Scalable Processing of Contemporary Semi-Structured Data on Commodity Parallel Processors – A Compilation-based Approach

Lin Jiang, Xiaofan Sun, Umar Farooq, and Zhijia Zhao
University of California, Riverside
ljian006@ucr.edu,xsun042@ucr.edu,ufaro001@ucr.edu,zhijia@cs.ucr.edu

Abstract

JSON (JavaScript Object Notation) and its derivatives are essential in the modern computing infrastructure. However, existing software often fails to process such types of data in a scalable way, mainly for two reasons: (i) the processing often requires to build a memory-consuming parse tree; (ii) there exist inherent dependences in processing the data stream, preventing any data-level parallelization.

Facing the challenges, developers often have to construct ad-hoc pre-parsers to split the data stream in order to reduce the memory consumption and increase the data parallelism. However, this strategy requires more programming efforts. Moreover, the pre-parsing itself is non-trivial to parallelize, thus introducing a new serial bottleneck.

To solve the dilemma, this work introduces a scalable yet fully automatic solution – a compilation system, namely ***JPStream***, that compiles standard JSONPath queries into parallel executables with bounded memory footprints. First, JPStream adopts a stream processing design that combines the querying and parsing into one pass, without generating any in-memory parse tree. To achieve this, JPStream uses a novel joint compilation technique that compiles the queries and the JSON syntax together into a single automaton.

Furthermore, JPStream leverages the "enumerability" of automaton to break the dependences and reason about the transition rules to prune infeasible cases. It also features a module that learns data constraints from the input data to enhance the pruning. Evaluation on real-world JSON datasets with standard JSONPath queries shows that JPStream can reduce the memory consumption significantly, by up to 95%, meanwhile achieving near-linear speedup on multicore and manycore processors.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304008

CCS Concepts • **Information systems → Query languages**; **Semi-structured data**; • **Theory of computation → Grammars and context-free languages**; • **Software and its engineering → Parsers**; • **Computer systems organization → Multicore architectures**.

Keywords JSON, semi-structured data, querying, parsing, pushdown automata, parallelization, multicore

ACM Reference Format:
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 *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304008

1 Introduction

JSON (JavaScript Object Notation) and its derivative data types (such as NetJSON [34], GeoJSON [33], JSON-LD [35], CoverageJSON [30], and etc.) form a family of contemporary semi-structured data (herein referred to as *JSON-family data types* or simply *JSON*). Together, they play a fundamental role in the modern computing infrastructure, ranging from cloud computing [24, 45] and microservice architectures [29, 61], to Internet of Things (IoT) [62] and NoSQL data stores [36, 43]. For example, major cloud providers, such as Azure [5], AWS [3], Firebase [31], and Oracle Cloud [25], all support JSON-based cloud services. Document data stores, such as MongoDB [49] and CouchDB [21], are built on JSON data.

Not only the popularity, but also the volume of JSON data grows quickly in recent years. For instance, the NASA Earth Exchange (NEX) project yielded over 20 TB climate data that is accessible from Amazon servers via JSON APIs [46]. Twitter produces tweets as a JSON data stream at a rate of 600 million per day [17]. Public data sources, like data.gov [8], provide REST APIs to access a broad range of scientific data primarily in JSON format, with sizes quickly increasing.

To efficiently process large-volume text data, recent work proposes hardware-implemented automata, such as automata processor [28], cache automata [59], and in-SRAM pushdown automata [20]. Despite their promising results, they are not yet readily available to the developers.

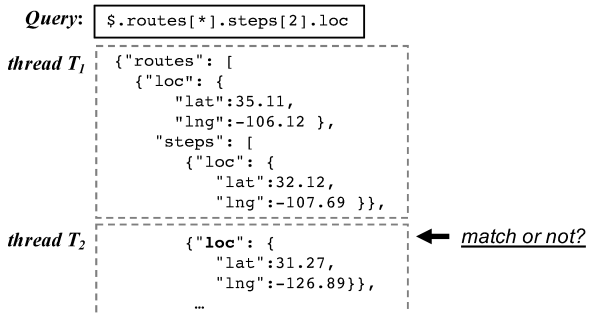

Figure 1. Challenge in Parallel JSON Data Processing[1].

In comparison, this work focuses on the use of commodity multicore and manycore processors to accelerate JSON data processing, which, unfortunately, remains an open problem, due to the following two challenges.

Challenge I - Preprocessing Costs. Traditional strategies for processing semi-structured data require a full parsing over the entire data stream before extracting information (substructures of interests). This often creates a high memory pressure for large data streams. For example, JSON-C [10], a popular JSON parser requires 8GB memory to process a 1GB JSON stream. Even worse, loading a JSON file with hundreds of megabytes can easily raise an `out-of-memory` exception on CouchDB [21]. In addition to the high-memory cost, preprocessing introduces significant yet unnecessary delay as the substructures out of interests are also processed anyway. These high costs of preprocessing greatly limit the scalability of semi-structured data processing.

Challenge II - Inherent Dependences: Exposing effective parallelism is key to the scalable data processing on parallel computer architectures. Unfortunately, semi-structured data processing exposes limited parallelism due to its inherently nested structures. Figure 1 illustrates this challenge with a piece of Google route data. Suppose the user is interested in the location of the third step in each route, denoted as a path expression `$.routes[*].steps[2].loc`. Except for the first thread, all the other threads cannot determine if the key `loc` satisfies the path conditions or not, due to the lack of relevant contexts. In general, such nested structures of the data serialize the processing of the whole JSON stream.

State of The Art. To cope with the preprocessing costs, stream programming paradigms, such as the SAX (Simple API for XML) model [38, 53], have been introduced. But, they simply provide callback APIs for handling the basic tokens, still requiring substantial programming efforts to design and implement a particular processing task. A more promising solution is to automatically generate deterministic automata to process the data in a streaming fashion [32]. However, so far, the automata are designed only for conventional XML. It remains an open question how to construct the automata for stream processing JSON data, which follows quite different syntactical structures (see Section 2).

As to parallelization, prior efforts propose to pre-scan the data to find "good" boundaries for partitioning [47, 48, 63]. However, this strategy has several limitations. First, it adds an extra pass over the entire data stream, which could be expensive for large data streams; Second, the pre-scan itself is a serial process; Third, despite the availability of "good" boundaries, they may not separate the data evenly, leading to potential load imbalance. At last, this strategy requires developers to write the pre-scanner for in a specific JSON structure, adding extra programming burden.

Recent work [39, 51] has shown the possibility to partition an XML stream arbitrarily with aggressive parallelization techniques, thus avoiding the pre-scan. However, due to some unique features of JSON, these techniques suffer from the *path explosion problem* (see Sections 2 and 4), failing to bring performance benefits for parallel JSON processing.

Overview of This Work. To solve these problems, this work proposes a compilation system, namely ***JPStream***, that is particularly designed for JSON-family data processing. For a given set of JSONPath queries, JPStream can automatically generate automata-based parallel executables with bounded memory footprints for JSON data processing. We next briefly describe the major components of JPStream.

- *Streaming Compilation.* The key to avoiding the expensive preprocessing is to adopt stream processing. However, this is non-trivial for JSON streams which are defined by Context-Free Grammars (CFG). The processing needs to perform both CFG parsing and query matching at the same time. JPStream achieves this with a *joint compilation*. It first compiles the queries (i.e., path expressions) and JSON syntax into two separate automata. Unlike conventional automata, these two automata are only partially defined. JPStream then "hooks" them together to form a new one, referred to as *streaming automaton*, which turns to be an unconventional pushdown automaton with double stacks. Encoded with both queries and JSON syntax, a streaming automaton performs parsing and querying simultaneously in a single pass (see Section 3).
- *Parallelizing Compilation.* Unlike the sequential automata that only start from the beginning of an input stream, the automata generated by JPStream can start from an arbitrary position. This flexibility is enabled by a set of parallelization techniques customized for JSON, including *state and stack element enumerations*, *query and syntax stack feasibility analyses*, and *automata resetting*. Together, they eliminate the *path explosion problem* encountered by the existing parallelization (see Section 4).
- *Runtime Optimization.* At last, JPStream also features a runtime that applies the *hints* (called data constraints) learned from the training input data to further reduce the costs of parallelization (see Section 5).

[1]Google route data is simplified and indented for better illustration.

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

Figure 2. BNF Grammar of JSON.

We prototyped JPStream in C language and used Pthreads for the generated code. Our evaluation using standard path expressions and a diverse set of JSON data has shown that JPStream-generated automata reduce the memory footprint by up to 95% comparing to preprocessing-based methods and scales with near-linear speedup on the commodity multicore and manycore processors (see Section 6).

