# ReCG: Bottom-Up JSON Schema Discovery Using a Repetitive Cluster-and-Generalize Framework

Joohyung Yun
POSTECH
Pohang, Republic of Korea
jhyun@dblab.postech.ac.kr

Byungchul Tak
Kyungpook National University
Daegu, Republic of Korea
bctak@knu.ac.kr

Wook-Shin Han
POSTECH
Pohang, Republic of Korea
wshan@dblab.postech.ac.kr

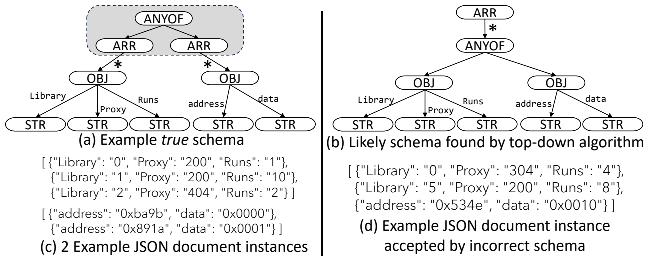
## ABSTRACT

The schemalessness, one of the major advantages of JSON representation format, comes with high penalties in querying and operations by denying various critical functions such as query optimizations, indexing, or data verification. There have been continuous efforts to develop an accurate JSON schema discovery algorithm from a bag of JSON documents. Unfortunately, existing schema discovery techniques, being a top-down algorithm, suffer from a fundamental lack of visibility into children nodes of JSON tree. With absence of the information about lower-level JSON elements, top-down algorithms need to employ assumptions and heuristics to decide the schema type of nodes. However, such static decisions are often violated in datasets which causes top-down algorithms to perform poorly. To overcome this, we propose an algorithm, called ReCG, that processes JSON documents in a bottom-up manner. It builds up schemas from leaf elements upward in the JSON document tree and, thus, can make more informed decisions of the schema node types. In addition, we adopt MDL (Minimum Description Length) principles systematically while building up the schemas to choose among candidate schemas the most concise yet accurate one with well-balanced generality. Evaluations show that our technique improves the recall and precision of found schemas by as high as 47%, resulting in 43% better F1 score while also performing 2.40× faster on average against the state-of-the-art.

## 1 INTRODUCTION

JSON (JavaScript Object Notation) is a widely used data representation format that has become de-facto standard for RESTful Web API's data exchanges [1, 26, 32, 51] and big data analytics [5, 10, 17, 30, 33, 45, 47]. Accordingly, efficient processing of massive volumes of JSON data and querying has become critical for services and operations. However, the efficiency of these operations is often hindered by the lack of schemas for the JSON documents at hand. Although the lack of strict schema enforcement of JSON format offers flexibility and rapid deployability, it instead exacerbates the data management costs by making it difficult to apply various operations [48]. Schemas are essential for many important use cases such as JSON validation [3, 37], JSON parsing acceleration [30, 33], migration of two data sources with different schemas [49], query formulation [16, 55], query optimizations [7, 9, 20, 35], fine-grained access control [48], JSON-to-relational data translation [13, 28, 53], and query answering [54]. Despite these practical benefits, JSON schemas are often nonexistent or unavailable to the users and, thus, have to be derived from a given set of JSON documents.

Due to the importance of JSON schemas, the JSON schema discovery (JSD) problem and topics related to the JSON schema formalization have been investigated by many researchers [3, 4, 6, 11, 17, 24, 37, 42, 48]. Despite many efforts, deriving correct JSON schemas from JSON documents remains a difficult problem due to several reasons. First, the ground truth schema set we want to derive may not exist in the first place for the given JSON documents. It is not uncommon that JSON documents are created without explicit schemas defined in advance. Users of JSON data prefer to create, load, and use the data quickly without the burden and delay from the schema construction task [34, 43]. The lack of a predefined schema also implies that the representation of objects in JSON can be inconsistent and has a high degree of variability [27]. Second, the derivation of a huge number of *correct* and *valid* schema sets is possible for a given JSON document set, but with varying degrees of generality. If the derived schemas are too general, the data validation becomes ineffective [42]. Moreover, the specific details of the schemas have to be eventually encoded into the query which increases the query processing overhead during the execution. On the other hand, if too specific, the size of the schema set increases and the schema handling costs rise significantly. A working solution to JSD problem should be able to find the schema with the right balance between representational simplicity and details within the vast search space of valid schemas.



```
[ {"Library": "0", "Proxy": "200", "Runs": "1"},
  {"Library": "1", "Proxy": "200", "Runs": "10"},
  {"Library": "2", "Proxy": "404", "Runs": "2"} ]
[ {"address": "0xba9b", "data": "0x0000"},
  {"address": "0x891a", "data": "0x0001"} ]
```

(c) 2 Example JSON document instances

```
[ {"Library": "0", "Proxy": "304", "Runs": "4"},
  {"Library": "5", "Proxy": "200", "Runs": "8"},
  {"address": "0x534e", "data": "0x0010"} ]
```

(d) Example JSON document instance accepted by incorrect schema

**Figure 1: Illustration of top-down approach's limitation. Top-down processing may lead to incorrect results (b) due to lack of visibility into descendant nodes.**

A JSON schema can be viewed intuitively as having a tree-like hierarchical structure as shown in Figure 1 (a). The intermediate nodes in this tree structure are either objects or arrays that recursively hold other objects, arrays, or primitive types. The goal of the JSON schema discovery (JSD) problem is to learn this template structure from a given bag of JSON documents. Two example JSON documents are shown in Figure 1 (c). The top-down style of JSON schema discovery, adopted by the state-of-the-art algorithms [4, 6, 42], starts processing from the root and it proceeds down to the subsequent child levels repeatedly. However, this top-down approach is constrained by inherent limitations. Whenever new node is encountered, it can be difficult to determine the correct

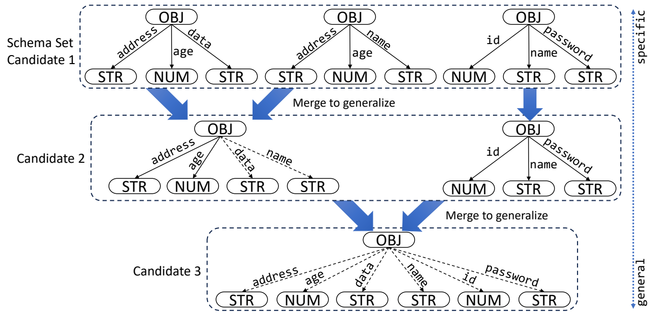


**Figure 2: Example schema sets with varying degrees of generality. More general schema sets are created by merging from the more specific set.**

node types (e.g., ARR, OBJ, ANYOF) since it has to be done without the knowledge of yet unseen lower-level nodes. This inevitably leads to the use of heuristics to make the best-effort decision of what the true type of the current node is. For example, at the shaded region of Figure 1 (a), the top-down algorithm is unable to differentiate two types of objects and, thus, comes up with the incorrect (and overly general) schema as shown in (b) which accepts a mixture of objects. In addition, existing algorithms make a decision of node types rigidly even when there are other possible candidates at each decision point. This leaves little chance to rectify incorrect decisions made at the upper level when the processing reaches the descendant nodes where decision errors become evident.

To overcome the limitations of top-down approaches, we have developed a methodology, called ReCG, that employs three techniques. *First, we design a novel bottom-up style JSON schema discovery algorithm* that departs from the existing approaches. We argue that the bottom-up style of processing is inherently better suited to the JSON schema discovery problem. At any intermediate node, the determination of a node type is better informed since the knowledge of descendant nodes is already acquired. Any two nodes of identical type can actually be disparate when their children and descendants are taken into account. Thus, with bottom-up style approach, we are able to reduce the uncertainty in making decisions and the reliance on the heuristics, the source of potential inaccuracy. *Second, we systematically explore candidate JSON schema sets comprehensively by incrementally generalizing them from specific schemas to the most general ones.* As Figure 2 illustrates, there can be schema sets with varying generality for the same JSON document sets at some point during the JSON schema discovery. We do not attempt to guess the correct one during processing as any choice has the potential to grow to be the true schema when schema discovery is completed. *Third, we employ the MDL (Minimum Description Length) metric to our JSON schema discovery problem as a metric that guides us to select the most promising candidate schema set during the navigation of the search space.* The MDL metric, used successfully in XTRACT [25] for computing DTD's information size, can be used here to compute the number of bits required to represent the schema and the data (JSON document) instances. We demonstrate in our evaluation its validity as a metric for assessing the goodness of JSON schema set.

$$
\begin{array}{lcll}
J & ::= & P \mid O \mid A & \\
P & ::= & \texttt{null} \mid \texttt{true} \mid \texttt{false} \mid n \mid s & (n \in \textbf{Number}, s \in \textbf{String}) \\
O & ::= & \{\ k_1 : J_1, \ldots, k_n : J_n\ \} & (n \geq 0, i \neq j \implies k_i \neq k_j) \\
A & ::= & [\ J_1, \ldots, J_n\ ] & (n \geq 0)
\end{array}
$$

**Figure 3: Grammar of JSON documents**

We have designed and implemented the prototype of our technique and evaluated it using 20 real-world JSON datasets. Our evaluation showed that ReCG delivered the 41.37∼42.60% improved F1 score compared to the state-of-the-art technique. We observed this was due to 43.54∼46.86% improvements in recall and 9.81∼17.20% improvements in precision. In terms of the algorithm running time, ReCG outperformed the state-of-the-art by 2.40×.

Overall, we make the following contribution in this work. First, we design a novel bottom-up algorithm, called ReCG for the JSON schema discovery problem that addresses major issues of existing algorithms. Accompanied by a technique for systematic search of schema sets by the specificity and the metric for assessing the goodness, our technique exhibits much-improved precision and recall against competing algorithms. Second, we provide supporting evidence that demonstrates the effectiveness of our technique using real-world data sets. Third, we make a prototype implementation of our technique as well as the dataset publicly available for the research community.

## 2 BACKGROUND

### 2.1 Definition

We use the *type* notation of Baazizi. et. al. [6] to a basis model of JSON documents and JSON schemas. We then extend the BNF-based notation of Baazizi. et. al. to a tree-based notation, for the sake of visualization.

*2.1.1 JSON document.* We adopt the definition of JSON document introduced in the work by Baazizi et. al. [6]. JSON documents are a set of strings that are derived from the grammar described in Figure 3. The derivation starts from a non-terminal $J$. It transitions to either $P$ (primitive JSON document), $O$ (object), or $A$ (array). The non-terminal $P$ further transitions to one of the following:

- *Number* type JSON document: C-like number, except the octal and hexadecimal formats. The set of these numbers are denoted as **Number**.
- *String* type JSON document: double-quoted sequence of zero or more Unicode characters. The set of such double-quoted sequences are denoted as **String**.
- *Boolean* type JSON document: `true` or `false`
- *Null* type JSON document: `null`

The non-terminal $O$ is derived to an object, which is an unordered set of (key, JSON document) pairs. An object begins with '{' and ends with '}', and the pairs are separated by ','. We denote the pairs within an object as *key-value pairs*. $A$ is derived to an array, which is an ordered sequence of JSON documents. We call the JSON documents within the sequence of array as *elements*. An array begins with '[' and ']', with elements separated by ','. The JSON document can be recursively defined. That is, JSON documents derived from $J$, can recursively occur within objects and arrays.

| | | | |
|---|---|---|---|
| $JS$ | ::= | { $SchCont$ } | |
| $SchCont$ | ::= | $strSch \mid numSch \mid boolSch \mid nullSch$ $\mid objSch \mid arrSch \mid anyOf$ | |
| $strSch$ | ::= | `"type": "string"` | |
| $numSch$ | ::= | `"type": "number"` | |
| $boolSch$ | ::= | `"type": "boolean"` | |
| $nullSch$ | ::= | `"type": "null"` | |
| $objSch$ | ::= | `"type": "object"` $(homO)^{?}$ $(hetO)^{?}$ | |
| $homO$ | ::= | $prop\ req^{?}$ | |
| $prop$ | ::= | `, "properties": {` $(key : JS\text{ , })^{*}$ `}` | |
| $req$ | ::= | `, "required": [` $(key\text{ , })^{*}$ `]` | |
| $key$ | ::= | $s$ | $(s \in \textbf{String})$ |
| $hetO$ | ::= | `, "additionalProperties":` $JS$ | |
| $arrSch$ | ::= | `"type": "array"` $(homA \mid hetA)$ | |
| $homA$ | ::= | `, "items":` $JS$ | |
| $hetA$ | ::= | `, "items": [` $(JS,)^{*}$ `]` | |
| $anyOf$ | ::= | `"anyOf": [` $(JS,)^{+}$ `]` | |

**Figure 4: BNF grammar of JSON schemas.**

*2.1.2 JSON Schema.* *JSON schema* is a set of strings in the Backus-Naur form grammar, introduced by Pezoa et. al. [37]. We adopt a subset of Pezoa et. al.'s production rules since real-life schemas use only a limited subset of grammars, as introduced specifically by Spoth et. al. [42]. The grammar of our interest is shown in Figure 4, and we denote a JSON schema as $\mathcal{S}$. In this grammar, non-terminals are expressed in *italic* font and the terminals are expressed in `consolas` font.

