Streaming Validation of JSON Documents Against Schemas

Alexis Le Glaunec
Rice University
Houston, USA
alexis.leglaunec@rice.edu

Angela W. Li
Rice University
Houston, USA
awl@rice.edu

Konstantinos Mamouras
Rice University
Houston, USA
mamouras@rice.edu

ABSTRACT

JSON is a popular data format for storing semi-structured data. We investigate the computational problem of JSON validation, which is the task of checking whether a JSON document adheres to a given schema. While there are several existing tools that support JSON validation, they implement offline algorithms that require loading the entire document in memory and creating the full parse tree before performing validation. This offline approach is constrained by the available system memory and is inappropriate when the data is presented as a stream. We propose an approach for performing streaming JSON validation that relies on a new class of pushdown automata that can process JSON documents in an online fashion. Our experimental results show that our approach uses substantially less memory and is faster than state-of-the-art tools.

PVLDB Reference Format:
Alexis Le Glaunec, Angela W. Li, and Konstantinos Mamouras. Streaming Validation of JSON Documents Against Schemas. PVLDB, 19(3): 509 - 522, 2025.
doi:10.14778/3778092.3778109

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://doi.org/10.5281/zenodo.17665294.

1 INTRODUCTION

JavaScript Object Notation (JSON) [26] is a data format that is widely used for exchanging and storing data. Among its main use cases are Web APIs, where JSON serves as the exchange format for requests and responses; configuration files, where it stores settings for software applications in a clean and easy-to-read format; and data storage with NoSQL databases like MongoDB [39] or RxDB [46]. More recently, relational databases such as Oracle [43] and MySQL [41] have added support for storing and querying JSON documents. Additionally, JSON is used in the development of mobile applications, IoT systems [54], and microservices. This versatility makes JSON ubiquitous in modern software ecosystems.

JSON is a flexible data format, but it lacks features that can ensure data consistency (e.g., type restrictions). This flexibility, although useful in many cases, can also lead to errors and misinterpretations when different systems exchange data. JSON Schema [27] is a schema language that is used to enforce data consistency over JSON documents. A JSON schema defines constraints over data types, values, required keys, etc. Common use cases of JSON Schema include structural validation (i.e., checking that a document conforms to the required structure), the definition of interfaces for communication with microservices (Postman [45]), and schema-aware serialization (JSON BinPack [52]) of data for efficient transmission.

In this work, we focus on the data validation problem. An interesting application of data validation can be found in both relational databases like Oracle [43] that extend the SQL query language to support schema validation, and in NoSQL databases like MongoDB [39] where JSON schemas are used to give some structure to the documents. A very popular tool for data validation is AJV [1], which has extensive support for the JSON Schema language and performs well for small JSON documents. There are many other JSON schema validators, and most of them are based on the same recursive algorithm. First, the algorithm parses the JSON document and JSON schema into abstract syntax trees (ASTs). Then, it traverses both ASTs to check if the document conforms to the schema. In the context of schema validation for databases (both for relational [41, 43] and NoSQL [39, 46] databases), the document AST can be too large to fit in main memory, thus restricting schema validation to only small files. This motivates our work on designing a streaming validation algorithm that ensures small memory footprint when handling large JSON documents.

A natural approach for streaming schema validation over hierachical data is to use pushdown automata, in particular models such as visibly pushdown automata (VPAs) [3]. One feature of JSON documents that make them a poor fit for VPAs is that the fields of objects are considered to be *unordered* and can therefore appear in any order. For example, if an automaton accepts `{"x": 1, "y": 2, "z": 3}`, then it should also accept the 5 other documents in which the fields are permuted. Encoding the set of all permutations using VPAs results in an exponential blowup in the size of the state space. For instance, the schema $\mathrm{Obj}(x_1 : \mathrm{Num}, \ldots, x_n : \mathrm{Num})$, which accepts JSON objects with n keys that are associated with numerical values, can be represented as a VPA that requires $\Omega(n!)$ states. This exponential blowup *always* arises when objects are used and therefore VPAs are inherently unsuitable for JSON data.

To overcome the limitations of classical pushdown automata and VPAs for JSON data, we propose here a more succinct variant that specifically addresses the issue of the possible reordering of fields in objects. One crucial idea is that we can extend the pushdown stack with more information regarding the set m of keys that appear in a JSON object. We use a succinct data structure for this purpose, namely a bit vector. Every time we see a string that has the role of a key, we update the set m at the top of the stack. When we see the closing brace of an object, we can then use this set m of keys to check against a guard that validates whether m is acceptable, i.e., all the required keys appear in the object. For a given schema,

Proceedings of the VLDB Endowment, Vol. 19, No. 3 ISSN 2150-8097.
doi:10.14778/3778092.3778109

we only need to consider a finite number of keys (i.e., those that appear in the schema) and for this reason the size of the set m is bounded. We call the class of automata that employ this feature JSON automata (JAs). The automata can be nondeterministic (NJAs) or deterministic (DJAs). Using this model of automata, the schema $\text{Obj}(x_1 : \text{Num}, \ldots, x_n : \text{Num})$ can be represented as a DJA with only 5 states (hence independent of the number n of keys) and a set m of keys of size n.

Main Contributions. The main contributions of this paper are the following:

(1) We introduce the model of JSON automata (JAs), which are appropriate for succinctly encoding JSON schemas.
(2) We propose a streaming algorithm for the validation of JSON documents against schemas. The algorithm is based on *deterministic JSON automata* (DJAs). The memory footprint of our validation algorithm is independent of the size of the input JSON document (assuming bounded height).
(3) We have implemented a schema validation tool based on our proposed algorithm. Experiments show that our tool has a substantially smaller memory footprint and higher throughput than state-of-the-art tools on practical workloads.

2 PRELIMINARIES

In this section, we provide a formal definition of JSON values, we describe JSON Schema [27], and we present our schema language, which is a subset of JSON Schema that covers its main features.

Informally, a JSON value can be viewed as a tree, where arrays and objects are rooted at internal nodes, whereas constant values, numbers, and strings are leaves of the tree. More formally, a ***JSON value*** is given by the grammar

$v, v_i ::=$	`null` | `false` | `true` |	[constant values]
	$n \mid s \mid$	[JSON number / string]
	`[` v_1 `,` $\ldots$ `,` v_n `]` |	[array]
	`{` k_1 `:` v_1 `,` $\ldots$ `,` k_n `:` v_n `}`	[object]

where n is a JSON number (as defined in the JSON standard), s is a JSON string, $k_1, \ldots, k_n$ are pairwise distinct strings (i.e., $i \neq j$ implies $k_i \neq k_j$), and $v_1, \ldots v_n$ are JSON values.

For an object `{` k_1 `:` v_1 `,` $\ldots$ `,` k_n `:` v_n `}`, the strings $k_1, \ldots, k_n$ are called *keys*. We call the values $v_1, \ldots, v_n$ of an array `[` v_1 `,` $\ldots$ `,` v_n `]` the *elements* of the array. The values $v_1, \ldots, v_n$ of an object or an array can be objects or arrays themselves.

Example 1. An example of a JSON value is given below for an array of three elements representing 2D points. Each element has two keys, `"x"` and `"y"`, to represent the point position. The values associated with `"x"` and `"y"` are JSON numbers.

```
[{"x":1.0,"y":1.0}, {"x": 2.0,"y":1.0}, {"x":5.0,"y":1.5}]
```

JSON Schema. JSON Schema [27] is a language used to define the structure of JSON documents. It uses JSON syntax to describe type and value domain constraints over JSON documents. The constraints that the language supports include the following:

– *String constraints*: value constraints on string length (min and max) and patterns (i.e., string matches a regular expression r).

```
{ "type": "array",
  "items": { "type": "object",
    "properties": {
      "x": { "type": "number" }, "y": { "type": "number" } },
    "required": ["x", "y"] } }
```

Figure 1: JSON schema representing an array of 2D points.

$s, s_i ::=$	Anything |	[any JSON value]
	Nothing |	[no value]
	Null |	[value `null`]
	Bool |	[Boolean value]
	Num |	[JSON number]
	Str |	[JSON string]
	$\text{Arr}(s)$ |	[array]
	$\text{Obj}(k_1 : s_1, \ldots, k_n : s_n; req; s_o)$ |	[object]
	$\text{Or}(s_1, \ldots, s_n)$ |	[union]
	$\text{And}(s_1, \ldots, s_n)$ |	[intersection]
	$\text{Var}(i)$	[variable]

Figure 2: Syntax of the JSON Schema Language (*JSL*).

– *Numerical constraints*: numbers have range (min and max) and multiplicity constraints.
– *Array constraints*: constraints over the array length, and the schema of the array elements.
– *Object constraints*: constraints over the keys that are required (i.e., must be present), and a schema for each key.
– *Type constraints*: they restrict JSON values to be a certain data type (object, array, string, number, integer, boolean, null).
– *Boolean operators*: combination of schemas to refine constraints with Boolean operators (`AND`, `OR`, `XOR`, `NOT`).

The list above is not exhaustive, but includes the main constraints that are used in practice. The JSON schema of Fig. 1 accepts arrays of 2D points (e.g., the value of Example 1).

A Core Schema Language. We introduce the *JSON Schema Language* (*JSL*), a formalism that captures the core features of JSON Schema, including type constraints, Boolean operators, required keys, and recursion. The syntax of schemas is presented in Fig. 2.

In *JSL*, an object schema $\text{Obj}(k_1 : s_1, \ldots, k_n : s_n;\ req;\ s_o)$ constrains JSON objects by restricting both their keys (also called *property names*) and values. Each k_i is a JSON string. The component *req* indicates which of these keys are required (formally, it is a subset of $\{k_1, \ldots, k_n\}$). The schema s_o constrains all other values that are not associated with a key $k_1, \ldots, k_n$, and corresponds to the keyword `additionalProperties` in JSON Schema. An array schema $\text{Arr}(s)$ constrains the set of valid arrays to those where each array element satisfies the subschema s.

We specify a *recursive schema* E using a system of equations: $\text{Var}(0) := s_0, \text{Var}(1) := s_1, \ldots, \text{Var}(n-1) := s_{n-1}$, where each schema s_i contains only variables among $\text{Var}(0), \ldots, \text{Var}(n-1)$.

