Low-Latency Streaming Evaluation of JSONPath Queries

Jana Kostičová[1]

[1]Faculty of Mathematics, Physics and Informatics, Comenius University, Mlynská dolina, Bratislava, Slovakia

Abstract

The recent standardization of the JSONPath language underscores the need for efficient JSONPath query evaluation over JSON (JavaScript Object Notation) data. In this work, we focus on real-time scenarios where available methods may have a too high latency. We identify a subset of JSONPath queries that enable efficient evaluation with a constant number of steps per event in the input JSON stream. This constant time evaluation guarantees low and predictable latency, making the approach highly suitable for real-time applications. The algorithm uses a streaming tree transducer (SST) as the underlying formal model. The JSONPath query is compiled into a finite state automaton (FSA) and the transducer keeps track of the evaluation status by storing FSA states in its stack. We present formal background, complexity analysis and implementation details of the algorithm.

Keywords

JSON, JSON querying, JSONPath, real-time data processing, low latency

1. Introduction

Fast and low-latency solutions are becoming a crucial part of today's data pipelines to satisfy the increasing demand for real-time (RT) or near-real-time processing (NRT). Such solutions are required to handle data continuously, with minimal delay, and overcome the inherent challenge of limited data availability at any given moment [1]. Querying data streams emerges as a versatile tool in this context - beyond direct output generation of filtered data, it serves as a necessary first step for various intricate processing tasks, such as data cleaning, entity resolution, data validation, data transformation, and feature engineering [2, 3].

This paper focuses on querying JSON data. JSON (JavaScript Object Notation) [4, 5] is an extremely popular tree-based format for data storage and data transmission thanks to its lightweight nature, easy readability, and straightforward mapping to object-oriented models. In case of tree data, the streaming processing involves traversing the data in a specific order (preorder) and using queries to address locations within the tree. The common query language for JSON is the recently standardized JSONPath language [6]. It is based on the XPath language used for querying another tree format XML (Extensible Markup Language) [7].

Most of the JSONPath processors are tree-based, i.e., they load all of the input data into memory and then perform the evaluation. This approach is apparently not suitable for RT / NRT scenarios, especially for large input data streams. Delays are introduced by the initial loading step, by evaluating the JSONPath queries against the in-memory representation, or by writing the data from the memory to the output stream. The memory footprint is typically proportional to the size of the input data.

During the streaming processing of JSON data the input is read sequentially and similarly the output is generated sequentially. A stack is typically used to store information related to the current tree level of the input data. If a part of the input data needs to be processed in memory, it must be temporarily stored in dedicated memory buffer(s). Basically, these parts are processed in a tree-based manner that inherently results in processing delays, as described above. Therefore, real-time scenarios demand minimal input buffering.

The presented algorithm is intended to be used in real-time data pipelines including rapid data transformation, such as restructuring data for aggregation purposes (e.g., combining social media posts into a unified format) and entity resolution tasks (e.g., matching customer records across disparate datasets). However, it is useful at each data processing task that requires low-latency querying JSON data streams.

The main contribution of this paper is two-fold. First, we present formal foundations that enable us to analyze the complexity of the streaming processing of JSONPath queries. Second, on the top of this formalism, we identify the JSONPath subset that can be evaluated consistently with low latency and design the algorithm that realizes such an evaluation. The algorithm consists of two steps: (1) the compilation step when the query is compiled into a finite automaton, and (2) the evaluation step based on a tree transducer, when the query is evaluated over an input JSON stream and an output JSON stream is

ITAT 2024 Information Technologies – Applications and Theory 2024

kosticova@dcs.fmph.uniba.sk (J. Kostičová)

generated. The evaluator performs a constant number of steps per single event in input JSON stream that ensures predictable consistent latency for RT / NRT processing. The proposed solution is compared with other streaming JSONPath evaluators.

1.1. Related Work

To our best knowledge, ours is the first approach to analyzing the JSONPath language for its suitability in scenarios with low-latency requirements. There are two streaming solutions that focus particularly on JSONPath queries. Both of them were presented before JSONPath standardization, and thus the languages being evaluated differ from the resulting standard in a few aspects. JsonSurfer [8] processes incoming events on the fly, but still builds a partial internal representation of input data. It cannot process queries with descendat axis and star selector. JPStream [9] compiles queries into a finite state automaton and combines it with parsing automaton to obtain the resulting streaming automaton. The authors also present a parallelization mechanism to speed up the evaluation. Both of these solutions lack latency analysis, and it generally varies for different queries.