We use '$\models$' symbol to represent 'accepted-by' or equivalently 'derived-from' relationship between a JSON document instance and a JSON schema. As an example, $j \models \mathcal{S}$ means $j$ satisfies $\mathcal{S}$'s constraints, and, we can interpret $j$ as having been derived from $\mathcal{S}$. We will also interchangeably say that $j$ is validated against $\mathcal{S}$.

In the grammar above, the right-side of non-terminals such as *strSch*, *objSch*, and *arrSch* specifies the JSON document's type using the keyword `"type"`. We call the schemas that are derived by its corresponding non-terminal as *type* schema (e.g., string schema, object schema). For example, if $\mathcal{S}$ =`{"type": "string"}` and $j$ =`"P1151"`, then $\mathcal{S}$ is a string schema and $j \models \mathcal{S}$.

*2.1.3 Homogeneity and Heterogeneity of Object and Array Schemas.* For object schemas, derivations from *homO* and *hetO* impose constraints on the key-value pairs of an object. Derivation from *homO* imposes objects to be *homogeneous*; keys that can be present within an object's key-value pairs are determined as *key*s. Also, the corresponding value for each *key* must be validated against the schema derived by *JS* (*key* : *JS*). The *req* lists the keys that must be present in an object. Meanwhile, derivation from *hetO* imposes objects to be *heterogeneous*. The number of key-value pairs is unconstrained and the keys may be any random string. However, the values must be validated against the schema derived from *JS*. We name object schemas that have *homO* non-terminal derived as *homogeneous object schema*s, and *hetO* derived as *heterogeneous object schema*s. Object schemas with derivations from both *homO* and *hetO* are *composite object schema*s.

For array schemas, derivations from *homA* and *hetA* impose constraints on the elements of an array. Derivation from *homA* imposes arrays to be *homogeneous*; the number of elements within an array must be fixed, and each element at index $i$ must be validated against the $i^{th}$ schema paired by the key `"items"`. Whereas, the derivation from *hetA* imposes arrays to be *heterogeneous*. The number of elements may vary for an array, but its elements should all be validated against the schema derived from the non-terminal *JS* paired by the key `"items"`. We name array schemas that have *homA* non-terminal derived as *homogeneous array schema*s, and *hetA* derived as *heterogeneous array schemas*. Unlike object schemas, we do not define composite array schemas.

Lastly, derivation from $anyOf$ is named as $anyOf$ schema. It can have many schemas derived from *JS*s paired by the keyword `"anyOf"`. An anyOf schema validates against a JSON document $j$ if any of its derivations can validate against $j$.

A set of JSON schema is a set of strings that are derived by the *Formal Grammar for JSON Schema* $G_{JSchema}$ [37]. Of the set of JSON schemas, we deal with a limited subset of JSON schemas in this work. This is because the entire set of JSON schemas contains too many possible cases of JSON schemas, and real-life schemas can be expressed with more constrained grammar. We adopt the basic grammar used by Baazizi et. al. [4] and extend it. The specific syntax of the schema is described as follows:

$$\mathcal{U} ::= +(\mathcal{S}_1, \ldots, \mathcal{S}_n)(n \geq 0)$$
$$\mathcal{S} ::= \mathcal{P} | \mathcal{O} | \mathcal{A}$$
$$\mathcal{P} ::= \texttt{Null} \mid \texttt{Bool} \mid \texttt{Num} \mid \texttt{Str}$$
$$\mathcal{O} ::= \{k_1 : \mathcal{U}_1 q_1, \ldots, k_n : \mathcal{U}_n q_n\} | \{\mathcal{U}^*\}(n \geq 0, \in \{?, !\})$$
$$\mathcal{A} ::= [\mathcal{U}_1, \ldots, \mathcal{U}_n] | [\mathcal{U}^*](n \geq 0)$$

## 2.2 JSON Document Instance and Schema Representation

*2.2.1 JSON Instance Tree.* We model JSON documents as *JSON instance tree*s, which are node-typed, node-labelled and edge-labelled trees. Instance trees are extensions from *JSON tree*s introduced by Hutter and Augsten et. al. [17]. We define an instance tree as $I = (V_I, E_I, \Lambda_I, \Psi_I, \Gamma_I)$ with the following notations.

- $V_I$ is a set of nodes.
- $E_I \subseteq V_I \times V_I$ is a set of edges. Edges are used to express the relationship between an object and its key-value pairs, and between an array and its elements. An edge is made per each key-value pair and element.
- $\Psi_I : V_I \rightarrow \{obj, arr, prm\}$ maps each node to its node type. A node type is the type of its corresponding JSON document. When visualizing an instnace tree, the type of a node is expressed the shape of a node: circle(*object*), square(*array*), and triangle(*primitive* type).
- $\Lambda_I : V_I \rightarrow$ **Number** $\cup$ **String** $\cup\{$`true`,`false`,`null`$,\epsilon\}$ maps a node to its label. $\epsilon$ means an empty label. The node label is the JSON document itself if it is of primitive type. The labels are nullable($\epsilon$) - they are null when the type of the node $\Psi_I(v)$ is either *object* or *array*.
- $\Gamma_I : E_I \rightarrow$**String** $\cup\{\epsilon\}$ maps each edge to its edge label. $\Gamma_I(e)$ corresponds to the key of an object. $\Gamma_I(e)$ is also nullable, and it is null when $e$ is an outgoing edge of a node $v$ where $\Lambda_I(v)$ is *array*. This is because arrays use indices instead of keys, and this is expressed via the sequence of outgoing edges.

We transform a JSON document to a JSON tree as follows. Primitive JSON documents each become nodes with type of *prm*. The nodes are further labeled with the values of JSON documents. Objects become nodes of type *obj* (labeled as $\epsilon$) with the nodes of its values as children. The edge connecting an object node to its child is labeled with the key corresponding the value of the child. An array is converted to an *arr* (labeled as $\epsilon$) node with the $i$-th element becoming the $i$-th child subtree. We define a *level* of an instance tree as the conventional definition of a level of a tree. We follow the conventional definition of a level of a tree as the *level* of an instance tree. The root of an instance tree is at level 1, and its children and descendants have levels 2 and above.

*2.2.2 Schema Tree.* We also model a JSON schema as a tree, as it is expressed in recursive forms that can be captured well as a tree. A schema tree is a node-labeled, edge-labeled, and edge-typed tree defined as $s = (V_S, E_S, \Lambda_S, \Gamma_S, \Phi_S)$ with the following definition.

- $V_S$ is a set of nodes. Each node represents a JSON schema.
- $E_S \subseteq V_S \times V_S$ is a set of edges.
- $\Lambda_S : V_S \rightarrow \{$ `NUM`,`STR`,`BOOL`,`NULL`,`OBJ`,`ARR`,`ANYOF` $\}$ maps each node to its node label which corresponds to the type of the schema.
- $\Gamma_S : E_S \rightarrow$ **String** $\cup \{ * , \epsilon \}$ maps each edge to its edge label. An edge label expresses either the derivation from *key* part in the right side of *prop*, or an `"additionalProperties"` of *hetO* part with a Kleene star $*$. Edge label $*$ indicates the heterogeneity of the schema node which is the source node of that edge, while *key* indicates homogeneity.
- $\Phi_S : E_S \rightarrow \{$ `Required`, `Optional`, $\epsilon \}$ maps each edge to its type. The type of edge is only meaningful when $\Lambda_S(v)$ of its source node $v$ is `OBJ`. It indicates whether a key of a homogeneous object schema is `required`, or `optional`. In our visualization, edges of `required` type are expressed as solid lines, and edges of `optional` are expressed as dotted lines.

For both instance trees and schema trees, we define an operator $v[l]$ for a node $v$ and a label $l$.

$$v[l] = \begin{cases} v' & \text{if } \exists l \in \textbf{String}\ ,\ e = (v, v')\ ,\ \Gamma(e) = l \\ v' & \text{else if } \exists l \in \mathbb{Z}^+\ ,\ e = (v, v')\ ,\ v' \text{ is } l^{th} \text{ child of } v \\ \epsilon & \text{otherwise} \end{cases} \quad (1)$$

It is an operator that returns $v'$, the child subtree of $v$ such that an edge $e = (v, v')$ exists which $\Gamma(e)$ is $l$, or the $l^{th}$ child subtree of $v$ if $\Gamma(e)$ are all null. If such an edge does not exist, the return value is $\epsilon$. We denote $e_l$ as an edge of which $\Gamma(e) = l$. We denote $edgelabels(v)$ as a set of labels of $v$'s outgoing edges.

## 2.3 Schema Assessment Metric: MDL Cost

Ideal set of JSON schemas should be general enough to accept all given positive JSON document instances and be specific enough to reject all negative JSON document instances. We need a method to quantify such quality of a set of JSON schemas to navigate through the search space. For this, we adopt the Minimum Description Length (MDL) principle [25, 38, 39] to our JSD problem.

The MDL principle, based on the information theory, offers a way to evaluate the adequacy of models inferred from a data set. It states that the best theory to infer a model from a set of data is the one that minimizes the sum of two components — *Schema Representation Cost* (SRC) and *Data Representation Cost* (DRC). The best model (i.e., set of JSON schemas in our problem settings) inferred from a set of data is the one that minimizes *i)* the bits required to express the model and *ii)* the number of bits required to encode the data using the model. The former tells us how general the set of schemas is since, intuitively, the more general it is the smaller the number of bits is needed to represent it. However, if the set of schemas is too general, the cost of the latter (i.e., data representation) increases. The sum of both is referred to as *MDL cost*. The set of schemas with a smaller MDL cost is considered better.

*2.3.1 Applying MDL Cost to the JSD Problem:* Given a JSON document set $D$, we define our cost function $MDLCost$ as:

$$MDLCost(\mathcal{Z}, D) = SRC(\mathcal{Z}) + DRC(\mathcal{Z}, D) \quad (2)$$

where $\mathcal{Z}$ is a set of schemas. A single schema will be denoted with $\mathcal{S}$. SRC corresponds to component *i)*, and DRC corresponds to component *ii)* of the above.

**SRC(Schema Representation Cost):** We calculate SRC for each $\mathcal{S}$ in $\mathcal{Z}$ and sum them all. We take two steps to encode a node-labeled, edge-labeled tree into a string of bits. First, we encode $\mathcal{S}$ to an equivalent string of symbols $Str(\mathcal{S})$, then encode $Str(\mathcal{S})$ again to an equivalent string of bits. To encode $\mathcal{S}$ to $Str(\mathcal{S})$, we follow the method used by Fishman et. al. [21], and extend it by expressing edge labels within the string. Specifically, a parent-children relationship is expressed with parentheses `(`, and `)`. Each edge label $l$ is written left to its corresponding child as a (edge label, child) pair. Each pair is separated with a comma '`,`'. Second, we encode a sequence of symbols into a sequence of bits [25]. Let $\Sigma$ be the set of symbols in sequences in $Str(\mathcal{S})$. $\mathcal{M}$ is the set of metacharacters { `OBJ`, `ARR`, `NUM`, `STR`, `BOOL`, `NULL`, `ANYOF`, `(`, `)`, `,`, `*`, `!`, `?` }. The length of $Str(\mathcal{S})$ is denoted with $n$. Then, $SRC(\mathcal{Z})$ can be defined as follows.

$$\begin{aligned} SRC(\mathcal{Z}) &= \sum_{\mathcal{S} \in \mathcal{Z}} SRC(\mathcal{S}) \\ SRC(\mathcal{S}) &= n \lceil log(|\Sigma \cup \mathcal{M}|) \rceil \end{aligned} \quad (3)$$

The log term is the number of bits needed to encode a symbol in $\Sigma \cup \mathcal{M}$. This number of bits is needed for all $n$ symbols.

