Demonstration Paper

Tagger: A Tool for the Discovery of Tagged Unions in JSON Schema Extraction

Stefan Klessinger
stefan.klessinger@uni-passau.de
University of Passau
Passau, Germany

Michael Fruth
michael.fruth@uni-passau.de
University of Passau
Passau, Germany

Valentin Gittinger
gittinge@fim.uni-passau.de
University of Passau
Passau, Germany

Meike Klettke
meike.klettke@ur.de
University of Regensburg
Regensburg, Germany

Uta Störl
uta.stoerl@fernuni-hagen.de
University of Hagen
Hagen, Germany

Stefanie Scherzinger
stefanie.scherzinger@uni-passau.de
University of Passau
Passau, Germany

ABSTRACT

This tool demo features an original approach to model inference or schema extraction from collections of JSON documents: We automatically detect tagged unions, an established design pattern in hand-crafted schemas for conditionally declaring subtypes. Our "Tagger" approach is based on the discovery of conditional functional dependencies in a relational encoding of JSON objects. We have integrated our prototype implementation in an open source tool for managing data models in schema-flexible NoSQL data stores. Demo participants can interactively apply different schema extraction algorithms to real-world inputs, and compare the extracted schemas with those produced by "Tagger".

1 INTRODUCTION

JSON is a highly popular format in data exchange, and JSON Schema is the *de facto* standard for declaring schemas for collections of JSON data. Scenarios for employing JSON Schema are manifold [17]. In this tool demo, we target the problem of *model inference*, specifically the extraction of a JSON Schema declaration from a collection of JSON documents, an active research field [3, 10, 22]. We focus on the so far unexplored discovery of *tagged unions*. Tagged unions are also known as *discriminated unions*, *labeled unions*, or *variant types*, and constitute a recommended design pattern in JSON Schema modeling [9]. They have been confirmed to appear in real-world schemas [4].

We next motivate tagged unions by an example. In the GeoJSON data to the left of Figure 1, the array beginning in line 2 holds several objects which are distinguished by the value of property `type`, i.e., `Point` and `LineString`. Point coordinates are encoded as an array of numbers, while line coordinates are encoded as an array of points. In such tagged unions, one property serves as the *tag* (in our example, property `type`), and implies a subschema for its sibling properties (in our example, the `coordinates`). In GeoJSON, the tag `type` distinguishes six different "geometries" [6].

Thus, compared to plain *union types*, tagged unions describe not only which different subschemas a property may have, but also under which conditions certain subschemas apply.

To the right of Figure 1, we show a matching subschema where `if-then-else` operators declare a tagged union. We informally describe the schema semantics: If the object in question has a property labeled `type` with the value `Point`, then the value of property `coordinates` (should the property exist) must be an array of integers. If the property `type` exists and has the value `LineString`, the value of `coordinates` (if present) must be an array of points, and hence, an array of integer arrays.

Approach and contribution. In this tool demo, we target the automated discovery of tagged unions in JSON documents. Our approach relies on the discovery of conditional functional dependencies (CFDs) [5], based on a relational encoding of the JSON objects reachable by the same path. To prevent overfitting, filtering heuristics are applied to the detected CFDs. The recognized CFDs are then translated to JSON Schema.

While our running example is based on GeoJSON, our approach is not restricted to this specific format, and we also work with further datasets in our tool demo.

Own prior work. We have integrated Tagger within Josch, an open source tool that was developed by us in earlier work, and that has been the subject of earlier tool demos [11, 12]. Josch integrates several schema extraction libraries, and allows to configure a sample size. The extracted schemas can be edited and input data can be validated against the schemas. Additionally, Josch integrates different libraries for JSON Schema containment checking, allowing to compare the extracted schemas.

We extended Josch by integrating Tagger, so that users may conveniently compare the schemas extracted by Tagger against other libraries within the same tool.

An article describing the theory behind Tagger in greater detail than possible here, as well as first experiments with real-world data, was presented at the DEco workshop [14].