Much prior research has focused on analyzing the streaming processing of XPath language. In [10] authors present a streaming algorithm that evaluates a subset of XPath called Univariate XPath in $O(|D||Q|)$ time, where $|D|$ is the size of the input document and $|Q|$ is the size of the query. The SPEX evaluator [11] compiles XPath query into a network of deterministic pushdown transducers and processes it in polynomial combined complexity. The algorithm proposed in [12] is based on deterministic nested word automata and runs in polynomial time. Although several of the approaches to XPath streaming evaluation possess strong formal foundations, their algorithms are not directly applicable to JSONPath due to inherent differences between the two languages. Furthermore, these solutions focus primarily on overall time or space complexity rather than latency and often support query constructs like backward axes and predicates that can lead to unpredictable latency behavior and hinder their applicability in RT/NRT processing.

2. Formal Foundations

This section presents a formal basis for analyzing the complexity of the streaming JSONPath evaluation. We establish a set of customized formal models represeting JSON data, JSONPath queries and JSON transformations.

2.1. JSON Abstraction

At the root level, JSON data contain a root value. This value can be one of the following:

- An *object* that consists of zero or more unordered name/value pairs (also called *members*).
- An *array* that consists of zero or more ordered values (also called *array elements*).
- A *literal value* without internal structure that is either a number value, string value, or one of the literal names (true, false, null).

JSON data can be represented as a tree that is obtained by a natural one-to-one mapping between values and internal nodes of the tree. Reading the JSON data stream then exactly corresponds to the preorder traversal of the constructed tree. We base our JSON abstraction on the model introduced in our previous work [13]. We label the nodes corresponding to the array elements by their index - similar technique is used in [14] . We omit literal values since they are not used in our evaluation algorithm. Section 5 describes how they are handled in implementation. We formally define two JSON structure types: subtrees with arbitrary root identifier labeled by any object name, and trees with the root identified by "0" and labeled by the designated root symbol "$".

Let Σ be an alphabet of object names. The set of *JSON subtrees* over Σ is denoted by S_Σ. We use dynamic level numbers [15] to identify nodes within the JSON subtree. The node identifier (id) is a sequence of numeric values, separated by point. The root node has some root id and the id of any other node consists of the parent's id, point, and the position of the node among its siblings. This approach allows us to trivially determine the relations between any two given nodes such as the ancestor-descendant relation, the parent-child relation, the sibling relation. We include zero in node ids to get consistency with the indices of JSON array elements. The JSON subtree always contains at least the root node, as the JSON data must consist of exactly one root value. One ambiguity exists in this model - the JSON subtree consisting of the root node only can represent a root value equal to either an empty object or to an empty array. However, this does not affect the design of our algorithm, as shown later.

The JSON subtree $s \in S_\Sigma$ is then defined as a pair $s = (V_s, \lambda_s)$ where

- $V_s \subseteq 2^{(\mathbb{N}_0 \cup \{.\})^*}$ is a set of node ids (shortly nodes), and

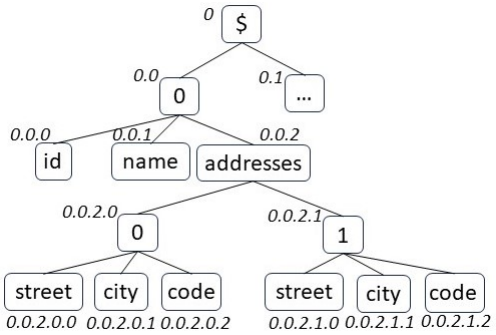

Figure 1: An example of JSON tree

- $\lambda_s : V_s \to (\Sigma \cup \mathbb{N}_0)$, $\Sigma \cap \mathbb{N}_0$, is the labeling function.

Note that the set of nonnegative integers $\mathbb{N}_0$ is used both for creating node ids and for labeling nodes by array indices. It is required that (1) node ids in V_s form a tree, and (2) in case of array node, the labels of its children form an ordered sequence of indices starting from zero.

A *JSON tree* over Σ is a JSON subtree $t = (V_t, \lambda_t)$ over Σ where

$$0 \in V_t,\ \$ \in \Sigma \text{ and } \lambda_t(0) = \$.$$

We denote the set of JSON trees over Σ by T_Σ. An example of a JSON tree is depicted in Fig. 1.

