Scalable Structural Index Construction for JSON Analytics

Lin Jiang
University of California, Riverside
Riverside, California
ljian006@ucr.edu

Junqiao Qiu
Michigan Technological University
Houghton, Michigan
junqiaoq@mtu.edu

Zhijia Zhao
University of California, Riverside
Riverside, California
zhijia@cs.ucr.edu

ABSTRACT

JavaScript Object Notation (JSON) and its variants have gained great popularity in recent years. Unfortunately, the performance of their analytics is often dragged down by the expensive JSON parsing. To address this, recent work has shown that building bitwise indices on JSON data, called *structural indices*, can greatly accelerate querying. Despite its promise, the existing structural index construction does not scale well as records become larger and more complex, due to its (inherently) sequential construction process and the involvement of costly memory copies that grow as the nesting level increases.

To address the above issues, this work introduces Pison – a more memory-efficient structural index constructor with supports of intra-record parallelism. First, Pison features a redesign of the bottleneck step in the existing solution. The new design is not only simpler but more memory-efficient. More importantly, Pison is able to build structural indices for a single bulky record in parallel, enabled by a group of customized parallelization techniques. Finally, Pison is also optimized for better data locality, which is especially critical in the scenario of bulky record processing. Our evaluation using real-world JSON datasets shows that Pison achieves 9.8X speedup (on average) over the existing structural index construction solution for bulky records and 4.6X speedup (on average) of end-to-end performance (indexing plus querying) over a state-of-the-art SIMD-based JSON parser on a 16-core machine.

PVLDB Reference Format:
Lin Jiang, Junqiao Qiu, and Zhijia Zhao. Scalable Structural Index Construction for JSON Analytics. PVLDB, 14(4): 694 - 707, 2021.
doi:10.14778/3436905.3436926

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/AutomataLab/Pison.

1 INTRODUCTION

JSON (JavaScript Object Notation) has emerged as a popular data type in modern software applications [36]. Its derivatives, such as NetJSON [33], GeoJSON [32], JSON-LD [34], CoverageJSON [29], and others, span multiple domains. Together, they play critical roles in microservices [28, 60], Internet of Things (IoT) [61], NoSQL data stores [35, 45], and cloud computing [24, 47]. As the popularity of JSON increases, its data volume grows faster than ever. Many web applications, such as social networks (e.g., Twitter [14] and Facebook [15]) and online shopping (e.g., Bestbuy [1] and Walmart [10]) continuously produce a broad range of data in JSON format through open APIs. Public data sources, like Data.gov [4], host more datasets in JSON format of sizes easily reaching several gigabytes. However, analyzing JSON data requires parsing – an expensive task. Recent study [44] shows that JSON data parsing takes over 80% of the total time for complex queries and even a larger portion for simple queries. Hence, it is critical to accelerate the parsing in order to make the querying over JSON data performant.

State of The Art. Traditional ways of JSON parsing involve stack-based abstract machines, known as *pushdown automata*. Basically, an automaton traverses a JSON record in serial and recognizes the nested syntactical structures with a stack. Most popular JSON parsers, like Jackson [8] and GSON [5], fall into this category. Recent work JPStream [38] proposes a dual-stack automaton to carry out path queries simultaneously with the parsing. However, an inherent limitation with the automata-based solutions is that they have to traverse the JSON data stream *character by character* to perform the parsing and query matching.

In fact, it is possible to "skip" irrelevant parts of the data stream with the help of indexing techniques. Recently, Mison [44] proposes a novel algorithm that generates bitwise indices (bitmaps) for the structural characters in a JSON record (i.e., ":" and ",") – *structural indices*. With the indices, a parser can directly jump into relevant positions of the JSON record to find matches. As the construction can leverage bitwise and SIMD-level parallelism, it can process tens of or even hundreds of characters simultaneously [44].

Despite its promise, there are several issues in the existing design of structural index construction that may limit its scalability as the records become larger and more complex. First, most steps in the structural index construction are inherently sequential, preventing it from taking advantage of the coarse-grained parallelism. For bulky records, lack of intra-record parallelism fundamentally limits the efficiency of structural index construction. Second, one critical step in the current design of structural index construction [44] involves many costly memory operations, which gets worsen as the record becomes larger and more deeply nested. At last, the existing design assumes the record can fit into the caches, which is not the case for bulky records – naively using the current design to process bulky records may suffer from poor data locality.

Overview of This Work. The primary goal of this work is to scale the structural index construction to larger and more complex JSON records. To achieve this, we identify all the dependences involved in each step of the index construction, then develop specialized parallelization solutions to "break" those dependences. Specifically, we address the dependences related to backlash sequences with *dynamic partitioning*. However, partitioning may still cut the JSON

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 4 ISSN 2150-8097.
doi:10.14778/3436905.3436926

strings and the nested structures of a JSON record. To handle broken JSON strings, we propose a combination of techniques, including a *contradiction-based context inference*, a *speculation technique* with fast *bitwise reprocessing*. Finally, to handle broken nested JSON structures, we leverage a *reduction-based* parallelization technique.

Besides parallelization, we also identify the inefficiencies in the bottleneck step of the construction and propose a new design which is not only easier to implement but also more efficient to execute, thanks to its reduced memory accesses. Our evaluation shows the new design brings 1.4X speedup to the total serial execution time.

Moreover, to cope with the large working set in processing bulky records, we propose to build structural indices word by word, rather than step by step for the whole record. This locality optimization itself brings 1.7X speedup on average for bulky records.

Finally, we integrated the above techniques and developed a structural index constructor called Pison. To ease the programming, Pison provides intuitive APIs that hide the details of index traversals. We evaluated Pison using a group of real-world JSON datasets with a focus on the bulky-record processing scenario. According to the results, Pison outperforms the existing structural index construction solution Mison [44] by 9.8X (on average) for bulky records, and achieves 4.6X speedup (on average) of end-to-end performance over the popular SIMD-based JSON parser simdjson[43]. The results confirm the effectiveness of the proposed techniques.

Contributions. This work makes three main contributions:

- First, it presents a group of parallelization techniques that enable intra-record data parallelism to the JSON structural index construction.
- Second, it proposes two key optimizations to the serial design of structural index construction: a redesign of the bottleneck step and a locality optimization.
- Finally, it implements the proposed ideas into a C++ library (Pison) and compares it with multiple state-of-the-art JSON tools using real-world datasets. The source code of Pison is available at https://github.com/AutomataLab/Pison.

Next, we provide the background of this work.

2 BACKGROUND

This section introduces JSON basics, followed by an overview of the current structural index construction and its limitations.

2.1 JSON and Its Querying

JSON Syntax. JSON data follows a rigorous syntax, which can be defined by a context-free grammar (CFG), as shown in Figure 1. At high level, there are two major structures: object and array. An object always starts with a left brace { and ends with a right brace }. Between them, there could be a series of key-value pairs, separated by commas, known as the attributes. By contrast, an array is placed between a pair of brackets [and]. Inside an array, there is a linear sequence of elements, separated by commas. Both object and array can be self-nested and also nested within each other. Here, we use the term "record" to refer to the top-level syntax structure in the nested JSON data, which could be either an object or an array.

Querying JSON Data. Essentially, each JSON record is a serialized hierarchy of an object or an array. A basic group of queries to JSON

```
  object ::= {} | {members}
 members ::= pair | pair, members
    pair ::= string:value
   array ::= [] | [elements]
elements ::= value | value, elements
   value ::= object | array |
             string | primitive
  string ::= "" | "characters"
```

(a) BNF Grammar of JSON

```
{"user": [ {
     "id": 1,
     "name": "\\\":A" },
    {"id": 2 }
  ]
}
```

(b) Sample JSON Data (Twitter)

Figure 1: JSON Grammar and Example.

data is to identify the sub-structures of interest inside the JSON hierarchy, known as *JSONPath queries* [36]. A JSONPath query specifies paths from the root of the hierarchy to the sub-structures of interest. For example, `$.user[0].id` asks for the `id` of `user[0]`, the first element of array `user`. A dot `.` refers to the attributes of an object; two dots (like `$..id`) refer to all recursive descendant objects. For all elements in an array, use star `*`, as in `$.user[*].id`. More details regarding the JSONPath queries can be found in [36].

Conventionally, evaluating JSON queries requires parsing a JSON record into a tree structure. Queries are evaluated by walking down the tree from the root and matching tree nodes against the queries. Many popular JSON tools follow this approach, like Jackson [8], GSON [5], RapidJSON [11], and FastJSON [2]. Alternatively, the parsing and querying can be merged into a single pass, as shown in JPStream [38], which avoids building any parse tree. In either approach, the parsing essentially simulates a pushdown automaton that consumes the JSON data character by character (i.e., one byte per time). However, modern CPUs can perform 64-bit (i.e., 8-byte) calculation, and even 512-bit calculation (i.e., 64-byte) with SIMD units [57]. In this perspective, existing automata-based parsers underutilize the fine-grained parallelism in modern CPUs.

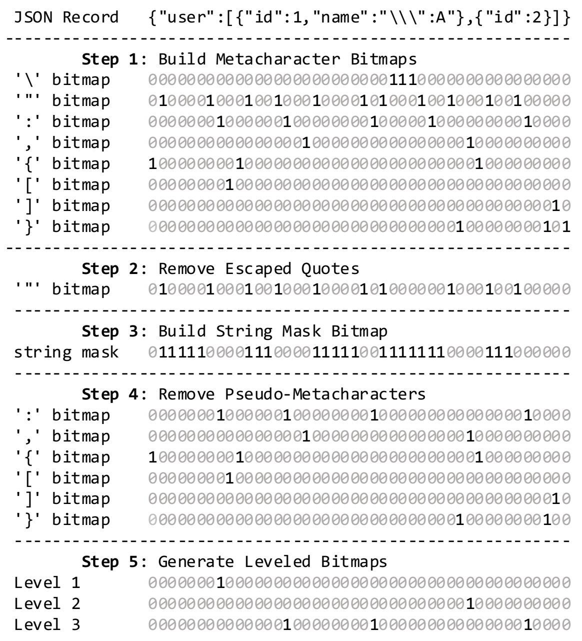
To effectively utilize bitwise and SIMD parallelism, recent work Mison [44] proposes a bitwise indexing scheme for JSON data. The basic idea is to construct bitwise indices on structural characters in a JSON record, such as ":" and ",". With the indices, a query can be quickly evaluated based on locations of object attributes and array elements of different levels. We next describe it.

