Journal of Information and Data Management, 2026, 17:1, doi: 10.5753/jidm.2026.5748

JFUSE: Json FUll Schema Extractor

Natália Banhara [**Federal University of Fronteira Sul** | ***natalia.banhara@outlook.com***]
Denio Duarte [**Federal University of Fronteira Sul** | ***duarte@uffs.edu.br***]
Geomar A Schreiner [**Federal University of Fronteira Sul** | ***gschreiner@uffs.edu.br***]
Samuel Feitosa [**Federal University of Fronteira Sul** | ***samuel.feitosa@uffs.edu.br***]

Universidade Federal da Fronteira Sul, Rodovia SC 484 - Km 02, Fronteira Sul, CEP 89815-899, Chapecó, SC, Brazil.

Received: 29 June 2025 • **Published:** 13 March 2026

Abstract Lately, we have witnessed a flood of data generated by several data-centric applications, and the generated data are available in a large fashion of formats. However, those data are mostly weakly structured, irregular, and incomplete; they do not follow a predefined schema. A challenging task is to understand how those data are organized and structured. JSON has become a trendy format for data-centric applications to store and share data. Its success is due to embodying structure and data in the same representation. This makes JSON documents loosely coupled with schemas. Still, schemas are essential for applications to deal with the data more efficiently. In this paper, we propose JFUSE, a tool to deal with the problem of discovering a schema from JSON collections. Besides inferring basic types (e.g., atomic types, arrays, and objects), JFUSE also discovers fields that represent keys in the collection, fields minimum/maximum constraint values, enumeration, tagged unions, metadata as data, objects as collections, and arrays as tuples. We propose a metamodel that can be easily transformed into any schema language (e.g., JSON Schema). Our experiments show that the proposed approach infers concise and correct schemas from (huge) JSON collections.

Keywords: Schema Discovering, JSON, Metamodel, tagged union, enumeration

1 Introduction

Lately, we witness a flood of data generated by several data-centric applications. Some areas of science, for example, are facing a huge increase of data volumes from satellites, telescopes, high-throughput instruments, sensor networks, accelerators, and supercomputers [Bell *et al*., 2009]. The generated data are mostly weakly structured, irregular, and incomplete [Kellou-Menouer *et al*., 2022]; additionally, they do not follow a predefined schema. To store and exchange this kind of data, JSON has become one of the most popular formats [Bourhis *et al*., 2017a]. JSON is a lightweight format and its documents are collections of key-valued pairs, i.e., it stores data (value) and metadata (key) altogether in a given document. This structure makes JSON documents be loosely coupled with schemas. On the other hand, applications that need to access this kind of data must have some reliable notion of its schema.

Schemas are essential for data-centric applications, and Kellou-Menouer *et al*. [2022] show some tasks where schemas can be useful:

- *Source Selection*: it allows to choose the best data source for what users are looking for.
- *Query Formulation*: schemas describe the content of a data source providing a better overview of the content to be queried. This is also true when a query must be decomposed or optimized. Knowing the document structure makes easier to query multiple data sources to find the best query decomposition.
- *Data Indexing*: the schema of a data source $\mathcal{D}$ may be a guide to built indexes on $\mathcal{D}$.
- *Query Answering*: the schema of a data source $\mathcal{D}$ allows to detemine whether or not a subset of $\mathcal{D}$ contains the answer of a given query.
- *Data integration*: the use of schema information to combine data sources is essential to achieve better results.
- *Data Quality Assessment*: schemas allow to use metrics to evaluate the quality of a data source, e.g., the completeness of a dataset reagarding its schema.
- *Data Partitioning*: when data must be distributed in multiple nodes, a schema enables a better partitioning plan.

Given a collection of JSON documents, extracting the latent schema presents a challenge due to its inherently flexible and dynamic nature [Cánovas Izquierdo and Cabot, 2013]. JSON allows nested objects, arrays, and various structures with no enforced schema, making it difficult to infer a consistent and accurate schema. These complexities and the advantages of using a schema create a necessity for schema extraction approaches.

Many schema extraction approaches have been proposed based on the importance of a schema [Frozza *et al*., 2018; Baazizi *et al*., 2019; Abdelhedi *et al*., 2021; Klessinger *et al*., 2023]. Each one explores the schema extraction on different facets of the JSON with distinct approaches. All approaches extract the basic JSON types (string, number, boolean, object, and array) but differ on the other JSON features (*e.g.* tagged union, and enumerations). The works of Frozza *et al*. [2018], Baazizi *et al*. [2019], and Abdelhedi *et al*. [2021] focus only on the extraction of the basic types. Namba [2021], Spoth *et al*. [2021], and Yun *et al*. [2024] propose complete

tools; however, they cannot discover tagged unions and enumerations. Although Klessinger *et al*. [2023] include tagged unions in the target schema, other types are not considered. Despite schema extraction being addressed in the state of the art, we are unaware of any approach that deals with the JSON structure complexity.

In this paper, we present *JFUSE* (Json FUll Schema Extractor), a novel approach for schema extraction. Our approach maps the JSON fields to vertices and uses edges to map relationships between them (*e.g.*, parenting or sibling). In the graph, we store information about the occurrence of each field and their relationship to facilitate the inference of the schema. Based on the graph, we generate a metamodel representing the schema rules. Our approach can extract information about the basic JSON data types and features like tagged unions, enumeration, data collections, and tuples encoded in arrays. It extends the work presented in [Banhara *et al*., 2024]. The new contributions are as follows: (i) two new constraints are added: keys (document identifiers); and minimum and maximum values, and (ii) new experiments are conducted.

We validate JFUSE executing two sets of experiments. The first one is to validate the approach against real-world datasets, evaluating the quality of the extracted schema. The second experiment focuses on proving the approach's concept, testing the extraction against a synthetic dataset created based on the different facets of a JSON (basic types, tagged union, array as tuple, metadata, object as collection, enumeration, keys, and minimum and maximum values). The results show a concise schema regarding the size of the input collections and a satisfactory execution time. Moreover, the experiments also showed that our approach is scalable.

