JSONSki: Streaming Semi-structured Data with Bit-Parallel Fast-Forwarding

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

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

ABSTRACT

Semi-structured data, such as JSON, are fundamental to the Web and document data stores. Streaming analytics on semi-structured data combines parsing and query evaluation into one pass to avoid generating parse trees. Though promising, its conventional design requires to parse the data stream in detail character by character, which limits the efficiency of streaming analytics.

This work reveals a wide range of *opportunities to fast-forward* the streaming over certain data substructures irrelevant to the query evaluation. However, identifying these substructures itself may need detailed parsing. To resolve this dilemma, this work designs *a highly bit-parallel solution* that intensively utilizes bitwise and SIMD operations to identify the irrelevant substructures during the streaming. It includes a new streaming model—recursive-descent streaming, for an easy adoption of fast-forward optimizations, a concept—structural intervals, for partitioning the data stream, and a group of bit-parallel algorithms implementing various fast-forward cases. The solution is implemented as a JSON streaming framework, called JSONSki. It offers a set of APIs that can be invoked during the streaming to dynamically fast-forward over different cases of irrelevant substructures. Evaluation using real-world datasets and standard path queries shows that JSONSki can achieve significant speedups over the state-of-the-art JSON processing tools while taking a minimum memory footprint.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**; • **Software and its engineering** → **Parsers**; • **Information systems** → **Semi-structured data**.

KEYWORDS

JSON, Parser, Semi-structured Data, SIMD, Bit-Parallel Algorithm

ACM Reference Format:
Lin Jiang and Zhijia Zhao. 2022. JSONSki: Streaming Semi-structured Data with Bit-Parallel Fast-Forwarding. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3503222.3507719

```
{ "coordinates" : [
    40.74118764, -73.9998279
  ],
  "user" : {
    "id" : 6253282
  },
  "place" : {
    "name" : "Manhattan",
    "bounding_box" : {
      "type" : "Ploygon",
      "pos" : [
        [-74.026675, 40.683935], …
} } } }
```

Figure 1: Geo-referenced Tweet in JSON [51]

1 INTRODUCTION

Recent years have seen a surge in adopting semi-structured data, like JSON (JavaScript Object Notation) and some of its variants, in the computing infrastructures, ranging from data transfer among loosely-coupled cloud services [12, 21, 25, 52] to the underlying data representation of popular document-based data stores, like MongoDB [44] and Firebase [29], to public data release through open APIs (e.g., data.gov [4]). Thus, the efficiency of semi-structured data processing is getting more critical to many modern software applications that require low data processing latency and strict memory budget. For example, a recent study [41] shows that parsing raw JSON data (using the default Jackson Parser [8] in Spark) can easily dominate the query evaluation time (>80%).

Figure 1 shows some attributes in a JSON Tweet object [1], which include `coordinates`, `user`, and `place` [51]. Note that the attributes themselves could be objects or arrays. In general, there are two processing schemes for semi-structured data:

- *Preprocessing scheme* first parses the data stream into some in-memory data structure (typically a tree), then evaluates queries by traversing the data structure. Examples of this scheme include common JSON tools, such as RapidJSON [11], FastJSON [2], Gson [6], and simdjson [40]. Though intuitive, this scheme suffers from a significant upfront delay due to the pre-parsing. Moreover, the parse tree may occupy a large chunk of memory for long data streams.
- *Streaming scheme* can naturally avoid the above issues by immediately consuming the semi-structured data stream, locating and extracting the data of interests on the fly. For example, JsonSurfer [13] and JPStream [35] evaluate path queries dynamically as they traverse the data stream without generating any in-memory trees. Thus, they only need to scan the data in one pass with a small memory footprint.

[1]The object is simplified and slightly modified for illustration purpose.

Despite its promises, the streaming scheme needs to parse the entire data stream character by character to recognize each token and each syntactical structure (e.g., an object) for query evaluation. This detailed parsing seriously limits the efficiency of streaming. In this work, we ask a key question: *is it possible and practical to "fast-forward" through the data stream to accelerate the processing?*

Opportunities. Interestingly, we find that, by leveraging the query and the syntax of semi-structured data, certain data segments could be "irrelevant" to the query evaluation, thus can be fast-forwarded—their tokens and syntactical structures need not to be fully examined and the query matching state updates could be bypassed. Consider the JSON object in Figure 1 in a data stream. Assume the query is to find the `name` of object `place` (i.e., `$.place.name`), then the following fast-forward opportunities present:

- First, the streaming may fast-forward over the first attribute, including its name "`coordinates`" and value, as it is an array, while the query looks for an object (`place` must be an object as it has an attribute `name`).
- When it comes to the second attribute, though it is an object, its name ("`user`") fails to match the "`place`" in the query, thus the streaming can fast-forward over its value—an object.
- After "`name`" is matched, the streaming may fast-forward over the remaining attributes of object `place`, in this case the attribute `bounding_box`, as attributes in a JSON object cannot share names (no more matches are possible).

The above cases are for objects in queries. Similar opportunities also exist for arrays. In this work, we systematically explore these opportunities and categorize them into basic groups.

However, without a priori knowledge about the input, a naïve way of implementing the above fast-forward ideas still needs to traverse the entire data stream in detail to find out the opportunities and to mark the boundaries of data segments (e.g., an object) for fast-forwarding. This essentially becomes a "chicken-egg" issue.

Practicality. To resolve the issue, we need an alternative yet much faster solution to implementing the fast-forward. For that purpose, we propose a *bit-parallel solution* that leverages bitwise and SIMD operations to perform the fast-forward actions. It includes i) a new streaming model for integrating the fast-forward optimizations—*recursive-descent streaming*, ii) an abstraction for partitioning the data stream into basic units—*structural intervals*, and iii) a set of *bit-parallel algorithms* implementing various fast-forward cases.

With bit-parallel fast-forward, the streaming no longer needs to scan the characters one by one, instead, it processes a word-size of characters all at once. In fact, some recent works, like simdjson [40], Mison [41], and Pison [34], also leverage the bit-parallelism in JSON data processing, but their uses are limited to locating the metacharacters and they all fall into the preprocessing scheme.

Based on the above techniques, a streaming framework for JSON data, called JSONSki, is implemented. It offers a set of bit-parallel fast-forward functions that can be naturally integrated into the streaming analytics. To demonstrate its efficiency, we compared JSONSki with several existing JSON tools using real-world datasets and standard path queries. The results have shown that JSONSki outperforms JPStream [35], a state-of-the-art streaming library, and simdjson [40], a popular SIMD-based parser substantially, in both the large and small record processing scenarios.

```
    object  ::=  { }|{ attributes }
attributes  ::=  attribute | attribute, attributes
 attribute  ::=  attribute-name : value
     array  ::=  [ ]|[ elements ]
  elements  ::=  element | element, elements
   element  ::=  value
     value  ::=  object|array|primitive
```

Figure 2: JSON Grammar

In summary, this work makes a three-fold contribution:

- First, it systematically reveals the fast-forward opportunities in the semi-structured data streaming (Section 3).
- Second, it presents a bit-parallel solution for implementing fast-forward optimizations with high-efficiency (Section 4).
- Finally, this work compares the proposed solution with some state-of-the-art JSON data processing tools and confirms its performance superiority (Section 5).

Next, we provide more background of this work.

2 BACKGROUND

We first introduce the basics of semi-structured data, then present the details of the two basic processing schemes.

Semi-Structured Data. Unlike relational data (stored in tables), semi-structured data do not have to follow a strict schema. In fact, the data is self-contained with its type information, structures, and values, thus exposing more flexibility. There are two major styles of semi-structured data: i) tag-based data, like XML, and ii) programming language-based data, like JSON. The former carries data in a hierarchy of pair tags, such as `<id>6536</id>`. XML was popular in the early days of web development. However, recent trends [15, 52] show that it is getting replaced by JSON.