The following notions and notations are used throughout this paper: Let s be a JSON subtree. We denote the (finite) set of all labels that occur in s by $\Lambda(s)$. We refer to the root node of s by $root(s)$ and to the preorder relation between nodes $u, v \in V_s$ by $u \prec v$. We define the set of events over some label alphabet Λ as $E(\Lambda) = \{begin(\sigma) \,|\, \sigma \in \Lambda\} \cup \{end\}$. An *event stream at* $v \in V_s$ is then defined recursively as follows:

$$stream(v) = \\ begin(\lambda_s(v))\ stream(v.0) \ldots stream(v.n)\ end,$$

where $v.0, \ldots, v.n$ are children of v. The *event stream* of the JSON tree t equals to the event stream at its root: $stream(t) = stream(0)$.

The structure of the JSON data can optionally be constrained by a schema. Based on this aspect, we distinguish schema-agnostic and schema-aware approaches to JSON processing. Since it is pretty common for real-world JSON data streams to be schemaless, in this paper we focus on schema-agnostic JSONPath evaluation. This approach introduces some additional issues and restrictions; most prominently, it is necessary to deal with an unknown input alphabet and the query set that can be processed with guaranteed low latency is more restricted. In Section 6 we outline future research that would benefit from schema availability.

2.2. JSONPath

Based on recently published IETF RFC 9535 [6] (hereafter referred to as the *JSONPath standard*), we use the following grammar to describe the JSONPath queries:

query	::=	$\$\,segment*$	
segment	::=	*axis* [*selector*(,*selector*)*]	
axis	::=	. (child) | .. (descendant)	
selector	::=	*name*	(name selector)
		| *index*	(index selector)
		| *	("any" selector)
		| *start*:*end*:*step*	(array slice selector)
		| ?(*predicate*)	(filter selector)

Filter selectors contain predicate - a boolean expression over other nodes and literal values. They are evaluated with respect to the *current node*. We do not provide details since their specification is rather complex and, as mentioned later, they are excluded from the queries under consideration. We shall use the common term *label selector* for the name and index selectors. We use *dot notation* for JSONPath queries throughout this paper.

A query starts with a special root selector $. It selects the root of the input JSON tree where the evaluation starts. Then a sequence of *segments* follows. Each segment consists of (1) an *axis* (child, descendant) that selects the node set to be processed (either children or descendants of the current node) and (2) a sequence of *selectors* that apply further node filtering on a given node set.

Example 1. *Consider the JSONPath query* $.[0].addresses..street *with three segments. When evaluated over JSON tree at Fig. 1, first segment selects the child of the root node being an array element indexed by* 0*, the second segment selects child object named addresses, and the last segment selects all descendants named street. Formally, the query returns literal values at nodes* 0.0.2.0.0 *and* 0.0.2.1.0*.*

Let θ be a query and t a JSON tree. We denote the list of JSON subtrees returned by evaluating θ over t according to the semantics described in the JSONPath standard by $eval(\theta, t)$.

2.3. Simple Streaming Transducer

Tree transducers are well-established formal models for performing tree or forest transformations. We adapt the streaming model from our previous work [13] due to its simplicity and ability to cover aspects of streaming transformations of tree data without any input buffering. We customize it to work directly on event streams rather than trees. We consider a basic model with a single output stream. The simple streaming transducer (SST) is a 7-tuple $T = (Q, \Lambda_{in}, \Lambda_{out}, \Gamma, q_0, Z_0, R)$, where

- Q is a finite set of states,
- Λ_{in} is the alphabet of input labels,
- Λ_{out} is the alphabet of output labels,
- Γ is an alphabet of stack symbols,
- q_0 is the initial state, and
- R is a finite set of rules of the form:

$$Q \times E(\Lambda_{in}) \times \Gamma \rightarrow (E(\Lambda_{out}))^* \times (Q \times \Gamma^*).$$

In addition, if $(q, e_{in}, z) \rightarrow e_{out}(q', z') \in R$ then e_{out} must be a substring of an event stream of some JSON tree.

The transducer rules can modify the state of the transducer, manipulate the top element of the stack, and optionally generate a sequence of output events. The data storage mechanism is limited to the state and the stack. We restrict a single rule to generate a JSON substream. This, however, does not guarantee that the overall output is correct JSON stream, as that must be ensured by the transducer rules.

3. Low-Latency Processing Constraints

In our previous work [13] we have discussed properties of XML transformations that allow for streaming processing without input buffering. We adapt them to the context of low-latency JSONPath query evaluation. We add the property of local evaluability to classify the query constructs according to the context necessary for their evaluation. We analyze the constructs of the JSONPath language and identify which of them violate these conditions for some input JSON trees.

