_jk       | 1834197          | 2983466          | 61%   |
| mesh            | 650573           | 723597           | 90%   |
| mesh.pretty     | 753399           | 1577353          | 48%   |
| numbers         | 150121           | 150124           | 100%  |
| random          | 461466           | 510476           | 90%   |
| twitterescaped  | 562408           | 562408           | 100%  |
| twitter         | 466906           | 631514           | 74%   |
| update-center   | 533177           | 533178           | 100%  |

for documents marine_jk and mesh that have many structural and pseudostructural characters per input byte.

- The label 2: strings 2: numbers refers to the time spent in number parsing in stage 2. Roughly a third of the CPU cycles are spent parsing numbers in the files canada, marine_jk, mesh, mesh.pretty and numbers. In other files, the time spent parsing numbers is negligible.

- The label 2: strings refers to the time spent parsing strings in stage 2. The string parsing time is a sizeable cost in the twitterescaped file. In this file, all non-ASCII characters have been escaped which makes string normalization more difficult.

About half the CPU cycles per input byte (between 0.5 and 3 cycles) are spent in stage 1. Much of this stage relies on vector processing (with SIMD instructions). Thus at least half of the processing time is directly related to SIMD instructions and branchless processing.

The running time of the parser depends on the characteristics of the JSON document. It can be useful to model the performance: e.g., an engineer could predict the execution time and budget accordingly. For this purpose, we used linear regression based on our dataset.
of files. Our dataset is relatively small, but we expect that it is large enough for a simple linear regression.

- Let \( F \) the number of floating-point numbers,
- \( S \) be the number of structural and semi-structural elements and
- \( B \) the number of bytes.

With a high accuracy \((R^2 \geq 0.99)\), we have the following cost models:

- The stage 1 running time (in CPU cycles) is \( 1.7 \times S + 0.62 \times B \) on Skylake and \( 1.7 \times S + 0.59 \times B \) on Cannon Lake.
- The stage 2 running time is \( 19 \times F + 8.7 \times S + 0.31 \times B \) on Skylake and \( 17 \times F + 9.1 \times S + 0.29 \times B \) on Cannon Lake.
- The total running time is \( 19 \times F + 11 \times S + 0.92 \times B \) on Skylake and \( 17 \times F + 11 \times S + 0.88 \times B \) on Cannon Lake.

The number of input bytes has a small coefficient (less than 1 cycle per input byte) but its contribution to the cost is still significant because there are many more bytes than structural elements or floating-point numbers.

In our model, a floating-point number not only has a direct cost, but also generates one pseudo-structural character and comprises several bytes: thus each number costs dozens of cycles.

4.4 Fewer Instructions

The main benefit of SIMD instructions is to do more work with fewer instructions. Thus we expect our parser to user fewer instructions. We present in Table 8 the number of instructions needed to parse various file using our three competitive parsers. We provide the results derived from the Skylake processors, but the number of instructions is practically the same on the Cannon Lake processor.

On average, simdjson uses about half as many instructions as sajson and RapidJSON. Our fastest competitor, sajson, uses between 1.5 and 3.5 times more instructions. RapidJSON uses between 1.8 and 4.7 times more instructions. The files where our advantage over sajson is relatively smallest are marine_ik, mesh.pretty and twitterescaped. These files involve either much number parsing, or expensive string normalization.

The simdjson parser uses fewer instructions overall, but it uses many more vector instructions. Table 7 provides the latency and throughput of the 256-bit vector instructions in simdjson. Most of them have minimal latency (1 cycle) and several instructions have high throughput: they can be issued twice per cycle (vpaddb, vpand, vpcmpeqb, etc.). All these vector instructions can be issued at least once per cycle.

4.5 Speed Comparison

We present raw parsing speeds in Fig. 9. In only a few instances, sajson can slightly surpass 1 GB/s and RapidJSON can slightly surpass 0.5 GB/s. On our Skylake (3.4 GHz) processor, our parser (simdjson) can achieve and even surpass 2 GB/s in six instances, and for gsoc-2018, we reach 3 GB/s.