JSON, originated from JavaScript, has a compact format. As Figure 2 shows, it has two basic structures: an *object*—a list of attributes enclosed by a pair of parentheses, such as `{"id":653, "name":"John"}`, where a colon separates an attribute name and its value, and an *array*—an ordered list of elements enclosed by a pair of brackets, like `[653, "John"]`. Moreover, an attribute value or an array element itself can be an object or array. Thus, they can form a complex multi-level structure, like the Tweet object in Figure 1. More details of JSON syntax are given by its standard [5]. We refer to the root-level object or array as a *JSON record*. Depending on the application, a JSON data stream may consist a single large JSON record or a sequence of small records.

Queries. Semi-structured data can be queried with *path expressions*, which specify the paths from the root of a record to one or multiple object attributes. Since the root (an object or an array) is anonymous, a path query uses a $ sign to refer to it. For array-type attributes, the query may carry index constraints. For example, `$.places[0:2]` refers to the first two elements in the `places` array. To select all array elements, the wildcard $*$ could be used, like `$.places[*]`. More details about path queries are available in [32].

As mentioned earlier, to evaluate JSON path queries, there are *preprocessing* and *streaming* schemes.

Preprocessing scheme first builds an in-memory data structure for each record, before evaluating queries. For example, most existing

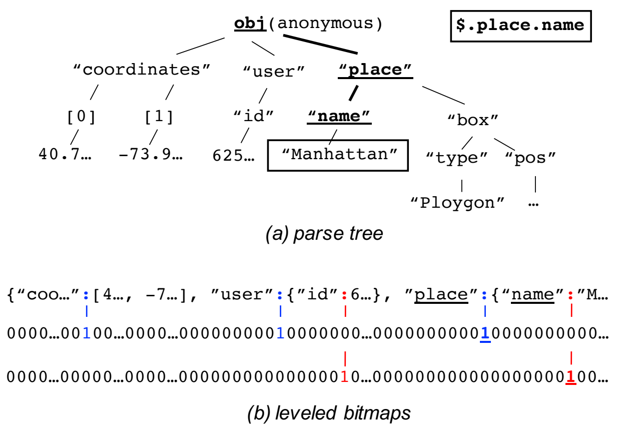

Figure 3: Preprocessing-based Query Evaluation

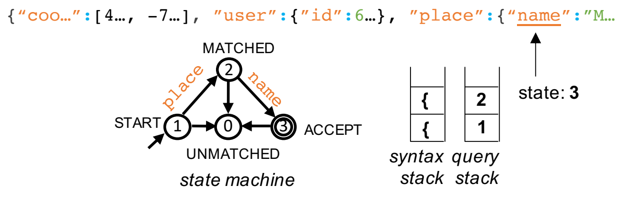

Figure 4: Streaming Query Evaluation

JSON tools, such as JSON-C [7], RapidJSON [11], FastJSON [2], Gson [6], and simdjson [40], first build a parse tree, then traverse it top down to evaluate path queries, as illustrated in Figure 3-(a).

Instead of creating a parse tree, Mison [41] and Pison [34] build bitmaps for colons and commas at each level of a JSON record, called *leveled bitmaps*. The *colon bitmap* helps locate the object attributes, while the *comma bitmap* helps find the array elements. Figure 3-(b) shows the colon bitmaps for the top-two levels of a JSON record. With them, the evaluation can quickly traverse the object attributes or array elements to match the query.

Regardless which data structure is constructed, preprocessing scheme introduces a significant delay upfront. Moreover, the built data structure itself may consume memory substantially, especially for data streams with large records.

Streaming scheme avoids the above issues by combining parsing and the query evaluation into a single pass [13, 35]. It immediately consumes the data as it is being traversed and its structures are being parsed. The key idea is to employ two stacks: a *query stack* for tracking the matching progress at different levels of a record and a *syntax stack* for recognizing the syntactical structures [35]. Figure 4 illustrates the basic idea. The state machine captures the query's matching logic. The syntax stack shows the current syntactical level in the record. The query stack shows that it has reached state 2 at the current level. After the token "`name`" is consumed, the current state would become 3 (an accept state), indicating a match.

Without the in-memory data structure construction, streaming evaluation's efficiency tends to be higher than the preprocessing scheme. However, its existing designs [13, 35] need to scan the input in detail to recognize every token and syntactical structure in the data stream, which fundamentally limits its efficiency. In the following, we will demonstrate that it is possible to fast-forward through the data stream without comprehensive parsing.

`[Key]`	$\Delta(q, c, K, *) \to (\delta(q, K), c, q : *)$
`[Val]`	$\Delta(q, c, V, q' : *) \to (q', c, *)$
`[Ary-S]`	$\Delta(q, c, \texttt{[}, *) \to (\delta(q, \texttt{[}), 0, qc : *)$
`[Ary-E]`	$\Delta(q, c, \texttt{]}, q'c' : *) \to (q', c', *)$
`[Com]`	$\Delta(q, c, \texttt{,}, *) \to (q, c + 1, *)$

Figure 5: Transition Rules of Query Automaton

3 STREAMING WITH FAST-FORWARDING

This section first introduces a new basic streaming model, then presents the fast-forward ideas under this model.

3.1 Streaming Model—Recursive Descent

The existing design [35] for streaming evaluation of path query is a dual-stack automaton based on 13 transition rules. This formal design makes it less intuitive to integrate various fast-forward ideas which are diverse and ad-hoc by nature. Though it might be possible to do so in theory, the resulted design would be complex and hard to maintain. For these reasons, we propose a new streaming model which is more amenable to the fast-forward optimizations.

The key to the new model is to employ a *recursive-descent parser* to drive the query matching automaton. It simplifies the transition rules as the syntax stack becomes a call stack which is automatically managed by the runtime system. Also, fast-forward logic can be easily added into the recursive functions. Next, we describe two major parts: *query automaton* and *recursive-descent streaming*.

Query Automaton. A path expression, like `$.place.name`, can be treated as a regular expression, so that the matching progress can be captured by a finite state machine [30, 35, 46]. Figure 4 shows a four-state machine, where ① is the START state, ③ is the ACCEPT state, and ⓪ is the UNMATCHED state. Unlike regular expressions, a path expression needs to be matched according to their levels in the data record, so a stack is needed for holding the current state at each level. Together, they form a pushdown automaton. For queries with array index constraints (like `pos[2:4]`), a counter will be associated with the state at the corresponding level to match the specified range. More details regarding this conversion can be found in our prior work [35].

The query automaton consumes five types of tokens: K, V, `[`, `]`, and `,`, where K is an attribute name, like "`place`", V represents all possible values of an attribute, and the others are metacharacters in JSON. Note that the comma tokens are only those used to separate array elements; the others are filtered out during the streaming.

Figure 5 lists the transition rules of the query automaton, one for each type of tokens, and each rule is in the format:

$$\Delta(\mathit{state}, \mathit{counter}, \mathit{token}, \mathit{stack}) \longrightarrow (\mathit{state}, \mathit{counter}, \mathit{stack})$$

Rule `[Key]` says, when an attribute name K is consumed, the automaton pushes the current state q onto the stack (stack elements are separated by colons), then updates the current state based on the state transition $\delta(q, K)$. Rule `[Val]` pops the stack top q' and

Algorithm 1 Recursive-Descent Streaming

```
 1: function object()
 2:     consume("{", qa.isAccept)  /* qa: query automaton */
 3:     if notEmpty() then
 4:         while true do
 5:             attribute = consumeAttrName(qa.isAccept)
 6:             consume(":", qa.isAccept)
 7:             qa.transition(attribute) /* update query automaton */
 8:             type = getAttributeType()
 9:             switch type do
10:                 case "object": object()  break
11:                 case "array": array()  break
12:                 case "primitive":  primitive()
13:             qa.transition(V) /* V denotes all values */
14:             if hasMoreAttributes() then
15:                 consume(",", qa.isAccept)
16:             else break
17:     consume("}", qa.isAccept)
18:
19: function array()
20:     consume("[", qa.isAccept)
21:     qa.transition("[")
22:     if notEmpty() then
23:         while true do
24:             type = getElementType()
25:             switch type do
26:                 case "object": object()  break
27:                 case "array": array()  break
28:                 case "primitive":  primitive()
29:             if hasMoreElements() then
30:                 qa.transition(",")
31:                 consume(",", qa.isAccept)
32:             else break
33:     consume("]", qa.isAccept)
34:     qa.transition("]", qa.isAccept)
```

uses it for the current state, upon receiving a value token V. Rules `[Ary-S]` and `[Ary-E]` work in a similar way, when brackets are consumed, except for two differences: First, when "`[`" is consumed, `[Ary-S]` pushes the current counter along with the current state onto the stack, then it resets the counter to zero, indicating the start of a new array. Correspondingly, when "`]`" is consumed, the stack top is popped out, and both the state and counter are reassigned with those values from the stack top. Finally, upon consuming a "`,`" (in an array), rule `[Com]` simply increments the counter.

