ETH zürich

ETH Library

Mison: A Fast JSON Parser for Data Analytics

Conference Paper

Author(s):
Li, Yinan; Katsipoulakis, Nikos; Chandramouli, Badrish; Goldstein, Jonathan; Kossman, Donald

Publication date:
2017-06

Permanent link:
https://doi.org/10.3929/ethz-b-000234616

Rights / license:
Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Mison: A Fast JSON Parser for Data Analytics

Yinan Li† Nikos R. Katsipoulakis‡* Badrish Chandramouli†
Jonathan Goldstein† Donald Kossmann†

†Microsoft Research
{yinali, badrishc, jongold, donaldk}@microsoft.com

‡University of Pittsburgh
katsip@cs.pitt.edu

ABSTRACT

The growing popularity of the JSON format has fueled increased interest in loading and processing JSON data within analytical data processing systems. However, in many applications, JSON parsing dominates performance and cost. In this paper, we present a new JSON parser called Mison that is particularly tailored to this class of applications, by pushing down both projection and filter operators of analytical queries into the parser. To achieve these features, we propose to deviate from the traditional approach of building parsers using finite state machines (FSMs). Instead, we follow a two-level approach that enables the parser to jump directly to the correct position of a queried field without having to perform expensive tokenizing steps to find the field. At the upper level, Mison speculatively predicts the logical locations of queried fields based on previously seen patterns in a dataset. At the lower level, Mison builds structural indices on JSON data to map logical locations to physical locations. Unlike all existing FSM-based parsers, building structural indices converts control flow into data flow, thereby largely eliminating inherently unpredictable branches in the program and exploiting the parallelism available in modern processors. We experimentally evaluate Mison using representative real-world JSON datasets and the TPC-H benchmark, and show that Mison produces significant performance benefits over the best existing JSON parsers; in some cases, the performance improvement is over one order of magnitude.

1. INTRODUCTION

JSON is a highly popular data format. It is used in most Web applications (e.g., Twitter and Facebook) and is quickly gaining acceptance as a schema-free data exchange format for enterprise applications. JSON is simple, flexible, and has high expressive power. For instance, JSON is a great way to represent bags and nested objects. The rising popularity of JSON has fueled increased interest in running analytical queries on JSON data. To meet this growing need, many data analytics engines (e.g., Apache Spark [33], Drill [2], Storm [32], and Jaql [12]) natively support the JSON data.

*Work performed during internship at Microsoft Research.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 10, No. 10
Copyright 2017 VLDB Endowment 2150-8097/17/06.

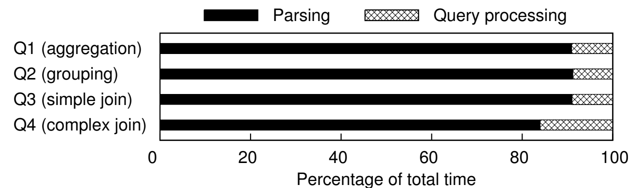


Figure 1: Parsing vs. Query processing Cost
Twitter Dataset, Queries from [30], Spark+Jackson

Nevertheless, JSON is a *raw data format*. That is, JSON data must be parsed before it can be further processed or analyzed. Unfortunately, parsing JSON data is expensive. As a result, with the current state of the art, JSON data needs to be parsed and loaded into a data processing engine in binary format before it can be analyzed efficiently. To demonstrate this observation, Figure 1 shows the results of an experiment running the queries from [30] on raw JSON data from Twitter using Spark, which internally uses the Jackson JSON parser [5]. It becomes clear that even for complex queries that involve joins and aggregations, the total cost of a query is dominated ($> 80\%$) by parsing the raw JSON data.

This paper presents a new JSON parser, called Mison. Mison is typically one order of magnitude faster than any existing JSON parser. Mison is general-purpose and supports a wide range of applications including analytics on JSON data, real-time streaming JSON data, and processing JSON messages at client machines.

How can Mison be so fast? Existing parsers such as Jackson [5] and Gson [4] are mature and have been optimized and tuned for many years. These parsers are based on *finite state machines (FSM)* which is the text-book approach to build parsers [9]. In contrast, the design of Mison is based on speculation and data-parallel algorithms. This design is motivated by three important observations:

- Applications typically only make use of certain fields. For instance, an application that analyzes Twitter data may only be interested in the timestamps, locations, and authors of a Twitter feed. FSM-based parsers can be lazy, but they need to sequentially traverse all the fields in JSON records; they have no way to jump to any field.
- JSON data has no schema. Nevertheless, there are typically only a limited number of different JSON structural variants in a JSON data stream or JSON collection. Furthermore, there is skew so that most JSON records follow a limited number of structures.
- To determine the exact position of a JSON field in a JSON record of a specific structure, a parser needs to look at a few kinds of characters only; e.g., the ":" character which delimits fields. We can use data-parallel instructions to find these characters and summarize structural information quickly.

Based on these observations, Mison follows a two-step approach. First, it builds a structural index (using data-parallel instructions)

to identify the positions of all fields. Second, it speculates on the schema and directly jumps to the position at which it may most likely find the field that the application is asking for. Due to this design, Mison avoids a great deal of wasted work incurred by existing JSON parsers by parsing irrelevant JSON fields. Even if an application is interested in *all* fields, Mison typically wins by exploiting data-parallel instructions (which is challenging to exploit in full by FSM-based parsers), yet the performance advantages of Mison are smaller in such cases.

One surprising result is that Mison is good for analytical applications on raw JSON data. Traditionally, running analytical queries efficiently involves parsing and shredding the JSON data first and storing it in a binary format such as Apache Parquet [3]. As we will show in the discussion of our performance experiments, this approach is still the most efficient approach if *all* the data is needed for analytics and many analytical queries need to be processed over the JSON data. Our experiments, however, also show that the overheads of running analytical queries on raw JSON data using Mison is surprisingly low, often as little as 50% and never worse than a factor of 3.5 for queries of the TPC-H benchmark. Furthermore, if only a small fraction of the data is ever analyzed or waiting for cooking data is fundamentally unacceptable (e.g., in real-time analytics), then analyzing JSON data in-situ with Mison becomes a viable and cost-effective option, compared to cooking all data into a binary format. In contrast, the overhead incurred by running analytical queries on raw JSON data using Jackson, the state-of-the-art JSON parser, is at least one order of magnitude, forcing organizations to cook all data for potential analytics.

The remainder of this paper is structured as follows: Section 2 contains background information. Section 3 presents a high-level overview of Mison. Sections 4 and 5 give details on the Mison structural index and speculative parsing. Section 6 describes our experimental results. Section 7 covers related work. Section 8 contains concluding remarks.

2. PRELIMINARIES

2.1 JSON Format

JSON (JavaScript Object Notation) [7, 13] is a lightweight, text-based, language-independent data interchange format. The JSON format can be recursively defined as follows (we omit the formal definitions of STRING and NUMBER in the interest of space):

TEXT = OBJECT...OBJECT

OBJECT = {STRING:VALUE, ..., STRING:VALUE}

ARRAY = [VALUE, ..., VALUE]

VALUE = OBJECT | ARRAY | STRING | NUMBER | `true` | `false` | `null`

A JSON *text* is a sequence of zero or more JSON objects. An *object* begins with a left brace ("{") and ends with a right brace ("}"), and contains a sequence of zero or more key/value pairs, separated by commas (","). A key/value pair is called a *field* of the object. Each key is followed by a single colon (":"), separating the key from its corresponding value. An *array* is an ordered collection of values, and is represented as brackets ("[" and "]") surrounding zero or more values, separated by commas. A value can be a string in quotes ("""), a number, `true` or `false`, `null`, an object, or an array. Thus, these structures can be nested. A string is a sequence of zero or more characters, wrapped in quotes. JSON uses backslash as an escape character for "\"", "\\", "\/", "\b", "\f", "\n", "\r", "\t". A number is like an integer or decimal number in C or Java. Whitespace can be inserted between any pair of tokens.

The JSON grammar standard ECMA-404 [7] specifies no behavior on duplicate keys. However, the use of JSON in applications is often more restrictive. The standard RFC-7159 [13] declares that "the names within an object *should* be unique", in the sense that software implementations often use a hash map to represent a JSON object and therefore have unpredictable behavior when receiving an object with duplicate keys. As per RFC-7159, most existing JSON parsers, e.g., Jackson [5] and Gson [4], either do not accept duplicate keys or take only one of the duplicated keys. In this paper, we assume that *all keys are unique within an object*.

In this paper, we intentionally distinguish between the terms *record* and *object*, in the interest of clarity. We use the term record to represent a top-level object in JSON text. In other words, an object is either a record, or a nested object within a record.

2.2 Data Parallelism

Parsing text data is potentially an ideal application to exploit data parallelism, as each element of text data, e.g., an 8-bit character in ASCII encoding, is much smaller than a processor word (64-bit ALU word or 128∼512-bit SIMD word). In this section, we discuss two techniques that exploit data parallelism: SIMD vectorization and bitwise parallelism. Our parsing approach presented in Section 4 combines both of them to achieve the best of both worlds.

SIMD Vectorization. Modern processors provide SIMD capabilities, the ability to perform the same operation on multiple data items simultaneously. Over the last two decades, mainstream processors have evolved to offer a large variety of SIMD instructions and meanwhile to widen the SIMD registers for higher throughput. Today, 256-bit wide SIMD is considered to be the standard, and latest processors such as Intel Knights Landing offer even wider 512-bit SIMD instructions.