The following facts should be pointed out: First, this work is focused on a schema-agnostic approach, where no information about the structure of the input data is available. When analyzing specific JSONPath construct, the worst-case input data is searched within the group of all valid JSON data. This can result in the identification of worst-case scenarios that are potentially more severe than those encountered in schema-bound environments. Second, members of an object are unordered in JSON format. This means that we must consider any member order when looking for the worst-case scenarios.

3.1. Order-Preserving Queries

The query θ is *order-preserving* if and only if the roots of the subtrees returned by θ are in preorder with respect to an arbitrary input JSON tree t. That is, if $s_1, s_2 \in eval(\theta, t)$ are two subtrees returned by evaluating θ over t such that s_1 is returned before s_2, then it must hold $root(s_1) \prec root(s_2)$.

See Fig. 2 a) for an example of a query that results in reordering input subtrees s_1 and s_2 and thus is not order-preserving. Consider that an evaluator with additional memory is employed for processing such query. Upon encountering s_1 the evaluator copies its contents into memory to store it for later usage (s_1 needs to be outputted later) while continuing query evaluation. When s_2 is encountered, it is outputted directly. After processing s_2, the stored copy of s_1 is outputted, introducing a delay proportional to its size. Since we consider schema-agnostic approach, the size of s_1 and consequently the length of the delay is not upper-bounded.

The following JSONPath constructs violate the order-preserving property for some input JSON tree:

- *Negative step in the array slice selector* results in the selection of the array elements in reverse order. This in general leads to reordering of some input subtrees.
- *Sequence of reordering selectors within a segment* again results in reordering some input subtrees. For example, a sequence of index selectors in reverse order is clearly reordering.

Note that the order-preserving property is violated only when dealing with array-based selectors due to the inherent order of array elements.

3.2. Branch-Disjoint Queries

The query θ is *branch-disjoint* if and only if the subtrees returned by θ do not overlap for any input JSON tree t. That is, if $s_1, s_2 \in eval(\theta, t)$ are two subtrees returned by evaluating θ over t, it must hold $root(s_1) \neq root(s_2).v'$. See Fig. 2 b) for an example of a non-branch-disjoint query that results in copying the subtree s_2. Here upon encountering the subtree s_2, it needs to be stored in the memory and outputted later that results in a delay of the length proportional to the size of s_2. In schema-agnostic approach, both the size of s_2 and the delay are not upper-bounded.

This property is violated (for some input JSON tree) by using a descendant axis in some of the query

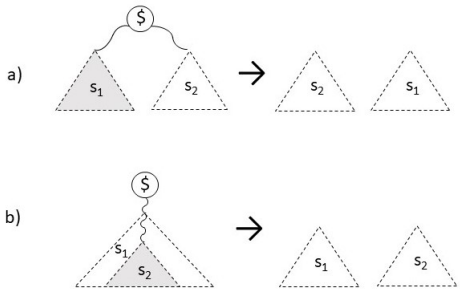

Figure 2: a) Non-order-preserving query b) Non-branch-disjoint query

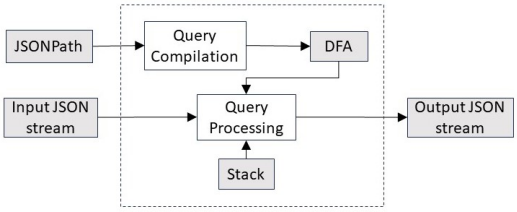

Figure 3: Low-latency evaluator

segments because it allows one to return two or more overlapping subtrees by the evaluation of that query.

3.3. Locally Evaluable Queries

The query θ is *locally evaluable* if and only if no forward context is needed to evaluate θ for arbitrary input JSON tree t. Formally: let $\theta = \theta'\ seg\ segList$ where seg is a segment and $segList$ is a list of segments (possibly empty), let $s_1 \in eval(\theta', t)$, $s_2 \in eval(\$\ seg\ segList, s_1)$, then seg does not refer to any node u such that $root(s_2) \prec u$.

In case of query that cannot be evaluated locally, when the evaluator encounters the subtree s_1, it needs to visit some forward context node u to be able to decide whether s_2 needs to be outputted. Thus, it is necessary to store s_2 in the memory to be able to evaluate such query in our processing model (enhanced with additional memory). In case of schema-agnostic approach, the size of s_2 is not limited and introduces delay proportional to that size.

The following JSONPath constructs violate this property for some input JSON tree:

- *Negative indices in array-based selectors* allow the access to array elements relative to the end of the array. To process a node at index $-i$, the evaluator must locate the $(n - i)$-th child of the current node, where n is the total number of children.
- *Filter selectors* can contain relative or absolute subqueries that may need some forward context to be evaluated.

