Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores

Meike Klettke
Universität Rostock
meike.klettke@uni-rostock.de

Uta Störl
Hochschule Darmstadt
uta.stoerl@h-da.de

Stefanie Scherzinger
OTH Regensburg
stefanie.scherzinger@oth-regensburg.de

Abstract: Although most NoSQL Data Stores are schema-less, information on the structural properties of the persisted data is nevertheless essential during application development. Otherwise, accessing the data becomes simply impractical.
In this paper, we introduce an algorithm for schema extraction that is operating outside of the NoSQL data store. Our method is specifically targeted at semi-structured data persisted in NoSQL stores, e.g., in JSON format. Rather than designing the schema up front, extracting a schema in hindsight can be seen as a reverse-engineering step. Based on the extracted schema information, we propose set of similarity measures that capture the degree of heterogeneity of JSON data and which reveal structural outliers in the data. We evaluate our implementation on two real-life datasets: a database from the Wendelstein 7-X project and Web Performance Data.

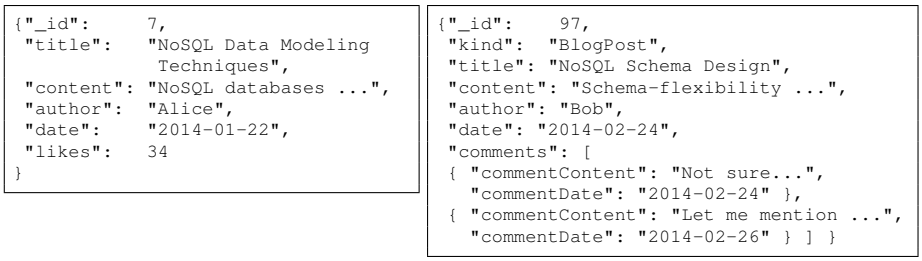
1 Introduction

Most NoSQL data stores do not enforce any interesting structural constraints on the data stored. Instead, the persisted data often has only *implicit* structural information, e.g., in NoSQL document stores that manage collections of JSON documents. Yet when accessing this data programmatically, it is vital to have some reliable notion of its structure, or schema. This holds for traditional application development, as well as when analyzing large scientific data sets: Without knowing the general structure of the data, and in particular its structural outliers, it becomes virtually impossible to do any pragmatic application development or meaningful data analysis. Yet when working with NoSQL data stores, the schema is frequently unknown, either due to rapid schema evolution in agile development, or, as we will show, when a scientific database contains data from different experiments.

NoSQL data stores excel at handling big volumes of data, and schema-less stores in particular are frequently chosen to manage data with a high degree of structural variety.

For instance, let us assume a startup releasing a blogging software. The JSON documents from Figure 1 have been persisted by two different versions of the application. In between the release producing the document in Figure 1(a) and the document in (b), the property *likes* has been dropped, and the functionality to comment on blogs has been added.

With the software-as-a-service model, it is common that applications are re-released into the cloud on a weekly, if not daily basis. In such settings, NoSQL data stores are particu-



(a) First-generation blogpost

(b) Second-generation blogpost

Figure 1: The structure of persisted JSON documents changes due to software- and schema evolution. Schema-free NoSQL data stores allow for heterogeneous documents to co-exist

larly popular back ends for data storage, since their schema-flexibility allows for heterogeneous data to co-exist. After all, most NoSQL data stores do not maintain a data dictionary and thus do not actively manage the schema of the data. This makes it unnecessary to migrate the persisted data with each new release of the software. At the same time, the complexity of handling heterogeneously structured data is building up [KSS14, SKS13].

When developers access heterogeneous data programmatically, it is vital to have some reliable notion of its schema. Let us emphasize on this point for different use cases:

- To build an interactive web application that reads and writes persisted data, e.g., a Java application using object mappers, developers need object mapper class definitions that robustly match blogpost documents from both generations. [SHKS15]
- Many NoSQL data stores provide a declarative query language. To formulate queries, developers need to know which attributes are present (or absent) in persisted objects.
- For OLAP-style data analysis, e.g., in the form of MapReduce jobs, developers also need to know which structure to expect when parsing JSON documents.

Thus, any pragmatic approach to these tasks requires some form of schema description. Figure 2 shows an instance of such a description in *JSON schema* [JSO14]. This specification matches both generations of blogposts in Figure 1, provides type information as well as information on the nesting of properties, and denotes which properties are required.

Out of this motivation, we present a new *schema extraction* approach that is custom-tailored to working with JSON documents. Our algorithm builds upon ideas proposed for extracting DTDs from collections of XML documents [MLN00], but takes into account the particularities of JSON data, such as the unordered nature of JSON documents (unlike XML, where the order of elements plays an essential role) and measures describing the structural heterogeneity of a set of JSON documents. Our algorithm outputs a JSON schema declaration. This declaration can then be the basis for providing convenient programmatic access to the persisted data, e.g.,

- by generating *type provider libraries* that allow convenient access to structured data

```
{  "type": "object",
   "properties": {
      "_id": { "type": "integer" },
      "title": { "type": "string" },
      "content": { "type": "string" },
      ...
      "comments": { "type": "array",
         "items": { "type": "object",
            "properties": {
               "commentContent": { "type": "string" },
               "commentDate": { "type": "string" } }
            "required": ["commentContent", "commentDate"] } } }
   "required": ["title", "content", "author", "date"] }
```

Figure 2: A JSON schema declaration for blogpost data

from within integrated development environments [SBT+12].

- by generating a hierarchy of object mapper class declarations, e.g., JPA [Jav09] or JDO [Jav03] Java class declarations, to be used in application development, or
- to generate validating JSON parsers directly from the JSON schema, in the tradition of parser generators and validating XML parsers.

The extracted JSON schema is an *explicit schema*. Our algorithm uses an internal graph structure that summarizes all structure variants encountered in the data. This allows us to also detect structural *outliers* (patterns that occur only in few data sets and might even be due to a errors during recording of the data). Further, we can determine the *degree of coverage* of the documents, a similarity measure capturing the structural homogeneity of a document collection.