The rest of this paper is organized as follows: Section 2 reviews the JSON data model and the main schema concepts. Section 3 presents the related work, highlighting the limitations of the existing approaches. Section 4 details our graph-based methodology, including the definitions of the metamodel. Section 5 presents the experimental results and discussion. Finally, Section 6 concludes the paper.

2 JSON and JSON Schema

JSON (JavaScript Object Notation) is a lightweight data interchange format commonly used in modern web development and transmission protocols [Bourhis *et al*., 2017a; Pezoa *et al*., 2016; Bourhis *et al*., 2017b]. A JSON is an unordered set of *key-value* pairs [Bourhis *et al*., 2017b]. Keys are strings; the values are weakly typed and may be primitive or complex [Baazizi *et al*., 2019]. Figure 1 shows a JSON example.

A primitive JSON type is a Boolean ($\mathbb{B}$), a number ($\mathbb{R}$), a string ($\mathbb{S}$), or a *NULL* value [Bourhis *et al*., 2017a]. In Figure 1, the key *type* (line 3) has an $\mathbb{S}$ (string) value (*'cinematography'*), another example, is the key *year*, which holds a $\mathbb{R}$ value.

A JSON value can also be from a complex type. A complex value type is either an object or an array. A JSON array $\mathcal{A} = [\tau_0, \tau_1, \ldots, \tau_N]$ is a sequence of N JSON values (primitive or complex) [Spoth *et al*., 2021]. In Figure 1, the key

```
1  {
2    "media": [{
3        "_id": 101,
4        "type": "cinematography",
5        "movie": {
6          "title": "Harry Potter and the Goblet of Fire",
7          "director": "Mike Newell",
8          "year": 2005,
9          "premiere_date": "11/25/2005",
10         "duration": "2:37:00",
11         "price": "15,00",
12         "evaluation": [{"stars":5 }],
13         "genres": ["fantasy","adventure"]}},
14     {
15       "_id": 102,
16       "type": "text",
17       "book": {
18         "title": "Harry Potter and the Goblet of Fire",
19         "author": "J.K. Rowling",
20         "year": 2000,
21         "premiere_date": "06/08/2000",
22         "pages": 480,
23         "price": "25,00",
24         "genres": ["fantasy","adventure" ]
25                 }
26     }],
27   "characters": {
28     "Harry Potter": 14,
29     "Hermione Granger": 14,
30     "Ronald Weasley": 14,
31     "Sirius Black": 46,
32     "Albus Percival Wulfric Brian Dumbledore": 113
33   } }
```

Figure 1. An extract of JSON collection: running example

media (line 2) is an array of complex elements and *genres* (lines 13 and 24) is an array of primitive elements (*i.e.*, $\mathbb{R}$ and $\mathbb{S}$).

In schema extraction, arrays are extracted as data collections [Spoth *et al*., 2021] since they are a sequence of values. However, in some cases, an array can represent an encoded tuple. In the example, the key *premiere_date* has three $\mathbb{R}$ values in each occurrence representing a date (month, day, and year) encoded in an array. A naive schema extraction approach will define the field as an ordinary array, losing information about its structure [Spoth *et al*., 2021].

The last complex type of JSON value is an object. A JSON object $\mathcal{O} = \{k_1 : \tau_1, \ldots, k_N : \tau_N\}$ is a set of keys $k_1 ... k_N$ mapped to values $\tau_1, \ldots, \tau_N$ of JSON types (primitive or complex) [Spoth *et al*., 2021]. The key *movie* in Figure 1 (lines 4 to 12) is an example of a complex JSON object. A JSON object is commonly used to encode a tuple since it has a tuple-like structure. For example, the key *movie* represents a movie object (or tuple) with attributes *title*, *year*, *director*, and each object of type *movie* tends to have a very similar structure. However, the key *characters* diverges from this traditional tuple-like structure and encodes a data collection with some characters of the 'Harry Potter' universe; if we consider another movie, the list of characters tends to be very different. Considering these two cases, on the one hand, we have a tuple that has a very predictable structure (with a few optional fields), and on the other, we have a more flexible structure that stores a metadata collection.

A JSON Schema is a set of rules that define the schema of a JSON [Pezoa *et al*., 2016]. Hence, a JSON is a set of unordered keys organized hierarchically. The schema defines the keys of a JSON and the kinds (types) of the values from

each key [Spoth *et al*., 2021]. The schema generally allows the user to define whether an attribute is optional or mandatory. Also, the schema allows the definition of occurrences of enumerations and tagged unions.

We consider an enumeration of a field with multiple occurrences and a low variability of values. For example, in Figure 1, the key *type* (lines 3 and 14) stores the *media* type, assuming just two possible values: 'cinematography' or 'text'. Any other value cannot be accepted for the field *type*.

A tagged union is a particular type of enumeration that allows conditional occurrences of one or more fields based on the value of a previous key/element [Spoth *et al*., 2021]. For example, in Figure 1, the key *type* is followed by either the key *movie* (line 4) or *book* (line 15), depending on whether the *type* is 'cinematography' or 'text'.

Our approach intends to consider all the facets presented here to discover a schema from a JSON collection, as shown in following sections.

3 Related Work

In this section we present some works related to JSON Schema extraction, where each of them deal with the problem using their own approach. By the end of this section, we list their results in comparison to what is proposed in this paper.

On the works of Frozza et al. (2018) and Baazizi et al. (2019) both emphasize JSON Schema extraction. The first consists of obtaining each key type, followed by removing the duplicated ones after sorting them, and finally creating a tree-based data structure called *Raw Schema Unified Structure* (RSUS), which is manipulated by *Model Driven Engineering* (MDE), enabling the JSON Schema development. On the other hand, the second focus on large datasets using the MapReduce framework, inferring types by using the Map operation as a first phase, in the reduce phase, equal types are merged to become one. On this work, two approaches were proposed: the first refers to the similarity between key types (*kind equivalence*), while the second restricts merging only objects with the duplicate nested keys (*label equivalence*).