Artifact availability. The demoed tool is open source:
https://github.com/sdbs-uni-p/tagger-edbt2023

2 TAGGED UNIONS IN JSON SCHEMA

We assume that our readers are familiar with JSON syntax and semantics. The JSON Schema language also uses JSON syntax. We refrain from formally introducing its full semantics, and instead defer to Pezoa et al. [19], and make do with an informal introduction of the operators relevant in our context.

Tagged unions constitute a design pattern used in practice [4, 9]. There are various patterns how to encode tagged unions in JSON Schema. In JSON Schema Draft 7 [24], we may employ `if-then-else` expressions which are of the following form:

`"if": {`S_1`}, "then": {`S_2`}, "else": {`S_3`}`

Here, S_1, S_2, and S_3 are subschemas. If S_1 is satisfied, S_2 must be satisfied as well, otherwise S_3 needs to be satisfied. Naturally, the `else` case may be omitted.

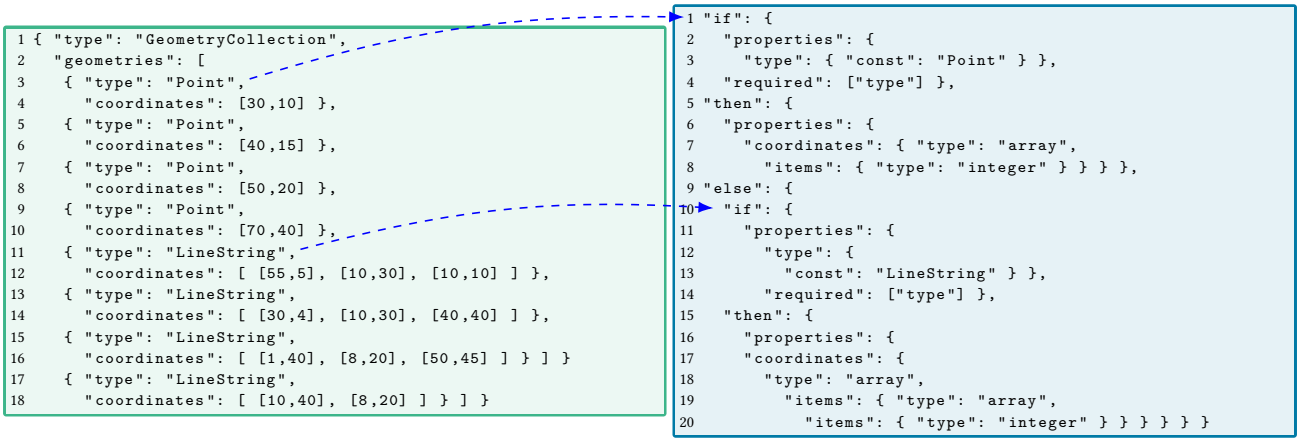

Figure 1: Left: GeoJSON sample. The value of property `type` (here `"Point"` and `"LineString"`) determines the subschema of property `coordinates`. Right: JSON Schema snippet declaring the tagged union. Dashed arrows map from data to schema.

Prior to Draft 7, tagged unions were often described by rewriting the implication $A \Rightarrow B$ to $\neg A \vee B$, using the already existing expressions for disjunction and negation. While this leads to less comprehensive schemas [9], it is actually one of the main use cases for the operator `not` in real-world schemas [4].

Example 2.1. Let us consider our running example. Figure 1 shows an `if-then-else` expression in a JSON Schema declaration for GeoJSON data. In the JSON documents on the left, the value of property `type` determines the subschema of property `coordinates`. Blue arrows point to the corresponding `if-then` expressions in the schema on the right-hand-side.[1]

3 RELATED WORK

Our work on Tagger builds upon existing work on schema extraction and dependency discovery. Accordingly, we review the related work in these areas.

JSON schema extraction. Within the last decade, several approaches for JSON schema extraction have been proposed. However, tagged unions have received very little attention so far: According to recent surveys [8, 23], each comparing the same five approaches [3, 10, 13, 16, 21], by Izquierdo et al. [13], Klettke et al. [15, 16], Sevilla et al. [21], Frozza et al. [10] and Baazizi et al. [3], three of the five analyzed approaches are able to detect *union types*. However, individual inspection reveals that none of them targets *tagged unions*. Notably, Baazizi et al. [3] outline how to extend their schema inference approach to include tagged unions. However, they do not provide an implementation for this particular feature. Moreover, our approach differs conceptually from their work, which relies on typing.