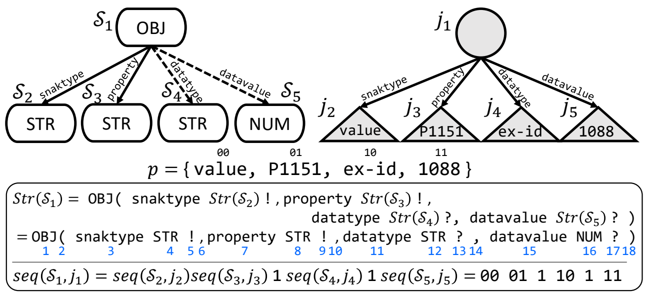
**DRC(Data Representation Cost):** DRC is defined in terms of the minimum number of bits needed to express $D$ using $\mathcal{Z}$.

$$\begin{aligned} DRC(\mathcal{Z}, D) &= \sum_{j \in D} \min_{\mathcal{S} \in \mathcal{Z}} DRC(\mathcal{S}, j) \\ DRC(\mathcal{S}, j) &= |seq(\mathcal{S}, j)| \end{aligned} \quad (4)$$

Let $seq$ be a function that returns the sequence of bits needed to express $j$ using $\mathcal{S}$. Since these choices are dependent on the schema node's type, we defined $seq$ differently for each type of schema. As a representative example, we give the definition of $seq$ function for homogeneous object schema $\mathcal{S}$:

$$seq(\mathcal{S}, j) = b_{(l_1, j)} seq(\mathcal{S}[l_1], j[l_1]) \dots b_{(l_m, j)} seq(\mathcal{S}[l_m], j[l_m]) \quad (5)$$

Here, we denote the labels for $\mathcal{S}$'s outgoing labels as $l_1, \dots, l_m$. Using one bit for each optional edge of $\mathcal{S}$, we encode whether the label of that edge has appeared (encode into '1') in $j$'s outgoing edges or not (encode into '0'). No bit is assigned ($\epsilon$) for a required



$p = \{$value, P1151, ex-id, 1088$\}$

$Str(\mathcal{S}_1) =$ OBJ( snaktype $Str(\mathcal{S}_2)$ ! ,property $Str(\mathcal{S}_3)$ ! , datatype $Str(\mathcal{S}_4)$ ?, datavalue $Str(\mathcal{S}_5)$ ? )
= OBJ( snaktype STR ! ,property STR ! ,datatype STR ? , datavalue NUM ? )
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

$seq(\mathcal{S}_1, j_1) = seq(\mathcal{S}_2, j_2) seq(\mathcal{S}_3, j_3)\ 1\ seq(\mathcal{S}_4, j_4)\ 1\ seq(\mathcal{S}_5, j_5) =$ 00 01 1 10 1 11

**Figure 5: An example calculation of MDL costs for an example schema $\mathcal{S}_1$ and an example instance $j_1$.**

edge of $\mathcal{S}$ since its label appears deterministically in $j$. Thus, given $\mathcal{S}$, the encoding rule of a bit for $\mathcal{S}$'s edge label $l$ and $j$ is expressed with $b_{(l,j)}$.

$$b_{(l,j)} = \begin{cases} \epsilon & \text{if } \Phi_S(e_l) = \texttt{Required} \\ 1 & \text{if } \Phi_S(e_l) = \texttt{Optional}\ ,\ l \in edgelabels(j) \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

*2.3.2 MDL Cost Example.* An example MDL cost calculation is given using Figure 5. We have one schema $\mathcal{S}_1$ and $D = \{j_1\}$.

**SRC Calculation.** We first encode $\mathcal{S}_1$ to a sequence of symbols $Str(\mathcal{S}_1)$. `Required` type edges are marked with ! symbol, and `Optional` type edges with ? symbol. Encoding of $Str(\mathcal{S}_1)$ that follows our definition is shown in Figure 5. The number of total symbols $n = 18$ (See blue counts under each symbol in Figure 5), and $\Sigma = \{$ `snaktype`, `property`, `datatype`, `datavalue` $\}$. Thus, $\lceil log(|\Sigma \cup \mathcal{M}|) \rceil = \lceil log(13+4) \rceil = 5$. Finally, $SRC(\mathcal{S}_1) = 18 * 5 = 90$.

**DRC Calculation.** Since $\mathcal{S}_1$ is a homogeneous object schema, we apply rule (D). The edges typed as ! are not accompanied by any bit, and the edges typed as ? are accompanied by 1. '1' indicates that the edges have those labels present in $j$. Function *seq* is recursively called for the child schemas $\mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_5$ and child instances $j_2, j_3, j_4, j_5$. For a primitive schema to encode primitive values, we need a set of all primitive values present in the dataset $D = \{j_1\}$. In this case, $|p| = 4$ (See $p$ in Figure 5). *seq* is expressed as simply encoding the index of each primitive value. As a result, $seq(\mathcal{S}_1, j_1) =$ `00 01 1 10 1 11`.

## 3 PROBLEM DEFINITION

Given a set of JSON documents $D$, there can be a large number of JSON schema sets with varying degrees of generality (or specificity) that can accept $D$. At one extreme, the most specific set consists of a set of schema whose set size equals the number of JSON documents, and each schema accepts only one JSON document instance. On the other side, we can have a set comprised of a single, most general schema that universally accepts any JSON documents. A true set of schemas we seek lies between these two extremes. Let $j$ be a JSON document ($j \in D$), $\mathcal{S}$ a single schema, and $\mathcal{Z}$ a set of schemas. We adopt the '$\models$' symbol such that $j \models \mathcal{S}$ means $j$ satisfies $\mathcal{S}$ as defined by Pezoa et al. [37] (Also see §2.1.2). We extend '$\models$' to $j$ and $\mathcal{Z}$, where $j \models \mathcal{Z}$ iff $\exists \mathcal{S} \in \mathcal{Z}\ s.t.\ j \models \mathcal{S}$.

We define JSON schema discovery (JSD) problem as the problem of deriving a set of schemas, denoted as $\mathcal{Z}$, from a given set of JSON documents in such a way that F1-score is maximized. Let $\mathcal{Z}_{D^+}$ be a set of schemas derived from $D^+$ which denotes a set of JSON documents we want to derive a set of schemas for. Informally we call $D^+$ a positive JSON document set. Similarly, we introduce a negative set $D^-$ as a set of JSON documents that should not be accepted by $\mathcal{Z}_{D^+}$.

$$\begin{aligned} \mathcal{Z} &= \underset{\mathcal{Z}_{D^+}}{\arg\max}\, F1(\mathcal{Z}_{D^+}, D^+, D^-) \\ &= \underset{\mathcal{Z}_{D^+}}{\arg\max}\, \frac{2 \times Recall(\mathcal{Z}_{D^+}, D^+) \times Precision(\mathcal{Z}_{D^+}, D^+, D^-)}{Recall(\mathcal{Z}_{D^+}, D^+) + Precision(\mathcal{Z}_{D^+}, D^+, D^-)} \end{aligned} \tag{7}$$

In Equation 7, we define 'Recall' of a given schema set $\mathcal{Z}_{D^+}$ as:

$$Recall(\mathcal{Z}_{D^+}, D^+) = \frac{|J'|}{|D^+|} \tag{8}$$

where $J' = \{j | j \in D^+, j \models \mathcal{Z}_{D^+}\}$. That is, we regard the ratio of positive JSON documents accepted by $\mathcal{Z}_{D^+}$ against the entire given JSON documents $D^+$ as the recall. Ideally, all input JSON documents $D^+$ should be accepted by $\mathcal{Z}_{D^+}$.

We also define 'Precision' of a given schema set $\mathcal{Z}_{D^+}$ as:

$$Precision(\mathcal{Z}_{D^+}, D^+, D^-) = \frac{|J^+|}{|J''|} \tag{9}$$

where $J'' = \{j | j \in \{D^+ \cup D^-\}, j \models \mathcal{Z}_{D^+}\}$ and $J^+ = \{j | j \in D^+\}$. If the derived set of schemas $\mathcal{Z}_{\mathcal{D}^+}$ is imperfect, $J''$ may contain both true and false positive JSON documents together. In computing the precision, we assume that we apply both $D^+$ and $D^-$ to $\mathcal{Z}_{D^+}$.

## 4 DESIGN OF RECG

### 4.1 Terminologies

We define a few terms specific to our ReCG algorithm needed in the rest of the paper. We use the term 'instance' to mean a tree-form JSON document instance where the context is unambiguous.

*4.1.1 level.* The term 'level' of a node is defined to be the length of the path from the root node of an instance tree to the target node. The root node of an instance is considered to be level 1.

*4.1.2 PD-instance.* It stands for '**P**artially-**D**erived' instance and refers to a JSON document instance that is partially transformed (through our derivation process) into a schema form. Thus, any nodes below some level $l$ in the instance tree have nodes converted into *schema nodes*. We use $p$ symbol to represent a PD-instance and $\mathcal{P}$ to represent a bag of PD-instances. Figure 6 shows examples of PD-instances. It depicts three PD-instances at different stages of derivation toward schema forms. PD-instance $p_1$ is made entirely of instance nodes but is trivially a PD-instance with no schema nodes. PD-instances $p_2$ and $p_3$ have parts of their nodes converted into schema node types.

*4.1.3 CD-instance.* We introduce the concept of CD-instance which stands for '**C**hildren-**D**erived' and the symbol $c$ is used to represent an individual CD-instance. It is a partial instance tree whose root node is the only instance node and all the descendant nodes are of

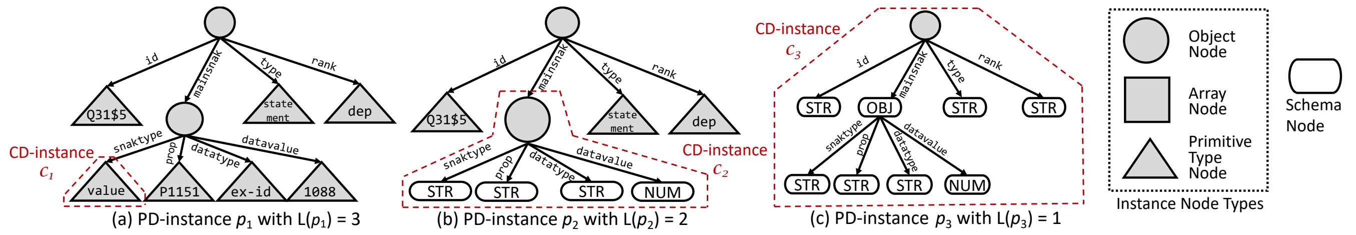


**Figure 6: The concepts of PD-instances and CD-instances.**

**Algorithm 1** High-level outline of ReCG's Algorithm

```
    In $D$ := A bag of JSON documents
    In $b$ := beam width
    Out $\mathcal{S}_D$ := A discovered JSON schema
1:  /* Set initial state with the input bag of JSON documents */
2:  $s_{init} \leftarrow \mathcal{J}_{input}$
3:  $s_{goal} \leftarrow \emptyset$
4:  $beam \leftarrow \{s_{init}\}$
5:  /* Perform beam search */
6:  while true do
7:      if $beam$.Begin().IsLeafState() then
8:          $s_{goal} \leftarrow$ GetLowest($beam$, $key = MDLCost$)
9:          break
10:     $nextStageStates \leftarrow \emptyset$
11:     for $s \in beam$ do
12:         $childrenStates \leftarrow$ GenerateChildrenStates($s$)
13:         $nextStageStates \leftarrow nextStageStates \cup childrenStates$
14:     SelectLowestK($nextStageStates$, $k = b$, $key = MDLCost$)
15:     $beam \leftarrow nextStageStates$
16: return $s_{goal}.\mathcal{Z}_{derived}$
```

schema node types as shown in Figure 6 with annotations $c_1$, $c_2$ and $c_3$. We use $C$ to indicate a bag of CD-instances. We also define a function $L$ that returns the level at which the root of CD-instances are located within a PD-instance, i.e., $L : \mathcal{P} \to \mathbb{Z}_{\geq 0}$.