Notation: For the rest of the paper, we specify schemas in *JSL*. The *JSL* schema for an array of 2D points (e.g., the array of Fig. 1) is $s_1 = \text{Obj}($`"x"` $: \text{Num},$ `"y"` $: \text{Num}; \text{Nothing})$ where the schema following the semicolon, namely $s_o = \text{Nothing}$, constrains the

$$
\begin{aligned}
[\![\mathsf{Anything}]\!]\ \vartheta &= \{v \mid v \text{ is a JSON value}\}\\
[\![\mathsf{Nothing}]\!]\ \vartheta &= \emptyset\\
[\![\mathsf{Null}]\!]\ \vartheta &= \{\texttt{null}\}\\
[\![\mathsf{Bool}]\!]\ \vartheta &= \{\texttt{true}, \texttt{false}\}\\
[\![\mathsf{Num}]\!]\ \vartheta &= \{n \mid n \text{ is a JSON number}\}\\
[\![\mathsf{Str}]\!]\ \vartheta &= \{s \mid s \text{ is a JSON string}\}\\
[\![\mathsf{Arr}(s)]\!]\ \vartheta &= \{\texttt{[}\, x_1\, \texttt{,}\, \ldots\, \texttt{,}\, x_n\, \texttt{]} \mid x_i \in [\![s]\!]\ \vartheta \text{ for every } i\}\\
[\![\mathsf{Obj}(m, req, s_o)]\!]\ \vartheta &= \{\texttt{\{}\, k_1\, \texttt{:}\, v_1\, \texttt{,}\, \ldots\, \texttt{,}\, k_n\, \texttt{:}\, v_n\, \texttt{\}} \mid\\
&\qquad \text{the keys in } req \text{ are among } k_1, \ldots, k_n, \text{ and}\\
&\qquad \text{for every } i = 1, \ldots, n:\\
&\qquad\quad \text{if } k_i : s \text{ is in } m, \text{ then } v_i \in [\![s]\!]\ \vartheta, \text{ and}\\
&\qquad\quad \text{if } k_i \text{ is not in } m, \text{ then } v_i \in [\![s_o]\!]\ \vartheta\}\\
[\![\mathsf{Or}(s_1, \ldots, s_n)]\!]\ \vartheta &= \textstyle\bigcup_{i=1}^{n} [\![s_i]\!]\ \vartheta\\
[\![\mathsf{And}(s_1, \ldots, s_n)]\!]\ \vartheta &= \textstyle\bigcap_{i=1}^{n} [\![s_i]\!]\ \vartheta\\
[\![\mathsf{Var}(i)]\!]\ \vartheta &= \vartheta(i)
\end{aligned}
$$

Figure 3: Interpretation of *JSL* schemas.

values of keys other than `"x"` or `"y"`. If the component s_o is omitted, it is assumed to be Anything. Optional keys are indicated by appending a question mark to the property name. The schema s_2 = Obj(`"x"` : Num, `"y"`? : Num) imposes the following constraints on the object properties: (i) the key `"x"` is required and the corresponding value must be a number, and (ii) the value for key `"y"` (if it is present) must be a number. The JSON document `{"x": 2}` is accepted by s_2 (because the key `"y"` is optional), but rejected by s_1 (because the required key `"y"` is missing).

Denotational semantics of JSL. Every schema is interpreted as a set of JSON values (which is a language of linearized trees), i.e., the values that *satisfy* the schema. A *variable assignment* is a (potentially partial) function ϑ that maps a variable $\mathsf{Var}(i)$ to a set $\vartheta(i)$ of JSON values. Informally, a variable assignment specifies an interpretation for each variable. In Fig. 3, we define a semantic function that gives the interpretation $[\![s]\!]\ \vartheta$ of a schema s w.r.t. the variable assignment ϑ. Implicit in this notation is the requirement that $\vartheta(i)$ is defined for every variable $\mathsf{Var}(i)$ that appears in s. Given a recursive schema E, which consists of the equations

$$\mathsf{Var}(0) := s_0,\ \mathsf{Var}(1) := s_1,\ \ldots, \mathsf{Var}(n-1) := s_{n-1},$$

we give the standard ***least fixpoint semantics*** [55] with the following approximation sequence $(\vartheta_j)_{j\geq 0}$ of variable assignments that only needs to be defined for the variables $\mathsf{Var}(0), \ldots, \mathsf{Var}(n-1)$: $\vartheta_0(i) = \emptyset$ and $\vartheta_{j+1}(i) = [\![s_i]\!]\ \vartheta_j$ for every $i = 0, 1, \ldots, n-1$ and every integer $j \geq 0$. We write $[\![\mathsf{Var}(i)]\!]_E = \bigcup_{j\geq 0} \vartheta_j(i)$ for the least fixpoints defined by the recursive schema E.

More generally, if s is a schema whose variables are among $\mathsf{Var}(0), \ldots, \mathsf{Var}(n-1)$ and E is a recursive schema that defines all of these variables, then $[\![s]\!]_E$ is the interpretation of s assuming a least fixpoint interpretation of the variables according to E.

The Validation Problem. Let s be a *JSL* schema, E be a list of equations for all the variables of s, and d be a JSON document. The validation problem asks whether d *is valid* for the schema s w.r.t. E, i.e., if $d \in [\![s]\!]_E$. It is known that the validation problem for JSON schema is in PTIME for schemas without dynamic references [4, 44]. For the rest of the paper, we make this assumption, as dynamic references are rarely used in practice.

3 JSON AUTOMATA

In this section, we introduce our model of JSON automata (JAs), which can be nondeterministic (NJAs) or deterministic (DJAs). These automata use a stack to handle the nested tree structure of JSON. The main difference from other "pushdown" models with a stack is that JAs deal with the potential re-ordering of key-value pairs in objects using a compact data structure (a bitmap) that records the keys that appear in each object (only those that are relevant for schema validation). This feature is essential for avoiding the exponential blowup from an explicit representation of all possible re-orderings (there are $\Theta(n!)$ of them, where n is the number of keys in an object). We also introduce a special symbol, other, which never belongs to a given set K of keys. Intuitively, it stands for "any other key" not listed in K, and is used for the additional properties in a schema. For convenience, we write $K_o = K \cup \{\mathsf{other}\}$.

3.1 Nondeterministic JAs

Definition 2 (NJA). Let K be a finite set of keys. A *nondeterministic JSON automaton* (NJA) over K is a tuple $\mathcal{A} = (Q, I, C, \Delta, F)$, where Q is a finite set of (control) states, $I \subseteq Q$ is the set of *initial* states, C is a finite set of *stack symbols*, $F \subseteq Q$ is the set of *final* or *accepting* states, and Δ consists of the following collection of *transition relations*:

$$
\begin{aligned}
&\Delta_\ell : Q \to \mathcal{P}(Q \times C) \text{ for } \ell \in \{\texttt{[}, \texttt{\{}\} \qquad \Delta_{\mathsf{Key}} : Q \times K_o \to \mathcal{P}(Q)\\
&\Delta_{\texttt{]}} : Q \times C \to \mathcal{P}(Q) \qquad\qquad \Delta_i : Q \to \mathcal{P}(Q)\\
&\Delta_{\texttt{\}}} : Q \times C \to \mathcal{P}(\mathrm{Pred}(K) \times Q)
\end{aligned}
$$

where $i \in \{\texttt{null}, \texttt{false}, \texttt{true}, \texttt{JNum}, \texttt{JStr}, \texttt{:}, \texttt{,}\}$.

For each $k \in K_o$, the transition relation $\Delta_k : Q \to \mathcal{P}(Q)$ is given by $\Delta_k(q) = \Delta_{\mathsf{Key}}(q, k)$ for every $q \in Q$.

Notation: Below we define some notation for NJA transitions:

$$
\frac{(q', c) \in \Delta_{\texttt{[}}(q)}{q \to^{\texttt{[}/\mathsf{push}(c)} q'} \qquad \frac{(q', c) \in \Delta_{\texttt{\{}}(q)}{q \to^{\texttt{\{}/\mathsf{push}(c)} q'}
$$

$$
\frac{q' \in \Delta_{\texttt{]}}(q, c)}{q \to^{\texttt{]}/\mathsf{pop}(c)} q'} \qquad \frac{(\varphi, q') \in \Delta_{\texttt{\}}}(q, c)}{q \to^{\texttt{\}}/\mathsf{pop}(c,\varphi)} q'}
$$

$$
\frac{q' \in \Delta_{\mathsf{Key}}(q, k) \qquad k \in K}{q \to^{k/\mathsf{add}(k)} q'} \qquad \frac{q' \in \Delta_{\mathsf{Key}}(q, \mathsf{other})}{q \to^{\mathsf{other}} q'}
$$

$$
\frac{q' \in \Delta_i(q) \qquad i \in \{\texttt{null}, \texttt{false}, \texttt{true}, \texttt{JNum}, \texttt{JStr}, \texttt{:}, \texttt{,}\}}{q \to^{i} q'}
$$

We will use this notation when drawing NJAs.

A transition relation $\Delta : Q \to \mathcal{P}(Q)$ can be viewed as the binary relation $\{(q, q') \in Q \times Q \mid q' \in \Delta(q)\}$ on Q.

The transition relation separates the cases of arrays and objects: $\Delta_{\texttt{]}}$ handles the closing bracket of arrays, while $\Delta_{\texttt{\}}}$ handles the closing brace of objects. The latter requires special treatment, since the transition depends on the set of keys that appear in the object.

A *stack entry* is either of the form $\mathtt{Arr}(c)$ or of the form $\mathtt{Obj}(c, m)$, where $c \in C$ and $m \subseteq K$. The component m is a subset of keys, which is used to remember the set of keys that have been seen so

$$\frac{(q',c) \in \Delta_{\texttt{[}}(q)}{(q,\sigma) \to^{\texttt{[}} (q', \sigma \cdot \texttt{Arr}(c))} \qquad \frac{(q',c) \in \Delta_{\texttt{\{}}(q)}{(q,\sigma) \to^{\texttt{\{}} (q', \sigma \cdot \texttt{Obj}(c,\emptyset))}$$

$$\frac{q' \in \Delta_{\texttt{]}}(q,c)}{(q, \sigma \cdot \texttt{Arr}(c)) \to^{\texttt{]}} (q', \sigma)} \qquad \frac{(\varphi, q') \in \Delta_{\texttt{\}}}(q,c) \qquad m \models \varphi}{(q, \sigma \cdot \texttt{Obj}(c,m)) \to^{\texttt{\}}} (q', \sigma)}$$

$$\frac{q' \in \Delta_{\texttt{Key}}(q,k) \qquad k \in K}{(q, \sigma \cdot \texttt{Obj}(c,m)) \to^{k} (q', \sigma \cdot \texttt{Obj}(c, m \cup \{k\}))}$$

$$\frac{q' \in \Delta_{\texttt{Key}}(q,\text{other})}{(q, \sigma \cdot \texttt{Obj}(c,m)) \to^{\text{other}} (q', \sigma \cdot \texttt{Obj}(c,m))}$$

$$\frac{q' \in \Delta_i(q) \qquad i \in \{\texttt{null}, \texttt{false}, \texttt{true}, \texttt{JNum}, \texttt{JStr}, \texttt{:}\}}{(q,\sigma) \to^{i} (q', \sigma)}$$

$$\frac{q' \in \Delta_{\texttt{,}}(q) \qquad \sigma \neq \emptyset}{(q,\sigma) \to^{\texttt{,}} (q', \sigma)}$$

Figure 4: Operational (i.e., execution) semantics for NJAs.

far in an object. A *stack* is a sequence of stack entries, which we write as $[e_0, e_1, \ldots, e_{k-1}]$ (the entry e_{k-1} is at the top of the stack). A *configuration* is a pair (q, σ), where q is a control state and σ is a stack. We write $[\,]$ for the empty stack. A configuration is *initial* (resp., *final*) if it is of the form $(q, [\,])$, where q is an initial (resp., final) state. A final configuration must have an empty stack, which is not always required in classical VPAs [3]. The transition relation Δ of the automaton induces a transition relation $\to$ (indexed by input symbols) on configurations, which is defined in Fig. 4. The notation $m \models \varphi$ means that m *satisfies* φ. A *path* π in $\mathcal{A}$ is an alternating sequence

$$cfg_0 \to^{x_0} cfg_1 \to^{x_1} \cdots \to^{x_{n-2}} cfg_{n-1} \to^{x_{n-1}} cfg_n,$$

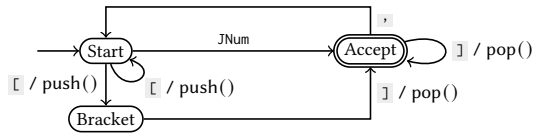
where each x_i is an input symbol and each triple $cfg_i \to^{x_i} cfg_{i+1}$ is a transition according to the definition of Fig. 4. We also define the *label* of the path as $\text{label}(\pi) = x_0 x_1 \ldots x_{n-1}$. A path is *accepting* if it starts with an initial configuration and ends with a final configuration. The automaton *accepts* an input sequence x if there exists an accepting path π with $\text{label}(\pi) = x$.

