Space-Efficient Streaming JSON Transformations

Michelle Gavlák[1], Jana Kostičová[1,*]

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

Abstract

JSON is a widely used format for data transmission and storage, and the number of streaming and real-time JSON processing scenarios is increasing. Therefore, there is a growing demand for efficient JSON processing algorithms that operate under the constraints of streaming environments, most notably limited memory and low latency. In this paper, we present a framework for analyzing the memory complexity of streaming JSON transformations. The framework introduces a transformation specification language together with a formal model, the JSON Stream Transducer, enabling theoretical analysis of space usage. We use the proposed framework to design and implement memory-efficient algorithms for several common operations on JSON streams. The evaluation shows that empirical memory usage is consistent with the theoretical bounds: stack-class transformations require $O(d)$ space, where d denotes the depth of the input JSON stream, while buffer-class transformations require $O(n)$ space in the worst case, where n is the size of the input JSON stream, but typically exhibit substantially lower memory usage due to partial buffering.

Keywords

JSON, JSON transformation, streaming processing, memory efficiency, stream transducer

1. Introduction

Many diverse sources generate continuous data streams, including sensors, IoT devices, user interaction monitoring systems, and Change Data Capture (CDC) mechanisms in databases. Such streams are used for streaming data analytics and a wide range of applications, including real-time recommendation systems, traffic monitoring, and operational monitoring. Managed streaming services have also become an important building block of the cloud offerings provided by major cloud vendors, while streaming technologies are equally relevant for on-premise information systems [1].

A core constraint of stream processing is that input data arrives incrementally rather than all at once, as in batch processing. Many real-time scenarios require output data to be generated with low latency, significantly limiting the time available for computation, including potentially expensive memory operations. Furthermore, components of stream-processing pipelines often operate under limited computational resources, making computationally and memory-intensive operations undesirable. These factors highlight the importance of memory-efficient streaming algorithms.

In this work, we focus on core streaming algorithms for the JSON data format. JSON [2, 3] has become a widely adopted text-based format for data interchange. Due to its hierarchical structure, processing JSON data introduces challenges that differ from those encountered with flat relational data [4]. The widespread use of JSON to represent data streams motivates the development of efficient streaming algorithms. Another use case where memory-efficient JSON stream processing is relevant is the transformation of large JSON inputs that cannot be fully stored in memory or where minimizing memory overhead during processing is important.

Related work. Most existing JSON transformers work in a traditional tree-based manner, performing the required operations on JSON data loaded entirely into memory. This approach may lead to unnecessary memory consumption. Such transformers are either commercial solutions, which often provide graphical user interfaces for specifying and executing transformations [5, 6], or open-source tools that are primarily based on command-line interfaces or embedded domain-specific languages

ITAT'26: Information Technologies – Applications and Theory, September 25–29, 2026, Biele Karpaty, Slovakia

*Corresponding author.

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

0009-0006-6594-2594 (J. Kostičová)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-09

CEUR-WS.org/Vol-4323/paper8.pdf

[7, 8, 9]. Currently, there is no single widely adopted transformation language for JSON; therefore, each transformer uses its own representation of transformation specifications. Although the JSONiq language and its implementations [10] are currently getting popular, its primary focus is querying rather than transformations, and due to its Turing-completeness it would have to be constrained significantly to allow for memory-complexity analysis.

Some tools operate on streaming JSON, but these are mostly limited to parsers and streaming query evaluators [11, 12, 13, 14] that provide little functionality for transforming the input JSON. A few libraries support streaming JSON transformations. Json-stream-transform [15] and fs2-data [16] are Scala-based libraries that primarily focus on transformations applied to selected locations within the JSON stream and integrate transformation logic into the Scala API. The Node.js library stream-json [17] is more flexible; however, it is a low-level toolkit that requires users to specify transformations programmatically. All these solutions provide only built-in support for simple single-path transformations, and although some of them make claims about a small memory footprint, none of them provide a formal characterization or systematic experimental evaluation of the memory complexity of their transformations.

Related data formats, such as XML, have dedicated languages and theoretical foundations for streaming transformations [18, 19]. However, these approaches are based on the XML data model and therefore cannot be directly applied to JSON without adaptation.

In this work, we present a new declarative transformation language inspired by existing approaches, and a formal model, called the JSON Stream Transducer, that together form a convenient framework for defining the streaming semantics of the specified transformations. We use this framework to formally describe several individual transformations, analyze their memory complexity, and validate the results through a prototype implementation and experimental evaluation. A key feature of the approach is that it provides formal memory guarantees for individual transformations. This allows memory requirements to be assessed before deployment and provides a basis for adapting the streaming pipeline accordingly, for example, by choosing a less memory-intensive transformation when it provides the desired result. The specification language and formal model are independent of any particular programming language, allowing the approach to be implemented in different programming environments. Unlike existing stream libraries, our solution also directly handles selected two-path transformations that require streaming synchronization of path evaluation. The composition of multiple transformations within a single specification is outside the scope of this work, as it introduces additional semantic and dependency issues.

2. Transformation framework

We present the concepts needed to describe the streaming algorithms formally - the basic concepts related to JSON format and JSONPath query language, the transformation specification language and the formal model JSON Stream Transducer that will be later used to capture the semantics of transformations specified in the proposed language.