Next, we explain how the query automaton can be driven by a recursive-descent parser.

Recursive-Descent Streaming. Recursive-descent parsing utilizes recursive functions to simplify the parser design [43]. It defines a function for each non-terminal in the grammar to recognize the corresponding structure(s). For non-terminals that are recursively defined, their functions naturally become recursive functions.

JSON has are two major non-terminals: `object` and `array`, after inlining other non-terminals (see Figure 2). Algorithm 1 presents their functions. Both recursively call to themselves and to each other. Thanks to the recursive design, the two functions are easy to follow—they recognize the structures of objects and arrays following their grammar rules. To achieve streaming query evaluation, we embed the query automaton transitions into the recursive descent parsing (highlighted in blue). Line 7 and 13 correspond to the `[Key]` and `[Val]` transitions in Figure 5, respectively. Similarly, Line 21 and 34 are about the `[Ary-S]` and `[Ary-E]` transitions. Finally, Line 30 corresponds to the `[Com]` transition. In addition, to output matches, the parser also checks the query automaton for the matching status (`qa.isAccept`). If it is in the ACCEPT state, the corresponding token would be outputted when it is consumed (e.g., at Line 2 and 5).

We name the above algorithm *recursive-descent streaming*. Next, we present the fast-forward ideas under this model.

3.2 Opportunities for Fast-Forward

By default, our streaming model processes the input token by token, so every token needs to be recognized and consumed. Moreover, these tokens are fed into the query automaton to make transitions. This actually may not be always necessary. Next, we show a series of cases where the streaming can fast-forward over certain data segments—their details need not to be fully examined, including the tokenization and syntactical analysis, not to mention query automaton updates. We next explain the intuitions behind this idea.

One intuition is about the *data type*, which can be inferred from the query, but has not yet been leveraged by the prior streaming model. For example, from `$.place.name`, we can infer that `place` is an object, as it has an attribute `name`. By contrast, expression `$.places[2:4].name` implies that `places` is an array. In this way, we can obtain the type of each attribute name in the path expression, except that at the last level (the inner-most level), which could be of any type. Based on such type information, the streaming may fast-forward over data segments with unmatched types. Besides that, the streaming may also leverage other query information, such as attribute names (e.g., `place`) and index constraints (e.g., `[2:4]`) to fast-forward over irrelevant syntactical structures.

Next, we categorize the fast-forward opportunities into five groups based on their intuitions.

G1: Fast-Forward to a Type-Specific Attribute or Element. An object consists of a list of attributes (see Figure 2), and the value of each attribute could be an object, an array, or a primitive. Based on the attribute type inferred from the query, we may fast-forward to the attribute of the matched type, except for the primitive. Because a primitive always appears at the last level of the path expression, thus the inference cannot find out if it is primitive or of other types. As Figure 6-(a) shows, based on the type of interest (an object), the streaming directly fast-forwards to the second attribute.

Similar scenarios also occur when the streaming traverses an array with elements of heterogeneous types. In this case, it may fast-forward to the element of matched type. Note that for queries with index constraints, we need to make sure that fast-forward does not exceed the specified index range.

G2: Fast-Forward over an Unmatched Attribute Value. For an attribute with matched type, the streaming extracts its name and feeds it to the query automaton to update the matching status. If the automaton fails to make matching progress—reaching the UNMATCHED state (state `0` in Figure 4), it would be unnecessary to further examine the attribute value. Depending on its type, the streaming may fast-forward over an object, an array, or a primitive. Figure 6-(b) shows an example where the streaming fast-forwards over the value of `user`, which fails to match "`place`".

G3: Fast-Forward over a Value and Output It. Once the query automaton reaches an ACCEPT state, the streaming would yield

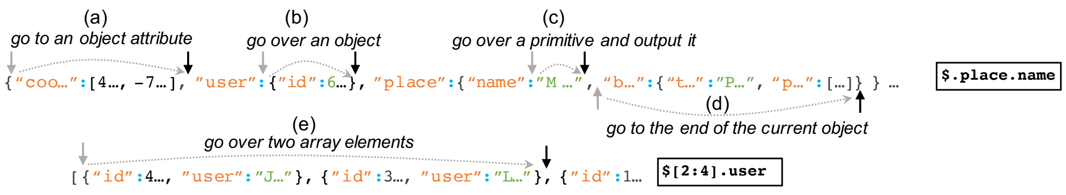
(a)
go to an object attribute
(b)
go over an object
(c)
go over a primitive and output it
{"coo…":[4…, -7…], "user":{"id":6…}, "place":{"name":"M …", "b…":{"t…":"P…", "p…":[…]} } …
$.place.name
(d)
go to the end of the current object
(e)
go over two array elements
[{"id":4…, "user":"J…"}, {"id":3…, "user":"L…"}, {"id":1…
$[2:4].user

Figure 6: Examples of Basic Fast-Forward Cases

Table 1: Five Groups of Fast-Forward Functions*

	Function	**Description** (*pos*: current streaming position)
G1	goToObjAttr()	move *pos* to the start of next object attribute
	goToAryAttr()	move *pos* to the start of next array attribute
	goToObjElem()	move *pos* to the start of next object element
	goToAryElem()	move *pos* to the start of next array element
	goToObjElem(K)	move *pos* ··· object elem. within K elements
	goToAryElem(K)	move *pos* ··· array elem. within K elements
G2	goOverObj()	move *pos* to the end of next object
	goOverAry()	move *pos* to the end of next array
	goOverPriAttr()	move *pos* to the end of next primitive attribute
	goOverPriElem()	move *pos* to the end of next primitive element
G3	goOverObj(out)	output an object and move *pos* to its end
	goOverAry(out)	output an array and move *pos* to its end
	goOverPriAttr(out)	output a prim. attr. and move *pos* to its end
	goOverPriElem(out)	output a prim. elem. and move *pos* to its end
G4	goToObjEnd()	move *pos* to the end of the current object
G5	goOverElem(K)	move *pos* to the end of next K elements
	goToAryEnd()	move *pos* to the end of the current array

*This is not a full list of functions; more functions are in Section 4.2.

an output (an attribute value or array element). In this case, the streaming may fast-forward the outputted part; the output details need not to be examined. Figure 6-(c) shows such a case where the query has been fully matched, so the streaming fast-forwards over the matched value and outputs it.

G4: Fast-Forward to the End of Current Object. In an object, once an attribute name is matched, the streaming may fast-forward over all following attributes of this object, as a JSON object cannot own duplicated attribute names, meaning no matches could be found in the remaining attributes. Figure 6-(d) shows a case where the streaming fast-forwards to the end of the object `place` after a prior attribute `name` has been matched.

G5: Fast-Forward over Out-of-Range Elements. Finally, some queries may carry array index constraints, like a range `[2:4]`. They expose opportunities to fast-forward over array elements out of the specified range. Figure 6-(e) shows that the streaming fast-forwards over the first two array elements as the query requests the third and the forth array elements (i.e., `[2:4]`). Similarly, fast-forward is also possible after the end of the specified range is reached.