Example 3 (NJA for Nested Arrays of Numbers)**.** We will describe the NJA that accepts documents that are nested arrays of numbers. It can be represented in *JSL* by the equation $X := \text{Or}(\text{Num}, \text{Arr}(X))$. The automaton of Fig. 5a recognizes X.

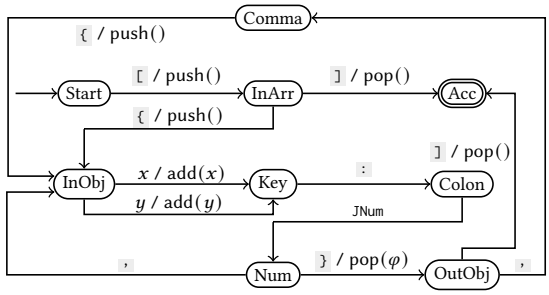
At the initial state 'Start', the outgoing transition on `JNum` corresponds to the base case and leads to the accepting state. The self-loop at 'Start' on `[` is for the recursive case of the schema. The outgoing transition from the state 'Bracket' is taken when an empty array is seen. At the 'Accept' state, the self-loop on `]` corresponds to returning from one level of recursion. If we encounter a comma and the stack is not empty, we remain at the same level. We omit the stack symbol for the push and pop transitions as the automaton has only one stack symbol. For simplicity, we omit the rejecting state, which is a sink. Missing transitions go to the rejecting state.

3.2 Constructions on NJAs

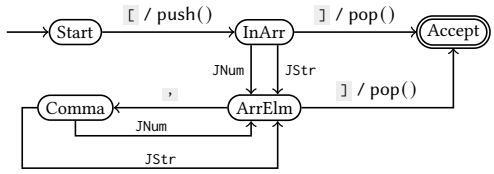
In this subsection, we present how NJAs are constructed from *JSL* schemas with examples for objects, arrays, unions, and recursion.



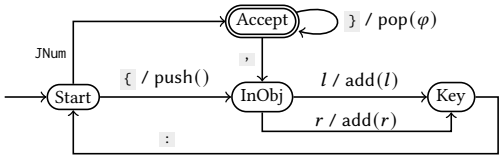
(a) Automaton A



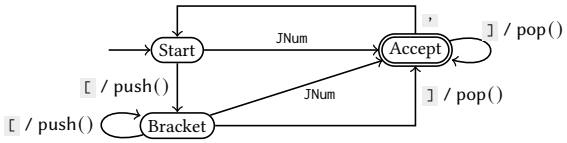
(b) Automaton B



(c) Automaton C



(d) Automaton D



(e) Automaton E

Figure 5: Examples of JSON automata (A–E).

Example 4. We consider the schema that accepts arrays of objects, where each object is a 2D point. Fig. 5b shows the automaton for the schema $\text{Arr}(\text{Obj}(\texttt{"x"} : \text{Num}, \texttt{"y"} : \text{Num}))$. We consider the JSON document from Example 1. We show part of the execution path of the NJA on this document. We omit the stack from a configuration whenever it does not change from the previous one.

$(\text{Start}, [\,]) \to^{\texttt{[}} (\text{InArr}, [\texttt{Arr}]) \to^{\texttt{\{}} (\text{InObj}, [\texttt{Arr}, \texttt{Obj}(\emptyset)]) \to^{\texttt{"x"}}$

$(\text{Key}, [\texttt{Arr}, \texttt{Obj}(\{\texttt{"x"}\})]) \to^{\texttt{:}} \text{Colon} \to^{\texttt{1.0}} \text{Num} \to^{\texttt{,}} \text{InObj} \to^{\texttt{"y"}}$

$(\text{Key}, [\texttt{Arr}, \texttt{Obj}(\{\texttt{"x"}, \texttt{"y"}\})]) \to^{\texttt{:}} \text{Colon} \to^{\texttt{1.0}} \text{Num} \to^{\texttt{\}}}$

$(\text{OutObj}, [\texttt{Arr}]) \to^{\texttt{,}} \text{Comma} \to^{\texttt{\{}} (\text{InObj}, [\texttt{Arr}, \texttt{Obj}(\emptyset)]) \to^{\texttt{"x"}}$

$\cdots \to^{\texttt{1.5}} \text{Num} \to^{\texttt{\}}} (\text{OutObj}, [\texttt{Arr}]) \to^{\texttt{]}} (\text{Acc}, [\,])$.

Example 5. We consider the schema Arr(Or(Str, Num)), which describes arrays in which each element can be either a string or a number. Fig. 5c presents the automaton for this schema. Notice that the automaton collapses the initial and the final states of the automata for the two subschemas, Str and Num, in the Or construct. We show its execution path on `[1, 2, "hi"]`.

$$(\text{Start}, [\,]) \to^{\texttt{[}} (\text{InArr}, [\texttt{Arr}]) \to^{\texttt{1}} \text{ArrElm} \to^{\texttt{,}} \text{Comma} \to^{\texttt{2}}$$
$$\text{ArrElm} \to^{\texttt{,}} \text{Comma} \to^{\texttt{"hi"}} \text{ArrElm} \to^{\texttt{]}} (\text{Accept}, [\,]).$$

Example 6. Consider the NJA that accepts *binary trees* that store numbers at the leaves. Such a tree can either be a Num or an object with two required keys, `"l"` and `"r"`, for the left and right subtrees. This data structure can be encoded by the following recursive schema: $X := \text{Or}(\text{Num}, \text{Obj}(\texttt{"l"} : X, \texttt{"r"} : X))$. Fig. 5d presents the automaton for this schema. We consider the document `{"l": {"l": 1, "r": 2}, "r": 3}`. This document describes a binary tree whose left subtree is a tree with two leaves storing the values 1 and 2, respectively, and whose right subtree is a leaf that stores the value 3. The execution path of the automaton on this JSON document is shown below.

$$(\text{Start}, [\,]) \to^{\texttt{\{}} (\text{InObj}, [\texttt{Obj}(\emptyset)]) \to^{\texttt{"l"}} (\text{Key}, [\texttt{Obj}(\{\texttt{"l"}\})]) \to^{\texttt{:}}$$
$$\text{Start} \to^{\texttt{\{}} (\text{InObj}, [\texttt{Obj}(\{\texttt{"l"}\}), \texttt{Obj}(\emptyset)]) \to^{\texttt{"l"}}$$
$$(\text{Key}, [\texttt{Obj}(\{\texttt{"l"}\}), \texttt{Obj}(\{\texttt{"l"}\})]) \to^{\texttt{:}} \text{Start} \to^{\texttt{1}} \text{Accept} \to^{\texttt{,}}$$
$$\text{InObj} \to^{\texttt{"r"}} (\text{Key}, [\texttt{Obj}(\{\texttt{"l"}\}), \texttt{Obj}(\{\texttt{"l"}, \texttt{"r"}\})]) \to^{\texttt{:}} \text{Start} \to^{\texttt{2}}$$
$$\text{Accept} \to^{\texttt{\}}} (\text{Accept}, [\texttt{Obj}(\{\texttt{"l"}\})]) \to^{\texttt{,}} \text{InObj} \to^{\texttt{"r"}}$$
$$(\text{Key}, [\texttt{Obj}(\{\texttt{"l"}, \texttt{"r"}\})]) \to^{\texttt{:}} \text{Start} \to^{\texttt{3}} \text{Accept} \to^{\texttt{\}}} (\text{Accept}, [\,])$$

In practice, NJA execution would be very slow as, in addition to maintaining a list of active states, we also need to maintain a list of stacks. Therefore, we define the deterministic variant of NJAs, which we call DJAs. We use DJAs for schema validation.

3.3 Deterministic JAs

Let K be a finite set of keys. A *key predicate* is a subset of $\mathcal{P}(K)$. A set KAt of key predicates that partitions $\mathcal{P}(K)$ is called a set of K*-atoms*. We will typically use letters α, β, γ for elements of KAt.

Definition 7 (DJA). Let K be a finite set of keys and KAt be a set of K-atoms. A *deterministic JSON automaton* (DJA) over K and KAt is a tuple $\mathcal{A} = (Q, q_0, C, \delta, F)$, where Q is a finite set of (control) states, $q_0 \in Q$ is the *initial* state, C is a finite set of *stack symbols*, $F \subseteq Q$ is the set of *final* or *accepting* states, and δ consists of the following collection of *transition functions*:

$$\delta_\ell : Q \to Q \times C \text{ for } \ell \in \{\texttt{[}, \texttt{\{}\} \qquad \delta_{\texttt{Key}} : Q \times K_o \to Q$$
$$\delta_{\texttt{]}} : Q \times C \to Q \qquad \delta_i : Q \to Q$$
$$\delta_{\texttt{\}}} : Q \times C \times KAt \to Q$$

where $i \in \{\texttt{null}, \texttt{false}, \texttt{true}, \texttt{JNum}, \texttt{JStr}, \texttt{:}, \texttt{,}\}$.

For each $k \in K_o$, the transition function $\delta_k : Q \to Q$ is given by $\delta_k(q) = \delta_{\texttt{Key}}(q, k)$ for every $q \in Q$.

We use the same notations for transitions as we did for NJAs. The only difference is the case of transitions on the right brace:

$$\frac{q' \in \delta_{\texttt{\}}}(q, c, \beta)}{q \to^{\texttt{\}}/\text{pop}(c,\beta)} q'} \qquad \frac{q' \in \delta_{\texttt{\}}}(q, c, \beta) \qquad m \models \beta}{(q, \sigma \cdot \texttt{Obj}(c, m)) \to^{\texttt{\}}} (q', \sigma)}$$

The symbol $\to$ above is for the (semantic) transition relation.

Example 8 (DJA for Nested Arrays of Numbers)**.** We want to accept arbitrarily nested arrays of numbers. The schema is defined with the equation $X := \text{Or}(\text{Num}, \text{Arr}(X))$. Fig. 5e presents the DJA for this schema. At the initial state, the outgoing transition on `JNum` corresponds to the base case and leads to the accepting state. The outgoing transition on `[` to state 'Bracket' is for the recursive case. At state 'Bracket', we can receive an arbitrary number of left brackets that increase the recursion level; we can take a `JNum` to remain at the current level; or we can take a right bracket to go back to the previous level of recursion. At the accepting state, we have a self-loop that takes a right bracket to return to the previous level of recursion, and we may also encounter a comma, in which case we remain at the same level.

Determinization. Next, we focus on the determinization procedure to convert an NJA into an equivalent DJA. The determinization of NJA is analogous to that of VPAs as presented in [3] and later refined in [51]. Compared to VPAs, NJAs make use of guards, thus requiring the use of K-atoms also known as the "minterms" in symbolic automata [20].