Contributions. This work makes a four-fold contribution:

- To our best knowledge, this work introduces the first automata-based stream processing model for JSON data, with a novel joint compilation technique.
- It offers a set of parallelization techniques customized for JSON processing, making it possible to avoid the path explosion problem.
- It designs a data constraints learning scheme that can improve the parallelization efficiency at runtime.
- Finally, this work prototypes the proposed ideas and evaluates the system comprehensively.

Next, we provide the necessary background for this work.

2 Background

This section introduces the basic concepts of JSON and its processing, with a focus on its unique features comparing to XML, and how these features complicates the parallelization.

Grammar-based JSON. Originated from JavaScript, JSON has evolved into a language-independent data representation with a formal context-free grammar (CFG) (see Figure 2). The grammar follows the conventions of C-family languages, making it an ideal data-exchange language cross platforms.

Note that JSON grammar is fundamentally different from that of XML – a markup language based on tag pairs. Unlike JSON with a rich syntax, XML is almost "grammar-less" – as long as the tags are well paired, the data is considered to be valid [2]. As shown later, this "minimal grammar" feature makes the design of XML stream processing model easier. By contrast, CFG-based JSON requires a stream processing model that can cope with CFG, plus the querying. In fact, this needs a non-conventional automaton (an automaton with double stacks) to process. Next, we introduce the two major JSON structures: `object` and `array` and explain how they could complicate the design of data-level parallelization.

[2]The XML grammar discussed here should not be confused with the user-defined grammar for a specific set of XML data, such as DTD or XSD.

JSON Objects. A JSON `object` consists of zero or multiple `key:value` pairs, enclosed by a pair of curly parentheses (see Figure 2). For example, `{"lat":35, "lng":-106}` is a `loc` object with two key-value pairs (i.e., the attributes). This object-oriented design allows JSON to naturally fit with the data types in most object-oriented languages.

Comparing to XML tags, JSON objects are more efficient for encoding information. For example, XML needs a longer string to encode the same information in the prior example:

```
<loc><lat>35</lat><lng>-106</lng></loc>
```

Despite its conciseness, the syntax of JSON objects makes an effective parallelization more challenging. Intuitively, when an XML stream is partitioned, a pair of tags might be broken into different chunks. However, an end tag, like `</loc>`, still carries some context information – it was inside a `loc`. By contrast, a JSON object ends *anonymously*, with simply a right curly bracket }. As shown later in Section 4, lacking the information of object names is a key reason causing *path explosion*, a problem that can result in significantly high parallelization overhead.

JSON Arrays. The other major JSON component is `array`. A JSON `array` consists of an ordered list of `values`, embraced by a pair of square brackets (see Figure 2). For example, `steps` in Figure 1 is an array of `loc` objects.

Note that arrays are a new syntactical feature comparing to tag-based XML. Moreover, JSON allows the `elements` in an `array` to be `objects` or even `arrays`, creating nested structures that are potentially interleaved. On one hand, these new features make JSON more expressive than XML; On the other hand, similar to the *anonymous ending* feature, they complicate the parallelization. For example, in a broken piece of JSON data, it might be hard to tell if an object is under an array or an object, both of which can encapsulate objects. This syntactical uncertainty is another important reason for the path explosion problem (see Section 4).

In sum, JSON's unique features make it a promising data language that gradually replaces tag-based XML in many domains [18, 58, 60]. Meanwhile, they also complicate the design of stream processing and the parallelization.

Querying JSON Data. In the processing of JSON data, the most basic queries are *path expressions* (a.k.a. JSONPath), which identify the substructures of interest [3]. JSONPath is the key building block of more advanced processing schemes (e.g., JSONiq [11]). As its name suggests, a path expression defines specific "paths" from the root element of the JSON data (an anonymous `object` or `array`) to the substructures of interest. For example, `$.routes[1].steps` specifies the `steps` object under the second element of `routes` array. The `$` denotes the anonymous root element. Note that, unlike the path expressions for XML data, array indexes must be used in JSON path expressions for accessing array elements. More details about JSONPath syntax can be found in [37].

[3]The counterpart for XML is XPath, which has the same expressiveness.

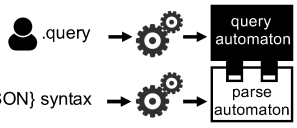

Figure 3. Illustration of Joint Compilation.

Next, we explain how JPStream uses path expressions and the JSON syntax for the purpose of stream processing.

3 Streaming Compilation

This section introduces the *joint compilation* technique that generates stream processing code for JSON.

3.1 Idea of Joint Compilation

The basic idea of joint compilation is illustrated in Figure 3. First, the queries and JSON syntax are compiled separately into two automata: (i) a *query automaton* for matching query results and (ii) a *parsing automaton* for recognizing the JSON syntactical structure. Note that, different from conventional automata, the query and parsing automata are only *partially defined*, with some transition rules missing. For example, the transition rules for handling an ending curly bracket `}` or an ending square bracket `]` are undefined in the query automaton. On the other hand, there are no concrete states defined in the parsing automaton.

The partial definitions of both automata are designed on purpose to facilitate the next compilation step – *automata hooking*. Basically, the two partially defined automata are "hooked" together in a way that a deterministic and complete set of transition rules are created. The resulted automaton is referred to as *streaming automaton*, which is capable of performing parsing and querying simultaneously. Next, we elaborate the generations of query and parsing automata, and the automata hooking, respectively.

3.2 Path Expression Compilation

Given a set of path expressions, JPStream compiles them into a single finite automaton with transitions partially defined.

The basic idea is inspired by a seminal work that designs deterministic automata for XML querying [32]. We treat a path expression as a regular expression, where a key in the path expression is considered as a symbol in the regular expression. For path expressions with predicates or logical operators, we break them down into subqueries, then group them into a set of regular expressions (also used in [51]). For example, a query `$.a[?(@.b||@.c)].d` is broken into four subqueries `$.a`, `$.a.b`, `$.a.c` and `$.a.d`, then grouped into a regular expression set `{a,ab,ac,ad}`. Following standard algorithms [19], a deterministic finite automaton (DFA) is generated for each regular expression. Then, the DFAs are connected to form a single DFA, which can recognize the matches for all the regular expressions. Formally, the resulted finite automaton M_{query} is a 5-tuple:

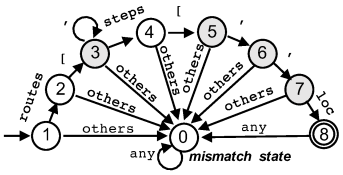

Figure 4. Automaton for `$.rountes[*].steps[2].loc`.

$$M_{query} = (Q, \Sigma, \delta, s_0, F) \quad (1)$$

where Q is the set of states, Σ is the input alphabet, δ is the set of transition rules, s_0 is the initial state, and F is the set of accept states. When the finite automaton transitions into an accept state, a match for a subquery is found.

Two things worth to mention here:

- The transition rule set δ is partially defined. In fact, it only handles three kinds of input symbols: a `key`, an open square bracket `[`, and a comma `,`, despite the existence of other symbols. The last two symbols are used for handling array constraints, as we will explain shortly.
- The DFA matches the results of the subqueries, which may still need to be merged and filtered to produce the final output. In the earlier example, `a` is finally outputted only if a match of `b` or `c` is found under `a`. Thus, there is another pass over the matches of subqueries.

Array Constraints Handling. Since JSON supports array, its path expressions must include array index constraints for accessing array elements, such as `steps[2]`, `steps[2:4]`, or `steps[*]`. To cope with these constraints, one option is to encode them into the automaton generation process. A trick for this is to reformat an array constraint with regular expressions of comma `,`, which separates array elements:

- A specific index range `[m:n]` [4] is reformatted to regular expression `[,{m,n}`, where `{m,n}` indicates the comma needs to be repeated `m` to `n` times.
- Similarly, the whole index range `[*]` is reformatted to regular expression `[,*`, where `*` indicates comma can be repeated any number of times.

Figure 4 shows the automaton generated for JSONPath `$.routes[*].steps[2].loc`. The states with shadows are the states used for handling array indexes.

In practice, we can also augment the automaton with a set of counters, each for an array index constraint appeared in the path expressions. This solution adds extra condition checks to the execution, but reduce the number of states.

3.3 JSON Syntax Compilation

To match the query results, it is required to recognize the syntactical structures of JSON streams. As JSON is CFL, the recognition requires a pushdown automaton.

[4] Negative indexes like `[-5:]` are not supported (requires backtracking).