Table 1 lists the above fast-forward cases as functions in five groups. Assume a global variable, `pos`, is kept during the streaming to mark the current position in the input, then fast-forward can be achieved by advancing `pos` to the corresponding target position. Before showing how these functions can be implemented, we first illustrate how they can be naturally integrated into the streaming.

3.3 Integration of Fast-Forward Functions

Due to space limits, in the following, we use the `object()` function in the recursive-descent streaming to illustrate the integration of fast-forward functions (see Algorithm 2).

First, the query automaton infers the attribute type of interest (Line 3), based on which, the streaming fast-forwards to the attribute of matched type (Line 6-9). If no more type-matched attributes exist, fast-forward reaches the end of the current object, leading to an early `return` of `object()` (Line 12). After the query automaton consumes the attribute name (Line 15), if it reaches the UNMATCHED state, it fast-forwards over the attribute value based on its type (Line 17-21). Otherwise, if the query automaton reaches the ACCEPT state, then the outputting-related functions would be called to output the value without examining its details (Line 22-26). Finally, either in the case of ACCEPT or MATCHED (made matching progress), the streaming would fast-forward to the end of the current object (Line 32-33), because only one attribute could be matched at each level.

Similarly, the fast-forward functions can also be integrated into the `array()` function in the recursive-descent streaming.

Note that some of the existing JSON parsers provide syntactical validation on the input. The use of fast-forward actions may affect this feature in the sense that the fast-forwarded data segments might not be fully validated due to the nature of fast-forwarding. As shown later, our implementations of fast-forward cases still offer certain validations like the parentheses/brackets pairing.

So far, we have presented the basic ideas of fast-forward, without showing their implementation details. In fact, the key challenge lies in implementing the fast-forward functions efficiently, which we will address in the next section.

4 BIT-PARALLEL FAST-FORWARDING

The basic question in implementing the fast-forward functions (e.g., going over an object) is the following:

how can the target position of fast-forward be identified?

Conventionally, this is a parsing problem. For example, to go over an object, the object needs to be recognized first, which requires tokenization and syntactical analysis. If the fast-forward functions are implemented in this way, the only benefit they bring would be skipping some query automaton updates. The main streaming cost, detailed parsing, would remain. To resolve this dilemma, we show a much faster way to implementing fast-forward functions—a highly bit-parallel solution, which bypasses the conventional parsing. The keys to this solution include an abstraction for partitioning the semi-structured data stream into basic units, *structural intervals*, and a *counting-based pairing strategy* for identifying the nesting levels and boundaries of objects and arrays.

Algorithm 2 Streaming with Fast-Forwarding (Partial)

```
function object()
    consume("{", qa.isAccept)  /* qa: query automaton */
    type_expected = qa.typeExpected()  /* infer the type */
    if notEmpty() then
        while true do
            switch type_expected do  /* fast-forward */
                case "object":  goToObjAttr()  break
                case "array":  goToAryAttr()  break
                case "unknown":
            if hasMoreAttributes() == false then /* if object ends */
                consume("}", qa.isAccept)
                return
            attribute = consumeAttrName(qa.isAccept)
            consume(":", qa.isAccept)
            qa.transition(attribute) /* update query automaton */
            type = getAttributeType()
            if qa.status == UNMATCHED then
                switch type do
                    case "object": goOverObj()  break
                    case "array": goOverAry()  break
                    case "primitive":  goOverPriAttr()
            else if qa.status == ACCEPT then
                switch type do
                    case "object": goOverObj(out)  break
                    case "array": goOverAry(out)  break
                    case "primitive":  goOverPriAttr(out)
            else if qa.status == MATCHED then
                switch type do
                    case "object": object()  break
                    case "array": array()  break
                    case "primitive":  primitive()
            if qa.status != UNMATCHED then
                goToObjEnd()
            qa.transition(V) /* V denotes all values */
            if hasMoreAttributes() then
                consume(",", qa.isAccept)
            else break
    consume("}", qa.isAccept)
```

4.1 Structural Intervals

Designing bit-parallel algorithms is notoriously challenging due to the low-level bit-manipulations. This is compounded by the recursive and nested structures of the data stream. To reduce the design complexity, we propose to partition the data stream into some basic data segments.

Definition 4.1. Given the current streaming position pos and a metacharacter of interest α, the ***structural interval*** for α is the sequence of consecutive characters between pos (inclusive) and the following closest α (exclusive).

Depending on the metacharacter of interest, there are different kinds of structural intervals, as demonstrated in Figure 7. Note that pseudo-metacharacters inside strings should be excluded.

A critical property of structural intervals is that they can be *constructed and accessed efficiently with bit-parallelism.*

Bit-Parallel Construction. Given the current streaming position pos, and a metacharacter α, the constructor builds an *interval bitmap* for characters in the data stream, such that only those bits within the interval are set to 1s, as illustrated by the "[" interval bitmap in Figure 8. The size of an interval bitmap is a word, denoted

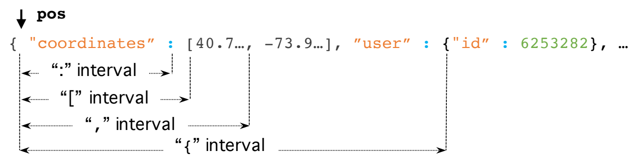
pos
{ "coordinates" : [40.7…, -73.9…], "user" : {"id" : 6253282}, …
":" interval
"[" interval
"," interval
"{" interval

Figure 7: Examples of Structural Intervals

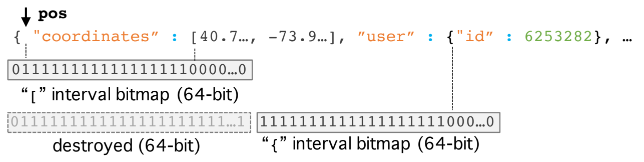
pos
{ "coordinates" : [40.7…, -73.9…], "user" : {"id" : 6253282}, …
01111111111111111110000…0
"[" interval bitmap (64-bit)
01111111111111111111111…1
destroyed (64-bit)
1111111111111111111000…0
"{" interval bitmap (64-bit)

Figure 8: Example of Interval Bitmap

as W. For example, on a 64-bit machine, the bitmap consists of 64 bits, representing 64 consecutive characters in the data stream.

If a structural interval spans multiple words, such as the "{" interval in Figure 7, the constructor builds interval bitmaps word by word, as shown in Figure 8. Note that an interval bitmap should be constructed after the prior one has been used and destroyed, to comply with the streaming design.

Function `builtInterval(pos, char)` in Algorithm 3 describes the construction process. First, it builds a bitmap for the given metacharacter (Line 3). The basic idea [34, 40, 41] is shown by function `builtMetacharBitmap(char)`. Here, we need to remove all the pseudo-metacharacters. To achieve this, a string bitmap is created (Line 17), then a logic AND is taken between the string bitmap and the raw metacharacter bitmap (Line 20). After this, the constructor marks the start position (Line 4), resets bits up to the start position to 0s (Line 5-6), marks the end position (Line 7), and generates the interval bitmap with a bitmap subtraction (Line 8).

In addition, Algorithm 3 also presents an alternative constructor, `nextInterval(char)`, for quickly building a series of structural intervals, and function `intervalEnd(interval)` for obtaining the end position of an interval. The latter uses leading-zero counting instruction to locate the mirrored end position [2], then converts it to the actual end. All the above designs expose bit-parallelism.

Next, we will show how the structural intervals can be leveraged for implementing the fast-forward functions.

4.2 Fast-Forward Algorithms

We start with the main design, then present the algorithm in detail for each group of fast-forward functions.