In earlier work, we have proposed a database-external schema management component, a layer used on top of a NoSQL data store [KSS14]. This layer is to preserve the flexibility of the NoSQL data store, while at the same time making it possible to detect and correct structural inconsistencies in the data. The schema extraction algorithm, as presented in this paper, is a vital building block towards this vision.

Structure. This article is organized as follows: The next section gives an overview on different classes of NoSQL data stores. In Section 3, we formalize NoSQL documents and JSON schema, a schema language that is currently under discussion in the community. We present our schema extraction algorithm based on the **S**tructure Identification **G**raph (SG) in Section 4. We detect structural outliers in the data (Section 4.5), and introduce similarity measures in Section 4.6. Since SGs can become too large to fit in memory, we propose an alternative approach that uses a **R**educed Structure Identification **G**raph (RG) in section 5. We conduct our experiments on real-world data. The main results are provided in Section 6. The article concludes with an outlook on how the schema extraction can be implemented as part of a holistic schema management component.

2 NoSQL Data Stores

NoSQL data stores vary hugely in terms of their data and query model, scalability, architecture, and persistence design. The most common and for our context best suited categorization is by data model. Therefore we distinguish *key-value stores*, *document stores*, and *extensible record stores* [Cat10, Tiw11]. Often, extensible record stores are also called *wide column stores* or *column family stores*.

Key-value stores persist pairs of a unique key and an opaque value. There is no concept of schema beyond distinguishing keys and values. Therefore, applications modifying the data have to maintain the structural constraints and therefore also the schema.

Document stores also persist key-value pairs, yet the values are structured as "documents". This term connotes loosely structured sets of name-value pairs, typically in JSON (JavaScript Object Notation) [Ecm13] format or the binary representation BSON, a more type-rich format. Name-value pairs represent the properties of data objects. Names are unique within a document, and name-value pairs are also referred to as key-value pairs.

Documents are hierarchical, so values may not only be scalar values or lists, but even nested documents. Documents within the same document store may differ in their structure, since there is no fixed schema. If a document store is schema-less, documents may effortlessly evolve in structure over time: Properties can be added or removed from a particular document without affecting the remaining documents.

Extensible record stores actually provide a loosely defined schema. Data is stored as records. A schema defines families of properties, and new properties can be added within a property family on a per-record basis. (Properties and property families are often also referred to as *columns* and *column families*.)

Typically, the schema cannot be defined up front and extensible record stores allow the ad-hoc creation of new properties. The Cassandra system [Tiw11, Apa13] is an exception among extensible record stores, since it is much more restrictive regarding schema. Properties are actually defined up front, even with a "create table" statement, and the schema is altered globally with an "alter table" statement.

NoSQL Data Stores in Scope of this Paper. Data in document stores is often represented in JSON format. This format contains implicit structural information and is an interesting starting-point for schema extraction. Therefore, the schema extraction methods we present in this paper primarily focus on JSON-based document stores. Schema extraction can also be applied to extensible record stores due to their concept of explicitly naming properties and property families. The approach presented in this paper can easily be transferred to extensible record stores. Since key-value stores are not aware of schema apart from distinguishing keys and values, we believe they are not in the main focus for schema extraction. However, data in key-value stores may also be stored in JSON or a similar format. Then, schema extraction may also be applied to key-value stores.

3 JSON Documents and JSON Schema

Several NoSQL databases store JSON documents. In JSON the entities are called *objects*. Objects are unordered enumeration of *properties*, consisting of *name/value* pairs [Ecm13]. The available basic data types are *number, string*, and *boolean*. JSON properties can be multi-valued, these structures are then called *arrays*. Arrays are ordered lists of values, written in square brackets and separated by commas.

Schema for NoSQL Data Stores. For declaring a schema and validating data against a schema, we need an explicit schema description. Since we do not want to propose our own language, resort to an already available schema description language: JSON Schema [JSO14] defines structural constraints on sets of JSON objects.

In this article, we do not introduce the complete schema language. Instead, we provide some intuition using a small example in Figure 2. An *object* `blogpost` is represented by the *properties* `title, content, author, date, likes,` and `comments`. These properties are listed, the type of each property is defined. All *required* properties are enumerated, all other properties that are defined in the schema are *optional*. In the schema in Figure 2, the properties `likes` and `comments` are optional. The property `comments` is defined as an array, consisting of objects. Within the objects the properties `commentContent` and `commentDate` occur. Both properties are enumerated in the list of required properties that is given within the definition of the `comments`. The schema in Figure 2 uses the JSON data types `string`, `array`, and `object`. It even contains a hierarchical nesting of objects. In the following, we derive a JSON schema from a collection of JSON documents.

4 Schema Extraction with a Structure Identification Graph (SG)

In this chapter, we introduce our method for schema extraction from a collection of JSON documents. Our approach is related to existing work on DTD extraction from XML documents [MLN00]. Figure 3 states the sub-tasks of our extraction approach.

A graph data structure summarizes the structural information of all persisted documents. The nodes represent JSON properties, nested objects, or arrays, and the edges capture the hierarchical structure of the JSON documents. In particular, nodes and edges are labeled with lineage information, i.e. lists that specify in which JSON documents the structural property is present. The graph structure will be defined in Section 4.2 in detail and the construction of the graph will be described in Section 4.4.

