JSON Schema Inference Approaches

Pavel Čontoš and Martin Svoboda(✉)

Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic
{contos,svoboda}@ksi.mff.cuni.cz

Abstract. Since the traditional relational database systems are not capable of following the contemporary requirements on Big Data processing, a family of NoSQL databases emerged. It is not an exception for such systems not to require an explicit schema for the data they store. Nevertheless, application developers must maintain at least the so-called implicit schema. In certain situations, however, the presence of an explicit schema is still necessary, and so it makes sense to propose methods capable of schema inference just from the structure of the available data. In the context of document NoSQL databases, namely those assuming the JSON data format, we focus on several representatives of the existing inference approaches and provide their thorough comparison. Although they are often based on similar principles, their features, support for the detection of references, union types, or required and optional properties differ greatly. We believe that without adequately tackling their disadvantages we identified, uniform schema inference and modeling of the multi-model data simply cannot be pursued straightforwardly.

Keywords: NoSQL databases · Schema inference · JSON

1 Introduction

An interesting feature of the majority of *NoSQL databases*, a newly emerged family of database systems, is the absence of an explicit schema for the stored data, which allows for greater flexibility and simplicity. Nevertheless, various situations still require the knowledge of the schema when performing operations such as data querying, migration, or evolution, and so there is a growing interest in *schema inference* approaches that allow us to create a schema when the explicit one simply does not exist.

In particular, there already exist several schema inference approaches for the *aggregate-oriented* group of NoSQL databases, i.e., databases based on the *key-value*, *wide-column* or *document* models. However, the inference process itself is nontrivial, and the resulting schemas often suffer from various issues. For example, derived entities may contain a large number of properties, including properties of the same name having different data types, as well as various kinds

This work was supported by Czech Science Foundation project 20-22276S and Charles University SVV project 260451.

of references between the documents (aggregates). The inferred schemas may be complicated even from the point of view of data modeling when commonly available tools and modeling languages would most likely be used (e.g., UML [18] is not capable of dealing with the mentioned properties of the same name but different types). Another obstacle also arises when querying the data. When a property data type or property content interpretation changes over time, it is difficult to properly construct an evolved query expression that returns the originally intended result.

In this paper, we focus on several existing representatives of the schema inference approaches dealing with collections of JSON [13] documents, namely the following ones: 1) approach proposed by Sevilla et al. [19] working with a concept of distinct versions of entities, 2) approach by Klettke et al. [15] utilizing a graph structure for the schema representation, capable of the detection of outliers, 3) approach by Baazizi et al. [2] that introduces compact yet complex and massively parallelizable schema inference method, 4) approach by Cánovas et al. [6] capable of inferring a schema from multiple collections of documents, and, finally, 5) recent approach by Frozza et al. [10] that is able to infer schemas including data types as they are introduced in BSON (*Binary JSON*) [5].

Our main goal is to provide a static analysis of these approaches to find and identify their strengths and weaknesses, compare them with each other, and verify how each individual approach copes with specific characteristics of JSON documents and non-uniform semi-structured data in general. We compare these approaches based on their capabilities to i) handle properties of the same name but different types, ii) distinguish required and optional properties based on their frequency of occurrences, iii) distinguish reference relationships between documents and nested documents, iv) work with arrays, v) represent the resulting schemas using proprietary or widely used means, respectively, and vi) scale.

The paper is organized as follows. In Sect. 2, we briefly summarize the JSON data format and show the constructs we are dealing with. Section 3 describes the actual schema inference process for the selected existing approaches and illustrates their differences on a simple running example and the corresponding inferred schemas. We then mutually compare these approaches in Sect. 4, present the related work in Sect. 5, and conclude in Sect. 6.

2 JSON Data Format and JSON Schema

JavaScript Object Notation (JSON) [13] is a widely used human-readable textual data format suitable for representing semi-structured, schema-less, often non-uniform data. From the logical point of view, it is based on trees.