2.1. JSON and JSONPath concepts

This section provides necessary notions and notations that are used later when defining the formal streaming algorithms for selected JSON transformations.

JSON concepts. A JSON document consists of a recursively defined root value, which is one of the following:

- an object, which is an unordered collection of key/value pairs (called *object members*),
- an array, which is an ordered collection of values (called *array elements*),
- a primitive value (number, string, `true`, `false`, or `null`).

In an object, no two members may have the same key. A simple example of JSON data is shown in Figure 1, with literal values highlighted in grey.

We label array elements by their indices to achieve a uniform structure for both objects and array elements; the technique was introduced in [4] . We omit literal values from the formal model since they are not used in the core transformation logic; however, they are handled by the implementation. We call the finite set of keys and array indices the *label alphabet*. It is required that the array indices in such an alphabet form a contiguous sequence starting from 0.

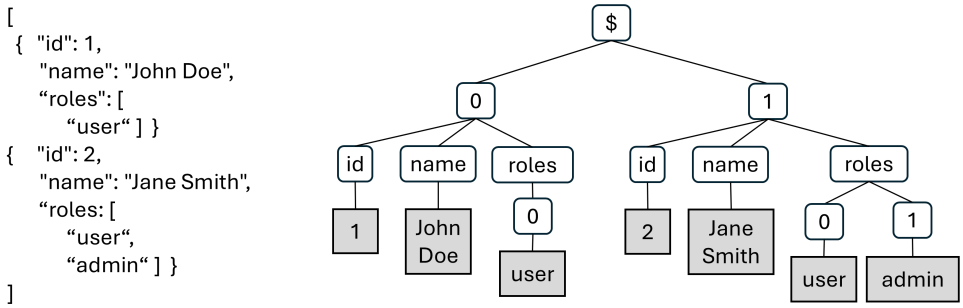


Figure 1: An example of JSON data and its tree representation

Let Λ be a label alphabet. During stream processing, a JSON document is represented as a sequence of events. For each label $\sigma \in \Lambda$, the event $start(\sigma)$ denotes the beginning of the corresponding JSON value, while the event end denotes the end of the most recently opened value. This mirrors the JSON syntax, where closing brackets do not repeat the corresponding key or array index. We denote the set of all such events over Λ by $E(\Lambda)$. Any sequence generated according to these nesting rules is called an *event stream* and we denote the set of all event streams over Λ by $ES(\Lambda)$. Objects consist of zero or more event streams labeled by object keys, whereas arrays consist of zero or more event streams labeled by consecutive array indices starting from 0.

JSONPath concepts. A standard language for querying JSON data is JSONPath [20]. The queries allow for complex navigation within tree structure as well as filtering based on predicates. Since this work focuses on streaming the core transformation logic, we limit ourselves to using simplified JSONPath queries only and adapt a path automaton from our previous work [14] for its execution. A simplified query is constructed using keys and array indices (*segments*), where keys are separated by dots and array indices are enclosed by square brackets. Each segment induces a single navigation step. We denote the label of the i-th segment of the JSONPath query $path$ by $path[i]$.

Example 1. *The JSONPath query* `$[1].roles[1]` *contains three segments. When evaluated over the JSON tree in Fig. 1, the first segment selects the child of the root node being an array element indexed by* 1*, the second segment selects child object named* $roles$*, and the last segment selects the child indexed by* 1*, so the selected value is literal string* admin.

2.2. JSON Transformation Specification

We define a new specification format, JSON Transformation Specification (JTS), which allows required transformations to be defined in a declarative way and their complexities to be analyzed. The language uses the JSON format and its syntax is based on the JSON Document Transforms (JDT) language [9]. However, unlike JDT, JTS does not employ a default merge behavior. Instead, the specification of each transformation is self-contained and contains all necessary parameters. This approach makes the specifications explicit and simplifies reasoning about transformation characteristics. In comparison with JDT, JTS provides additional transformation verbs and supports a richer set of transformation attributes.

We define the following transformation verbs: *identity*, *rename*, *remove*, *replace*, *add*, *copy*, and *move*. These transformation types were selected as they represent the most common operations in existing transformation specification languages and therefore provide a suitable basis for analyzing memory complexity. The parameters of each transformation type are represented as transformation attributes.

The relationship between transformation verbs and attributes is summarized in Table 1. The exact semantics of the transformation types is expressed by the algorithms presented in Section 3.

	identity	rename	remove	replace	add	copy	move
path	N/A	required	required	required	required	required	required
destPath	N/A	N/A	N/A	N/A	N/A	required	required
key	N/A	required	N/A	optional	optional	optional	optional
value	N/A	N/A	N/A	required	required	N/A	N/A
index	N/A	N/A	N/A	N/A	optional	optional	optional

Table 1
Required and optional parameters for each operation

Example 2. *The following copy transformation specified in JTS copies the second element of the* `roles` *array of the second record to the second position in the* `roles` *array of the first record. When applied to the sample JSON data in Fig. 1, it copies the string value* admin *from the second person to the* `roles` *array of the first person, after the* user *value.*

```
[
 {
   "copy": {
     "path": "$[1].roles[1]",
     "destPath": "$[0].roles",
     "index": 1 }
 }
]
```

We limit the current specification language to a single transformation verb per specification. Although the JTS format is designed to support multiple transformation verbs with an explicitly specified order, their composition is outside the scope of this work because it introduces additional semantic and dependency issues. While some combinations are straightforward to process (e.g., a *rename* transformation applied to independent objects or a *replace* transformation applied to independent array values or objects), other combinations may induce conflicts that need to be resolved, such as attempting to process a value that was removed by a preceding transformation or matching a path on an already modified branch. Moreover, the result may depend on the order in which transformations are applied. Supporting multiple transformation verbs therefore requires a comprehensive analysis of the dependencies and composition semantics between specific transformation types, which we leave for future work.