Main Design. Conceptually, the data stream is processed *interval by interval* to find the target position of fast-forward. However, as an interval may span multiple words, the data stream is actually processed word by word (see Section 4.1). First, every W characters (word size) in the data stream are converted into relevant interval bitmaps. Then, based on these bitmaps, fast-forward employs a

[2]In fact, a metacharacter bitmap is always created in mirrored way [34, 40, 41].

Algorithm 3 Structural Interval Construction and Access

```
/* Build an interval bitmap for a metacharacter */
function buildInterval(pos, char)
    bitmap = buildMetacharBitmap(char)
    b_start = 1 << pos /* mask start position */
    mask_start = b_start^(b_start − 1) /* mask bits up to start */
    bitmap = bitmap & ~mask_start /* reset bits up to start to 0s */
    b_end = bitmap & −bitmap /* mask end position */
    b_interval = b_end − b_start /* create the interval bitmap */
    return b_interval

/* Construct the bitmap for a given metacharacter */
function buildMetacharBitmap(char)
    if bitmap_char != null then /* to avoid rebuilding */
        return bitmap_char
    /* construct a bitmap to mask all the characters inside strings */
    if bitmap_string == null then
        bitmap_string = buildStringBitmap() /* from [34, 40] */
    bitmap_char = buildRawCharBitmap(char)
    /* remove metacharacters in strings */
    bitmap_char = bitmap_char & bitmap_string
    return bitmap_char

/* Build an interval bitmap between first two 1s in a bitmap */
function nextInterval(char)
    bitmap = buildMetacharBitmap(char)
    b_start = bitmap & −bitmap /* get rightmost 1 */
    bitmap = bitmap & (bitmap − 1) /* remove rightmost 1 */
    b_end = bitmap & −bitmap /* get rightmost 1 again */
    b_interval = (b_end − b_start) /* create the interval bitmask */
    return b_interval

/* Get the position of the end of an interval (i.e., rightmost 1) */
function intervalEnd(interval)
    pos = lzcnt(interval) /* position of leftmost 1 */
    pos = wordSize() − pos /* mirror the position */
    return pos
```

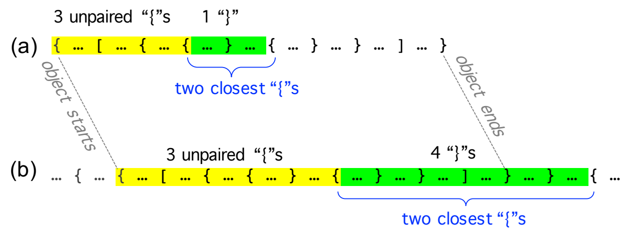

Figure 9: Pairing Properties within and across Objects

counting-based pairing strategy to locate the end of an object or array. The strategy is backed up by the following property of JSON:

Lemma 4.2. *In a nested JSON object, assume α and β are two closest "{"s, and the number of "}"s between them is n_{close}. Also assume the number of unpaired "{"s before α, including α, is n_{open}. Then, based on the pairing between "{" and "}", it is obvious that $n_{close} < n_{open}$. A similar conclusion can be drawn for a nested JSON array regarding "[" and "]".*

Figure 9-(a) shows an example: between the two closest "{"s marked on the bottom, $n_{close} = 1$, while $n_{open} = 3$.

Based on Lemma 4.2, we can infer the following pairing property for a data stream with multiple JSON objects.

Algorithm 4 G2 Fast-Forward Functions

```
1:  function goOverObj()
2:      consume("{")
3:      num_{ = 1 /* number of unpaired "{" */
4:      while true do
5:          if interval == null then
6:              interval = buildInterval(pos,"{")
7:          else
8:              interval = nextInterval("{")
9:          bitmap_} = getMetacharBitmap("}")
10:         bitmap_} = bitmap_} & interval /* get "}"s within interval */
11:         num_} = __popcnt(bitmap_}) /* count "}"s in this interval */
12:         if num_} < num_{ then
13:             num_{ = num_{ − num_} + 1
14:         else /* enough or more "}"s found */
15:             pos = getPosition(bitmap_}, num_{) /* object end */
16:             break
17:
18: function goOverPriAttr()
19:     interval = buildInterval(pos, ",")
20:     bitmap_} = getMetacharBitmap("}")
21:     bitmap_} = bitmap_} & interval
22:     if bitmap_} == 0 then /* current object does not end */
23:         pos = intervalEnd(interval)
24:     else
25:         pos = getPosition(bitmap_}, 1) − 1
```

Theorem 4.3. *Consider an object in a JSON stream, assume α is a "{" within the object, and β is its closest next "{". Also assume the number of "}"s between them is n_{close}, and the number of unpaired "{"s in this object before α, including α, is n_{open}. Then, if $n_{close} \geq n_{open}$, the object ends between α and β. A similar conclusion can be drawn for a JSON stream with multiple arrays regarding "[" and "]".*

Figure 9-(b) shows an example: between the two closest "{"s marked on the bottom, $n_{close} = 4$, while $n_{open} = 3$. As a result, the object of interest, which is from Figure 9-(a), ends between the two referred "{"s. In fact, based on the value of n_{open}, we can locate where the object ends—the third "}" between the two referred "{"s.

Based on the above insights, we first discuss the algorithm for fast-forward functions in G2 group, then move to the discussions of algorithms for other function groups.

Algorithms of G2 Functions. Algorithm 4 shows the algorithms of two representative functions. Take function `goOverObj()` as an example. The algorithm first consumes the beginning "{" and sets the number of unpaired "{"s to 1 (Line 2-3). Then, it builds the "{" interval (Line 5-8), and counts the number of "}"s in this interval (Line 9-11). If that is less than the number of unpaired "{"s so far, the latter would be updated by subtracting the number of "}"s, plus the new one at the beginning of this interval (Line 13). If there are more "}"s, the object must end within this interval, and its ending "}" should be the `num_{`-th one (Line 15).

The algorithm for `goOverPriAttr()` builds a comma interval first (Line 21), then collects the "}"s within this interval (Line 22-23). If no "}" exists in this interval, it sets the streaming position pos with the interval end; otherwise, it moves the pos to the position before the object ends (Line 24-27). The algorithm for `goOverAry()` and `goOverPriElem()` can be designed similarly, except that all the occurrences of parentheses are replaced with brackets.

Algorithms of G1 Functions. Algorithm 5 shows the algorithm of a representative function `goToObjAttr()`. To fast-forward to an

Algorithm 5 G1 Fast-Forward Functions

```
function goToObjAttr() /* Go to the next attribute */
    while hasMoreAttributes() do
        interval = buildInterval(pos, ":")
        pos = intervalEnd(interval) + 1
        type = getAttributeType()
        switch type do
            case "primitive": goOverPriAttrs()  break
            case "object":  break while-loop
            case "array":  goOverAry()

function goOverPriAttrs() /* Go over primitive attributes */
    interval = buildInterval(pos, "{", "[")
    bitmap_} = getMetacharBitmap("}")
    bitmap_} = bitmap_} & interval
    if bitmap_} == 0 then /* object has not ended */
        pos = intervalEnd(interval)
    else
        pos = getPosition(bitmap_}, 1)
```

attribute of object-type, the algorithm builds a colon interval to quickly reach the next attribute (Line 3-4). This allows it to check the type of its value without extracting the attribute name (Line 5). If the value is not an object, the algorithm calls a fast-forward function to go over the value. The algorithm stops when it reaches the object-type attribute or the end of the object.

The algorithms for other G1 functions are similar, except that for going to a type-specific array element, there is no need to construct the colon intervals, and for queries with index constraints, the fast-forward should track a counter to stay within the range.

In addition, Algorithm 5 also shows an enhanced fast-forward function `goOverPriAttrs()` to go over some primitive attributes together. The idea is to move as far as possible till it reaches the next "`{`" or "`[`" (Line 12). If the current object has not ended (Line 15), then it has successfully went through these primitive attributes. Otherwise, it goes to the end of the object (Line 18). A function `goOverPriElems()` can be implemented similarly to fast-forward over a sequence of primitive elements in an array.