The unit of data stored in a JSON document database is a *document* and corresponds to a single JSON *object*. An object consists of an unordered set of *name-value* pairs called *properties*. Property *name* is a string, while *value* can be atomic of any primitive type *string*, *number*, and *boolean*, or structured in a form of an embedded object or *array*. If no value is to be assigned, a property may be present and bound to the `null` meta-value. Although property names *should* be unique, we actually expect they *must* be (in accordance to the existing systems such as MongoDB[1]). Finally, an array is an ordered collection of items, which can be either atomic values or nested objects or arrays, possibly with duplicates.

```
{ "name": "Smíchovské Nádraží",
  "location": { "latitude": "50.0608367N", "longitude": "14.4093753E" },
  "timetable": [
    { "line": "B",
      "departure": [ "10:10", "10:20", "10:30", "10:40", "10:50" ]
    } ] }
```

```
{ "_id": "SMN_E",
  "name": "Smíchovské Nádraží",
  "location": { "latitude": "50.0597611N", "longitude": "14.4092244E" },
  "timetable": [
    { "line": 190,
      "stop_id": "NBE_O",
      "departure": [ "10:00", "10:15", "10:30" ] },
    { "line": 125,
      "stop_id": "SKA_A",
      "departure": [ "10:05" ] } ] }
```

Fig. 1. Collection of two sample JSON documents

In order to illustrate the mutual differences of the selected schema inference approaches, we will use a collection of two sample JSON documents, each describing a public transport stop in Prague. They are depicted in Fig. 1.

We believe we can omit a detailed description of the involved properties (as their meaning is self-explanatory) and only focus on parts that will become important in relation to the schema inference: i) data type of property `line` is a string in case of the first document while in the second one it is a number, ii) properties such as `_id` and `stop_id` appear only in the second document, iii) property `_id` may be treated as a document identifier while property `stop_id` as a reference, iv) property `departure` has a different number of elements across the documents, and v) some document databases allow us to use extended data types defined in BSON, thus we could easily assume that `_id` would be a property of type `ObjectID` instead of an ordinary string.

To validate the structure of JSON documents, JSON Schema [14] was proposed as a human and machine-readable format. It takes into account the evolution of JSON documents and features of schema-free databases. E.g., JSON Schema introduces i) union types, ii) distinguishes between required and optional properties, and iii) allows us to use extended data types, i.e. `ObjectID`.

3 JSON Schema Inference Approaches

In this section, we describe basic principles of the selected approaches. For each one of them, we also present the resulting inferred schemas for our sample input JSON collection so that these techniques can easily be compared together.

[1] https://www.mongodb.com/.

Let us start with an approach proposed by Sevilla et al. [19]. Within three steps and using *MapReduce* [7], their algorithm i) reduces an input collection of JSON documents into a set of structurally different documents, ii) discovers various versions of entities and their properties, and iii) identifies relationships between these entities, including references.

Versioned schema inferred by this approach is illustrated using JSON format in Fig. 2. The algorithm detects all the entities and properties while entities are further divided into versions that differ by the existence of such a property, its data type, or reference. The approach is able to detect references between the documents (property `stop_id` refers to entity `Stop`). The existence of versioned entities avoids the necessity of having union types, and the approach also distinguishes between the required and optional properties (intersection or union of properties across versions of the same entity need to be calculated). The only inferred data types are standard types defined by JSON.

```
{ "entities":
  { "Location":
      { "Location_1": { "latitude": "String", "longitude": "String" } },
    "Stop":
      { "Stop_1":
          { "name": "String", "location": "Location_1",
            "timetable": [ "Timetable_1" ] },
        "Stop_2":
          { "_id": "String", "name": "String", "location": "Location_1",
            "timetable": [ "Timetable_2" ] } },
    "Timetable": {
      "Timetable_1": { "line": "String", "departure": [ "String" ] },
      "Timetable_2": { "line": "String", "stop_id": "ref(Stop)",
        "departure": [ "String" ] } } } }
```

Fig. 2. Sample inferred schema for the Sevilla et al. algorithm [19]

Klettke et al. [15] proposed a schema inference algorithm for JSON document collections in MongoDB. The approach works with a so-called *Structure Identification Graph* (SIG) containing everything needed for the inference.