[Obj-S]	$\Delta(\{, *) \rightarrow \{$	[Ary-S]	$\Delta([, *) \rightarrow [$
[Key]	$\Delta(\texttt{key}, *) \rightarrow \texttt{key}$	[Obj-E]	$\Delta(\}, \varepsilon : \{) \rightarrow \varepsilon$
[Val-Obj-E]	$\Delta(\}, \texttt{key}:\{) \rightarrow \varepsilon$	[Elt-Obj-E]	$\Delta(\}, [:\{) \rightarrow [$
[Ary-E]	$\Delta(], \varepsilon : [) \rightarrow \varepsilon$	[Val-Ary-E]	$\Delta(], \texttt{key}:[) \rightarrow \varepsilon$
[Elt-Ary-E]	$\Delta(], [:[) \rightarrow [$	[Key-Val]	$\Delta(,, \{) \rightarrow \{$
[Elt-Pmt]	$\Delta(\texttt{primitive}, [) \rightarrow [$	[Elt]	$\Delta(,, [) \rightarrow [$
		[Val-Pmt]	$\Delta(\texttt{primitive}, \texttt{key}) \rightarrow \varepsilon$

Figure 5. Transition Rules of Parsing Automaton.

Stateless Parsing. Unlike the conventional CFG parsing that tracks a parsing state to build the parse tree, our pushdown automaton only needs to recognize the syntactical structures (e.g., `object`, `array`, and `key-value`), without connecting them into a tree. This critical difference makes a *stateless automaton* sufficient for our purpose. Later, we will see this "statelessness" is also important for the automata hooking.

A stateless pushdown automaton $M_{parsing}$ is a 3-tuple:

$$M_{parsing} = (\Sigma, \Gamma, \Delta) \tag{2}$$

where $\Sigma = \{\{, \}, [\ ], ,, \texttt{prim}$, and $\texttt{key}\}$ is the input alphabet, $\Gamma = \{\{, [$, and $\texttt{key}\}$ is the stack alphabet, and Δ is the transition rule set (listed in Figure 5). Each rule follows a format of $\Delta(c, s) \rightarrow s'$, where c, $c \in \Sigma$ is the current input symbol, and s is the stack content, which contains an ordered list of symbols from the stack alphabet Γ. Symbols in s are separated by comma `:`. Note that, for conciseness, the rules in Figure 5 show at most two top elements in the stack. The arrow $\rightarrow$ indicates a transition that changes the stack by performing `pop` or `push` operations. In general, the transitions rules are not difficult to follow. One detail worth to mention is that the colon `:` in a key-value pair is treated as part of a `key` token. For example, `"steps":` is recognized as a `key`.

So far, we have introduced the generations of both query and parsing automata. Next, we explain how to hook them.

3.4 Automata Hooking

The basic idea is to use the parsing automaton to drive the execution of query automaton, so that they can coordinate to accomplish the parsing and querying tasks.

Transitions Rule Embedding. When parsing automaton recognizes some JSON syntactical structures relevant to the queries (i.e., `key`, `[`, and `,`), it will feed them to the query automaton to trigger state transitions. For this purpose, we embed the transition rules of query automaton δ into three transition rules of the parsing automaton: `[Ary-S]`, `[Key]`, and `[Elt]`, which handle the alphabet of query automaton.

Query State Recording. Note that the progress of query matching may be lost when the processing moves to a lower JSON structural level. For example, after a `steps[]` array is processed, the automaton (see Figure 4) needs to return to the state before it encounters the `steps[]`. However, the query automaton itself is unable to memorize these older states. We solve this by introducing another stack – *query stack*. When the processing moves to a lower level (e.g., reading a `[`), the current query automaton state is pushed onto the query stack. Correspondingly, a state is popped out of the query stack and to serve as the current query state when the processing returns to a higher level of JSON structure.

[Obj-S]	$\Delta(q, \{, *, *) \rightarrow (q, *, \{)$
[Ary-S]	$\Delta(q, [, *, *) \rightarrow (\delta(q, [), q, [)$
[Key]	$\Delta(q, \texttt{key}, *, *) \rightarrow (\delta(q, \texttt{key}), q, \texttt{key})$
[Obj-E]	$\Delta(q, \}, \varepsilon, \varepsilon : \{) \rightarrow (q, \varepsilon, \varepsilon)$
[Val-Obj-E]	$\Delta(q, \}, q', \texttt{key}:\{) \rightarrow (q', \varepsilon, \varepsilon)$
[Elt-Obj-E]	$\Delta(q, \}, *, [:\{) \rightarrow (q, *, [)$
[Ary-E]	$\Delta(q,], q', \varepsilon : [) \rightarrow (q', \varepsilon, \varepsilon)$
[Val-Ary-E]	$\Delta(q,], q' : *, \texttt{key}:[) \rightarrow (q', \varepsilon, \varepsilon)$
[Elt-Ary-E]	$\Delta(q,], q', [:[) \rightarrow (q', \varepsilon, [)$
[Val-Pmt]	$\Delta(q, \texttt{primitive}, q', \texttt{key}) \rightarrow (q', \varepsilon, \varepsilon)$
[Elt-Pmt]	$\Delta(q, \texttt{primitive}, *, [) \rightarrow (q, *, [)$
[Key-Val]	$\Delta(q, ,, *, \{) \rightarrow (q, *, \{)$
[Elt]	$\Delta(q, ,, *, [) \rightarrow (\delta(q, ,), *, [)$

Figure 6. Transition Rules of Streaming Automaton.

In this way, the parsing automaton coordinates with the query automaton, executing like a single automaton. Here, we refer to the hooked automaton as *streaming automaton*. Formally, a streaming automaton is an 8-tuple:

$$M_{streaming} = (\Sigma, \Gamma_q, \Gamma_s, \Delta, \delta, Q, s_0, F) \tag{3}$$

where the input alphabet Σ and syntax stack alphabet Γ_s are from the parsing automaton, the query stack alphabet Γ_q, along with Q, s_0, and F are from the query automaton, and Δ is the transition rule set, embedded with the ones δ from the query automaton. Figure 6 lists the rules in the format:

$$\Delta(s, c, qs, ss) \rightarrow (s', qs', ss') \tag{4}$$

saying when the automaton is in state s, after reading the symbol c, it will transition to state s', meanwhile, the query and syntax stacks qs and ss will be changed to qs' and ss' with pop/push operations, respectively.

Example. Figure 7 shows a streaming automaton execution. The input is from Figure 1 and the query automaton is given in Figure 4. For space limit, some transitions are omitted.

Initially, the automaton is in State 1 with both stacks empty. After reading the first symbol `{`, rule `[Obj-S]` pushes `{` onto the syntax stack without changing the state. For the next symbol `"routes":`, rule `[Key]` pushes `K` and State 1 onto the query and syntax stacks, respectively, then changes the state to 2 (see Figure 4 for state transitions). Next, after reading symbol `[`, rule `[Ary-S]` pushes the symbol and the current state to the stacks again, then updates the state to 3. By following the transition rules, this automaton processes the remaining input symbols in a similar way. Once it reaches state 8, an accept state, a match to the query is found. Finally, after finishing the last symbol `}`, the automaton halts with the query and syntax stacks both empty again.

However, the execution of a streaming automaton remains sequential. Next, we explain the ways to parallelize it.

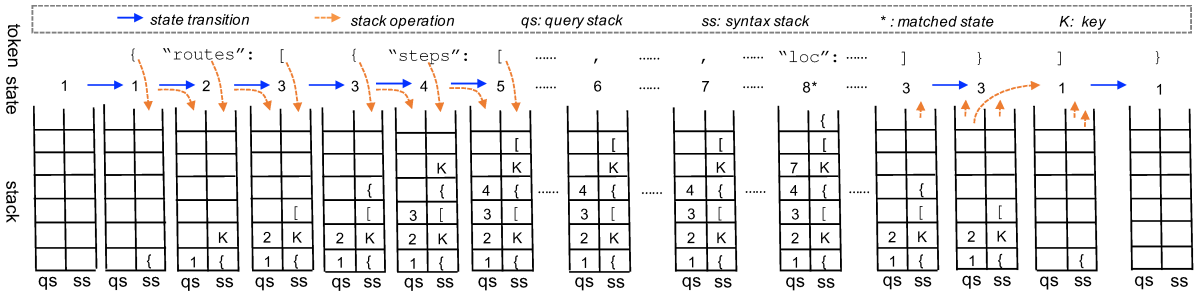

Figure 7. An Example of Streaming Automaton Execution (input and query automaton are from Figures 1 and 4).

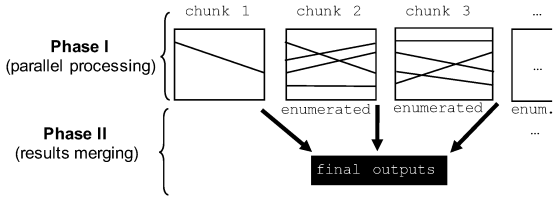

Figure 8. Two-Phase Parallelization.

4 Parallelizing Compilation

At a high level, the parallelization of a streaming automaton execution follows two phases: (i) parallel processing and (ii) results merging, as illustrated in Figure 8.