The purpose of parsing is to access the data contained in the document. It would not be helpful to quickly parse documents if we could not, later on, access the parsed tree quickly. In Table 9 we present our results while parsing the twitter document and finding all unique user.id (SELECT DISTINCT “user.id”
Table 7: 256-bit SIMD instructions: their latency (in cycles) and throughput (cycles/instruction) on Skylake processors [11].

| Instruction | Description                  | Latency | Throughput |
|-------------|------------------------------|---------|------------|
| vpaddb      | Add 8-bit integers           | 1       | 0.5        |
| vpalignr    | Concatenate pairs of 16-byte blocks | 1       | 1          |
| vpand       | Compute the bitwise AND      | 1       | 0.5        |
| vpcmpqeqb   | Compare 8-bit integers (==)  | 1       | 0.5        |
| vpcmpgtb    | Compare 8-bit integers (>)   | 1       | 0.5        |
| vperm2i128  | Shuffle integers             | 3       | 1          |
| vpmaddub    | Compute max of 8-bit integers| 1       | 0.5        |
| vpor        | Compute the bitwise OR       | 1       | 0.5        |
| vpsubbfb    | Shuffle bytes                | 1       | 1          |
| vpsrlf      | Right shift 32-bit integers  | 1       | 1          |
| vpsrlw      | Right shift 16-bit integers  | 1       | 1          |
| vpsubbub    | Subtract 8-bit integers      | 1       | 0.5        |
| vptest      | Test for zero                | 3       | 1          |

Table 8: Instructions per byte required to parse and validate documents.

| File            | simdjson | RapidJSON | sajson | RapidJSON/simdjson | sajson/simdjson |
|-----------------|----------|-----------|--------|--------------------|----------------|
| apache_builds   | 5.6      | 15.9      | 10.0   | 2.8                | 1.8            |
| canada          | 12.9     | 26.2      | 20.9   | 2.0                | 1.6            |
| cimt_catalog    | 5.3      | 11.7      | 11.1   | 2.2                | 2.1            |
| github_events   | 4.9      | 15.5      | 10.1   | 3.2                | 2.1            |
| gsoc-2018       | 3.2      | 15.0      | 11.2   | 4.7                | 3.5            |
| instruments     | 6.4      | 15.3      | 12.6   | 2.4                | 2.0            |
| marine_jk       | 13.4     | 23.7      | 20.6   | 1.8                | 1.5            |
| mesh_pretty     | 9.0      | 17.0      | 14.9   | 1.9                | 1.7            |
| mesh            | 14.3     | 27.2      | 23.3   | 1.9                | 1.6            |
| numbers         | 11.7     | 25.9      | 18.8   | 2.2                | 1.6            |
| random          | 8.9      | 19.6      | 15.4   | 2.2                | 1.7            |
| twitter         | 5.5      | 14.3      | 11.5   | 2.6                | 2.1            |
| twitterescaped  | 9.3      | 16.5      | 13.7   | 1.8                | 1.5            |
| update-center   | 6.2      | 18.4      | 12.1   | 3.0                | 2.0            |
| Average         | 8.3      | 18.7      | 14.7   | 2.4                | 1.9            |

FROM tweets), a query from Tahara et al. [27]. We report the throughput in GB/s to parse and scan the resulting tree. Our parser is again twice as fast as the reference parsers. We fully parse and then fully traverse the result, in two stages, without any optimization. We achieve 1.8 GB/s, down from 2.2 GB/s, when merely parsing on our Skylake processor. Unsurprisingly, a full parsed-tree traversal takes a fraction of the time required to fully parse the original document—reaching the equivalent of 10 GB/s in our case.

In Table 10, we present the parsing speed in gigabytes per second (GB/s) for several different parsers. In particular, we present results regarding RapidJSON using both the default configuration and the faster version that we use elsewhere (with insitu processing). In several cases, insitu processing is faster (apache_build, gsoc-2018, twitter, etc.) up to a factor of two, while in other cases, such as all files made mostly of numbers, the difference is negligible. Compared with RapidJSON without insitu string processing, our parser (simdjson) can be more than five times faster. RapidJSON further allows us to disable character encoding validation or to use higher precision number parsing, we do not report the results for these cases. For sajson, we use both the default dynamic-memory allocation and the faster version with static-memory allocation which we use elsewhere. The dynamic-memory allocation leads to a significant performance penalty, but we expect that it makes the parser more conservative in its memory usage. For reference, we also include several other popular C/C++ parsers even though we found them all to be lacking regarding their validation: the Dropbox parser.

7 https://github.com/dropbox/json11
Table 9: Throughput in GB/s to parse and then select all distinct user.id values from the parsed tree, using the file twitter. We provide corresponding time for merely parsing and generator a parsed tree, or merely traversing the parsed tree looking for distinct user.id values.