Let K be a finite set of keys and $\mathcal{A} = (Q, I, C, \Delta, F)$ be an NJA over K. Suppose that KAt is a set of K-atoms so that every predicate φ that appears in $\mathcal{A}$ can be written as a (disjoint) union $\varphi = \beta_0 \cup \beta_1 \cup \cdots \cup \beta_{k-1}$ of K-atoms $\beta_i \in KAt$. We will define now the DJA $\mathcal{B}$ that results from determinizing $\mathcal{A}$ w.r.t. KAt. One could choose KAt to be the atoms of the Boolean algebra generated by the predicates that appear in $\mathcal{A}$.

We define a *powerstate* to be an element of the set $2^{Q \times Q}$ or, equivalently, the set of functions $Q \times Q \to 2$. That is, a powerstate can be viewed as a binary relation on Q or as a square Boolean matrix indexed by the NJA states. For powerstates S and T, we write $S \circ T$ to denote their composition (as relations). That is,

$$S \circ T = \{(q, q'') \mid (q, q') \in S \text{ and } (q', q'') \in T \text{ for some } q'\}.$$

We write $Id_Q = \{(q, q) \mid q \in Q\}$ for the identity powerstate. The states of the DJA $\mathcal{B}$ are the powerstates of $\mathcal{A}$. The initial state of $\mathcal{B}$ is Id_Q. The transition function on a symbol i that is either one of `null`, `false`, `true`, `JNum`, `JStr`, `:`, `,` or a key (element of K_o) is given by $S \to^i S \circ \Delta_i$. The stack symbols C of $\mathcal{B}$ are the powerstates of $\mathcal{A}$. For the transition function on the left bracket and brace, we define

$$S \to^{\texttt{[}/\text{push}(S)} Id_Q \quad \text{and} \quad S \to^{\texttt{\{}/\text{push}(S)} Id_Q\,.$$

The transitions on a right bracket are given as follows:

$$T \to^{\texttt{]}/\text{pop}(S)} S \circ \text{Upd}_A(T), \text{ where}$$
$$\text{Upd}_A(T) = \{(q, q') \mid q \to^{\texttt{[}/\text{push}(c)} q_1 \text{ and } (q_1, q_2) \in T \text{ and } q_2 \to^{\texttt{]}/\text{pop}(c)} q' \text{ for some } c, q_1, q_2\}\,.$$

The transitions on a right brace are given as follows:

$$T \xrightarrow{\texttt{}/\mathrm{pop}(S,\beta)} S \circ \mathrm{Upd}_O(T), \text{ where}$$

$$\mathrm{Upd}_O(T) = \{(q, q') \mid q \xrightarrow{\texttt{\{}/\mathrm{push}(c)} q_1 \text{ and } (q_1, q_2) \in T \text{ and } q_2 \xrightarrow{\texttt{}/\mathrm{pop}(c,\varphi)} q' \text{ for some } c, q_1, q_2, \varphi\}\,.$$

A powerstate S is defined to be final in $\mathcal{B}$ if it satisfies $S(I) \cap F \neq \emptyset$, where $S(I) = \{q' \mid q \in I \text{ and } (q, q') \in S \text{ for some } q\}$.

In fact, determinization is only needed for a specific class of NJAs, that we call 1-ambiguous. The notion of 1-unambiguity was first defined for regular expressions in [15] to extend ambiguity [11] to a 1-character lookahead and was adapted later on to DTD schemas in the context of XML [18, 29]. We define the set of first symbols for a schema s as $\mathit{first}(s) = \{a \mid aw \in [\![s]\!] \text{ for some } w\}$.

Definition 9 (1-ambiguity). A schema is called *1-ambiguous* if it contains a subschema of the form $\mathsf{Or}(s_1, \ldots, s_n)$ such that there are indexes i, j with $i \neq j$ and $\mathit{first}(s_i) \cap \mathit{first}(s_j) \neq \emptyset$. A schema is *1-unambiguous* if it is not 1-ambiguous.

Operationally, when we have a 1-ambiguous union $\mathsf{Or}(s_1, \ldots, s_n)$, we cannot choose among the subschemas $s_1, \ldots, s_n$ with a lookahead of only one symbol. When a schema is 1-unambiguous, its NJA is essentially deterministic and can be directly translated into an equivalent DJA, thus avoiding determinization altogether. For example, consider the 1-ambiguous schema $\mathsf{Or}(\mathsf{Arr}(\mathsf{Num}), \mathsf{Arr}(\mathsf{Str}))$: after seeing the opening token `[`, there is no deterministic way (without looking further ahead) to choose between the two array subschemas. In contrast, the schema $\mathsf{Or}(\mathsf{Num}, \mathsf{Str})$ is 1-unambiguous: the first token (either a number or a string in valid documents) uniquely determines the subschema that should be chosen.

4 STREAMING ALGORITHM FOR JSON VALIDATION

In this section, we propose a streaming algorithm for validating JSON documents against schemas. Our algorithm uses the automata (NJAs and DJAs) of Section 3.

We view a JSON document as a stream of bytes (or, more abstractly, as a stream of Unicode characters). Conceptually, the algorithm can be thought of as a pipeline of three stages: (1) the first stage converts the input byte stream into a stream of JSON tokens, (2) the second stage enriches the token stream with additional information about string tokens that are used as keys in the document (i.e., as property names in objects), and (3) the third stage executes an automaton (DJA) that captures the restrictions of the schema.

For the first stage, a standard tokenization or lexing algorithm (see, e.g., [35]) can be used to convert the JSON character stream into a stream of *JSON tokens*, which are of the following form:

`[` `{` `]` `}` `null` `false` `true` n s `:` `,`

where n is a JSON number and s is a JSON string.

The JSON token stream is not suitable as input for NJAs and DJAs, because these automata need to receive input symbols that distinguish between strings that are used as values (we represent them with the symbol `JStr`) and strings that are used as keys (represented with the symbol `JKey`). Making this distinction requires contextual information, i.e., we essentially have to parse the JSON token stream. We do not need, however, to construct the parse tree

```
   // State of algorithm
1  mode ← Start;   stack ← [ ]
2  Function GetEToken(tk : Token):
3      if mode = Start then
4          if tk = [ then stack.push(Arr(0)); mode ← Start; return tk
5          else if tk = { then stack.push(Obj(0)); mode ← Key; return tk
6          else if tk = ] then
7              e ← stack.last() // top of the stack
8              if e = Some(Arr(0)) then
9                  stack.pop(); ValueCompletion(); return tk
10             else error "parsing error"
11         else if tk is primitive value then ValueCompletion(); return tk
12         else error "parsing error"
13     else if mode = Key then
14         if tk = } then
15             e ← stack.last() // top of the stack
16             if e = Some(Obj(0)) then
17                 stack.pop(); ValueCompletion(); return tk
18             else error "parsing error"
19         else if tk = JStr(s) then
               // translate string to key identifier or other
20             k ← stringToKey(s); mode ← Colon; return JKey(k)
21         else error "parsing error"
22     else if mode = Colon then
23         if tk = : then mode ← Start; return tk
24         else error "parsing error"
25     else if mode = CommaArr then
26         if tk = ] then ValueCompletion(); return tk
27         else if tk = , then mode ← Start; return tk
28         else error "parsing error"
29     else if mode = CommaObj then
30         if tk = } then ValueCompletion(); return tk
31         else if tk = , then mode ← Key; return tk
32         else error "parsing error"
33     else if mode = Finished then error "parsing error"
34     else error "internal error (should be unreachable)"
35 Function ValueCompletion():
36     e ← stack.pop()
37     if let Some(e) = e then
38         if let Arr(n) = e then
39             e ← Arr(n + 1) // update counter for array elements
40             stack.push(e)
41             mode ← CommaArr
42         else if let Obj(n) = e then
43             e ← Obj(n + 1) // update counter for object fields
44             stack.push(e)
45             mode ← CommaObj
46         else error "internal error (should be unreachable)"
47     else // empty stack, top level
48         mode ← Finished
```

Figure 6: Streaming algorithm for generating e-tokens from the JSON token stream. The algorithm also checks that the stream is well-formed as a JSON value.

(i.e., the JSON AST). This gives rise to the notion of *extended tokens* or *e-tokens* for brevity. Additionally, we translate a key (string) into an identifier that is easier to use for lookups. The idea is that for a fixed schema S, there is a finite number of keys that appear in S and we can assign identifiers from a finite set K to them. In practice, these identifiers are integers. Keys that do not appear in S do not need to be distinguished and for this reason we associate all of them with the special identifier other. The streaming algorithm of Fig. 6 performs the transformation of the JSON token stream into the corresponding e-token stream, given a function *stringToKey* that takes a JSON string s and returns a key identifier. The function *stringToKey* is specific to a schema. Notice the use of *stringToKey* in line 20 of Fig. 6.

```
// DJA 𝒜 = (Q, q0, C, δ, F) for schema validation
// include the state of the algorithm of Fig. 6
q ← q0   // current state of DJA
vstack ← [ ]   // stack for DJA (i.e., for schema validation)
Function StreamingValidation(tk : Token):
    tk ← GetEToken(tk)   // generate e-token
    if tk = [ then (q', c) ← δ_[(q); vstack.push(Arr(c)); q ← q'
    else if tk = { then (q', c) ← δ_{(q); vstack.push(Obj(c, ∅)); q ← q'
    else if tk = ] then
        if let Some(Arr(c)) = vstack.pop() then q ← δ_](q, c)
        else error "parsing error"
    else if tk = } then
        if let Some(Obj(c, m)) = vstack.pop() then
            β ← the unique K-atom β ∈ KAt satisfying m ∈ β
            q ← δ_}(q, c, β)
        else error "parsing error"
    else if tk = k ∈ K then
        if let Some(Obj(c, m)) = vstack.pop() then
            e ← Obj(c, m ∪ {k}); vstack.push(e); q ← δ_Key(q, k)
        else error "parsing error"
    else if tk = other then
        if let Some(Obj(_, _)) = vstack.last() then q ← δ_Key(q, other)
        else error "parsing error"
    else if tk = i ∈ { null , false , true , JNum, JStr, : } then q ← δ_i(q)
    else if tk = , then
        if vstack ≠ [ ] then q ← δ_,(q)
        else error "parsing error"
    else error "internal error (should be unreachable)"
    return (q ∈ F) ∧ (vstack is empty)
```

Figure 7: Streaming DJA Execution for Schema Validation.

At a high-level, the algorithm of Fig. 6 implements a state machine that can be in any one of six modes:

Start *Key* *Colon* *CommaArr* *CommaObj* *Finished*

The mode *Start* indicates that we expect to construct a JSON value with the next tokens. When we are in mode *Key*, we expect the next token to be a key or possibly the closing brace of an object. The algorithm also maintains a stack that stores entries of the form `Arr`(n) or of the form `Obj`(n). The symbols `Arr` and `Obj` indicate the parsing of an array and object respectively. The values n that are stored keep track of the current sizes of arrays and objects. This information is useful for correctly identifying empty arrays `[ ]` and empty objects `{ }`. Notice that the algorithm parses the input without constructing the JSON document AST. This means that the memory footprint is bounded above by the height of the JSON document (i.e., the depth of nesting of arrays and objects).

Given an input schema S, we first construct an NJA $\mathcal{A}$ that recognizes the JSON documents (given in the form of e-token sequences) that adhere to S. The NJA is then converted into an equivalent DJA $\mathcal{B}$ using the determinization procedure described in Section 3.