Bitwise Parallelism. An alternative vectorization technique is to exploit bitwise parallelism. With this approach, data parallelism is achieved by exploiting the laws of arithmetic rather than specialized hardware. Table 1 lists three bitwise manipulations [23] that provide a fast way to find and manipulate the rightmost 1-bit in a bitmap (the binary representation of an integer value). The notations **R**, **E**, and **S** are used to denote the three manipulations respectively, throughout this paper. Each operator uses only one logical operator ($\wedge$: logical AND, $\oplus$: exclusive OR) and one minus operator to implement the manipulation. Although the example value x in Table 1 contains only 8 bits, we can operate on 64-bit values and achieve 64-way parallelism using today's 64-bit processors.

Manipulations	Descriptions	Examples
x	An integer value	$(11101000)_2$
$\mathbf{R}(x) = x \wedge (x-1)$	Remove the rightmost 1 in x	$(11100000)_2$
$\mathbf{E}(x) = x \wedge -x$	Extract the rightmost 1 in x	$(00001000)_2$
$\mathbf{S}(x) = x \oplus (x-1)$	Extract the rightmost 1 in x and smear it to the right	$(00001111)_2$

Table 1: Bitwise Manipulations [23] and Examples

Modern processors also have specialized instructions for some operations that are computationally expensive to be converted to a sequence of bitwise and arithmetic operations. The parsing approach proposed in this paper relies on one of these instructions, called `popcnt`. `popcnt` computes the digit sum of all bits of a value; i.e., it counts the number of 1s in the value. Given the example value x in Table 1, $\texttt{popcnt}(x) = 4$.

3. OVERVIEW OF MISON

This section provides an overview of Mison. The following two sections describe details of the data structures and algorithms.

The basic idea of Mison is to *push down both projections and filters into the parser*. A key challenge for achieving these features

```
{"id":"id:\"a\"", "reviews":50, "attributes":{"breakfast":false, "lunch":true, "dinner":true, "latenight":true},
 "categories":["Restaurant", "Bars"], "state":"WA", "city":"seattle"}
{"id":"id:\"b\"", "reviews":80, "attributes":{"breakfast":false, "lunch":true, "latenight":false, "dinner":true},
 "categories":["Restaurant"], "state":"CA", "city":"san francisco"}
{"id":"id:\"c\"", "reviews":120, "attributes":{"delivery":true, "lunch":true, "dessert": true, "dinner":true},
 "categories":["Restaurant"], "state":"NY", "city":"new york"}
{"id":"id:\"d\"", "name":"Alice", "age":40, "favorites":30}
{"id":"id:\"e\"", "reviews":70, "attributes":{"breakfast":true, "lunch":true, "dinner":true, "latenight":false},
 "categories":["Restaurant", "Brunch"], "state":"CA", "city":"los angels"}
{"id":"id:\"f\"", "reviews":20, "attributes":{"breakfast":true, "lunch":true, "latenight":true, "dinner":true},
 "categories":["Restaurant", "Brunch", "Bars"], "state":"IL", "city":"chicago"}
```

Figure 2: Sample JSON Data (Yelp)

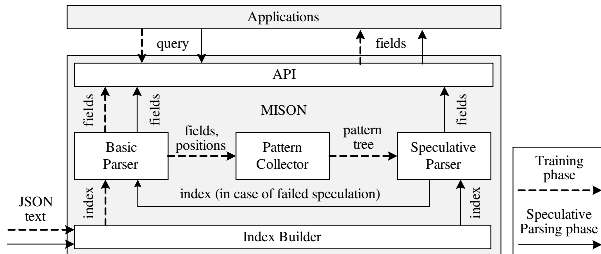


Figure 3: System Architecture of Mison

is to jump directly to the correct position of a queried field without having to perform expensive tokenizing steps to find the field. A straightforward idea is to predict the position of a queried field based on previously seen patterns in the dataset. However, this line of thinking is inherently impracticable because a single variable size field could dramatically impact the locations of all other fields, making the field positions inherently unpredictable.

In contrast, Mison uses a two-level approach to quickly locate queried fields. At the upper level, Mison speculatively predicts the *logical locations* of queried fields (e.g., second sub-field of the third field) based on previously seen patterns in the dataset. Unlike physical positions in JSON text, these logical locations are independent of the sizes of other fields. At the lower level, Mison builds structural indices on JSON text on the fly during parsing, to map logical locations to *physical locations* (e.g., 156th character). The two levels correspond to the two key techniques used in Mison: structural index (Section 4) and speculative parsing (Section 5).

Running example. Throughout this paper, we use a running example of six JSON records, based on the Yelp dataset (see Section 6). Figure 2 shows the running example. Five of the six records contain three primitive fields (`id`, `state`, `city`), one array field (`categories`), and one nested field (`attributes`) at the root level. Each nested "attributes" object includes up to four Boolean fields. Notably, the fourth record in the dataset represents a different kind of entity and thus exhibits a completely different structure.

3.1 System Architecture

Mison selects a small portion of the input JSON records as a training set to learn common patterns in the dataset. Figure 3 shows the architecture of the Mison library, along with data flows in the training and speculative parsing phases. An application can issue a query to Mison through its API. The query includes the fields that are needed for the application and might contain other hints such as the structural information of queried fields to further enhance the parsing speed. This API is described in Section 3.2.

JSON data from external sources are loaded into the *index builder*, which builds a structural index for each record on the fly. The structural index of a record maps the logical position of any field in the record to its physical location in the text array. After the record is fully parsed, the index is destroyed immediately. As a result, through its life cycle, the index always resides in the CPU caches, and is almost never read from or written to main memory. The index structure and building method are described in Section 4.1 and 4.2, respectively.

During the training phase (dashed data flows in Figure 3), the *basic parser* uses the index to parse the record without exploiting previously seen patterns. In this phase, Mison parses and extracts all fields, builds statistics (*pattern collector*), and returns the fields that the application asked for. This basic parsing approach is described in Section 4.3. The pattern collector is described in Section 5.1. It creates a summary representation of all patterns observed in the training phase; we call this summary a *pattern tree*.

During the speculative parsing phase (solid data flows in Figure 3), the structural index is used by the *speculative parser*, which predicts the logical positions of queried fields based on the pattern tree, and probes the index to find the physical locations of these fields. Each speculation is verified by comparing the field name at the predicted physical position to the queried field name. In case of a match, the speculative parsing is successful and the parsed fields are returned to the application. Otherwise, the speculative parser ventures another speculation based on another (less frequent) pattern of the pattern tree. If all speculations fail (unobserved pattern from the training phase), Mison invokes the basic parser as a fallback option. Speculative parsing is described in Section 5.3.

3.2 Programming Interface

The API of Mison is tailored to the design goals of Mison. Unlike the APIs of existing parsers that essentially iterate over all JSON syntax tokens (e.g., RecordBegin, FieldName, etc.), the API of Mison iterates over only the queried fields, intentionally ignoring irrelevant fields and other syntax tokens.

To create a Mison parser instance, the user needs to specify a list of fields that are required to be parsed, called *queried fields*. The list of queried fields is called a *query*. In data analytics systems, the queried fields can often be inferred from the analytical queries; e.g., the fields used in *SELECT, WHERE, GROUP BY*, and *ORDER BY* clauses of SQL queries. Each field in the field list is specified by a path expression using the dot notation. A one-dimensional array field is indicated by a pair of brackets following the field name in the path expression. Mison also allows arbitrary nesting of arrays and objects in path expressions. For instance, the expression "x[].y[][]" represents an array of objects (x) that contain a two-dimensional array (y). Once the parser is created, *field IDs* are assigned to the specified fields, according to the order of these fields in the list. For instance, given a query ("reviews", "city", "attributes.breakfast", "categories[]") for the running example (Figure 2), the field IDs of the four fields "reviews", "city", "attributes.breakfast", and "categories" are 0, 1, 2, 3, respectively.

The actual parsing is driven by two iteration methods. The first method, NextRecord(), skips the remaining of the current record and moves the cursor of the parser to the beginning of the next record. The second method, NextField(), returns the field ID of the next encountered field of interest in the current record until there

are no more queried fields. If a queried field is missing in a JSON record, Mison either skips the record or returns a null value for the queried field, depending on configurations. For an array field, each element in the array is returned individually, similar to a primitive field. Taking the first record of Figure 2 and the example query from the previous paragraph, the parser returns a sequence of field IDs as follows: 0 (reviews), 2 (attribute.breakfast), 3 (categories), 3 (categories), 1 (city), EndOfRecord. Two "categories" fields are returned because the "categories" array has two elements.

In addition to projection push-down, the Mison techniques also enable users to push down filters into the parser. To enable this feature, Mison allows users to partition the field list into *field groups*, and parses field groups in a one-after-the-other fashion. For example, given the example query, if the user specifies the query in two groups: {{"attributes.breakfast", "categories"}, {"reviews", "city"}}, the parser first returns the field IDs of all fields in the first field group, resulting in the sequence of 2 (attributes.breakfast), 3 (categories), 3 (categories). The user can then evaluate the filter predicates with the parsed field values, and skip the remaining fields in the second group if the record does not pass all filters.

4. STRUCTURAL INDEX

Given Mison's design, the most critical operation is to quickly locate a set of queried fields in a JSON text. This section presents a data structure called structural index to bring this capability to a JSON parser. A structural index maps the logical position of a field (e.g., second sub-field of the third field) to its physical position (e.g., 156$^{\text{th}}$ character) in the JSON text. Conceptually, the Mison structural index is similar to metadata information embedded in some binary formats such as Parquet, but it is built on-the-fly as a side-effect of parsing.