First, the data stream is partitioned into chunks (almost) evenly [5], with each chunk processed by a thread (running a streaming automaton). After all threads finish, the results are merged to form the final outputs. The main challenge comes from the data dependences across the partitioned chunks where some JSON structures (e.g., a JSON object or array) might be broken into different chunks. In the following, we will discuss the techniques for breaking the dependences, analyze their costs, and more importantly, reduce the costs.

4.1 Breaking Dependences

As Figure 7 shows, the execution of a streaming automaton involves a series of transitions, each of which depends on the prior state and stacks. This can be symbolically reflected by the transition rule structure in Equation 4. Formally, we define *state dependence* and *stack dependence* as follows:

Definition 1. *In a streaming automaton execution, suppose T_i and T_j $(i < j)$ are two arbitrary transitions. Based on the transition rule structure in Equation 4, T_i writes to state s before T_j reads from it, thus indicating a true dependence (i.e., read-after-write) between the two transitions. We refer to this dependence as* ***state dependence****. Similarly, there also exist true dependences on the two stacks qs and ss between the two transitions, which we refer to as* ***query stack dependence*** *and* ***syntax stack dependence****, respectively.*

[5]The lexer is adjusted to avoid breaking a token.

These three kinds of dependences serialize the execution of a streaming automaton from the beginning to the end. Next, we show how these dependences can be broken by leveraging some "enumerability" properties of automata.

- ***Breaking State Dependences.*** Based on the definitions, the number of states in a streaming automaton is finite. Thus, we can enumerate all the states when the state is unknown. For example, by enumerating all the 9 states in Figure 4, we ensure the correct state is always covered.
- ***Breaking Stack Dependences.*** Unlike state dependences, it is much harder to break stack dependences as the stacks can grow arbitrarily deep. Fortunately, we *do not have to know all the contents of the stacks at once.* According to the transition rules (see Figure 6), at most the top two elements of the stacks are accessed when the automaton reads a symbol. This property allows us to *gradually* enumerate the stack contents *on-demand.* When a transition needs to access an empty stack (due to the partitioning), we enumerate symbols in the stack alphabet (i.e., Γ_q or Γ_s).

For each enumerated case, we fork a new execution path. That is, an enumeration causes an execution path to *diverge* into multiple. If path divergence occurs frequently, the path maintaining overhead may outweigh parallelization benefits. We next show how fast the number of paths can grow.

Path Complexity Analysis. Suppose thread t_i is assigned to process chunk i and the number of paths is denoted as P.

At the beginning of chunk i, without knowing the state, thread t_i enumerates all the states. Thus the number of paths becomes $P = |Q|$, where Q is the state set. Then, thread t_i reads the first symbol c. Based on the transition rules (see Figure 6), if c is `{`, `[`, or `key` (i.e., the first three rules), then all the execution paths would proceed as normal. If c is `}`, then there are three potential rules (i.e., the 4th-6th rules) applicable. Which one is the actual depends on the stacks. Unfortunately, the stacks are all empty at this moment. Thus, thread t_i needs to enumerate all the three cases, with the assumptions that corresponding elements are in the stacks. Furthermore, the 5th rule `[Val-Obj-E]` needs to reset the current state with the top element of the query stack. Since the query stack is empty, thread t_i has to enumerate the

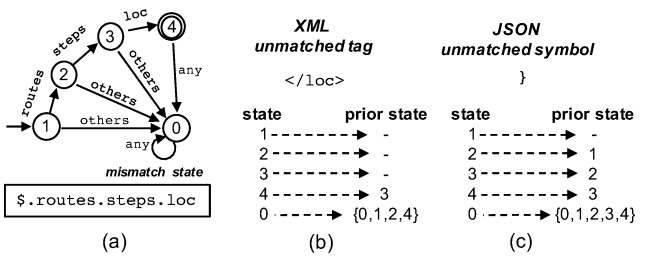

Figure 9. Query Stack Feasibility Inference: XML vs. JSON

query stack alphabet Γ_q (i.e., $|\Gamma_q|$ paths). Putting it together, when c is }, each of the existing paths diverges into $|\Gamma_q| + 2$ paths. Thus, we have $P = |Q| \cdot (|\Gamma_q| + 2)$. Similarly, we can analyze the numbers of paths when c equals other input symbols. In fact, the worst case happens to the symbol], where the number of paths $P = |Q| \cdot 3|\Gamma_q|$, because the query stack is accessed in all the three potential rules. If this worst situation keeps happening in the following processing of this chunk, the number of paths would become $P = |Q| \cdot (3|\Gamma_q|)^k$, where k is the number of unmatched]. As $\Gamma_q = Q$, we have the following conclusion:

Theorem 1. *The worst-case path complexity in a parallel streaming automaton execution is:*

$$O(3^k \cdot |Q| \cdot |\Gamma_q|^k) = O(3^k \cdot |Q|^{k+1}) \tag{5}$$

where Q and Γ_q are the input and query stack alphabets, and k is the number of unmatched].

There are two factors in Equation 5: $O(3^k)$ and $O(|Q|^{k+1})$. Both are caused by data partitioning. The first factor is due to the syntactical uncertainty (needed elements missing in the syntax stack) and the second factor is due to the state uncertainty (needed elements missing in the query stack). Hereinafter, we refer to the two uncertainties as *syntactical complexity* and *state complexity*, respectively.

Both complexities can cause the number of execution paths to increase exponentially, which may cause the path maintenance costs to surpass the parallelization benefits and even run out of the main memory for a relatively small piece of chunk. We refer to this issue as *path explosion problem.*

4.2 Feasible Paths Inference

In this section, we discuss a couple of solutions that could help alleviate the path explosion problem by addressing the state and syntactical complexities, respectively.

Query Stack Feasibility Inference. Recent work [39, 51] has shown the possibility to leverage the state transitions of query automaton to infer the (top) elements in the query stack, for parallelizing XML's automaton execution.

The basic idea of the inference is illustrated in Figures 9-(a) and (b) using a simple query. When an unmatched end tag is met (e.g., </loc>), we can infer the top element in the query stack for each enumerated state. In fact, the top element is the state before meeting the open tag <loc>. Based on the structure of the query automaton, we can find all the possible states that transition to an enumerated state after reading <loc>. For example, only State 3 transitions to State 4 after reading <loc>. After the inference, we can find there are five execution paths left – the same as the number of states. In fact, it is provable that the number of paths is bounded by a constant (i.e., |Q|) with the query stack feasibility inference.

However, this approach fails to work effectively for the parallelization of JSON data processing for two reasons:

- ***Syntactical Complexity.*** First, in the case of JSON, there are two complexities causing exponential path growth: state complexity and syntactical complexity. Query stack feasibility inference might help address the first, but not the second, which is not directly relevant to the states. Thus, the path explosion problem remains unsolved.
- ***Anonymous Ending Symbol.*** Even for state complexity, the above inference cannot reduce the complexity to the linear level as in the case of XML. Because, unlike XML where an end tag (e.g., </loc>) carries the tag name, a JSON ending symbol (e.g., }) is anonymous, as shown in Figure 9-(c). Without the key name, there would be more states possibly serving as the prior state. For example, State 2 could transition to State 3 and State 1 could transition to State 2 (see Figure 9-(c)). Despite they correspond to different input symbols, they all have to be considered to ensure the correctness. Consequently, the number of execution paths increases from 5 to 8. This increase could be accumulated as more unmatched symbols are met.

In sum, due to the special features of JSON, query stack feasibility inference itself is insufficient for reducing the costs of parallelization. Next, we present a JSON-customized feasibility inference for the syntactical complexity.

Syntax Stack Feasibility Inference. With the partitioning, the syntax stack may lack needed element(s) to determine the next transition rule after reading a symbol. To address this syntactical uncertainty, our idea is *to infer the syntax stack by looking ahead more input symbols.* In fact, we have:

Theorem 2. *By looking forward at most two input symbols, the transition rule of a streaming automaton can be uniquely determined without the knowledge of the syntax stack.*

Theorem 2 can be proved with a brute-force enumeration. For each symbol that may cause syntactical uncertainties (4 such symbols), we enumerate all the possible combinations of its following two symbols, including the empty symbol ε ($7 * 7 + 7 + 1 = 57$ cases). Thus, there are $4 * 57 = 228$ cases overall. For each case, we run the streaming automaton on the three symbols. If the automaton halts before finishing, the case is infeasible; Otherwise, we record the correspondence between the input symbols and the first transition rule to the table. By exhaustively searching the correspondence space, we ensure the coverage of all the feasible situations.

Table 1. Syntax Stack Feasibility Inference (* means any symbol).