The third stage of the pipeline receives the stream of e-tokens and simulates the execution of the DJA $\mathcal{B}$. We describe this with the algorithm of Fig. 7. Unsurprisingly, the algorithm closely follows the abstract execution semantics that we presented in Section 3 (in terms of the transition relation $\rightarrow$ on automata configurations).

Time and Space Complexity. The space complexity of the validation algorithm (Fig. 7) is $O(h)$, where h is the input document height. The memory usage of the algorithms from Fig. 6 and Fig. 7 is proportional to the stack sizes, which are bounded above by the document height. In practice, the height of a document is much smaller than its size. Our algorithm is therefore expected to have a small memory footprint, which we experimentally validate in Section 5. The time complexity of DJA execution is $O(1)$ per token, because performing a transition is implemented as a table lookup. The caveat with deterministic automata is that they can be large in the worst case (this is inherent in the model, similarly to the case when DFAs are exponentially larger than NFAs). We have observed that most real-world schemas give rise to DJAs of manageable size, and therefore schema validation with our approach is fast.

5 EXPERIMENTS

We have implemented the schema validation algorithm of §4 using the Rust programming language. In this section, we evaluate the performance of our tool to answer the following research questions:

RQ1. [Memory] What is the memory benefit of our streaming approach compared to existing state-of-the-art tools?

RQ2. [Throughput] Is our tool competitive in terms of throughput compared to state-of-the-art tools?

Experimental Setup. The experiments were executed on a desktop computer running Ubuntu 22.04 and equipped with an Intel Core i9-12900K CPU (with 16 cores) and 32 GB of memory. We used Rust 1.81.0 to compile our tool. For each experiment, we conducted 10 trials and report the mean of the measurements.

Tools. We evaluate the performance of our proposed tool, **DJA**, against five state-of-the-art JSON Schema validators. The tools are: (1) **AJV** [1], one of the most widely used JSON Schema validators for JavaScript; (2) **Boon** [12] and (3) **JsonSchema-Rust** (JSR) [28], which are validators written in Rust; (4) **Rec**, our Rust implementation of the classical recursive offline validation algorithm; and (5) **VPA** [16], a schema validator based on VPAs. These tools fall into two categories: (1) offline validators (AJV, Boon, JsonSchema-Rust, and Rec), which validate against a parse tree of the input JSON document, and (2) streaming validators (DJA and VPA), which operate directly over the stream of input tokens. For all offline tools except AJV, we use SIMD-JSON [33], a state-of-the-art SIMD-accelerated JSON parser, to build parse trees of JSON documents. AJV instead relies on the native JavaScript JSON parser. Since offline tools construct the full parse tree for the input document, their memory consumption grows with document size. AJV compiles JSON schemas into specialized JavaScript functions, whereas the other offline tools do not generate schema-specific validation code.

Methodology. To measure the memory footprint of each tool, we record the maximum resident set size (RSS) of its main process. The RSS accounts for the memory allocated on the stack, heap, and memory-mapped pages and thus provides a reliable estimate of the peak RAM usage during execution. In practice, this measure overestimates the memory consumption of our streaming validator DJA. However, this will not be an issue for the conclusions we will draw, since DJA's memory footprint on large input documents is orders of magnitude smaller than that of offline validators.

5.1 VPA Evaluation

The only streaming implementation for JSON Schema validation that we are aware of is [50] (see also [16]). In [16], the authors

Table 1: Simple schemas for comparing with the VPA tool.

Name	Schema
arr-num	Obj(`"inner"` : Arr(Num))
arr-str	Obj(`"inner"` : Arr(Str))
2d-points	Obj(`"x"` : Arr(Num), `"y"` : Arr(Num))
2d-points-obj	Obj(`"table"` : Arr(Obj(`"x"` : Num, `"y"` : Num)))

explore the approach of learning automata for schema validation from examples. They use 1-SEVPAs [2, 25], which is a subclass of VPAs that can be efficiently minimized. If the learning procedure takes too long, then it is cut short, which results in an approximation to the desired automaton. In practice, learning can take several weeks (as reported in [16]), which makes it impractical in many settings. To circumvent the difficulty in learning correct automata, we restrict our experiments to schemas for which we can manually create the minimal 1-SEVPA that precisely implements the schema.

We consider four simple schemas (see Table 1): **arr-num**, which accepts objects containing an array of numbers, such as

$$\texttt{\{ "inner" : [}\, n_1 \,\texttt{,}\, \ldots \,\texttt{,}\, n_k \,\texttt{] \}};$$

arr-str, which differs from arr-num in that the arrays contain strings; **2d-points**, which describes a columnar representation of a table containing 2D points, such as

$$\texttt{\{ "x" : [}\, m_1 \,\texttt{,}\, \ldots \,\texttt{,}\, m_k \,\texttt{] , "y" : [}\, n_1 \,\texttt{,}\, \ldots \,\texttt{,}\, n_k \,\texttt{] \}};$$

and **2d-points-obj**, an alternative representation where the table is given as an array of 2D objects, such as

$$\texttt{\{ "table" : [\{ "x" :}\, m_1 \,\texttt{, "y" :}\, n_1 \,\texttt{\} ,} \ldots \texttt{, \{ "x" :}\, m_k \,\texttt{, "y" :}\, n_k \,\texttt{\}] \}}.$$

We also include two larger schemas from prior work [16]: **basic-types**, which accepts objects with properties of various types (object, string, number, array), and **vscode** [53], which specifies the format of Visual Studio Code snippets.

Fig. 8 presents the throughput of the tools over the 6 schemas. For all schemas, the VPA tool has the lowest throughput, around 20 MB/s, which is an order of magnitude slower than the other tools. This is because the VPA tool first tokenizes the input document into a token stream that is fed to the automaton, and uses a key graph to handle reordering of object keys, further slowing execution. Notice that some of the tools cannot perform validation when the document size becomes large because they run out of memory. An important aspect of the VPA tool is that it operates on symbolic documents; for example, in the array-of-numbers schema, only inputs of the form `{ "inner" : [ "\D" , ... , "\D" ] }` are accepted, where `"\D"` represents a value of type 'number'.

The VPA tool lacks a direct schema-to-SEVPA translation. As discussed, it instead relies on a learning procedure to approximate the SEVPA from examples. Learning can take several weeks for some schemas. Since learning is too slow and can produce an incorrect automaton, we exclude the VPA tool from our later experiments.

5.2 Micro-benchmarks

We consider here three families of schemas that explore the asymptotic behavior of our DJA-based algorithm.

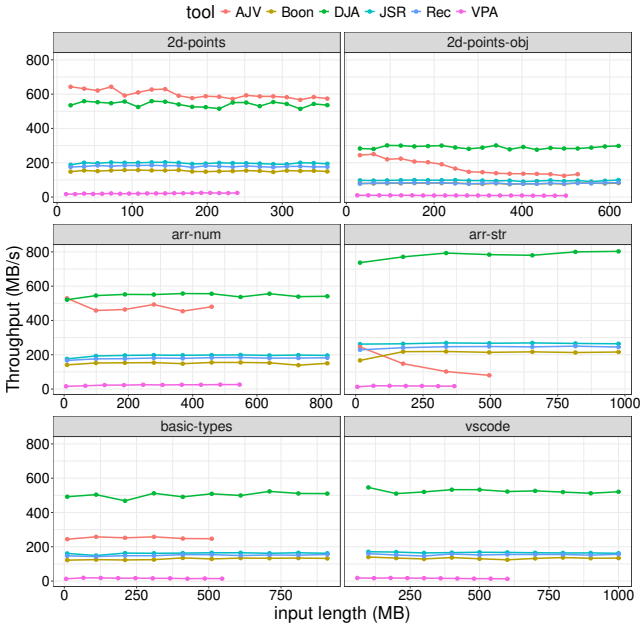


Figure 8: Performance comparison between AJV, Boon, JSR, DJA, Rec and VPA over the 6 schemas of §5.1.

Input Generation. We have created an input generator that produces a JSON document that is valid for a given schema s. To scale up the experiments over large input documents, we perform validation using the schema Arr(s) on the JSON document $\texttt{[}\, d_1 \,\texttt{,}\, \ldots \,\texttt{,}\, d_k \,\texttt{]}$ where $d_1, \ldots, d_k$ are JSON documents generated by our input generator so that they are accepted by schema s.

Conjunction of Objects. For the first micro-benchmark, we focus on the equivalent schemas

$$s_{and} = \text{And}(\text{Obj}(k_1 : \text{Str}), \ldots, \text{Obj}(k_m : \text{Str})) \text{ and}$$
$$s_{obj} = \text{Obj}(k_1 : \text{Str}, \ldots, k_m : \text{Str})\,.$$

For s_{obj}, the classical offline validator can efficiently check the presence of all keys (and the types of the correponding values) in $O(n)$ time with a single pass over the input, where n is the size of the input document. In contrast, validating against s_{and} needs $O(m \cdot n)$ time because the input document is validated against each of the subschemas Obj(k_i : Str), thus traversing the input tree m times.

Fig. 9 compares the performance of the offline tools and our DJA-based algorithm for this micro-benchmark (s_{and}). We observe that the performance of the offline tools is consistent with $O(m \cdot n)$ time complexity. The performance of our DJA-based algorithm is consistent with $O(n)$ time complexity. AJV performs relatively well due to code generation, whereas Boon is the worst-performing tool. In our tool, the DJA is built using a product construction, and therefore the number of states increases with the number m of subschemas in s_{and}. However, after removing unreachable and dead-end states, most states are eliminated, leading to almost no performance decrease as m increases. The constructed DJA remembers the keys seen so far with a bit vector of size m (number of keys) and verifies that the value is a string. If all keys $k_1, \ldots, k_m$

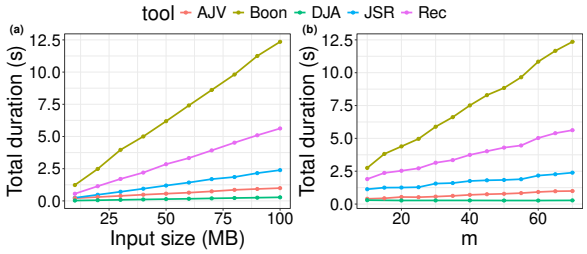


Figure 9: Performance comparison between AJV, Boon, JSR, DJA and Rec over the conjunction-of-objects micro-benchmark for (a) $m = 70$ schemas and varying input size (between 10 and 100 MB), and for (b) input size $n = 100$ MB and varying number of schemas (between 10 and 70 schemas).

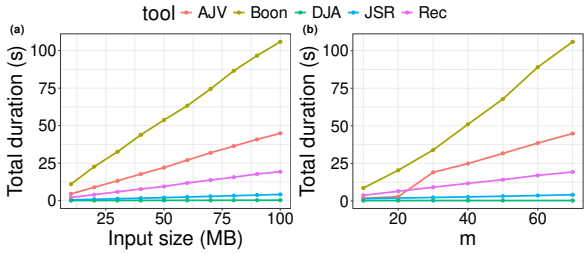


Figure 10: Performance comparison between AJV, Boon, JSR, DJA and Rec over the disjunction-of-objects micro-benchmark for (a) $m = 70$ schemas and varying input size (between 10 and 100 MB), and for (b) input size $n = 100$ MB and varying number of schemas (between 10 and 70 schemas).

are present in the object, we transition to the accepting state when the closing brace is seen. We observe $O(n)$ running time in practice because the bit vector data structure provides efficient support for the operations of adding a new key and checking if all the required keys were encountered.