2.2 Structural Index Construction

This section summarizes the structural index construction proposed in Mison [44], with some steps adopted from simdjson [43]. In general, the construction has five steps [1], as illustrated in Figure 2.

Step 1: Build Metacharacter Bitmaps. It first builds a bitmap for each metacharacter in JSON, including `,`, `:`, `[`, `]`, `{`, `}`, `"`, and `\`. In the bitmap, 1s represent the positions of the metacharacter in the record. The bitmap can be constructed by comparing the metacharacter against the characters in the JSON record one by

[1]We slightly modified steps in [44] for better illustration.



```
JSON Record   {"user":[{"id":1,"name":"\\\":A"},{"id":2}]}
------------------------------------------------------------
        Step 1: Build Metacharacter Bitmaps
'\' bitmap    00000000000000000000000001110000000000000000
'"' bitmap    01000010001001000100001010001001000100100000
':' bitmap    00000001000000100000000100000100000000010000
',' bitmap    00000000000000001000000000000000010000000000
'{' bitmap    10000000010000000000000000000000001000000000
'[' bitmap    00000000100000000000000000000000000000000000
']' bitmap    00000000000000000000000000000000000000000010
'}' bitmap    00000000000000000000000000000000100000000101
------------------------------------------------------------
        Step 2: Remove Escaped Quotes
'"' bitmap    01000010001001000100001010000001000100100000
-----------------------------------------------------------
        Step 3: Build String Mask Bitmap
string mask   01111100001110000111110011111110000111000000
-----------------------------------------------------------
        Step 4: Remove Pseudo-Metacharacters
':' bitmap    00000001000000100000000100000000000000010000
',' bitmap    00000000000000001000000000000000010000000000
'{' bitmap    10000000010000000000000000000000001000000000
'[' bitmap    00000000100000000000000000000000000000000000
']' bitmap    00000000000000000000000000000000000000000010
'}' bitmap    00000000000000000000000000000000100000000100
-----------------------------------------------------------
        Step 5: Generate Leveled Bitmaps
Level 1       00000001000000000000000000000000000000000000
Level 2       00000000000000000000000000000000010000000000
Level 3       00000000000000100000000100000000000000010000
```

Figure 2: Example Structural Index Construction.

one. For better efficiency, SIMD instructions can be leveraged to compare multiple characters simultaneously. Assume that the SIMD width is 256-bit, then a 8-bit metacharacter is duplicated 32 times to populate a 256-bit vector (*_mm256_setr_epi8*). The vector is then compared against the characters in JSON data, 256 bits (32 bytes) each time (*_mm256_cmpeq_epi8*). The result is in a 256-bit vector where each 8-bit lane is either all 1s or all 0s. Finally, the most significant bit of each lane is selected and packed into a 32-bit integer (*_mm256_movemask_epi8*). On a 64-bit machine, two such 32-bit integers are combined into a `long` type. This step runs in $O(\frac{8n}{W})$ instructions, where n is the number of characters in the JSON record and W is the SIMD width (e.g., 256).

Note that the metacharacters may appear inside a string, in which case they are not actually effective in defining the JSON structure. To exclude them, we need to first identify the strings in JSON, which are marked by quotes. However, a quote may be escaped by \, which can be further escaped, as in \\\". The next three steps are to find the actual strings in a JSON record and exclude the metacharacters in strings from the corresponding bitmaps.

Step 2: Remove Escaped Quotes. This step excludes all the escaped quotes – quotes following an odd number of consecutive \s, from the quote bitmap. To achieve this, Mison [44] iterates through all the quotes with backslashes ahead. In comparison, simdjson [43] uses a bitwise solution: it locates the starting and ending positions of backslash sequences, then finds odd-length backslash sequences based on the fact that a sequence of characters that starts at an odd (or even) position and ends at an even (or odd) position must have an odd length, thus, any following quotes should be escaped. More details about step and its implementation can be found in [43].

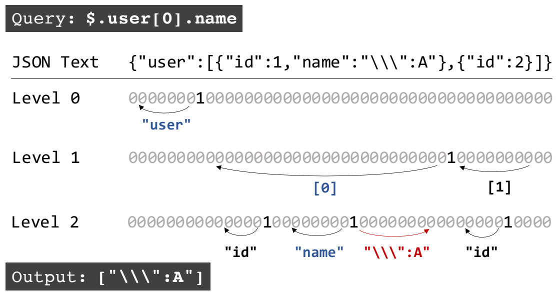

Query: $.user[0].name
JSON Text {"user":[{"id":1,"name":"\\\":A"},{"id":2}]}
Level 0 00000001000000000000000000000000000000000000
"user"
Level 1 00000000000000000000000000000000010000000000
[0]
[1]
Level 2 00000000000000100000000100000000000000010000
"id"
"name"
"\\\":A"
"id"
Output: ["\\\":A"]

Figure 3: Querying via Structural Indices.

Figure 2 (Step 2) shows the updated quote bitmap, with one escaped quote (the 4th to the right) removed. This step needs $O(\frac{n}{w})$ instructions, where w is the word size.

Step 3: Build String Mask Bitmap. Next, it generates a string mask bitmap, where 1s correspond to characters in strings. Here, a bitwise approach can be also adopted from simdjson [43]. First, a prefix-sum of `XOR` is applied to the quote bitmap. The resulted bit at index i equals to the `XOR` of all bit values up to i (inclusive). In fact, this operation can be implemented by performing a carry-less multiplication `PCLMUQDQ` between the quote bitmap and a 64-bit value with all 1s. This step also runs in $O(\frac{n}{w})$ instructions.

Step 4: Remove Pseudo-Metacharacters. This step removes those metacharacters that appear in strings, from their corresponding bitmaps. It can be implemented as an `ANDNOT` operation between the metacharacter bitmaps and the string mask bitmap.

Step 5: Generate Leveled Bitmaps. The final step is to separate the colon and comma bitmaps based on the levels that colons and commas appear in the record, dictated by the brackets `[`, `]`, `{`, `}`. As shown in Figure 2 (step 5), the top level is an object with one key-value pair, so there is a 1 at Level 1, corresponding to a colon. Next, as the value in the key-value pair is an array of two elements, Level 2 has one 1, for the comma in the array. This process repeats until it reaches the deepest level in the record (or in the path query). These leveled bitmaps are the final outputs – structural indices.

To implement this step, Mison [44] proposes to scan the brackets (`[`, `]`, `{`, `}`) bitmaps and employ a stack to recognize the nesting levels of colons and commas. Though the idea is intuitive, this solution actually involves expensive memory copy operations. We will elaborate this step in detail later in the paper.

In summary, the structural indices can be effectively constructed by taking advantages of bitwise and SIMD parallelism. For example, on a 64-bit processor, the construction can operate on 64-bit vectors, where a bit corresponds to one character (byte) in the JSON record. That is, the structural index construction can process 64 characters simultaneously. With the use of SIMD instructions, some of these operations can operate on vectors of 256 bits or even 512 bits, which can significantly accelerate the JSON data processing.

Querying Structural Indices. With the structural indices, we can quickly evaluate JSONPath queries. As illustrated in Figure 3, for a given JSONPath query (`$.user[0].name`), the evaluation starts from the top level of JSON structure (an object in the example), locates its key (`"user"`) through backward parsing from all 1s in this level, finds the range in the bitmap for its value – between this 1 and the next 1, then moves to the next level. This process stops when no match is found in a certain level or continues until the whole query expression has been matched.

2.3 Scalability Issues for Bulky Records

Despite its exploration of fine-grained parallelism, the existing structural index construction design [44] fails to scale well to bulky JSON records with complex structures, due to three reasons.

- ***No Intra-Record Data Parallelism***. Existing construction processes each record only in serial. For bulky records, the serial processing can seriously limit the scalability, resulting significant delay for answering a query.
- ***Heavy Memory Operations***. The last step (bottleneck step) of the existing construction requires to *duplicate* the colon and comma bitmaps K times, where K is the number of nesting levels in the record or query expression. For bulky records, making copies of whole bitmaps are very expensive. Then, the algorithm masks bits with overlapped ranges, with an increasing overhead as it moves to deeper levels. Finally, it employs a stack which also makes intensive memory copies during the push and pop operations.
- ***Poor Data Locality***. A simple adoption of the existing index construction algorithm to bulky records may suffer from poor data locality, as the generated intermediate bitmaps may not fit into the CPU cache(s).

The primary goal of this work is to bring coarse-grained data parallelism to the structural index construction such that a bulky record can be processed in parallel effectively. Moreover, this work also tries to improve the memory access efficiency by redesigning the critical step of structural index construction algorithm (Step 5) and offering a more locality friendly construction process.

3 DATA-PARALLEL CONSTRUCTION

For a sequence of small JSON records, parallelism naturally exists among different records. The challenge lies in the parallelization of building indices for a single bulky JSON record, which could cause significant delay when it is processed sequentially.

3.1 Dependences

When a bulky JSON record is partitioned, as illustrated in Figure 4, most steps of the structural index construction (except Steps 1 and 4) involve certain kinds of dependences. For Step 2 (removing escaped quotes), when a backslash sequence is broken into two partitions, we need to know the number of backslashes appearing at the suffix of the prior partition to tell if a following quote is escaped or not. For Step 3 (building string mask bitmap), we need to know whether the current partition starts inside a string or not. Finally, for Step 5 (generating leveled bitmaps), we need to know at which level of the JSON structure the current chunk begins.