3.4. Identified Low-Latency JSONPath Subset

Based on the analysis above, we define the subset of JSONPath language suitable for low-latency streaming processing by the following rules for queries:

- A segment contains either a single selector or a sequence of label and array slice selectors that preserve the ascending order of selected array indices.
- Only nonnegative values for array indices in index/array slice selectors are allowed.
- Only a positive value for the *step* operator in the array slice selector is allowed.
- Filter selectors (predicates) are excluded.

Within the JSONPath query segment, the order of index selectors must be preserved, while name selectors can appear in any order due to the unordered nature of JSON object members. The descendant axis is allowed in queries, as it typically satisfies the branch-disjoint condition in most practical scenarios. In cases where the branch-disjoint condition is violated, a runtime error is explicitly reported, since the query cannot be processed within our model's constraints. This deviation from standard JSONPath semantics is necessary to guarantee low-latency evaluation. Filter selectors are excluded - although some array-based filter selectors not necessarily violate local evaluability, it would require a separate complex analysis to define this subset clearly and distinguish it from non-locally evaluable ones.

We refer to the identified JSONPath subset as the *low-latency subset* later in this paper, and we mention several approaches to address some of its limitations in Section 6.

4. Low-Latency Evaluator

The proposed SST-based evaluator consists of the query compilation step and the query processing step (see Fig. 3.). This section presents both tasks in detail and discusses their complexity.

4.1. Query Compilation

The subset of JSONPath defined in Section 3 can be rewritten to regular expressions, but first we

need to establish a finite alphabet that constrains the language described.

Let θ be a query; then the query label alphabet $\Lambda(\theta)$ consists of all labels that appear within the query (both the object names and the indices) plus the finite set of indices within the ranges defined by the array slice operators in θ. As mentioned earlier, in the context of schema-agnostic streaming processing, neither the set of object names nor array indices is known in advance. Since traditional finite automata work on finite input alphabets, we need to overcome this issue. It is easy to observe that when evaluating θ, handling of input symbols not in $\Lambda(\theta)$ is exactly the same. Therefore, we can parse these symbols into a single token represented by a special placeholder label σ_{ph}. Compilation to regular expressions is based on these principles:

- Label selector: maps to the label itself.
- Star selector: represents a choice of all labels.
- Array slice selector: represents a choice of indices contained within the defined range (we allow only fixed upper bound for the array slice selector).
- Selector sequence: represents a choice of regular subexpressions for particular selectors.
- Child axis: maps to a concatenation operation.
- Descendant axis: represents a concatenation followed by a Kleene star over the query alphabet and the placeholder σ_{ph}, and another concatenation (in this order).

Formally, we then rewrite queries to regular expressions as follows:

Let $\theta = \$$ be a query without any segment, then

$$regexp(\theta) = \$.$$

Let $\theta = \theta'\ axis\ [sel_1, \ldots, sel_n]$ be a query with at least one segment, then

$$regexp(\theta) = regexp(\theta')\,\gamma\,(regexp(sel_1) \mid \ldots \mid regexp(sel_n))$$

where $\gamma = \varepsilon$ for the child axis and $\gamma \in (\Lambda(\theta) \cup \{\sigma_{ph}\})^*$ for the descendant axis, and $regexp(sel_i)$, $i \in \{1, \ldots, n\}$ is the regular expression for the selector sel_i defined as follows:

Label selector: If $sel_i = name$ or $sel_i = index$ then

$$regexp(sel_i) = sel_i.$$

Star selector: If $sel_i = *$ then

$$regexp(sel_i) = (\sigma_1 | \ldots | \sigma_n), \quad \sigma_j \in \Lambda(\theta) \cup \{\sigma_{ph}\},\ j \in \{1, \ldots, n\}.$$

Array slice selector: If $sel_i = (start{:}end{:}step)$, $start, end \geq 0$, $step > 0$ then

$$regexp(sel_i) = (in_1 | \ldots | in_n), \text{ where } start \leq in_j < end,\ (in_j - start) \equiv 0 \pmod{step},\ j \in \{1, \ldots, n\}.$$

We denote the deterministic finite automaton (DFA) recognizing the language described by $regexp(\theta)$ by $A(\theta)$. Note that this is the language of all label sequences that can appear in a path starting from the root of the tree to the matched subtree. The automaton processes words over the alphabet $\Lambda(\theta) \cup \{\sigma_{ph}\}$ and can be constructed by standard algorithms [16]. We assume, without loss of generality, that $A(\theta)$ is complete.