Interestingly, building such a structural index for a record can be done one order of magnitude faster than fully parsing the record. First, building a structural index relies on a linear operation that merely looks for structural characters (e.g., ":", "{", and "}") with no branching. In contrast, fully parsing JSON data involves complex switch statements and inherently unpredictable branches. Second, the simplified tokenization process of building a structural index can be implemented by arithmetic and logical operators on a vector of characters by exploiting data parallelism available on modern CPUs. In summary, this indexing approach converts control flow into data flow, largely eliminating unpredictable branches and reducing CPU cycles burnt on each byte.

In this section, we first focus on the simple cases where the query does not contain any array fields, and relax this assumption and extend our solution to support array fields in Section 4.4. We also assume that the input JSON data is in the ASCII encoding. However, it is straightforward to extend our techniques to support other encodings such as UTF-8 and Unicode.

4.1 Index Data Structure

In this paper, we use a *field index* to represent the logical position of a field. We say the field index of a field in an object is k, if the field is the k-th top-level field of the object. For example, the field index of "categories" in the first record of Figure 2 is 4, whereas "attributes.dinner" has a field index of 3 in the object "attributes" nested in the first record.

A structural index maps logical to physical locations of fields in a JSON object. Formally, a structural index of an object takes as input the field index of a field in this object, and outputs the physical location of this field in the JSON text.

The physical location of a field can be represented by the position of the colon character that is placed between the key and the value

```
       Original text   {"id":"id:\"a\"","reviews":50,"a
Word0  Mirrored text   a",05:"sweiver",""/a"/:di":"di"}
          Level 1      00000100000000000000000000100000
          Level 2      00000000000000000000000000000000
       Original text   ttributes":{"breakfast":false,"l
Word1  Mirrored text   l",eslaf:"tsafkaerb"}:"setubirtt
          Level 1      00000000000000000000010000000000
          Level 2      00000000100000000000000000000000
       Original text   unch":true,"dinner":true,"lateni
Word2  Mirrored text   inetal",eurt:"rennid",eurt:"hcnu
          Level 1      00000000000000000000000000000000
          Level 2      00000000000010000000000000100000
       Original text   ght":true},"categories":["Restau
Word3  Mirrored text   uatseR"]:"seirogetac",{eurt:"thg
          Level 1      00000000100000000000000000000000
          Level 2      00000000000000000000000000010000
       Original text   rant","Bars"],"state":"WA","city
Word4  Mirrored text   ytic","AW":"etats",["sraB","tnar
          Level 1      00000000001000000000000000000000
          Level 2      00000000000000000000000000000000
       Original text   ":"seattle"}
Word5  Mirrored text                       {"elttaes":"
          Level 1      00000000000000000000000000000010
          Level 2      00000000000000000000000000000000
```

Figure 4: Example Colon Index: Two Levels, Record 1 of Figure 2 (Index is Built on the Mirrored Text)

of the field. The advantages of using the colon character to locate a field are threefold. First, there is a one-to-one mapping between colon characters and fields in a JSON text. Second, searching colon characters in a JSON text can be implemented in a fast way, as hinted at the beginning of the section and shown in detail in Section 4.2. Third, the field name and value are placed immediately before and after the colon character. Thus, it is efficient to access both the key and the value, starting from the position of the colon character.

Concretely, a structural index a list of bitmap indices, called *leveled colon bitmaps*, each of which uses one bit per character to indicate if the corresponding character is a colon character of a field at a particular level. The colon bitmaps are designed to be on a per-level basis to make objects nested in a record independent of each other. Thus, even a highly dynamic nested object in a record has no impact on the field indices of other fields in the record. The number of leveled colon bitmaps that are needed to be constructed depends on the queried fields. If the deepest queried field is at the l-th level, colon bitmaps are built only up to the l-th level.

Figure 4 illustrates the leveled colon bitmaps built for the first record of Figure 2 and the example query discussed in Section 3.2. As the deepest queried field ("attributes.breakfast") is at the 2nd level, we build colon bitmaps for the top two levels only. For ease of illustration, we assume 32-bit processor words. Within each word of bitmaps, the 32 bits are organized in little-endian order; i.e., the order of bits is reversed within a word. In the figure, we also mirror the JSON text to make it more readable in the reversed order. The Level 1 bitmap indicates all colons that correspond to top-level fields in the record. Note that there is also a colon character inside the string value of the field "id". However, its corresponding bit is turned off in the bitmap, as it does not represent any structural information. The Level 2 bitmap sets bits corresponding to the colons within the nested object "attributes".

4.2 Building Structural Indexes

There are three structural characters that define the structure of a JSON record: colon ":" (key/value separator), left brace "{" (object begin), and right brace "}" (object end). In addition, we need to detect when these characters are used within JSON strings and do not define structure. Thus, Mison also tracks structural charac-

Original text	`{"id":"id:\"a\"","reviews":50,"a`
Mirrored text	`a",05:"sweiver",""/a"/:di":"di"}`
Step 1:	
'\' bitmap:	00000000000000000010010000000000
'"' bitmap:	01000010000000101100100001010010
':' bitmap:	00000100000000000000001000100000
'{' bitmap:	00000000000000000000000000000001
'}' bitmap:	00000000000000000000000000000000
Step 2:	
structural '"':	01000010000000101000000001010010
Step 3:	
string mask:	10000011111111001111111110011100
Step 4:	
structural ':':	00000100000000000000000000100000
L1 ':' mask:	00000100000000000000000000100000
L2 ':' mask:	00000000000000000000000000000000

Figure 5: Building Bitmaps for Word 0 of Figure 4

ters of JSON strings: quote “"” (string begin/end), and backslash “\” (escape character).

The basic idea is to track these structural characters in the JSON text and build bitmap indices on the positions of these characters. These bitmaps are then transformed to leveled colon bitmaps using the bitwise parallelism technique (Section 2.2). Creating the bitmaps is done in the four steps described in the remainder of this subsection.

4.2.1 Step 1: Building Structural Character Bitmaps

The first step is to build bitmap indices on structural characters. As the size of each character is 8 bits, we implement this step using SIMD vectorization. For each structural character, we use the SIMD comparison instruction to compare a vector of n characters (e.g., $n = 32$ for 256-bit SIMD) to a vector that contains n copies of the structural character. The result is a vector in which each 8-bit lane contains either all 1's or all 0's. We then use an additional SIMD instruction to convert the result vector to a bitmap by selecting the most significant bit of each lane. Figure 5 (Step 1) shows the structural bitmaps built on the first word shown in Figure 4.

This is the only step in Mison that relies on SIMD vectorization. Once our initial bitmaps are created, we exploit bitwise parallelism (see Section 2.2) to manipulate bitmaps in the next three steps.

4.2.2 Step 2: Building Structural Quote Bitmaps

Given the quote and backslash bitmaps, the second step generates a structural quote bitmap to represent the begin and end positions of all strings, thereby excluding escaped quotes within JSON strings. As per the specification of JSON format, a structural quote in JSON text is a quote character following zero or an even number of consecutive backslash characters. For example, the first and last quotes in the record “`{"x\"y\\":10}`” are structural quotes, while the one in the middle is an escaped quote inside a string.

We convert a quote bitmap to a structural quote bitmap by turning off bits corresponding to non-structural quotes in the quote bitmap. To implement this conversion in a bit-parallel fashion, we first find two-character subsequences “`\"`” by performing a logical AND between the backslash bitmap and the one-bit-right-shifted quote bitmap. For each 1 in the result bitmap, we compute the length of the consecutive 1s in the backslash bitmap starting at the position of this 1, by using the `popcnt` instruction and bitmap manipulations discussed in Section 2.2. In the interest of space, the pseudocode is omitted here. Figure 5 (Step 2) shows the structural quote bitmap built on the example word.

This implementation runs in $O(\frac{n}{w} + p)$ instructions, where n is the number of characters in the input text, w is the word size in bits, and p is the number of the subsequences “`\"`” in the input text.

Algorithm 1 BuildStringMask(b_{quote})

Input: b_{quote}: the structural quote bitmap
Output: b_{string}: the string mask bitmap

1: $n := 0$ ▷ the number of quotes in b_{quote}
2: **for** $i := 0$ **to** $b_{quote}.|word| - 1$ **do**
3: $\quad m_{quote} := b_{quote}.word_i$
4: $\quad m_{string} := 0$ ▷ the string mask
5: $\quad$**while** $m_{quote} \neq 0$ **do**
6: $\quad\quad m := \mathbf{S}(m_{quote})$ ▷ extract and smear the rightmost 1
7: $\quad\quad m_{string} := m_{string} \oplus m$ ▷ extend m_{string} to the rightmost 1
8: $\quad\quad m_{quote} := \mathbf{R}(m_{quote})$ ▷ remove the rightmost 1
9: $\quad\quad n := n + 1$
10: $\quad$**if** $n \mod 2 = 1$ **then**
11: $\quad\quad m_{string} := \overline{m_{string}}$ ▷ flip m_{string} if necessary
12: $\quad b_{string}.word_i := m_{string}$
13: **return** b_{string}

4.2.3 Step 3: Building String Mask Bitmaps

The next step is to transform the structural quote bitmap to a string mask bitmap, in which a bit is on if the corresponding character is inside of a JSON string, and is off if the corresponding character is outside of any JSON strings (bits at the boundaries of strings can be either 0 or 1).

