SIGMOD: G: Capturing Data-inherent Dependencies in JSON Schema Extraction

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

1 PROBLEM AND MOTIVATION

JSON is a popular semi-structured data exchange format widely used across various technological domains. It describes data as key-value pairs, often referred to as *properties*. JSON is essential in web applications for data transmission and in document stores such as MongoDB or Couchbase. Even relational database management systems such as PostgreSQL and MySQL support JSON data types. A sample JSON instance from log data generated in the game World of Warcraft [4] is shown in Fig. 1a. It describes two kinds of events, discriminated by the *value* of property *type*: depending on the value of *type*, either properties *resourceChange* and *resourceChangeType* or property *overheal* are present. Although *JSON instances* are self-describing, they may be accompanied by an explicitly declared schema, commonly encoded in the JSON Schema language.

JSON Schema [1] allows to describe and constrain JSON data. It is the de-facto standard for schema description in JSON and adopted across many different use cases. Schemastore.org [2] lists over 800 curated and publicly available schemas, providing specifications ranging from configuration files, workflows, and pipelines, to components of content management systems, and video games. JSON Schema is supported by a wide range of tools and libraries in many different programming languages. It allows data analysts to define and enforce constraints on the data, which aids in identifying and correcting errors in JSON data sets. The conformity of a JSON instance to a JSON Schema can be analyzed with a wide range of *validation tools*. This improves the reliability and quality of data. Furthermore, the schema provides a documentation of the data structure to data consumers.

Consider, again, our example of log data from World of Warcraft. A JSON Schema description of the data is given in Fig. 1b. This schema enforces that a document conforms to exactly one (indicated by *oneOf*) of two subschemas: (1) a string *type* and two numbers *ResourceChange* and *ResourceChangeType* with no additional properties or (2) a string *type* and an optional number *overheal* with no additional properties. This constitutes a *union type* description of the JSON instances in Fig. 1a.

In practice, JSON instances often come without a schema. Consequently, automatic *extraction of schemas* from collections of JSON instances is an actively researched field, with practical applications in data management (e.g., describing data lakes) and in software engineering (e.g., designing robust web APIs) but also in fields outside of computer science, where a precise description of data structures and constraints on them are valuable.

Virtually all existing approaches (e.g., [6, 17, 20, 25, 26]) focus on capturing the structure of JSON instances and largely ignore atomic property values, apart from deriving basic types (e.g., strings, integers) or computing statistics such as minimum and maximum values. In particular, they ignore dependencies between property values and subschemas of sibling properties. However, an empirical analysis over a large corpus of real-world JSON Schema declarations revealed that hand-crafted schemas do contain conditional dependencies, where the value of one property has implications for its sibling properties [7].

```
{"type": "resourcechange",
 "resourceChange": 30,
 "resourceChangeType": 17},
```

```
{"type": "heal",
 "overheal": 2548}
```

(a) Two JSON instances.

```
{"oneOf": [
  {"type": "object",
    "properties": {
      "type": {"type": "string"},
      "resourceChange": {"type": "number"},
      "resourceChangeType": {"type": "number"}},
    "required": ["type", "resourceChange",
                 "resourceChangeType"],
    "additionalProperties": false},
  {"type": "object",
    "properties": {
      "type": {"type": "string"},
      "overheal": {"type": "number"}},
    "required": ["type"],
    "additionalProperties": false}]}
```

(b) JSON Schema definition for lines 2-9 in (a), using a union type.

```
{"oneOf": [
  {"type": "object",
    "properties": {
      "type": {"const": "resourcechange"},
      "resourceChange": {"type": "number"},
      "resourceChangeType": {"type": "number"}},
    "required": ["type", "resourceChange",
                 "resourceChangeType"],
    "additionalProperties": false},
  {"type": "object",
    "properties": {
      "type": {"const": "heal"},
      "overheal": {"type": "number"}},
    "required": ["type"]}],
    "additionalProperties": false}
```

(c) JSON Schema definition for lines 2-9 in (a), using a tagged union.

Figure 1: World of Warcraft log data (edited for conciseness).

