A Tree Based Visualizable Method for JSON Schema Inference from JSON Data.

Prabhleen Kaur Saini  
Department of Computer Science  
University of Texas at Dallas

Sai Prasad Matam  
Enabler Inc.

Bhadrachalam Chitturi  
Department of Computer Science  
University of Texas at Dallas

I. INTRODUCTION

JSON, a lightweight data interchange format, has gained widespread adoption due to its human-readable nature and flexibility, making it compatible with various programming languages. With over 40% of websites utilizing JSON and nearly 95% of APIs exchanging data in JSON format, its prevalence underscores its importance in modern web development.[7]

JSON’s schemaless nature, while advantageous for rapid application deployment and resilience to data irregularities, presents challenges in detecting structural mismatches between expected and actual data. This limitation becomes particularly burdensome in scenarios requiring data integration, where understanding and managing JSON documents from multiple sources can pose significant hurdles[8]. It can also become a burden in data integration scenarios (e.g., consuming JSON-based APIs) where it becomes necessary to discover at least partially the underlying structure in order to properly process the data[9]. Therefore, web developers must often interact with APIs publishing a set of JSON based services and face the problems of understanding and managing the JSON documents returned by those services. The problem gets worse when developers need to compose several JSON-based services since their implicit structure can differ. For instance, digesting the data returned by a query service to call another service later on. There already exist several schema inference approaches. The inference process itself is nontrivial, and the resulting schemas often suffer from various issues. For example, derived entities may contain a large number of properties, including properties of the same name having different data types, as well as various kinds of references between the documents (aggregates).

In this study, we explore various approaches and algorithms for schema inference from JSON data, highlighting their strengths, limitations, and areas for improvement. We delve into the literature to review existing methodologies proposed by researchers, analyzing their effectiveness in addressing the complexities inherent in JSON schema inference. Additionally, we present our own algorithm, which addresses specialized scenarios that previous methods may not have adequately handled. It operates in two stages: generating a tree representation of the data and converting it into a JSON schema. We provide a detailed analysis of the algorithm’s time complexity and showcase its performance through examples and results.

A. Challenges of Schema Inference

Schema inference from JSON data poses several challenges due to the dynamic nature of JSON documents and the diversity of data structures they can represent. Following examples highlight challenges with examples:

1) Optional Properties

In above examples user reviews for some product are given, to extract json schema accurately we need to represent that rating is optional as it is not present in all reviews.

Handling optional properties presents a challenge in schema inference as JSON documents may contain properties that are not mandatory across all instances. Inferring whether a property is required or optional requires careful analysis of the data.

2) Union Types

Union types, where a property can have multiple possible data types, complicate schema inference.
The phone_number property is represented as both a string and an integer across different documents. Inferring a union type to accommodate both variations is essential for accurate schema representation. Determining the correct type for a property becomes challenging when it can vary across different instances.

3) References
In a JSON schema, a $ref keyword is a JSON Pointer to a schema, or a type or property in a schema.

```json
{  
  "address": {  
    "type": "object",  
    "properties": {  
      "street": {  
        "type": "string"  
      },  
      "city": {  
        "type": "string"  
      }  
    },  
    "required": [  
      "street",  
      "city"  
    ]  
  },  
  "person": {  
    "type": "object",  
    "properties": {  
      "name": {  
        "type": "string"  
      },  
      "address": {  
        "$ref": "#/definitions/address"  
      }  
    }  
  }
}
```

In the above example the person schema includes a reference ($ref) to the address schema, indicating that a person object should contain an address following the structure defined in the address schema.

Inferring references between JSON documents poses a challenge as it requires identifying patterns such as naming conventions or explicit references within the data.

4) Complex Data Types - Hash, Sets, and Tuples

```json
{  
  "person_1": {  
    "name": "John Doe",  
    "age": 30  
  },  
  "person_2": {  
    "name": "Jane Smith",  
    "age": 25  
  }
}
```

In this example, each key (person_1, person_2) in the map represents a unique identifier for a person, but it is not part of the schematic information or metadata. Maps, sometimes also called dictionaries, are similar in structure to JSON objects but semantically different, because the key of the set is not a part of the schematic information (metadata), but it is part of actual data [6]. Schema inference faces challenges in handling complex data types like hash tables, sets, and tuples within JSON documents.

5) Extended or User-defined Data Types

```json
Example Data
[1600, "Dallas", "Avenue", "NW"]
[1600, "Dallas", "Avenue"]

Schema
{  
  "type": "array",  
  "prefixItems": [  
    { "type": "number" },  
    { "type": "string" },  
    { "enum": ["Street", "Avenue"]},  
    { "enum": ["NW", "NE", "SW"] }  
]
```

Supporting extended or user-defined data types adds complexity to schema inference as it requires recognizing custom data structures and incorporating them into the inferred schema.