4.2. Query Processing

We present a query processing algorithm by means of a construction of a streaming tree transducer. Let θ be a query and $A(\theta) = (Q_A, \Sigma_A, \delta_A, p_0, F_A)$. Then the simple streaming transducer evaluating θ over an input JSON stream is constructed as follows:

$$T(\theta) = (Q, \Lambda_{in}, \Lambda_{out}, \Gamma, q_e, p_0, R)$$

- $Q = \{q_{eval}, q_{gen}, q_{err}\}$,
- $\Lambda_{in} = \Lambda_{out} = \Sigma_A$,
- $\Gamma = Q_A \cup Q_{A,g}$, where $Q_{A,g} = \{p_g \mid p \in Q_A\}$,
- R contains the following rules:

a) Evaluation rules:

$$\begin{aligned}
(q_{eval}, start(\sigma), p) &\rightarrow (q, pp'), \text{ where } \delta_A(p, \sigma) = p', \text{ and} \\
&\quad p' \notin F_A \Rightarrow q = q_{eval}, \\
&\quad p' \in F_A \Rightarrow q = q_{gen}, \\
(q_{eval}, end, p) &\rightarrow (q_{eval}, \varepsilon).
\end{aligned}$$

b) Generation rules:

$$\begin{aligned}
(q_{gen}, start(\sigma), p) &\rightarrow start(\sigma)(q, pp'_g), \text{ where } \delta_A(p, \sigma) = p', \text{ and} \\
&\quad p' \notin F_A \Rightarrow q = q_{eval}, \\
&\quad p' \in F_A \Rightarrow q = q_{err}, \\
(q_{gen}, end, p) &\rightarrow end(q_{gen}, \varepsilon), \text{ where } p \in Q_{A,g}, \\
(q_g, end, p) &\rightarrow end(q_{eval}, \varepsilon), \text{ where } p \neq Q_{A,g}.
\end{aligned}$$

4.2.1. Evaluation

The q_{eval} state is used to evaluate the query. At the beginning, the stack contains the initial state of DFA $A(\theta)$ and we encounter the *start*(\$) event of the JSON stream. The first transition is made based on the transition of $A(\theta)$ for the initial state and the label \$. Let assume $A(\theta)$ moves into the state p'.

- If p' is one of the final states of $A(\theta)$, it means that we have found a match for the given query and that we need to send the subtree of the current node to the output. Therefore, the transducer moves to the generating state q_{gen} puts the next state of $A(\theta)$ on the stack.
- If p' is not a final state of $A(\theta)$, it means that we have not found a match and we need to proceed with the evaluation. Thus, the transducer remains in the evaluation state q_{eval} and puts the next state of $A(\theta)$ on the stack.

4.2.2. Generation

In the generating state q_{gen}, the transducer puts the whole subtree to the output. It continues in evaluation only in order to detect the scenarios when input query violates the branch-disjoint condition on given input. In that case it enters the error state q_{err}. The evaluation is based on the copies of DFA states (denoted by the subscript g) so that the transducer is able to recognize the end of the subtree and return to the evaluation state.

4.2.3. Complexity Summary

It is easy to see that the transducer performs a single transition at each event. Implementation of this transition actually consists of several operations (reading the input stream, parsing, DFA transition, manipulating the stack, generating the output stream). However, as shown in the following section, the latency remains consistent. A stack of the size proportional to the depth of the input JSON tree is used. In addition, the size of the constructed DFA counts towards the overall memory footprint used by the evaluator. Since this work is not focused on minimizing the memory usage, further optimizations might be possible to reduce the DFA's size.

5. Implementation

We prototyped a low-latency SST-based evaluator in Java to validate our design of an SST-based algorithm for JSONPath evaluation, to ensure that our design translates to actual performance as intended and to compare it to other JSONPath evaluators. We used the Jackson streaming parser [17] for event-based parsing. Java was chosen as it is well-suited for rapid prototyping that facilitates our primary goals mentioned above.

The following mapping issues have been identified and resolved:

- *Root symbol.* Since the root symbol \$ does not occur in JSON data, it is generated at the beginning of the evaluation.
- *Arrays.* JSON does not explicitly index array elements, thus the indices are generated at evaluation time. For each array value, we store the current size of the array (*size*) on the stack. When a new array element is encountered, the *size* value is popped out of the stack, and a surrogate label *size* is generated that represents the index of given element. The *size* value is incremented and pushed back on to the stack. Array delimiters are filtered out since they are not used by the transducer.
- *Placeholder symbol.* All symbols not mentioned in regular expression are replaced by the placeholder symbol σ_{ph}.
- *Literal values.* The proposed algorithm abstracts from literal values. As the low-latency JSONPath subset does not use literal values in queries, they are skipped in the evaluation mode and outputted in the generating mode.