Once again, consider our example in Fig. 1. State-of-the-art approaches for JSON Schema extraction create a schema similar to Fig. 1b. However, current approaches are not designed to detect that the occurrence of these properties depends on the value of property *type*. A sample encoding of this dependency is shown in Fig. 1c. Although this schema looks very similar to the schema discussed in Fig. 1b, it offers a fundamentally different and much more precise description of the data: instead of specifying the occurrence of properties *ResourceChange* and *ResourceChangeType* (or *overheal*) for any string *type*, it describes a *tagged union*, enforcing that these properties may only occur when *type* has the value "resourcechange" (or "heal").

We target the challenge of discovering tagged unions in JSON data and provide the first well-principled approach to detect the conditional existence of (at least) one property based on the values of another property (which we call *tag*). Our approach is theoretically based on extended Conditional Functional Dependencies (eCFDs) [10], and is accompanied by a working implementation and an evaluation.

2 BACKGROUND AND RELATED WORK

JSON data model. We adopt the grammar from [5] to describe the syntax of JSON values, specifically basic values, objects, and arrays. Basic values B consist of strings s, numbers n, Booleans, and the null value. Objects O represent sets of key-value pairs, and arrays A represent sequences of values.

$$\begin{array}{lll} J ::= & B \mid O \mid A & \\ B ::= & \text{null} \mid \text{true} \mid \text{false} \mid n \mid s & n \in \text{Num}, s \in \text{Str} \\ O ::= & \{l_1 : J_1, \ldots, l_n : J_n\} & n \geq 0, \quad i \neq j \Rightarrow l_i \neq l_j \\ A ::= & [J_1, \ldots, J_n] & n \geq 0 \end{array}$$

JSON Schema. JSON Schema is the de-facto standard for defining the structure of JSON data. The JSON Schema language uses JSON syntax. For a formal introduction of the JSON Schema language, we refer to Bourhis et al. [9] and Pezoa et al. [23].

JSON schema extraction. Recent surveys [11, 27] offer an overview of JSON schema extraction. Several of the approaches examined support the extraction of union types [6, 17, 20, 25] but they do not support tagged unions. Notably, Baazizi et al. [6] outline how their approach — based on typing — could be extended to recognize such patterns. We pursue a different approach, relying on the discovery of dependencies, which we capture as constraints in JSON Schema.

Spoth et al. [26] use a clustering-based approach to reduce ambiguities in schema extraction, for instance, collections being encoded as objects instead of arrays. Mior [22] presents a monoid-based approach to schema extraction, offering a configurable set of features, including additional data statistics or the discovery of enums (i.e., defining properties that only assume a small number of different values through their values instead of their data type). Durner et al. [13] propose an approach for fast JSON data analysis that divides the data into *tiles* and extracts local schemas. However, tagged unions are considered in neither of these approaches.

Conditional functional dependencies. Our previous work on JSON schema extraction [18] detects *tagged unions*, where the type of a property depends on the value of another property, the so-called *tag*. This earlier approach relies on the detection of Conditional Functional Dependencies (CFDs) [8], CFDs were first introduced for relational data, in the context of data cleaning. Several approaches for CFD mining have been proposed [15, 16, 21, 24].

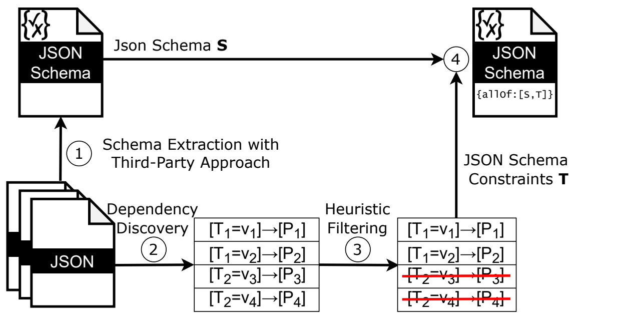


Figure 2: Conceptual Architecture Overview

Bravo et al. [10] propose extended Conditional Functional Dependencies (eCFDs). eCFDs extend CFDs, since they support negation and disjunction, yet satisfiability remains decidable. In the approach presented here, we employ eCFDs to capture dependencies in the data, allowing us to also support optional properties. We discuss eCFDs in more detail in Section 3.