Addressing these challenges effectively is crucial for accurate and comprehensive schema inference from JSON data.

II. Literature Review
The approach proposed by Sevilla et al. employs a three-step methodology utilizing MapReduce [1]. Firstly, it reduces a collection of JSON documents into structurally distinct ones. Following this reduction, the approach aims to identify various versions of entities and their corresponding properties. Finally, it focuses on the identification of relationships within the data, including references between entities. The process is designed to provide insights into the structure and evolution of the dataset. Notably, the authors emphasize the significance of versioned schemas in understanding the dynamic nature of the underlying data. However, it does not handle empty arrays correctly.

Klettke et al. present a schema inference algorithm tailored for MongoDB, leveraging a Structure Identification Graph [2].
In this method, nodes in the graph correspond to JSON properties, while edges model the hierarchical structure of the data. The algorithm includes the representation of metadata through nodeList and edgeList components. The SIG-based approach aims to capture the inherent hierarchical relationships within the JSON documents.

Cánovas et al. [3] propose an iterative process tailored for web services dealing with collections of JSON documents. The approach unfolds in three main stages: first, the extraction of a schema for each individual document; second, the creation of a schema for each collection; and finally, the merging of individual collection schemas into a unified schema. Despite its effectiveness, this approach comes with certain drawbacks, including the absence of reference detection, limited recognition of union types, optional properties, arrays, and extended data types.

Frozza et al. contribute an approach focused on inferring the schema for a single collection of JSON documents [4]. The process involves four key steps: the creation of raw schemas for individual documents, grouping of identical raw schemas, unification of schemas, and the construction of the final global JSON schema. While similar in the goal of schema merging, this approach is tailored specifically for a single collection.

Bazizzi et al. [5] introduce a two-phase inference algorithm based on Apache Spark. In this approach, the Map function processes JSON documents during the Reduce phase to identify union types, required, optional, and repeated elements. The algorithm aims to enhance the type reduction process, providing a more nuanced understanding of the data.

The comprehensive work by Veinhardt and Koupil [6], in which they compare the above inference algorithms, highlights the following challenges. A notable issue is scalability; despite many approaches being designed to scale well, the lack of parallelization is a common limitation. Technologies like MapReduce and Apache Spark are often used to address this. There are also challenges with handling complex data types like maps and tuples, requiring new approaches. Improving how entity relationships are modeled is crucial for advancing schema inference methods. The existing JSON Schema specification lacks support for complex integrity constraints, suggesting the potential use of an Object Constraints Language for a more sophisticated constraint definition. Multi-model schema inference faces hurdles like system-dependent data retrieval, calling for universal algorithms, a unified inference process, and careful consideration of how the output schema is represented for compatibility across different scenarios.

### III. DATA PREPARATION

In this section, we describe the process of data preparation. We extended mock JSON data and corresponding schemas obtained from the experiment conducted by Ivan Veinhardt Lattak and Pavel Koupil [6]. The mock data encompassed various types of JSON structures, including:

- Primitive Types
- Simple Arrays
- Simple Objects
- Nested Arrays
- Nested Objects
- Optional Values
- Nullable Values
- Dynamic Data Types
- References

To facilitate comprehension, we developed a framework capable of displaying both the data and the corresponding JSON schema. Furthermore, we intend to extend this framework to highlight discrepancies between the generated schema and the actual schema.

### IV. ALGORITHM

The algorithm operates in two stages. In the first stage, it constructs a tree structure representing the data. In the second stage, it converts this tree into a JSON schema.

#### A. Generating Tree Representation

The first step in our algorithm involves constructing a tree representation of the input data. If the input data is a dictionary, we iterate through each key-value pair. For each pair, we recursively generate a tree for the corresponding value and add it as a child to the current node, with the key serving as its label. Additionally, we record all keys in the properties as required fields. Similarly, if the input data is a list, we iterate through each element, generating a tree for each element and adding it as a child with its index as the label. Ultimately, this step yields a tree structure that encapsulates the data's hierarchical organization.

**1) TreeNode Class:** The algorithm utilizes a custom TreeNode class to represent the nodes of the tree. Each node contains the following attributes:

- parent: The parent node of the current node.
- label: The label of the current node.
- typeLabel: The data type of the current node.
- children: A dictionary containing the child nodes of the current node.
- props: A dictionary containing properties of the current node, such as required keys and possible data types.