Focusing primarily on the measuring the latency of query processing step, additional features (e.g., compiled query storage), although important in real-world use cases, have not been implemented.

We compared the presented evaluator with both JSONPath streaming evaluators mentioned in Section 1.1. For the Java-based evaluators (our SST-based implementation and JsonSurfer [8]), we employed the JMH framework [18] to measure running time per single event, mitigating GC impact through extended warmup, forced GC, and JMH's built-in techniques. The C-based JPStream [9] was evaluated using the `clock_gettime` system call, providing sufficient precision for our comparative analysis.

As a proof-of-concept optimized for benchmarking, the implementation of SST-based evaluator strictly follows the code structure required by JMH. On the other hand, JsonSurfer needed some code restructuring to make JMH-based evaluation possible. As JPStream makes two passes over the input and the latency is then inherently high, only the first pass was measured and the second one was skipped,which resulted in excluding filter expressions from the test queries. Two test suites were used:

- *Test suite 1* consists of test data and queries from the JPStream project [9], while excluding constructs outside the low-latency subset. Extra queries with descendant axis were added. This suite was aimed to ensure that the SST-based evaluator exhibits low and consistent latency for these queries and to enable comparison with existing solutions.
- *Test suite 2* uses the same test data, but queries with constructs beyond the low-latency subset were chosen (negative indices, negative steps, filter expressions, reordering selectors, descendant axis violating branch-disjoint condition). This suite was intended to confirm the need for high-latency processing of these constructs by other evaluators. It was not meant to be processed by SST-based evaluator as it does not support them intentionally.

	Test suite 1	Test suite 2	Method
SST-based evaluator	85 to 88 ns, me < 2.15 ns	N/A	JMH
JsonSurfer	333 to 347 ns, me < 8.2 ns	N/A	JMH
JPStream	20 to 2615 ns	N/A	clock_ gettime

Table 1
Comparison with JsonSurfer and JPStream ("me" stands for margin error)

The evaluation results are shown in Table 1. For Test suite 1, the time per event (including parsing and output generation) exhibits lowest values and highest consistency in the case of the SST-based evaluator presented in this paper. The values for the other two evaluators are still reasonably low and consistent. Although they are highest in case of the JPStream, it must be taken into account that different measurement method was used. Moreover, since we operate at the nanosecond level, these particular differences might be considered not significant. Evaluation showed that JsonSurfer and JPStream are not able to process the whole low-latency JSONPath subset, so they are less powerful than the SST-based evaluator when latency matters. JsonStream does not support star selector and descendant axis, JPStream does not support multiple selectors and step in array slice selector. Although these two evaluators demonstrate low latency for supported test queries, it is important to mention that they lack formal latency analysis, preventing a clear understanding of their latency guarantees for different query types.

Surprisingly, queries in Test suite 2 failed to process correctly with both JsonSurfer and JPStream, so it was not possible to perform planned measurements. JsonSurfer produced incorrect results, while JPStream either does not support the constructs explicitly (negative indices, multiple selectors, step operator), requires additional pass over the input (filter expressions), or yields unhandled exception (queries with descendant axis violating the branch-disjoint condition).

6. Conclusion and Future Work

This work analyzes a new JSONPath standard and identifies a powerful subset of JSONPath queries that can be evaluated over JSON data streams performing a constant number of steps at each event. An evaluation algorithm has been presented that is based on finite automata and tree transducers. The constant time complexity per input event is guaranteed and leads to low-latency processing as required in real-time / near-real-time scenarios. The algorithm operates within a basic processing model characterized by schema-agnosticism, single-query execution, sequential processing, and a single output stream. The presented evaluator has been compared with two other JSONPath streaming evaluators, and the results show that it not only supports the larger low-latency subset, but even more importantly, based on its formal background it gurantees the latency to be low and consistent for all queries from the low-latency subset and any input JSON data.

The work is in progress; to enhance the baseline algorithm, future work will focus on refining JSONPath predicate analysis to identify a processable subset and optimizing the compiled query's space complexity.