Input	Following Two Symbols	Syntax Stack	Rule
}	}* or ,key	*:key:{	[Val-Obj-E]
}	,[or ,{ or]* or ,prim	*:[:{	[Elt-Obj-E]
}	ε or {*	ε:{	[Obj-E]
]	}* or ,key	*:key:[	[Val-Ary-E]
]	,[or ,{ or]* or ,prim	*:[:[	[Elt-Ary-E]
]	ε or [*	ε:[	[Ary-E]
prim	}* or ,key	*:key	[Val-Pmt]
prim	,[or ,{ or]* or ,prim	*:[	[Elt-Pmt]
,	key*	*:{	[Key-Val]
,	[* or {* or prim*	*:[	[Elt]

Table 1 shows our findings. The first two columns list the input symbols that may cause syntactical uncertainty and the potential following two symbols, respectively. The last two columns show the contents that should be in the syntax stack (instead of being empty) and the corresponding transition rule that should be applied. With this technique, we reduce the syntactical complexity from $O(3^k)$ to $O(1)$.

4.3 Towards a Bounded Number of Paths

Though the syntactical complexity has been addressed in Section 4.2, the number of paths may still grow exponentially due to the state complexity, for which, the existing query stack feasibility inference fails due to the anonymous ending symbol property of JSON (Section 4.2). In this section, we discuss a new strategy to bound the number of paths.

On-demand Automata Resetting. The basic strategy is to reset a parallel streaming automaton once it fails to uniquely determine the next possible transition.

Recall the reason that an enumerated execution path may diverge is the lack of needed elements from the query and/or syntax stack(s) when reading certain input symbol. Here, we refer to such a symbol as an *unmatched symbol*. When an unmatched symbol is met, we can apply the query and syntax stack feasibility inferences from Section 4.2. If they fail to uniquely determine the next transition, we will first skip the symbol, then reset the streaming automaton, which includes re-enumerating all the states as the current and emptying the stacks, just like processing a fresh new chunk. This strategy may leave some unmatched symbols unprocessed, for which the processing is postponed to the results merging phase. We refer to this scheme as *on-demand automata resetting*. As an execution path never diverge, on-demand automata resetting bounds the number of execution paths by the number of states $|Q|$, that is, $P = O(|Q|)$.

Data Units. Essentially, automata resetting further breaks the input chunks into even smaller pieces. To distinguish them from the original partitioning, we refer to these smaller pieces within a chunk as *data units*.

During the merging phase, the results of data units will be merged, with the unmatched symbols in-between processed. This may add extra costs to the merging phase, depending on the number of data units. According to our evaluation, the number of data units in a chunk is quite manageable, resulting in only marginal merging costs.

4.4 Results Merging

First, during the parallel processing, for each data unit i, there is a mapping maintained:

$$\mathcal{M}_i(M_j^s) = M_j^e \tag{6}$$

where M_j^s is a possible starting configuration of unit i's streaming automaton and M_j^e is the corresponding ending configuration. The temporarily matched results are recorded separately for each starting configuration. After the parallel processing, the correct starting configuration of each unit can be identified one by one. For example, with the (always correct) ending configuration of the first data unit, a (serial) streaming automaton can further process the unmatched symbols between the first and second data units (if any). After that, the configuration of the automaton would be the correct starting configuration for the second data unit. In this way, we merge all the data units (along with unmatched symbols). During the units merging, the matched results on the correct path are separated as the actual matches of subqueries. Finally, the matches of subqueries are merged and filtered to produce the final outputs (see Section 3.2).

Input Errors. Sometimes, a JSON data stream may contain syntactical errors, which may alter the transitions of parallel streaming automata. However, since the merging of results is performed sequentially, we can still ensure the detections of all syntactical errors during the merging phase.

5 Runtime Optimization

The prior section leverages the transition rules and local input symbols to reduce the number of enumerated paths. In this section, we exploit the data properties to further prune paths [6]. The intuition is that the elements in JSON data streams often follow some patterns. For instance, in Figure 1, `steps` is always under `routes`, while `loc` could be under `routes` or `steps`. We refer to these casual relations among JSON objects and arrays as *data constraints*.

Knowing the data constraints may help further prune the paths. Consider the serial automaton execution in Figure 7 (w/ input in Figure 1). We can record the correspondence between the current state and the next input symbol. As shown in Table 2, there is only a subset of states actually happened to encounter a specific `key` or a `[`. For example, the second row says when the automaton meets `steps`, it is always at State 3. By feeding such information to the query stack feasibility inference, the enumerated paths could be further pruned. Note that, only `key` and `[` appear in Table 2, because, with automata resetting, the path enumeration can only happen before `key`, `[`, or `{`, so other symbols are not

[6] Similar properties are also exploited in XML data [39].

Table 2. Example Data Constraints Hash Table

Input Symbol	Feasible States (Paths)
`"routes":`	{1}
`"steps":`	{3}
`"loc":`	{5,6,7}
`"lat":`	{0}
`"lng":`	{0}
`[`	{2,4}

recorded. Symbol { is removed from the table as every state might meet it, failing to provide any useful constraints.

However, this observation-based optimization is not safe when the actual input does not follow the same "pattern" as the one used for collecting the data constraints. As a result, the correct path may be pruned. For example, suppose in the actual input, `lat` and `lng` appear under the `routes`, then using the feasible State 0 in Table 2 could be incorrect. Therefore, to ensure the correctness, we also need some correctness checking and reprocessing mechanisms.

Based on the above discussion, we provide JPStream an offline data constraint learner and an online component for constraint integration. The learner takes training inputs from the users to build symbol-state correspondences, which are then exported as a hash table. The integration part tracks the uses of query stack feasibility inference and automaton resetting events during a parallel automaton execution. Once observed, the integration component uses the current symbol to query the hash table to get the feasible states. Then, it takes an intersection between the returned states and the existing state set. Later, in the results merging phase, the integration component verifies if the correct state was covered. If not, it will reprocess the incorrectly interpreted unit, but never go beyond a unit, thanks to the automaton resetting.

6 Evaluation

This section evaluates the automata generated by JPStream.

6.1 Implementation

We prototyped JPStream in C language and used Pthread for the parallel implementation of streaming automata. JPStream supports the standard JSONPath queries [37].

Given a JSONPath query, JPStream first parses it into an abstract syntax tree (AST) that facilitates the generation of the query automaton. For less number of states, JPStream uses counters for handling array index constraints instead of state transitions (see Section 3.2). JPStream then hooks the query automaton to the pre-compiled parsing automaton to form a serial streaming automaton. For the parallelizing compilation, JPStream adopts a double-tree data structure to reduce the path maintenance costs [39, 51]. It implements the syntax stack feasibility inference (Section 4.2) and supports the on-demand automata resetting scheme (Section 4.3). In addition, JPStream also features an offline data constraint learner that takes additional training inputs to learn the data constraints, and a runtime integration module to apply the constraints (Section 5). To provide a bounded the memory footprint, JPStream uses a threshold to limit the maximum size of data loaded each time. When a data stream is larger than the threshold, JPStream performs multiple load-process cycles. The default value of this threshold is set to 250MB.

6.2 Methodology

We compare the automata generated by JPStream with the state-of-the-art methods for parallel semi-structured data processing and also a group of state-of-the-practice JSON tools, in terms of performance and memory usage.

Table 3. Methods in Evaluation.

Abbr.	Methods
XMLStream	An adoption from parallel XML processor [39, 51]
JsonSurfer	A manually crafted tool for streaming JsonPath [15]
JSON-C	An open-source JSON parser in C [10]
Jackson-JSON	An open-source JSON parser in Java [12]
RapidJSON	A JSON parser in C++ from Tencent [14]
FastJSON	A parsing-based JSON tool in Java from Alibaba [4]
JPStreamNR	Non-restartable JPStream automata
JPStreamR	Restartable JPStream automata
JPStreamR+	JPStreamR with data constraints learning

Methods. Table 3 lists the methods used in our evaluation. XMLStream [39, 51] is a parallelization solution recently developed for XML data processing. Note that this solution cannot be directly applied to JSON data due to the syntax differences (see Section 2). Here, we ported their ideas to the JSON data processing. JsonSurfer [15], as far as we know, is the only stream processing implementation for JSONPath querying. Note that it is not based on automata. As it is Java implemented, for a fair comparison with our C-based automata, we also rewrote JsonSurfer in C. For parsing-based JSON processing, many tools exist. We pick JSON-C [10], Jackson-JSON [12], RapidJSON [14], and FastJSON [4] for their popularities and industrial supports. For Jackson-JSON and FastJSON, since they are implemented in Java, we first warmed up the JVM before measuring their performance, for their best results. As to our methods, we evaluated three versions of JPStream: (i) the one with automata resetting disabled; (ii) the one with automata resetting; and (iii) the one with both automata resetting and data constraints learning.