To construct structural indices in parallel, we need to effectively address the above dependence challenges. In the following of this section, we will introduce an assembly of parallelization techniques that are integrated together to solve the dependences, including (i) *dynamic partitioning*, (ii) *contradiction-based context inference*, (iii) *speculation*, and (iv) *reduction*-style parallelization. Note that the last one (reduction-style parallelization) is based on a redesign of the Step 5 in the existing construction algorithm.

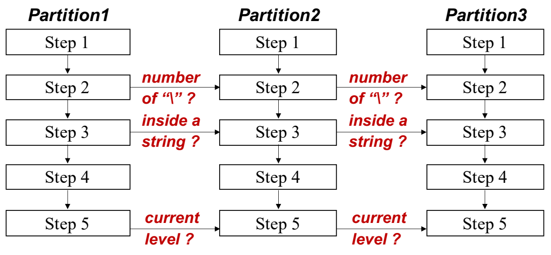


Figure 4: Dependences in Index Construction.

3.2 Dynamic Partitioning

Naively partitioning a JSON record may break a string, a keyword (e.g., `true` or `null`), or a sequence of backslashes, into different chunks. The first and third cases are related to the dependences in Step 3 and Step 2, respectively (see Figure 4). One way to avoid such cases is adjusting the boundary (e.g., moving it to the left) between two chunks dynamically such that no strings, keywords, or backslash sequences get cut. However, for a string, avoiding a bad cutting is non-trivial. Considering a piece of JSON data $\cdots$ "$\cdots$, in general, it is hard to tell if the quote is the start or the end of a string. For this reason, we only use this dynamic partitioning to avoid cutting of backslash sequences and keywords. Broken strings will be addressed next with more advanced techniques.

3.3 Contradiction-based Context Inference

In fact, issues related to broken strings also arise in other contexts of parallel processing. Recently, Ge and others [30] addressed the ambiguity issue caused by broken strings in parallel CSV parsing with a pattern-based approach. Two string patterns (each with two chars) are carefully designed for CSV data, such that once they are observed, the parsing ambiguity can be immediately resolved. Though the idea is inspiring, it is unclear how similar patterns can be designed for JSON whose syntax is much more complex. Next, we will present a *contradiction*-based string context inference. Unlike prior work [30], this inference is not based on the specific data syntax, thus may also work for data types beyond JSON.

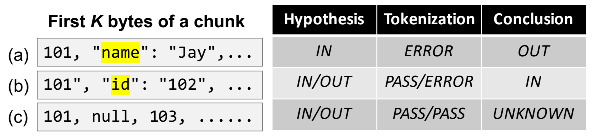
The key insight behind the contradiction-based inference is that *characters inside a string, most of the time, do not form valid tokens.* For example, in string `"user"`, `user` cannot be interpreted as any valid JSON token(s). In fact, any string with some alphabet letters ([a-zA-Z]), except keywords[2] and scientific numbers (e.g., `2e+7`), cannot be tokenized successfully. Based on this insight, we propose to build *hypothesis* and leverage *contradiction* to infer the string status at the beginning of a JSON partition.

Algorithm 1 shows the contradiction-based context inference. First, it assumes that the input chunk starts inside a string (i.e., $hypothesis$ = IN). Based on that, it tries to recognize the tokens in the first K bytes of the chunk. Once it hits an error of tokenization, it gets a contradiction – the current assumption is wrong, therefore

[2]Alphabetical keywords in JSON: `true`, `false`, and `null`.

Algorithm 1 Contradiction-based Context Inference

```
 1: Inputs: head: the first K bytes of a chunk
 2: Outputs: IN/OUT/UNKNOWN: string status at beginning
 3: Procedure:
 4: while hypothesis = {IN, OUT} do
 5:     while TRUE do
 6:         /* reach the end of the head */
 7:         if head.hasNextToken() == END then
 8:             break
 9:
10:         /* cannot recognize the next token */
11:         if head.hasNextToken() == ERROR then
12:             return opposite(hypothesis)
13:
14:         head.nextToken()
15:
16: return UNKNOWN
```



	First K bytes of a chunk	Hypothesis	Tokenization	Conclusion
(a)	`101, "name": "Jay",...`	*IN*	*ERROR*	*OUT*
(b)	`101", "id": "102", ...`	*IN/OUT*	*PASS/ERROR*	*IN*
(c)	`101, null, 103, ......`	*IN/OUT*	*PASS/PASS*	*UNKNOWN*

Figure 5: Examples of Context Inference.

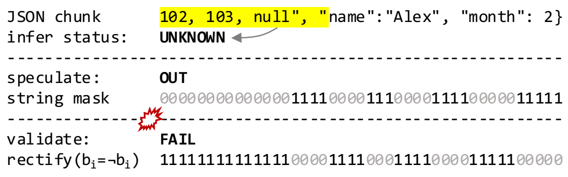
it returns the opposite status – OUT (Line 11-12), as the conclusion. An example of such cases is Figure 5-(a). Otherwise, if the first K bytes are all successfully tokenized (Line 7), the algorithm would fail to draw any conclusion, because some sequences of characters can be interpreted in either way correctly: part of a string or valid JSON tokens, like the example in Figure 5-(c). In this case, the algorithm tries the opposite assumption – *hypothesis* = OUT, and tokenizes the first K bytes again. If it encounters a tokenization error (i.e., a *contradiction*), it would return IN as the conclusion, like the case in Figure 5-(b). Finally, if both attempts fail to draw any conclusion, the algorithm returns UNKNOWN, as in the case of Figure 5-(c).

The more bytes (i.e., larger K) the algorithm checks, usually the higher chances it can draw a conclusion, but this will also incur higher overhead. In evaluation, we set $K = 64$ by default, and found it rarely fails to draw the conclusion. But, what if it does fail to draw any conclusion? We next address such cases with speculation.

3.4 Speculative Parallelization

Note that, even when the conclusion is UNKNOWN, the chances for being inside and being outside a string are usually not equal. In fact, most of the time, they are highly biased, for the same reason mentioned earlier: it is rarely seen that characters inside a JSON string form valid JSON tokens. In another word, if the inference has successfully recognized the first K bytes as valid JSON tokens, it is very unlikely they are part of a string. Therefore, if we speculatively consider the status is OUT, there is a high chance that it is true.

Based on this intuition, we propose a speculative parallelization scheme for structural index construction. If the context inference of one chunk returns UNKNOWN, the construction enters into speculative mode, in which chunks with UNKNOWN inference results are processed speculatively by assuming they start from a position outside a string. The next question is when the speculation is validated. If we validate it after all the five steps are finished (on each chunk), once a speculation fails, we have to rerun all the steps on the corresponding chunk. To minimize the misspeculation cost,



```
JSON chunk        102, 103, null", "name":"Alex", "month": 2}
infer status:     UNKNOWN
-------------------------------------------------------------
speculate:        OUT
string mask       00000000000000111100001110000111100000011111
-------------------------------------------------------------
validate:         FAIL
rectify(bi=¬bi)   11111111111111000011110001111000011111100000
```

Figure 6: Bitwise Rectification for Misspeculation.

we should validate the speculation right after Step 3. To achieve this, we add a synchronization between Steps 3 and 4, where chunks in speculative modes are validated against the ground truth (IN or OUT) of prior non-speculative chunks. In cases some speculation fails, a naive handling is reprocessing the corresponding chunks (up to Step 3). In fact, this is an overkill for this particular kind of misspeculation. We will present a faster alternative shortly.

Similar to our situation, the pattern-based approach [30] used in parallel CSV parsing may also fail to resolve the parsing ambiguity sometimes, thus turns to speculation for parallelization. However, as we show next, for structural index construction, the misspeculation can be handled efficiently at bit level with *bitwise rectification*.

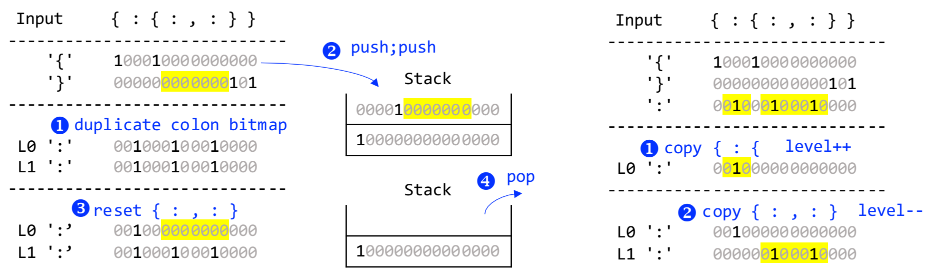
Bitwise Rectification. Instead of reprocessing the JSON chunk that was misinterpreted, we found it is possible to directly recover the correct bit values from the incorrect string mask bitmap with a simple bitwise logic operation $b_i = \neg b_i$. Figure 6 illustrates this idea with an example JSON chunk. In the example, the string status inference fails to draw any conclusion (i.e., UNKNOWN), thus the index construction switches to the speculative mode and assumes the string status is OUT. However, during the validation, it turns out the correct string status is IN. Interestingly, as shown in Figure 6, the correct string mask bitmap is exactly the "opposite" of the incorrect bitmap. Essentially, this is due to the *parity* of quotes in defining strings – a string always starts from an odd-number quote and ends at the next (even-number) quote. An incorrect interpretation of the quote parity would flip all the following string definitions. Based on this observation, we can generate the correct bitmap by flipping values in the incorrect bitmap, that is, $b_i = \neg b_i$. As expected, we found this bitwise rectification is much faster than reprocessing the JSON chunk (up to Step 3).

3.5 Parallel Generation of Leveled Bitmaps

The last dependence to address is the "level" at the beginning of a chunk in Step 5 (generating leveled bitmaps). Before introducing the solution, we first summarize the existing design of Step 5 from Mison [44] and present a new design for this step which is simpler yet more efficient. After that, we will explain how the new design can be parallelized based on the reduction-style parallelism.

3.5.1 Redesign of Step 5. The main task of Step 5 is to separate colons and commas of different levels into different bitmaps.