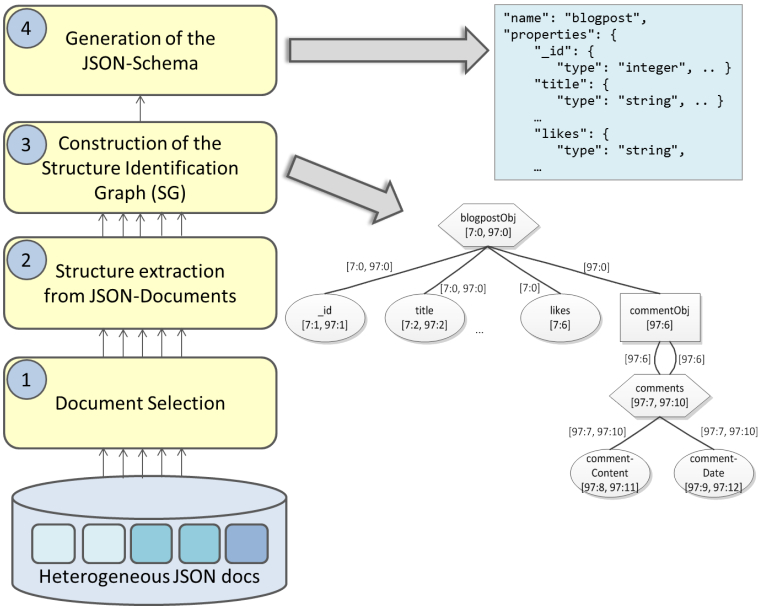


Figure 3: The sub-tasks of schema extraction

4.1 Document Selection

Our schema extraction algorithm derives an explicit schema from a collection of JSON documents. In a preselection step, we choose to run the schema extraction on either the *complete collection* or *selected subsets of JSON documents*.

There are several use cases motivating this document preselection step:

- It may be of interest to divide the collection of JSON documents into groups (distinguished by date, timestamp, or version number), and then extract the *schema for each group or version*. This can reveal the evolution of the schema over time.
- If a split attribute exists that can be applied for *distinguishing the JSON documents of a collection*, then we can build schema for a subset of documents. We can, for instance, distinguish documents from different servers, from different geographical places, or documents that describe different kinds of devices.

In summary, we preselect a NoSQL subset if different kinds of JSON documents are organized in the same collection. In all cases, the preprocessing can be done if a *JSON property or property set* for selecting a subset exists.

4.2 Definition of the Structure Identification Graph (SG)

We now define our internal graph data structure for schema extraction, with node and edge labels that capture the complete schema structure. Our schema extraction algorithm in Section 4.4 is based on this graph. The schema information from *all JSON documents* of a Collection or subset of a Collection is stored in a so-called **S**tructure Identification **G**raph.

Definition 3: A **S**tructure Identification **G**raph $SG = (V, E)$ is a directed graph where

- V is a finite set of vertices or *nodes*. Each node $v_i \in V$ is represented by a tuple ($nodeLabel_i$, $nodeIDList_i$) where
  - $nodeLabel_i$ is a name of the node, composed by
    - $path_i$, the path information starting from the document root and
    - $name_i$, the identifier of v_i which contains the property name
  - $nodeIDList_i$ is a *list* of $nodeID$s, that specifies in which JSON documents a node v_i occurs. A $nodeID$ is constructed by $docID_j : i$ where
    - $docID_j$ is the ID of the JSON document J_j within a Collection C
    - i is a unique node number within the JSON document
- $E \subseteq V \times V$ is a set of directed *edges* or arcs of SG with
  - $e \in E$ is represented by a tuple $(v_k, v_l, IDList_l)$, with v_k, $v_l \in V$, and $v_k \neq v_l$. $IDList_l$ is a list of $nodeID$s. It specifies for a node v_l under which parent v_k it occurs.

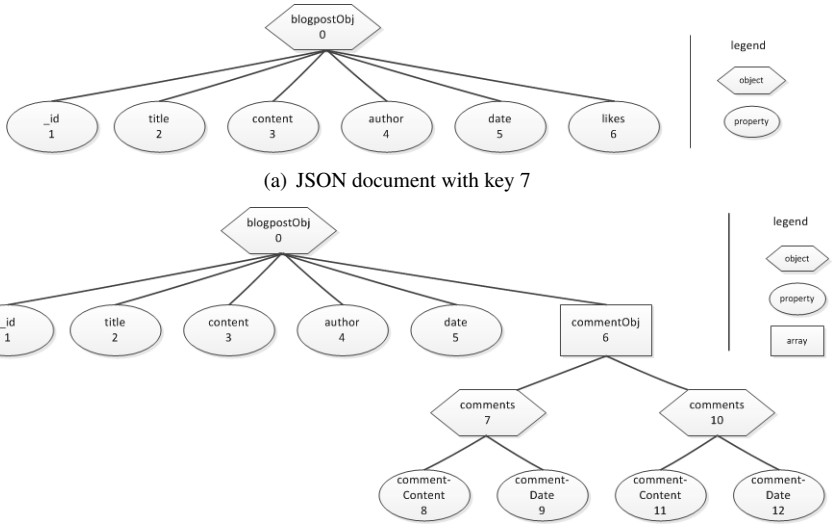
4.3 Example for JSON Schema Extraction

In the following, we provide an example. In a `blogpost` application, we assume the two entities (JSON documents) from Figure 1. The structure representation of the `blogpost` documents is shown in Figure 4(a). We assign unique node numbers in *preorder*. This simple numbering scheme is sufficient here, because the numbers only serve to identify the components during schema extraction. We assume that the JSON documents are not changed during schema extraction.

We want to derive the *structure identification graph* SG that represents the structure of all entities. For the JSON documents in Figure 4, we construct the SG in Figure 5. Same structures in the original data are summarized in SG. The node and edge labels reference the nodes from the input data. We next provide detailed algorithm for constructing an SG.

4.4 Schema Extraction Algorithm

We next describe the algorithm for constructing the SG when processing JSON documents. As we have pointed out in Section 2, our approach may be extended to work with



(a) JSON document with key 7

(b) JSON document with key 97

Figure 4: Structure representation of the entities `blogpost` from Figure 1

extensible record stores as well.