Nodes in this graph represent JSON properties (one node for each distinct property name), while edges model the hierarchical structure of the documents for which the schema is being constructed. Besides other metadata, each node is associated with a so-called *nodeList* describing the detected occurrences of a given property in the input documents. Similarly, each edge is associated with a so-called *edgeList* describing where these occurrences are structurally located.

The sample inferred schema, provided in Fig. 3, is described using JSON Schema. The algorithm is able to detect all the entities, including properties that may be assigned by multiple data types. This *union* type is used, e.g., in case of a property `line`. The approach does not detect references and uses only data types known by JSON, i.e., it does not use any extended data type. The approach also detects required properties (e.g., properties `latitude` and `longitude` in `location`).

```
{ "type": "object", "properties":
  { "_id": { "type": "string" }, "name": { "type": "string" },
    "location":
      { "type": "object", "properties":
        { "latitude": { "type": "string" }, "longitude": { "type": "string" } },
        "required": [ "latitude", "longitude" ] },
    "timetable":
      { "type": "array", "items":
        { "type": "object", "properties":
          { "line": { "oneOf": [ { "type": "string" }, { "type": "integer" } ] },
            "stop_id": { "type": "string" },
            "departure": { "type": "array", "items": { "type": "string" } } },
          "required": [ "line", "departure" ] } } },
  "required": [ "name", "location", "timetable" ] }
```

Fig. 3. Sample inferred schema for the Klettke et al. algorithm [15]

Baazizi et al. [2] proposed yet another inference algorithm, in this case consisting of two phases only. Based on *Apache Spark*[2], the input collection of JSON documents is first processed by the *Map* function, so that during the *Reduce* phase the union types, as well as required, optional, and repeated elements are identified.

Schema inferred by this approach, as illustrated in Fig. 4, is represented using a compact proprietary language. The optional properties are marked by a question mark symbol `?` (e.g. properties `_id` and `stop_id`), union types by `+` (property `line` may be either *Str* or *Num*), and *repeated* items of arrays by an asterisk `*` (array `departure` contains elements of type `Str`). Extended data types nor references between the documents are discovered by this approach.

```
{ _id: Str?,
  name: Str,
  location: { latitude: Str, longitude: Str },
  timetable: [ { line: (Str+Num), stop_id: Str?, departure: [ (Str)* ] } ] }
```

Fig. 4. Sample inferred schema for the Baazizi et al. algorithm [2]

Another approach, proposed by Cánovas et al. [6], is deployed in the environment of web services providing collections of JSON documents, where each collection is expected to contain documents with similar but not necessarily the same structure. The approach is based on an iterative process in which each JSON document contributes to the extension of an already generated schema. This process consists of three parts: i) extraction of a schema for every document, ii) creation of a schema for each collection, and iii) merging of the schemas of individual collections together into a single resulting schema.

Schema inferred by this approach, illustrated in Fig. 5, is visualized by UML. The approach does not detect references between documents, so the relationships

[2] http://spark.apache.org/.

are only modeled through nested documents. The algorithm does not recognize union types nor optional properties. In order to simplify the visualization, it uses the most generic data types (e.g., property `line` is of a type *EString*). The approach also does not detect arrays (property `departure` is of a simple type *EString* instead of an array of strings). No extended data types are discovered.

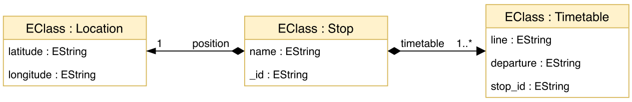

Fig. 5. Sample inferred schema for the Cánovas et al. algorithm [6]

The last representative approach we covered in this paper is the one proposed by Frozza et al. [10], allowing for the inference of a schema for just one collection of JSON documents. Since it also supports the extraction of particular data types from a broader set of atomic types as they are introduced in BSON, it is therefore suitable especially when working with MongoDB database system. The inference process consists of the following four steps: i) creation of a raw schema for individual input documents, ii) grouping of the same raw schemas together, iii) unification of these schemas, and iv) construction of the final global JSON schema.