Existing Design. Figure 7-(a) illustrates the basic idea of Step 5 in Mison [44]. For easier explanation, the figure only shows curly brackets and colons that capture JSON objects, but the idea can be easily extended to cover JSON arrays. First, it duplicates the colon bitmap K times, where K is the number of levels in the JSON record or in the query expression (e.g., two times for `$.user.name`). The



(a) Prior Design: Stack + Bit Reset (b) New Design: Counter + Bit Copy

Figure 7: Generation of Leveled Bitmaps: Existing Design (left) vs. New Design (right).

duplicated bitmaps, denoted as $L_0 \cdots L_{K-1}$, will be turned into the leveled bitmaps; Then, it tries to locate the objects from inner levels to outer levels by traversing the bitmaps of { and } with the help of a stack. Basically, from left to right, the bitmap corresponding each { is pushed onto the stack. The stack top has the beginning position of the inner most object (the position with 1). From there, Mison scans forward to find the first } – the ending position of the inner most object. Then, it resets the bits in the same range in L_0, as highlighted in Figure 7-(a). After that, Mison pops the stack, moves to the second inner most level, and repeats the same operations. This process continues until Mison reaches the outer most level of the JSON structure. In practice, we found that an implementation based on the above design executes efficiently for small JSON records, taking 30-40% of the total construction time. However, for bulky records, this ratio is raised to 60-70%, making Step 5 the bottleneck of structural index construction. Our further investigation reveals two main factors limiting its efficiency:

- *Intensive Bit Value Updates.* First, bitmap duplication involves intensive memory writes to (pre-allocated) bitmaps. Second, from an inner level to an outer level, as the object range becomes larger, more bits need to be reset. The complexity of the bit resetting is $O(\frac{n}{w} \cdot b)$, where n is the number of bytes, w is the word size, and b is the number of brackets.
- *Expensive Stack Operations.* Employing a stack to record the nesting level and the start positions of objects needs to copy bitmaps onto and off the stack (push and pop), which are relatively expensive operations.

To address the efficiency issues in the existing design of Step 5, we next introduce a new algorithm for generating leveled bitmaps.

New Design. Figure 7-(b) illustrates the basic idea of the new design. First, instead of duplicating the colon bitmap and reseting the bits, the new design first allocates leveled bitmaps with all `0`s (`calloc()`), then copies bits of different levels from the colon bitmap to the corresponding leveled bitmaps, which reduces the bit value updates.

Second, the new design recognizes objects from outer levels to inner levels – the opposite of the existing solution. This avoids the use of stacks for recording the beginning positions in the existing design. In this case, a counter, like variable `level` in Figure 7-(b), is sufficient for recognizing the levels of objects.

More details of this new design are shown in Algorithm 2, which covers the generation of leveled bitmaps for both fields inside an

Algorithm 2 Generating Leveled Bitmaps

```
1: Inputs: bitmaps $b_{colon}, b_{comma}, b_{lbracket}, b_{rbracket}$
2: Outputs: leveled bitmaps $l[0 \ldots k-1]$
3:
4: Procedure:
5: $level$ = -1
6: for each word $w$ in bitmaps do
7:     $w_{bracket} = w_{lbracket} \vee w_{rbracket}$
8:
9:     /* if no brackets, copy all bits from $w_{colon}$ or $w_{comma}$ */
10:    if $w_{bracket}$ == 0 then
11:        if $w_{colon} \neq 0$ then /* part of an object */
12:            $w_{l[level]} = w_{colon}$
13:        else/* part of an array */
14:            $w_{l[level]} = w_{comma}$
15:    else
16:        /* iterate over intervals separated by brackets */
17:        $w_{begin}$ = 1 /* beginning position of an interval */
18:        $done$ = false
19:        while $done$ == false do
20:            if $w_{bracket} \neq 0$ then /* locate next interval */
21:                $w_{end} = E(w_{bracket})$ /* end of an interval */
22:                $w_{bracket} = R(w_{bracket})$
23:                $w_{interval} = w_{end} - w_{begin}$
24:                $w_{begin} = w_{end}$
25:            else/* locate the last interval */
26:                $w_{end} = 2^{|w|-1}$ /* end of the word */
27:                $w_{interval} = (w_{end} - w_{begin}) \vee w_{end}$
28:                $done$ = true;
29:
30:            /* copy bits of this interval to leveled bitmap */
31:            if $w_{colon} \wedge w_{interval} \neq 0$ then
32:                $w_{l[level]} = w_{l[level]} \vee (w_{colon} \wedge w_{range})$
33:            else
34:                $w_{l[level]} = w_{l[level]} \vee (w_{comma} \wedge w_{range})$
35:
36:            /* update level based on left/right bracket */
37:            if $w_{end} \wedge w_{lbracket} > 0$ then
38:                $level = level + 1$
39:            if $w_{end} \wedge w_{rbracket} > 0$ then
40:                $level = level - 1$
41: return $l[0 \ldots k-1]$
```

object and members inside an array. The inputs to the algorithm include bitmaps of colon, comma, left bracket, and right bracket. Note that $b_{lbracket}$ combines bitmaps of { and [, and $b_{rbracket}$ combines bitmaps of } and]. Since both kinds of brackets add levels, there is no need to distinguish between them (the same as in [44]). The outputs of the algorithm are leveled bitmaps, each of which consists of the 1s from the colon and comma bitmaps in the corresponding level.

Initially, the algorithm marks the current `level` to -1 (Line 5). Then, it iterates over the bitmaps word by word. For better efficiency, it combines two words $w_{lbracket}$ and $w_{rbracket}$ into $w_{bracket}$

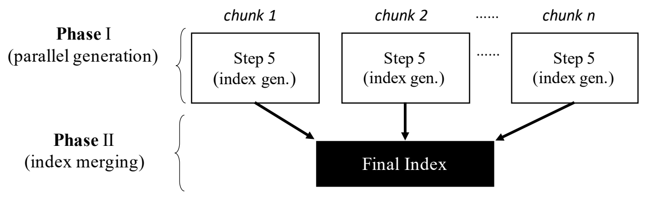


Figure 8: Two-Phase Index Construction.

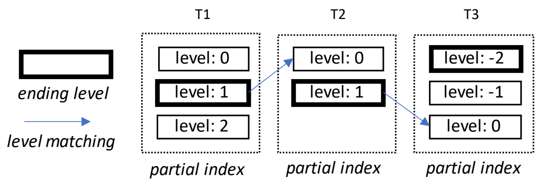


Figure 9: Level Alignment.

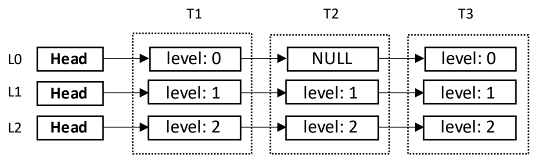
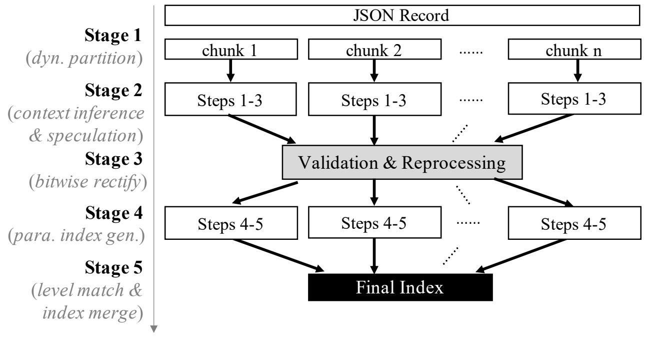


Figure 10: Index Merging.



JSON Record
Stage 1 (dyn. partition)
chunk 1
chunk 2
chunk n
Stage 2 (context inference & speculation)
Steps 1-3
Steps 1-3
Steps 1-3
Stage 3 (bitwise rectify)
Validation & Reprocessing
Stage 4 (para. index gen.)
Steps 4-5
Steps 4-5
Steps 4-5
Stage 5 (level match & index merge)
Final Index

Figure 11: Workflow of Data-Parallel Construction.

(Line 7). If $w_{bracket}$ contains only 0s (no brackets), it simply copies the current word of colon/comma bitmap to the corresponding leveled bitmap (Line 10-14); Otherwise, the algorithm iterates over all the intervals separated by two consecutive brackets. Line 21-28 are to find the current interval. Here, the algorithm uses functions E and R defined in Mison [44] to extract the right most bracket and remove it, respectively. Once the interval is located, the algorithm copies the word from colon/comma bitmap to the same interval of the corresponding leveled bitmap (Line 31-34). Finally, it updates the current `level` based on the bracket is left or right. The same process repeats until all the words of the bitmaps are processed.

Next, we show that the new design of Step 5 can be parallelized based on a reduction-style parallelism. Note that, in theory, the prior design of Step 5 [44] might also be parallelized in a similar fashion, but the reduction in that case would become more involved for its stack-based design and the efficiency may suffer even more with multiple threads, given its intensive memory operations.

3.5.2 Reduction and Index Merging. Partitioning a JSON record makes it difficult to tell at which level the beginning of a chunk is. To "break" the level dependence among JSON chunks, we adopt a *reduction*-style parallelization with two phases: (i) parallel leveled index generation and (ii) index merging, as illustrated in Figure 8.

In the first phase, all JSON chunks are processed in parallel under the assumption that they all start from Level 0. The outputs of this phase are the partial leveled bitmaps for individual JSON chunks, as depicted in Figure 9. After all chunks have been processed, the construction enters into the second phase, where levels of different chunks are aligned with those in the prior chunks one by one – the ending level of chunk i is the beginning level of chunk $i + 1$. In the example in Figure 9, thread T1 ends at Level 1, which is aligned with Level 0 of thread T2. As a result, the original Level 1 of thread T2 becomes Level 2, which is then aligned with Level 0 of thread T3. Thus, levels -2 and -1 of T3 turn into levels 0 and 1, respectively.

After adjusting the levels, the partial bitmaps are connected based on their actual levels, forming an array of linked lists, as shown in Figure 10. Note that the outputs from parallel index construction are slightly different from those in serial index construction, where each leveled bitmap is a single array. In principle, this difference may increase the cost of bitmap accessing. However, since the number of chunks (i.e., #CPU cores) is relatively very small comparing to the number of bits, this extra cost is small (within 2%).