Abdelhedi et al. (2021) proposed the *ToNoSQLSchema* tool, which uses *Model Driven Architecture* (MDA) to generate a NoSQL Schema from documents, instead of extracting the JSON Schema. The authors propose six transformation rules: the first creates a collection (*DB_Schema*) from a NOSQL database; the second groups each input collection into a *CollectionSchema*; the third infers atomic types, replacing values with types; The fourth traverses complex structures, applying the third rule when atomic keys are found; and the last two rules are used to create the structure for mono and multivalued keys.

Namba (2021) applies machine learning to enhance the JSON Schema extraction by distinguishing keys that represent data. The work proposes six attributes: (*i*) the Intrinsic Characteristics Domain, (*ii*) the Central Tendency Domain, (*iii*) the Statistical Dispersion Domain, (*iv*) the Distribution Shape Domain, (*v*) the Semantic and Contextual Similarity Domain, and (*vi*) the Structural Similarity Domain. Those features are used to build a labeled dataset to infer whether or not a pair (key, value) represents metadata or data.

Table 1. Table comparing the information inferred from each related work.

Reference	BT	TU	Meta	Col	T	E
Frozza et al. (2018)	Y	N	N	N	N	N
Baazizi et al. (2019)	Y	N	N	N	N	N
Abdelhedi et al. (2021)	Y	N	N	N	N	N
Namba (2021)	Y	N	Y	Y	Y	N
Klessinger et al. (2022)	Y	Y	N	N	N	N
Spoth et al. (2021)	Y	N	Y	Y	Y	N
Yun et al. (2024)	Y	N	Y	Y	Y	N
JFUSE	Y	Y	Y	Y	Y	Y

Klessinger et al. (2022) aim to discover tagged unions by detecting dependencies between a value and specific structures. They generate a tree from the values in a JSON collection so that a *relational encoding* can be created and dependencies between keys can be found.

Spoth et al. (2021) developed *Jxplain*, which uses heuristics to reduce schema ambiguities. They mention that most tools do not consider objects can appear with the structure of a collection, and arrays can have the structure of a tuple. For that, they calculate Key-Space and Type Entropy. The first considers that keys tend to vary more on collections, whether types have the opposite behavior. They also identify Multi-Entity Collections using a bi-clustering technique.

The ReCG [Yun *et al*., 2024] follows a different approach. The proposed algorithm transforms all JSON documents in a forest. Based on a botton-up approach, the trees are clustered based on the structures of their children. For clustering, the MDL (Minimum Description Lenght) principle is applied to create a general schema that is capable of accepting all JSON documents and being minimal in length. All generated subschemas are merged using the MDL principle and the viability of executing the merge. The experimental results show that the approach is not prohibitive, although when the forest has many trees, the performance of the system is reduced.

To compare the related work with our proposal, we show Table 1, summarizing the features listed on each of the previously presented paper. The columns *BT*, *TU*, *Meta*, *Col*, *T*, and *E*, stand for Basic Types (e.g., atomic, objects, and arrays), Tagged Unions, Metadata, Collections, Tuples, and Enumeration.

Note that all approaches, as expected, extract basic types (*i.e.*, primitive and complex). The work of Frozza et al. (2018), Baazizi et al. (2019) and Abdelhedi et al. (2021) focus only on the extraction of this kind of type. Namba and Mior (2021), Spoth et al. (2021), and Yun et al. (2024) propose very complete tools, however they lack discovering tagged unions and enumerations. Although Klessinger et al. (2022) include tagged union in the target schema, other types are not considered. JFUSE, on the other hand, can discover the main JSON collections facets.

4 JSON-Extract

This section describes JFUSE, our approach to discovering schema in JSON collections. Firstly, we show how to represent a schema collection as a data structure; we choose a graph structure representation.

4.1 Graph Representation

JSON collections may be easily viewed as a graph, where fields are vertices and sub-schema associated with a field are connected to the parent by edges. Furthermore, graphs allow a straightforward and fast way of traversing between parents, siblings, and children, which is highly valuable when building the proposed schema.

The following definitions formalize how a JSON collection is loaded to the main memory.

Definition 1 *JSON Graph*. A JSON graph is a directed graph built from a JSON collection defined by the tuple $G = (V, E)$. ◇

Definition 2 *Vertex*. A vertex $\nu \in V$ is a tuple ν=< $l, \mathcal{T}, c, isKey, isEnum, isTU, \Lambda$> where l is the vertex's label and represents a field name, $\mathcal{T}$ is a tuple <t_1:occ_1, ..., t_n:occ_n> (possibly unitary) found in l instances (where t_i:occ_i is a key:value element in which t_i represents a type and occ_i the number of occurrences of t_i in l), c is the number of occurrences of l in the collection, $isKey$ is set to True if l is a key, $isEnum$ indicates if l contents is an enumeration, $isTU$ states if l defines a tagged union, and Λ stores a set (possibly empty) of possible values for l in V. ◇

The following example illustrates how Definition 2 is applied to build the vertices of our proposed graph.

Example 1 *Given the JSON collection from Figure 1, the following vertices belong to the graph built from the collection:*

- ν_1=<*media,*<*arr:1*>*,1, False, False, False, NULL*>
- ν_2=<*_id,*<*str:2*>*,2, True, False, False, NULL*>
- ν_3=<*type,*<*str:2*>*,2, False, True, True, {cinematography, text}*>
- ν_4=<*movie,*<*obj:1*>*,1, False, False, False, NULL*>
- ν_5=<*title,*<*str:2*>*,2, False, False, False, NULL*>
- ν_6=<*director,*<*str:1*>*,1, False, False, False, NULL*>
- ν_7=<*year,*<*num:2*>*,2, False, False, False, NULL*>
- ν_8=<*genres,*<*arr:2*>*,2, False, True, False, {fantasy, adventure}*>

Note that the field *type*, for example, appears twice in the collection, and in both cases, it is a string. The same goes for *title* and *year*.