Construction of the Structure Identification Graph SG. First, we build the SG from the input data of the NoSQL database. The nodes in the JSON documents are traversed and numbered in preorder. For each node of the input data, an adding or extending of a node in SG `addNode` and an adding or extending of an edge from the node to its parent node `addEdge` is performed. The node numbers specify in which order the SG is constructed.

input data: JSON document collection C
initialize SG: $V = \emptyset$; $E = \emptyset$;
foreach $J_x \in C$ **do**:
  $i = 0$; // initialize counter
  // storing the root node of the JSON document
  SG.addNode(J_x, $\emptyset$, $rootname$, i);
  i=i+1;
  foreach $/path_j/name_j \in J_x$ **do**:
    SG.addNode(J_x, $path_i$, $name_i$, i);
    p= J_x.getID($path_i$);
    SG.addEdge(J_x, $path_i$, $name_i$, p)
    i=i+1;

When a node is added, we distinguish two cases: if the node does not yet exist in the SG, we *add the node*. The node is stored with a node name and with a list that contains only one reference: the `docID` of the current node and the unique node number `i`. If the node already exists in the SG (since the same node occurred in some other input document) then the current node information is appended to the list that is stored with the node.

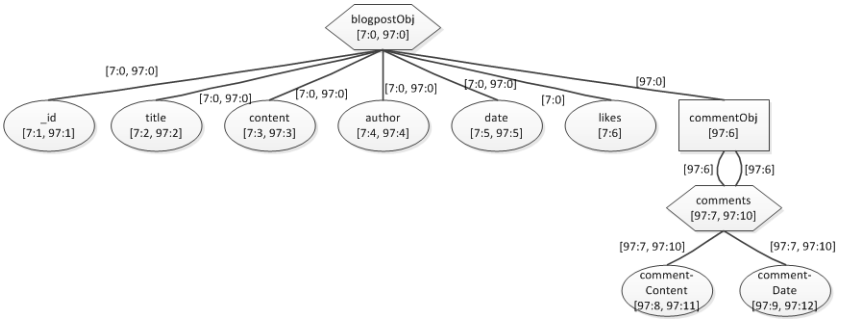


Figure 5: Structure identification graph for the sample entities

def SG.addNode(J_x, $path_i$, $name_i$, i):
  if $(/path_i/name_i) \in SG$:
    $V : (/path_i/name_i, list) \longrightarrow (/path_i/name_i, list$ + J_x.getDocID(): $i)$
  else
    $V := V \cup (/path_i/name_i, [J_x$.getDocID() : $i])$

Adding an edge is similar. If the edge does not yet exists in the SG, it is added, otherwise the edge list is updated by appending the `docID` and the unique node number `p` of the parent node.

def SG.addEdge(J_x, $path_i$, $name_i$, p):
  if $(/path_i/name_i) \in SG$:
    $E : (/path_i, /path_i/name_i, list) \longrightarrow (/path_i, /path_i/name_i, list$ + J_x.getDocID(): $p)$
  else
    $E := E \cup (/path_i/name_i, J_x$.getDocID(): $p])$

The traversal of the JSON documents in *preorder* ensures that the parent nodes are processed before their children.

Schema extraction. In the next step, we derive the JSON schema from the SG.

input data: SG - Structure Identification Graph
foreach $/path_i/node_i \in SG$ **do**:
  generate JSON description for $node_i$;
  if (SG.getNodeList($path_i$).length() == SG.getEdgeList($path_i$, $name_i$).length()):
    $/path_i/node_i$ is required
  else:
    $/path_i/node_i$ is optional

We give an example illustrating required and optional nodes. In the edge lists, only those IDs occur that are also available in the parent node lists. Furthermore, each ID in the

node and edge lists is *unique* (because it is a combination of the documentID and a unique nodeID). So each ID can occur in the node and edge lists at least once. Combining these two facts, we can conclude: if the lists coincide (then the list lengths are identical), the property is required. In Figure 6, the property `title` is *required* because the edge list and the node list of the parent node coincide. If the edge list contain fewer IDs than the parent node list then the property is optional. An example for that case is the property `likes` in Figure 6. It is *optional* because the edge list is a subset of the parent node list and the list length of the edge list is smaller than the list length of the parent node list.

Since the calculation of the list length is more efficient than the comparison of the complete edge and parent node lists, it is used in the algorithm given above.

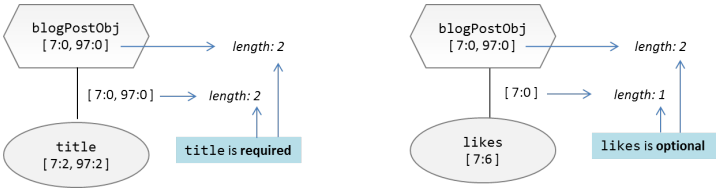


Figure 6: Required and optional properties in the structure identification graph SG

Finding the type of schema components. The above algorithm introduces the general idea of schema extraction from JSON documents. Its implementation is slightly more involved, since it additionally derives the *data type* (string, number, boolean, array, or object) of each schema component from the input JSON documents. This type information is also stored in the nodes of the schema identification graph SG.
If in the input instances the same properties with different data types exist, a union data type, for instance `"oneOf":[{"type":"string"},{"type":"integer"}]` is stored in the JSON schema.

4.5 Detection of Structural Outliers

The previous section described the JSON schema generation from the structure identification graph. The graph with its nodes, edges, node labels, and edge labels delivers even more information than merely a valid schema. We can use it to derive *statistics*, to find *outliers* in the data, and to calculate *measures* that capture how regular the documents within a JSON collection are. Figure 7 visualizes these subtasks within the overall workflow.