To our knowledge, we present the first implementation of JSON Schema extraction capable of detecting tagged unions.

Based on their approach, Baazizi et al. [2] further propose an interactive tool for schema extraction, allowing the user to tune the precision of the schema to the desired level.

A recent contribution by Spoth et al. [22] focuses on identifying and reducing ambiguities in schema extraction. While addressing common encoding patterns found in real-world JSON data, this approach does not feature the discovery of tagged unions either.

[1] JSON Schema employs a conditional semantics: Adding `required` in the `if` clause causes it to be only true when the properties (here: `type`) are actually present.

To the best of our knowledge, tagged unions were also not addressed by work on XML schema discovery. An overview of the literature on this topic is available in our earlier work [14].

Dependency discovery. Our approach relies on the discovery of conditional functional dependencies in a relational encoding of JSON objects. In the relational model, functional dependencies [1] capture dependencies between attribute values. Conditional functional dependencies [5] are a generalization and apply only to a subset of tuples (commonly identified by a conjunctive query).

Mior [18] approaches the issue of finding nested functional and inclusion dependencies in JSON data, adapting available algorithms for relational data. However, this approach does not consider the special case of conditional functional dependencies.

4 THE TAGGER APPROACH

Our discovery of tagged unions relies on a relational encoding of all JSON objects reachable by the same labeled path from the document root. The underlying assumption is that such objects have related semantics. We next provide the intuition behind this encoding, describe the discovery of dependencies, and finally, how we can capture them as constraints in JSON Schema.

Relational encoding. The relational encoding assigns two columns for each property: one for its value and one for its type. For the GeoJSON data from Figure 1 (left), and the path `/geometries[*]` (using JSONPath syntax, matching the array starting in line 2), each object in the array of geometries corresponds to one tuple, and is assigned some unique identifier. We record the atomic value and subschema of properties `type` and `coordinates`, such as `type`.value and `type`.type.

We encode each in a separate column, and restrict ourselves to recording primitive values (consequently, `coordinates`.value is not captured). We obtain the following tuples for the objects starting in lines 3 and 11:

id	type.value	type.type	coordinates.type
3	"Point"	"string"	t
11	"LineString"	"string"	t'

Above, t and t' encode the two distinct types of coordinates occurring in Figure 1:

t = {"type": "array", "items": {"type": "integer"}}

t' = {"type": "array", "items": {"type": "array", "items": {"type": "integer"}}}

For further details, including examples, we refer to [14].

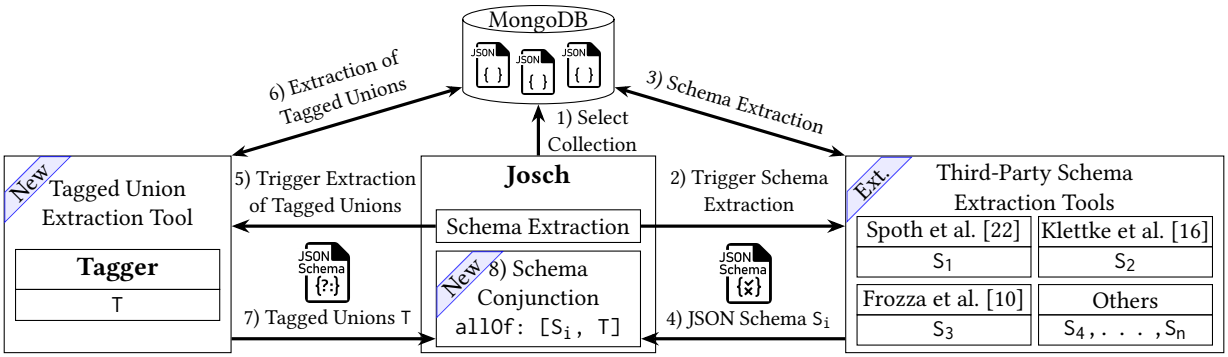

Figure 2: An overview of the system architecture, consisting of Josch, MongoDB, third-party schema extraction tools and Tagger. Contributions original to this demo are labeled as "New". The Josch module labeled "Ext." has been extended by several new third-party schema extraction tools.