Platforms. All experiments run on two servers. The first one is a 16-core machine equipped with two Intel 2.10 GHz Xeon EE5-2620 v4 processors and 64 GB of RAM. The second server is a 64-core machine equipped with Intel Xeon Phi 7210 processors and 96 GB of RAM. Both servers run on CentOS 7 and are installed with GCC 4.8.5. In the following, we refer to the two servers as *Xeon* and *Xeon-Phi*. All programs are compiled with `O3` optimization. The timing results reported are the average of 10 repetitive runs. All CPUs are warmed up before evaluation. We do not report 95% confidence interval of the average when the variation is not significant.

Table 4. JSONPath queries. #sub shows the number of sub-queries in each query structure.

Query	Data	Query structure	#sub	#matches
BB1	BB	$.pd.shl[1:5].sel[1:2]	1	95
BB3	BB	$..p..st	1	0
TT1	TT	$[*].ur..is	1	312,460
TT2	TT	$[*].qs.en.um.[*].nm	1	9,580
NSPL2	NSPL	$.dt[20:200:2][:]	1	3,913
UHD1	UHD	$.dt[10:1000][:]	1	19,820

Query	Data	Query structure	#sub	# matches
BB2	BB	$.pd[?(@.s||@.pid||@.nm||@.sr||@.tp||@.sd||@.nw)].cp[1:2].id	9	459,333
TT3	TT	$[*].qs.*.me[?(@.id&&@.ids&&@.idc&&@.mu&&@.mh&&@.ud&&@.url&&(@.eu||@.tp))].sz.lg.w	9	32,750
NSPL1	NSPL	$.mt.*.co[?(@.id||@.fg||@.dtn||@.cc.lg)]..nm	6	44
AIL1	AIL	$.fe[?(@.tp&&(@.id||@.gn)].pp.nm	5	21,546
AIL2	AIL	$.fe[1:7].fm.cd[:][:][10:20][:]	1	2,088
GMD1	GMD	$[-15:].rt[?@.cr].bd.*.lt	4	28
GMD2	GMD	$[*].rt[?(@.bd||@.cr].le[?@.sa].st[0:3].dt.vl	6	17,147
UHD2	UHD	$.mt.vi.co[?(@.id&&@.nm&&@.dtn)].cc.tp[1:120].it	6	169

Table 5. Dataset Statistics.

Data	#objects	#arrays	#key-value	#primitive	depth
BB	1,916,574	4,882,921	40,763,630	35,880,709	4
TT	2,898,910	4,197,990	31,472,660	30,910,730	7
NSPL	613	3,509,815	1,669	84,235,938	7
AIL	64,675	46,858,734	905,058	93,395,898	3
GMD	10,357,445	43,943	29,034,886	21,051,619	9
UHD	262	1,511,234	719	30,224,745	8

Datasets. Table 5 summarizes the statistics of JSON datasets used in our evaluation, including Best Buy product dataset [2], Twitter global tweet stream [16], National Statistics Post Code Lookup (NSPL) dataset for the United Kingdom [13], Indigenous Land Use Agreements dataset [9], Google Maps Directions dataset [6], and Washington State Department of Health dataset [7]. The training datasets for JPStreamR+ are extracted from the first 1% of the testing inputs.

Query Sets. Table 4 shows our evaluated queries. It covers all the basic types of JSONPath queries that are commonly used [37]. The rightmost two columns record the numbers of subqueries and matches in each query, respectively.

6.3 Comparison with Existing Methods

Note that, due to the inherent dependences, all the existing methods only execute sequentially, except for XMLStream. This set of experiments were executed on Xeon.

Comparison with Streaming Methods. Figure 10 reports the execution time of stream processing methods. First, we notice XMLStream runs out of memory for all the queries. Because it does not address the syntactical complexity (see Section 4.2), leading to exponential path growth that quickly consumes all the machine memory (64GB).

JsonSurfer successfully processed all the queries, but its performance is limited by the serial execution. Single-thread JPStreamR outperforms JsonSurfer, thanks to its use of a query automaton. By contrast, JsonSurfer uses the parsing stack to match queries, which may examine deeper in the stack for a complete comparison. This is especially inefficient for queries with .. wildcard, such as TT1 and NSPL1. As indicated by Figure 10, the gaps in these two cases are even larger. On top of that, when using all the 16 cores, JPStreamR outperforms JsonSurfer by an order of magnitude.

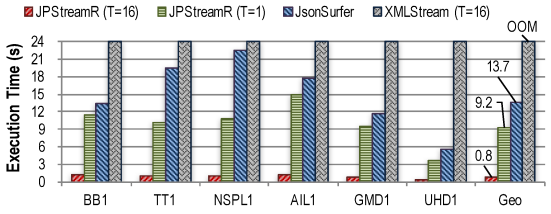

Figure 10. Performance of Streaming-based Methods*.
*The original JsonSurfer only supports a subset of queries. For the supported ones, its performance is 1-1.45X faster than JPStreamR (T=1), after the JVM warmup.

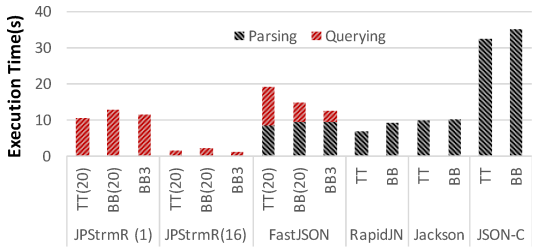

Figure 11. Comparison with Parsing-based Methods (Time).

Comparison with Parsing-based Methods. In Figure 11, JPStreamR is compared against four parsing-based methods. TT(20) and BB(20) are two query sets, each with 20 queries. For JSON tools without querying supports, we simply report their parsing time. Most parsers finish the parsing of our datasets within 10 secs, except for JSON-C (took over 30 secs). In comparison, single-threaded JPStreamR spent a little more than 10 secs, comparable to other JSON tools. On top of that, JPStreamR also finishes the querying at the same time, while FastJSON needs two separate phases. The total processing time of FastJSON surpasses JPStreamR in our evaluated cases. When all the 16 threads are used, JPStreamR outperforms all the evaluated parsing-based methods significantly.

Besides performance, JPStreamR also shows advantages in the memory footprint, as indicated in Figure 12. Thanks to its stream processing model, JPStreamR saves up to 95% of memory consumption. In fact, unlike the other JSON tools that consume more memory for larger inputs, the memory footprint of JPStreamR is bounded.

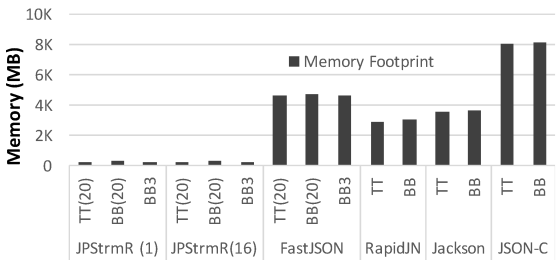

Figure 12. Comparison with Parsing-based (Memory).

6.4 Performance of Parallel Execution

Since the only other parallel method, XMLStream, runs out of memory in all the tested cases, we mainly focus on the three versions of JPStream-generated automata in this section.

Speedup. Figure 13 reports the speedups of parallel automata executions over the serial one. Among the three versions, JPStreamR+ yields the best speedup for all evaluated queries, thanks to its data constraint learning. On average, it achieves 12.1X on 16-core Xeon and 53.1X on 64-core Xeon Phi. For the other two versions, JPStreamR performs better in general, for its on-demand automata resetting scheme.

To get deeper insights, we also examined the number of execution paths, the costs of results merging, and the costs of using the data constraint integration.

Number of Execution Paths. The more paths a parallel streaming automaton needs to maintain, the less benefits it brings. Table 6 reports the statistics about the number of execution paths. The "avg." columns show the average number of paths to maintain by a streaming automaton. As the last row (i.e., "Geo") shows, JPStreamNR maintains 2.29 paths on average across all the tested cases. In comparison, JPStreamR maintains 1.88 paths and JPStreamR+ maintains only 1.15 paths. These statistics largely echo the performance differences among the three versions of JPStream. The "max" columns show the maximum number of paths to maintain. We can see JPStreamNR maintains up to 119 paths. For the same case, JPStreamR+ only maintain 3 paths, due to the path pruning enabled by data constraint integration.