Adding statistics to a JSON schema. Adding statistics during JSON schema extraction is a straightforward extension. For each component of the schema, we can record the information how often this component occurred in relationship to its parent node. This information can also be derived from the SG. In our example from Figure 5, the property `likes` occurred in 50 % of the JSON documents. The information can be helpful to

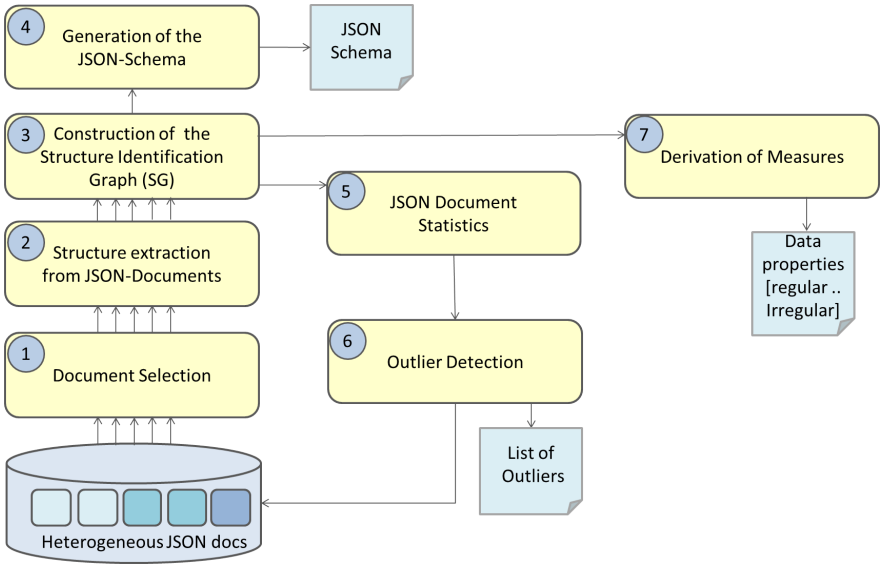


Figure 7: Schema extraction, detection of outliers, and calculation of measures

understand the input data.

Errors and Outliers in NoSQL Databases. Before we start with the outlier detection, we want to motivate why errors or outliers frequently occur in NoSQL databases. The main reason is that most NoSQL database systems do not check any structural constraints, so all well-formed JSON documents can be stored. Often, the data is collected over long periods of time. Different applications and different users have added and changed data. The lack of schema management tools makes it difficult to detect data containing errors or diverging property names.

For this reason, we are looking for structural outliers (that could be errors) with our method. Within our algorithm we cannot distinguish between

- structures that only occur rarely (schema divergences) and
- actual errors in the NoSQL data.

Deciding whether a finding is correct or an error requires user interaction. We automatically suggest *candidates*, but the *final error classification* remains a task for the user.

Detecting outliers in NoSQL data. With the structure identification graph SG, we can find all structural outliers, i.e., all variants that are under a given threshold ϵ. It delivers two kinds of outliers:

1. *Additional properties.* We can derive which properties exist only in some JSON documents (in less than ϵ (in %) of the documents). The list of these properties is

determined by calculating for each $v \in V$:

$edgeList$= SG.getEdgeList($/path_i$, $/path_i/name_i$)
$parentList$ = SG.getNodeList($/path_i$)

If $\frac{|edgeList|}{|parentList|} * 100 < \epsilon$ then a node only occurs rarely. JSON documents that contain this node are outliers. $outliers$ = $edgeList$ delivers references to all JSON documents that contain the node. Figure 8(a) shows an example.

2. *Missing properties.* Another kind of structural outliers that we can find with the SG are missing properties. If a property occurs in nearly all data sets and is only missing in some JSON documents ($\frac{|edgeList|}{|parentList|} * 100 > 100 - \epsilon$) then we obtain the list of outliers as $outliers = parentList - edgeList$. Figure 8(b) shows an example.

In the running example consisting only of 2 documents, we cannot detect outliers, because the example does not contain enough documents. We therefore extend the sample structure identification graph SG and represent both cases in Figure 8.

(a) Additional components (b) Missing components

Figure 8: Finding outliers in the structure identification graph SG

With a semiautomatic approach, the user can decide whether the structure is correct and whether it shall be considered in the schema representation. Otherwise, the user decides that a property shall not considered in the target schema and classifies the corresponding JSON documents as errors. We do not need to touch the stored NoSQL database again, the information which data sets are incorrect can be derived from the SG.

In summary, our approach does not support automatic cleaning and correction of the data, to avoid information loss.

4.6 Measures for Variability

In this section, we introduce and calculate measures for the *degree of coverage* of the JSON documents, that capture the *structural homogeneity* of a document collection.

Let's start with some definitions. Each property in a JSON document J_i can be addressed by a complete path expression, starting from the root. A property $p_k = /path_k/name_k \in J_i$ if the property with the specified `path` and `name` is available in J_i.

The *document size* $|J_i|$ is defined as the total number of properties in a JSON document. The common properties of two JSON documents J_i and J_j are defined as follows:

$$J_i \cap J_j = \{p_k \mid p_k \in J_i, p_k \in J_j\} \tag{1}$$

Degree of coverage. We further define the *degree of coverage* between two JSON documents J_i and J_j.

$$cov(J_i, J_j) = \frac{1}{2} \cdot \frac{|J_i \cap J_j|}{|J_i|} + \frac{1}{2} \cdot \frac{|J_i \cap J_j|}{|J_j|} \tag{2}$$

The coverage specifies the overlap of two documents. It delivers a value between 0 and 1 that captures the structural similarity of two documents. We can generalize the coverage definition for an arbitrary subset $J_i..J_k$ of k JSON documents:

$$cov(J_1..J_k) = \frac{1}{k} \cdot \sum_{i=1}^{k} \frac{|\bigcap_{j=1}^{k} J_j|}{|J_i|} \tag{3}$$

This measure considers the coverage of the documents $J_1..J_k$ of a collection C. It can efficiently be calculated. If the cov value is 0 the JSON documents are completely heterogeneous, if the value is 1 then all documents have exactly the same structure. A disadvantage is that this measure is strongly influenced by a few JSON documents, even one JSON document with divergent structure changes the result. Other measures, for instance the Jaccard coefficient often used for determining the similiarity has the same disadvantage.

$$jac(J_1..J_k) = \frac{\bigcap_{j=1}^{k} J_j}{\bigcup_{j=1}^{k} J_j} \tag{4}$$

One JSON document with divergent structure can cause that the Jaccard measure delivers the value 0.