(a) Skylake

| parser     | parse + select | parse only | select only |
|------------|----------------|------------|-------------|
| simdjson   | 1.8            | 2.2        | 11.7        |
| RapidJSON  | 0.7            | 0.8        | 10.8        |
| sajson     | 0.9            | 1.0        | 15.5        |

(b) Cannon Lake

| parser     | parse + select | parse only | select only |
|------------|----------------|------------|-------------|
| simdjson   | 1.6            | 2.0        | 9.4         |
| RapidJSON  | 0.6            | 0.7        | 9.7         |
| sajson     | 0.8            | 0.9        | 13.6        |

5 Conclusion and Future Work

Though the application of SIMD instructions for parsing is not novel [5], our results suggest that they are underutilized in popular JSON parsers. We expect that many of our strategies could benefit existing JSON parsers like RapidJSON. It may even be possible to integrate the code of our parser (simdjson) directly into existing libraries.

JSON is one of several popular data formats such as Protocol Buffers, XML, YAML, MessagePack, BSON, CSV, or CBOR. We expect that many of our ideas would apply to other formats.

JSON documents are all text. Yet we frequently need to embed binary content inside such documents. The standard approach involves using base64 encoding. Base64 data can be decoded quickly using SIMD instructions [21]. Because number parsing from text is expensive, it might be fruitful to store large arrays of numbers in binary format using base64.

Intel has produced a new family of instruction sets with wider vector registers and more powerful instructions (AVX-512). Our Cannon Lake processor supports these instructions, including the AVX512-VBMI extension, which is relevant to the byte processing required for this work. Future research should assess the benefits of AVX-512 instructions.

Many of our strategies are agnostic to the specific architecture of the processor. Future research should assess our performance on other processors, such as those of the ARM or POWER families [15].

In our current implementation of simdjson, we complete the entire stage 1 before completing stage 2. For large documents, we could interleave stage 1 and stage 2. If needed, stage 1 can be further parallelized with thread-15

---

15 The simdjson library works on 64-bit ARM processors.
Table 10: Parsing speed in gigabytes per second (GB/s) for several different parsers (Skylake)

|                | simdjson | Rapidx | Rapidx ins. | sajson dyn. | sajson | Droppbox | fastjson | gjson | utrajson | json | jsjson | jsoncpp | JSON for Modern C++ |
|----------------|---------|--------|-------------|-------------|--------|----------|----------|-------|----------|------|-------|---------|-------------------|
| apache_builds  | 2.3     | 0.48   | 0.80        | 0.84        | 1.2    | 0.16     | 0.28     | 0.91  | 0.37     | 0.07 | 0.29  | 0.15    | 0.10             |
| canada         | 1.1     | 0.43   | 0.43        | 0.60        | 0.76   | 0.07     | 0.22     | 0.95  | 0.48     | 0.01 | 0.07  | 0.04    | 0.05             |
| citm_catalog   | 2.5     | 0.86   | 1.0         | 0.85        | 1.2    | 0.24     | 0.39     | 1.1   | 0.65     | 0.19 | 0.36  | 0.20    | 0.14             |
| github_events  | 2.5     | 0.46   | 0.79        | 1.0         | 1.1    | 0.15     | 0.28     | 0.89  | 0.38     | 0.52 | 0.28  | 0.14    | 0.09             |
| gsoc-2018      | 3.2     | 0.51   | 1.0         | 0.96        | 1.1    | 0.23     | 0.31     | 1.1   | 0.44     | 0.16 | 0.53  | 0.26    | 0.12             |
| instruments    | 2.1     | 0.59   | 0.76        | 0.70        | 1.1    | 0.14     | 0.34     | 0.95  | 0.43     | 0.24 | 0.25  | 0.13    | 0.10             |
| marine_lijk    | 0.94    | 0.45   | 0.45        | 0.51        | 0.67   | 0.07     | 0.22     | 0.77  | 0.37     | 0.18 | 0.07  | 0.04    | 0.06             |
| mesh           | 0.95    | 0.41   | 0.41        | 0.52        | 0.62   | 0.09     | 0.20     | 0.77  | 0.38     | 0.05 | 0.06  | 0.04    | 0.06             |
| mesh_pretty    | 1.5     | 0.67   | 0.67        | 0.64        | 0.92   | 0.15     | 0.32     | 1.1   | 0.62     | 0.09 | 0.13  | 0.07    | 0.11             |
| numbers        | 1.1     | 0.45   | 0.45        | 0.55        | 0.77   | 0.09     | 0.24     | 0.88  | 0.48     | 0.65 | 0.07  | 0.04    | 0.06             |
| random         | 1.4     | 0.38   | 0.58        | 0.46        | 0.81   | 0.10     | 0.26     | 0.79  | 0.30     | 0.03 | 0.18  | 0.09    | 0.07             |
| twitter        | 2.2     | 0.45   | 0.72        | 0.73        | 1.0    | 0.14     | 0.28     | 0.80  | 0.42     | 0.27 | 0.33  | 0.14    | 0.10             |
| twitterescaped | 1.2     | 0.38   | 0.62        | 0.62        | 0.84   | 0.11     | 0.26     | 0.72  | 0.36     | 0.26 | 0.27  | 0.11    | 0.08             |
| update-center  | 1.9     | 0.38   | 0.64        | 0.50        | 0.86   | 0.11     | 0.25     | 0.71  | 0.31     | 0.06 | 0.25  | 0.10    | 0.07             |