The following example illustrates this conversion on the example word shown in Figure 5. Initially, the string mask bitmap is set to all 0s. In the first iteration, we first compute a mask m to turn on all bits at the right of the first quote by extracting the rightmost 1 and smearing it to the right in the quote bitmap word. This manipulation is implemented in a bit-parallel way, by using the bitwise operator $\mathbf{S}()$ shown in Table 1. The string mask m_{string} is set to m in this iteration. Then, we remove the rightmost 1 from the word by applying $\mathbf{R}()$. After that, the original second-rightmost 1 (quote) becomes the rightmost 1 (quote) in the next iteration. In the second iteration, we compute m by turning on all bits at the right of the second quote. This mask is then XORed with m_{string}, to extend m_{string} by turning on all bits between the first and second quotes, and turning off all bits at the right of first quote. Thus, m_{string} is now the mask on the first string. We continue this process to incrementally extend m_{string} to the remaining quotes. After the first four iterations (two pairs of quotes), we have generated a string mask that includes the first two strings “`id`” and “`id:\"a\"`”. In the interest of space, we omit the remaining iterations in the example. Figure 5 (Step 3) shows the produced string mask on the example word.

	Original text	`{"id":"id:\"a\"","reviews":50,"a`
	Mirrored text	`a",05:"sweiver",""/a"/:di":"di"}`
Init	m_{quote}	01000010000000101000000001010010
	m_{string}	00000000000000000000000000000000
Iter1	$m = \mathrm{S}(m_{quote})$	00000000000000000000000000000011
	$m_{string} = m_{string} \oplus m$	00000000000000000000000000000011
	$m_{quote} = \mathrm{R}(m_{quote})$	01000010000000101000000001010000
Iter2	$m = \mathrm{S}(m_{quote})$	00000000000000000000000000011111
	$m_{string} = m_{string} \oplus m$	00000000000000000000000000011100
	$m_{quote} = \mathrm{R}(m_{quote})$	01000010000000101000000001000000
Iter3	$m = \mathrm{S}(m_{quote})$	00000000000000000000000001111111
	$m_{string} = m_{string} \oplus m$	00000000000000000000000001100011
	$m_{quote} = \mathrm{R}(m_{quote})$	01000010000000101000000000000000
Iter4	$m = \mathrm{S}(m_{quote})$	00000000000000001111111111111111
	$m_{string} = m_{string} \oplus m$	00000000000000001111111110011100
	$m_{quote} = \mathrm{R}(m_{quote})$	01000010000000100000000000000000

As can be observed from this example, if an odd number of iterations are executed, all bits in m_{string} need to be flipped to be used as an string mask. This trick remains useful even when a string is across word boundaries, as we count quotes not only in the current word but also in all words that have been processed.

Algorithm 2 BuildLeveledColonBitmap(b_{colon}, b_{left}, b_{right}, l)

Input: b_{colon}, b_{left}, b_{right}: the structural colon, structural left brace, and structural right brace bitmaps
l: the number of nesting levels
Output: $b_{0...l-1}$: the leveled colon bitmaps

```
 1: for i := 0 to l−1 do
 2:     b_i := b_colon                      ▷ copy colon bitmap to leveled colon bitmaps
 3: for i := 0 to b_right.|word| − 1 do
 4:     m_left := b_left.word_i; m_right := b_right.word_i
 5:     repeat                              ▷ iterate over each right brace
 6:         m_rightbit := E(m_right)        ▷ extract the rightmost 1
 7:         m_leftbit := E(m_left)          ▷ extract the rightmost 1
 8:         while m_leftbit ≠ 0 and (m_rightbit = 0 or m_leftbit < m_rightbit) do
 9:             S.Push({i, m_leftbit})      ▷ push left bit to stack
10:             m_left := R(m_left)         ▷ remove the rightmost 1
11:             m_leftbit := E(m_left)      ▷ extract the rightmost 1
12:         if m_rightbit ≠ 0 then
13:             {j, m_leftbit} := S.Pop()   ▷ find the matching left brace
14:             if 0 < |S| ≤ l then         ▷ clear bits at the upper level
15:                 if i = j then           ▷ nested object is inside a word
16:                     b_{|S|−1}.word_i := b_{|S|−1}.word_i ∧ ¬(m_rightbit − m_leftbit)
17:                 else                    ▷ nested object is across multiple words
18:                     b_{|S|−1}.word_j := b_{|S|−1}.word_j ∧ (m_leftbit − 1)
19:                     b_{|S|−1}.word_i := b_{|S|−1}.word_i ∧ ¬(m_rightbit − 1)
20:                     for k := j+1 to i−1 do
21:                         b_{|S|}.word_k := 0
22:         m_right := R(m_right)           ▷ remove the rightmost 1
23:     until m_rightbit = 0
24: return b_{0...l−1}
```

Algorithm 3 GenerateColonPositions($index$, $start$, end, $level$)

Input: $index$: the structural index
$start, end$: the start and end positions of the targeted object
$level$: the level of the targeted object
Output: c: an list of positions of colons.

```
1: for i := ⌊start/w⌋ to ⌈end/w⌉ do
2:     m_colon := index.b_level.word_i
3:     while m_colon ≠ 0 do                 ▷ iterate over each 1 in m_colon
4:         m_bit := E(m_colon)              ▷ extract the rightmost 1
5:         offset := i · w + popcnt(m_bit - 1)   ▷ offset of the 1
6:         if start ≤ offset ≤ end then
7:             c.Append(offset)
8:         m_colon := R(m_colon)            ▷ remove the rightmost 1
9: return c
```

Algorithm 1 shows the pseudocode for this step. In the outer loop, we iterate over all the words in the structural quote bitmap (we use $b.word_i$ and $b.|word|$ to denote the i-th word and the number of words in bitmap b, respectively). For each word, we have an inner loop (Line 5-9) to iterate over all 1s in the word. In each iteration, we first manipulate the rightmost 1 in the quote mask to extend the string mask to the position of the 1, and then remove the rightmost 1 from the quote mask. By continuing this process until there are no more 1s in the word, we iterate over all 1s from the right to the left. The string mask generated in the inner loop needs to be flipped if an odd number of quotes have been processed. This algorithms runs in $O(\frac{n}{w}+q)$ instructions, where q denotes the number of structural quotes in the JSON text.

4.2.4 Step 4: Building Leveled Colon Bitmaps

In the last step, we first apply the string mask bitmap to the colon/left brace/right brace bitmaps to filter out characters within JSON strings. The produced bitmaps are called structural colon/left brace/right brace bitmaps, respectively.

Algorithm 2 shows the pseudocode for the final step of the building phase that takes as input the structural colon, left brace, right brace bitmaps and produces leveled colon bitmaps. The basic idea is straightforward: for each right brace, we push all prior left braces to a stack, and then pop out the top one as the matching left brace for the current right brace; given the pair of matching left/right braces, we write the colons between them to a corresponding leveled colon bitmap.

Algorithm 2 implements this idea in a bit-parallel fashion. In the outer loop (Line 3-23), we iterate over all words in the left and right brace bitmaps simultaneously. In the inner loop (Line 5-23), we iterate over each 1 in the current word in the right brace bitmap, i.e., m_{right}. Similar to the algorithm used in Step 3, this is done by repeatedly removing the rightmost 1 (Line 22) until there is no more 1s in m_{right}. For each 1 in m_{right}, we find all prior 1s in the current word of the left brace bitmap, i.e., m_{left}, and push their positions into the stack S (Line 8-11). To do that, we repeatedly remove the rightmost 1 in m_{left} (Line 10) until the mask of the extracted rightmost bit ($m_{rightbit}$) of m_{left} is arithmetically greater than that of m_{right} (Line 8), meaning that the left brace is beyond the right brace. In Line 13, we pop out the top left brace from the stack as the matching left brace of the current right brace, and then update the region between the pair of matching left/right braces in leveled colon bitmaps. This process can also be done efficiently with bitwise manipulations. For example, if the left and right braces are in the same word, we can generate a mask on the characters between them by simply performing $m_{rightbit} - m_{leftbit}$ (Line 16). For instance, if $m_{rightbit} = (01000000)_2$ and $m_{leftbit} = (00000100)_2$, then $m_{rightbit} - m_{leftbit} = (00111100)_2$.

This algorithm runs in $O(\frac{n}{w}+b)$ instructions, where b denotes the total number of braces. Figure 5 (Step 4) shows the produced Level 1 and Level 2 colon bitmaps on the example word.

4.3 Basic Parsing with Structural Index

In this section, we introduce a basic parsing method that uses a structural index to locate queried fields, without exploiting any common patterns in the dataset (the speculative parsing method is described in Section 5).

Algorithm 3 shows the pseudocode for a key building block of this parsing method: generating colon positions of all top-level fields of an object. The start/end positions as well as the level of the object are passed as arguments to the method. In the outer loop, we iterate over all words that overlap with the $[start, end]$ range, in the corresponding leveled colon bitmap (we use w to denote the word size in bits). For each word, we repeatedly extract and remove the rightmost 1 until there is no more 1s in the word (in Line 3-8), iterating over each quote in the JSON text. The offset of each 1 in the bitmap, i.e., the position of the corresponding colon, is computed based on its word id and its offset inside the word (in Line 5). To compute the offset inside the word, we count the number of bits at the right of the 1 (in Line 5), using the `popcnt` instruction.

To parse a whole record, we first invoke Algorithm 3 to obtain the positions of top-level colons in the record. For each colon found, we scan the JSON text backward from the colon position to retrieve the field name, which is then searched against a hash table containing all queried field names. If the field name is included in the hash table, we either return the field value in case of an atomic field, or recursively call Algorithm 3 to find the colon positions of the nested object at the next level in case of an object field. In the latter case, the positions of the current colon and the immediately next colon are used as the start and end positions passed into the method, respectively.