Schema inferred by this approach, materialized in Fig. 6, is described by JSON Schema. The approach is able to detect union types (e.g., property `line` may be either string or number). It also distinguishes between the required and optional properties (as every discovered entity may always contain a list of required properties), yet no references are discovered by this approach. In our sample data, we could easily derive the property `_id` to be an instance of the `ObjectID` type if that property was originally set to, e.g., `ObjectID("SMN_E")` instead of an ordinary string.

4 Comparison

Having described all the selected schema inference approaches, we can now mutually compare their main characteristics, as well as advantages and disadvantages. In particular, we focus on i) basic principles and scalability of the involved algorithms, i.e., ways how schemas are inferred and proprietary data structures utilized, ii) output formats, i.e., means how the inferred schemas are represented, iii) eventual support for data types beyond the JSON format itself, iv) distinction of required and optional properties in the inferred schemas, v) dealing with properties of the same name but different data types, i.e., distinguishing between simple and union types, and vi) discovering references between documents. Table 1 summarizes the identified differences and observations.

```
{ "$schema": "http://json-schema.org/draft-06/schema",
  "type": "object", "properties":
  { "_id": { "type": "string" }, "name": { "type": "string" },
    "location": {
      "type": "object", "properties":
        { "latitude": { "type": "string" }, "longitude": { "type": "string" } },
      "required": [ "latitude", "longitude" ], "additionalProperties": false },
    "timetable":
      { "type": "array", "items":
        { "type": "object", "properties":
          { "line": { "anyOf": [ "string", "number" ] },
            "stop_id": { "type": "string" },
            "departure":
              { "type": "array", "items": { "type": "string" }, "minItems": 1,
              "additionalItems": true } },
          "required": [ "line", "departure" ], "additionalProperties": false },
        "minItems": 1, "additionalItems": true } },
  "required": [ "name", "location", "timetable" ], "additionalProperties": false }
```

Fig. 6. Sample inferred schema for the Frozza et al. algorithm [10]

Basic Principles and Scalability. The majority of approaches extract schema information from all the documents stored in the input collection without initially reducing its size, i.e., the number of documents. Exceptions include the approaches by Sevilla et al. and Frozza et al., which initially select just sort of a minimal collection of mutually distinct documents such that it can still be correctly used to derive the schema for all the input documents. A common feature of all the approaches is the replacement of values of properties by names of the primitive types encountered. In addition, this step is usually parallelized using distributed solutions such as MapReduce or Apache Spark, which greatly improves the scalability. Up to our knowledge, the only approach that is not parallelized, so scalability is limited, is Frozza et al.

Output Format. The textual JSON Schema format is used for the inferred schema description by the majority of the approaches, yet they differ in the details. Baazizi et al. use their own and minimalistic proprietary language based on the JSON Schema. Baazizi et al. support the repeating type to describe repeated types in arrays, too. Sevilla et al. represent schema as a model that conforms to a schema metamodel [19], which can be textually described by JSON. Finally, Cánovas et al. represent schemas visually as class diagrams.

Table 1. Comparison of the selected approaches

	Sevilla et al.	Klettke et al.	Baazizi et al.	Cánovas et al.	Frozza et al.
Scalability	Yes	Yes	Yes	Yes	No
Output	Model	JSON Schema	Proprietary	Class diagram	JSON Schema
Data Types	JSON	JSON	JSON	JSON	BSON
Optional	Yes	Yes	Yes	No	Yes
Union Type	No	Yes	Yes	No	Yes
References	Yes	No	No	No	No

Additional Data Types. All the approaches support the basic set of primitive types (string, number, and boolean), as well as complex types, i.e., nested objects and arrays, as they are defined by JSON itself. In addition, and as the only approach, Frozza et al. support extended data types introduced by BSON.