Acknowledgements The vectorized UTF-8 validation was motivated by a blog post by O. Goffart. K. Willets helped design the current vectorized UTF-8 validation. In particular, he provided the algorithm and code to check that sequences of two, three and four non-ASCII bytes match the leading byte. The authors are grateful to W. Mula for sharing related number-parsing code online. The software library has benefited from the contributions of T. Navenmec, K. Wolf, T. Kennedy, F. Wessels, G. Fotopoulos, H. N. Gies, E. Gedda, G. Floros, D. Xie, N. Xiao, E. Bogatov, J. Wang, L. F. Peres, W. Bolsterlee, A. Karandikar, R. Urban, T. Dyson, I. Dottsenko, A. Milovidov, C. Liu, S. Gleason, J. Keiser, Z. Bjornson, V. Baranov, I. A. Daza Dillon and others.

The work is supported in part by the Natural Sciences and Engineering Research Council of Canada under grant RGPIN-2017-03910.

A Effect of Minification

To ease readability, JSON documents may contain a variable number of white-space characters between atoms, within objects and arrays. Intuitively, these superfluous characters should reduce parsing speed. To verify this intuition, we mini-fied the documents prior to parsing. We find that all three parsers (simdjson, RapidJSON, sajson) use fewer CPU cycles to parse minified documents, see Table 11. However, the benefits in processing speed is often less than the benefits in storage. For example, the minified apache_builds file is 74% of the original, yet the processing time is only reduced to between 82% and 90% of the original—depending on the parser. The sajson parser often benefits more from minification. Thus while simdjson is more than 2.1 times faster than sajson on the documents prior to parsing, see Table 11. However, the parsers (simdjson, RapidJSON, sajson) use fewer CPU cycles to parse minified documents, see Table 11. However, the benefits in processing speed is often less than the benefits in storage. For example, the minified apache_builds file is 74% of the original, yet the processing time is only reduced to between 82% and 90% of the original—depending on the parser. The sajson parser often benefits more from minification. Thus while simdjson is more than 2.1 times faster than sajson on the original twitter document, it is only 1.9 times faster after minification.

B Effect of Specific Optimizations

In a complex task like JSON parsing, no single optimization is likely to make a large difference. Nonetheless, it may be useful to quantify the effect of some optimizations.

- We can compute the string masks (indicating the span of the strings) using a single carry-less multiplication between a word containing the location of the quote characters (as 1-bits) and a word containing only ones (see § 3.1.1). Alternatively, we can achieve the same result with a series of shifts and XOR:

  ```c
  uint64_t mask = quote xor (quote << 1);
  mask = mask xor (mask << 2);
  mask = mask xor (mask << 4);
  mask = mask xor (mask << 8);
  mask = mask xor (mask << 16);
  mask = mask xor (mask << 32);
  
  // Instead of extracting the set bits using our optimized algorithm (see Fig. 6 in §3.1.4), we can use a naive approach:
  while(s != 0) {
    *b++ = idx + trailingzeros(s);
    s = s & (s - 1);
  }
  ```

- Instead of using vectorized classification (§ 3.1.2), we can use a more naive approach where we detect the locations of structural characters by doing one vectorized comparison per structural character, and doing a bitwise OR. Similarly, we can detect spaces by doing one comparison per allowable white-space character and doing a bitwise OR.