2.3. JSON Stream Transducer

The formal model used to describe the transformation algorithms is based on the Simple Streaming Transducer (SST) introduced in our previous work [14]. In this work, we extend SST with a queue and a set of queue-manipulation rules inspired by the queue automata [21]. A JSON Stream Transducer is a 9-tuple $T = (Q, \Lambda_{\text{in}}, \Lambda_{\text{out}}, \Gamma, \Pi, \{\rightsquigarrow, \bot\}, q_0, Z_0, R)$, where:

- Q is a finite set of states,
- Λ_{in} and Λ_{out} are alphabets of input and output labels, respectively,
- Γ is an alphabet of stack symbols,
- Π is an alphabet of queue symbols, $\& \notin \Pi$ is a special "no-dequeue" symbol,
- $\{\rightsquigarrow, \bot\}$ denote input movement symbols: reading the next event and halting, respectively,
- q_0 is the initial state, Z_0 is the initial stack symbol,
- R is a finite set of rules: $Q \times E(\Lambda_{\text{in}}) \times \Gamma \times (\Pi \cup \{\&, \varepsilon\}) \to (E(\Lambda_{\text{out}}))^*(Q \times \Gamma^* \times \Pi^* \times \{\rightsquigarrow, \bot\})$.

The transducer performs a transformation of input streams from $ES(\Lambda_{in})$ to output streams from $ES(\Lambda_{out})$. Its rules induce state transition, optional input movement, optional output generation, optional stack manipulation, and optional queue manipulation. The generated output must be a valid subsequence of an event stream from $ES(\Lambda_{out})$. However, the overall output is not guaranteed to form a valid JSON stream; this must be ensured by the logic of the rule set R. Regarding the queue manipulation, the transducer by default removes the element at the head of the queue and appends a sequence of elements (possibly empty) to the queue tail. The special symbol $\&$ indicates that no element is removed from the queue in the current step and ε denotes an empty queue.

3. Formal streaming algorithms

The proposed algorithms share several common subtasks. Instead of defining each algorithm independently, we decompose the shared functionality into reusable rule sets. Transformation algorithms are then obtained by synchronizing these rule sets via state transitions. We define three memory-usage classes based on whether a rule set or the entire algorithm utilizes a stack and/or a queue: (1) the no memory class, which uses neither the stack nor the queue; (2) the stack class, which uses the stack only; and (3) the buffer class, which uses both the stack and the queue.

3.1. Shared rule sets

Each shared rule set performs a specific subtask within the transformation logic. We group rule sets according to the memory-usage classes defined above.

3.1.1. No memory class

Both stack and queue alphabets are empty. Since neither the stack nor the queue is manipulated, their contents are omitted from the rules for simplicity.

Generating rules Q_{gen} read the input events and copy them to the output without modification.

$$(q_{gen}, e) \quad \rightarrow \quad e\,(q_{gen}, \rightsquigarrow)\,, \;\; e \text{ is an event}$$

3.1.2. Stack class

The stack alphabet Γ contains all integers $0 \leq i \leq n$ where n is the length of the evaluated path, plus the auxiliary symbol "-". If not stated otherwise, the following notations are used: $\sigma \in \Lambda_{in}$ is an input symbol, $j \geq 0$ is an integer, $\gamma \in \Gamma$ is a stack symbol, and $e_{in} \in \Lambda_{in}$ is an input event. Since the queue is not manipulated, its contents are omitted from the rules for simplicity.

Evaluation rules $Q_{eval}(path)$ track the progress of a JSONPath match. The stack symbol "-" represents an error state during evaluation of the current branch. After task completion, a transition is made to the synchronizing state q_{match}.

$$\begin{array}{lcll}
(q_{eval}, start(\sigma), j) & \rightarrow & start(\sigma)\,(q_{eval}, j(j+1), \rightsquigarrow)\,, & \mathit{path}[j] = \sigma \\
(q_{eval}, start(\sigma), j) & \rightarrow & start(\sigma)\,(q_{eval}, j-, \rightsquigarrow)\,, & \mathit{path}[j] \neq \sigma \\
(q_{eval}, start(\sigma), -) & \rightarrow & start(\sigma)\,(q_{eval}, --, \rightsquigarrow) & \\
(q_{eval}, end, \gamma) & \rightarrow & end\,(q_{eval}, \varepsilon, \rightsquigarrow)\,, & \gamma \in \{i \mid i > 0\} \cup \{-\} \\
(q_{eval}, start(\sigma), n-1) & \rightarrow & (q_{match}, (n-1)n, \bot)\,, & \mathit{path}[n-1] = \sigma
\end{array}$$