An essential concept regarding databases is key. A key is a set of attributes (possibly unary) uniquely identifying a record. In a JSON document, a field (or a set of fields) may represent a key. For example, $\_id$ (ν_2) is a key on our running example. We consider a field as a key if and only if its type is string or numeric and the uniqueness respects a threshold (Thr_k). Thr_k is used to handle potential typos. For instance, when exporting data in JSON format, an application may duplicate one or more records. Users can adjust Thr_k to manage the duplicates if they know this possibility. If Thr_k is set to 1, the field must contain unique values.

In the following, we define how a field becomes an enumeration and, if so, a tagged union. We use three thresholds to help discovering enumeration and tagged unions: (*i*) thr_Λ to identify whether or not the content of a field may be an enumeration, (*ii*) Thr_t to indicate if the field content is dominated by a given type, and (*iii*) Thr_{str} to check the length of the string in the content of a given field.

Definition 3 *Enumeration*. A field from a JSON collection is set as an enumeration (i.e., *isEnum* is true) if and only if $\nu \in V$ is associated with a set of values Λ such that: (i) $\mid \Lambda \mid \leq thr_\Lambda$ and (ii) let t' in $\mathcal{T}$ be a tuple with a key:value t_i:occ_i:

- $\frac{t'.occ_i}{\sum_{k=1}^{|\mathcal{T}|} t_k.occ_k} \geq Thr_t$; and
- If t_i is a string type , let λ' be the value with the maximum length in Λ, $\lambda' \leq Thr_{str}$

◇

The intuition behind Definition 3 is as follows: (i) the number of unique values of a given field cannot be greater than a threshold (Thr_Λ), (ii) the unique values must have a dominant type ($Thr_t \geq n\%$ where n is the value), (iii) if the predominant type is a string, the length of the larger value cannot be greater than a threshold (Thr_{str}), since string enumerate values tend to be small, and (iv) float values tend not to compose enumeration values. For example, given the set of *genres* values equals to Λ={*fantasy, adventure*}, Thr_Λ equals to 10, Thr_t equals to 0.8, and Thr_{str} equals to 20, *genres* is considered an enumeration.

Definition 4 *Tagged Union*. A field from a JSON collection is set as a tagged union (i.e., isTU is true) if and only if $\nu \in V$ is an enumeration and its siblings respect the following: (i) let λ_1 be a distinct value of vertex ν, so $\nu.\lambda_1 \rightarrow v'$ are ensured in G. ◇

The relationship $\alpha \rightarrow \beta$ (*i.e*, a functional dependency) from Definition 4 means that α determines the value of β. In our approach, we borrow the functional dependency definition from the database theory to state that, given two vertices $\nu, \nu' \in V$ and $\lambda_1 \in \Lambda$ in ν, $\nu.\lambda_1$ determines a sub-schema ν'.

The following example shows the use of enumeration and tagged union.

Example 2 *Given the JSON collection from Figure 1, the field* type *is an enumeration since it comprises two distinct values:* cinematography *and* text*. And, it is a tagged union because when its value is cinematography, its right sibling is the field* movie*; otherwise, it is the field* book*. The relationships are* type.cinematography→ movie *and* type.text→ book*. On the other hand, the field* genres *is also an enumeration; however, it is not a tagged union.*

Definition 5 *Edge*. An edge $\varepsilon \in E$ is a tuple ε=<(ν_s, ν_t), $rs, c_\varepsilon, lv_\varepsilon$> where ($i$) ($\nu_s, \nu_t \in V$) is a pair representing the source and target of the edge, respectively, (ii) *rs* is the relationship between ν_s and ν_t and can it assume p or s indicating that ν_s is parent or sibling of ν_t, respectively, (iii) c_ε stores the number of occurrences of *rs* between ν_s and ν_t, and (iv) lv_ε is a list (possibly empty) that stores the values appearing when ν_t is a sibling of ν_s. ◇

The components of a tuple in an edge ε are used as follows: (i) *rs* is employed in two flavors: first, to identify the subschema of a field when the relationship is p or to determine if ν_s is a tagged union candidate, (ii) c_ε controls whether or not a relationship rs is mandatory ($\frac{c_\varepsilon}{c \ in \ \nu_s} > Thr_m$), *i.e.*, the ratio of the number of occurrences of ν_s and the number of occurrences of the relationship with ν_t is greater than a threshold, and (iii) lv_ε is used to build a tagged union type.

Example 3 *Still using the JSON collection from Figure 1, the following edges belong to the built graph:*

- ε_1=<*(book, genres),p,2,NULL*>
- ε_2=<*(movie, director) ,p,1,NULL*>
- ε_3=<*(book, author),p,1,NULL*>
- ε_4=<*(type, book),s,1,('text')*>
- ε_5=<*(type, movie),s,1,('cinematography')*>

The above definitions state how we build a data structure to represent a JSON collection and use it to extract enumerations and tagged unions. Note that we need to set some thresholds (*i.e.*, Thr_k, Thr_Λ, Thr_t, Thr_{str}, and Thr_m) to allow our approach to work correctly. We run some experiments to identify the best values for the thresholds. In Section 5, we present the values used in the main experiments.

Finally, Figure 2 shows a graph representation from JSON collection in Figure 1. We use ellipses to represent all vertices, except for tagged unions represented by diamonds (field *type*), enumeration by houses (field *genres*), metadata as data by rectangles (vertices *string* and *numeric*), and vertices affected by tagged unions are reached by dotted edges (fields *movie* and *book*). Note that, for clarity, not all sibling edges appear in the graph, and we do not show the content of the vertices and edges.