Disjunction of Objects. We consider now the family of schemas $\mathrm{Or}(s_{m-1}, \ldots, s_0)$, where $s_i = \mathrm{Obj}(k_0 : \mathrm{Str}, \ldots, k_i : \mathrm{Str})$ for $i = 0, \ldots, m-1$. It holds that $[\![s_{i+1}]\!] \subseteq [\![s_i]\!]$ for every i, and therefore $\mathrm{Or}(s_{m-1}, \ldots, s_0)$ is equivalent to $s_0 = \mathrm{Obj}(k_0 : \mathrm{Str})$. Using the classical offline validation algorithm, the subschemas $s_{m-1}, \ldots, s_0$ are evaluated in order until one of them accepts the input document. Let us consider how the schema is evaluated against a document of the form `{` k_0 `:` str `}`, where str is a JSON string. The validator has to evaluate all subschemas s_i, since only s_0 accepts the document.

Fig. 10 compares the performance between the offline tools and our DJA-based algorithm for this benchmark. The results for the offline tools are consistent with $O(m \cdot n)$ time complexity. The running time of DJA is independent of m. Among the offline tools, JSR is the best-performing and Boon is the worst-performing. For this family of schemas, the union is 1-ambiguous (Def. 9). Our approach uses a union construction for automata, followed by determinization (this is needed for 1-ambiguous schemas). However, our construction does not lead to a performance penalty (for this family of schemas), because the number of K-atoms does not blow up. Our approach leads to an execution that corresponds to the automaton for the simplified schema $\mathrm{Obj}(k_0 : \mathrm{Str})$.

Linked List. As discussed in §4, the memory footprint of our algorithm is proportional to the height of the input document. We will now investigate the worst-case space complexity of our algorithm using the recursive schema of a linked list $X := \mathrm{Obj}($ `"value"` $: \mathrm{Num},$ `"next"` $? : X)$. We use linked lists because the size and height of the document are linearly related.

Fig. 11 shows the memory footprint of the offline tools against our DJA-based implementation for document height up to 1 million. DJA uses an explicit stack on the heap to deal with the nesting in documents. However, all other tools rely on the call stack for recursive tree traversal. They cause a stack overflow for document height that exceeds 20,000 (5,000 for AJV), because the maximum stack size is set to 16 MiB by default on our Linux system. To handle longer lists for our experiments, we manually increase the maximum stack

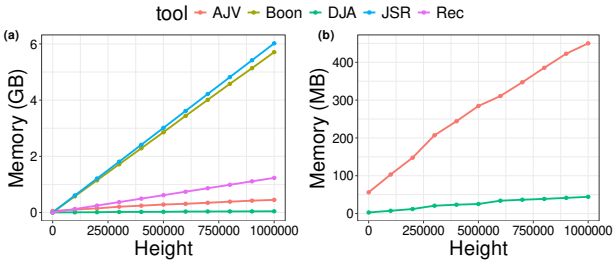


Figure 11: Performance comparison between (a) AJV, Boon, JSR, DJA and Rec, and (b) AJV against DJA over linked list schema.

size to 16 GiB. In Fig. 11(a), we observe that all tools have a memory footprint that is linear in document height. In Fig. 11(b), we focus on the tools AJV and DJA.

JSR	Boon	Rec	AJV	DJA
5879	5573	1203	439	42.5

The table above shows the memory footprint of all tools (in MB) for document height equal to 10^6. AJV, which is the most memory-efficient offline tool, uses at least 10 times more memory than DJA.

5.3 Database Benchmark

We evaluate the throughput and memory consumption of the offline tools against our streaming tool over a benchmark of databases. JSON is a popular format to export and import databases. We have chosen this benchmark because validating a database against a schema is a common application. When a database is loaded into a database system, type and value domain checks are often performed. JSON Schema can be used to represent the required format for the database. In addition to relational databases, schema validation can also be performed in NoSQL databases (such as MongoDB [39]) over semi-structured documents. These applications require an efficient validator both in terms of throughput and memory consumption.

Databases. We evaluate the performance of the tools over 15 schemas, which specify the formats of real-world datasets. The datasets are: (1) **AirBnB**, a collection of AirBnB apartment listings, (2) **ArXiv**, a list of scientific publications available on arXiv, (3)

Table 2: Benchmarks for the JSON Schema validators.

Dataset	Description	Size	Height
AirBnB	AirBnB listings	697 MB	3
ArXiv	Scientific publications	4.4 GB	4
Clang	Clang AST	1 GB	280
Corporations	Incorporated businesses in New York	1.8 GB	2
Crime	Crimes in Chicago	5.3 GB	4
EclipseLink	Java persistence framework	1.1 GB	10
Estate	Real estate sales in Connecticut	320 MB	4
Euro	Exchange rate of euro	530 MB	2
Foot-events	Football events	8.2 GB	6
Fraud-detection	Neo4j graph for fraud detection	396 MB	4
Health	Healthcare providers in Washington	629 MB	2
Offshore-leaks	Neo4j graph of the Panama papers	3.8 GB	3
Twitch	Neo4j graph of the Twitch platform	5.1 GB	3
Yelp-review	Yelp reviews	5.3 GB	2
Yelp-user	Yelp users	3.4 GB	2

Clang, the AST generated by the Clang C++ compiler for the coreutils codebase [19], (4) **Corporations**, a collection of incorporated businesses in the city of New York since 1800, (5) **Crime**, a list of the crimes in Chicago since 2001, (6) **EclipseLink**, a collection of data structures that are serialized using a Java framework for persistence, (7) **Estate**, a dataset that contains the real estate sales in the state of Connecticut, (8) **Euro**, a dataset that contains the daily exchange rates of the euro against foreign currencies, (9) **Foot-event**, a database for football game statistics, (10) **Fraud-detection**, a graph database that contains information about cases of financial fraud, (11) **Health**, a list of healthcare providers in the state of Washington, (12) **Offshore-leaks**, a graph database that shows the connections between companies and people in the Panama papers leak, (13) **Twitch**, a graph database for the Twitch streaming platform, (14) **Yelp-review**, a collection of reviews from Yelp, and (15) **Yelp-user**, an anonymized version of the Yelp user database. The input sizes and heights of the databases are listed in Table 2.

Memory Footprint. Table 3 shows the maximum RSS (resident set size) used by the tools over the Database benchmark. Due to size constraints imposed by the parser (SIMD-JSON [49] can handle document size up to 4 GB), we have to truncate some datasets. JavaScript restricts string sizes to 1 GB, which limits the document size supported by AJV to at most 1 GB. For DJA, the RSS measurement substantially overestimates the memory footprint of our algorithm, as it includes the memory used for the program code (which is typically much larger than the state of our algorithm).

We observe in Table 3 that DJA offers substantial memory savings: between 65× (for Euro) and 557× (for Corporations) compared to the most memory-efficient offline tool (AJV). The offline tools require a large amount of memory because they store the entire parse tree of the JSON document. The JavaScript parser (used by AJV) is more space-efficient in representing the parse tree compared to SIMD-JSON (used by Boon, JSR, and Rec).

Fig. 12 shows the maximum RSS used by the tools as a function of the input size. We have drawn linear regression lines to make the relationship between RSS and input size more apparent. DJA has a small memory footprint because we use a compact representation for the transition relations and the stack size is bounded above by the document height, which is generally small. Notice in Fig. 12 and Table 3 that the offline tools Boon, JSR, and Rec have

Table 3: Maximum RSS (in MB) for AJV, Boon, JsonSchema-Rust, DJA and Rec. The Reduction column is the ratio of AJV over DJA.

Dataset	AJV	Boon	JSR	DJA	Rec	Reduction
AirBnB	1045	5064	5065	2	5064	522
ArXiv	762	7447	7447	3	7445	254
Clang	894	15625	15625	2	15624	447
Corporations	1114	3343	3343	2	3342	557
Crime	620	6613	6613	2	6611	310
EclipseLink	380	4234	4234	2	4233	190
Estate	386	2618	2619	2	2616	193
Euro	195	716	716	3	715	65
Foot-events	1009	9648	9648	2	9646	504
Fraud-detection	573	4171	4171	3	4170	191
Health	762	2716	2716	3	2714	254
Offshore-leaks	1115	4177	4177	3	4176	372
Twitch	409	1978	1978	3	1977	136
Yelp-review	1089	2897	2898	2	2896	544
Yelp-user	1060	6161	6161	3	6160	353

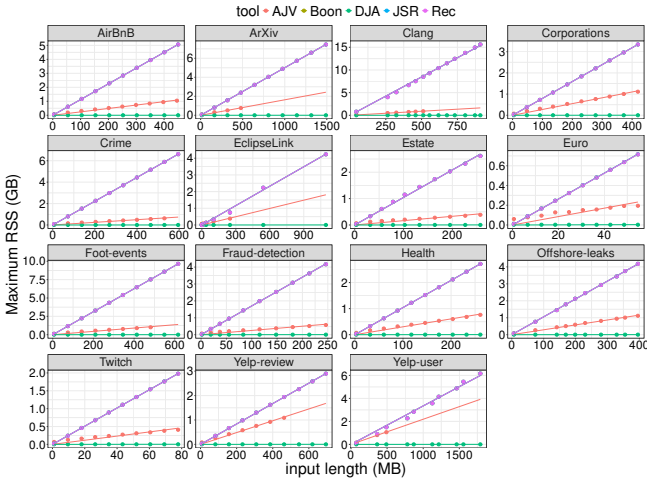


Figure 12: Maximum RSS (Resident Set Size) of the validators over the benchmark schemas for increasing input sizes.

almost the same memory footprint. This is because they all use the SIMD-JSON [33] parser. The memory footprint of our DJA implementation remains small, even for the Clang dataset that has the largest document height within the benchmark (height 280). As the input increases, the advantage of our streaming approach (in terms of memory footprint) increases compared to the offline algorithms. These results show the relevance of our tool for dealing with schema validation over large documents.

Throughput. Table 4 presents the throughput (in MB/s) of the tools over the Database benchmark. We observe that, for all datasets, DJA is the fastest tool. The second fastest tool is AJV. We believe that the speed of AJV can be attributed to its code generation feature. The speedup ratio of our implementation is between 1.2× (for Yelp-user) and 3.8× (for Fraud-detection) compared to AJV. This shows that our DJA implementation is competitive with the other tools in terms of throughput. Therefore, our implementation uses a small amount of memory (~2 MB on average) compared to the offline

Table 4: Average throughput (in MB/s) for AJV, Boon, JSR, DJA and Rec. Input size is fixed to the largest input size supported by the tools. The Speedup column is the ratio DJA/AJV.

Dataset	AJV	Boon	JSR	DJA	Rec	Speedup
AirBnB	355	135	156	457	106	1.3
ArXiv	360	273	303	900	264	2.5
Clang	220	94	97	370	81	1.7
Corporations	240	179	203	552	85	2.3
Crime	290	124	140	406	104	1.4
EclipseLink	545	367	471	787	335	1.4
Estate	213	134	145	481	108	2.3
Euro	220	106	121	354	101	1.6
Foot-events	144	91	101	351	66	2.4
Fraud-detection	83	76	90	338	66	3.8
Health	209	133	152	454	111	2.2
Offshore-leaks	367	142	160	524	106	1.4
Twitch	188	65	76	307	58	1.6
Yelp-review	723	321	352	1218	289	1.7
Yelp-user	1152	378	411	1395	329	1.2

Table 5: Running time and memory usage of the tools over the Corpus benchmark. Total duration in seconds, average and max durations in milliseconds, memory in MB.

tool	dur.	avg. dur.	max. dur.	avg. mem	max. mem
DJA	595	21	65	2	3
AJV	1651	60	354	107	309
JSR	1739	63	144	119	332
Boon	2020	73	316	119	332
Rec	2186	79	694	118	331

tools that construct the parse tree (which can take up to a few GB for databases), and is also faster.