Figure 6 illustrates the parsing process with the example structural index. In this figure, we only show some structural characters in the JSON text due to lack of space. The parsing starts from Level 1, and scan over all 1s in the bitmap using Algorithm 3. For

the nested object field (e.g., “attributes”), we use the 1s before and after the nested object in Level 1 to define a range, and iterate over all 1s in Level 2 within this range to find all queried fields within the nested object (e.g., “attributes.breakfast”). As can be observed from this figure, objects within a record are completely independent of each other during the parsing process, due to the leveled nature of the index.

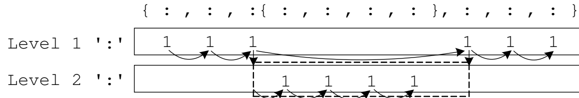


Figure 6: Example: Parsing with Structural Index

4.4 Supporting Array Fields

To support array fields, we extend our solution by: 1) building three more bitmap indices on left brackets, right brackets, and commas, in Step 1 of the building phase; 2) building leveled comma bitmaps for nesting array levels in Step 4 of the building phase, similar to building leveled colon bitmaps for nesting object levels; and 3) using the leveled comma bitmaps to locate array elements during parsing. In the interest of space, the detailed description of these algorithms is omitted in this paper. We note that the extended solution is only needed if the query (rather than the JSON record) contains array fields. If not, the technique presented in Section 4.1-4.3 can be used directly, even if the dataset includes array fields.

5. SPECULATIVE PARSING

In this section, we present the speculative parsing technique for Mison, that aims to exploit common patterns in a JSON data collection (or a JSON data stream) in order to enhance the parsing speed. Although the basic parsing method introduced in Section 4.3 outperforms conventional FSM-based solutions, it still has to retrieve and verify all field names in a record, greatly limiting performance. Speculation allows us to predict the logical locations of queried fields based on previously seen patterns, and only retrieve fields at the speculated locations, which then need to be verified. On a misprediction, we fall back on the basic parsing method.

JSON records usually follow an overall structure or template that does not vary significantly across records. Our speculation mechanism focuses on an aspect of this structure or template: the fields and their field indices in each object. The advantages of this focus are threefold. First, given the field index of a field in an object, we can directly jump to the location of the field in the JSON text, by using the structural index introduced in Section 4. Second, a speculation on the field index of a field can be effectively verified by retrieving and examining the field name in the JSON text. Third, field indices of top-level fields of an object are independent of the sizes of these fields and the hierarchical structure of nested fields.

In order to gain a better understanding on how the structure of each JSON object varies, we conducted workload analysis on a set of representative real-world JSON datasets (see Section 6.1), and summarized three types of structural variance, namely:

- **Extra or missing fields:** Objects in a dataset have extra fields or missing fields. This often happens when the dataset is generated by different versions of the same program.
- **Out-of-order fields:** Fields in objects are arranged in different orders. This type of variance often occurs within highly dynamic objects that are (deeply) nested in records.
- **Mixed entities:** Records in a dataset logically represent different kinds of entities, and exhibit completely different structures. This type of variance is largely due to the combination of multiple datasets representing different entities.

In practice, a real-world dynamic JSON dataset often contains one or a mix of multiple types of structural variance. An effective speculation mechanism should be able to accommodate *all* these types of structural variance.

5.1 Pattern Tree

In Mison, users specify a list of fields that are needed to be parsed. These fields are called *atomic fields*, as these fields contain an atomic value from the view of Mison[1]. Accordingly, a field is called an *object field* if it is an ancestor of an atomic field, i.e., its path is a prefix of an atomic field’s path.

Each occurrence of an object field in a JSON text introduces a *pattern* of the object field, which is defined as a sequence of top-level queried fields within the object, along with their field indices.

For example, given the running example, consider the following query: {“reviews”, “city”, “attributes.breakfast”, “attributes.lunch”, “attributes.dinner”, “attributes.latenight”, “categories”}. According to our definitions, all these seven fields are atomic fields. In addition, we have two object fields: the field “attributes” and a field representing the whole record, called the root field. The first record of the dataset introduces a pattern of the root field: { reviews (2), attributes (3), categories (4), city (6)}. The numbers in parenthesis are the field indices of the corresponding fields. Note that the fields that are not requested by the user are excluded from the pattern, and thus the field indices in the sequence are not consecutive integers. Additionally, the record also introduces a pattern of the nested “attributes” object field: { attributes.breakfast (1), attributes.lunch (2), attributes.dinner (3), attributes.latenight (4)}.

To bring the speculation mechanism into a JSON parser, we design a data structure, called *pattern tree*, to cope with *all* the three types of structural variance. A pattern tree is constructed for each object field, and summarizes the common patterns of the object field in a (large) collection of JSON records.

5.1.1 Data Structure

A pattern tree of an object field f is a multi-way tree structure of height h, where h is the number of queried child atomic/object fields of f. Each root-to-leaf path in the tree corresponds to a unique pattern of f. The path starts from a sentinel root node, followed by nodes corresponding to each queried field in the pattern, ordered by their field indices. Each node in the tree contains three attributes: field, field index, and weight. The weight attribute indicates how many objects exhibit the same pattern for f. If two patterns share a common prefix where both fields and their field indices in the two patterns are the same, we combine the matching nodes in the two paths and add the weights of these matching nodes together. In the tree structure, all child nodes of each internal node are sorted in descending order of their weights.

If any queried field is missing from a pattern, we extend the corresponding path in the pattern tree to include the missing fields, making it have the same length as other paths. The field indices of the newly added nodes are set to be zero to indicate that those fields do not present in this pattern. These nodes are always placed before all ordinary nodes along each path.

Figure 7 illustrates two pattern trees built on the running example dataset, for the query {“reviews”, “city”, “attributes.breakfast”, “attributes.lunch”, “attributes.dinner”, “attributes.latenight”, “categories”} (ignore the dashed path for now). These pattern trees are constructed based on the first five records in the dataset. Each node in the figure is labeled in the form of “FieldName@FieldIndex

[1]The “atomic” value could actually be a JSON object if users specify to parse a field as a generic value or a raw string. In this case, one can think of the object as an atomic string value.

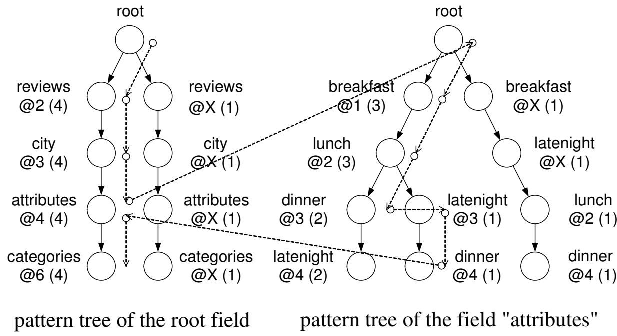


Figure 7: Pattern Trees for Figure 2 and 7 Fields (each node labled with "FieldName@ FieldIndex(weight)")

(weight)". As shown on the left side of the figure, the pattern tree of the root field has two root-to-leaf paths. The left path corresponds to the four records that share a same pattern at the root level, whereas the right path corresponds to a record (the fourth record) representing a different type of entity and does not contain any queried fields. This pattern tree illustrates an example of how to accommodate the "mixed entities" type of structural variance. The right side of the figure shows the pattern tree of the "attributes" object field. The tree has three root-to-leaf paths, corresponding to three unique patterns detected from the four records. The leftmost path represents the most common pattern. The path in the middle corresponds to a pattern that has a pair of out-of-order fields, compared to the most common pattern. The corresponding pattern of the rightmost path includes only two fields. Despite this, the other two missing queried fields are still included in the path but their field indices are marked as "X" to indicate their absence in the pattern. The middle and rightmost paths represent examples of out-of-order fields and missing fields, respectively.

As real-world datasets turn to exhibit strong similarity in the structure of objects, the size of a pattern tree is often very small in practice (as we will show in Section 6.1). However, in the worst case, the size of a pattern tree might become unmanageable, if the object field is highly dynamic. To address this problem, we use a threshold to control the size of a pattern tree: if the weight of a pattern is less than the threshold, the pattern is very unlikely to be used in the parsing phase, and thus is removed from the pattern tree.

5.2 Training Phase

We use a small subset of records in the data collection as a training set to "train" pattern trees. During the training phase, records are parsed with the basic parsing method (see Section 4.3). Along with parsing, patterns of all object fields are also generated, and are collected to incrementally build pattern trees. Given a pattern, we traverse the corresponding pattern tree from the root to a leaf according to the fields and field indices in the pattern, updating the weights of matching nodes along the path and creating new nodes if necessary. When the training phase is completed, we apply the threshold discussed above to finalize all pattern trees.

5.3 Speculative Parsing Phase

As pattern trees capture all common patterns frequently appearing in the training set, the speculative parsing algorithm attempts to parse new objects following one of these patterns, starting from the most common pattern to less common patterns. If none of the patterns in the tree matches the object to have parsed, the speculative parsing is considered to be failed. In this case, the parser falls back on the basic parsing method (see Section 4.3).