4.2 Tuples, Collections, and Metadata

Section 2 shows that it is common sense that array types are composed of collections, and object types are composed of tuples. However, some JSON documents do not follow that. If the content of a given array $\mathcal{A}$ is very similar. The similarity is calculated based on the content type and a threshold (see Definition 6). $\mathcal{A}$'s content can be seen as a tuple. The same reasoning can be applied to objects: if the content is dissimilar, it represents a collection of other objects. Besides, a subschema may represent data instead of metadata. For example, the content of field *characters* is not a list of *field:value*; it is a list of names with ages, and, in this case, they are not metadata. We cannot extract a rule like *characters*: {"harry potter": integer "hermione granger": integer} because the characters' names and ages represent data. In the following, we formalize some definitions to identify when content is a tuple or a collection. When it is a tuple, the content may be considered data.

Definition 6 *Array as Tuple. Given an array content $\mathcal{C}$, $\mathcal{C}$ is seen as a tuple if and only if $\mathcal{C}$ is composed of Thr_{arr} of same elements.* ◇

Note that, from the definition, when the content of an array is very similar, we can consider it as a tuple. A threshold is used to take into account noise in the content.

Definition 7 *Object as Collection. Given an object content $\mathcal{C}$, $\mathcal{C}$ is seen as a collection if and only if $\mathcal{C}$ contains Thr_{obj} of dissimilar elements.* ◇

Note that from the definition, when the content of an object contains some dissimilarity, we can consider it as a collection. The threshold Thr_{obj} is used to take into account noise in the content.

Finally, we define the problem of data being represented as metadata. This problem was raised in [Namba, 2021; Spoth *et al*., 2021], and the goal is to find when a JSON sub-schema represents data instead of metadata.

Definition 8 *Metadata as data. Given a content $\mathcal{C}$ of an object seen as a collection, if $\mathcal{C}$ is composed only of <key>:<value> and more than Thr_{dt}% are optional, $\mathcal{C}$ represents data instead of metadata in the collection.* ◇

Note that, from Definition 8, the threshold Thr_{dt}% plays an essential role in considering a pair <*key*>:<*value*> as data instead of metadata. For example, if a label *L* occurs 50 times, its child *Lc* occurs 48 times, and Thr_{dt}% is set to 95, *Lc* would be mandatory.

The content of the field *characters* (Line 23 in Figure 1) is a case of metadata as data: the characters name and age are data. Representing them as metadata, we should use another type of representation, for example, <name>:$\mathbb{R}$. Instead, the model is $\mathbb{S}$:$\mathbb{R}$. The optionality of the content leads our approach to consider it as data or metadata.

4.3 JSON Metamodel

We propose a metamodel to represent a conceptual schema for JSON collections. Our metamodel is expressed using BNF-like metasyntax (Backus-Naur form). BNF is a formal way to describe a language and, in our approach, a JSON schema. It consists of a set of terminal and non-terminalIt consists of a set of terminal and non-terminal symbols. The symbols derive a language using production rules in the form *left-hand-side::=right-hand-side*, where LHS (Left-Hand-Side) is a non-terminal symbol, and RHS (Right-Hand-Side) is a sequence of symbols (terminals or non-terminals). The meaning of a production rule is the LHS (a non-terminal symbol) may be replaced by the expression represented by RHS.

Accordingly, our meta JSON schema language is defined as follows:

⟨*atm-type*⟩ ::= $\mathbb{S}$ | $\mathbb{R}$ | $\mathbb{B}$ | $\mathbb{D}$ | $\mathbb{T}$ | $\mathbb{TS}$ | *null*

⟨*field-name*⟩ ::= $(\mathbb{S})^+$

⟨*opt-field*⟩ ::= ⟨*field-name*⟩?

⟨*atm-field*⟩ ::= ⟨*field-name*⟩ ':' ⟨*at-type*⟩

⟨*arr-type*⟩ ::= '[' ⟨*arr-value*⟩, . . . , ⟨*arr-value*⟩ ']'