Algorithms of G3 Functions. Their algorithms are similar to those for G2 functions, except that they also output the characters that have been fast-forwarded. We can simply embed the outputting statements into the G2 functions at the interval level.

Algorithm of G4 Function. Function `goToObjEnd()`, in fact, can be implemented like `goOverObj()`, because both of them want to reach the end of an object. The only difference is that `goToObjEnd()` occurs inside an object (between two attributes), while `goOverObj()` occurs right before an object. By removing the `consume("{")` at Line 2 in Algorithm 4, we can get the algorithm for `goToObjEnd()`.

Algorithms of G5 Functions. Function `goOverElems(`K`)` can be implemented by traversing the comma intervals with an index counter. For each element, a fast-forward function is called based on its type. Function `goToAryEnd()` can be implemented similarly like `goToObjEnd()` except that the parenthesis bitmaps are replaced with the bracket bitmaps.

So far, we have presented the algorithms for all the five groups of fast-forward functions listed in Table 1.

Table 2: Methods in Evaluation

JPStream	A state-of-the-art JSON streaming library [35]
RapidJSON	A popular conventional JSON parser from Tencent [11]
simdjson	A popular SIMD-based JSON parser [40]
Pison	A structural index-based JSON preprocessor [34]
JSONSki	JSON streaming with bit-parallel fast-forward

5 EVALUATION

This section evaluates the proposed streaming with bit-parallel fast-forwarding and compares it with the existing methods.

5.1 Methodology

We implemented the fast-forward functions discussed above and the recursive descent streaming in C++. Together, they are referred to as *JSONSki*. JSONSki supports basic JSONPath notations, including root ("`$`"), child operator ("`.`" or "`[]`"), array index ("`[n]`"), index range ("`[m:n]`"), and wildcard ("`*`"). One missing operator in the current version is descendant elements ("`..`"), which may limit fast-forward capabilities, as the types of the descendant elements cannot be inferred. We plan to add support for this operator in the future. All the bitwise and SIMD instructions that JSONSki utilizes are commonly available on modern CPUs.

Though JSONSki invokes fast-forward functions based on the path queries, developers may exploit these fast-forward functions for more opportunities in their own JSON analytics.

Processing Scenarios. We cover two common scenarios of JSON data processing: (i) a single large record; and (ii) a sequence of small records. The second scenario exposes task-level parallelism, while the first scenario does not. All inputs are preloaded into the memory before the processing. Each input with small records is stored in an array, along with an offset array for starting positions.

Table 3: Feature Comparison among Methods

Method	Processing Strategy	Speculative Parallelism	Bitwise Parallelism	Fast-forward
JPStream	Streaming	✓	–	–
RapidJSON	Preprocessing	–	–	–
simdjson	Preprocessing	–	✓	–
Pison	Preprocessing	✓	✓	–
JSONSki	Streaming	–	✓	✓

Methods. We compare JSONSki with several representative JSON tools: JPStream [35], simdjson [40], RapidJSON [11], and Pison [34]. Tables 2 and 3 list these methods and their relevant features. Note that, though simdjson and Pison use bitwise and SIMD parallelism, their uses are limited to the identification of metacharacters. On the other hand, JPStream and Pison support speculative execution, allowing them to process a single (large) record in parallel. The current design of JSONSki lacks speculation, though we are not aware of any parts of its design prevent it from adopting speculation optimization. In our evaluation, we enable speculation for JPStream and Pison in the single large record processing scenario; many small records can already be processed in parallel—speculation only hurts the performance for its extra costs.

Table 4: Dataset Statistics

Data	#objects	#arrays	#attr	#prim.	#sub	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

Datasets. The datasets are collected from real-world applications, including tweets stream from Twitter (TT) developer API [14], product dataset from Best Buy (BB) [1], National Statistics Postcode Lookup (NSPL) dataset from United Kingdom [9], Google Maps Directions (GMD) dataset [3], Walmart (WM) product dataset [10], and Wikipedia (WP) entity dataset [16]. Table 4 lists the structural statistics. For easier comparison, we made each dataset roughly the same size—1GB. Most datasets are in two formats: one single large record or a series of small records. The column #sub lists the number of small records in each dataset.

Table 5: JSONPath Queries

ID	Query structure	#matches
TT1	$[*].en.urls[*].url	88,881
TT2	$[*].text	150,135
BB1	$.pd[*].cp[1:3].id	459,332
BB2	$.pd[*].vc[*].cha	8,857
GMD1	$[*].rt[*].lg[*].st[*].dt.tx	1,716,752
GMD2	$[*].atm	270
NSPL1	$.mt.vw.co[*].nm	44
NSPL2	$.dt[*][*][2:4]	3,509,764
WM1	$.it[*].bmpr.pr	15,892
WM2	$.it[*].nm	272,499
WP1	$[*].cl.P150[*].ms.pty	15,603
WP2	$[10:21].cl.P150[*].ms.pty	35

Path Queries. Table 5 lists all the JSONPath queries used in the evaluation. For each dataset, we constructed two queries. The last column lists the number of matches. Together, they provide a good coverage of the common path query structures, as well as different levels of complexity and selectivity. Similar query structures have been used for evaluation by the prior work [34, 35, 41].

All experiments ran on a server with two Intel 2.1GHz Xeon E5-2620 v4 CPUs and 128GB RAM. The CPUs support 64-bit ALU instructions and 256-bit SIMD instruction set. This server runs on CentOS 7 and is installed with G++ 7.4.0. All C++ programs were compiled with "-O3" flag.

5.2 Overall Performance

We first compare the performance of different methods under the two basic processing scenarios.

Performance on Single Large Records. First, we evaluated all the methods on the single large record of each dataset (1GB per record). Figure 10 reports the total execution time. For preprocessing-based methods, it includes preprocessing and querying time. For JPStream and Pison, Figure 10 also reports their parallel execution time using 16 speculative threads (the same as the #cores).

First, the results clearly show that JPStream and RapidJSON run significantly slower than the other methods. This is mainly due to the character-by-character processing and the lack of bitwise and SIMD parallelism. By contrast, simdjson, Pison, and JSONSki simultaneously process a batch of characters. In specific, JSONSki runs 12.3× faster than JPStream, confirming the importance of adopting bit-parallelism in semi-structured data streaming.

Among the methods with bit-parallelism, JSONSki achieves 4.8× speedup over simdjson and 3.1× over Pison, on average. These benefits mainly come from three aspects: First, JSONSki not only utilizes bit-parallelism for identifying metacharacters, but also uses it for locating the boundaries of objects and arrays (see Section 4). Second, JSONSki can fast-forward over a large ratio of the data stream, without examining its details (see Section 3.2). Third, with streaming design, JSONSki consumes the data within the caches. By contrast, both simdjson and Pison need to first construct some data structure (a parse tree or indexing bitmaps) before evaluating the queries. Note that Pison can also skip parsing some parts of the data stream in detail, however, to achieve that, it still needs to build the structural indices for the entire data stream beforehand.

Note that for a single record, JSONSki has to traverse it in serial due to the dependences involved. In comparison, both JPStream and Pison support speculative execution to break dependences. When the speculation is enabled, the results show that the single-threaded JSONSki still beats JPStream with 16 threads by about 28%, but runs slower than Pison with 16 threads by 48%. We expect the slowdown would be addressed after speculation is added to JSONSki.

In addition, we noticed that JSONSki takes significantly less time (around 0.01s) in two cases: NSPL1 and WP2. It turns out that the queries find all matches in the early part of the data stream, so JSONSki fast-forwards a high ratio of data stream, which will be discussed in detail in Section 5.3.

Performance on Small Records. Figure 11 reports the execution time in processing a sequence of small records using a single thread. Overall, the performance results are similar to those on the large ones, except that most methods become slightly faster, thanks to the better cache locality—small records may fit into the caches. Note that JSONSki runs a bit longer on average (from 0.40s to 0.41s), due to the exclusion of two cases (NSPL1 and WP2) which are not applicable to the small-record scenario.