Algorithm 4 shows the pseudocode for parsing an object with

Algorithm 4 ParseObject(tree): parsing with a pattern tree

Input: *tree*: the pattern tree of an object
Output: **true** if success, **false** if failed.
1: *node* := *tree*.root
2: **repeat**
3: *success* := **false**
4: **for** each child node *child* of *node* **do**
5: *success* := Verify(*child*.field, *child*.field_index)
6: **if** *success* **then**
7: **if** *child*.field is an atomic field **then**
8: Store the field position
9: **else if** *child*.field is an object field **then**
10: Parse the nested object *child* recursively
11: *node* := *child*
12: **break**
13: **until** *success* = **false**
14: **return** *noce*.is_leaf

a pattern tree. The algorithm traverses the pattern tree starting from the root node. On each node accessed, we invoke the Verify() method to examine if the speculated position is correct. This method first converts the speculated field index to the physical location of the field, using the structural index (see Section 4), and then retrieves the field name in the JSON text at the speculated position. The speculation is then verified by comparing the retrieved field name to the field name in the node. For an absent node, we look up a Bloom filter built on all field names to quickly verify the absence of the node (the details of using Bloom filter is omitted in this paper). If the verification is passed, we move down to the next level. Otherwise, we move to its next sibling node. If a node represents an object field, we recursively drill down to the pattern tree of the nested field. To parse a whole record, we invoke the ParseObject() method on the pattern tree of the root field.

Figure 7 shows the traversal path (the dashed path) for parsing the sixth record in the running example, using the pattern trees constructed on the first five records. In this path traversal, we start from the root node of the root field, and choose the leftmost branch at each level until reaching the node "attributes@4", which is an object field and contains its own pattern tree. After jumping to the pattern tree on the right side and continuing to pick the leftmost child, we reach the node "dinner@3". By examining the field name, the parser realizes that the 3rd field of the "attributes" object is not "dinner". In this case, we shift to its sibling node and find that the next field is actually a "latenight" field at index 3. We continue this traversal down the tree and successfully complete the parsing.

6. PERFORMANCE EVALUATION

We ran our experiments on a workstation with a 3.5GHz Intel Xeon Broadwell-EP (E5-1620 v3) processor, and 16GB of DDR3 main memory. The processor supports a 64-bit ALU instruction set as well as a 256-bit SIMD instruction set. The machine runs 64-bit Windows 10 operating system.

Implementation. We implemented Mison in C++ as a static library. The first step of the structural index building process (Section 4.2.1) was implemented using 256-bit SIMD instructions in AVX2, while the bit-parallel algorithms present in Section 4.2.2-4.2.4 were implemented using ordinary 64-bit instructions. In the evaluation below, we use the first 1000 records in each dataset as the training set to construct pattern trees, and only keep patterns that occur more than 1% of the time in the training phase. The cost of the training phase is included in the reported execution time.

In all the experiments discussed in this paper, we preloaded the JSON data into main memory to exclude I/O time. We used this main memory setting to focus on the best performance of JSON

parsers. This is especially important with the growing popularity of modern memory-optimized systems such as Apache Spark [11,33]. Additionally, high bandwidth I/O devices such as SSDs, 100Gbps Ethernet, and RDMAs, have emerged as powerful solutions to eliminate the I/O bottleneck.

We ran all experiments using a single thread. We have also experimented using multiple threads working on independent data partitions, and observed that Mison achieved near-linear scalability. In the interest of space, we omit these results.

6.1 Micro-Benchmark Evaluation

Baselines. We compare Mison to three best-of-breed JSON parsers: *Jackson* [5], *Gson* [4], and *RapidJSON* [6]. Jackson is largely known as the standard JSON library for Java, and has been used in many open-source projects, e.g., Apache Spark and Apache Drill. Gson is another leading JSON parser in Java from Google. To serve as a yardstick in C++, we also include the leading C++ based JSON parser, RapidJSON. Collectively, these three parsers represent the current state-of-the-art JSON parsers. For each parser, we used the streaming mode API, which is considered to be the most efficient way and with the lowest processing and memory overhead.

We use the following three real-world JSON datasets.

Twitter dataset. The first dataset was collected from the streaming API[2] of the social network Twitter, where users post and interact with Twitter feeds. Each JSON record represents a single Twitter feed entity and contains 25-30 top-level fields which might include deeply nested data. For instance, each record embeds a user object that identifies the author of the feed and contains 37-38 nested fields. JSON records in the Twitter dataset are highly dynamic: parsers need to tolerate the addition of new fields and variance in ordering of fields.

Yelp dataset. The second dataset is from the Yelp dataset challenge program[3]. We used the Yelp business JSON file. Each record in the file corresponds to a Yelp business entity and contains 16 top-level fields.

GitHub dataset. The third dataset was collected from GitHub's API by the GHTorrent project [19]. We used the JSON records in the dataset that model the GitHub's user entity. Unlike the Twitter and Yelp datasets, all user JSON records in the GitHub dataset have a fixed structure: each record contains 31 fields that are arranged in the same order.

6.1.1 Experiment 1: Vary Characteristics of Fields

In the first experiment, we compare the parsing speed of Mison to the three competitors, and evaluate how this comparison is affected by the characteristics of parsed fields, including the nesting object/array level of the parsed fields, the complexity of the pattern trees built for the parsed fields, and whether the parsed fields are optional fields. In this experiment, we use the Twitter dataset, as it is the most dynamic dataset among the three datasets.

Table 2 summarizes the characteristics of the eight queries we used in this experiment. Q1-Q4 are from [30]. These four queries extract 1-2 fields, some of which are nested field at the second level. Q5 also parses two fields, one of which has many possible positions. As shown in Table 2, the root-level pattern tree of this query is fairly complex, and contains 7 different root-to-leaf paths. Q6 extracts a root-level field ("id") and a field inside an optional object field. As illustrated in the pattern tree, this optional object field ("retweeted_status") appears in only 39.7% of tweets. Finally, Q7 and Q8 extract a root-level field and another field that is a primitive field and an array field nested in an array field, respectively.

[2]https://dev.twitter.com/overview/api

[3]https://www.yelp.com/dataset_challenge

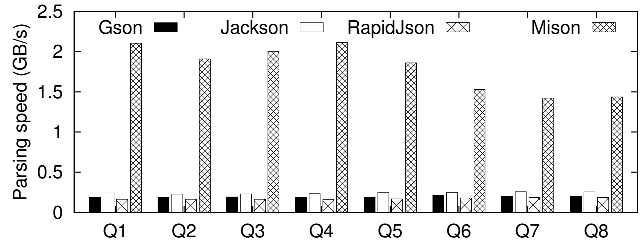


Figure 8: Exp 1: Throughput (GB/s), Twitter, Q1-Q8

	Queries	Level 1 pattern tree	Level 2 pattern tree
Q1	{user.id}	user@11(80.7%) user@12(19.3%)	id@0(100%)
Q2	{user.id, retweet}	user@11(80.7%) retweet@17(40.6%) retweet@18(38.9%) user@12(19.3%) retweet@18(16.6%) retweet@21(1.8%)	id@0(100%)
Q3	{user.id, user.lang}	user@11(80.7%) user@12(19.3%)	id@0(100%) lang@18(100%)
Q4	{user.name, reply_name}	reply_name@10(80.7%) user@11(80.7%) reply_name@11(19.3%) user@12(19.3%)	name@3(100%)
Q5	{user.lang, lang} *(complex pattern tree)*	user@11(80.7%) lang@24(33.8%) lang@23(25.1%) lang@26(15.0%) lang@25(5.6%) user@12(19.3%) lang@24(10.8%) lang@26(6.5%) lang@28(1.8%)	lang@18(100%)
Q6	{id, retweeted _status.id} *(optional field)*	id@1(100%) retweeted@X(60.3%) retweeted@16(39.7%)	id@0(100%)
Q7	{id, entities .urls[].url} *(array field)*	id@1(100%) entities@20(55.5%) entities@19(40.6%) entities@23(2.6%)	urls@1(100%)
Q8	{id, entities .urls[].indices[]} *(nested array field)*	id@1(100%) entities@20(55.5%) entities@19(40.6%) entities@23(2.6%)	urls@1(100%)

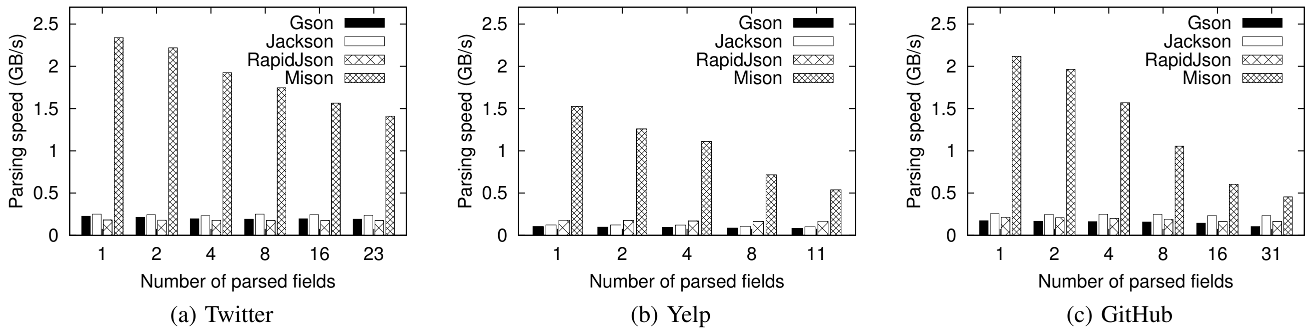
Table 2: Characteristics of the eight experimental queries on the Twitter dataset (Q1-Q4 are from [30])

Figure 8 shows the parsing throughput (amount of parsed data per second) of Mison, Gson, Jackson, and RapidJSON for each query. Mison is between 5.5 and 9.1 times faster than the best competitor in this experiment. The best case for Mison (8∼9X) is for simple queries on mandatory fields (Q1-Q4). The parsing speed of Mison degrades slightly on Q5 which involves more speculation because one of the queried fields of Q5 can be at many different locations. The advantage of Mison drops further for Q6-Q8 because Mison needs to build additional auxiliary data structures for the fields required in these queries: bloom filters for Q6 and leveled comma bitmaps to parse the arrays for Q7 and Q8 (Section 4.4).