5.4 Corpus Benchmark

The Corpus benchmark [5] consists of a large collection of JSON schemas collected in 2020 from open-source GitHub repositories. We filter out the schemas that cannot be fully resolved locally (because they have external references). We compare the performance of the offline validators AJV, Boon, JSR and Rec against our streaming tool DJA over the remaining schemas. For each schema, we generate a valid input document using our input generator. The size of the each input document is roughly 10 MB.

Performance Results. Table 5 summarizes the duration and memory footprint for all tools over the Corpus benchmark. The column 'duration' gives the total running time to process the entire Corpus benchmark (almost 30,000 schemas). The column 'average duration' (resp., 'maximum duration') gives the average (resp., maximum) per-schema running time.

The results show that DJA has the smallest total duration, being at least 3× faster than the offline algorithms, and indicate that our DJA-based algorithm can compete against optimized implementations such as AJV. We also observe that the average memory footprint of DJA is orders of magnitude smaller than that of the offline tools. Among the offline tools, AJV has the smallest average memory footprint at around 107 MB. Boon, JSR and Rec all use the SIMD-JSON parser, and we observe that they have the same average memory footprint. This indicates that the maximum RSS

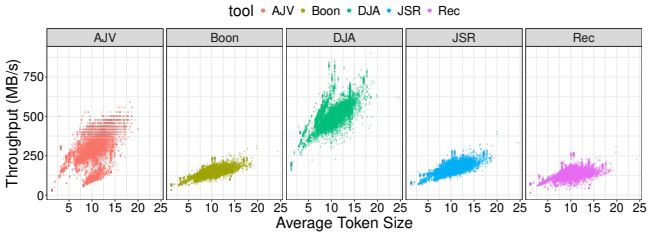


Figure 13: Relationship between throughput and average token size over the Corpus benchmark.

measurement is determined by the size of the parse tree that is stored in memory. As expected, our DJA tool has a constant memory footprint as it is a streaming implementation. Compared to the other offline tools, AJV uses less memory, which is consistent with the observations we had for the database benchmark.

Throughput versus Average Token Size. Fig. 13 presents the relationship between throughput (in MB/s) and the average token size of the input document (in number of bytes). We observe that, for all tools, the throughput is influenced by the average token size. This is expected because the validation algorithm has to process each token of the input document. If the validator is given fewer tokens for the same input document size, then the validation time decreases and hence the throughput increases.

5.5 Summary of Main Observations

We conclude the section with a discussion of our main observations.

Summary of RQ1. For large input documents (between 100 MB and 10 GB) of the Database benchmark, our tool DJA reduces the memory footprint between 65 and 557 times. For the smaller input documents (roughly 10 MB) of the Corpus benchmark, DJA uses only 2 MB (overestimation) of memory on average. So, the memory footprint of DJA is more than 50 times smaller compared to AJV. These results indicate that DJA can provide significant memory footprint reduction for many practical workloads.

Summary of RQ2. For the first two micro-benchmarks of §5.2 (i.e., the conjunction-of-objects and disjunction-of-objects schema families), our DJA-based algorithm has $O(n)$ running time, whereas the other tools have $O(m{\cdot}n)$ running time. For the Database and Corpus benchmarks, DJA has good performance (in terms of throughput) against the other tools. DJA is up to 3.8× faster for Database and has an average speedup of almost 3× for Corpus. So, DJA is competitive against other tools.

6 RELATED WORK

Schema validation has been explored in the context of both XML and JSON, more specifically, for the DTD [21], XML Schema [10, 56], and JSON Schema [27] languages. ***Offline algorithms*** for JSON schema validation have been described in [13, 14, 44]. These algorithms perform a recursive traversal of the JSON document AST (parse tree) while checking the satisfaction of the relevant subschemas. The main drawback of these offline algorithms is the memory needed to store the parse tree of the document.

To tackle the issue of high memory usage, [18, 47, 48] have considered the use of ***Finite State Machines*** (FSM) to validate DTDs in an online fashion. The authors of [48] show that nondeterministic automata can be used for non-recursive DTD validation. Later, [18] uses the notion of one-unambiguity [15] to propose a DFA construction for non-recursive DTD validations when all regexes are one-unambiguous. As a follow-up of [48], [47] refines the subclass of streaming recursive DTDs where validation can be performed with an FSM. The paper [40] proposes a block-by-block algorithm for parallel streaming validation where each block is processed by a Boolean circuit. It demonstrates that validation for non-recursive DTDs can be performed with AC^0 circuits and identifies a subclass that can be validated with $WLAC^0$ circuits. These FSM-based methods do not extend to JSON schemas. By default, JSON objects may contain additional keys, each with a valid JSON value. Since the membership problem for the Dyck-2 language of well-parenthesized/bracketed words requires a stack [36], checking for JSON validity also requires an unbounded stack, which makes FSMs insufficient for most JSON schemas.

Another approach is based on the more expressive model of ***One-Counter Automata*** [7, 22, 23]. The paper [18] gives sufficient conditions for recursive DTDs to be validated by a one-counter automaton. More recently, [9] extends one-counter automata with k registers to store information about relevant nodes and use the counter to store the current depth. While this technique can help support more schemas using constant memory, it faces the same limitation as FSM regarding JSON schema validation and cannot be used to check for validity of JSON documents.

To overcome the limitations of FSMs and counter automata, stack-based approaches based on ***Visibly Pushdown Automata*** [3] (VPAs) have been proposed for streaming schema validation. In [48], a deterministic automaton with a bounded stack, similar to VPAs, is proposed for schema validation of recursive DTDs. More recently, [31] has proposed a multi-pass algorithm for DTD validation based on the First-Child-Next-Sibling encoding of XML trees with $O(\log^2 n)$ space complexity and $O(\log n)$ time per item. While this approach has a low memory footprint, [32] shows that for every DTD one can construct a deterministic modular VPA [2] with constant time per symbol and space bounded by the document height, which is more efficient than the multi-pass algorithm. JSON's unordered keys make VPAs impractical for schema validation, as VPAs have to encode every possible key order. A solution for this issue is explored in [16]. This work uses a learning algorithm to build a VPA for a given schema with a fixed key ordering and then relies on a "key graph" to succinctly store all the possible reordering of the keys. Compared to our DJA approach, the learned VPA approximates the original JSON Schema, and may take several days to learn the automaton for a single schema, making the approach difficult to use in practice.

Several works [4, 13, 44] are concerned with formalizing the ***JSON Schema*** [27] language. Pezoa et al. [44] formalize the syntax and semantics of JSON Schema. They also show that the validation problem is PTIME-complete. If the schema does not contain the `uniqueItems` keyword, schema validation can be performed in time $O(m \cdot n)$, where m is the size of the schema and n is the size of the JSON document. Bourhis et al. [13] consider the notion of well-formedness for recursive schemas to prevent infinite recursion. Attouche et al. [4] present a formal description of newer versions of JSON Schema, where dynamic references are used to refine a data structure. They show for "modern" JSON Schema (draft 2019-09) that dynamic references make the validation problem PSPACE-complete. Habib et al. [24] study the problem of subschema checking to find bugs between versions of a given JSON schema. They present a procedure to canonicalize JSON schema into a simpler language over which the subschema check is performed.

The incremental validation problem [6, 8] is about validating an XML document after an update is done (insertion, deletion, or renaming), a problem that is highly relevant to databases.

7 LIMITATIONS AND CONCLUSION

Limitations & Future Work. Our DJA-based tool supports all schemas that can be written using our schema language *JSL*, which encompasses the core features of JSON Schema. It does not, however, fully support JSON Schema yet. In future work, we plan to include support for more features. We will support a feature corresponding to the `prefixItems` keyword that is used to constrain the elements of a finite prefix of an array. For example,

```
"prefixItems": [{"type": "string"}, {"type": "number"}]
```

requires that the first element is a string, and the second element is a number. We can extend *JSL* with a construct $\mathsf{Arr}(s_0, \ldots, s_{n-1}; s)$, where $s_0, \ldots, s_{n-1}$ are schemas constraining the first n elements of an array. *Cardinality constraints* for arrays and objects can be easily handled by maintaining (on the stack) the current size of an object or an array. *Pattern properties* and *predicates* over primitive values (e.g., numbers and strings) can be handled when the JSON tokens are generated (by enriching tokens with additional relevant information). Regular expressions ("regexes") are used with the keyword `patternProperties` and with string patterns (keyword `pattern`). Simple regexes could be integrated with the tokenizer for efficiency (see, e.g., [35]), but more challenging regexes could potentially require specialized algorithms for efficient matching (see, e.g., [30, 34] for bounded repetition, [17, 37] for lookaround assertions, [38] for the semantic issues of disambiguation, and [42] for backreferences). Another feature that we plan on implementing (keyword `contains`) requires that at least one array element satisfies a specified schema. It can be implemented with a DJA by extending the array stack entry $\mathtt{Arr}(n)$ with a Boolean value indicating whether an element satisfying s has been encountered.

Conclusion. We have investigated the problem of streaming schema validation for large JSON documents. We have proposed a model of JSON automata called DJA for the streaming validation of JSON schemas. We have shown that our DJA-based implementation greatly reduces the memory footprint for schema validation while maintaining competitive throughput compared to the existing tools.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive comments. This research was supported in part by the US National Science Foundation award CCF 2340479.

REFERENCES

[1] AJV 2024. AJV. Available at https://ajv.js.org/. [Online; Accessed 03 Dec, 2025].

[2] Rajeev Alur, Viraj Kumar, Parthasarathy Madhusudan, and Mahesh Viswanathan. 2005. Congruences for Visibly Pushdown Languages. In *Automata, Languages and Programming (ICALP 2005) (LNCS)*, Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung (Eds.), Vol. 3580. Springer, Berlin, Heidelberg, 1102–1114. https://doi.org/10.1007/11523468_89

[3] Rajeev Alur and Parthasarathy Madhusudan. 2004. Visibly Pushdown Languages. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing (STOC '04)*. ACM, New York, NY, USA, 202–211. https://doi.org/10.1145/1007352.1007390

[4] Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2024. Validation of Modern JSON Schema: Formalization and Complexity. *Proceedings of the ACM on Programming Languages* 8, POPL, Article 49 (2024), 31 pages. https://doi.org/10.1145/3632891

[5] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2021. A JSON Schema Corpus. https://github.com/sdbs-uni-p/json-schema-corpus.