Subtree processing rules $Q_{genSubtree}, Q_{skipSubtree}$ traverse an entire subtree and either emit it to the output or prune it, respectively. The information about the current level within the JSON tree is kept by the stack symbol "-". The rules eventually transition to q_{gen} or $q_{endSubtree}$; an explicit synchronizing state could be introduced if required. It holds $e_{out} = e_{in}$ for generating rules, and $e_{out} = \varepsilon$ for skipping rules.

$$
\begin{array}{lcll}
(q, e_{in}, \gamma) & \rightarrow & e_{out}(q, \gamma-, \rightsquigarrow), & e_{in} \text{ is a start event} \\
(q, e_{in}, -) & \rightarrow & e_{out}(q, \varepsilon, \rightsquigarrow), & e_{in} \text{ is an end event} \\
(q_{genSubtree}, e, n) & \rightarrow & e\,(q_{endSubtree}, n, \bot)\,, & e \text{ is an end event} \\
(q_{skipSubtree}, e, n) & \rightarrow & (q_{gen}, n, \bot)\,, & e \text{ is an end event}
\end{array}
$$

Position search rules $Q_{findPos}(i)$ search for position i within the current structure. If i is the default value -1 for objects, or if i exceeds the number of elements in the current array, the position after the last element is selected. After task completion, the rules transition to the synchronizing state $q_{matchPos}$.

$$
\begin{array}{lcl}
(q_{findPos}, start(\sigma), n) & \rightarrow & start(\sigma)\,(q_{findPos}, n-, \rightsquigarrow)\,,\ \sigma \neq i \\
(q_{findPos}, start(i), n) & \rightarrow & (q_{matchPos}, n, \bot) \\
(q_{findPos}, start(\sigma), -) & \rightarrow & start(\sigma)\,(q_{findPos}, --, \rightsquigarrow) \\
(q_{findPos}, end, -) & \rightarrow & end\,(q_{findPos}, \varepsilon, \rightsquigarrow) \\
(q_{findPos}, end, n) & \rightarrow & (q_{matchPos}, n, \bot)
\end{array}
$$

3.1.3. Buffer class

The stack alphabet is the same as above, the queue alphabet Π contains exactly all input events from $E(\Lambda_{in})$. If not stated otherwise, we use the following notations: $\sigma \in \Lambda_{in}$ is an input symbol, $j \geq 0$ is an integer, $\gamma \in \Gamma$ is a stack symbol, and $e_{in} \in \Lambda_{in}$ is an input event.

Subtree enqueueing rules $Q_{meminSubtree}, Q_{meminSkipSubtree}$ work in the same way as subtree processing rules, but along with traversing the current subtree, they also temporarily store events in the queue for later emission. They eventually transition to the synchronizing state $q_{endSubtree}$. It holds $e_{out} = e_{in}$ for generating rules, and $e_{out} = \varepsilon$ for skipping rules.

$$
\begin{array}{lcll}
(q, e_{in}, \gamma, \&) & \rightarrow & e_{out}(q, \gamma-, e_{in}, \rightsquigarrow), & e_{in} \text{ is a start event} \\
(q, e_{in}, -, \&) & \rightarrow & e_{out}(q, \varepsilon, e_{in}, \rightsquigarrow), & e_{in} \text{ is an end event} \\
(q, e_{in}, n, \&) & \rightarrow & e_{out}\,(q_{endSubtree}, n, e_{in}, \bot)\,, & e_{in} \text{ is an end event}
\end{array}
$$

Free enqueueing rules $Q_{meminSkip}$ store events in the memory. A transition to a different state is managed within the synchronization rules (see next section).

$$
\begin{array}{lcl}
(q, start(\sigma), \gamma, \&) & \rightarrow & (q, \gamma, start(\sigma), \rightsquigarrow) \\
(q, end, \gamma, \&) & \rightarrow & (q, \gamma, end, \rightsquigarrow)
\end{array}
$$

Dequeueing rules Q_{memout} flush the contents of the queue to the output stream one event at a time. After task completion, the next state is q_{gen}.

$$
\begin{array}{lcl}
(q_{memout}, e_{in}, \gamma, \pi) & \rightarrow & \pi(q_{memout}, \gamma, \varepsilon, \bot) \\
(q_{memout}, e_{in}, \gamma, \varepsilon) & \rightarrow & (q_{gen}, \gamma, \varepsilon, \bot)
\end{array}
$$

3.2. Transformations

We define a complete algorithm for each transformation that synchronizes the use of shared rule sets. As in the case of the shared rules, we group rule sets according to the defined memory-usage classes. A transformation algorithm belonging to a given class may only use rule sets from the same class or from a lower memory-usage class. The stack and queue alphabets are the same as those defined for the corresponding rule sets.

3.2.1. No memory class

Identity transformation: The initial state is q_{gen} and only the generating rules are used. Memory complexity is constant, since neither a stack nor a queue is used.

3.2.2. Stack class

Transformations in this class are parameterized by a fixed attribute *path*. They start in $Q_{eval}(path)$ and, upon reaching q_{match}, the required action is performed. Afterward, control transitions to Q_{gen}, which copies the remainder of the input unchanged. The initial stack symbol is "0" and it represents the initial state of the path evaluation. Note that path evaluation cannot be performed in constant memory: the evaluation proceeds along a particular branch of the JSON tree, and since the depth of a JSON tree can be arbitrarily large, a finite-state machine would require an infinite number of states to perform correct matching.