6.1.2 Experiment 2: Vary Number of Fields

In the second experiment, we study the sensitivity of Mison to the number of fields in a query. For each dataset, we vary the parsed fields from one field to all root-level fields. By design, the performance of Mison decreases with the fraction of the fields in a JSON record that are needed by an application. Figure 12 shows the results for the three different data sets (Twitter, Yelp, and GitHub).

As can be seen in Figure 12, Mison beats all other JSON parsers in all cases, but the advantage of Mison decreases with the number of fields increases. In the worst case where parsers retrieve all



(a) Twitter (b) Yelp (c) GitHub

Figure 9: Exp 2: Throughput (GB/s): Varying Number of Fields, All Datasets

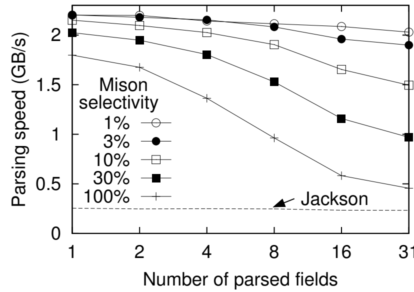


Figure 10: Exp 3: Throughput (GB/s), Vary Selectivity, GitHub

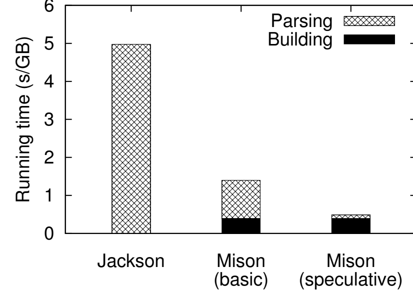


Figure 11: Exp 4: Time per GB: Basic vs. Speculation, Twitter

31 fields of the GitHub dataset without projection push-down, Mison still outperforms the best existing JSON parser (Jackson) by a factor of 2 for making better use of data parallelism and making use of structural indexes, thereby avoiding branch mispredictions introduced by the FSM-based parsers. The throughput of Mison improves, as the fewer fields are required by the application. Even though Jackson partially supports "projection push-down" by offering a special SkipChildren() method, the performance of Jackson (and the other parsers) is not impacted much by the number of fields requested by the application. The reason is that Jackson cannot use the SkipChildren() method to take full advantage of projection push-down because it cannot jump directly to the position of the queried field; instead, Jackson must fully parse skipped fields by calling the NextToken() method internally.

6.1.3 *Experiment 3: Vary Selectivity of Filters*

Next, we evaluate the performance of Mison when pushing down filters into the parser. In this experiment, we use the GitHub dataset and perform a filter on one field. If the parsed value of this field matches the filter predicate, we continue to parse the other queried fields. Filter push-down is not supported by the other parsers because it requires the capability to skip over fields to find the next matching field (the same capability that is needed for projection push-down).

Figure 10 plots the parsing speed of Mison for a varying number of fields (projection push-down) and for different queries that vary the selectivity of the pushed-down predicate from 1% (highly selective filter) to 100% (no filter). As a baseline, Figure 10 also shows the performance of Jackson which is nearly independent of the selectivity of the predicate. As expected, the performance of Mison is better than Jackson under all circumstances, including the extreme case with all fields (no projections) and all records (no filter). Furthermore, this experiment confirms the results of Experiment 2 that Mison's performance improves with fewer fields involved in the query. However, in the presence of highly selective filters, Mison's sensitivity to the "number of fields" parameter drops because Mison does not need to parse fields of records that are filtered out by the predicate. Formally, let s denote the selectivity of the filter, and f denote the number of fields in the projection list of the query. Then, the average number of fields that Mison needs to parse per record is $1+s\cdot f$. As a result, pushing down filters is particularly effective when the query has a long projection list.

6.1.4 *Experiment 4: Basic vs. Speculative Parsing*

To better understand the performance characteristics of Mison, our fourth experiment compared the basic parsing method (Sec 4.3) and speculative parsing method (Sec 5.3), thereby measuring the time breakdown for the building and parsing phases of Mison. This experiment was done using the Twitter dataset and a query that asks for the first and last fields of every record. The results for the Yelp and GitHub datasets were similar so that we do not show them for brevity. As a baseline, we used Jackson in this experiment because it was the fastest existing JSON parser for this experiment.

Figure 11 shows the results. Even the basic Mison parser (without speculation) outperforms Jackson by a factor of 3.6. With speculation, the speed-up is a factor of 10.2. Comparing the basic Mison with the full Mison parser with speculation, we observe that both parsers take the same amount of time to construct the structural index. However, speculation dramatically reduces the cost of parsing by skipping (mostly successfully) to the right fields of the records, rather than inspecting all fields. Therefore, both main contributions of this work, structural indexes and speculation, are needed to achieve high performance.

6.2 TPC-H Benchmark

The goal of Experiment 1 to 4 was to show how Mison performs vs. state-of-the-art JSON parsers in different scenarios. All these results were achieved using micro-benchmarks that involved real-world JSON data and simple queries and application logic that was applied to each record individually. We now turn to experiments that study the utility of Mison to process complex analytical queries on raw JSON data. To this end, we use the TPC-H benchmark and Apache Spark [11, 33] (version 2.1.0) to process the 22 TPC-H queries. For this experiment, we used a TPC-H database with a

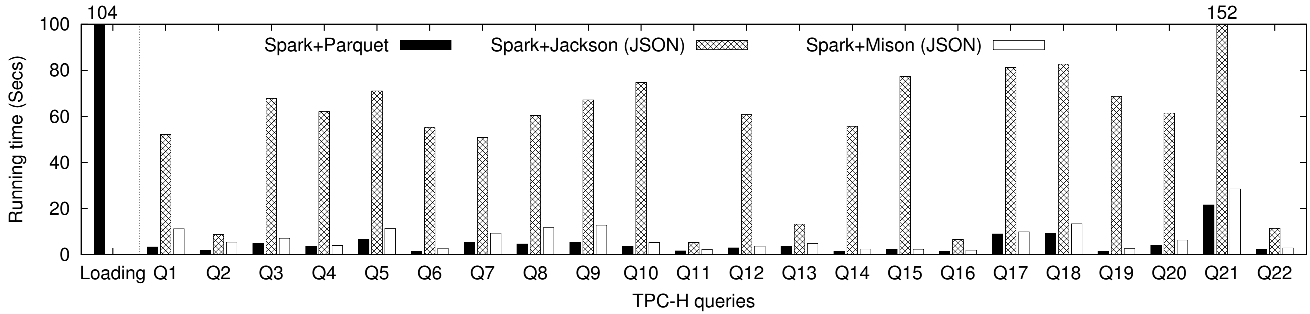


Figure 12: Performance comparison among Parquet, Jackson, and Mison with the TPC-H queries using Apache Spark

scale factor of 1 and converted each table to a flat JSON file. The total size of the JSON files is approximately 3GB. As a baseline, we run the TPC-H benchmark using Spark on a binary database (using Parquet); the size of the binary database was 0.6GB which is of course much smaller than the 3GB JSON data.

We studied three different solutions: (1) Spark on Parquet which is the best possible way to implement the TPC-H benchmark using Spark. It took about 100 seconds to cook the TPC-H JSON data into the Parquet format using Spark; these 100 seconds are not included in the results presented below, giving this baseline an unfair advantage. (2) Spark + Jackson which is the standard and deeply integrated way to execute Spark queries on JSON data. (3) Spark + Mison. In this variant, we manually pushed down the projections and filters of the TPC-H queries into Mison. Mison, then, parsed the relevant JSON data for each query, serialized the results using Apache Avro [1], and finally loaded and deserialized the Avro data using Spark. If Mison were better integrated into Spark (e.g., like Jackson), we could do much better and avoid the unnecessary serialization and deserialization of intermediate query results to and from Avro.

Figure 12 shows the execution times of all three variants. Obviously, "Spark+Parquet" performs best. The surprising result is that "Spark+Mison" is not far behind for most queries. The reason is that Mison takes advantages of selection and projection push-down and, therefore, pays the price of JSON parsing only for the relevant data that is needed in the queries. "Spark+Jackson", however, shows poor performance for all queries and is not attractive for most analytical scenarios because it needs to parse all the records of a table involved in a query.

To give more detail, Table 3 summarizes the number of accessed attributes and the selectivity of filters for each TPC-H query. It becomes clear that the overheads of the "Spark+Mison" over the "Spark+Parquet" approach are particularly small if the query uses few fields, has highly selective predicates, and the query is complex and involves many joins, sub-queries, and group-bys.

For most TPC-H queries, the overhead of processing raw JSON with Mison is insignificant. However, there are a few queries on which the overhead is considerable. Below, we analyze the performance of Mison for the queries on which Mison is more than 2X slower than Parquet.

Q1 and Q6 are the two simple scan queries in the benchmark. Parquet outperforms Mison by 3.4X and 2.1X on these two queries, respectively. The query processing cost accounts for a relatively small portion of the total running time, making the parsing operation hinder the overall performance. For Q6, Mison shows a slightly better relative performance to Parquet than Q1, mainly because the scan selects only 1.9% records and does not need to parse all queried fields in most cases.