⟨*arr-value*⟩ ::= (⟨*atm-type*⟩ | ⟨*arr-type*⟩ | ⟨*obj-type*⟩)$^+$

⟨*array*⟩ ::= ⟨*field-name*⟩ ':' ⟨*arr-type*⟩

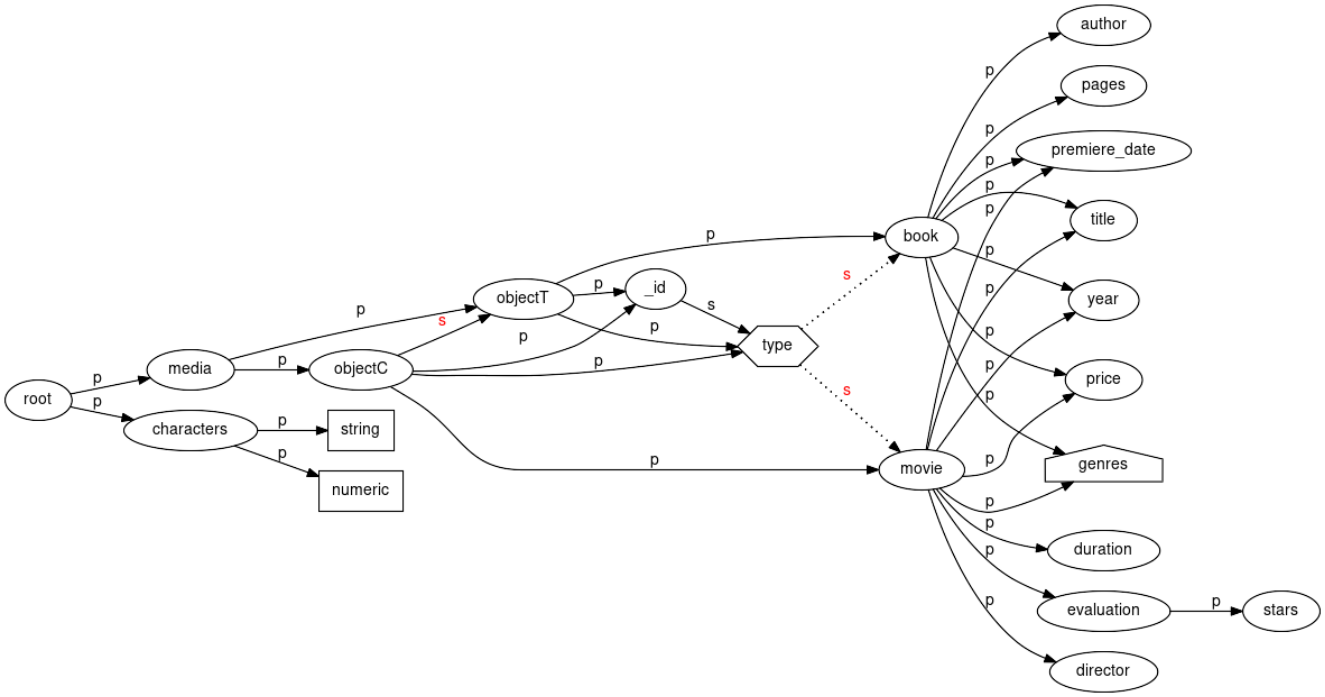

Figure 2. Graph representing the collection in Figure 1.

⟨*obj-type*⟩ ::= '{' (⟨*atm-field*⟩ | ⟨*array*⟩ | ⟨*object*⟩)$^+$ '}'

⟨*object*⟩ ::= ⟨*field-name*⟩ ':' ⟨*obj-type*⟩

where (i) <*atm-type*> defines the atomic types $\mathbb{S}$, $\mathbb{R}$, $\mathbb{B}$, $\mathbb{D}$, $\mathbb{T}$, $\mathbb{TS}$, and *null* represent a string, numeric, boolean, date, time, timestamp, and *null* value, respectively and (ii) <*field-name*> represents a valid field name in JSON collections. The other constructors follow the same reasoning. Regarding arrays, the array type (<*arr-type*>) does not control the minimum or maximum elements allowed in a given array

Finally, enumeration and tagged union production rules can be defined as follows:

⟨*enum*⟩ ::= ⟨*field-name*⟩ ':[' ⟨*atm-type*⟩, ..., ⟨*atm-type*⟩ ']'

⟨*tagged-union*⟩ ::= 'IF' ⟨*enum-cond*⟩ 'THEN' (⟨*atm-field*⟩ | ⟨*array*⟩ | ⟨*object*⟩)

⟨*enum-cond*⟩ ::= ⟨*field-name*⟩ '.' ⟨*atm-type*⟩

Moreover, JFUSE extract two additional facets:

- When an atomic type represents a key, its type is suffixed with k. For instance, `email: S k` indicates that the field `email` represents a string key in the JSON document. This suffix is a clear indicator of the field's key status. In addition to the threshold for dealing with typos in the values of candidate keys (Thr_k), we introduce another threshold to prevent certain string fields from being considered as keys: the string length (Thr_{len}). This feature helps JFUSE to disregard some fields as key candidates, such as *address* and *comments*. However, it is essential to note that Thr_{len} does not apply to *email*, *OID*, and hexadecimal values.
- Numeric fields and their minimum and maximum values can be extracted. In this case, we decorate the RHS of the production rule with `(min=vmin max=vmax)`, where `vmin` and `vmax` correspond to the minimum and maximum values allowed for the field, respectively. For example, in our metamodel, a field age can be represented as `age:R(min=1 max=100)`.

In both above cases, the user can select, using parameters, whether or not those features are extracted. This level of customization puts the user in control of the extraction process, ensuring that it aligns with their specific requirements.

The following example shows an instance of our metamodel representing the JSON collection from our running example (Figure 1).

Example 4

root ::= *{**media:** arr_m **characters:** str_ch}*
arr_m ::= [*obj_a*]
obj_a ::= ***_id:**$\mathbb{R}k$ IF **type.cinematography** THEN obj_c*
| *IF **type.text** THEN obj_t*
obj_c ::= ***movie:** obj_m*
obj_t ::= ***book:** obj_b*
obj_m ::= ***title:**$\mathbb{S}$ **director:**$\mathbb{S}$ **year:**$\mathbb{R}$ **premiere_date:**$\mathbb{D}$*
***duration:**$\mathbb{R}$ **price:** $\mathbb{N}$ **genres**:arr_g*
obj_b ::= ***title:**$\mathbb{S}$ **author:**$\mathbb{S}$ **year:**$\mathbb{R}$ **premiere_date:**$\mathbb{D}$*
***pages:**$\mathbb{R}$ **price:**$\mathbb{R}$ **evaluation**: obj_s*
***genres**: arr_g*
obj_s ::= ***stars**: $\mathbb{R}$*
arr_g ::= [***fantasy** , **adventure***]
str_ch ::= *{($\mathbb{S}$: $\mathbb{R}$)$^+$}*

Note that *obj_a* represents a tagged union, *arr_g* represents an enumeration, *str_ch* is modeled as pairs *string:string* because the field *characters* is considered a collection and not an object, and *_id* models numeric key.

5 Results and Discussion

To demonstrate the quality of JFUSE, we used both real and synthetic JSON collections in our experiments. First, we specified the environment in which the experiments took place, and then, we presented both real and synthetic experiment results.