## 4.2 Schema Search Space

Our search space consists of states that represent various stages of a schema discovery process. The initial single state is made entirely of JSON document instances (i.e., instance forest). As the schema discovery progresses, the search space fans out with different candidate states that contain partially derived schema sets, the PD-instances. At each stage of ReCG's bottom-up processing, it generates a spectrum of candidate schema sets that have a varying degree of generalities.

*4.2.1 Definition of State.* We define a state as a set of PD-instances whose $L(p)$ values are identical for those in which CD-instance exists. A state is denoted as $s_{i,j}$ with two designators $i$ and $j$.

- $i$: The stage number. (Stage is defined below.)
- $j$: A state identifier within a stage. At $i^{th}$ stage, there can be multiple states that contain PD-instances. PD-instances in these states at the same stage differ by the degree of generality.

**Algorithm 2** GenerateChildrenStates Function

```
    In $s_{in}$ := An input state
    Out $childrenStates$ := A set of children states
1:  $childrenStates \leftarrow \emptyset$
2:  $C \leftarrow$ GetCDInstances($s_{in}$)
3:  $C_{prm}, C_{arr}, C_{obj} \leftarrow$ GroupByType($C$)
4:  $\mathcal{Z}_{prm} \leftarrow$ DerivePrimitiveSchemaSet($C_{prm}$)
5:  $\mathcal{Z}_{arr} \leftarrow$ DeriveArraySchemaSet($C_{arr}$)
6:  $\{\mathcal{Z}_{obj_1}, \ldots, \mathcal{Z}_{obj_n}\} \leftarrow$ DeriveCandObjSchemaSets($C_{obj}$)
7:  for $\mathcal{Z}_{obj_i} \in \{\mathcal{Z}_{obj_1}, \ldots, \mathcal{Z}_{obj_n}\}$ do
8:      $childrenStates$.Insert(GenerateState($s_{in}, \mathcal{Z}_{prm}, \mathcal{Z}_{arr}, \mathcal{Z}_{obj_i}$))
9:  return $childrenStates$
```

*4.2.2 stage.* Within our *search space* of candidate schemas, the stage of a state $s$ is defined as the length of the path from the initial state $s_{0,1}$ to $s$ minus one. Thus, a stage number starts from 0. Stage of a state can also be computed by the following equation:

$$stage(s) = maxHeight(D) - L(p') + 1 \tag{10}$$

where $p'$ is any PD-instance in the state that has one or more CD-instances. The stage number increments as the schema derivation proceeds whereas the $L$ value decrements.

The initial state is $s_{0,1}$ and it contains a bag of input JSON document instances, $D^+$ with no levels yet converted into schema node types. ReCG produces a final set of candidate solution states when it reaches $L(p) = 0$ for all $p$ in a state.

## 4.3 ReCG Algorithm Overview

At the high level, ReCG's main algorithm follows a breadth-first beam search with configurable beam width (default=3) as in Algorithm 1 and 2. Our search is guided by the MDL principle [25, 38, 39] (See §2.3) and its cost valuation technique that gives preference to the one requiring a smaller amount of bits to express the schema and the instances. The end goal is to come up with a state that contains the most concise and accurate set of schemas for the given input JSON document instances. To cover as large search space as possible, a significant part of ReCG's effort is expended on generating and evaluating candidate schema sets of various generality which is embodied within the DeriveCandObjScehamSets of Algorithm 2.

*4.3.1 Illustrative Example.* To provide an intuition behind ReCG's algorithm, we describe an example using a sample of real-world JSON document instances in Figure 7. It shows a schema discovery process of ReCG with three simplified JSON instances. The root

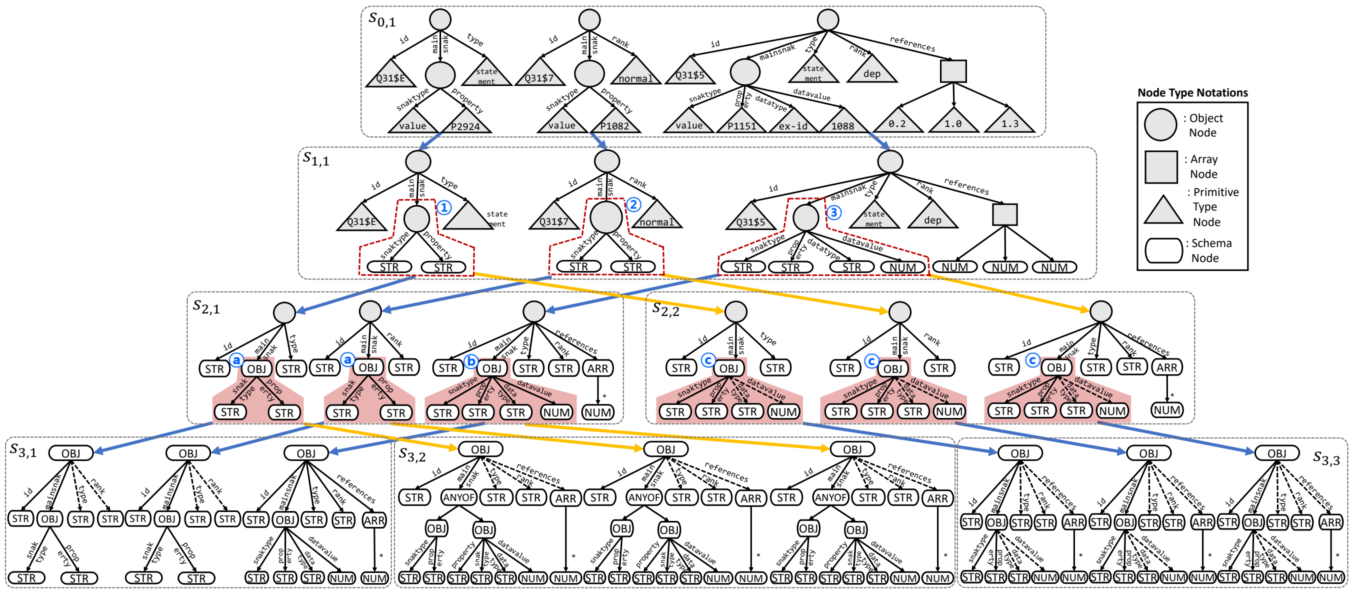


**Figure 7: An illustration of ReCG's search space using JSON documents picked from the Wikidata dataset and simplified. The resolution of schema nodes proceeds from the leaf to the root (i.e., bottom-up) as the state transitions from top to bottom.**

node $s_{0,1}$ contains initial JSON document instances or PD-instances with no nodes yet converted to schema nodes. ReCG's bottom-up schema discovery process starts from the leaf nodes of the instance trees. The transition from $s_{0,1}$ to $s_{1,1}$ shows that the instance nodes at level 3 corresponding to the primitive types are first converted to respective schema nodes of STR and NUM. Once all nodes at level 3 are resolved of their schema node types, it proceeds to nodes at level 2, transitioning from $s_{1,1}$ to $s_{2,1}$ and $s_{2,2}$. Nodes at level 2 of PD-instances include a mix of primitive data, object nodes, and array nodes. These object nodes are, by our definition, CD-instances (labeled in the figure as ①, ② and ③). To resolve the object nodes' schema types, we perform clustering of CD-instances by their structural similarities. Each resulting cluster is considered to represent a distinct object node type. We determine the object node types (ⓐ and ⓑ) and this completes the schema derivation at level 2 of $s_{2,1}$.

However, the schemas ⓐ and ⓑ just derived in $s_{2,1}$ are not the only way to satisfy the CD-instances ①, ② and ③. From this clustering result, we perform an incremental merging of the schemas to produce more general schemas. The state $s_{2,2}$ (generalized from $s_{2,1}$) in the figure shows an outcome of this schema generalization. This example is intentionally made simple that we have a single object type ⓒ, generalized from ⓐ and ⓑ. The processing of real-world data requires many more repeated generalizations to result in a much larger search space.

## 4.4 Schema Node Type Resolution

In this subsection, we describe how ReCG converts instance node types to schema node types at one stage. There are three types of nodes in an instance tree - object ($obj$), array ($arr$) and primitive ($prm$) type. We explain how we handle them to resolve the type of schema nodes.

*4.4.1 Schema Node Type Resolution of Object Type.* Among the three types of nodes, determining the schema node type for an object node is the most involving. Let us assume the current processing is at level $l$ where all nodes below $l$ are already resolved of their schema node types. The goal here is to determine a correct composition of schema node types of the object nodes at level $l$ that produced all observed CD-instances. Only the object nodes and the CD-instances attached to them (e.g., ①, ②, ③ of Figure 7) are of interest for now. The array and primitive nodes are handled separately.

Our handling of object nodes is based on the following line of reasoning. Given a set of CD-instances, we can treat each distinct CD-instance's tree structure as one separate schema node type. It represents the least general (and the most specific) schemas. On the other hand, we can construct the most general 'singleton' schema that encompasses all the CD-instances. The true composition of schema nodes lies somewhere in-between. We enumerate all possible schema node sets from the most specific to the most general following these two steps.

- *Initial clustering of CD-instances:* We first cluster (with DBSCAN technique, epsilon=0.5) the CD-instances. These initial clusters are the most specific set of schema nodes and also the basis for generating more general schema sets. For each cluster, one schema tree is generated.
- *Repetitive generalization of schema nodes:* From the initial schemas in the previous step, we perform incremental generalization. This generalization occurs in a hierarchical manner until no more generalization is beneficial. This expands our search space and ReCG algorithm explores them in a breadth-first manner. Further details of this *repetitive generalization* is given in §4.7.

These steps are expressed in Algorithm 3.

**Algorithm 3** DeriveCandObjSchemaSets Function Definition

```
    In $C$ := Object CD-instances
    Out $\{\mathcal{Z}_1, \dots, \mathcal{Z}_n\}$ := A set of set of schemas
 1: candSchSets ← ∅; candClustSets ← ∅
 2:     /* STEP 1 : CD-instance Clustering */
 3: $\{C_1, \dots, C_k\}$ ← ClusterCdInstances($C$)
 4: candClustSets.Insert($\{C_1, \dots, C_k\}$)
 5:     /* STEP 2 : Repetitive Generalization */
 6: candClustSets.InsertAll(RepetitivelyGeneralize($\{C_1, \dots, C_k\}$))
 7:     /* Derive schemas from clusters */
 8: for $\{C_1, \dots, C_i\} \in$ candClustSets do
 9:     $\mathcal{Z} \leftarrow \emptyset$
10:     for $C_j \in \{C_1, \dots, C_i\}$ do
11:         $\mathcal{S}$ ← DeriveSchemaFromCluster($C_j$)
12:         $\mathcal{S}$.AssignCDInstances($C_j$)
13:         $\mathcal{Z}$.Insert($\mathcal{S}$)
14:     candSchSets.Insert($\mathcal{Z}$)
15: return candSchSets
```

*4.4.2 Schema Node Type Resolution of Array Type.* We observe that arrays in the real-world datasets are typically heterogeneous as shown in the 'HomArr' column of Table 1. Out of 20 datasets we used, only three contained homogeneous array types. Thus, we assume that arrays are heterogeneous by default. To derive array schemas, we generalize (i.e., all edges connected to array elements' labels are turned into '$*$') the arrays. Generalization reduces the array schema to the form of a single '$*$' edge per a unique sub-schema tree attached to it. However, if the array element has nested structures, not just simple primitive types, then the array schema after this generalization can have a multi-level structure. Then, we perform the clustering once rather than repetitively as was done for objects.

We derive a homogeneous array for a cluster when array CD-instances within a cluster satisfy a few conditions. First, the number of elements in the array should be the same for all CD-instances. Second, the schemas of children at the same indices should be the same. If these conditions are met, it is a homogeneous array type.

*4.4.3 Schema Node Type Resolution of Primitive Type.* Node type resolution of primitive types is done trivially and unambiguously by simply converting values into their corresponding types. For example, we convert as these: `1`→`NUM`, "`hello`"→`STR`, `true`→`BOOL` and `null`→`NULL`.