As an alternative, it is possible to introduce a more sophisticated measure that considers all subsets S of a JSON collection C= $J_1..J_n$ with at least 2 documents. With $|S|$ the number of documents in a subset S and $|M|$ the size of M, we can now calculate:

$$M = \{S|S \subseteq (J_1..J_n), |S| >= 2\} \tag{5}$$

$$cov_{sub}(J_1...J_n) = \frac{1}{|M|} \cdot \sum_{S \in M} cov(S) \tag{6}$$

This measure delivers better results especially for collections that are homogeneous and contain only one or a few JSON documents that feature another structure. An disadvantage is that this measure cannot be efficiently calculated for large collections.

That's why we proceed with the simpler coverage measure (Formula 3), and define on it the *supplement* of each JSON document J_i within a JSON collection $C = J_1..J_n$.

$$sup(J_i, C) = \frac{|J_i| - |\bigcap_{j=1}^{n} J_j|}{|J_i|} \tag{7}$$

Required and optional schema components. We consider properties that occur in all JSON documents $J_1..J_n$ of a collection C required properties. Properties are optional if they occur in at least one, but not in all JSON documents of a collection C.

$$req(C) = \{p \mid \forall J_i \in C : p \in J_i\} \tag{8}$$

$$opt(C) = \{p \mid \exists J_i, J_j \in C : p \in J_i \wedge p \notin J_j\} \tag{9}$$

In JSON documents with a homogeneous structure, most properties are required. Heterogeneous JSON documents mainly have optional properties. These characteristics are also reflected in the schemas of the JSON document collections.

We want to determine the degree of coverage with Formula (3) for our running example from Figure 1.

$$cov(J_i, J_j) = \frac{1}{2} \cdot \frac{|J_i \cap J_j|}{|J_i|} + \frac{1}{2} \cdot \frac{|J_i \cap J_j|}{|J_j|} = \frac{1}{2} \cdot \frac{5}{6} + \frac{1}{2} \cdot \frac{5}{12} = 0.625 \tag{10}$$

The degree of coverage can be calculated in parallel to the schema extraction. $\bigcap_{j=1}^{n} J_j$ can be derived by traversing SG. The counting of the properties of each JSON document $|J_i|$ and storage of this number is an easy-to-realize extension of the algorithm.

5 Reduced Structure Identification Graph (RG)

We also have implemented a second variant of the schema extraction approach that uses a **R**educed Structure Identification **G**raph (RG). We have seen in Section 4 that the schema extraction algorithm does only base on the node and edge lists lengths (not on the list elements). In this variant, the graph does not contain references to the original JSON

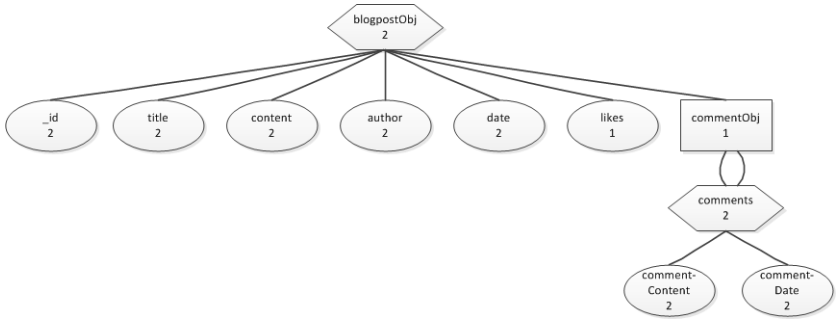


Figure 9: Reduced structure identification graph (RG) for the blogpost application

documents, we only count the *number of occurrences* and store it for each *node* of the graph. Figure 9 shows RG for the running example.

We can introduce this reduced structure (in contrast to the original approach [MLN00]) because JSON documents are more simple in structure than XML documents. Let's take a small XML example for illustrating it (figure 10):

```
<book>
    <author> .. </author>
    <author> .. </author>
    <title> .. </title>
    ...
</book>
```

```
<book>
    <editor> .. </editor>
    <title> .. </title>
    ...
</book>
```

Figure 10: Sample XML documents

If we would count the occurring elements in both XML documents then we would get the result that `book`, `title` and `author` occur twice. The elements `book` and `title` are required in our small example, but `author` is optional, because the element is not available in the second XML-document. So, in XML the number of occurrences is not sufficient to find out the required and optional elements. In contrast, in JSON documents, each property (with its complete path) is unique in the JSON documents. The number of occurrences corresponds with the number of JSON documents containing the specified property.

That's why, we can use the simpler graph structure RG for the JSON schema extraction algorithm. With this data structure, we can derive the schema, we also can determine *statistics* and we can derive the *coverage value*. It is not possible to determine *outliers* with the data structure. The table in Figure 11 compares both alternatives of the schema extraction algorithm.

(*) Also in this variant, it is possible to detect outliers. From the JSON schema and additional statistics, we can derive which properties p_i occur only rarely ($< \epsilon$). In the next step, we query the original JSON data set and find out the documents that contain the property p_i. In MongoDB, for example, we can execute the following query:

	Structure Identification Graph (SG)	Reduced Structure Identification Graph (RG)
JSON Schema Extraction	+	+
Document Statistics	+	+
Similarity Measures	+	+
Outlier Detection	+	- (*)

Figure 11: Comparison of the structure extraction and outlier finding with a structure identification graph SG and a reduced structure identification graph RG

```
db.collection.find({pi: {$exists: true}})
```

If we find out that a property p_j occurs in the majority of documents ($> (100\% - \epsilon)$), we select the matching JSON documents from the NoSQL database with the following query:

```
db.collection.find({pj: {$exists: false}})
```

We only have to consider that at the moment not all NoSQL databases support negations in queries. For instance, in the current versions of Cassandra we cannot express queries containing a `not exists`.