Rename transformation: Upon reaching q_{match}, the transducer replaces the current key with the specified one (denoted as key).

$$(q_{match}, start(\sigma), n) \rightarrow start(key)(q_{gen}, n, \rightsquigarrow)$$

Remove transformation: Upon reaching q_{match}, the transducer transitions to $Q_{skipSubtree}$.

$$(q_{match}, start(\sigma), n) \quad \rightarrow \quad (q_{skipSubtree}, n, \rightsquigarrow)$$

Replace transformation: Upon reaching q_{match}, the transducer replaces the current subtree with the specified $value$ and optionally renames the key.

$$\begin{array}{lll} (q_{match}, start(\sigma), n) & \rightarrow & start(\sigma) value(q_{skipSubtree}, n, \rightsquigarrow), \quad key = NULL \\ & \rightarrow & start(key)\ value(q_{skipSubtree}, n, \rightsquigarrow), \quad key \neq NULL \end{array}$$

Add transformation: Upon reaching q_{match}, the transducer finds the exact insertion position via $Q_{findPos}(i)$, where i corresponds to the value of the specified *index* (or -1 if unspecified). It then outputs either key or index (depending on whether the parent context is an object or an array, respectively) followed by the specified value. If a new element is inserted into an array, the indices of all subsequent elements need to be incremented by one. This operation is performed by the state q_{inc} and after its completion the transducer transitions to q_{gen}. We do not list the corresponding incrementing rules here. They are required in the formal model because array elements are explicitly indexed; however, they are not needed in the implementation, where array indices are determined implicitly by the position. For the same reason, the implementation does not need to know the array size at this point, which is denoted by $size$ in the rules.

$$\begin{array}{lll} (q_{match}, start(\sigma), n) & \rightarrow & start(\sigma)\,(q_{findPos(index)}, 0, \rightsquigarrow), \quad index \neq NULL \\ & \rightarrow & start(\sigma)\,(q_{findPos(-1)}, 0, \rightsquigarrow), \quad index = NULL \\ (q_{matchPos}, start(i), n) & \rightarrow & start(i)\ value\ end\ start(i+1)\,(q_{inc}, n-, \rightsquigarrow) \\ (q_{matchPos}, end, n) & \rightarrow & start(key)\ value\ end\,(q_{gen}, n, \perp), \quad key \neq NULL \\ & \rightarrow & start(size)\ value\ end\,(q_{gen}, n, \perp), \quad key = NULL \end{array}$$

Stack-class transformations utilize a stack to track input nesting depth and evaluate paths, executing exactly one push per start event and one pop per end event in all states except q_{gen}. The state q_{gen} is used only at the final stage of the transformation. Therefore, the auxiliary space complexity is $O(d)$, where d denotes the depth of the input tree. Since JSON parsers often enforce a maximum nesting depth for security reasons, d is typically bounded by a constant in real-world applications.

3.2.3. Buffer class

Transformations in this class are parameterized by the fixed attributes *path* and *destPath*, which identify the source subtree and the destination position, respectively. The transformation copies or moves the source subtree to the destination position. If the destination structure is an array, the index of the inserted element can be specified. If it is an object, a new key can be specified. The two paths must be branch-disjoint; that is, neither path may be a prefix of the other. These transformations manipulate

both the stack and the queue. The stack is used to evaluate *path* and *destPath*, as well as to track the current nesting level in the input stream for subtasks where this information is required (e.g., buffering the source subtree).

The corresponding transducer is the synchronous composition of two child transducers: the source transducer evaluating *path* and the destination transducer evaluating *destPath*. Its states are pairs consisting of the source and destination states, and its stack contains pairs of source and destination stack symbols. The transformation starts in (q_{eval}, q_{eval}). The synchronization rules are applied whenever no transition is defined in the shared rule sets for at least one of the child transducers. Otherwise, both transducers transition simultaneously according to the shared rule sets, each updating its own state and stack. Only one transducer is allowed to manipulate memory at a time, and the input event is copied to the output only if both transitions emit the event (in that case it is always the same event). The copy and move transformations differ only in the processing of the matched source subtree.

Copy transformation: The algorithm distinguishes two cases based on which path is matched first. If the source path is matched first (source-first scenario), the source subtree is buffered immediately while the destination transducer continues evaluating the parameter *destPath*. Once the destination position is reached, the buffered subtree is emitted. If the destination path is matched first (destination-first scenario), the algorithm buffers all intervening events until the source subtree is matched. The buffered events are subsequently replayed after the source subtree has been emitted. For simplicity, we do not list all synchronization rules here. However, state transitions for both scenarios are shown in Fig. 2; the states that induce synchronization transitions are highlighted in grey.