Table 6. Number of Execution Paths (Xeon Phi)

	JPStreamNR		JPStreamR		JPStreamR+	
Query	avg.	max	avg.	max	avg.	max
BB1	6.02	16.0	3.70	8.0	1.01	2.0
BB2	11.46	37.0	6.45	15.0	1.01	3.0
TT1	4.45	119.0	2.53	7.0	1.41	3.0
TT2	3.76	42.0	1.89	7.0	1.16	4.0
NSPL1	1.15	6.0	1.15	6.0	1.11	4.0
NSPL2	1.35	15.0	1.35	15.0	1.07	3.0
AIL1	1.75	25.0	1.74	10.0	1.19	3.0
AIL2	2.26	28.0	2.25	11.0	1.65	5.0
GMD1	1.07	23.0	1.06	10.0	1.01	2.0
GMD2	1.27	44.0	1.24	15.0	1.01	2.0
UHD1	1.26	6.0	1.26	6.0	1.13	3.0
UHD2	1.70	15.0	1.70	15.0	1.17	3.0
Geo	2.29	22.5	1.88	9.8	1.15	3.0

Table 7. Costs of Results Merging

Dataset		#Chunks	#Units	Chunk	Chunk+Unit
Xeon-Phi	BB	64	146	0.004%	0.004%
	TT	64	176	0.005%	0.006%
	NSPL	64	116	0.001%	0.001%
	AIL	64	292	0.003%	0.004%
	GMD	64	317	0.001%	0.002%
	UHD	64	64	0.002%	0.002%
	Geo	64	185	0.002%	0.003%

Table 8. Path Coverage Accuracy and Reprocessing Cost

Query		Acc.	Cost	Query		Acc.	Cost
Xeon	BB1	97.62%	0.13%	Xeon-Phi	BB1	95.21%	0.52%
	BB2	97.62%	0.08%		BB2	97.26%	0.26%
	TT1	92.86%	0.14%		TT1	96.59%	0.55%
	TT2	95.24%	0.13%		TT2	97.73%	0.56%
	TT3	92.86%	0.13%		TT3	95.45%	0.79%

Results Merging Costs. Table 7 reports the merging costs. First, the costs are consistently less than 0.001% in all the tested cases. Second, units merging does not add extra costs significantly comparing to the chunk merging, thanks to the relatively small number of units (3rd column).

Costs of Online Constraints Integration. In principle, the online data constraints integration is a double-edged sword, as it may eliminate a correct path due to the discrepancy between the training and testing inputs. For this reason, we examine the accuracy of covering the correct execution path and the cost of reprocessing if the correct one is missed. The results are reported in Table 8. The queries that are not shown have 100% accuracies. The results indicate, by using the first 1% of the testing inputs as the training ones, for most test cases, JPStreamR+ successfully covered the correct paths. For the ones that it misses, the accuracy is beyond 90%, with reprocessing costs consistently less than 1%.

Table 9. Scalability over Data Size on Xeon

Data Size	Exe. Time (1 core)	Exe. Time (16 cores)	Speedup
250 MB	3.022 s	0.245 s	12.335
1 GB	11.371 s	0.900 s	12.636
4 GB	46.573 s	3.619 s	12.868
16 GB	192.829 s	14.829 s	13.003

6.5 Scalability

Scalability over Number of Cores. Figure 14 presents the speedup curves of JPStream on Xeon-Phi. In general, all three versions exhibit linear speedup increase up to 64 cores, with different increasing rates. For JPStreamNR, the curve is less smooth, because the number of execution paths could vary significantly under different partitions. For the other two, the automaton resetting largely limits the maximal number of execution paths, yielding smoother curves.

Scalability over Input Sizes. Table 9 reports the speedup of JPStreamR+ with different data-size thresholds on Xeon. The results indicate that, in general, there is a tradeoff between the memory footprint and the performance gain.

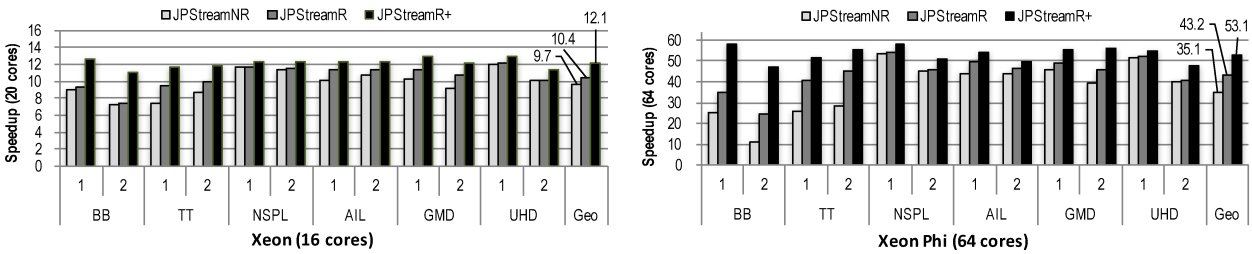

Figure 13. Speedups of JPStream-generated Automata on Xeon and Xeon Phi Servers.

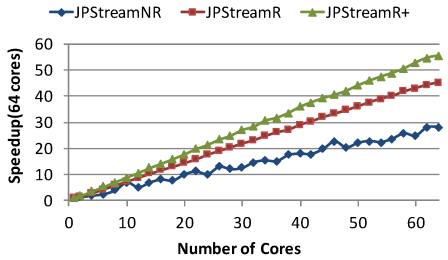

Figure 14. Scalability over Number of Cores on Xeon-Phi

7 Related Work

Semi-structured Data Processing. A large body of prior work focuses on the expressiveness of semi-structured data querying and the efficiency of executing concurrent queries, especially for XML data. For evaluating XML path queries, representative methods include automata-based [32, 63], array-based [40], and stack-based algorithms [26]. Among them, a seminal idea by Green and others is to combine a set of XPath expressions into a DFA to improve the evaluation efficiency [32], which influences many others, including two widely used XML tools, Y-filter [27], XMLTK [22], as well as JPStream, though ours is for JSON data processing.

Unlike XML with well-developed streaming computation models, most JSON tools [4, 10, 12, 14] rely on parsing the whole data stream to extract information. JsonSurfer [15] is the only stream processing tool for JSON that we are aware of, but it runs less efficiently for the lack of automata. Barenghi and others [23] use operator precedence grammars (OPG) for parallel JSON parsing, without querying support. Unlike others, MISON [42] exploits bitwise parallelism with SIMD supports. It pre-constructs structural indexes for JSON data to speed up the querying. But the index construction can only execute sequentially due to dependences, thus cannot scale on multicore machines. Parabix [44] also uses bitwise parallelism to accelerate XML parsing, but not querying. Palkar and others apply filters [54] on the raw JSON byte stream before querying the data to accelerate the processing. Pavlopoulou and others [55] rewrite JSONiq queries into XQuery to leverage an existing XML framework (Apache VXQuery [1]) to support parallel processing of multiple JSON data streams, rather than a single stream.

Parallel Finite Automata. Finite automata (a.k.a. FSMs) are difficult to parallelize. A traditional parallelization is to use prefix-sum [41], which is also based state enumeration. An implementation of this method [50] has been optimized for SIMD instructions. Speculative parallelization of FSMs has also been exploited [56, 57, 64, 65] by breaking the transition dependences with state predictions. Though providing useful insights, the above work cannot be directly applied to the parallelization of semi-structured data processing, which essentially requires the use of stack-based automata.

Some recent work introduces parallel pushdown automata for processing XML data, such as PP-Transducer [51] and GAP [39]. They share some high-level ideas with JPStream. However, they suffer from the path explosion problem when ported to JSON. AT-GIS [52] is a recent work for parallel processing GeoJSON, a derivative of JSON. However, its applicability to the generic JSON remains unclear.

8 Conclusion

This work addresses the challenges in scalable JSON-family data processing with a compilation system, called JPStream. JPStream automatically generates parallel executables with low-memory footprints for JSON data processing. At its core is the design of a streaming computation model and a set of customized parallelization techniques. The model is enabled by a joint compilation idea that combines the queries and JSON syntax into a unified automaton. The parallelization leverages a syntactical feasibility inference, an on-demand automaton resetting scheme, and a data constraint learner to address the path explosion raised by the unique features of JSON. Finally, evaluation confirms the effectiveness of JPStream in generating highly scalable automata executables with bounded memory footprints, showing superiority over the state-of-the-art methods and existing JSON solutions in terms of performance and memory consumption.

Acknowledgments

The authors would like to thank the anonymous reviewers for their time and comments. This material is based upon the work supported in part by National Science Foundation (NSF) Grants No. 1565928 and 1751392, and Hellman Foundation.

References

[1] Apache VXQuery. https://vxquery.apache.org/.

[2] Best Buy developer API. https://bestbuyapis.github.io/api-documentation/. Retrieved: 2018-07-01.

[3] Best practices for reading JSON data. https://docs.aws.amazon.com/athena/latest/ug/. Retrieved: 2018-07-01.

[4] A fast JSON parser/generator for Java. https://github.com/alibaba/fastjson/. Retrieved: 2018-07-01.

[5] Getting started with JSON features in Azure SQL database. https://docs.microsoft.com/en-us/azure/sql-database/sql-database-json-features. Retrieved: 2018-07-01.