We present the number of cycles per input byte in Table 12 with the three optimizations disabled one by one. The standard error of our measure is about 0.02 cycles per byte so small differences (< 0.05 cycles) may not be statistically significant. The fast index extraction reduces the cost of the stage 1 by over 10% in several instances (e.g., twitter, update-center). The carry-less multiplication appear has gains of over 5% in some instance, the gains reach nearly 20% in one instance (mesh). Vectorized classification is similarly helpful.
Table 11: Millions of cycles required to parse and validate selected documents (Skylake), before and after minification.

| file            | ratio (%) | parser       | original | minified | original/minified (%) |
|-----------------|-----------|--------------|----------|----------|-----------------------|
| apache_builds   | 74        | simdjson     | 0.19     | 0.17     | 86                    |
|                 |           | RapidJSON    | 0.55     | 0.49     | 90                    |
|                 |           | sajson       | 0.34     | 0.28     | 82                    |
| citm_catalog    | 29        | simdjson     | 2.34     | 1.64     | 70                    |
|                 |           | RapidJSON    | 6.30     | 3.49     | 55                    |
|                 |           | sajson       | 4.79     | 2.40     | 50                    |
| github_events   | 82        | simdjson     | 0.09     | 0.08     | 90                    |
|                 |           | RapidJSON    | 0.29     | 0.25     | 86                    |
|                 |           | sajson       | 0.20     | 0.16     | 81                    |
| gsoc-2018       | 92        | simdjson     | 3.58     | 3.39     | 95                    |
|                 |           | RapidJSON    | 11.2     | 10.6     | 95                    |
|                 |           | sajson       | 8.76     | 8.19     | 95                    |
| instruments     | 49        | simdjson     | 0.36     | 0.29     | 80                    |
|                 |           | RapidJSON    | 0.98     | 0.75     | 77                    |
|                 |           | sajson       | 0.70     | 0.47     | 68                    |
| marine_iK       | 61        | simdjson     | 10.6     | 10.2     | 96                    |
|                 |           | RapidJSON    | 22.3     | 20.2     | 91                    |
|                 |           | sajson       | 15.1     | 12.9     | 85                    |
| mesh            | 90        | simdjson     | 2.56     | 2.54     | 99                    |
|                 |           | RapidJSON    | 6.06     | 6.00     | 99                    |
|                 |           | sajson       | 3.88     | 3.70     | 96                    |
| mesh.pretty     | 48        | simdjson     | 3.64     | 3.19     | 88                    |
|                 |           | RapidJSON    | 8.50     | 6.56     | 77                    |
|                 |           | sajson       | 5.61     | 4.08     | 73                    |
| random          | 90        | simdjson     | 1.22     | 1.19     | 97                    |
|                 |           | RapidJSON    | 2.96     | 2.90     | 98                    |
|                 |           | sajson       | 2.08     | 1.96     | 94                    |
| twitter         | 74        | simdjson     | 0.99     | 0.88     | 90                    |
|                 |           | RapidJSON    | 3.08     | 2.40     | 78                    |
|                 |           | sajson       | 2.11     | 1.63     | 77                    |

C Large Files

All three fast parsers (simdjson, RapidJSON, sajson) mostly read and write sequentially in memory. Furthermore, the memory bandwidth of our systems are far higher than our parsing speeds. Thus we can expect them to perform well even when all of the data does not fit in cache. To verify that we can still process JSON data at high speed when the input data exceeds the cache, we created a large file (refsnp-unsupported35000) made of the first 35,000 entries from a file describing human single nucleotide variations (refsnp-unsupported.json). Table 13 shows that we far exceed 1 GB per second with simdjson in this instance although the file does not fit in CPU cache.

References

1. Alagiannis I, Borovica R, Branco M, Idreos S, Ailamaki A (2012) NoDB in Action: Adaptive Query Processing on Raw Data. Proc VLDB Endow 5(12):1942–1945
2. Boncz PA, Graefe G, He B, Sattler KU (2019) Database architectures for modern hardware. Tech. Rep. 18251, Dagstuhl Seminar
3. Bonetta D, Brantner M (2017) FAD.Js: Fast JSON Data Access Using JIT-based Speculative Optimizations. Proc VLDB Endow 10(12):1778–1789
4. Bray T (2017) The JavaScript Object Notation (JSON) Data Interchange Format [https://tools.ietf.org/html/rfc8259] internet Engineering Task Force, Request for Comments: 8259
5. Cameron RD, Herdy KS, Lin D (2008) High Performance XML Parsing Using Parallel Bit Stream Technology. In: Proceedings of the 2008 Conference of the Center for Ad-
Table 12: Performance in cycles per bytes of the simdjson parser during stage 1 over several files. The first numerical column presents the original performance, with all optimizations. The second column presents the results without the carry-less multiplication. The third column presents the results with a naive bit extraction. The fourth column presents the results with naive classification—without vectorized classification.