Schema and constraint definition. XML is widely regarded as the precursor to JSON. It is a semi-structured data format with implicit structural information and an optional explicit schema. Schema languages for XML, such as XML Schema [28], support the definition of constraints. In version 1.1 of XML Schema, *assertions* allow to encode constructs such as tagged unions. Yet, to the best of our knowledge, there is no work on the automatic detection of tagged unions in XML data.

Metrics for Schema Quality. Evaluating the quality of extracted Schemas faces the challenge that there is usually no ground-truth schema available. In the existing literature on JSON Schema extraction, there is no agreed-upon metric for the evaluation of JSON Schemas. Baazizi et al. [6] report the type size of extracted schemas as a measure of succinctness. Spoth et al. [26] allocate a randomly selected subset of JSON data for validation, while the remainder is utilized for schema extraction. They then assess the recall of the extracted schema by determining the fraction of the validation set that conforms to the extracted schema. This gives a notion of how well the schema generalizes to unseen data, a desirable property in an environment where additional data is encountered regularly. Further, they introduced *schema entropy* as a metric to approximate the precision of a (JSON) schema. It measures the number of possible combinations of properties (and their types) that a schema permits. We discuss schema entropy in more detail in Section 4.

3 UNIQUENESS AND APPROACH

Existing approaches for JSON Schema extraction largely focus on the structure of the data. Although they are able to detect the co-occurrence of certain properties, they are unable to recognize whether the co-occurrence depends on the values of a certain property. We present the first approach that, by considering dependencies in JSON data, is able to detect whether certain value-based conditions for co-occurrence apply. Our goal is to produce schemas that precisely capture these constraints and that are also comprehensible for humans.

ID	*TAG_VALUE*	*PROPERTY_NAME*
1	resourcechange	resourceChange
1	resourcechange	resourceChangeType
7	heal	overheal

(a) Relational encoding of World of Warcraft data in Fig. 1a.

$\psi_1 = (R_1 : [\mathit{TAG\_VALUE}] \rightarrow \emptyset, \mathit{PROPERTY\_NAME}, T_1)$, where the pattern tableau T_1 is

TAG_VALUE	*PROPERTY_NAME*	$\emptyset$
{resourcechange}	{resourceChange, resourceChangeType}	
{heal}	{overheal}	

(b) eCFD encoding the constraints in Figure 1.

Figure 3: (a) A relational encoding of the JSON data in Fig. 1a and (b) an eCFD encoding the dependency between the value of tag *condition* and the existence of sibling properties.

Architecture. A conceptual overview of our system is shown in Fig. 2. Starting from a set of JSON documents on the bottom left, we (1) extract a JSON Schema S using any third-party approach. In the same JSON documents, we (2) perform eCFD discovery, using the relational encoding we described. This yields a (potentially large) list of candidates that hold on the data. Next, we (3) apply our configurable heuristics[1], yielding a reduced list of eCFDs that will be encoded in the schema. Finally, we (4) translate the constraints imposed by the eCFDs into JSON Schema T and create a conjunctive schema with the JSON Schema S. The resulting schema `{"allOf":[`S, T`]}` enforces that an instance must conform to both schemas.

Relational Encoding. Our approach relies on a relational encoding of JSON data. Specifically, we use triples of the form:

$$(ID, TAG\_VALUE, PROPERT\_NAME)$$

ID uniquely identifies every instance of tags, TAG_VALUE contains the value of the corresponding tag, and $PROPERTY\_NAME$ contains a property that co-occurs as a sibling of the tag. The relational encoding contains one triple for each combination of a tag and a co-occurring property. An exemplary relational encoding of the data in Fig. 1a is sketched in Fig. 3a.

Dependency Discovery. To capture the existence of properties based on the value of a tag, we use eCFDs, proposed by Bravo et al. [10]. Their support for disjunction allows to capture dependencies between a tag value and the existence of an *optional* property, e.g., in our log data example, that a "heal" event implies that a property *overheal* may occur.