[6] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. 2004. Incremental Validation of XML Documents. *ACM Transactions on Database Systems* 29, 4 (2004), 710–751. https://doi.org/10.1145/1042046.1042050

[7] Vince Bárány, Christof Löding, and Olivier Serre. 2006. Regularity Problems for Visibly Pushdown Languages. In *Annual Symposium on Theoretical Aspects of Computer Science (STACS 2006) (LNCS)*, Bruno Durand and Wolfgang Thomas (Eds.), Vol. 3884. Springer, Berlin, Heidelberg, 420–431. https://doi.org/10.1007/11672142_34

[8] Denilson Barbosa, Alberto O. Mendelzon, Leonid Libkin, Laurent Mignet, and Marcelo Arenas. 2004. Efficient Incremental Validation of XML Documents. In *Proceedings of the 20th International Conference on Data Engineering (ICDE 2004)*. IEEE, USA, 671–682. https://doi.org/10.1109/ICDE.2004.1320036

[9] Corentin Barloy, Filip Murlak, and Charles Paperman. 2021. Stackless Processing of Streamed Trees. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2021)*. ACM, New York, NY, USA, 109–125. https://doi.org/10.1145/3452021.3458320

[10] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. 2004. DTDs Versus XML Schema: A Practical Study. In *Proceedings of the 7th International Workshop on the Web and Databases: Colocated with ACM SIGMOD/PODS 2004 (WebDB '04)*. ACM, New York, NY, USA, 79–84. https://doi.org/10.1145/1017074.1017095

[11] Ronald Book, Shimon Even, Sheila Greibach, and Gene Ott. 1971. Ambiguity in Graphs and Expressions. *IEEE Trans. Comput.* C-20, 2 (1971), 149–153. https://doi.org/10.1109/T-C.1971.223204

[12] Boon 2024. Boon. Available at https://github.com/santhosh-tekuri/boon. [Online; Accessed 03 Dec, 2025].

[13] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoč. 2017. 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)*. ACM, New York, NY, USA, 123–135. https://doi.org/10.1145/3034786.3056120

[14] Pierre Bourhis, Juan L. Reutter, and Domagoj Vrgoč. 2020. JSON: Data Model and Query Languages. *Information Systems* 89 (2020), 101478. https://doi.org/10.1016/j.is.2019.101478

[15] Anne Brüggemann-Klein and Derick Wood. 1998. One-Unambiguous Regular Languages. *Information and Computation* 140, 2 (1998), 229–253. https://doi.org/10.1006/inco.1997.2688

[16] Véronique Bruyère, Guillermo A. Pérez, and Gaëtan Staquet. 2023. Validating Streaming JSON Documents with Learned VPAs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2023) (LNCS)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.), Vol. 13993. Springer, Cham, 271–289. https://doi.org/10.1007/978-3-031-30823-9_14

[17] Agnishom Chattopadhyay, Angela W. Li, and Konstantinos Mamouras. 2025. Verified and Efficient Matching of Regular Expressions with Lookaround. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '25)*. ACM, New York, NY, USA, 198–213. https://doi.org/10.1145/3703595.3705884

[18] Cristiana Chitic and Daniela Rosu. 2004. On Validation of XML Streams Using Finite State Machines. In *Proceedings of the 7th International Workshop on the Web and Databases: Colocated with ACM SIGMOD/PODS 2004 (WebDB '04)*. ACM, New York, NY, USA, 85–90. https://doi.org/10.1145/1017074.1017096

[19] Coreutils 2025. Coreutils - GNU core utilities. Available at https://www.gnu.org/software/coreutils/. [Online; Accessed 03 Dec, 2025].

[20] Loris D'Antoni and Rajeev Alur. 2014. Symbolic Visibly Pushdown Automata. In *Computer Aided Verification (CAV 2014) (LNCS)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, Cham, 209–225. https://doi.org/10.1007/978-3-319-08867-9_14

[21] DTD 1998. Document Type Definitions. Available at https://www.w3.org/XML/1998/06/xmlspec-report-v20.htm. [Online; Accessed 03 Dec, 2025].

[22] Patrick C. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. 1968. Counter Machines and Counter Languages. *Mathematical Systems Theory* 2, 3 (1968), 265–283. https://doi.org/10.1007/BF01694011

[23] Sheila A. Greibach. 1969. An Infinite Hierarchy of Context-Free Languages. *J. ACM* 16, 1 (1969), 91–106. https://doi.org/10.1145/321495.321503

[24] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2021. Finding Data Compatibility Bugs with JSON Subschema Checking. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. ACM, New York, NY, USA, 620–632. https://doi.org/10.1145/3460319.3464796

[25] Malte Isberner. 2015. *Foundations of Active Automata Learning: An Algorithmic Perspective*. Ph.D. Dissertation. Technical University Dortmund, Germany. https://doi.org/10.17877/DE290R-16359

[26] JSON 2017. JSON. Available at https://www.rfc-editor.org/info/rfc8259. [Online; Accessed 03 Dec, 2025].

[27] JsonSchema 2022. JSON Schema. Available at https://json-schema.org/draft/2020-12/json-schema-core. [Online; Accessed 03 Dec, 2025].

[28] JsonSchema-Rust 2024. JsonSchema-Rust. Available at https://github.com/Stranger6667/jsonschema. [Online; Accessed 03 Dec, 2025].

[29] Christoph Koch and Stefanie Scherzinger. 2007. Attribute Grammars for Scalable Query Processing on XML Streams. *The VLDB Journal* 16, 3 (2007), 317–342. https://doi.org/10.1007/s00778-005-0169-1

[30] Lingkun Kong, Qixuan Yu, Agnishom Chattopadhyay, Alexis Le Glaunec, Yi Huang, Konstantinos Mamouras, and Kaiyuan Yang. 2022. Software-Hardware Codesign for Efficient In-Memory Regular Pattern Matching. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. ACM, New York, NY, USA, 733–748. https://doi.org/10.1145/3519939.3523456

[31] Christian Konrad and Frédéric Magniez. 2013. Validating XML Documents in the Streaming Model with External Memory. *ACM Transactions on Database Systems* 38, 4, Article 27 (2013), 36 pages. https://doi.org/10.1145/2504590

[32] Viraj Kumar, Parthasarathy Madhusudan, and Mahesh Viswanathan. 2007. Visibly Pushdown Automata for Streaming XML. In *Proceedings of the 16th International Conference on World Wide Web (WWW 2007)*. ACM, New York, NY, USA, 1053–1062. https://doi.org/10.1145/1242572.1242714

[33] Geoff Langdale and Daniel Lemire. 2019. Parsing Gigabytes of JSON per Second. *The VLDB Journal* 28, 6 (2019), 941–960. https://doi.org/10.1007/s00778-019-00578-5

[34] Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2023. Regular Expression Matching Using Bit Vector Automata. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1, Article 92 (2023), 30 pages. https://doi.org/10.1145/3586044

[35] Angela W. Li and Konstantinos Mamouras. 2025. Efficient Algorithms for the Uniform Tokenization Problem. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1, Article 133 (April 2025), 27 pages. https://doi.org/10.1145/3720498

[36] Frédéric Magniez, Claire Mathieu, and Ashwin Nayak. 2010. Recognizing Well-Parenthesized Expressions in the Streaming Model. In *Proceedings of the Forty-Second ACM Symposium on Theory of Computing (STOC '10)*. ACM, New York, NY, USA, 261–270. https://doi.org/10.1145/1806689.1806727

[37] Konstantinos Mamouras and Agnishom Chattopadhyay. 2024. Efficient Matching of Regular Expressions with Lookaround Assertions. *Proceedings of the ACM on Programming Languages* 8, POPL, Article 92 (2024), 31 pages. https://doi.org/10.1145/3632934

[38] Konstantinos Mamouras, Alexis Le Glaunec, Wu Angela Li, and Agnishom Chattopadhyay. 2024. Static Analysis for Checking the Disambiguation Robustness of Regular Expressions. *Proceedings of the ACM on Programming Languages* 8, PLDI, Article 231 (2024), 25 pages. https://doi.org/10.1145/3656461

[39] MongoDB 2024. MongoDB. Available at https://www.mongodb.com/docs/manual/reference/operator/query/jsonSchema/. [Online; Accessed 03 Dec, 2025].

[40] Filip Murlak, Charles Paperman, and Michał Pilipczuk. 2016. Schema Validation via Streaming Circuits. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '16)*. ACM, New York, NY, USA, 237–249. https://doi.org/10.1145/2902251.2902299

[41] MySQL 2025. MySQL. Available at https://dev.mysql.com/doc/refman/8.4/en/json-validation-functions.html. [Online; Accessed 03 Dec, 2025].

[42] Taisei Nogami and Tachio Terauchi. 2025. Efficient Matching of Some Fundamental Regular Expressions with Backreferences. In *50th International Symposium on Mathematical Foundations of Computer Science (MFCS 2025) (Leibniz International Proceedings in Informatics (LIPIcs))*, Paweł Gawrychowski, Filip Mazowiecki, and Michał Skrzypczak (Eds.), Vol. 345. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 81:1–81:19. https://doi.org/10.4230/LIPIcs.MFCS.2025.81

[43] Oracle 2025. Oracle. Available at https://docs.oracle.com/en/database/oracle/oracle-database/23/adjsn/oracle-database-support-json.html/. [Online; Accessed 03 Dec, 2025].

[44] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. 2016. Foundations of JSON Schema. In *Proceedings of the 25th International Conference on World Wide Web (WWW '16)*. International World Wide Web

Conferences Steering Committee, Republic and Canton of Geneva, CHE, 263–273. https://doi.org/10.1145/2872427.2883029

[45] Postman 2024. Postman. Available at https://json-schema.org/blog/posts/postman-case-study. [Online; Accessed 03 Dec, 2025].

[46] RxDB 2024. RxDB. Available at https://rxdb.info/. [Online; Accessed 03 Dec, 2025].

[47] Luc Segoufin and Cristina Sirangelo. 2006. Constant-Memory Validation of Streaming XML Documents Against DTDs. In *Database Theory – ICDT 2007 (LNCS)*, Thomas Schwentick and Dan Suciu (Eds.), Vol. 4353. Springer, Berlin, Heidelberg, 299–313. https://doi.org/10.1007/11965893_21

[48] Luc Segoufin and Victor Vianu. 2002. Validating Streaming XML Documents. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '02)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/543613.543622

[49] SIMD-JSON 2025. simdjson : Parsing gigabytes of JSON per second. Available at https://github.com/simdjson/simdjson/. [Online; Accessed 03 Dec, 2025].

[50] Gaëtan Staquet. 2022. *Validating Streaming JSON Documents With Learned VPAs*. https://doi.org/10.5281/zenodo.7309690

[51] Nguyen Van Tang. 2009. A Tighter Bound for the Determinization of Visibly Pushdown Automata. In *International Workshop on Verification of Infinite-State Systems (INFINITY 2009) (Electronic Proceedings in Theoretical Computer Science)*, Vol. 10. Open Publishing Association, 62–76. https://doi.org/10.4204/eptcs.10.5

[52] Juan Cruz Viotti and Mital Kinderkhedia. 2022. Benchmarking JSON BinPack. *arXiv preprint arXiv:2211.12799* (2022).

[53] VscodeSnippet 2025. Schema for Visual Studio Code snippets. Available at https://raw.githubusercontent.com/Yash-Singh1/vscode-snippets-json-schema/main/schema.json. [Online; Accessed 03 Dec, 2025].

[54] Philipp Wehner, Christina Piberger, and Diana Göhringer. 2014. Using JSON to Manage Communication Between Services in the Internet of Things. In *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, USA, 1–4. https://doi.org/10.1109/ReCoSoC.2014.6861361

[55] Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA.

[56] XSD 2012. XML Schema. Available at https://www.w3.org/TR/xmlschema11-1. [Online; Accessed 03 Dec, 2025].