Dependencies. From the relational encoding, we derive conditional functional dependencies. We refer to Bohannon et al. [5] for a formal definition, and remain on the level of intuition.

We restrict ourselves to unary dependencies of the form

$$[A.\text{value} = const] \rightarrow [B.\text{type} = \sigma]$$

with only one attribute on the left-hand and right-hand side, where the left-hand-side denotes the value of the candidate tag, and the right-hand-side a property with an implied subschema. Above, A and B are distinct property labels, *const* is a basic-value constant, and σ the implied subschema.

Using state-of-the-art algorithms [7], we can then derive the following dependencies, with t and t' defined as above:

$$[\texttt{type.value} = \text{"Point"}] \rightarrow [\texttt{coordinates.type} = t] \quad (1)$$

$$[\texttt{type.value} = \text{"LineString"}] \rightarrow [\texttt{coordinates.type} = t'] \quad (2)$$

These dependencies describe that if the value of `type` is the string constant `Point`, the type of property `coordinates` must be an array of integers (1) and if the value of `type` is `LineString`, `coordinates` must be an array of integer arrays (2).

In working with real-world JSON data, not all detected dependencies are meaningful. This is an inherent problem in dependency discovery. We can apply a range of practical heuristics, such as a configurable threshold and the removal of trivial constraints, to reduce overfitting to the input. We describe these heuristics in more detail in our workshop paper [14]. Our demo participants may interactively explore their effects.

Encoding in JSON Schema. Once the conditional functional dependencies are discovered, it is straightforward to encode them as constraints in JSON Schema, along the example in Figure 1.

Limitations. Generally, CFD discovery is computationally expensive [20], scaling exponentially in the number of attributes. We therefore restrict our algorithm in the current state to unary CFDs. Moreover, our current implementation is main memory based, which imposes a restriction on the input size.

5 SYSTEM ARCHITECTURE

Figure 2 shows the overall system architecture, where the Tagger approach is integrated into the tool Josch. The modular architecture makes it easy to integrate available schema extraction tools. In our demo, we employ several tools for JSON Schema extraction, such as the implementations of Klettke et al. [16], Frozza et al. [10] and Spoth et al. [22].

The collections of JSON documents are managed in a MongoDB instance. The user selects a collection of JSON documents (step 1), and initiates the extraction of a JSON Schema, leveraging an integrated third-party tool of their choice (step 2). The third-party tool performs schema extraction (step 3) and returns the resulting JSON Schema S_i to Josch (step 4).

Next, the extraction of tagged unions with Tagger is triggered (step 5). Tagger extracts tagged unions from the same documents as the third-party approach (step 6) and returns the JSON Schema encoding of the tagged unions T (step 7). Finally, the conjunction `allOf: [`S_i`, T]` of both schemas is created, yielding a composite schema that enforces the tagged unions *on top of* schema S_i (step 8). Thus, we can immediately leverage state-of-the-art tools for schema extraction.

6 DEMONSTRATION OVERVIEW

Tagger offers an interactive GUI for extracting and comparing JSON Schema. Figure 3 shows a screenshot of this GUI, with a sample JSON document on the left and the extracted composite JSON Schema on the right.

In addition, our demo participants may glimpse "under the hood" of Tagger, and inspect the discovered dependencies before and after applying heuristics for filtering.

Our demo uses several datasets, such as geo-spatial data (GeoJSON data from open government data and OpenStreetMap, data in the related TopoJSON format, encoding maps of Germany and the EU) or metadata about New York Times articles. We include data with missing properties, and discuss from case-to-case, which patterns can be handled by Tagger, and which patterns require further research.