6 Experiments

We have implemented our schema extraction algorithm and conducted experiments on the configuration database of the Wendelstein 7-X project [SLB+12]. This database has been in use for several years to record configurations of plasma experiments. It currently holds more than 120 different collections to record information on devices, as well as parameters that control the settings of experiments. Most collections within this database are structured and regular, yet there are also structural outliers. Until today, the database has been used without any explicit schema description.

Our implementation has extracted one JSON schema for each collection. We implemented the two alternatives for schema extraction as introduced in Sections 4 and 5 in Python. Our experiments ran on an Intel Core i7-461000U CPU @ 2.70GHz machine with 8.00 GB RAM. In most applications scenarios, schema extraction is run as a batch process, so we do not require runtimes that allow for real time user interactions. However, runtime is still a critical criterion for the feasibility of schema extraction. Thus we study the runtime of our algorithms to acquire ballpark numbers, and to determine whether the method is even applicable in practice. Moreover, we decided to use an off-the-shelf commodity machine, rather than a powerful data server, since we want to observe if the algorithm can be executed without special hardware.

The following performance measurements show the execution times of schema extraction using a structure identification graph introduced in Section 4. We represent the performance measures for the 30 most data-intensive Collections of the Wendelstein-7-X dataset.

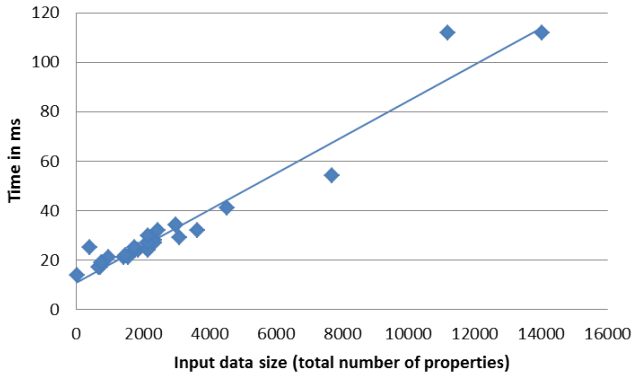


Figure 12: Schema extraction for the different collections of the Wendelstein data

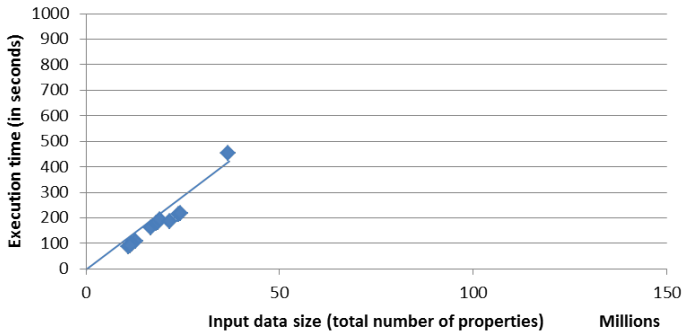
On the x-axis, the size of the input data is shown. We determine the total number of structural information that means the sum of the number of properties, arrays, and objects in the JSON collection. As an example: One collection contains 206 JSON documents, each JSON document has 22 properties, so the input data size of this collection is 4532. Our experiment shows linear growth with the size of the input data (see Figure 12).

The schema extraction approach based on the Structure Identification Graph SG (Section 4) works very well for small data sets. An obvious bottleneck is that all nodes and edges carry references to the original data. For collections containing many data sets these lists become very large. For that, we applied the alternative approach based on the RG that was introduced in Section 5.

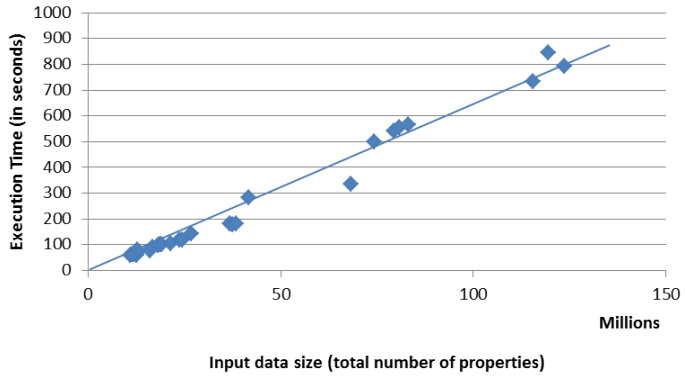
To compare both approaches, we tested them with yet another dataset. We chose a larger MongoDB database storing about 800 collections in 150 GigaByte of data. The database contains statistical data of Web Performance Measures. It describes different technical system parameters, logdata, and different performance data of several Web Information Systems of the Pagebeat project [FBH+14]. Figure 13 shows some results of the schema extraction for this data. The SG construction for the larger data set 13(a) is only possible for data sets up to 40 million input nodes (about 500 Mega Byte when stored in MongoDB). The method with RG could be realized for all collections of the data set up. Again, the algorithm runtime is linear in the size of the input data.

7 Related Work

There is a large body of work on schema extraction from XML documents ([MLN00], [DCjWS06], [HNW06]). Our approach builds upon the work of Moh, Lim, and Ng, who have developed a DTD Miner for extracting a schema for a given set of XML documents [MLN00]. They capture the structure of each XML document with a tree, and then



(a) Schema Extraction with SG



(b) Schema Extraction with RG

Figure 13: Schema extraction on different MongoDB collections on Web usage data

derive a graph representation which summarizes all structural variants. Finally, a DTD is derived that identifies optional elements and repeating elements. In deriving a JSON schema, as done in this paper, the document order and repetition of elements do not play a role. This makes it possible to build the reduced graph, as discussed in Section 5.

The authors in [BNSV10] give a comprehensive survey over the extraction of DTDs and regular expressions from XML documents in particular. They point out the challenges imposed by the order of siblings in XML documents, and the fact that cross-references between nodes actually define graphs, rather than trees. With JSON documents, however, the order among sibling nodes does not play a role.