For the join queries Q2, Q8, and Q9, the speedups of Parquet over Mison is 3.2X, 2.5X, and 2.4X, respectively. These three queries have no filters on the fact tables and select a relatively large subset of attributes from the fact tables (3/5, 5/16, and 6/16, respectively). In these scenarios, the parsing cost is higher than the query processing cost, even though multiple joins are involved in these queries. Nevertheless, we observe that these queries contain fairly selective filters on dimension tables (0.08%, 0.04%, and 5.3%, respectively). This characteristic introduces an opportunity to push down broadcast hash joins into the parsing level as well, i.e., for each parsed key in the fact table, we look up the key against a hash table built on the filtered dimension tables, and then only parse other queried fields if the hash table contains a matching key. An interesting direction for future work is to extend Spark to support this optimization, closing the performance gap between Mison and Parquet on this class of queries.

	Largest table (LT)	% of queried fields in LT	Selectivity on LT	QP complexity	Mison / Jackson	Parquet / Mison
Q1	Lineitem	7 / 16	98.5%	Low	4.6X	3.4X
Q2	Partsupp	3 / 5	100%	Medium	1.6X	3.2X
Q3	Lineitem	4 / 16	54.0%	Medium	9.5X	1.5X
Q4	Lineitem	3 / 16	63.2%	Medium	15.6X	1.1X
Q5	Lineitem	4 / 16	100.0%	Medium	6.3X	1.7X
Q6	Lineitem	4 / 16	1.9%	Low	19.8X	2.1X
Q7	Lineitem	5 / 16	30.4%	Medium	5.4X	1.7X
Q8	Lineitem	5 / 16	100%	Medium	5.2X	2.5X
Q9	Lineitem	6 / 16	100%	Medium	5.2X	2.4X
Q10	Lineitem	6 / 16	24.6%	Medium	14.1X	1.4X
Q11	Partsupp	4 / 5	100%	Medium	2.3X	1.4X
Q12	Lineitem	5 / 16	0.5%	Medium	16.3X	1.3X
Q13	Orders	3 / 9	98.9%	Medium	2.7X	1.3X
Q14	Lineitem	4 / 16	1.2%	Medium	22.6X	1.6X
Q15	Lineitem	5 / 16	3.7%	Medium	32.5X	1.0X
Q16	Partsupp	2 / 5	100%	Medium	3.4X	1.4X
Q17	Lineitem	3 / 16	100%	Medium	8.2X	1.1X
Q18	Lineitem	2 / 16	100%	Medium	6.2X	1.4X
Q19	Lineitem	6 / 16	2.1%	Medium	25.8X	1.7X
Q20	Lineitem	4 / 16	15.2%	Medium	9.6X	1.6X
Q21	Lineitem	4 / 16	100%	High	5.3X	1.3X
Q22	Orders	2 / 9	100%	Medium	3.9X	1.3X
				Geometric Mean:	7.5X	1.6X

Table 3: Characteristics of the TPC-H queries

In summary, Mison exhibits comparable query processing performance to Parquet in most cases, without having to pay an up-front investment in cooking the data into a binary format. This investment is heavy in most applications and many applications only analyze a small fraction of the data. Additionally, data cooking has several other disadvantages which are not easily quantifiable. JSON is more flexible than binary data and it is a standard so that JSON data can be embedded into a larger variety of different analytical systems. Considering all these factors, we believe that, with high-performance parsers like Mison, the conventional wisdom regarding up-front data cooking for analytics must be reconsidered: there may be many applications for which it is better not to cook the data and carry out analytics directly on raw JSON using Mison.

7. RELATED WORK

The growing popularity of the JSON format has fueled increased interest in building or extending analytical systems to load and process JSON data. Such analytical systems generally fall into two categories. In the first category are systems that offer SQL-like or programming interfaces for querying raw JSON data without a pre-defined schema. Examples of such systems include Apache Spark [11, 33], Drill [2], Jaql [12], and Trill [15]. These systems are ideal applications to which Mison can be applied, because they can leverage Mison to push down projections and filters from the analytical queries into the parser (as shown in Section 6.2).

Systems in the second category require an up-front loading process to cook semi-structured data into a more efficient representation that can be accessed by data processing systems. Such approaches include Dremel [27], Argo [16], Sinew [30], and Oracle's approach [26]. Both Apache Spark and Drill could be also used in this mode by explicitly converting JSON data to Parquet [3]. Mison is also applicable to the loading process of such systems, especially when a user only needs to load a fraction of raw data by either selecting certain fields or applying filters.

Our work is closely related to the body of work in loading and processing raw text data in database systems [8, 10, 17, 21, 28]. The index of Mison is largely inspired by the pioneering work NoDB [10,21], which builds structural index on raw CSV files, and adaptively uses the index to load the raw data. Muhlbauer et. al. improve the speed of parsing and loading CSV files with several optimizations including using SIMD instructions [28]. However, unlike these prior work that focuses on flat CSV files, Mison is proposed to support highly flexible semi-structured data that might contain nested objects and arrays, as well as structural variants. To achieve this goal, we propose novel approaches including bitmap-based index and speculation-based parsing methods in this paper.

The parsing performance issue was also identified for the XML format [29]. Many techniques (e.g., [14, 20, 24, 31]) have been developed to accelerate XML parsing. In particular, Deltarser [31] is designed to quickly identify portions in a XML record that differ from previous records, and only parse these "delta" portions. This approach is similar to Mison in that both methods leverage common patterns in the dataset. Parabix [14] exploits the bitwise data parallelism as well to accelerate XML parsing. However, Mison differs from it in the use of speculation and structural index.

The proposed structural index relies on the bitwise manipulations observed in previous work [23], to exploit the bitwise parallelism. Similar techniques have already been adopted in the context of database systems, such as the scan primitive [22,25]. The speculation mechanism proposed in this paper is related to the automatic schema generation problem [18] in the sense that they both leverage the common patterns in JSON data. However, our approach focuses on discovering the structural information such as the logical positions of fields, while their approach targets detecting functional dependencies to guide the schema generation.

8. CONCLUSIONS AND FUTURE WORK

With the increasing demand for data analytics on raw data, there is a critical need for a fast JSON parser. This paper proposes an approach called Mison that addresses this need by pushing down both projections and filters into the parsing level. Mison is based on speculation and data-parallel algorithms to quickly locate queried fields without having to perform expensive tokenizing steps. Our experimental studies demonstrate that Mison is faster than existing JSON parsers (in some cases by over an order of magnitude), and show that it presents a promising approach for fast data analytics.

For future work, we plan to explore methods to incrementally convert and load raw text data to a binary and column-oriented data representation using Mison in analytical engines, and study the impact of Mison on query optimization. We also plan to extend the proposed techniques to support other common text data formats such as CSV, XML, and HTML.

9. REFERENCES

[1] Apache Avro. `https://avro.apache.org/`.

[2] Apache Drill. `https://drill.apache.org/`.

[3] Apache Parquet. `https://parquet.apache.org/`.

[4] Google Gson. `https://github.com/google/gson`.

[5] Jackson. `https://github.com/FasterXML/jackson`.

[6] RapidJSON. `http://rapidjson.org/`.

[7] The JSON Data Interchange Format. Standard ECMA-404, Oct. 2013.

[8] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible loading: access-driven data transfer from raw files into database systems. In *EDBT*, 2013.

[9] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[10] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: efficient query execution on raw data files. In *SIGMOD*, 2012.

[11] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: relational data processing in spark. In *SIGMOD*, 2015.

[12] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011.

[13] E. T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, Mar. 2014.

[14] R. D. Cameron, E. Amiri, K. S. Herdy, D. Lin, T. C. Shermer, and F. Popowich. Parallel scanning with bitstream addition: An XML case study. In *Euro-Par*, 2011.

[15] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB*, 8(4):401–412, 2014.

[16] C. Chasseur, Y. Li, and J. M. Patel. Enabling JSON document stores in relational systems. In *WebDB*, 2013.

[17] Y. Cheng and F. Rusu. Parallel in-situ data processing with speculative loading. In *SIGMOD*, 2014.

[18] M. DiScala and D. J. Abadi. Automatic generation of normalized relational schemas from nested key-value data. In *SIGMOD*, 2016.

[19] G. Gousios. The ghtorrent dataset and tool suite. In *MSR*. IEEE Press, 2013.

[20] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[21] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my data files. here are my queries. where are my results? In *CIDR*, 2011.

[22] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1(1):622–634, 2008.

[23] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 12th edition, 2009.

[24] C. Koch, S. Scherzinger, and M. Schmidt. XML prefiltering as a string matching problem. In *ICDE*, pages 626–635, 2008.

[25] Y. Li and J. M. Patel. BitWeaving: fast scans for main memory data processing. In *SIGMOD*, 2013.

[26] Z. H. Liu, B. C. Hammerschmidt, and D. McMahon. JSON data management: supporting schema-less development in RDBMS. In *SIGMOD*, 2014.

[27] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[28] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant loading for main memory databases. *PVLDB*, 6(14):1702–1713, 2013.

[29] M. Nicola and J. John. XML parsing: a threat to database performance. In *CIKM*, 2003.

[30] D. Tahara, T. Diamond, and D. J. Abadi. Sinew: a SQL system for multi-structured data. In *SIGMOD*, 2014.

[31] T. Takase, H. Miyashita, T. Suzumura, and M. Tatsubori. An adaptive, fast, and safe XML parser based on byte sequences memorization. In *WWW*, 2005.

[32] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy. Storm@twitter. In *SIGMOD*, 2014.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.