## 4.5 CD-instance Clustering

During the processing at a certain level, we have a bag of CD-instances generated from a set of schema nodes whose true forms are unknown yet. The goal of CD-instance clustering is to find out these source object schema nodes whose type can be homogeneous, heterogeneous, composite, or a mix of them. Let $\mathcal{Z}_{hom}$, $\mathcal{Z}_{het}$, and $\mathcal{Z}_{comp}$ denote the set of homogeneous, heterogeneous, and composite object schemas, respectively. We want to find the true $\mathcal{Z}_{hom}$, $\mathcal{Z}_{het}$, and $\mathcal{Z}_{comp}$ sets. The CD-instance clustering described here needs to be performed *only once when the first child is created per upper stage's state.* Starting with this state from the clustering, adjacent sibling states are generated one after another with incrementally higher generality until no more general schemas are obtainable. This repetitive generalization and search space traversal strategy is discussed in more detail in §4.7.

*4.5.1 Distance Measure.* To perform effective clustering of CD-instances, we need to define a suitable distance measure. Since CD-instance is conceptually a tree, it may be that we need a similarity measure that can handle a tree data structure. However, we can treat them as a *set* whose elements are made of only the immediate children from the root object node. Any descendants can be ignored at this point since they have already been converted into singleton schema nodes with unique node IDs assigned from earlier steps of processing. Thus, CD-instances can be regarded as a set of (edge label, schema ID) pairs of the flat one-level trees.

This naturally leads our design of distance measure between CD-instances to be based on the Jaccard distance metric [29]. Intuitively, the more edge labels two CD-instances share, the higher the similarity should be. However, we adopt a more fine-grained policy by taking into account not just edge information, but also the schema IDs attached to them. For two given CD-instances, each edge is assigned the following scores:

- 1 if both the edge labels and the schema IDs match
- 0.5 if only the edge labels match, but schema IDs differ

Let $\mathcal{S}$ denote a schema and $c[l]$ point to the child node connected with the edge label $l$ in CD-instance $c$. We define $E(c)$ and $ES(c)$ as:

$$E(c) := \{l | l \in edgelabels(c)\}$$
$$ES(c) := \{(l, \mathcal{S}) | l \in edgelabels(c), \mathcal{S} = c[l]\}$$

That is, $E(c)$ represents the set of edge labels, and $ES(c)$ the set of (edge label, schema ID) pairs. Then, $|E(c_1) \cap E(c_2)|$ is the number of common edges. And, $|ES(c_1) \cap ES(c_2)|$ is the number of common edge labels that also have matching schema IDs. Our distance measure $\mathcal{D}$ is defined as:

$$\mathcal{D}(c_1, c_2) := 1 - \frac{|E(c_1) \cap E(c_2)| + |ES(c_1) \cap ES(c_2)|}{2|E(c_1) \cup E(c_2)|} \tag{11}$$

Let us consider two CD-instances $c_5, c_6$ in Figure 8 as an example. The parts (edges labels, schema IDs) common to both CD-instances are colored in red, and otherwise black. The union of edge labels is `full_text`, `text range`, `entities`, and `extended entities`. Two CD-instances have 3 labels `full_text`, `text range`, `entities` in common. Of those, `entities` is penalized by 0.5 for not having the same schema for $c$[`entities`] in both $c_5$ and $c_6$. Then, the distance between $c_5$ and $c_6$ is computed as $\mathcal{D}(c_5, c_6) = 1 - \frac{2.5}{4} = 0.375$.

*4.5.2 Two-Phase Clustering.* `ReCG` performs clusterings in two phases to identify homogeneous, heterogeneous, and composite object schema nodes, The first clustering is carried out with the goal of identifying *clusters of homogeneous/composite objects* and *outliers.* We use DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [18, 52] as our clustering method. It is expected that if some of the CD-instances are actually from homogeneous or composite schemas, they would exist in sufficient quantity to form clusters. For composite objects, a simple preprocessing is necessary to facilitate the discovery of composite object schema nodes. Recall that composite object schemas have a mix of fixed edge labels and

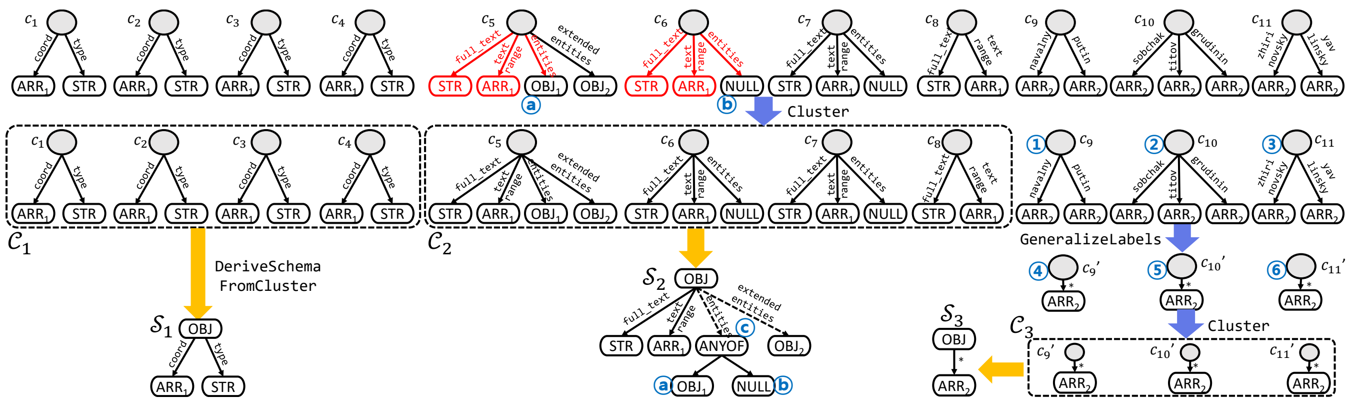


**Figure 8: Clustering example that illustrates how homogeneous, composite, and heterogeneous schemas are derived.**

the edges with '∗' whereas all edge labels of homogeneous object schema nodes are fixed. Since '∗' edges manifest as rarely seen labels in JSON document instances, we change any edge labels whose occurrence is below the threshold count to '∗'. The threshold is currently set to be 10 empirically. With this preprocessing applied, we run the DBSCAN clustering and obtain the clusters that represent homogeneous and composite object nodes.

In the second phase of clustering, we set out to find object nodes of heterogeneous type. Any instances flagged as outliers by DBSCAN are the target of the second phase clustering. We assume here that instances generated from the heterogeneous object schema nodes would be classified as outliers because of a high diversity of edge labels and insufficient quantity per edge labels. Thus, in these outlier instances that are supposedly from heterogeneous schemas, we generalize objects by converting edge labels to '∗'. This eliminates diversities of edge labels and the only factor that determines the distance becomes the schemas of the children. We perform the second DBSCAN clustering on them to find clusters of heterogeneous objects.

## 4.6 Schema Derivation From Each Cluster

To derive a schema, we collect two metadata from each CD-instance $c$ in a cluster: *i)* Edge labels in $c$ with their counts, and *ii)* Schema IDs of children for each edge label. A schema is derived in the following steps.

(1) Schema node $v$ is generated with $\Lambda_S(v) =$ OBJ.
(2) Edge is generated per edge label present in the metadata. Each edge $e$ is assigned a label in the metadata. We assign $\Phi_S(e_l)$ Required if the edge with the label $l$ is always present in all CD-instance $c$ in the cluster $C$.
(3) A schema tree is assigned to the destination of each edge $e_l$. We examine all the CD-instances $c$ in $C$ for $c[l]$, and aggregate distinct schemas. If there is only a single distinct schema, we just assign it to the edge's destination. However, when two or more distinct schemas exist, we derive another ANYOF node, and assign that node to the edge's destination.

For example, CD-instances at cluster $C_2$ in Figure 8 have four distinct edge labels: `full_text`, `text range`, `entities`, and `extended entities`. We first make four outgoing edges from an object node and assign these four labels to each edge. We then count the occurrences of each edge label, which are 4, 4, 3, 1. Labels `full_text`, and `text range` appear in every CD-instance. Thus, $\Phi_S(e_{\texttt{full\_text}})$, and $\Phi_S(e_{\texttt{text range}})$ are marked as Required, and others as Optional. Then, for each label $l$, we examine all $c[l]$ for distinct schemas. For label `entities`, there are two distinct schemas ⓐ and ⓑ. In the derived schema, ANYOF node (ⓒ) is appended.

**Algorithm 4** RepetitivelyGeneralize

```
    In {C_1, ..., C_k} := a set of object clusters
    Out {{C_1, ..., C_{k-1}}, ..., {C_1, ..., C_{k-j}}} := a set of set of clusters
 1: candClustSets ← ∅
 2: currClustSet ← {C_1, ..., C_k}  ▷ A set of clusters to be repetitively merged
 3: /* Repetitively merge a pair of clusters, until no viable pair exists */
 4: while true do
 5:     mergeCands ← ∅
 6:     /* Check viableness of a merge */
 7:     for (C_i, C_j) ∈ currClusts × currClustSet where i < j do
 8:         if Viable(C_i, C_j) then
 9:             mergeCands.Insert( (C_i, C_j) )
10:     if mergeCands.Empty() then break
11:     /* Merge two clusters with the least distance */
12:     (C_i, C_j) ← ArgMin_{(C_i,C_j)}(mergeCands, key = D^c)
13:     currClustSet.Insert( Merge(C_i, C_j) )
14:     currClustSet.Delete(C_i); currClustSet.Delete(C_j)
15:     candClustSets.insert(currClustSet)
16: return candClustSets
```

## 4.7 Repetitive Generalization of Schemas

The CD-instance clustering results described in previous subsections produce the most specific schema set ReCG can generate at a stage. This clustering is performed when the processing reaches a state and the first child state is to be created. Subsequently, from this initial schema set, ReCG produces incrementally more general schema sets to form a series of sibling states. Here, we describe how ReCG generalizes the schema set repetitively to obtain schema sets of varying generality up to the highest generality. The overall process of repetitive merging is expressed in Algorithm 4.

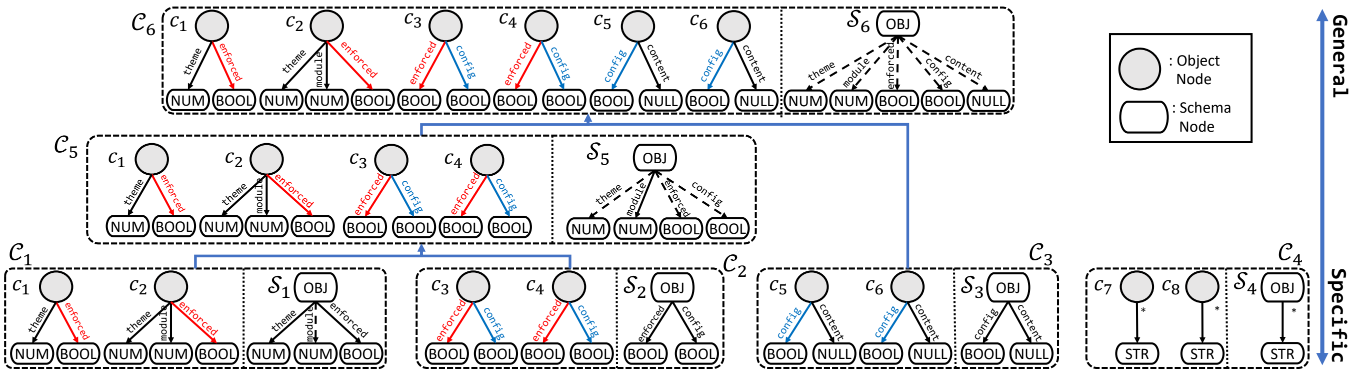


**Figure 9: Example of repetitive generalization via merging in simplified Twitter dataset.**

*4.7.1 Hierarchical Merging.* We generate sets of schemas with varying generality through hierarchical clustering in which two closest schemas are merged iteratively. Hierarchical merging is guided by two criteria: the *viableness* of the merge and the MDL cost.

Let us first define notations at the cluster level. Recall previously defined notations for individual CD-instances: $c_i[l]$ points to the child node connected with the edge label $l$ in CD-instance $c_i$ and $E(c_i)$ is the set of edge labels of $c_i$.

$$\begin{aligned} \mathbf{E}^c(C) &= \{l \mid l \in \cup_{i=1}^{n} E(c_i), c_i \in C\} \\ \mathbf{S}^c(C) &= \{s \mid s \in \cup_{i=1}^{n} c_i[l],\ l \in E(c_i),\ c_i \in C\} \\ \mathbf{T}^c[l](C) &= \{s \mid s \in c_i[l], c_i \in C\} \end{aligned} \tag{12}$$

$\mathbf{E}^c$ represents the set of all edge labels present within a cluster $C$, and $\mathbf{S}^c$ the set of children schemas within the cluster $C$. $\mathbf{T}^c[l](C)$ returns the set of schemas that are present under edges of label $l$ for all CD-instances in $C$. For each pair of clusters, we check whether merging two clusters is viable or not using the following conditions.