| file          | original | without carry-less | naive bit extraction | naive classification |
|---------------|----------|--------------------|----------------------|----------------------|
| canada        | 0.90     | 0.94               | 0.96                 | 0.96                 |
| citm_catalog  | 0.75     | 0.78               | 0.81                 | 0.81                 |
| github_events | 0.74     | 0.77               | 0.74                 | 0.83                 |
| gsoc-2018     | 0.65     | 0.67               | 0.69                 | 0.68                 |
| instruments   | 0.81     | 0.84               | 1.01                 | 0.88                 |
| marine_ik     | 1.02     | 1.09               | 1.25                 | 1.09                 |
| mesh          | 0.90     | 1.07               | 1.07                 | 0.99                 |
| mesh.pretty   | 0.77     | 0.82               | 0.82                 | 0.85                 |
| numbers       | 0.83     | 0.86               | 0.88                 | 0.89                 |
| random        | 1.22     | 1.26               | 1.57                 | 1.31                 |
| twitter       | 0.90     | 0.92               | 1.14                 | 1.00                 |
| twitterescaped| 0.79     | 0.86               | 0.96                 | 0.85                 |
| update-center | 0.85     | 0.87               | 1.09                 | 0.91                 |

Table 13: Throughput in GB/s when parsing a large (84 MB) file: refsnp-unsupported

| parser          | Skylake | Cannon Lake |
|-----------------|---------|-------------|
| simdjson        | 1.4     | 1.3         |
| RapidJSON       | 0.56    | 0.44        |
| sajson          | 0.93    | 0.84        |

14. Green TJ, Gupta A, Miklau G, Onizuka M, Suciu D (2004) Processing XML Streams with Deterministic Automata and Stream Indexes. ACM Trans Database Syst 29(4):752–788
15. Kostoulas MG, Matsa M, Mendelsohn N, Perkins E, Heifets A, Mercedi M (2006) XML Screamer: An Integrated Approach to High Performance XML Parsing, Validation and Deserialization. In: Proceedings of the 15th International Conference on World Wide Web, ACM, New York, NY, USA, WWW ’06, pp 93–102
16. Lemire D, Kaser O (2016) Faster 64-bit universal hashing using carry-less multiplications. Journal of Cryptographic Engineering 6(3):171–185
17. Li Y, Katsipoulakis NR, Chandramouli B, Goldstein J, Kossmann D (2017) Mison: A Fast JSON Parser for Data Analytics. Proc VLDB Endow 10(10):1118–1129, DOI 10.14778/3115404.3115416
18. Liu ZH, Hammerschmidt B, McMahon D (2014) Json data management: Supporting schema-less development in rdbs. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, SIGMOD ’14, pp 1247–1258
25. Palkar S, Abuzaid F, Bailis P, Zaharia M (2018) Filter before you parse: faster analytics on raw data with Sparser. Proceedings of the VLDB Endowment 11(11):1576–1589

26. Pavlopoulou C, Carman Jr EP, Westmann T, Carey MJ, Tsotras VJ (2018) A Parallel and Scalable Processor for JSON Data. In: EDBT’18

27. Tahara D, Diamond T, Abadi DJ (2014) Sinew: A SQL System for Multi-structured Data. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, SIGMOD ’14, pp 815–826

28. Takase T, Miyashita H, Suzumura T, Tatsubori M (2005) An Adaptive, Fast, and Safe XML Parser Based on Byte Sequences Memorization. In: Proceedings of the 14th International Conference on World Wide Web, ACM, New York, NY, USA, WWW ’05, pp 692–701

29. Xie D, Chandramouli B, Li Y, Kossmann D (2019) FishStore: Faster ingestion with subset hashing. In: Proceedings of the 2019 International Conference on Management of Data, ACM, New York, NY, USA, SIGMOD ’19, pp 1711–1728

30. Xu Q, Siyamwala H, Ghosh M, Suri T, Awasthi M, Guz Z, Shayesteh A, Balakrishnan V (2015) Performance analysis of NVMe SSDs and their implication on real world databases. In: Proceedings of the 8th ACM International Systems and Storage Conference, ACM, New York, NY, USA, SYSTOR ’15, pp 6:1–6:11

31. Zhang Y, Pan Y, Chiu K (2009) Speculative p-DFA for parallel XML parsing. In: High Performance Computing (HiPC), 2009 International Conference on, IEEE, pp 388–397