Figure 12 reports the time of parallel execution using 16 threads, where each thread is assigned to process one small record each time. The results indicate that JSONSki achieves 9.5× and 3.0× speedups over JPStream and Pison, respectively. Also, comparing with the serial performance (Figure 11), we find that JPStream, Pison, and JSONSki, scale reasonably well on the 16-core machine, realizing speedups of 11.9×, 11.8×, and 10.3×, respectively.

Memory Overhead. The memory cost is also critical to applications of semi-structured data. Figure 13 presents the memory footprints of different methods in processing a large record. For their streaming design, JPStream and JSONSki only take around 1GB (their memory consumption is actually configurable by adjusting the input buffer size). While the other methods need a substantial amount of extra memory for holding the preprocessing results, like a parse tree (simdjson and RapidJSON) or structural indices (Pison). On average, their memory costs are around 2-3GB.

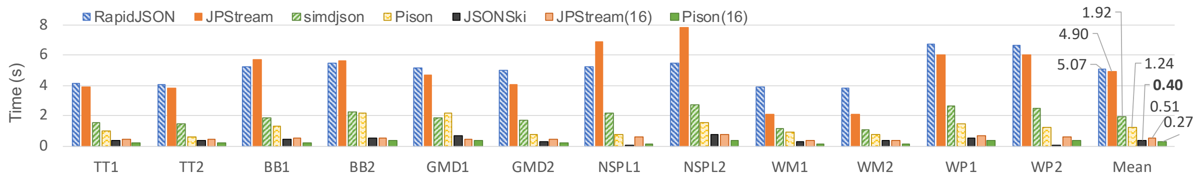

Figure 10: Performance on a Single Large Record (JPStream(16) and Pison(16) use 16 speculative threads)

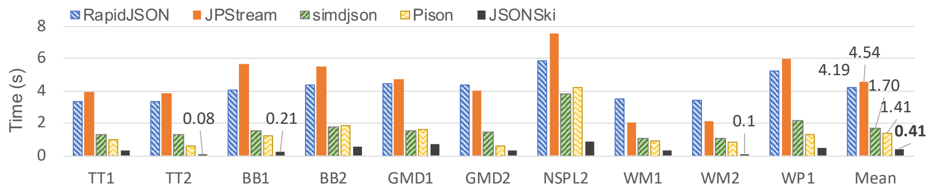

Figure 11: Sequential Performance on a Series of Small Records (All methods use a single thread)

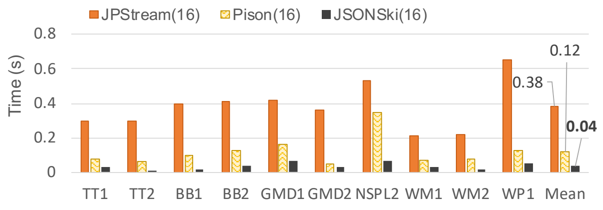

Figure 12: Parallel Performance on a Series of Small Records

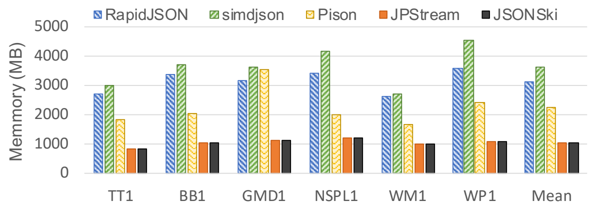

Figure 13: Memory Footprint Comparison

Table 6: Fast-Forward Ratios by Function Groups

Query	G1	G2	G3	G4	G5*	Overall
TT1	**12.80%**	**78.22%**	0.22%	**8.20%**	–	99.44%
TT2	0.00%	1.17%	2.28%	**95.62%**	–	99.07%
BB1	**14.34%**	0.72%	0.49%	**82.19%**	0.75%	98.49%
BB2	**89.24%**	**8.73%**	0.02%	< 0.01%	–	97.99%
GMD1	**13.18%**	0.04%	1.06%	**83.13%**	–	97.41%
GMD2	0.02%	**99.97%**	< 0.01%	0.00%	–	99.99%
NSPL1	< 0.01%	< 0.01%	< 0.01%	**99.99%**	–	99.99%
NSPL2	**83.45%**	0.00%	1.55%	< 0.01%	**10.94%**	95.94%
WM1	**97.97%**	0.13%	0.01%	1.66%	–	99.77%
WM2	< 0.01%	0.33%	1.90%	**96.56%**	–	98.79%
WP1	1.47%	**83.08%**	0.01%	**14.77%**	–	99.33%
WP2	< 0.01%	0.02%	< 0.01%	0.01%	**99.96%**	99.99%

*G5 functions are only for queries with index constraints

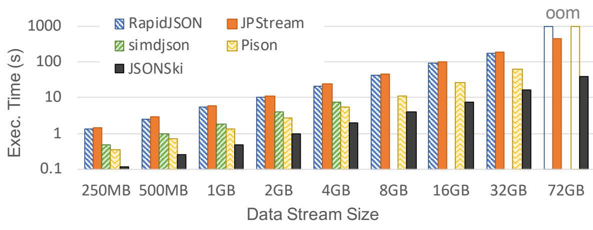

Figure 14: Scalability Comparison (BB1)
(Both axes are in log scale; simdjson supports up to 4GB.)

5.3 Benefits Breakdown

To examine the benefits of JSONSki, we define *fast-forward ratio*—the ratio between the characters fast-forwarded and the total data stream length, to estimate the effectiveness of fast-forward. Table 6 reports these ratios under different fast-forward function groups in the scenario of single large record processing.

First, as the last column of Table 6 shows, the overall fast-forward ratios for all five groups of functions are very high across all the evaluated queries—all above 95%. This confirms the effectiveness of fast-forward in practice. For less than 5% characters that are not fast-forwarded, they are mostly the attribute names and some metacharacters that have to be extracted for the matching purpose.

Second, the results reveals the uneven contributions of different fast-forward functions to the overall fast-forward ratio. Non-trivial contributions (> 5%) are bolded in Table 6. For example, under TT1, G2 functions achieve a ratio of 78.22%, while G3 functions only fast-forward over 0.22% of the data. In general, the contributions of functions highly depend on the matching process. Except for G3 functions, all other four groups of fast-forward functions made non-trivial contributions in at least one query. For G3 functions, their effectiveness highly depends on the querying selectivity, as they are designed for outputting. For queries with relatively lower selectivity, their contributions tend to be higher.

5.4 Scalability

Finally, we briefly demonstrate the scalability of JSONSki in terms of input size of a single large record. We constructed a data stream of the BB dataset with its size ranging from 250MB to 72GB (the tested machine has 128GB memory). Figure 14 reports the execution time of different methods under query BB1. The results show that, as the data stream size increases, the execution time of JSONSki tends to grow linearly just like the other methods. In addition, we find that the two processing-based methods (RapidJSON and Pison) failed to scale up to an input of 72GB; both of them suffered from the out-of-memory error. The other processing-based method simdjson only supports JSON records up to 4GB.

6 RELATED WORK

This section discusses existing work on semi-structured data under three topics: i) character-by-character processing, ii) bit-parallel processing, and iii) fast-forwarding and filtering.

Character-by-Character Processing. Most existing works on raw semi-structured data processing follows a character-by-character processing style. For XML querying, existing methods are either based on automata [22, 28, 30, 46, 53], arrays [37], or stacks [26, 38]. Similarly, most existing JSON query evaluators [2, 7, 8, 11] convert each JSON record into in-memory tree structure before getting the substructure of interests.

However, as discussed earlier, preprocessing may introduce an upfront delay and requires extra memory. To address such issues, streaming scheme has been proposed, first for XML data [30, 36, 46] then for JSON data, such as JSONSurfer [13] and JPStream [35]. More details about the streaming design have been provided as background in Section 2.