Optional Properties. All properties that are contained in all the input documents in a collection are marked as *required*. Otherwise, when a property does not appear in at least one of them, it is marked as *optional*. Apparently, a set of required properties forms the skeleton of all the documents. Therefore, the visualization of a schema containing only these required properties can be significantly more comfortable for the users to grasp, especially when these documents contain a large number of different optional features that would occur only rarely. Furthermore, this visualization gives the users a very good idea of the structure of the documents. The majority of approaches we covered can distinguish between these two kinds of properties. They only differ in the way of their detection. In particular, approaches by Klettke et al. and Frozza et al. calculate the differences in occurrences of individual properties versus occurrences of their parental properties. When a parental property occurs more frequently, the property is marked as optional. Sevilla et al. is able to detect optional properties by the set operations over versions of entities. Baazizi et al. detect the optional properties during the fuse of types. Finally, the approache by Cánovas et al. is not capable of distinguishing the required and optional properties at all. Thus, all the properties in these cases must then be considered as required.

Union Type. The JSON format natively allows for the data evolution, e.g., a situation when properties of newly added objects may have different types compared to the older ones. Schema inference approaches must, therefore, deal with different types occurring within just one property. Most of the examined approaches work with the concept of the *union* type, where a property may contain several different types at once. In contrast, the approaches by Sevilla et al. and Cánovas et al. use just the most generic of the detected types in such cases. The advantage of the union type is accuracy, simply because we do not lose information about the involved data types. On the other hand, the principle of the most generic type is better visualizable and programmable, because we are able to perform (de)serialization through just a single data type, the generic one. For the purpose of the schema visualization, the widely used models, namely UML and ER, cannot associate properties with more than one type at a time, and, thus, it is better to use the most generic type in this case, but at the expense of the loss of the information accuracy, as outlined.

References. The only approach that detects relationships between the documents, i.e., references, is Sevilla et al. When a JSON property is named following the `entityName_id` suffix convention, then the entity named `entityName` is referenced (if it exists). It means that a reference relationship is created between the referring and referenced entities in the inferred schema.

5 Related Work

To a certain extent, several existing JSON schema inference approaches were experimentally compared in works by Frozza et al. [10] and Feliciano [9]. However, they considered different aspects, worked with not that many approaches, and, most importantly, did not assume the multi-model context. Moreover, schema inference is desirable not only for collections of JSON documents but for semi-structured and non-uniform data in general.

In case of mature XML [21], there are a number of heuristic-based [16] and grammar-inferred approaches [3]. Although both JSON and XML are semi-structured hierarchical formats, inference approaches for XML documents are not directly applicable to JSON because of the significant differences between the two formats. While elements in XML are ordered, names of these elements can appear repeatedly, and elements may contain attributes, properties in JSON objects are unordered and without duplicates as for their names.

Although not that many, there are also approaches dealing with other logical models and formats used within the family of NoSQL databases. Wang et al. [20] suggested a schema management approach for document databases, where frequently occurring structures are grouped using hierarchical structures. The approach proposed by DiScala and Abadi [8] solves the problem of transforming JSON documents from key-value repositories into flat relational structures. The inference of schemas for RDF documents is discussed by Gallinucci et al. [11], where aggregate hierarchies are identified. Bouhamoun et al. [4] then focus on the scalable processing of large amounts of RDF data by extracting patterns for existing combinations of individual properties.

JSON Schema is often used to describe an inferred schema from JSON document collections. The formal model of this language is dealt with by Pezoa et al. [17]. Description of the type system of JSON is designed by Baazizi et al. [1].

6 Conclusion

In this paper, we provided a mutual comparison of five selected representative JSON schema inference approaches, each of which solves a different subset of issues arising from the usage of the document NoSQL databases. As observed, especially the detection of references between the individual documents seems to be a challenging issue, not just since only one of the examined approaches actually recognizes such references, however, only to a very limited and questionable extent. Another open area lies in the visualization and modeling of the inferred schemas because the existing tools do not allow us to visualize all the derived constructs, namely, the union type.

We believe that in order to be able to infer schemas even for the non-uniform data maintained within the family of multi-model databases, the identified drawbacks of the existing approaches first need to be sufficiently tackled. Only then the acquired knowledge can be exploited, and the individual existing solutions extended to the unified inference of truly multi-model schemas. This step is apparently not straightforward, as it is envisioned by Holubová et al. [12], where several open and challenging areas of multi-model data processing are outlined.