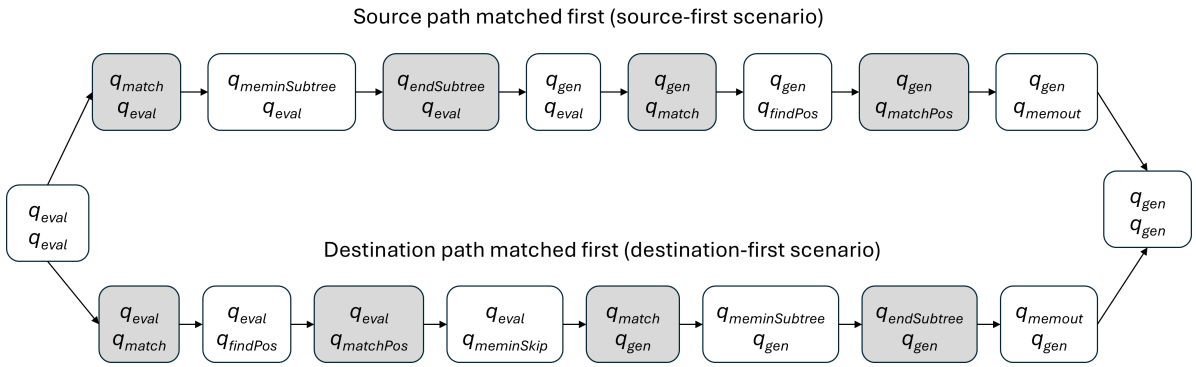


Figure 2: State transitions for the copy transformation: source-first and destination-first scenarios

Move transformation: The move transformation works in a way very similar to the copy transformation. In the source-first scenario, the source transducer does not generate the source subtree to the output stream while buffering it, meaning it operates in state $q_{meminSkipSubtree}$ instead of $q_{meminSubtree}$. In the destination-first scenario, the source subtree is not stored in memory; thus, the source transducer operates in $q_{genSubtree}$ instead of $q_{meminSubtree}$.

Transformations in this class use both the stack and the queue. The stack requires $O(d)$ space, where d is the depth of the JSON tree, as in the case of the stack class. The queue is active only in states $q_{meminSubtree}$, $q_{meminSkipSubtree}$, and $q_{meminSkip}$. The memory complexity depends on the case encountered:

- **Source path matched first.** The source subtree of size t_s is buffered, resulting in memory complexity $O(d) + O(t_s) = O(t_s)$.
- **Destination path matched first.** The data between the destination and source, denoted by $|dest, src|$, must be buffered. For move transformations, this yields $O(d) + O(|dest, src|) = O(|dest, src|)$. For copy transformations, the source subtree must also be buffered, giving $O(d) + O(t_s) + O(|dest, src|) = O(t_s + |dest, src|)$.

In the worst case, the auxiliary memory may grow to $O(n)$, where n is the size of the input JSON stream.

Example 3. *An example of a copy transformation in the destination-first scenario is shown in Fig. 3. It operates on the sample JSON data and JTS introduced earlier and follows the processing pipeline for the destination-first scenario. The data stored in memory are highlighted in grey and they consist of the events between the destination path match and the source path match, together with the subtree at the source path match.*

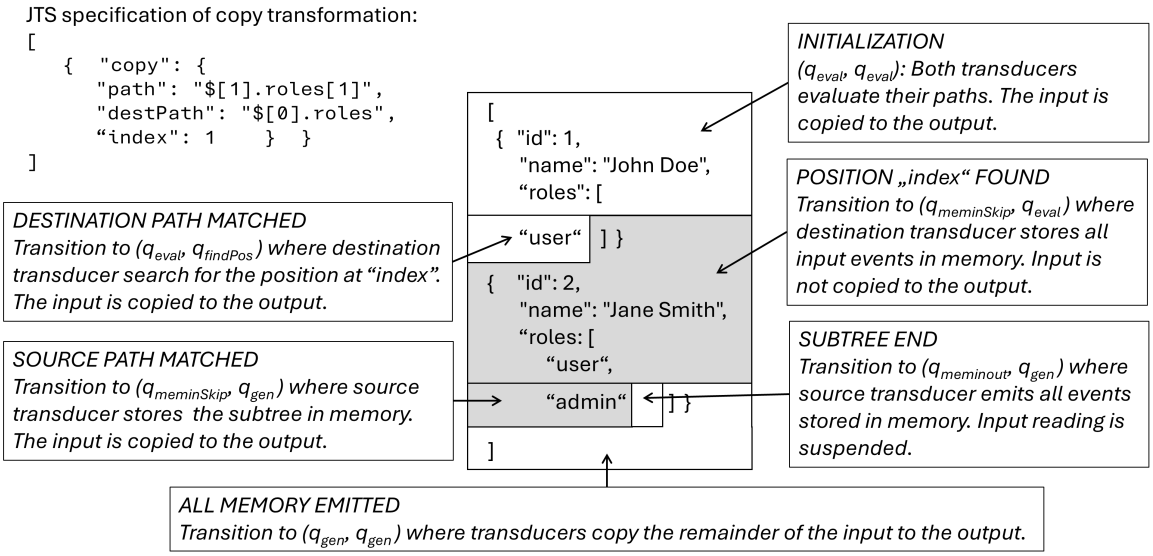


Figure 3: An example of specifying and processing a destination-first copy transformation

4. Prototype streaming transformer

The prototype implementation was developed in Java, and the Jackson library [11] was used (1) to parse input data in streaming mode, (2) to parse the specification file in memory, and (3) to generate output data in streaming mode. The object model follows the formal algorithms introduced above and contains a hierarchy of three transducer classes: *Transducer* (no memory model), *StackTransducer*, and *BufferTransducer*. A separate class *BufferSyncTransducer* represents an orchestration layer that internally coordinates two instances of *BufferTransducer* (one for the source path and the other for the destination path). Based on the specification file, the corresponding transducer is instantiated and executed. The implementation of transducer states follows the State design pattern, where each state is represented as a separate class.