Traditionally, eCFDs and CFDs are employed in relational data cleaning and repair [14]. We are, to our knowledge, the first to employ eCFDs and CFDs as a basis for deriving more sophisticated and informative schemas for semi-structured, hierarchical data.

In introducing eCFDs, we remain on the level of intuition and refer to the existing work for a formal definition. In Fig. 3b, an eCFD ψ_1, derived from the JSON data in Fig. 1 is provided. Generally, an eCFD $\psi = (R : X \rightarrow Y, Y_p, T_p)$ consists of a relational schema R and an embedded Functional Dependency $X \rightarrow Y$. Here, X, Y, Y_p are subsets of attributes in R, with $Y \cap Y_p = \emptyset$ and T_p is a pattern tableau. In our example, ψ_1 enforces that if property *type* has the value "resourcechange", then any co-occurring property must have the name *resourceChange* or *resourceChangeType* and if property *type* has the value "heal", then any co-occurring property must have the name *overheal.*

[1]This is only a conceptual overview of the architecture. For efficiency, we actually apply the heuristic filters as early as possible during the discovery process.

Limitations. In real-world schemas, such as those available on SchemaStore.org [2], we have observed that such dependencies are usually declared between sibling properties of the same JSON object. Accordingly, we restrict our detection to tags and properties reachable by the same path. Note that this is merely a practical decision and not a technical limitation of our approach. Further, our approach only captures dependencies that are satisfied by all JSON instances. Robustness against irregularities, e.g., rare violations of dependencies due to dirty data,is subject to future work.

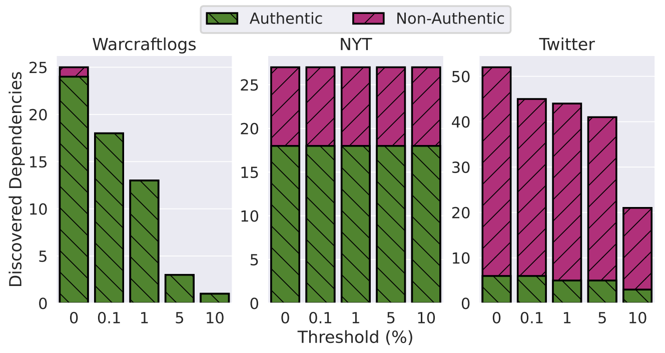
Heuristic filtering. Heuristic filtering of dependencies is vital in detecting dependencies in relational data, to improve performance, and to avoid overfitting [12]. In our context of JSON schema extraction, it is even more important to discard dependencies with low support in the data; otherwise, the extracted schema becomes too verbose for humans to consume. We therefore employ several heuristics to prune the search space and to remove unfit candidates:

1) We ignore *required* properties as the right side of an eCFD.
2) Analogously, a property with a single-valued domain cannot be a tag (i.e., on the left side of an eCFD).
3) Further, we expect properties that serve as tags to have a relatively small domain size. Our implementation allows to define a maximum domain size for candidate tags.
4) We argue that empirically, properties with complex data types (i.e. *objects* or *arrays*) usually do not serve as a tag and only consider tags with primitive values.
5) Properties that rarely occur in the data may coincidentally (and incorrectly) display the patterns we are looking for. We implement a configurable threshold that requires properties to occur with certain support to be considered.

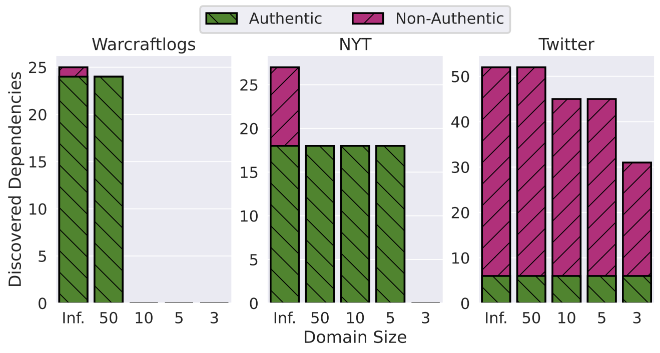
4 RESULTS AND CONTRIBUTIONS

We now present our experiments using real-world JSON dataset of event logs, New York Times articles, and Twitter data.