In [BCNS13], the authors propose a type system for JSON objects as well as a type inference algorithm. This serves as a basis for defining the semantics of JSON query languages.

The JSON type system in [CGSB12] comes with a MapReduce-based type inference al-

gorithm. In the context of NoSQL data stores, scalability is vital, since we may need to handle data sets in the *big data* category. At the time of writing this paper, there are no benchmarks effectively demonstrating the scalability of this approach yet.

In data exchange, schema extraction plays a vital rule. IBM's BlueMix portfolio [IBM14] offers a *schema discovery* component for importing JSON data from Cloudant into the data warehousing product Dynamite. Any received data that does not match this schema is sent to a bucket, where it can then be inspected manually.

In [KSS14], we introduce the idea of an external schema management component for NoSQL databases. In [SKS13] we further propose a language for declaring schema evolution operations for extensible record stores and for NoSQL document stores.

8 Summary and Future Work

In this article, we introduce two variants of schema extraction for NoSQL databases. The first one is similar to the DTD Miner [MLN00] that was developed for XML documents. The second algorithm relies on an optimization that we can make because JSON documents are simpler in structure than XML documents. With both approaches, we can generate a JSON schema from an available data set. We show that the internal data structure that we use for schema discovery can also be used for collecting statistics on the input data, as well as for detecting structural outliers. For instance, we can derive a similarity measure that captures how regularly the data within a collection is structured.

We tested our approaches on different JSON collections managed in MongoDB. With the Web Statistic data from the Pagebeat project [FBH+14], our outlier detection found several outlier documents that were not previously known to the owners of this data. Knowing about outliers is an important prerequisite for improving the data quality.

In our long term vision, we see our schema extraction algorithm as part of a powerful schema management component for NoSQL databases. Systems that do not actively manage the schema actually force software developers to maintain the schema constraints within in the application logic. To assist developers in their daily work, we are designing dedicated tools for handling and managing the schema for NoSQL data stores, while paying full respect to their strongpoint, their schema-flexibility.

References

[Apa13] Apache Cassandra, 2013. `http://cassandra.apache.org/`.

[BCNS13] Véronique Benzaken, Giuseppe Castagna, Kim Nguyen, and Jérôme Siméon. Static and Dynamic Semantics of NoSQL Languages. In *Proc. 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 101–114, New York, NY, USA, 2013. ACM.

[BLS99] Andreas Brandstädt, Van Bang Le, and Jeremy P. Spinrad. *Graph Classes - A Survey*. SIAM Monographs of Discrete Mathematics and Applications, 1999.

[BNSV10] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. Inference of concise regular expressions and DTDs. *ACM Trans. Database Syst.*, 35(2), 2010.

[Cat10] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, 2010.

[CGSB12] Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Universität Della Basilicata. Typing Massive JSON Datasets, 2012.

[DCjWS06] Theodore Dalamagas, Tao Cheng, Klaas jan Winkel, and Timos Sellis. A methodology for clustering XML documents by structure. *Information Systems*, 31:187–228, 2006.

[Ecm13] Ecma International. *The JSON Data Interchange Format*, 2013. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

[FBH+14] Andreas Finger, Ilvio Bruder, Andreas Heuer, Martin Klemkow, and Steffen Konerow. PageBeat - Zeitreihenanalyse und Datenbanken. In *GI-Workshop: Grundlagen von Datenbanken'*, 2014.

[HNW06] Jan Hegewald, Felix Naumann, and Melanie Weis. XStruct: Efficient Schema Extraction from Multiple and Large XML Documents. In *ICDE Workshops*, page 81, 2006.

[IBM14] IBM. IBM Blue Mix, 10 2014. `https://ace.ng.bluemix.net/`.

[Jav03] Java Data Objects Expert Group. *JSR 12: Java Data Objects*, 2003.

[Jav09] Java Persistence 2.0 Expert Group. *JSR 317: Java Persistence 2.0*, 2009.

[JSO14] JSON Schema Community. *JSON Schema*, 2014. `http://json-schema.org`.

[KSS14] Meike Klettke, Stefanie Scherzinger, and Uta Störl. Datenbanken ohne Schema? - Herausforderungen und Lösungs-Strategien in der agilen Anwendungsentwicklung mit schema-flexiblen NoSQL-Datenbanksystemen. *Datenbank-Spektrum*, 14(2):119–129, 2014.

[MLN00] Chuang-Hue Moh, Ee-Peng Lim, and Wee-Keon Ng. DTD-Miner, A Tool for Mining DTD from XML Documents. In *Proc. WECWIS*, 2000.

[SBT+12] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, et al. F#3.0 - Strongly-Typed Language Support for Internet-Scale Information Sources. Technical Report MSR-TR-2012-101, Microsoft Research, September 2012.

[SHKS15] Uta Störl, Thomas Hauff, Meike Klettke, and Stefanie Scherzinger. Schemaless NoSQL Data Stores Object-NoSQL Mappers to the Rescue? In *Proc. 16. BTW, March, 2015 Hamburg, Germany*, 2015.

[SKS13] Stefanie Scherzinger, Meike Klettke, and Uta Störl. Managing Schema Evolution in NoSQL Data Stores. *Proc. DBPL*, CoRR, abs/1308.0514, abs/1308.0514, 2013.

[SLB+12] Anett Spring, Marc Lewerentz, Torsten Bluhm, Peter Heimann, Christine Hennig, Georg Köhner, Hugo Kroiss, Johannes G. Krom, Heike Laqua, Josef Maier, Heike Riemann, Jörg Schacht, Andreas Werner, and Manfred Zilker. A W7-X experiment program editor - A usage driven development. In *Proc. 8th IAEA Technical Meeting on Control, Data Acquisition, and Remote Participation for Fusion Research*, pages 1954 – 1957, 2012.

[Tiw11] Shashank Tiwari. *Professional NoSQL*. John Wiley & Sons, 2011.