References

1. Baazizi, M.A., Colazzo, D., Ghelli, G., Sartiani, C.: A type system for interactive JSON schema inference. In: ICALP 2019. LIPIcs, vol. 132, pp. 101:1–101:13 (2019). https://doi.org/10.4230/LIPIcs.ICALP.2019.101
2. Baazizi, M.-A., Colazzo, D., Ghelli, G., Sartiani, C.: Parametric schema inference for massive JSON datasets. VLDB J. **28**(4), 497–521 (2019). https://doi.org/10.1007/s00778-018-0532-7
3. Bex, G.J., Neven, F., Schwentick, T., Vansummeren, S.: Inference of concise regular expressions and DTDs. ACM Trans. Database Syst. **35**(2), 1–47 (2010). https://doi.org/10.1145/1735886.1735890
4. Bouhamoum, R., Kellou-Menouer, K., Lopes, S., Kedad, Z.: Scaling up schema discovery for RDF datasets. In: ICDEW 2018, pp. 84–89. IEEE (2018). https://doi.org/10.1109/ICDEW.2018.00021
5. BSON: Binary JSON (2012). http://bsonspec.org/spec.html
6. Cánovas Izquierdo, J.L., Cabot, J.: Discovering implicit schemas in JSON data. In: Daniel, F., Dolog, P., Li, Q. (eds.) ICWE 2013. LNCS, vol. 7977, pp. 68–83. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39200-9_8
7. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008). https://doi.org/10.1145/1327452.1327492
8. DiScala, M., Abadi, D.J.: Automatic generation of normalized relational schemas from nested key-value data. In: SIGMOD 2016, pp. 295–310. ACM (2016). https://doi.org/10.1145/2882903.2882924
9. Feliciano Morales, S.: Inferring NoSQL data schemas with model-driven engineering techniques. Ph.D. thesis, Universidad de Murcia (2017)
10. Frozza, A.A., dos Santos Mello, R., da Costa, F.d.S.: An approach for schema extraction of JSON and extended JSON document collections. In: IRI 2018, pp. 356–363 (2018). https://doi.org/10.1109/IRI.2018.00060
11. Gallinucci, E., Golfarelli, M., Rizzi, S., Abelló, A., Romero, O.: Interactive multidimensional modeling of linked data for exploratory OLAP. Inf. Syst. **77**, 86–104 (2018). https://doi.org/10.1016/j.is.2018.06.004
12. Holubová, I., Svoboda, M., Lu, J.: Unified management of multi-model data. In: Laender, A.H.F., Pernici, B., Lim, E.-P., de Oliveira, J.P.M. (eds.) ER 2019. LNCS, vol. 11788, pp. 439–447. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-33223-5_36
13. JavaScript Object Notation (JSON) (2013). http://www.json.org/
14. JSON Schema (2019). https://json-schema.org/
15. Klettke, M., Störl, U., Scherzinger, S.: Schema extraction and structural outlier detection for JSON-based NoSQL data stores. In: Datenbanksysteme für Business, Technologie und Web (BTW 2015), pp. 425–444 (2015)
16. Mlýnková, I., Nečaský, M.: Heuristic methods for inference of XML schemas: lessons learned and open issues. Informatica **24**(4), 577–602 (2013)
17. Pezoa, F., Reutter, J.L., Suarez, F., Ugarte, M., Vrgoč, D.: Foundations of JSON schema. In: Proceedings of the 25th International Conference on World Wide Web, pp. 263–273 (2016). https://doi.org/10.1145/2872427.2883029

18. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Pearson Higher Education (2004)
19. Sevilla Ruiz, D., Morales, S.F., García Molina, J.: Inferring versioned schemas from NoSQL databases and its applications. In: Johannesson, P., Lee, M.L., Liddle, S.W., Opdahl, A.L., López, Ó.P. (eds.) ER 2015. LNCS, vol. 9381, pp. 467–480. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25264-3_35
20. Wang, L.: Schema management for document stores. Proc. VLDB Endow. **8**(9), 922–933 (2015). https://doi.org/10.14778/2777598.2777601
21. Extensible Markup Language (XML) 1.0 (Fifth Edition) (2013). https://www.w3.org/TR/REC-xml/