Experimental Evaluation. Our experimental evaluation is based on the metrics used in our preliminary work [18], which has a reproduction package available online [19]. We evaluated the number of dependencies discovered in different datasets for the varying configurations of our heuristics. Specifically, we investigate (1) how changing the threshold for a fixed maximum domain size influences the number of discovered dependencies and (2) how changing the maximum domain size for a fixed threshold affects the number of discovered dependencies. We manually classify the discovered dependencies as authentic (i.e., dependencies that are intuitive and do not just hold coincidentally on the data) and non-authentic. This manual classification is, of course, subjective, and thus a potential threat to validity. We consider each individual property that depends on a certain tag as a single dependency. Recall, for instance, the example in Fig. 1c, lines 2-9: the properties *resourceChange* and *resourceChangeType* must be present if the *type* is "resourcechange"; we consider these as two dependencies.



(a) Discovered eCFDs for varying Threshold, unlimited Domain Size



(b) Discovered eCFDs for varying Domain Size, no Threshold

Figure 4: Effect of Threshold and Domain Size Heuristics on Disovered eCFDs

Datasets. We investigate three different datasets: (1) The *Warcraftlogs* dataset contains 6005 JSON instances describing combat log events obtained through the API of an online service that collects and aggregates player-provided data from the video game World of Warcraft [4]. (2) The *NYT* dataset, obtained through the New York Times archive API [3], consists of 4418 JSON instances that represent NYT articles from September 2019. (3) The *Twitter* dataset, obtained through the Twitter API (before Twitter was renamed to *X*), contains the messages and metadata of 19316 tweets, represented as individual JSON instances.

Evaluation of Threshold Heuristic. We analyze the effect of the threshold on the number of discovered dependencies, shown in Fig. 4a. The figure shows the number of discovered authentic and non-authentic dependencies for different thresholds. A threshold of 0% indicates that the threshold heuristic is disabled. The domain size of potential tags is not limited.

In the *Warcraftlogs* dataset, we discover 24 authentic and 1 non-authentic dependency with the threshold being disabled. All authentic dependencies describe the existence of sibling properties based on the value of property *type*, akin to the example in Fig. 1, whereas the non-authentic dependency describes the existence of a property that only occurs twice in the whole dataset and coincidentally appears only for a certain ID value. Increasing the threshold to 0.1% removes the non-authentic dependency, but also 6 authentic dependencies. Further increasing the threshold worsens the results by removing an increasing amount of authentic dependencies.

In the *NYT* dataset, we find 27 dependencies with disabled threshold of which 18 are authentic. The number of discovered dependencies is invariant for the threshold values we investigated. This can be attributed to the fact that all dependencies, authentic and non-authentic, enforce the existence of two disjoint sets of properties, describing either thumbnails or extra large images in more detail. We consider the conditional existence of these properties based on the values of properties describing image dimensions (specifically, properties *height* and *width*) as non-authentic dependencies.

The dataset *Twitter* features 52 dependencies when disabling the threshold. We classify only 6 of these dependencies as authentic. Increasing the threshold reduces the number of non-authentic dependencies more drastically than the number of authentic dependencies, but even for a threshold of 10%, the majority of dependencies are non-authentic. The large proportion of non-authentic dependencies can be attributed to the recursive structure of the dataset: tweets can reference other tweets, and thus we encounter the same structure on different nesting-depths (i.e., on different paths). Our tool identifies properties by their full path and is unable to detect recursion, causing our heuristics to fall short if some dependencies only occur at a certain nesting depth.