Our planned demonstration scenario is as follows:

(1) We introduce JSON Schema, point out tagged unions occurring in real-world schemas (e.g., as used in Minecraft, and GitHub issue forms), and the concept of applying subschemas conditionally using `if-then-else`.
(2) We give an overview of the capabilities of third-party schema extractors and their limitations.
(3) We then present Tagger. Participants are invited to interact with the GUI, extracting schemas from both synthetic and real world JSON documents provided by us, as well as ones they may enter and edit (see ① and ② in Figure 3):
- A composite schema ②, consisting of the schema extracted by a third party tool ③ and Tagger, can be extracted from the JSON documents ⑥.

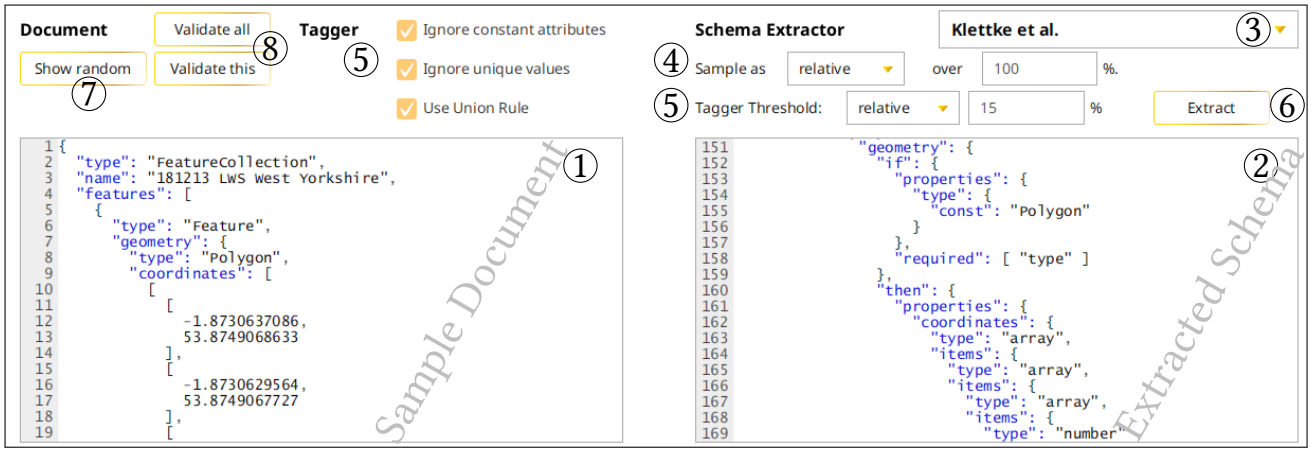

Figure 3: The GUI for schema extraction with Tagger, highlighting the JSON input data ①, the extracted schema ②, selection of the third-party schema extractor ③, settings for sampling ④, and Tagger heuristics ⑤, as well as buttons for schema generation ⑥, random document selection ⑦ and validation of documents against the schema ⑧.

- Participants may experiment with the heuristics available in Tagger ⑤ and the sample size ④, extracting coarse- to well-fitted, and ultimately, over-fitted schemas. This reveals the difficulties in configuring suitable settings. Further, they may find that settings working well for one dataset yield poor results on another.
- We prepare a number of "negative" examples of JSON documents that are not used during schema extraction. These examples deliberately violate the schema to showcase the added value of tagged unions over plain union types. We insert these negative examples in the editable text field ① and validate them against the previously extracted schema.
- Participants may peek under the hood of Tagger, inspecting the dependencies detected by Tagger in a concise format, similar to the notation used in Section 4, and showing the number of detected dependencies before and after heuristics-based filtering.

With our demo, we aim to inspire ideas for extracting schemas that resemble hand-written schemas and therefore, are more likely to be considered comprehensible to human consumers.