**2) Comparing Nodes:** Once we have the tree representation of the data, we proceed to compare nodes with corresponding elements in the input data. This comparison is crucial for updating the tree structure to accurately reflect the data's characteristics. If there is no existing node for an element, we generate a tree for the element and return it. For dictionaries, we ensure that keys and their associated values are appropriately matched and updated within the tree. Similarly, for lists, we compare elements with existing nodes and update the tree accordingly. Throughout this process, we update node properties to reflect any changes or additions, such as required keys and allowable data types.

#### B. Visualization of Tree

After constructing the tree representation of the data, it is often helpful to visualize the hierarchical structure for better understanding. We utilize the Graphviz library to generate...
A. generateTree(data) Function

The function construct a tree representation of the input data. Its time complexity depends on the size and structure of the input data.

- For a dictionary input, the function iterates over each key-value pair, resulting in a time complexity of $O(m)$. If an element is nested, it calls generateTree recursively, so (tree has max height $h$ which also denotes how deeply the input data is nested) then resulting in a time complexity will be $O(m \cdot h)$.

Overall, the time complexity of the generateTree function is $O(m \cdot h)$.

B. compare(element, node) Function

The function compares nodes with elements and updates the tree accordingly. Its time complexity also depends on the size and structure of the input data. Where $m$ is the maximum number of nodes at each height and $h$ denote the height of the tree.

- In the worst case, when the input data is a dictionary and each key-value pair is new, the function may traverse the entire tree to find matching nodes for each element of input data, resulting in a time complexity of $O(m \cdot h)$.
- If the input data is a list and each element is new, the function may traverse the entire tree to find matching nodes for each element of input data, resulting in a time complexity of $O(m \cdot h)$.

Overall, the time complexity of the compare function is $O(m \cdot h)$.

C. data2Tree(data) Function

The data2Tree function converts input data to a tree representation. Its time complexity depends on the size and structure of the input data.

- For a list input, the function iterates over each element and calls the compare function, resulting in a time complexity of $O(n \cdot T(\text{compare}))$.
- For a dictionary input, the function calls the compare function once, resulting in a time complexity of $O(n \cdot T(\text{compare}))$.

Overall, the time complexity of the data2Tree function is $O(n \cdot m \cdot h)$.

D. schemaGenerator(mode) Function

The schemaGenerator function generates JSON schema from the tree representation. Its time complexity depends on the size and structure of the tree. Let $n$ denote the number of nodes in the tree.

- The function traverses each node in the tree once, resulting in a time complexity of $O(n)$.

Overall, the time complexity of the schemaGenerator function is $O(n)$.

E. Time complexity of the algorithm

Given an input with $n$ elements and $m$ is the maximum number of nodes at each height, $h$ is the height of the tree, the overall time complexity will be:

$$T(n) = T(\text{generateTree(data)}) + T(\text{compare(element, node)}) + T(\text{data2Tree(data)}) + T(\text{schemaGenerator(mode)})$$

$$= O(m \cdot h) + O(m \cdot h) + O(n \cdot m \cdot h) + O(n)$$

$$= O(n \cdot m \cdot h)$$

VI. CONCLUSION AND FUTURE SCOPE

Our algorithm efficiently handles various JSON data types and infers JSON schema. We represent the first element as a tree and then refine this tree as we process subsequent elements and finally generate a JSON schema. Thus, one can visualize the entire process. However, there are avenues for further improvement; e.g. handling of the references and user-defined data types. Additionally, accommodating language-specific constructs presents challenges.

In summary, while our algorithm represents progress in JSON schema inference, ongoing refinement and expansion are necessary. By addressing these challenges, we strive to improve the accuracy, efficiency, and versatility of schema inference from JSON data.
VII. RESULTS

A. Primitive Data Types

Data (JSON):

```
[{
  "_id": 1,
  "timestamp": "2021-02-06T16:31:32.029Z",
  "published": true
},
{
  "_id": 2,
  "timestamp": "2021-02-10T18:02:29.706Z",
  "published": false
},
{
  "_id": 3,
  "timestamp": "2021-04-10T04:19:45.489Z",
  "published": true
},
{
  "_id": 4,
  "timestamp": "2021-06-01T16:21:06.187Z",
  "published": true
},
{
  "_id": 5,
  "timestamp": "2021-01-24T11:53:18.245Z",
  "published": false
}
]
```

Schema (JSON):