Evaluation of Tag Domain Size Heuristic. We now investigate the effect of the maximum tag domain size on the number of dependencies discovered, shown in Fig. 4b. The figure shows the number of authentic and non-authentic dependencies discovered for varying limits on the maximum tag domain size ("Inf." meaning unlimited) and with the threshold heuristic disabled. Note that the leftmost bar for each dataset in Fig. 4b has the same configuration as the leftmost bars in Fig. 4a . For a more detailed discussion of the datasets, we refer to the previous paragraphs.

In the *Warcraftlogs* data, imposing a maximum tag domain size of 50 removes the non-authentic dependency without removing any authentic dependency. Further reducing the maximum tag domain size to 10 removes all authentic dependencies because they all depend on the same tag, *type*, which assumes 23 different values. However, an appropriate limit on the domain size is a highly effective filter, leaving only (and all) authentic dependencies.

Likewise, choosing a suitable limit for the maximum type domain size is very effective on the *NYT* dataset, removing all non-authentic dependencies while keeping all authentic ones.

Finally, in the *Twitter* dataset, our domain size heuristic can barely reduce the number of non-authentic dependencies. As discussed above, this can be attributed to the recursive structure of the data, which our approach is currently unable to handle adequately.

Schema Entropy. Requirements on a (JSON) schema depend on individual use cases and are highly subjective. Consequently, there is no agreed-upon metric for JSON Schema quality as of yet. The existing literature uses different approaches for measuring the quality of extracted schemas, including Schema Entropy introduced by Spoth et al. [26]. Schema entropy was introduced as a proxy measure for the precision of a schema. It measures the number of possible combinations of properties (and types) a schema permits. In a basic schema description without conjunction, disjunction, and negation, the number of property combinations (P) grows exponentially with the amount of optional properties. The schema entropy value is given as $\log_2 P$.

In the following example, we illustrate that this metric is unable to capture the higher precision that a schema with tagged unions offers over a schema using union types.

Example. Consider the JSON Schema snippet in Fig. 1b. The schema specifies an object with either (1) property *type* and two required (i.e., not optional) properties *resourceChange* and *resourceChangeType* or (2) property *type* and an optional property *overheal*. Alternative (1) allows for exactly one combination of properties consisting of all properties that are defined in the subschema. Alternative (2) allows for two combinations of properties, either just the property *type* or both, properties *type* and *overheal* must be present. Thus, the schema entropy value of this schema is $\log_2 3$. Now, let us consider the JSON Schema snippet in Fig. 1c, refining the previous schema by enforcing a specific value for the property *type*, depending on the co-occurring properties. However, the schema permits the same combinations of properties (and types) and thus, both schemas have equal schema entropies.

Extending the concept of schema entropy to capture more intricate schema design patterns, such as tagged unions, would be highly beneficial to the field, allowing one to compare different schema extraction approaches more in-depth. Yet, designing such a metric remains an open challenge.

Conclusion. With this work, we advance the field of JSON Schema extraction by 1) extending our early prototype [18], moving from CFDs to eCFDs and thereby allowing to detect more complex dependencies; 2) contributing a novel approach based on eCFDs to capture common patterns in JSON data. Specifically, we target the schema design pattern where the value of one property implies the (optional) existence of sibling properties. This is a novel domain of application for eCFDs, which have traditionally been restricted to data cleaning and data repair in relational data. We thereby lift the applicability of CFDs to a more complex data model; 3) developing effective heuristics to reduce computational complexity and avoid overfitting; and ultimately, 4) performing an experimental evaluation of our heuristics based on real-world JSON data.

Although our experimental analysis showed the effectiveness of our heuristics, it also revealed the necessity to refine our approach to recursive data structures in the future. In future work, we, moreover, intend to systematically extend our experimental evaluation, including a scalability analysis on datasets of different sizes. We also plan to analyze runtime and the quality of results when employing sampling techniques. Exploring the use of large language models to automatically decide whether discovered dependencies are authentic is another interesting direction for future work.

Acknowledgments. This work was funded by *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation) grant #385808805. I thank Thomas Hütter for the fruitful discussions on schema entropy. I thank Stefanie Scherzinger for her continuous guidance and support.