Based on the definitions mentioned in Section 4, our approach was developed using the C++ programming language. It can be found at *https://github.com/NathyBanhara/JFUSE*. The experiments were performed on a server machine with four Intel(R) Xeon(R) CPU E7- 4850 (2.00GHz) and 128 GB RAM, running a Linux 4.15.0-50 kernel (Ubuntu 18.04.2 LTS distribution). We ran an empirical experiment on five datasets (four real and one synthetic) and studied some JSON collections to find suitable values for all the thresholds, which are: $Thr_k = 1$, $Thr_{len} = 16$, $Thr_m \geq 0.9$, $Thr_t \geq 0.5$, $Thr_{str} \leq 20$, $Thr_{arr} \geq 0.9$, $Thr_{obj} \leq 0.1$, and $Thr_{dt} \geq 0.7$.

5.1 Real Data Experiments

Four JSON document collections were used to test our tool with real data. The first one was taken from [Spoth *et al.*, 2021]. The study case refers to pharmaceutical data (*PHC*), which allows testing on scenarios such as objects as collections and enumeration detection, besides basic types. The second one regards Russia's 2018 election tweets user activity (TWC). Obtained from Kaggle[1], the dataset contains tweet records, which is interesting due to its many optional fields. The last two were taken from MongoDB Sample Dataset[2], and the sample data was taken from *AirBnB* and *Suppliers* (a mock office supply company). We ran the experiments five times to ensure that there would be no discrepancy in the execution time, and it was verified since the standard deviation was less than 1%. Table 2 summarizes the experiments' results. Columns *Time WK* and *Time WoK* show the average running time considering keys and min/max extraction and without them, respectively.

Table 2 shows that the pharmaceutical experiments, with a size of 165Mb and 7,226,980 keys, had an average performance time of 1m59.947s. The TWC schema, a much more extensive collection with 11,5GB and 420,022,871 keys, was generated after about 110m46.484s. The *AirBnB* and *Suppliers* collections follow the same execution time pattern regarding their sizes (**Size**) and number of fields (**Fields**).

Table 2. Table showing the results obtained from experiments with real data.

Col.	Size	Fields	Time WK	Time WoK	SKeys
Supliers	4.64MB	244,628	5.35s	4.02s	21
Airbnb	97,3MB	1,850,383	44.72s	30.42s	92
PHC	165MB	7,226,980	2m32.40s	1m59.947s	11
TWC	11,5GB	420,022,871	151m46.20s	110m46.484s	210

We have a wide range of collections sizes, from a hundred thousand to a hundred million. TWC is 8.7 times bigger than PHC, having 58 times more fields than PHC. However, concerning the execution time, TWC was 69 times slower than PHC. The same is true for *AirBnB* and *Suppliers*, but the ratio between the size and execution time is linear (7.5 times in size and running time). Note that when we choose the extract keys and min/max values, the execution time increases around 40%.

We believe that the number of keys impacts the computational performance more than the size of collections, and because of that, our approach scales well when facing huge collections. Moreover, the column *SKeys* shows the number of keys in the resulting schema: *Suppliers* comprises 21 keys, *AirBnB* 92 keys, PHC 11 keys, and TWC 210 keys. It shows that the built schemas are concise.

Figure 3 shows the schema extracted from PHC (based on the meta-model described in Section 4). As can be seen, the `cms_prescription_counts.object` represents an object as collection, and the metadata inside it is defined as data, which means the optional fields in the `cms_prescription_counts.object` are greater than Thr_{dt}. On the other hand, `region`, `gender`, `years_practicing`, and `settlement_type` are all enumerations. Note that some numeric fields (i.e., `brand_name_rx_count` and `generic_rx_count`) are decorated with their minimum and maximum because we ran this experiment to extract such values.

5.2 Synthetic Experiments

We built four new synthetic collections from the Figure 1 template to produce collections containing every type that JFUSE intends to discover (*i.e.*, atomic types, keys, min/max values, tagged unions, metadata, objects as collections, arrays as tuples, and enumeration). We ran the experiments five times for the datasets, and we reported the execution time average and the standard deviation. All experiments consider the keys and min/max values to be extracted. Table 3 shows some statistics from this experiment.

Note that the execution times follow the number of fields in the collections. For example, the third collection contains 2,800,207 fields (Column **Fields**), and it took 145.06s to extract the schema; the fourth collection, on the other hand, is around 10 times greater than the second one and took around 10 times longer. Looking at the standard deviation (**Std (s)**), we see that all five executions had similar times since the variation is around 2%. Regarding the size of the schemas, our approach is stable, especially when collections follow a pattern (as our synthetic collection does); see column **SK** in Table 3.

Table 3. Results obtained from experiments with synthetic data.

Objects	Sz (Mb)	Fields	Avg(s)	Std (s)	SK
10,000	12.07	279,891	12.74	0.89	15
50,000	60.4	1,400,349	70.83	1.41	15
100,000	121.22	2,800,207	145.06	2.61	15
500,000	603,918	14,000,402	724.53	12.32	15

5.3 Discussion

We manually compared the extracted schemas to samples of the input collections, and we confirmed that JFUSE could

[1] www.kaggle.com/datasets/borisch/russian-election-2018-twitter
[2] github.com/neelabalan/mongodb-sample-dataset

```
root::= npi: 𝕊
        provider_variables: provider_variables.object
        cms_prescription_counts: cms_prescription_counts.object
provider_variables.object::= brand_name_rx_count: R(min=0; max=33601)
                             region: [Northeast, West, South, Midwest]
                             gender: [F, M]
                             years_practicing: [3, 6, 7, 4, 2, 1, 5, 8]
                             specialty: 𝕊
                             generic_rx_count: R(min=0; max=111186)
                             settlement_type: [urban, non-urban]
cms_prescription_counts.object::= (𝕊:ℝ)+
```

Figure 3. Schema extracted from PHC JSON collection.

extract all the facets it intended to do: keys, min/max values, enumeration, tagged union, metadata as data, collections, and tuples (see Section 4). Moreover, the resulting schemas are concise regarding the size of the input collections, and the execution time is satisfactory. We use a synthetic collection to provide a proof of concept for our definitions stated in Section 4. The experiments also showed that our approach is scalable and the extracted schema is stable (Column **SK** from Table 3). Finally, our metamodel can be used as a source to build any JSON schema-language-like.