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Json Schema Inference",
  "description": "Schema for given list of data items",
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "_id": {
        "type": "int"
      },
      "timestamp": {
        "type": "str"
      },
      "published": {
        "type": "bool"
      }
    }
  },
  "required": [
    "timestamp",
    "published",
    "_id"
  ]
}
```
B. Object Input

Data (JSON):

```json
{
  "_id": 1,
  "author": {
    "first_name": "John",
    "last_name": "Doe",
    "phone_number": "518-555-0168",
    "location": {
      "latitude": "+48.875000",
      "longitude": "+123.393333"
    }
  }
}
```

Schema (JSON):

```json
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Json Schema Inference",
  "type": "object",
  "properties": {
    "_id": {
      "type": "int"
    },
    "author": {
      "type": "object",
      "properties": {
        "first_name": {
          "type": "str"
        },
        "last_name": {
          "type": "str"
        },
        "phone_number": {
          "type": "str"
        },
        "location": {
          "type": "object",
          "properties": {
            "latitude": {
              "type": "str"
            },
            "longitude": {
              "type": "str"
            }
          }
        }
      }
    },
    "required": [
      "latitude",
      "longitude"
    ]
  },
  "required": [
    "_id",
    "author"
  ]
}
```
C. Multiple Inputs - Array of Objects

Data (JSON):

```json
[  
  {  
    "_id": 1,  
    "attachments": [  
      {  
        "url": "/image.png"  
      },  
      {  
        "url": "/document.pdf"  
      }  
    ]  
  },  
  "nested_arrays": [  
    [  
      0  
    ]  
  ],  
  {  
    "_id": "2",  
    "attachments": [  
      {  
        "url": "/image.png"  
      },  
      {  
        "url": "/document.pdf",  
        "views": 1000  
      }  
    ],  
    "nested_arrays": [  
      [  
        "0"  
      ]  
    ]  
  }]
```

Schema (JSON):

```json
{"$schema": "https://json-schema.org/draft/2020-12/schema",  
"title": "Json Schema Inference",  
"type": "array",  
"items": {  
  "type": "object",  
  "properties": {  
    "_id": {  
      "anyOf": [  
        {"type": "int"},  
        {"type": "str"}  
      ]  
    },  
    "attachments": {  
      "type": "array",  
      "items": {  
        "type": "object",  
        "properties": {  
          "url": { "type": "str"}  
        }  
      }  
    },  
    "nested_arrays": {  
      "type": "array",  
      "items": {  
        "type": "array",  
        "items": {  
          "anyOf": [  
            {"type": "int"},  
            {"type": "str"}  
          ]  
        }  
      }  
    }  
  }  
},  
"required": [  
  "attachments",  
  "nested_arrays",  
  "_id"
]}
```
D. Union Types

Data (JSON):

```json
[
  {
    "user_id": "U001",
    "name": "John Doe",
    "phone_number": "1234567890"
  },
  {
    "user_id": "U002",
    "name": "Jane Smith",
    "phone_number": 9876543210
  }
]
```

Schema (JSON):

```json
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Json Schema Inference",
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "user_id": {
        "type": "str"
      },
      "name": {
        "type": "str"
      },
      "phone_number": {
        "anyOf": [
          {
            "type": "int"
          },
          {
            "type": "str"
          }
        ]
      }
    }
  },
  "required": ["name", "user_id", "phone_number"]
}
```
E. Null Data

Data (JSON):

```json
[{
  "_id": 1,
  "timestamp": "2021-06-01T16:21:06.187Z",
  "published": true
},
{
  "_id": 2,
  "timestamp": "2021-06-01T16:21:06.187Z",
  "published": false
},
{
  "_id": 3,
  "timestamp": null,
  "published": null
}
]
```

Schema (JSON):

```json
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Json Schema Inference",
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "_id": {
        "type": "int"
      },
      "timestamp": {
        "anyOf": [
          {
            "type": "NoneType"
          },
          {
            "type": "str"
          }
        ]
      },
      "published": {
        "anyOf": [
          {
            "type": "NoneType"
          },
          {
            "type": "bool"
          }
        ]
      }
    },
    "required": ["_id"
  ]
}
```
ACKNOWLEDGMENT

First and foremost, I want to thank Professor Chitturi for giving me the opportunity to work on this project and for introducing me to our collaborator Mr. Sai Matam. I thank Professor Chitturi for his constant support, feedback, and for providing all the necessary resources, enabling me to maximize my learning from this project.

I would also like to thank Mr. Sai Prasad Matam for suggesting this particular topic and for his invaluable guidance, support, and mentorship throughout the course of this study. His expertise and insightful feedback have greatly enriched my understanding and shaped the direction of my work.

REFERENCES