REFERENCES

[1] 2024. JSON Schema. Retrieved April 28, 2024 from https://json-schema.org/
[2] 2024. JSON SchemaStore. Retrieved April 28, 2024 from https://www.schemastore.org/
[3] 2024. New York Times Archive API. Retrieved April 28, 2024 from https://developer.nytimes.com/docs/archive-product/1/overview
[4] 2024. Warcraft Logs. Retrieved April 28, 2024 from https://www.warcraftlogs.com/api/docs
[5] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2022. Witness Generation for JSON Schema. *CoRR* abs/2202.12849 (2022). arXiv:2202.12849 https://arxiv.org/abs/2202.12849
[6] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Parametric schema inference for massive JSON datasets. *VLDB J.* 28, 4, 497–521.
[7] 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.* 102–112.
[8] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2007. Conditional Functional Dependencies for Data Cleaning. In *Proc. ICDE.* 746–755.
[9] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. 2017. JSON: Data model, Query languages and Schema specification. In *Proc. PODS.* 123–135.
[10] Loreto Bravo, Wenfei Fan, Floris Geerts, and Shuai Ma. 2008. Increasing the Expressivity of Conditional Functional Dependencies without Extra Complexity. In *Proc. ICDE.* 516–525.
[11] Pavel Contos and Martin Svoboda. 2020. JSON Schema Inference Approaches. In *Proc. ER (Workshops).* 173–183.
[12] Yuefeng Du, Derong Shen, Tiezheng Nie, Yue Kou, and Ge Yu. 2017. Discovering context-aware conditional functional dependencies. *Frontiers Comput. Sci.* 11, 4 (2017), 688–701.
[13] Dominik Durner, Viktor Leis, and Thomas Neumann. 2021. JSON Tiles: Fast Analytics on Semi-Structured Data. In *Proc. SIGMOD.* 445–458.
[14] Wenfei Fan and Floris Geerts. 2012. *Foundations of Data Quality Management.* Morgan & Claypool Publishers.
[15] Wenfei Fan, Floris Geerts, Laks V. S. Lakshmanan, and Ming Xiong. 2009. Discovering Conditional Functional Dependencies. In *Proc. ICDE.* 1231–1234.
[16] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. 2011. Discovering Conditional Functional Dependencies. *IEEE Trans. Knowl. Data Eng.* 23, 5 (2011), 683–698.
[17] 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.
[18] Stefan Klessinger, Meike Klettke, Uta Störl, and Stefanie Scherzinger. 2022. Extracting JSON Schemas with Tagged Unions. In *Proc. DEco@VLDB.*
[19] Stefan Klessinger, Meike Klettke, Uta Störl, and Stefanie Scherzinger. 2022. *Extracting JSON Schemas with Tagged Unions (Reproduction Package).* https://doi.org/10.5281/zenodo.6985647
[20] 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. GI, 425–444.
[21] Jiuyong Li, Jixue Liu, Hannu Toivonen, and Jianming Yong. 2013. Effective Pruning for the Discovery of Conditional Functional Dependencies. *Comput. J.* 56, 3 (2013), 378–392.
[22] Michael J. Mior. 2023. JSONoid: Monoid-based Enrichment for Configurable and Scalable Data-Driven Schema Discovery. arXiv:2307.03113 [cs.DB]
[23] Felipe Pezoa, Juan L. Reutter, Fernando Suárez, Martín Ugarte, and Domagoj Vrgoc. 2016. Foundations of JSON Schema. In *Proc. WWW.* 263–273.
[24] Joeri Rammelaere and Floris Geerts. 2018. Revisiting Conditional Functional Dependency Discovery: Splitting the "C" from the "FD". In *Proc. ECML PKDD.* 552–568.
[25] Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesús García Molina. 2015. Inferring Versioned Schemas from NoSQL Databases and Its Applications. In *Proc. ER.* 467–480.
[26] 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.
[27] Ivan Veinhardt Latták. and Pavel Koupil. 2022. A Comparative Analysis of JSON Schema Inference Algorithms. In *Proc. ENASE.* 379–386.
[28] W3C. 2012. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures.* W3C Recommendation. https://www.w3.org/TR/xmlschema11-1/