*Definition 4.1 (viableness).* A merge of two clusters $C_1, C_2$ is considered *viable* if any one of the following conditions holds.

$$\left(\mathbf{E}^c(C_1) \cap \mathbf{E}^c(C_2) - \{*\} \neq \emptyset\right) \wedge \left(\mathbf{T}^c[*](C_1) == \mathbf{T}^c[*](C_2)\right) \tag{13}$$

$$\left(\mathbf{S}^c(C_1) \subseteq \mathbf{T}^c[*](C_2)\right) \vee \left(\mathbf{S}^c(C_2) \subseteq \mathbf{T}^c[*](C_1)\right) \tag{14}$$

$$\mathbf{S}^c(C_1) \cap \mathbf{S}^c(C_2) \neq \emptyset \tag{15}$$

Equation 13 checks whether there is an overlap between the labels of two clusters, except '$*$'. Merge between two homogeneous clusters without overlapping any label would result in deriving a schema that validates objects with unseen combinations of edge labels. Equation 14 checks if one cluster $C_1$ can be captured entirely by the heterogeneous pattern of another cluster $C_2$. If it is true, the labels in $C_1$ will be generalized to '$*$' once such merge is performed. Equation 15 checks if two heterogeneous patterns can be generalized further.

If a merge of two clusters is determined to be *viable*, we calculate the distance between two clusters using the SRC (Schema Representation Cost) component within the MDL. The distance is defined as the difference of SRC before and after merging two clusters $C_1, C_2$ into $C_m$. The pair with the smallest SRC cost difference (i.e., the smallest change in generality) is merged. Suppose the schemas derived from each cluster as $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_m$. Then, the distance $\mathcal{D}$ between two clusters $C_1, C_2$ is defined as:

$$\mathcal{D}^c(C_1, C_2) := \frac{|SRC(\mathcal{S}_1) + SRC(\mathcal{S}_2) - SRC(\mathcal{S}_m)|}{SRC(\mathcal{S}_m)} \tag{16}$$

Figure 9 illustrates the hierarchical merge. Initially there are four clusters $C_1, C_2, C_3$, and $C_4$. The edge labels `enforced` and `config` are common among clusters. There are two viable pairs of merges, $(C_1, C_2)$ and $(C_2, C_3)$. Of those, the pair $(C_1, C_2)$ is merged to cluster $C_5$ since they show the least distance. In the next step, only $(C_3, C_5)$ is viable for merge. They are merged to $C_6$. $(C_4, C_6)$ is not a viable pair of merge, and thus the merging phase ends.

## 4.8 Time Complexity Analysis

Let the number of JSON document instances be the input size $n$. The max height of the search space is denoted as $H$. The beam width is $b$. And, $s$ indicates the stage number. We also introduce the average branching factor (i.e., fan out factor) of the instance tree $f$. Let $m$ be the number of CD-instances at stage $s$. The number of CD-instance $m$ can be expressed in terms of $n$ as:

$$m = n \times f^{(H-s)} \tag{17}$$

The cost of a single stage consists of *i)* CD-instance clustering and *ii)* repetitive generalization via hierarchical clustering. The time complexity of DBSCAN [52] clustering is known to be O($n^2$) and hierarchical clustering is O($n^3$) [36]. Let us first express the cost of clustering. We perform the clustering by sampling as follows to avoid the excessive cost of clustering all CD-instances.

(1) Sample a fixed number of CD-instances, denoted as $\hat{m}$, from the current bag of CD-instances.
(2) Perform DBSCAN clustering on $\hat{m}$ CD-instances.
(3) Select the dominant cluster and its corresponding schema.
(4) Eliminate CD-instances that match the found schema from the current bag of CD-instances.
(5) Create a cluster using the eliminated CD-instances.
(6) Repeat until all CD-instances are depleted.

Let us assume an average reduction ratio $r$ by which the number of CD-instances reduces at the elimination step above. Then, the total number of clustering to be performed is $\log_r m$. Clustering requires a constant $\hat{m}^2$ computations. Elimination of CD-instances requires $m/r^i$ oeprations where $i$ is the iteration count of the above steps. Putting all these together, we obtain:

$$\begin{aligned}\sum_{i=0}^{\log_r m}\left(\hat{m}^2+\frac{m}{r^i}\right) &= \hat{m}^2(1+\log_r m)+m\cdot\sum_{i=0}^{\log_r m}\frac{1}{r^i}\\ &= \hat{m}^2(1+\log_r m)+m\cdot\left(1+\frac{1-1/r^{\log_r m+1}}{1-1/r}\right)\\ &= \hat{m}^2(1+\log_r m)+m\cdot\left(1+\frac{rm-1}{rm}\cdot\frac{r}{r-1}\right)\\ &= \hat{m}^2+\hat{m}^2\log_r m+m+\frac{r}{r-1}\cdot m-\frac{1}{r-1}\end{aligned} \tag{18}$$

Since there are $\log_r m$ clusters, the time complexity of hierarchical clustering becomes $(\log_r m)^3$. Combining these two, the time complexity of processing a single stage becomes:

$$O(\hat{m}^2+\hat{m}^2\log_r m+\frac{2r-1}{r-1}\cdot m-\frac{1}{r-1}+(\log_r m)^3)=O(m) \tag{19}$$

Now, let us express the time complexity in terms of $n$. Since there are $H$ number of stages, the overall time complexity in terms of $n$ (Equation 17) can be expressed as:

$$\begin{aligned}\sum_{s=0}^{H-1}O(m) &= \sum_{s=0}^{H-1}O(n\cdot f^{H-s})\\ &= O\left(\sum_{i=1}^{H}n\cdot f^i\right)\\ &= O\left(n\sum_{i=1}^{H}f^i\right)\end{aligned} \tag{20}$$

From Equation 20, since the summation part does not contain the input size $n$, they can be eliminated and the final time complexity becomes $O(n)$.

## 5 EVALUATION

We evaluated our technique on the following aspects

- Quality of discovered schema in F1 metric (§5.2): We measured the recall, precision, and F1 using positive samples of real-life datasets, and synthetically generated negative samples. Our ReCG algorithm showed 42% and 45% higher F1 score compared to Jxplain and KReduce in all 20 datasets, respectively.
- Validity of MDL costs as a schema quality measure(§5.3): ReCG's MDL cost was 5× smaller than competitors indicating superior conciseness and preciseness of discovered schemas. Observed MDL costs exhibited strong negative correlation with the accuracy of -0.84.
- Scalability comparison in terms of dataset sizes (§5.4): We measure the scalability of ReCG as well as two other competing techniques using 20 datasets. ReCG produced 2.40× better scalability than the main competitor.

### 5.1 Experimental Setup

*5.1.1* **Hardware and Software Settings.** We conducted our experiment on a machine with Intel Xeon E5-2680 v4 @2.40GHz CPU and 756 GB of RAM with the OS of Ubuntu 20.04. ReCG's was implemented with C++ and was compiled with C++ 8.3.0. Our competitors were implemented in scala [4] and spark [42].

*5.1.2* **Compared Techniques.** Our ReCG technique is compared against two other techniques - Jxplain and KReduce.

- Jxplain [42]: It is the most recently proposed JSD algorithm that uses a top-down schema generation approach. It uses *key-space entropy* to determine the heterogeneity of objects, then performs *Bimax-Merge* clustering algorithm based on keys if homogeneity is confirmed.
- KReduce [4]: It also uses a top-down schema generation approach. KReduce mainly focuses on the efficiency of the algorithm. It tends to over-simplify the JSD problem by only finding homogeneous object schemas for objects, and heterogeneous array schemas for arrays. Thus, KReduce is expected to be fast but also expected to show a low F1 score on datasets that do not conform to its assumptions.

*5.1.3* **Datasets.** We employed 20 datasets, of which 12 were real-life datasets and 8 were synthetic. Real-life datasets contained JSON document instances with their ground truth schemas. For the datasets of NYT [44], Twitter [46], Pharamaceutical [2], Wikidata [50], Yelp [14], VK [22], ETH [19], ThaiMovies [41], JSON schema was not directly provided and had to be manually constructed from their formal documentation about their JSON documents. For the datasets of NYT, Twitter, Pharamaceutical, Wikidata, Yelp, VK, ETH, ThaiMovies, JSON schema was not directly provided and had to be manually constructed from their formal documentation about their JSON documents.

Synthetic datasets are generated from schemas obtained from JSON schema store [31] , which is a repository storing real-life schemas. JSON schema store does not provide the corresponding JSON document instances for each schema, and thus synthetic instances had to be generated. A total of ten thousand instances were synthesized for each schema using two software tools, DataGen [40] and json-schema-faker [12], that generate instances that conform to the given schema.

Various characteristics of the datasets are presented in Table 1. It shows the characteristics of the ground truth schema $\mathcal{S}_G$, and its corresponding positive document set $D^+$ for each dataset. For the schema $\mathcal{S}_G$, we report the height of the tree, the total number of all nodes, and the nodes of each type. Also for $D^+$, we show the number of instances and the average number of nodes per instance.

We generated a negative document set $D^-$ for each dataset, using the ground truth schema $\mathcal{S}_G$. The generation process followed the steps described below:

(1) Modification of $\mathcal{S}_G$ into schema $\mathcal{S}_G^-$, a schema that can accept instances that $\mathcal{S}_G$ rejects.
(2) Generation of synthetic dataset $D^{-\prime}$. Every synthetic instance in $D^-$ is accepted by $\mathcal{S}_G{}^-$.
(3) Validation of each synthetic instance $j^- \in D^{-\prime}$ against $\mathcal{S}_G$. The instances that are validated by $\mathcal{S}_G$ are dropped.

The size of $D^-$ was made equal to the size of $D^+$ for each dataset.

**Table 1: Statistics of 20 datasets used in experiments.**

| Dataset | | Schema | | | | | | | | Instance | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Type | Name | Height | $\lvert V_S \rvert$ | # of OBJ nodes | | | # of ARR nodes | | # of ANYOF nodes | $\lvert D^+ \rvert$ | avg($\lvert V_I \rvert$) |
| | | | | Hom Obj | Het Obj | Com Obj | Hom Arr | Het Arr | | | |
| Real World | NYT | 6 | 92 | 9 | 0 | 0 | 0 | 3 | 14 | 10k | 85.21 |
| | Twitter | $\infty$ | $\infty$ | 20 | 1 | 0 | 12 | 10 | 16 | 10k | 206.16 |
| | Github | 11 | 3471 | 171 | 0 | 3 | 0 | 29 | 335 | 10k | 116.64 |
| | Pharmaceutical | 3 | 12 | 2 | 1 | 0 | 0 | 0 | 0 | 10k | 31.77 |
| | Wikidata | 14 | 179 | 31 | 7 | 0 | 0 | 8 | 15 | 10k | 1927.96 |
| | Yelp | 5 | 79 | 7 | 1 | 0 | 0 | 0 | 5 | 10k | 12.32 |
| | VK | 11 | 335 | 40 | 0 | 0 | 0 | 7 | 2 | 10k | 30.50 |
| | ETH | 8 | 112 | 8 | 0 | 0 | 1 | 6 | 6 | 10k | 1004.69 |
| | Iceberg | 4 | 9 | 1 | 1 | 0 | 0 | 1 | 0 | 1523 | 1288.30 |
| | Ember | 6 | 68 | 8 | 1 | 0 | 0 | 9 | 0 | 10k | 902.86 |
| | GeoJSON | 8 | 41 | 6 | 0 | 0 | 2 | 5 | 1 | 10k | 52.65 |
| | ThaiMovies | 8 | 112 | 14 | 0 | 0 | 0 | 11 | 6 | 1364 | 433.79 |
| Synthetic | RDB | 3 | 13 | 1 | 0 | 1 | 0 | 1 | 0 | 10k | 14.76 |
| | AdonisRC | 7 | 64 | 5 | 2 | 2 | 0 | 9 | 3 | 10k | 27.77 |
| | HelmChart | 7 | 50 | 4 | 0 | 1 | 0 | 6 | 1 | 10k | 33.76 |
| | Dolittle | 6 | 52 | 14 | 6 | 0 | 0 | 3 | 1 | 10k | 48.82 |
| | Drupal | 6 | 100 | 10 | 7 | 0 | 0 | 17 | 5 | 10k | 47.96 |
| | DeinConfig | 8 | 97 | 3 | 1 | 2 | 0 | 13 | 17 | 10k | 44.94 |
| | Ecosystem | 6 | 120 | 5 | 3 | 1 | 0 | 12 | 9 | 10k | 132.59 |
| | Plagiarize | 4 | 15 | 2 | 1 | 1 | 0 | 0 | 2 | 10k | 8.23 |

## 5.2 Accuracy Comparison

We measured and compared the accuracy of discovered schemas in terms of the F1 score. Recall and precision are computed as defined in § 3 for all 20 datasets in § 5.1.3. Similar to Jxplain, we sampled 1%, 10%, 50%, and 90% of positive sample sets from $D^+$, and they were given as input to each algorithm. The test dataset comprised 10% of instances sampled from $D^+$ (those that do not overlap with the input instances), and 90% of instances sampled from $D^-$. The ratio of 1:9 was chosen to mimic a realistic situation of negative samples outnumbering positive samples.