The implementation was adapted to work with Jackson parsing events. In particular, it is necessary to track the sizes of the arrays encountered along the current branch. For this purpose, a separate index stack is used. Since its size is bounded by the depth of the input stream, it does not affect the asymptotic space complexity.

The prototype streaming transformer, including the evaluation code and input datasets, is available at https://github.com/jkosticova/JSON_Stream_Transformer.

4.1. Evaluation

We evaluated the proposed algorithms by measuring memory allocations for each transformation across different input sizes and specifications. We used the Java Virtual Machine (JVM) thread allocation counter exposed via `ThreadMXBean.getThreadAllocatedBytes()`. To make this method work, we refactored the code so that all heap allocations performed by our code occur during initialization, before the core transformation process begins. Since the initialization cost depends solely on the input specification, it contributes only a constant amount to the measured allocations.

We performed 500 warm-up runs and 500 measurement runs, and reported the arithmetic mean of the measured values. In addition, the JVM was configured with a fixed heap size and consistent garbage collection settings to improve reproducibility across executions. We also included a baseline algorithm that only parses a JSON file in streaming mode and immediately generates a one-to-one output. This scenario allows us to isolate memory allocations of the core transformation logic from those incurred by the Jackson library.

4.2. No memory class and stack class

The goal of this evaluation scenario is to verify that a transformer run exhibits constant additional memory usage after subtracting allocations attributed to the Jackson library. This assumption follows from the fact that transducer initialization is performed only once at the beginning of the transformation. For stack-class transformations, stack size must also be taken into account. Based on the formal analysis, the stack size is $O(d)$ with respect to the input depth. However, since the input datasets have a fixed depth, it can be treated as constant.

We prepared the test data using the JSON Generator tool [22]. By adjusting the repetition parameter of top-level array elements, we got files of sizes 118 KB, 2467 KB, 9867 KB and 24667 KB, each of them with a nesting depth of four levels. For each transformation, a dedicated specification file was prepared. The results are shown in Table 2. Each dataset size is associated with two metrics: total memory allocations (including Jackson-based JSON parsing and generation, which depend on input size), and the difference relative to the baseline. This second metric isolates memory usage of the transformation logic, including transducer initialization and stack operations.

The results show that for each transformation type, the memory allocated by the transformation core process (the *Diff* column) remains low and exhibits only minor variation across input datasets of different sizes, supporting the theoretical expectation.

4.3. Buffer class

We first analyze memory allocations across different input sizes while keeping the amount of buffered data constant. We reuse the datasets from the previous experiment and evaluate two specification variants for both copy and move transformations: one for the source-first scenario, and the other for the destination-first scenario. As expected from the theoretical analysis, the additional memory allocations remain constant across input sizes, since the amount of buffered data does not change. However, destination-first specifications consistently require more memory than source-first specifications due to a larger portion of the input being buffered (Table 2).

	Dataset 24667KB		Dataset 9867KB		Dataset 2467KB		Dataset 118KB	
Transformation	Total	Diff	Total	Diff	Total	Diff	Total	Diff
baseline	14998256	0	6000176	0	1501136	0	73296	0
identity	15005176	6920	6007096	6920	1508056	6920	80216	6920
rename	15007824	9568	6009744	9568	1510704	9568	82864	9568
remove	15007184	8928	6009104	8928	1510064	8928	82224	8928
replace	15008764	10508	6010684	10508	1511644	10508	83804	10508
add	15009012	10756	6010932	10756	1511892	10756	84052	10756
copy (src-first)	15008571	10315	6010491	10315	1511451	10315	83618	10322
move (src-first)	15008571	10315	6010491	10315	1511451	10315	83619	10323
copy (dest-first)	15116352	118096	6118272	118096	1619232	118096	191526	118230
move (dest-first)	15116279	118023	6118200	118024	1619160	118024	191454	118158

Table 2
Memory allocations of stack-class transformations and buffer-class transformations for different input sizes (in bytes)

We further analyze the destination-first case using a fixed large input dataset (24667 KB) and multiple specifications that vary the number of elements between the destination and source matching positions (denoted by $|d, s|$). For the source-first case, we modify the 118 KB dataset by scaling the size of the matched source subtree to values ranging from 130 to 15000 elements. The results are summarized in Tables 3 and 4; only the differences with respect to the baseline algorithm are shown. In both scenarios, memory allocation increases linearly with the amount of data stored in the buffer, confirming the theoretical results. As expected, the allocation cost of initialization is largely independent of the input size.