In summary, we address multiple types of dependences involved in the structural index construction with an assembly of customized parallelization techniques. Next, we integrate them.

3.6 Putting it All Together

Figure 11 shows the high-level workflow of data-parallel structural index construction. From the top to the bottom, the construction process is divided into five stages. The first stage partitions the JSON record into chunks with *dynamic partitioning* to avoid breaking any keywords and backlash sequences. By default, the number of chunks n is set to the number available CPU cores. After partitioning, the construction leverages the *contradiction-based context inference* to find out if each chunk starts inside a string or not. When some inferences fail, *speculation* is immediately triggered to process the corresponding chunks speculatively. If speculation is enabled for at least on one JSON chunk, the construction would next enter into Stages 2, 3, and 4, one by one; Otherwise, it skips Stage 3 and combines Stages 2 and 4 (i.e., all five steps are completed without synchronization). Before entering Stage 3, all the parallel executions of different chunks (in Stage 2) should have been finished, that is, a `barrier` is required. In Stage 3, chunks with speculation are validated *in order* based on the actual string statuses (i.e., inside a string or not). When some misspeculation is detected, the corresponding input chunk would be reprocessed (only Step 3). For fast reprocessing, *bitwise rectification* would be applied. The outputs of after Stage 4 are partial structural indices built for each input chunk. Finally, Stage 5, performs the index level matching and merging to produce the final structural indices.

4 LOCALITY OPTIMIZATION

So far, our design of parallel structural index construction (including a new Step 5) is performed step by step. In each step, it traverses the JSON record and/or its (intermediate) bitmaps entirely. Since steps share the access of some bitmaps (e.g., one step writes to it and another step reads from it), repetitively traversing them cause poor data locality. This issue might less be a concern for small JSON records, but could become serious when processing large records, as the memory footprint can easily exceed cache capacities.

Instead of constructing the bitmaps step by step, in fact, we can build them word by word [3], where a word consists of only a few bytes (e.g., 8 bytes on 64-bit machines). Figure 12 illustrates the two granularities of bitmap construction.

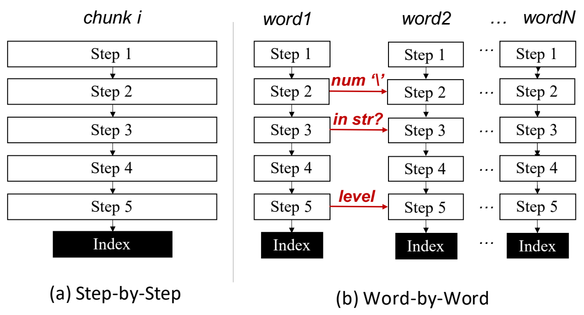


Figure 12: Index Construction Granularities.

To implement word-by-word bitmap construction, the algorithm has to generate and consume bitmaps partially and incrementally. This requires recording some "context" of each step between two adjacent words. For example, after finishing one word at Step 2, the algorithm needs to record how many consecutive \s have been observed by the end of the current word. Despite these extra work of "bookkeeping", the benefits of word-by-word processing are significant for bulky JSON records, as we will report later.

5 QUERYING USING INDEX

In this section, we first present the APIs for accessing the bitmaps and demonstrate their usage in evaluating common JSON path queries, then we further discuss the strategies for enabling parallel query evaluation on bulky JSON records.

APIs. To simplify the programming, we hide the low-level bitmap traversal details into a set of high-level JSON data accessing APIs that are similar to the existing tools [43]. The APIs include one *BitmapConstructor* class for generating leveled bitmaps in parallel and one *BitmapIterator* class for navigating through the leveled bitmaps and locating keys in objects or elements in arrays.

Algorithm 3 demonstrates an example usage of these APIs for evaluating query `$.user[0].name`. Note that even though the bitmap indices are constructed in parallel (Line 6), the bitmap traversal

Algorithm 3 API Usage Example (query `$.user[0].name`)

```
1:  Procedure:
2:  Inputs: $input$ : JSON record(s)
3:  Outputs: $result$: a list of matched JSON contents
4:
5:  $BitmapConstructor\ bc$ = new $BitmapConstructor()$
6:  $Bitmap\ bm = bc.construct(input, 16)$ /* with 16 threads */
7:  $BitmapIterator\ iter = bc.getIterator(bm)$
8:  if $iter.isObject()$ && $iter.moveToKey($"user"$)$ then
9:      $iter.down()$ /* value of "user" */
10:     if $iter.isArray()$ && $iter.moveToIndex(0)$ then
11:         $iter.down()$ /* "user[0]" */
12:         if $iter.isObject()$ && $iter.moveToKey($"name"$)$ then
13:             $result = getValue()$
```

(Line 8-13) remains serial. To make JSON analytics scalable, we next describe a strategy that enables parallel bitmap traversals.

Parallel Query Evaluation. Given the indices of many small JSON records, we can easily traverse them in parallel and evaluate the query on different records independently. However, for a bulky JSON record, it is non-trivial to traverse its indices in parallel. Our parallelization of the index traversal stems from a simple yet critical observation [4] – a bulky JSON record usually consists of a JSON array at an upper level (closer to the root level) which dominates the size of the JSON records and divides lower levels of the JSON record into many smaller elements. We refer to such an array as the *dominating array*. Based on this observation, we propose to first locate the index level of the dominating array, from where we then traverse different elements in the array in parallel. The criteria for defining the dominating array are configurable. In our experiments, we require the array to occupy 80% of the whole record in size and consist of at least 256 elements. Elements of the dominating array are processed using a thread pool where each thread gets a copy of the current bitmap iterator and proceeds independently. For better load balancing, elements of the dominating array are inserted into a *worklist* and then consumed on demand by worker threads.

6 EVALUATION

This section presents evaluation results of the proposed techniques, with a focus on the parallel performance on bulky JSON records.

6.1 Methodology

We implemented the parallel JSON structure index constructor in C++, namely Pison, and used Pthread for its parallelization. We compare Pison with the existing solution Mison [44]. As Mison is not publicly available, we use a third-party implementation of Mison, called Pikkr [7], as well as our own implementation of Mison for this comparison. Moreover, we also implemented an optimized version of Mison, denoted as Mison+, where Steps 2 and 3 are implemented using ideas from simdjson [43] (see Section 2).

In addition, we compare Pison with simdjson [43] – a popular SIMD-based JSON processing tool, RapidJSON [11] – a popular JSON parser based on character by character processing, as well as JPStream [38] – a streaming-based JSON tool. Table 1 lists all the methods in our evaluation.

We evaluate Pison for processing a sequence of small JSON records and individual bulky JSON records in terms of processing time and memory consumption. Table 2 reports the statistics of

[3]Note that word-by-word processing could also be adopted in Mison [44].

[4]According to real-world datasets [1, 3, 9, 10, 13, 16].

Table 1: Methods in Evaluation

Method	Brief Description
simdjson	A SIMD-based JSON parser [43]
CMison	Our C++ implementation of Mison [44]
Mison+	Improved Mison (Steps 2-3) based on simdjson [43]
Pikkr	Third-party implementation of Mison in Rust [7]
RapidJSON	A JSON parser in C++ from Tencent [11]
JPStream	A parallel streaming JSON processor in C [38]
Pison(SbS)	Pison with step-by-step processing (this work)
Pison(WbW)	Pison with word-by-word processing (this work)

JSON datasets used in our evaluation. These include Best Buy (BB) product dataset [1], tweets from Twitter (TT) developer API [13], Google Maps Directions (GMD) dataset [3], National Statistics Postcode Lookup (NSPL) dataset for UK [9], Walmart (WM) product dataset [10], and Wikipedia (WP) entity dataset [17]. The default size of each dataset is approximately 1GB for easy comparison. Each dataset forms a single large JSON record. To create scenarios of small records processing, we manually extracted the dominating array from each dataset, broke it into smaller records, and inserted a new line after each small record – a common way to organize small JSON records. The number of small records for each dataset is shown in the column #subrec. in Table 2.

Table 2: Dataset Statistics

Data	#objects	#arrays	#K-V	#prim.	#subrec.	depth
TT	2.39M	2.29M	26.5M	24.3M	150K	11
BB	1.91M	4.88M	40.7M	35.8M	230K	7
GMD	10.3M	43K	29.0M	21.0M	4.44K	9
NSPL	613	3.50M	1.66K	84.2M	1.74M	9
WM	333K	34K	8.19M	9.92K	275K	4
WP	17.3M	6.53M	53.2M	35.0M	137K	12

To evaluate the querying performance, we first include 8 queries used in Mison [44] on Twitter dataset (TT1-TT8), then for each other dataset, we created a JSONPath query, as shown in Table 3. Note that 8 of out the 13 queries consist of two subqueries. For bulky records made of small records, we add a prefix `[*].` to each of its queries. Together, they cover common patterns of path queries, as well as queries of different complexities and selectiveness.

All experiments were conducted on a 16-core machine equipped with two Intel 2.1GHz Xeon E5-2620 v4 CPUs and 64GB RAM. The CPUs support 64-bit ALU instructions, 256-bit SIMD instruction set, and the carry-less multiplication instruction (`pclmulqdq`). Both servers run on CentOS 7 and are installed with G++ 7.4.0 and JDK 1.8.0_191. All C++ programs are compiled with "-O3" optimization flag. In the case of small records, they are stored in a single array with the beginning position of each record stored in a separate `offset` array. The timing results are the average of 10 runs; no 95% confidence interval is shown when the variation is not significant.

Next, we first report the performance of serial and parallel index constructions of different methods.