6 Conclusion

We introduced a novel approach to extracting schema from JSON collections. The key distinguishing features of our tool are:

- Our tool has the unique capability to discover tagged unions, a feature that is not straightforward to extract. This is particularly valuable as a conditional value of an object's property (the tag) may imply subschemas for sibling properties.
- Field values may be considered enumeration based on a threshold. This allows the tool to handle variations in data representation, enhancing the accuracy of schema extraction.
- Fields that are keys for the collection and fields that are constrained by minimum and maximum values can also be extracted.
- JFUSE can also distinguish between tuples and collections, thereby accurately identifying the content of arrays and objects. This capability significantly improves the reliability of the schema extraction process.
- We propose a metamodel that can be transformed into any schema language.
- It captures data encoded as metadata, *i.e.*, although a field is encoded as an object, it may represent collections where each element maps keys to values. For example, the field `characters` from Figure 1 is encoded as *character name* and their *age*.

Our experiments demonstrated that our approach successfully extracted all the JSON schema facets we proposed. The execution time was satisfactory, and the extracted schema was concise and accurate.

Although we believe that JFUSE is fully functional, some adjustments could be proposed: (*i*) fields with the same name but in different contexts of type enumeration could be handled separately, (*ii*) check possible cycles in the graph (*e.g.*, caused by tagged unions or two fields having the same name with parent-child relationship), (*iii*) testing the scalability and schema stability in real collections by growing it exponentially, (*iv*) conducting some experiments in real collections valid regarding a schema to compare the extracted schema against a real one, and (*v*) automate the determination of threshold values, allowing the tool to identify the optimal values for a given collection.

Funding

Natália Banhara was partially funded by Universidade Federal da Fronteria Sul under process number PES-2021-0458.

Authors' Contributions

DD, GS, and SF contributed to the conception of this study. NB and GS performed the experiments. All authors contribute equally to write this manuscript and also read and approved the final version.

Competing interests

The authors declare that they do not have competing interests.

Availability of data and materials

The datasets generated and/or analysed during the current study are available in `github.com/NathyBanhara/JFUSE`

References

Abdelhedi, F., Brahim, A. A., Rajhi, H., Ferhat, R. T., and Zurfluh, G. (2021). Automatic extraction of a document-oriented nosql schema. In *ICEIS (1)*, pages 192–199. DOI: dx.doi.org/10.5220/0010433501920199.

Baazizi, M.-A., Colazzo, D., Ghelli, G., and Sartiani, C. (2019). Parametric schema inference for massive JSON datasets. *The VLDB Journal*, 28:497–521. DOI: doi.org/10.1007/s00778-018-0532-7.

Banhara, N., Schreiner, G. A., da Silva Feitosa, S., and Duarte, D. (2024). Enumeration, tagged unions, tuples, and collections: A novel approach to extracting json schema. In *Simpósio Brasileiro de*

Banco de Dados (SBBD), pages 234–246. SBC. DOI: doi.org/10.5753/sbbd.2024.240239.

Bell, G., Hey, T., and Szalay, A. (2009). Beyond the data deluge. *Science*, 323(5919):1297–1298. DOI: doi.org/10.1126/science.1170411.

Bourhis, P., Reutter, J. L., Suárez, F., and Vrgoč, D. (2017a). JSON: data model, query languages and schema specification. In *Proceedings of the 36th ACM SIGMOD-SIGACT*. DOI: doi.org/10.1145/3034786.3056120.

Bourhis, P., Reutter, J. L., Suárez, F., and Vrgoč, D. (2017b). Json: data model, query languages and schema specification. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on principles of database systems*, pages 123–135. DOI: doi.org/10.1145/3034786.3056120.

Cánovas Izquierdo, J. L. and Cabot, J. (2013). Discovering implicit schemas in json data. In *Web Engineering: 13th International Conference, ICWE 2013, Aalborg, Denmark, July 8-12, 2013. Proceedings 13*, pages 68–83. Springer. DOI: doi.org/10.1007/978-3-642-39200-9_8.

Frozza, A. A., dos Santos Mello, R., and da Costa, F. d. S. (2018). An approach for schema extraction of JSON and extended JSON document collections. In *IRI*. IEEE. DOI: doi.org/10.1109/IRI.2018.00060.

Kellou-Menouer, K., Kardoulakis, N., Troullinou, G., Kedad, Z., Plexousakis, D., and Kondylakis, H. (2022). A survey on semantic schema discovery. *The VLDB Journal*, 31(4):675–710. DOI: doi.org/10.1007/s00778-021-00717-x.

Klessinger, S., Klettke, M., Störl, U., and Scherzinger, S. (2023). Extracting JSON schemas with tagged unions. *arXiv preprint arXiv:2306.07085*. DOI: doi.org/10.48550/arXiv.2306.07085.

Namba, J. (2021). Enhancing JSON schema discovery by uncovering hidden data. In *VLDB 2021 PhD Workshop*. DOI: urn:nbn:de:0074-2971-6.

Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., and Vrgoč, D. (2016). Foundations of json schema. In *International World Wide Web Conferences*, WWW '16. DOI: 10.1145/2872427.2883029.

Spoth, W., Kennedy, O., Lu, Y., Hammerschmidt, B., and Liu, Z. H. (2021). Reducing ambiguity in JSON schema discovery. In *Proceedings of the 2021 SIGMOD*. DOI: doi.org/10.1145/3448016.3452801.

Yun, J., Tak, B., and Han, W.-S. (2024). Recg: Bottom-up json schema discovery using a repetitive cluster-and-generalize framework. *Proc. VLDB Endow.*, 17(11):3538–3550. DOI: 10.14778/3681954.3682019.