[6] Google Map directions API. https://developers.google.com/maps/documentation/directions/start/. Retrieved: 2018-07-01.

[7] Health care provider credential data. https://data.wa.gov/api/views/qxh8-f4bd/. Retrieved: 2018-07-01.

[8] The home of the U.S. government's open data. https://www.data.gov/. Retrieved: 2018-07-01.

[9] Indigenous land use agreements. https://data.gov.au/dataset/indigenous-land-use-agreements-registered-or-in-notification/. Retrieved: 2018-07-01.

[10] JSON-C - a JSON implementation in C. https://github.com/json-c/json-c. Retrieved: 2018-07-01.

[11] JSONiq: The JSON query language. http://jsoniq.org/. Retrieved: 2018-07-01.

[12] Main portal page for the Jackson project. https://github.com/FasterXML/jackson/. Retrieved: 2018-07-01.

[13] National statistics postcode lookup UK. https://data.gov.uk/dataset/national-statistics-postcode-lookup-uk/. Retrieved: 2018-07-01.

[14] RapidJSON. http://rapidjson.org/. Retrieved: 2018-07-01.

[15] A streaming JsonPath processor in Java. https://github.com/jsurfer/JsonSurfer/. Retrieved: 2018-07-01.

[16] Twitter developer API. https://developer.twitter.com/en/docs/. Retrieved: 2018-07-01.

[17] Twitter usage statistics. http://www.internetlivestats.com/twitter-statistics/. Retrieved: 2018-07-01.

[18] 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: 2018-07-01.

[19] Alfred V Aho. *Compilers: principles, techniques and tools (for Anna University), 2/e*. Pearson Education India, 2003.

[20] Kevin Angstadt, Arun Subramaniyan, Elaheh Sadredini, Reza Rahimi, Kevin Skadron, Westley Weimer, and Reetuparna Das. ASPEN: A scalable In-SRAM architecture for pushdown automata. In *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2018.

[21] Apache Fundation. Apache CouchDB. http://couchdb.apache.org/. Retrieved: 2018-07-01.

[22] Iliana Avila-Campillo, Todd J Green, Ashish Gupta, Makoto Onizuka, Demian Raven, and Dan Suciu. Xmltk: An xml toolkit for scalable xml stream processing. 2002.

[23] Alessandro Barenghi, Stefano Crespi-Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. Parallel parsing made practical. *Sci. Comput. Program.*, 112:195–226, 2015.

[24] Roy H. Campbell and Reza Farivar. Cloud computing applications, part 1: Cloud systems and infrastructure. https://www.coursera.org/learn/cloud-applications-part1. Retrieved: 2018-07-01.

[25] Arijit Chakraborty. An introduction to REST and JSON. https://blogs.oracle.com/cloud-platform/an-introduction-to-rest-and-json. Retrieved: 2018-07-01.

[26] Songting Chen, Hua-Gang Li, Jun'ichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Twig^{2}stack: Bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 283–294, 2006.

[27] Yanlei Diao, Peter M. Fischer, Michael J. Franklin, and Raymond To. Yfilter: Efficient and scalable filtering of XML documents. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 341–342, 2002.

[28] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[29] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[30] W3C Editor. Overview of the CoverageJSON format. https://w3c.github.io/sdw/coverage-json/. Retrieved: 2018-07-01.

[31] Chris Esplin. Firebase data modeling. https://howtofirebase.com/firebase-data-modeling-939585ade7f4. Retrieved: 2018-07-01.

[32] Todd J Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing xml streams with deterministic automata. In *International Conference on Database Theory*, pages 173–189. Springer, 2003.

[33] GeoJSON Working Group. The GeoJSON specification (RFC 7946). http://geojson.org/. Retrieved: 2018-07-01.

[34] Netork Working Group. NetJSON: data interchange format for networks. http://netjson.org/rfc.html. Retrieved: 2018-07-01.

[35] W3C JSON-LD Community Group. JSON for linking data. https://json-ld.org/. Retrieved: 2018-07-01.

[36] Venkat N Gudivada, Dhana Rao, and Vijay V Raghavan. NoSQL systems for big data management. In *Services (SERVICES), 2014 IEEE World Congress on*, pages 190–197. IEEE, 2014.

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

[38] IBM. XML-SAX (parse an XML document). https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_73/rzasd/zzxmlsa.htm. Retrieved: 2018-07-01.

[39] Lin Jiang and Zhijia Zhao. Grammar-aware parallelization for scalable XPath querying. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 371–383. ACM, 2017.

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

[41] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, 1980.

[42] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: A fast JSON parser for data analytics. *PVLDB*, 10(10):1118–1129, 2017.

[43] Yishan Li and Sathiamoorthy Manoharan. A performance comparison of SQL and NoSQL databases. In *Communications, computers and signal processing (PACRIM), 2013 IEEE pacific rim conference on*, pages 15–19. IEEE, 2013.

[44] Dan Lin, Nigel Medforth, Kenneth S. Herdy, Arrvindh Shriraman, and Robert D. Cameron. 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*, pages 373–384, 2012.

[45] Logicworks. The future of AWS' cloud: Infrastructure as an application. https://www.cloudcomputing-news.net/news/2016/jun/02/the-future-of-aws-cloud-infrastructure-as-an-application/. Retrieved: 2018-07-01.

[46] Isaac Lopez. Amazon hosting 20 TB of climate data. https://www.datanami.com/2013/11/12/amazon_hosting_20_tb_of_open_climate_data/. Retrieved: 2018-07-01.

[47] Wei Lu, Kenneth Chiu, and Yinfei Pan. A parallel approach to XML parsing. In *7th IEEE/ACM International Conference on Grid Computing (GRID 2006), September 28-29, 2006, Barcelona, Spain, Proceedings*, pages 223–230, 2006.

[48] Wei Lu and Dennis Gannon. Parallel XML processing by work stealing. In *Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*, pages 31–38. ACM, 2007.

[49] MongoDB. MongoDB extended JSON. https://docs.mongodb.com/manual/reference/mongodb-extended-json/. Retrieved: 2018-07-01.

[50] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 529–542, 2014.

[51] Peter Ogden, David Thomas, and Peter Pietzuch. Scalable XML query processing using parallel pushdown transducers. *Proceedings of the VLDB Endowment*, 6(14):1738–1749, 2013.

[52] Peter Ogden, David B. Thomas, and Peter R. Pietzuch. AT-GIS: highly parallel spatial query processing with associative transducers. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1041–1054, 2016.

[53] Oracle. Parsing an XML file using SAX. https://docs.oracle.com/javase/tutorial/jaxp/sax/parsing.html. Retrieved: 2018-07-01.

[54] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Filter before you parse: Faster analytics on raw data with sparser. *Proceedings of the VLDB Endowment*, 11(11):1576–1589, 2018.

[55] Christina Pavlopoulou, E. Preston Carman Jr., Till Westmann, Michael J. Carey, and Vassilis J. Tsotras. A parallel and scalable processor for JSON data. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018.*, pages 576–587, 2018.

[56] Junqiao Qiu, Zhijia Zhao, and Bin Ren. Microspec: Speculation-centric fine-grained parallelization for fsm computations. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, pages 221–233. IEEE, 2016.

[57] Junqiao Qiu, Zhijia Zhao, Bo Wu, Abhinav Vishnu, and Shuaiwen Leon Song. Enabling scalability-sensitive speculative parallelization for fsm computations. In *Proceedings of the International Conference on Supercomputing*, page 2. ACM, 2017.

[58] Sqlizer. A brief history of JSON. https://blog.sqlizer.io/posts/json-history/. Retrieved: 2018-07-01.

[59] Arun Subramaniyan, Jingcheng Wang, Ezhil RM Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. Cache automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 259–272. ACM, 2017.

[60] TwoBitHistory. The rise and rise of JSON. https://twobithistory.org/2017/09/21/the-rise-and-rise-of-json.html. Retrieved: 2018-07-01.

[61] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference (10CCC), 2015 10th*, pages 583–590. IEEE, 2015.

[62] Philipp Wehner, Christina Piberger, and Diana Gohringer. Using JSON to manage communication between services in the internet of things. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*, pages 1–4. IEEE, 2014.

[63] Ying Zhang, Yinfei Pan, and Kenneth Chiu. A parallel xpath engine based on concurrent NFA execution. In *16th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2010, Shanghai, China, December 8-10, 2010*, pages 314–321, 2010.

[64] Zhijia Zhao and Xipeng Shen. On-the-fly principled speculation for FSM parallelization. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 619–630, 2015.

[65] Zhijia Zhao, Bo Wu, and Xipeng Shen. Challenging the embarrassingly sequential: parallelizing finite state machine-based computations through principled speculation. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 543–558. ACM, 2014.