Figure 10 presents the aggregated summary of comparison results. Raw measurements of 10% of 20 datasets are given in Table 2. All algorithms were run using default parameters, in our case $beamWidth = 3$, $sampleSize = 1000$. Overall, ReCG gave superior F1 scores against competitors irrespective of the data set proportions. F1 score of ReCG was higher than Jxplain and KReduce by 42% and 45%, respectively. Cases that Jxplain failed to run were excluded from F1 calculation. Higher F1 score of ReCG was contributed more by the recall than precision as confirmed in Figure 10 (b) and (c).

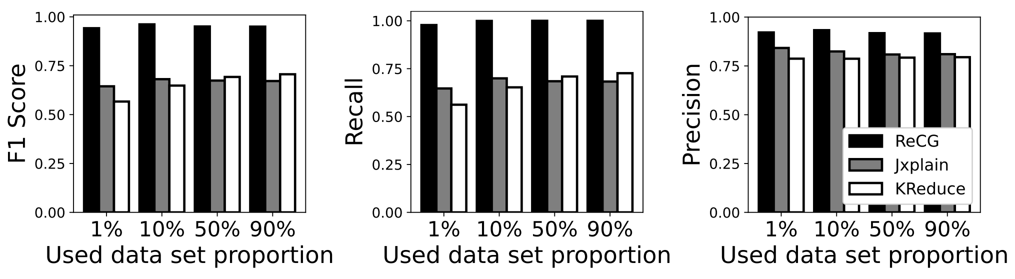
According to our investigation, the lower F1 score of Jxplain and KReduce can be attributed to the following four factors. *First, it was the inability to correctly partition heterogeneous objects or arrays that should be recognized as different schemas types.* In Jxplain, once a heterogeneous object schema node is derived, it does not further partition them even though there can be more than one different types of heterogeneous object schemas. In case of KReduce, it does not support heterogeneous object schemas. As a result, Jxplain and KReduce both show low precision on ETH, GeoJSON, and Drupal datasets. This is due to the generation of a single heterogeneous object (or array), from multiple heterogeneous objects (or arrays). The resulting discovered schema is thus more general than the ground truth schema, thus, low precision. *Second, it is the inability to correctly derive a schema node.* Jxplain and KReduce both use their own heuristics that do not always conform to all datasets. They derive homogeneous object schema nodes incorrectly where it is the heterogeneous object schema in the ground truth schema. This causes two algorithms to show low recall in Iceberg, Wikidata datasets. *Third,* Jxplain *and* KReduce *both do not handle composite object schemas.* They both treat composite object schemas as homogeneous object schema nodes resulting in a descriptive (i.e., having many schema nodes) homogeneous object schema that lists all the appearing edge labels within the input instance forest. As a result, we observed that Jxplain and KReduce both showed low recall on RDB, AdonisRC, HelmChart, DeinConfig, Ecosystem, and Plagiarize. *Fourth,* Jxplain *and* KReduce *are unable to partition homogeneous and heterogeneous objects at the same time.* KReduce does not assume ANYOF schema nodes that have two or more object schema nodes as children. On the other hand, Jxplain only assumes ANYOF schema nodes that have two or more homogeneous object schema nodes as children. KReduce finds a single homogeneous object schema, and Jxplain partitions the objects into multiple homogeneous object schemas, both resulting in low recall. Jxplain and KReduce both showed low recall on Dolittle and Drupal datasets, which had both homogeneous object schema nodes and heterogeneous object schema nodes as children of ANYOF schema nodes.

## 5.3 MDL Cost Analysis

We compared and analyzed MDL costs of the schemas produced by Jxplain, KReduce and ReCG to observe the quality of produced

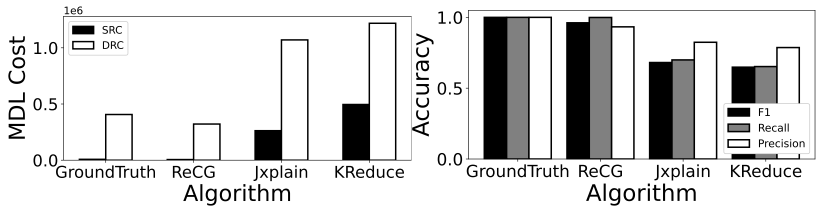
**Table 2: Recall, Precision, and F1 Score for schemas discovered by each algorithm on 10% of all datasets.**

| Dataset | ReCG | | | Jxplain | | | KReduce | | |
|---|---|---|---|---|---|---|---|---|---|
| | Recall | Precision | F1 | Recall | Precision | F1 | Recall | Precision | F1 |
| NYT | 1.00 | 1.00 | 1.00 | Runtime Error | | | 1.00 | 1.00 | 1.00 |
| Twitter | 1.00 | 1.00 | 1.00 | 0.02 | 1.00 | 0.03 | 0.99 | 1.00 | 1.00 |
| Github | 1.00 | 1.00 | 1.00 | 0.48 | 1.00 | 0.64 | 1.00 | 1.00 | 1.00 |
| Pharmaceutical | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.85 | 1.00 | 0.92 |
| Wikidata | 1.00 | 1.00 | 1.00 | Time Out | | | 0.47 | 1.00 | 0.64 |
| Yelp | 1.00 | 0.70 | 0.83 | 0.97 | 0.77 | 0.86 | 1.00 | 0.36 | 0.53 |
| VK | 1.00 | 1.00 | 1.00 | 0.99 | 1.00 | 0.99 | 0.99 | 1.00 | 1.00 |
| Iceberg | 1.00 | 1.00 | 1.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Ember | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.81 | 1.00 | 0.89 |
| ETH | 1.00 | 1.00 | 1.00 | 0.76 | 0.31 | 0.44 | 1.00 | 0.32 | 0.48 |
| GeoJSON | 1.00 | 0.96 | 0.98 | 1.00 | 0.71 | 0.83 | 1.00 | 0.42 | 0.59 |
| ThaiMovies | 0.99 | 1.00 | 1.00 | 0.85 | 1.00 | 0.92 | 0.99 | 0.95 | 0.97 |
| RDB | 1.00 | 0.84 | 0.91 | Time Out | | | 0.53 | 0.85 | 0.65 |
| AdonisRC | 1.00 | 0.76 | 0.86 | 1.00 | 0.82 | 0.90 | 0.34 | 0.94 | 0.50 |
| HelmChart | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.57 | 1.00 | 0.72 |
| Dolittle | 1.00 | 0.70 | 0.82 | 0.33 | 1.00 | 0.49 | 0.67 | 0.79 | 0.73 |
| Drupal | 1.00 | 0.96 | 0.98 | 0.06 | 0.82 | 0.11 | 0.01 | 0.13 | 0.01 |
| DeinConfig | 1.00 | 0.97 | 0.98 | 1.00 | 0.93 | 0.96 | 0.33 | 1.00 | 0.50 |
| Ecosystem | 1.00 | 1.00 | 1.00 | Runtime Error | | | 0.28 | 1.00 | 0.44 |
| Plagiarize | 1.00 | 0.78 | 0.88 | Time Out | | | 0.31 | 1.00 | 0.47 |



(a) F1 score (b) Recall (c) Precision

**Figure 10: Accuracy comparison with varying dataset sizes.**



(a) Average MDL cost (b) Average accuracy

**Figure 11: Average MDL cost and accuracy of the ground truth schema, and the schemas found by three algorithms for 20 datasets. The proportion of the used dataset is 10%.**

schemas. The measurements were made on all 20 datasets and the input $J_{input}$ was comprised of 10% of $D^+$. Figure 11 shows the average of MDL costs and accuracy measures. The numbers from the ground truth schema are also plotted together for reference.

Figure 11 (a) shows that ReCG obtained the least average MDL cost among three algorithms. ReCG's MDL cost came out to be 4.1× and 5.2× smaller than Jxplain and KReduce, respectively, while maintaining high F1 score of 0.95. Both Jxplain and KReduce received about 29.5% and 31.1% less F1 score than ReCG, respectively. Among the components of MDL costs, the SRC of ReCG were superior – 45.82× and 86.48× smaller than Jxplain and KReduce, respectively. This indicates that the schema discovered by ReCG is

**Table 3: Correlation of SRC, DRC and MDL cost with metrics of accuracy.**

| | Recall | Precision | F1 score |
|---|---|---|---|
| SRC | -0.66 | -0.54 | -0.62 |
| DRC | -0.65 | -0.06 | -0.58 |
| MDL | -0.90 | -0.38 | -0.84 |

more concise without the loss of precision or recall. In comparison with the ground truth schema, the MDL cost of ReCG's discovered schema was very close to that of the ground truth schema, within 21.0% differences on average. On 14 datasets, ReCG's schema scored even less MDL costs than the ground truth schema due to the following two reasons. First, there were cases (NYT, Twitter, Github, VK, ThaiMovies, GeoJSON) where the ground truth schema itself was described as too general according to the official documentation. Second, ReCG found some frequent CD-instances and clustered them to derive homogeneous object schemas, which were originally labeled as heterogeneous objects according to the ground truth schema (Pharmaceutical, Wikidata, Yelp, Ember, ETH, RDB, HelmChart, DeinConfig). This resulted in higher SRC, but much lower DRC compared to the ground truth schema.

There were only two exception cases in VK and AdonisRC datasets, where KReduce and Jxplain showed less MDL cost than ReCG. This turned out to be the issue of epsilon value mentioned in §5.2. ReCG's result schema was more general than it should have been with higher MDL cost than the competitors. VK and AdonisRC were the schemas that conformed well to the assumptions of KReduce and Jxplain. (Jxplain assumes that all arrays are heterogeneous. KReduce assumes objects are always homogeneous and arrays are always heterogeneous.)

We found that the MDL costs had a strong negative correlation with F1 scores as shown in Table 3. The correlation came out to be a strong negative correlation of -0.84 between MDL and F1 score. This was contributed the most by the strong negativity with *Recall* (-.90) rather than with *Precision* (-.38). SRC and DRC had roughly the same degree of correlation with recall and precision. These observations indicate that the conciseness of a schema (i.e., SRC) has a stronger relation with its generality.

## 5.4 Scalability with Dataset Size

We compared the scalability of three algorithms by varying the data sizes from 10% to 100% with 10% step. Table 4 is the average runtime measurements at selected data sizes of 10%, 50%, and 100%.

Overall, ReCG outperformed Jxplain by a significant margin (1.69× to 2.40×) on average but KReduce was faster by about 50%. ReCG's performance is positioned approximately in the middle of KReduce and Jxplain. Jxplain tends to perform better than ReCG on the datasets if they contain a small number of distinct keys with only homogeneous objects. VK, Ember and ThaiMovies belong in this category. This is because Jxplain determines objects' heterogeneity per a distinct sequence of labels present in the input JSON documents. Thus, a smaller number of distinct keys reduces the computational overhead significantly. On the other hand, ReCG outperforms Jxplain for the datasets with a moderate or large number of distinct keys.

Interestingly, the execution time of Yelp, AdonisRC, Ecosystem and Plagiarize datasets showed a sudden drop in runtimes of ReCG. This was due to the sudden reduction of the total number of states as the portion of the used data set increased. CD-instances that were previously determined to be separate clusters when the data was small were grouped as a single cluster when the number of CD-instances increased and new CD-instances were introduced.