Acknowledgments. This work was partly funded by *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation) grant #385808805. We thank Frozza et al. [10] and Spoth et al. [22] for sharing their code. We further thank Thomas Kirz for typesetting the illustrations.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.
[2] Mohamed Amine Baazizi, Clément Berti, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2020. Human-in-the-Loop Schema Inference for Massive JSON Datasets. In *Proc. EDBT*. 635–638.
[3] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Parametric schema inference for massive JSON datasets. *VLDB J.* 28, 4, 497–521.
[4] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2021. An Empirical Study on the "Usage of Not" in Real-World JSON Schema Documents. In *Proc. ER*, Vol. 13011. 102–112.
[5] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2007. Conditional Functional Dependencies for Data Cleaning. In *Proc. ICDE*. 746–755.
[6] Howard Butler, Martin Daly, Allan Doyle, Sean Gillies, Tim Schaub, and Stefan Hagen. 2016. The GeoJSON Format. RFC 7946.
[7] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. 2016. Data Cleaning: Overview and Emerging Challenges. In *Proc. SIGMOD*. 2201–2206.
[8] Pavel Contos and Martin Svoboda. 2020. JSON Schema Inference Approaches. In *Proc. ER*, Vol. 12584. 173–183.
[9] Michael Droettboom. 2022. Understanding JSON Schema. https://json-schema.org/understanding-json-schema/reference/conditionals.html. Draft 2020-12.
[10] Angelo Augusto Frozza, Ronaldo dos Santos Mello, and Felipe de Souza da Costa. 2018. An Approach for Schema Extraction of JSON and Extended JSON Document Collections. In *Proc. IRI*. 356–363.
[11] Michael Fruth, Kai Dauberschmidt, and Stefanie Scherzinger. 2021. Josch: Managing Schemas for NoSQL Document Stores. In *Proc. ICDE*. 2693–2696.
[12] Michael Fruth, Kai Dauberschmidt, and Stefanie Scherzinger. 2021. New Workflows in NoSQL Schema Management. In *Proc. SEA-Data@VLDB*, Vol. 2929. 38–39.
[13] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2013. Discovering Implicit Schemas in JSON Data. In *Proc. ICWE*, Vol. 7977. 68–83.
[14] Stefan Klessinger, Meike Klettke, Uta Störl, and Stefanie Scherzinger. 2022. Extracting JSON Schemas with tagged unions. In *Proc. DEco@VLDB*, Vol. 3306. 27–40.
[15] Meike Klettke, Hannes Awolin, Uta Störl, Daniel Müller, and Stefanie Scherzinger. 2017. Uncovering the evolution history of data lakes. In *Proc. IEEE BigData*. 2462–2471.
[16] Meike Klettke, Uta Störl, and Stefanie Scherzinger. 2015. Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In *Proc. BTW*, Vol. P-241. 425–444.
[17] Benjamin Maiwald, Benjamin Riedle, and Stefanie Scherzinger. 2019. What Are Real JSON Schemas Like? — An Empirical Analysis of Structural Properties. In *Proc. ER*, Vol. 11787. 95–105.
[18] Michael J. Mior. 2021. Fast Discovery of Nested Dependencies on JSON Data. *CoRR* abs/2111.10398 (2021).
[19] Felipe Pezoa, Juan L. Reutter, Fernando Suárez, Martín Ugarte, and Domagoj Vrgoc. 2016. Foundations of JSON Schema. In *Proc. WWW*. 263–273.
[20] Joeri Rammelaere and Floris Geerts. 2018. Revisiting Conditional Functional Dependency Discovery: Splitting the "C" from the "FD". In *Proc. ECML PKDD*, Vol. 11052. 552–568.
[21] Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesús García Molina. 2015. Inferring Versioned Schemas from NoSQL Databases and Its Applications. In *Proc. ER*, Vol. 9381. 467–480.
[22] William Spoth, Oliver Kennedy, Ying Lu, Beda Christoph Hammerschmidt, and Zhen Hua Liu. 2021. Reducing Ambiguity in JSON Schema Discovery. In *Proc. SIGMOD*. 1732–1744.
[23] Ivan Veinhardt Latták. and Pavel Koupil. 2022. A Comparative Analysis of JSON Schema Inference Algorithms. In *Proc. ENASE*. 379–386.
[24] Austin Wright, Henry Andrews, and Geraint Luff. 2018. JSON Schema Validation: A Vocabulary for Structural Validation of JSON. https://datatracker.ietf.org/doc/draft-handrews-json-schema-validation/01/