	$\|d, s\| = 22400$	$\|d, s\| = 11200$	$\|d, s\| = 5600$	$\|d, s\| = 2800$	$\|d, s\| = 1400$	$\|d, s\| = 130$
Transf.	Diff	Diff	Diff	Diff	Diff	Diff
copy	18745030	9377334	4693535	2351598	1180190	118094
move	18744950	9377270	4693462	2351422	1180118	118022

Table 3
Memory allocations of buffer-class transformations for variable size of buffered data for the destination-first scenario (in bytes)

	$t_s = 15000$	$t_s = 10000$	$t_s = 7500$	$t_s = 5000$	$t_s = 2500$	$t_s = 1000$	$t_s = 130$
Transf.	Diff	Diff	Diff	Diff	Diff	Diff	Diff
copy	1810359	1210359	910359	610359	310421	130402	26011
move	1810340	1210340	910340	610340	310370	130372	25945

Table 4
Memory allocations of buffer-class transformations for variable size of buffered data for the source-first scenario (in bytes)

5. Conclusion and future work

In this paper, we introduced a specification language and a formal transducer model to analyze the memory complexity of streaming JSON transformations. Using the defined transducer, we formally described streaming algorithms for common structural transformations on JSON objects and arrays, including selected two-path transformations that impose additional synchronization complexity on the streaming process. We analyzed the memory complexity of these algorithms and categorized them into discrete memory complexity classes. Our empirical evaluation of the prototype Java implementation validated the theoretical bounds: no-memory transformations require constant memory, while stack-class transformations require $O(d)$ memory, where d is the depth of the input data; buffer-class operations require memory proportional to the size of the buffered data. These results provide a basis for assessing whether the memory requirements of a transformation are acceptable for a given streaming environment before deployment. If a required transformation has high memory requirements, the analysis can guide decisions about resource allocation, pipeline architecture, or alternative transformations.

The current work focuses on specifying and analyzing individual transformations. Supporting multiple transformations within a single specification is left for future work, as their composition introduces additional semantic and complexity challenges that require a separate analysis. It would also be beneficial to extend the proposed framework to support more complex JSONPath queries. For example, incorporating the wildcard operator would allow all array elements to be processed using simple transformations such as *rename* or *replace*. However, the semantics become less clear when applying *copy* or *move* transformations, as the wildcard may match multiple source or destination paths. Another possible direction for future work is to extend the transformer with schema analysis. Such a

feature could help refine the memory requirements, as a schema provides additional information about the maximum size of a specific JSON subtree or fragment.

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT-5.6, Claude Sonnet 5, and Gemini 3.6 Flash to check grammar, spelling, and improve sentence clarity. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. Fragkoulis, P. Carbone, V. Kalavri, A. Katsifodimos, A survey on the evolution of stream processing systems, The VLDB Journal 33 (2024) 507–541.

[2] ECMA-404: The JSON Data Interchange Syntax, Ecma International, 2017. https://ecma-international.org/publications-and-standards/standards/ecma-404/.

[3] T. Bray, The JavaScript Object Notation (JSON) Data Interchange Format, IETF, 2017. RFC 8259.

[4] P. Bourhis, J. L. Reutter, F. Suárez, D. Vrgoč, JSON: Data model, Query languages and Schema specification, in: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 123–135.

[5] Altova, MapForce, 2026. https://www.altova.com/mapforce.

[6] Talend, Talend Studio, 2026. https://help.qlik.com/talend/.

[7] S. Dolan, jq, 2026. https://github.com/jqlang/jq.

[8] Bazaarvoice, Jolt, 2026. https://github.com/bazaarvoice/jolt.

[9] Microsoft, JSON Document Transforms, 2026. https://github.com/microsoft/json-document-transforms.

[10] D. Graur, R. Röthlisberger, A. Jenny, G. Fourny, F. Drozdowski, C. Konigsmark, I. Müller, G. Alonso, Addressing the Nested Data Processing Gap: JSONiq Queries on Snowflake Through Snowpark, in: 2024 IEEE 40th International Conference on Data Engineering (ICDE), 2024, pp. 5252–5265.

[11] FasterXML, Jackson, 2026. https://github.com/FasterXML/jackson.

[12] L. Wang, JsonSurfer, 2026. https://github.com/wanglingsong/JsonSurfer.

[13] 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, p. 79–92.

[14] J. Kostičová, Low-Latency Streaming Evaluation of JSONPath Queries, in: ITAT, 2024, pp. 43–51.

[15] S. Berner, json-stream-transform, 2026. https://github.com/sambe/json-stream-transform.

[16] L. Satabin, Y. Heiber, fs2-data, 2026. https://github.com/gnieh/fs2-data.

[17] E. Lazutkin, stream-json: Streaming JSON processing library for Node.js, 2026. https://github.com/uhop/stream-json.

[18] O. Becker, Extended SAX filter processing with STX, in: Proceedings of the Extreme Markup Languages® 2003 Conference, 4-8 August 2003, Montréal, Quebec, Canada, 2003.

[19] J. Dvořáková, Automatic Streaming Processing of XSLT Transformations Based on Tree Transducers, Informatica 32 (2008) 373–382.

[20] S. Gössner, G. Normington, C. Bormann, JSONPath: Query Expressions for JSON, IETF, 2024. RFC 9535.

[21] D. C. Kozen, Automata and Computability, Undergraduate Texts in Computer Science, Springer, 1997.

[22] JSON Generator, 2026. https://app.json-generator.com.