The presented evaluator is designed to process a well-defined subset of JSONPath queries based on a specific set of rules. These limitations can potentially be relaxed by exploring several extensions to the basic processing model, such as

- incorporating knowledge about the structure of the input JSON data (e.g., by employing schema-aware or learning approaches),
- introducing parallelism and /or parallel output streams.

As mentioned earlier, making assumptions about the structure of the input JSON data may considerably limit the set of possible input streams. If this leads to the exclusion of input streams that participate in the worst-case scenarios identified in this work, a larger subset of JSONPath can be possibly processed by

the low-latency evaluator. Parallelism and parallel output streams allow us to process multiple tasks at the same time, and such an approach could eliminate some situations where one task is waiting for another in a sequential model. However, the exact benefit of this approach depends on the particular definition of the parallel processing model (e.g., the way in which the output streams are synchronized).

Another possible direction for future work involves extending the processing model to evaluate multiple queries in parallel. While the current design focuses on single-query processing, a straightforward approach to achieve multiquery evaluation could involve: a) compiling each query into DFA, and b) maintaining a stack of state tuples, where each tuple tracks the evaluation state for a specific query's DFA. However, this approach presents several challenges: new branch conflicts may arise due to parallel evaluation of multiple queries, and the overall space overhead of DFAs can increase significantly. For the latter case, techniques on combining DFA states could be explored [19].

References

[1] S. Chakravarthy, Q. Jiang, Stream Data Processing: A Quality of Service Perspective: Modeling, Scheduling, Load Shedding, and Complex Event Processing, Advances in Database Systems, Springer US, 2009.

[2] V. Christophides, V. Efthymiou, T. Palpanas, G. Papadakis, K. Stefanidis, An Overview of End-to-End Entity Resolution for Big Data, ACM Comput. Surv. 53 (2020).

[3] A. Zheng, A. Casari, Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists, O'Reilly Media, 2018.

[4] ECMA-404 The JSON data interchange syntax, 2nd Edition, ECMA, 2017. https://ecma-international.org/publications-and-standards/standards/ecma-404/.

[5] The JavaScript Object Notation (JSON) Data Interchange Format, Internet Engineering Task Force (IETF), 2017. https://datatracker.ietf.org/doc/html/rfc8259.

[6] JSONPath: Query Expressions for JSON, Internet Engineering Task Force (IETF), 2024. https://datatracker.ietf.org/doc/html/rfc9535.

[7] Extensible Markup Language (XML) 1.0 (Fifth Edition), W3C Recommendation, W3C, 2008. http://www.w3.org/TR/REC-xml.

[8] A streaming JsonPath processor in Java, 2024. https://github.com/jsurfer/JsonSurfer/, Retrieved: 2024-06-08.

[9] L. Jiang, X. Sun, U. Farooq, Z. Zhao, 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, ASPLOS '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 79–92.

[10] G. Gou, R. Chirkova, Efficient Algorithms for Evaluating XPath over Streams, in: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07, Association for Computing Machinery, New York, NY, USA, 2007, pp. 269–280.

[11] D. Olteanu, SPEX: Streamed and Progressive Evaluation of XPath, IEEE Transactions on Knowledge and Data Engineering 19 (2007) 934–949.

[12] O. Gauwin, J. Niehren, Streamable Fragments of Forward XPath, in: B. Bouchou-Markhoff, P. Caron, J.-M. Champarnaud, D. Maurel (Eds.), Implementation and Application of Automata, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 3–15.

[13] J. Dvořáková, Automatic Streaming Processing of XSLT Transformations Based on Tree Transducers, in: C. Badica, M. Paprzycki (Eds.), Advances in Intelligent and Distributed Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 86–94.

[14] P. Bourhis, J. L. Reutter, F. Suárez, D. Vrgoč, JSON: Data model, Query languages and Schema specification, PODS '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 123–135.

[15] T. Böhme, E. Rahm, Supporting Efficient Streaming and Insertion of XML Data in RDBMS, in: Proc. 3rd Int. Workshop Data Integration over the Web (DIWeb) 2004, 2004.

[16] A. V. Aho, S. Ravi, J. D. Ullman, Compilers: Principles, Techniques, and Tools (1st ed.), Addison-Wesley, 1986.

[17] Main Portal page for the Jackson project, 2024. https://github.com/FasterXML/jackson, Retrieved: 2024-06-08.

[18] Java Microbenchmark Harness (JMH), 2024. https://github.com/openjdk/jmh, Retrieved: 2024-06-08.

[19] Y. Diao, M. J. Franklin, High-Performance XML Filtering: An Overview of YFilter, IEEE Data Eng. Bull. 26 (2003) 41–48.