Table 3: JSONPath Queries

Queries TT1-TT8 are from Mison [44]

ID	Query structure	#matches	#visited fields
TT1	{ ur.id }	150,135	1,979,268
TT2	{ ur.id, rtct }	300,270	3,076,477
TT3	{ ur.id, ur.la }	300,270	4,681,698
TT4	{ ur.nm, rp }	300,270	2,279,538
TT5	{ ur.la, la }	300,270	6,767,260
TT6	{ id, rtst }	252,524	3,243,333
TT7	{ id, en.urls[*].url }	239,016	3,615,763
TT8	{ id, en.urls[*].idc[*] }	327,897	3,862,762
BB	{ pd[*].cp[1:3].id }	459,332	7,362,758
GMD	{ rt[*].lg[*].st[*].dt.tx }	1,716,752	5,215,576
NSPL	{ mt.vw.co[*].nm }	44	121
WM	{ it[*].bmpr.pr, it[*].nm }	288,391	7,867,449
WP	{ cl.P[*].ms.pty }	15,603	1,974,693

6.2 Index Construction Performance

Figure 13 reports the time of index construction for bulky JSON records using different methods. We exclude JPStream since it does not construct any indices. Among them, RapidJSON and simdjson create parse trees while the other methods create structural indices.

RapidJSON vs. Others. First, according to the results, RapidJSON takes substantially longer to construct indices than other evaluated methods, even though it is known for its superior performance than many other popular JSON parsers (e.g., FastJSON [2]). The reason is that RapidJSON does not leverage bitwise parallelism and SIMD instructions. The results confirm the effectiveness of leveraging fine-grained parallelism in JSON data processing.

Mison vs. simdjson vs. Serial Pison. For methods with bitwise parallelism and SIMD supports, the two versions of Mison (Pikkr and CMison) take slightly longer time. In comparison, Mison+ runs faster than the prior two, by 31% and 11%, thanks to its optimized Steps 2 and 3 based on simdjson. Following these, simdjson and serial Pison(SbS) show similar performance on average. Note that the only difference between Mison+ and serial Pison(SbS) is the new design of Step 5 (Section 3.5.1). Therefore, the time difference between the two shows the benefits of this new design – 1.41X speedup on average, which is substantial for an optimization of one step. Finally, serial Pison(WbW) performs the best among all the serial methods. The performance gap between Pison(WbW) and Pison(SbS) indicates the benefits of locality optimization (Section 4), which is 1.67X speedup on average, demonstrating the importance of the proposed locality optimization. Adding the benefits of the new design of Step 5 and the locality optimization together, we find that serial Pison(WbW) runs 3.1X faster than Pikkr, 2.6X faster than CMison, and 2.3X faster than Mison+.

So far, the comparison are among only serial methods. Next, we discuss the performance of parallel Pison – the main contribution of this work. Note that we cannot run other methods in parallel due to the lack of parallelization.

Parallel Pison vs. Others. First, running two versions of Pison (SbS and WbW) in parallel with eight threads, achieves 3.3X and 3.8X speedups over the serial counterparts, respectively. The sub-linear speedups indicate that the structural index construction is not only computation-intensive, but also memory-intensive; while

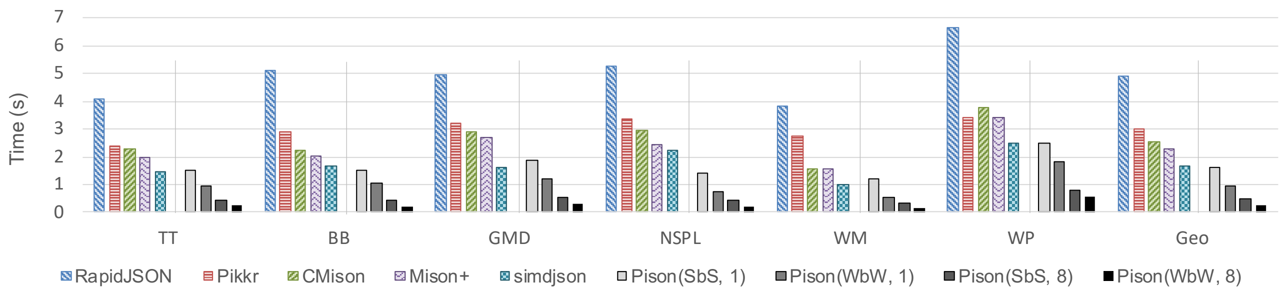


Figure 13: Comparison of Index Construction Time of Different Methods on Bulky JSON Records.

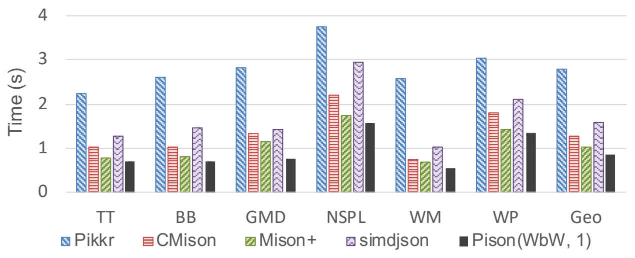


Figure 14: Comparison of Index Construction Time of Different Methods on Small JSON Records.

Table 4: Time Breakdown by Steps

Entry is [seq. time(s) : para. time(s) w/ 8 threads] of Pison(SbS)

	Step 1	Step 2	Step 3	Step 4	Step 5
TT	0.48 : 0.14	0.03 : 0.01	0.06 : 0.01	0.12 : 0.05	0.83 : 0.25
BB	0.59 : 0.18	0.05 : 0.01	0.06 : 0.01	0.15 : 0.06	0.65 : 0.20
GMD	0.65 : 0.19	0.05 : 0.02	0.06 : 0.01	0.21 : 0.06	0.92 : 0.27
NSPL	0.65 : 0.18	0.06 : 0.03	0.07 : 0.02	0.20 : 0.06	0.42 : 0.15
WM	0.59 : 0.17	0.06 : 0.01	0.08 : 0.01	0.19 : 0.06	0.27 : 0.10
WP	0.67 : 0.19	0.06 : 0.01	0.06 : 0.01	0.15 : 0.07	1.55 : 0.51
Geo SP	3.55X	5.04X	6.54X	2.83X	3.04X

the parallelization helps address the former, the performance could be limited by the latter. Despite this limitation, when compared to the other indexing methods, parallel Pison still shows significant improvements – 19X faster than RapidJSON, 11.6X faster than Pikkr, 9.8X faster than CMison, and 8.8X faster than Mison+.

So far, the performance results are for building indices on bulky JSON records – the focus of Pison. For completeness, we also report the performance of small record indexing.

Small Record Indexing. For many small records, parallelism can be easily achieved at the task level, we thus only report the serial performance of Pison for fairness. Figure 14 shows the performance results of different methods, which are consistent with those in large record indexing, except that the two implementations of Mison and its optimized version Mison+ run relatively faster than they do in large record indexing. The reason is that when the records are small, they can easily fit into the caches; the locality of their step-by-step processing gets much improved.

Next, we break down the benefits of parallelization by steps and evaluate the effectiveness of the parallelization techniques in detail.

6.3 Benefits and Costs Breakdown

Time Breakdown by Steps. Table 4 reports the sequential and parallel execution times of for each step in the structural index construction, as well as the averaged parallelization speedup of each step. The results show that parallelization improvements vary among steps. Step 3 achieves the highest speedup – 6.5X, while Step 4 achieves the lowest – 2.8X. As discussed earlier, the variation of benefits mainly depends on the memory-computation ratio. Step 3 is relatively more complex computation-wise, meanwhile only writes results to one bitmap (see Figure 2). By contrast, Step 4 only involves a single bitwise operation (ANDNOT), but needs to write to six bitmaps. As a result, Step 4 is more memory-bound than Step 3, leading to less speedups. Similar reasoning also holds to the other three steps. After reporting the time and speedups, we next further evaluate the main parallelization techniques used in different steps.

Context Inference. To parallelize Step 3, we proposed contradiction-based context inference to find if a chunk starts inside a string. To confirm its effectiveness, we profiled the success rate of context inferences in all the parallel executions reported in Figure 13. The results show that all the inferences succeeded, which indicates that a contradiction is derived in all the evaluated cases.

Speculation. Since no context inferences failed in evaluation, we artificially failed a few context inferences when 8 threads are used to examine the performance under speculative mode. Thus, 20 threads entered into the speculative mode across 6 datasets. Interestingly, none of them failed, which demonstrates the effectiveness of the heuristic used by the speculation – usually characters inside a string do not form valid tokens (see Section 3.4). Despite the high speculation accuracy, the performance suffers from another aspect – loss of data locality. As discussed in Section 3.5.2, once entering speculative mode, the validation stage would isolate the word-by-word optimizations within Steps 1-3 and Steps 4-5. Figure 15 shows the cost of speculation – reducing speedups from 3.76X to 2.45X.

In principle, misspeculation can still happen, in which cases there are costs of recovering. To measure these costs, we manually flipped the speculated string status for 1 and 2 speculative threads, respectively. Figure 15 reports the costs of reprocessing. With 2 cases of misspeculation, the speedup only drops from 2.45X to 2.33X. This is due to the bitwise rectification introduced in Section 3.4.

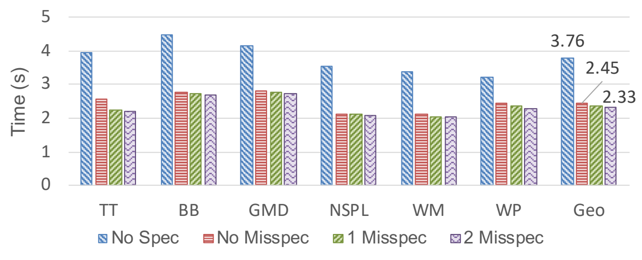


Figure 15: Costs of Speculation and Recovering.

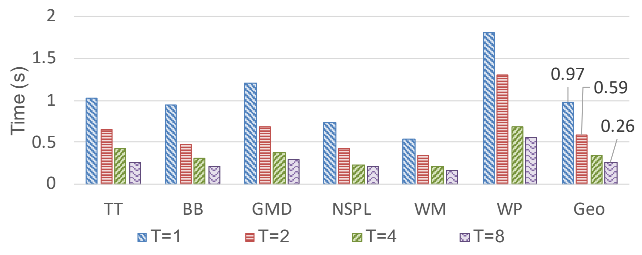


Figure 16: Scalability over Number of Threads.