Bit-Parallel Processing. Recently, some works utilize SIMD and bitwise parallelism to accelerate semi-structured data processing. Mison [41] leverages bit-parallelism to create structural indices for a single JSON record. To improve the scalability, Pison [34] proposes customized parallelization techniques (e.g., speculation) to build indices in parallel for a single JSON record. Some of these ideas are inspired by NoDB [18, 19, 33, 39], which builds structural index on raw CSV files. Unlike the above methods, simdjson [40] adopts bitwise processing to speed up the parsing tree generation. All the above methods use bit-parallelism to build in-memory data structures during the preprocessing. By contrast, JSONSki leverages bit-parallelism to accelerate the fast-forward during the streaming.

Besides JSON, bitwise parallelism has also been applied for other data processing, like regular expression matching [24], finite-state machine [49], bit-stream processing [48], substring searching [24, 47], XML parsing [42], delimiter-separated data parsing [50], and data processing in some database systems [17, 23, 27, 45].

Fast-Forwarding and Filtering. To boost the performance of JSON data processing, Sparser [47] filters out certain JSON records before parsing and querying, based on the substrings appeared in the query. If a JSON record contains no specified substrings, it would be filtered out. Clearly, its effectiveness depends on the query selectivity. If a query can find matches in most records, the filtering only brings limited benefits (or even slows it down). Note that this filtering is orthogonal to the fast-forward in this work. The former is inter-record and is unaware of the record structure, while the latter is intra-record and is based on the record structure.

As introduced earlier, Mison [41] and Pison [34] also try to skip some detailed parsing. They achieve this by building structural indices for the entire record beforehand (see Section 2).

For XML data, some earlier work mentioned fast-forward, but the contexts and the high-level ideas are different from this work. First, XHints [31] proposes to insert artificial "hints" into the XML data stream to indicate the portions to be fast-forwarded. However, this solution sacrifices the generality, as the "hints" carry special semantics beyond the XML syntax. Alternatively, Amagasa and others [20] propose to fast-forward XML elements that fail to match the path queries, similar to our G2 functions. To do so, they designed a streamlined parser to fast-forward XML attributes.

Though the above works explored ideas similar to fast-forward, there are several special challenges this work needs to address in the *streaming* context. First, unlike prior works where bitmaps (indices) are constructed for the entire record step by step, streaming scheme lacks a "global view" —it is only aware of the characters that it has already processed. This requires creating bitmaps in the absence of well-defined start and end positions of objects, arrays, and etc. To deal with the "missing contexts", we introduced structural intervals, which are essentially partial bitmaps with varying lengths.

Second, in the preprocessing scheme, the bitmaps (indices) are complete, which makes the fast-forward straightforward: jump to a colon (in an object) to match a field or count the commas in an array to match the elements. By contrast, due to the unavailability of the "global bitmaps", the fast-forward algorithms in the streaming scheme needs to be customized under different "partial contexts"— as shown by the diversity of fast-forward functions in this work.

At last, it is non-trivial to integrate the fast-forward ideas into the existing streaming processor (JPStream) due to its formal design. To fill this gap, this work designed a new streaming model based on recursive-descent parsing to easily adopt fast-forward ideas.

7 CONCLUSION

Streaming query evaluation is a promising scheme for processing semi-structured data. However, its conventional design requires to parse the entire data stream in detail. In this work, we reveal a series of opportunities of fast-forward over substructures of the data stream that are irrelevant to the query evaluation. To tap into its full potential, we propose a set of bit-parallel algorithms for realizing different fast-forward cases. Based on these, we developed JSONSki, a new JSON streaming framework. Our evaluation shows that JSONSki is able to fast-forward over 95% of the data stream for common query structures, bringing significant speedups over the existing streaming and non-streaming methods.

ACKNOWLEDGMENTS

We thank all anonymous reviewers for their constructive comments and our shepherd Dr. Linhai Song for his time and efforts in helping with the paper revision. This material is based upon work supported by the National Science Foundation under Grant No. 1751392. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

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.]. Introducing JSON. https://www.json.org/. Retrieved: 2019-07-01.

[6] [n. d.]. A Java serialization/deserialization library to convert Java Objects into JSON and back. https://github.com/google/gson. Retrieved: 2020-05-10.

[7] [n. d.]. JSON-C - A JSON implementation in C. https://github.com/json-c/json-c/. Retrieved: 2020-05-01.

[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.]. Run JavaScript Everywhere. https://nodejs.dev/. Retrieved: 2019-07-01.

[13] [n. d.]. A streaming JsonPath processor in Java. https://github.com/jsurfer/JsonSurfer/. Retrieved: 2020-05-01.

[14] [n. d.]. Twitter Developer API. https://developer.twitter.com/en/docs/. Retrieved: 2019-07-01.

[15] [n. d.]. Why JSON will continue to push XML out of the picture. https://www.ctl.io/developers/blog/post/why-json-will-continue-to-push-xml-out-of-the-picture. Retrieved: 2019-07-01.

[16] [n. d.]. Wikimedia Miscellaneous Files. https://archive.org/details/wikidata-json-20150202. Retrieved: 2020-05-20.

[17] 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.

[18] 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.

[19] 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.

[20] Toshiyuki Amagasa, Mana Seino, and Hiroyuki Kitagawa. 2013. Energy-efficient XML stream processing through element-skipping parsing. In *2013 24th International Workshop on Database and Expert Systems Applications*. IEEE, 254–258.

[21] Amazon. 2021. Extracting Data from JSON. https://docs.aws.amazon.com/athena/latest/ug/extracting-data-from-JSON.html. Retrieved: 2021-07-01.

[22] 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.

[23] 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.

[24] 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.

[25] Arijit Chakraborty. [n. d.]. An Introduction to REST and JSON. https://blogs.oracle.com/cloud-platform/an-introduction-to-rest-and-json. Retrieved: 2019-07-01.

[26] 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.

[27] 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.

[28] 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.

[29] Chris Esplin. [n. d.]. Firebase Data Modeling. https://howtofirebase.com/firebase-data-modeling-939585ade7f4. Retrieved: 2019-07-01.

[30] 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.

[31] Akhil Gupta and Sudarshan S Chawathe. 2004. *Skipping Streams with XHints*. Technical Report.

[32] Stefan Gössner. [n. d.]. JSONPath - XPath for JSON. http://goessner.net/articles/JsonPath/. Retrieved: 2018-07-01.

[33] 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*.

[34] Lin Jiang, Junqiao Qiu, and Zhijia Zhao. 2020. Scalable structural index construction for JSON analytics. *Proceedings of the VLDB Endowment* 14, 4 (2020), 694–707.

[35] 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

[36] 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.

[37] Vanja Josifovski, Marcus Fontoura, and Attila Barta. 2005. Querying XML streams. *The VLDB Journal* 14, 2 (2005), 197–210.

[38] 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.

[39] 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.

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

[41] 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

[42] 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

[43] C Louden Kenneth. 1997. Compiler Construction: Principles and Practice. *Course Technology* (1997).

[44] MongoDB. [n. d.]. MongoDB Extended JSON. https://docs.mongodb.com/manual/reference/mongodb-extended-json/. Retrieved: 2019-07-01.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] Junqiao Qiu, Zhijia Zhao, and Bin Ren. 2016. Microspec: Speculation-centric fine-grained parallelization for FSM computations. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. 221–233.

[50] 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

[51] Twitter. 2021. Data dictionary: Standard v1.1. https://developer.twitter.com/en/docs/twitter-api/v1/data-dictionary/object-model/geo. Retrieved: 2021-07-01.

[52] TwoBitHistory. [n. d.]. The Rise and Rise of JSON. https://twobithistory.org/2017/09/21/the-rise-and-rise-of-json.html. Retrieved: 2019-07-01.

[53] 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.