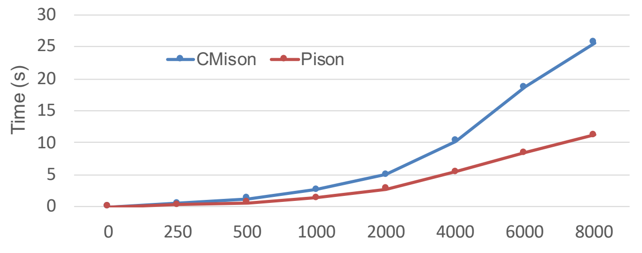


Figure 17: Scalability over Record Size.

Without this technique, we have to reprocess the chunk, in which case the cost would be quite significant.

Costs of Index Merging. The costs associated with index merging are of two types: (i) the cost of index merging – the second phase of the two-phase parallel index construction and (ii) the extra cost of accessing of merged (linked) indices. For the former, we profiled the cost of each phase and found that the merging cost is less than 0.01% for all test cases. This is because the index merging only needs to connect the partial indices of different chunks together (see Figure 10), including the adjustment of index levels (Figure 9), both of which are low-cost operations. For the latter, our profiling results show that overhead of accessing linked indices is 1.5% on average, comparing to the single-array indices from serial construction.

Scalability. Figure 16 reports the parallel performance of Pison with different numbers of threads. The results show that additional benefits from more threads diminish slightly as the number of threads grows. This is due to that the structural index construction is both computation and memory intensive. As computations get parallelized with more threads, the memory bandwidth gets more saturated, limiting the performance benefits.

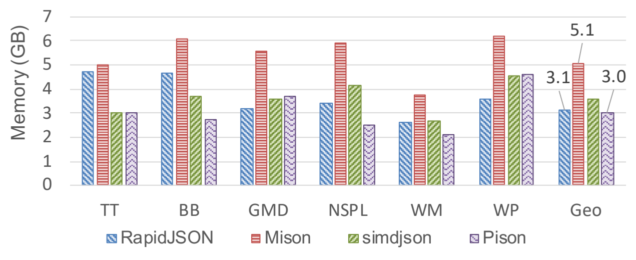


Figure 18: Comparison of Memory Consumption

Table 5: End-to-End Time (s) Comparison.

	JPS(8)	RapidJ	Mison+(1)	simdjson	Pison(1)	Pison(8)
TT	6.64	4.73	4.92	2.13	4.10	0.86
BB	1	5.27	2.61	1.84	1.55	0.34
GMD	1.01	5.18	3.64	1.88	2.32	0.58
NSPL	1.02	5.26	2.46	2.22	0.76	0.22
WM	0.43	3.98	2.14	1.19	1.2	0.25
WP	1.38	6.7	3.67	2.64	2.21	0.6
Geo	1.26	5.12	3.11	1.93	1.76	0.42
Sum	11.48	31.12	19.44	11.9	12.14	2.85

To better understand how Pison and Mison scale as the record size increases, we further vary the size of the record in the BB dataset from 256MB to 8GB. Figure 17 shows the scalabilities of Mison and Pison as the record size increases. As the trend indicates, the larger the records are, the more time saving Pison provides comparing to Mison.

Memory Consumption. The other concern in processing large records is the memory consumption. As shown in Figure 18, the memory footprint of Pison is about 3.0GB on average, which is the least among the indexing or parsing-based methods. Comparing to step-by-step processing, the word-by-word processing of Pison eliminates the needs of storing large intermediate bitmaps, resulting in 40.8% less memory footprint. This figure does not include the memory footprint of JPStream, which in fact is configurable (set it to 1GB) thanks to its streaming-style processing strategy.

6.4 End-to-End Performance

Finally, we evaluate end-to-end performance that includes both index construction and query evaluations. The queries used for this evaluation are from Table 3.

Table 5 reports the end-to-end performance of different methods under evaluation. Except for JPStream, the time for all the other methods include both the index construction time and the querying time. For bulky JSON records, we cache the indices and reuse them for multiple queries. For dataset TT, this means the 8 queries share the constructed indices. In the case of JPStream, a streaming-based method, its execution time is only about the querying.

JPStream vs. Others. First, the performance results of JPStream shows a large gap (over 5.2X) between the case of TT and cases of other datasets. In fact, JPStream performs the worst in the case of TT among all evaluated methods. The reason is that, for TT, multiple individual queries are evaluated. Without any indices, the evaluation has to traverse the entire raw dataset once for each query, causing repeated overhead, which is expected as JPStream is designed for streaming scenarios. For other datasets where a single query is evaluated, JPStream clearly shows better performance than other methods, except for parallel Pison. On average, parallel Pison runs 3X faster than JPStream when both running with 8 threads.

Parallel Pison vs. Others. Among the indexing-based methods, parallel Pison achieves 12.2X speedup over RapidJSON, 7.4X speedup over Mison+, 4.6X speedup over simdjson, and 4.2X over its own serial version. Note that for Mison+, we do not have the parallel implementation for its querying. However, we find that its indexing time alone (2.28s) already takes significantly longer than the total running time of parallel Pison (0.42s).

In summary, the above results confirm that the efficiency of parallel Pison, showing substantial performance boosts over a set of state-of-the-art JSON processing tools.

7 RELATED WORK

This section discusses existing research on raw semi-structured data processing and the parallelization of large-record processing.

Raw Semi-structured Data Processing. There is a rich body of research on processing raw semi-structured data such as XML and JSON. Typical solutions often involve in some forms of automata [21, 27, 62] and stacks [25, 41], which are essential in recognizing the nesting structures and matching the queries. Most existing JSON tools [2, 6, 8, 11] follow this direction and convert the JSON data stream into in-memory tree structures before querying. However, it takes time and memory to generate the parsing trees [38]. One way to avoid the cost of parsing is to adopt streaming schemes [12, 31, 38, 40, 52]. For instance, JSONSurfer [12] directly evaluates path queries without any pre-parsing of the JSON data. However, it is inefficient in performing the query matching due to a lack of the capability of tracking the matching status. JPStream [38] improves this by compiles a set of JSONPath queries and JSON syntax into a dual stack pushdown automaton which records both the parsing status and the matching status of queries.

One common limitation with the above methods is the "one-character-each-time" processing strategy. To deal with this limitation, Mison [44] proposes to build structural indices with bitwise operations, which can process tens of or even hundreds of characters simultaneously with the help of bitwise and SIMD-level parallelism. This idea originates from NoDB [19, 20, 37, 42], which builds structural index on raw CSV files and adaptively uses the index to load the data. In fact, simdjson [43] also adopts this bitwise processing in its first stage of JSON parsing. Besides JSON data, the idea of bitwise processing is also seen in other contexts, including regular expression matching [23], XML parsing [46], as well as some database systems [18, 22, 26, 51]. Based on bitwise processing, Sparser [53] applies filtering before performing the parsing to further accelerate the processing.

Parallel Processing of Large Records. In addition to the exploration of bitwise and SIMD-level parallelism, it is also important to exploit coarse-grained parallelism (such as multicores) in the processing of individual large records. In fact, many efforts have been put into the parallelization of XML stream processing [39, 48, 52, 54, 58], including the use of hardware accelerators [49, 50]. The key in enabling parallel processing of individual XML record is “breaking” the dependences in the data processing. For example, in [48, 54], a pre-scan is applied to the XML record first to partition it according its high-level structures. In another work [52], the authors design parallel pushdown transducers that numerate all the possible states at the beginning of an arbitrary XML partition to break the state dependences. To reduce the cost of state enumeration, GAP [39] leverages the XML grammar to prune infeasible states. By contrast, there are few studies in the parallelization of JSON stream processing. JPStream [38] adopts ideas from parallel XML processing [39, 52] to JSON processing so that an individual large JSON record can be effectively processed in parallel. This shares a similar goal with Pison.

Besides the above work, there are also prior studies that design speculative parallelization for parallel processing other types of raw data [56, 59, 63–65]. For example, HPar [63] proposes speculative parsing of HTML documents. In [56, 64, 65], FSMs are executed speculatively to perform pattern matching over unstructured textual data. Recently, Parparaw [59] leverages speculation techniques to process delimiter-separated raw data in parallel, and PBS [55] speculatively parallelizes bitstream processing by modeling it into FSM computations. As far as we know, Pison is the first work that leverages speculation for parallel indexing of JSON data.

8 CONCLUSIONS

Constructing structural indices for JSON data has shown promises in accelerating JSON analytics, but its serial design makes it difficult to scale to large and complex JSON records. This work addresses this challenge by introducing intra-record parallelism and redesigning the structural indices construction process. It proposes an assembly of parallelization techniques that make it possible to construct the structural indices of individual JSON records in parallel. Evaluation on datasets collected from real-world applications show that the developed system – Pison, surpasses the performance of state-of-the-art tools, including simdjson and Mison, in both small-record and large-record processing scenarios.

ACKNOWLEDGMENTS

We would like to thank all anonymous reviewers for their time and constructive feedback. This material is based upon the work supported in part by National Science Foundation (NSF) Grants No. 1751392.

REFERENCES

[1] [n.d.]. Best Buy Developer API. https://bestbuyapis.github.io/api-documentation/. Retrieved: 2019-05-01.
[2] [n.d.]. A fast JSON parser/generator for Java. https://github.com/alibaba/fastjson/. Retrieved: 2020-05-01.
[3] [n.d.]. Google Map Directions API. https://developers.google.com/maps/documentation/directions/start/. Retrieved: 2019-07-01.
[4] [n.d.]. The home of the U.S. Government’s open data. https://www.data.gov/. Retrieved: 2019-07-01.
[5] [n.d.]. A Java serialization/deserialization library to convert Java Objects into JSON and back. https://github.com/google/gson. Retrieved: 2020-05-10.
[6] [n.d.]. JSON-C - A JSON implementation in C. https://github.com/json-c/json-c/. Retrieved: 2020-05-01.
[7] [n.d.]. JSON parser which picks up values directly without performing tokenization in Rust. https://github.com/pikkr/pikkr. Retrieved: 2020-05-10.
[8] [n.d.]. Main Portal page for the Jackson project. https://github.com/FasterXML/jackson/. Retrieved: 2020-05-01.

[9] [n.d.]. National Statistics Postcode Lookup UK. https://data.gov.uk/dataset/national-statistics-postcode-lookup-uk/. Retrieved: 2019-07-01.
[10] [n.d.]. Product Lookup API. https://developer.walmartlabs.com/docs. Retrieved: 2019-05-10.
[11] [n.d.]. RapidJSON. http://rapidjson.org/. Retrieved: 2020-05-01.
[12] [n.d.]. A streaming JsonPath processor in Java. https://github.com/jsurfer/JsonSurfer/. Retrieved: 2020-05-01.
[13] [n.d.]. Twitter Developer API. https://developer.twitter.com/en/docs/. Retrieved: 2019-07-01.
[14] [n.d.]. Twitter Usage Statistics. http://www.internetlivestats.com/twitter-statistics/. Retrieved: 2019-07-01.
[15] [n.d.]. Using the Graph API. https://developers.facebook.com/docs/graph-api/using-graph-api/. Retrieved: 2019-05-10.
[16] [n.d.]. Wikidata:Database download. https://www.wikidata.org/wiki/Wikidata:Database_download. Retrieved: 2019-05-10.
[17] [n.d.]. Wikimedia Miscellaneous Files. https://archive.org/details/wikidata-json-20150202. Retrieved: 2020-05-20.
[18] Azza Abouzied, Daniel J Abadi, and Avi Silberschatz. 2013. Invisible loading: access-driven data transfer from raw files into database systems. In *Proceedings of the 16th International Conference on Extending Database Technology*. 1–10.
[19] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2012. NoDB: efficient query execution on raw data files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 241–252.
[20] Ioannis Alagiannis, Renata Borovica-Gajic, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2015. NoDB: efficient query execution on raw data files. *Commun. ACM* 58, 12 (2015), 112–121.
[21] Iliana Avila-Campillo, Todd J Green, Ashish Gupta, Makoto Onizuka, Demian Raven, and Dan Suciu. 2002. XMLTK: An XML toolkit for scalable XML stream processing. *Database Research Group (CIS)* (2002), 2.
[22] Spyros Blanas, Kesheng Wu, Surendra Byna, Bin Dong, and Arie Shoshani. 2014. Parallel data analysis directly on scientific file formats. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 385–396.
[23] Robert D Cameron, Thomas C Shermer, Arrvindh Shriraman, Kenneth S Herdy, Dan Lin, Benjamin R Hull, and Meng Lin. 2014. Bitwise data parallelism in regular expression matching. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 139–150.
[24] Roy H. Campbell and Reza Farivar. [n.d.]. Cloud Computing Applications, Part 1: Cloud Systems and Infrastructure. https://www.coursera.org/learn/cloud-applications-part1. Retrieved: 2019-07-01.
[25] Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K Selçuk Candan. 2006. Twig2Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proceedings of the 32nd international conference on Very large data bases*. 283–294.
[26] Yu Cheng and Florin Rusu. 2014. Parallel in-situ data processing with speculative loading. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1287–1298.
[27] Yanlei Diao, Peter Fischer, Michael J Franklin, and Raymond To. 2002. Yfilter: Efficient and scalable filtering of XML documents. In *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE, 341–342.
[28] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*. Springer, 195–216.
[29] W3C Editor. [n.d.]. Overview of the CoverageJSON format. https://w3c.github.io/sdw/coverage-json/. Retrieved: 2019-07-01.
[30] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. 2019. Speculative distributed CSV data parsing for big data analytics. In *Proceedings of the 2019 International Conference on Management of Data*. 883–899.
[31] Todd J Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. 2003. Processing XML streams with deterministic automata. In *International Conference on Database Theory*. Springer, 173–189.
[32] GeoJSON Working Group. [n.d.]. The GeoJSON Specification (RFC 7946). http://geojson.org/. Retrieved: 2019-07-01.
[33] Netork Working Group. [n.d.]. NetJSON: data interchange format for networks. http://netjson.org/rfc.html. Retrieved: 2019-07-01.
[34] W3C JSON-LD Community Group. [n.d.]. JSON for Linking Data. https://json-ld.org/. Retrieved: 2019-07-01.
[35] Venkat N Gudivada, Dhana Rao, and Vijay V Raghavan. 2014. NoSQL systems for big data management. In *Services (SERVICES), 2014 IEEE World Congress on*. IEEE, 190–197.
[36] Stefan Gössner. [n.d.]. JSONPath - XPath for JSON. http://goessner.net/articles/JsonPath/. Retrieved: 2018-07-01.
[37] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. 2011. Here are my data files. here are my queries. where are my results?. In *Proceedings of 5th Biennial Conference on Innovative Data Systems Research*.
[38] Lin Jiang, Xiaofan Sun, Umar Farooq, and Zhijia Zhao. 2019. Scalable Processing of Contemporary Semi-Structured Data on Commodity Parallel Processors - A Compilation-based Approach. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 79–92. https://doi.org/10.1145/3297858.3304008
[39] Lin Jiang and Zhijia Zhao. 2017. Grammar-aware Parallelization for Scalable XPath Querying. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 371–383.
[40] Vanja Josifovski, Marcus Fontoura, and Attila Barta. 2005. Querying XML streams. *The VLDB Journal* 14, 2 (2005), 197–210.
[41] Arseny Kapoulkine. [n.d.]. pugixml: a Light-weight, simple and fast XML parser for C++ with XPath support. http://pugixml.org/. Retrieved: 2019-07-01.
[42] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. 2014. Adaptive query processing on RAW data. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1119–1130.
[43] Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. *The VLDB Journal* 28, 6 (2019), 941–960.
[44] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: A Fast JSON Parser for Data Analytics. *PVLDB* 10, 10 (2017), 1118–1129. https://doi.org/10.14778/3115404.3115416
[45] Yishan Li and Sathiamoorthy Manoharan. 2013. A performance comparison of SQL and NoSQL databases. In *Communications, computers and signal processing (PACRIM), 2013 IEEE pacific rim conference on*. IEEE, 15–19.
[46] Dan Lin, Nigel Medforth, Kenneth S. Herdy, Arrvindh Shriraman, and Robert D. Cameron. 2012. Parabix: Boosting the efficiency of text processing on commodity processors. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*. 373–384. https://doi.org/10.1109/HPCA.2012.6169041
[47] Logicworks. [n.d.]. The future of AWS' cloud: Infrastructure as an application. https://www.cloudcomputing-news.net/news/2016/jun/02/. Retrieved: 2019-07-01.
[48] Wei Lu and Dennis Gannon. 2007. Parallel XML processing by work stealing. In *Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*. ACM, 31–38.
[49] Abhishek Mitra, Marcos Vieira, Petko Bakalov, Walid Najjar, and Vassilis Tsotras. 2009. Boosting XML filtering with a scalable FPGA-based architecture. *arXiv preprint arXiv:0909.1781* (2009).
[50] Roger Moussalli, Robert J Halstead, Mariam Salloum, Walid A Najjar, and Vassilis J Tsotras. 2011. Efficient XML Path Filtering Using GPUs. In *ADMS@ VLDB*. Citeseer, 9–18.
[51] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. 2013. Instant loading for main memory databases. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1702–1713.
[52] Peter Ogden, David Thomas, and Peter Pietzuch. 2013. Scalable XML query processing using parallel pushdown transducers. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1738–1749.
[53] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2018. Filter Before You Parse: Faster Analytics on Raw Data with Sparser. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1576–1589.
[54] Y. Pan, W. Lu, Y. Zhang, and K. Chiu. 2007. A Static Load-Balancing Scheme for Parallel XML Parsing on Multicore CPUs. In *Proc. of the 7th International Symposium on Cluster Computing and the Grid (CCGRID)*.
[55] Junqiao Qiu, Lin Jiang, and Zhijia Zhao. 2020. Challenging Sequential Bitstream Processing via Principled Bitwise Speculation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 607–621.
[56] Junqiao Qiu, Zhijia Zhao, Bo Wu, Abhinav Vishnu, and Shuaiwen Leon Song. 2017. Enabling scalability-sensitive speculative parallelization for FSM computations. In *Proceedings of the International Conference on Supercomputing*. ACM, 2.
[57] James Reinders. [n.d.]. Intel AVX-512 Instructions. https://software.intel.com/en-us/articles/intel-avx-512-instructions. Retrieved: 2019-06-01.
[58] B. Shah, P. Rao, B. Moon, and M. Rajagopalan. 2009. A Data Parallel Algorithm for XML DOM Parsing. *Database and XML Technologies, Lecture Notes in Computer Science* 5679 (2009).
[59] Elias Stehle and Hans-Arno Jacobsen. 2020. ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data. *Proc. VLDB Endow.* 13, 5 (Jan. 2020), 616–628. https://doi.org/10.14778/3377369.3377372
[60] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference (10CCC), 2015 10th*. IEEE, 583–590.
[61] Philipp Wehner, Christina Piberger, and Diana Gohringer. 2014. Using JSON to manage communication between services in the Internet of Things. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*. IEEE, 1–4.
[62] Ying Zhang, Yinfei Pan, and Kenneth Chiu. 2010. A parallel xpath engine based on concurrent NFA execution. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*. IEEE, 314–321.

[63] Zhijia Zhao, Michael Bebenita, Dave Herman, Jianhua Sun, and Xipeng Shen. 2013. HPar: A practical parallel parser for HTML–taming HTML complexities for parallel parsing. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 4 (2013), 1–25.

[64] Zhijia Zhao and Xipeng Shen. 2015. On-the-Fly Principled Speculation for FSM Parallelization. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*. 619–630. https://doi.org/10.1145/2694344.2694369

[65] Zhijia Zhao, Bo Wu, and Xipeng Shen. 2014. Challenging the "embarrassingly sequential": parallelizing finite state machine-based computations through principled speculation. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*. 543–558. https://doi.org/10.1145/2541940.2541989