Jxplain exhibited exponential growth in runtime for datasets of Dolittle and Drupal. It failed to run to completion on datasets of Wikidata, RDB, and Plagiarize. This is mainly because of the $O(n2^n)$ time complexity of Jxplain's *Bimax-Merge* algorithm that is used in partitioning homogeneous objects. Jxplain is designed in a way that first clusters objects, and then iteratively picks the smallest cluster and checks if it is a subset of any other pair of clusters. The number of clusters is anticipated to be small for homogeneous objects. However, when heterogeneous objects are falsely detected as homogeneous objects the number of clusters becomes very big. Each cluster is checked whether it is a subset of every combination of clusters, which results in $O(n2^n)$ time complexity.

## 6 RELATED WORK

*JSON Schema discovery Techniques:* Baazizi et al. [4] proposed a technique, KReduce, made of schema type inference followed by schema type reduction. In the type inference, it first generates the most specific schemas from each JSON document by labeling types of input JSON values such as primitive types, records(i.e., objects), and arrays. Then, they are iteratively merged via the binary fusion operator that returns either a merged (or fused) schema or a union of them. KReduce assumes that the input bag of JSON is from a single schema, record types (or objects) are homogeneous and array types are heterogeneous. We found datasets violating these assumptions. The extended version of this work [5] follows the same principle but introduces two more merge operators: kind-driven and label-driven reductions. The selection of the most suitable merge operator is relegated to the user. The extended version of this work [5] follows the same principle but adds parameters to choose *Reduce* operators that dictate how to fuse two schemas. The *Reduce* operators differ by how strictly they judge the equivalence of two input types. The most lenient *kind-based equivalence* produces the most general and concise schema while the more strict syntactic congruence gives higher precision.

Spoth et al. [42] proposed a top-down algorithm, Jxplain, that addressed shortcomings of KReduce and advanced the state-of-the-art. In determining whether objects or arrays were of homogeneous or heterogeneous type ('tuple' or 'collection' in their terminology), they utilized a threshold based on the key-space entropy concept. (In their terminology, 'tuple' and 'collection' are used for 'homogeneous' and 'heterogeneous', respectively.) To further recognize different object types, Jxplain used a Bimax & GreedyMerge clustering based only on the set of keys. However, Jxplain assumed that all arrays were of heterogeneous types which we found to be violated often in the real-world datasets. Also, we observed the existence of composite types in the real-world dataset, but

**Table 4: Algorithm execution time comparison of ReCG against Jxplain and KReduce.**

| Used data set proportion | ReCG | | Jxplain | | | KReduce | | |
|---|---|---|---|---|---|---|---|---|
| | Avg runtime | Stdev | Avg runtime | Stdev | Relative Performance | Avg runtime | Stdev | Relative Performance |
| 10% | 1852.07 ms | 115.85 | 3125.53 ms | 160.89 | 1.69 | 883.99 ms | 34.89 | 0.48 |
| 50% | 6408.44 ms | 358.49 | 10128.23 ms | 322.14 | 1.58 | 2689.82 ms | 130.62 | 0.42 |
| 100% | 11031.01 ms | 548.53 | 26479.15 ms | 592.52 | 2.40 | 5006.07 ms | 340.87 | 0.45 |

they were not handled in Jxplain. Performance-wise, the Bimax & GreedyMerge clustering has a high impact since it was O($n2^n$). Bimax clusters objects using their set of keys by checking subset relationship. GreedyMerge checks if an element is a subset of any other possible combination of other clusters, and merges them if such a combination is found. The worst case time complexity of Bimax-Merge is O($n2^n$), where each object is found as a cluster in Bimax, and no subset relationship is found in GreedyMerge.

The work by Frozza et al. [23] is another top-down JSD algorithm that has a similar algorithmic structure as KReduce [4, 5]. It first derives a schema for each JSON document by converting values into corresponding types. Then, it aggregates (i.e., collapses) the schemas having identical tree structures and forms RSUS (Raw Schema Unified Structure) which is converted into a final JSON schema. This algorithm also uses a tree-structure to capture the hierarchical structure of JSON documents before generating a schema, but is different from the schema tree we defined. The algorithm lacks handling of objects' optional key fields, heterogeneous objects, and homogeneous array schemas.

These top-down style algorithms suffer from the fundamental problem of limited visibility into lower-level objects. This requires simplifying assumptions or heuristics that are often violated in real-world datasets. ReCG avoids this issue by building up schemas from the lowest level of the JSON document tree hierarchy.

*Schema Discovery for Semi-structured Data Types:* XTRACT [25] addresses the problem of inferring DTD (Document Type Definition) from XML documents in regular expressions. DTD inference corresponds to a single stage processing in the bottom-up JSON schema generation process. XTRACT utilizes MDL to find the best set of regular expressions that expresses the set of XML documents. It also favors a set of regular expressions at the equilibrium point of both conciseness and preciseness of DTDs. The conciseness of a DTD is expected to have negative correlation with the generality of a DTD. Unlike ReCG's partition-then-derive approach, XTRACT takes generalize-factor-MDL approach to build DTDs. Unlike ReCG, XTRACT takes generalize-factor-MDL←need to find better words approach to build DTDs. Bex et al. [8] solves the problem of finding the best regular expression from a set of strings. Their proposed iDReGex learns an automaton that (1) accepts all input strings, and (2) shows the maximum likelihood against the given set of strings. It then strives to translate the automaton to an equivalent regular expression. The best regular expression is chosen using both the MDL cost and how restrictive a regex is. FlashProfile [15] is an algorithm that solves a similar problem of finding a syntactic profile (disjunction of regex-like patterns) from a set of strings that minimizes a cost based on regularization. The main difference with ReCG is that FlashProfile gives a bound for the number of clusters, while ReCG does not. The cost indicates 'the most precise $k$ patterns', which cannot be applied to ReCG since it is unknown how many schemas to derive for each stage. The cost of FlashProfile also regularizes between the conciseness and specificity of the patterns. Its assumption that every symbol within a syntactic profile matches a symbol within a string, cannot be applied to JSON objects (Schema nodes that are typed optional may not match any instance node). Solutions developed for XML data schema discovery are not applicable to JSD because JSON and XML have these fundamental differences. First, XML tags have strict order defined whereas JSON objects do not. Second, XML does not support the concepts of JSON's array and heterogeneous objects.

## 7 CONCLUSION

In this work, we presented a novel bottom-up algorithm for JSON schema discovery problem, called ReCG. We identified the fundamental weakness of existing top-down approaches for the JSD problem that comes from the lack of information about the low-level structure of the JSON documents. We hypothesized that bottom-up processing could avoid such problems and generate more robust and accurate schemas for real-world datasets. Treating JSD as a search problem, our algorithm constructs a large comprehensive set of candidate schemas of varying degrees of generality and selects top $n$ most likely candidates by the MDL criteria. Our evaluation revealed that ReCG outperformed the state-of-the-art competitor by 43% in terms of F1 score and showed 2.40× better performance.

## REFERENCES

[1] Tarfah Alrashed, Jumana Almahmoud, Amy X. Zhang, and David R. Karger. 2020. ScrAPIr: Making Web Data APIs Accessible to End Users. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (, Honolulu, HI, USA,) *(CHI '20)*. ACM, New York, NY, USA, 1–12.

[2] Roam Analytics. 2024. Prescription-based Prediction. https://www.kaggle.com/datasets/roamresearch/prescriptionbasedprediction. Accessed: 2024-01-29.

[3] Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2024. Validation of Modern JSON Schema: Formalization and Complexity. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 1451–1481.

[4] Mohamed Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Schema Inference for Massive JSON Datasets. In *Proceedings of the Conference on Extending Database Technology (EDBT)*. 222–233.

[5] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Schemas and Types for JSON Data: From Theory to Practice. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, New York, NY, USA, 2060–2063.

[6] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2022. Parametric Schema Inference for Massive JSON Datasets. *The VLDB Journal* 28, 4 (mar 2022), 497–521.

[7] Véronique Benzaken, Giuseppe Castagna, Dario Colazzo, and Kim Nguyen. 2006. Type-Based XML Projection.. In *VLDB*, Vol. 6. 271–282.

[8] Geert Jan Bex, Wouter Gelade, Frank Neven, and Stijn Vansummeren. 2010. Learning deterministic regular expressions for the inference of schemas from XML data. *ACM Transactions on the Web (TWEB)* 4, 4 (2010), 1–32.

[9] Kevin S Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. 2011. Jaql:

A scripting language for large scale semistructured data analysis. *Proceedings of the VLDB Endowment* 4, 12 (2011), 1272–1283.

[10] Daniele Bonetta and Matthias Brantner. 2017. FAD.Js: Fast JSON Data Access Using JIT-Based Speculative Optimizations. *Proc. VLDB Endow.* 10, 12 (aug 2017), 1778–1789. https://doi.org/10.14778/3137765.3137782

[11] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoč. 2017. JSON: Data model, Query languages and Schema specification. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (, Chicago, Illinois, USA,) *(PODS '17)*. ACM, New York, NY, USA, 123–135.

[12] Alvaro Cabrera. 2016. *JSON Schema Faker*. Retrieved December 10, 2023 from https://github.com/json-schema-faker/json-schema-faker

[13] Craig Chasseur, Yinan Li, and Jignesh M Patel. 2013. Enabling JSON Document Stores in Relational Systems.. In *WebDB*, Vol. 13. 14–15.

[14] Yelp Dataset. 2024. Yelp Dataset JSON. https://www.yelp.com/dataset/documentation/main. Accessed: 2024-01-29.

[15] Julien Delarue. 2014. Flash profile. *Novel techniques in sensory characterization and consumer profiling* (2014), 175–206.

[16] Alin Deutsch, Lucian Popa, and Val Tannen. 2006. Query reformulation with constraints. *SIGMOD Rec.* 35, 1 (mar 2006), 65–73.

[17] Dominik Durner, Viktor Leis, and Thomas Neumann. 2021. JSON Tiles: Fast Analytics on Semi-Structured Data. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. ACM, New York, NY, USA, 445–458.

[18] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, Vol. 96. 226–231.

[19] Etherscan. 2024. APIs documentation. https://docs.etherscan.io. Accessed: 2024-01-29.

[20] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2006. Rewriting regular XPath queries on XML views. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 666–675.

[21] Daniel H Fishman and J Minker. 1970. *On the number of trees with n terminal nodes*. Technical Report.

[22] VK for developers. 2024. API. https://dev.vk.com/ru/reference. Accessed: 2024-01-29.

[23] 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 *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. 356–363. https://doi.org/10.1109/IRI.2018.00060

[24] Enrico Gallinucci, Matteo Golfarelli, and Stefano Rizzi. 2018. Schema profiling of document-oriented databases. *Information Systems* 75 (2018), 13–25.

[25] Minos Garofalakis, Aristides Gionis, Rajeev Rastogi, Sridhar Seshadri, and Kyuseok Shim. 2000. XTRACT: A system for extracting document type descriptors from XML documents. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 165–176.

[26] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API data fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. ACM, New York, NY, USA, 725–736. https://doi.org/10.1145/3368089.3409719

[27] Thomas Hütter, Nikolaus Augsten, Christoph M. Kirsch, Michael J. Carey, and Chen Li. 2022. JEDI: These Aren't the JSON Documents You're Looking For.... In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. ACM, New York, NY, USA, 1584–1597.

[28] Lubna Irshad, Li Yan, and Zongmin Ma. 2019. Schema-based JSON data stores in relational databases. *Journal of Database Management (JDM)* 30, 3 (2019), 38–70.

[29] Paul Jaccard. 1912. The distribution of the flora in the alpine zone. 1. *New phytologist* 11, 2 (1912), 37–50.

[30] Lin Jiang, Junqiao Qiu, and Zhijia Zhao. 2020. Scalable structural index construction for JSON analytics. *Proc. VLDB Endow.* 14, 4 (dec 2020), 694–707.

[31] Mads Kristensen. 2017. *SchemaStore*. Retrieved December 10, 2023 from https://github.com/SchemaStore/schemastore

[32] Markus Lanthaler and Christian Gütl. 2013. Model your application domain, not your JSON structures. In *Proceedings of the 22nd International Conference on World Wide Web* (Rio de Janeiro, Brazil) *(WWW '13 Companion)*. ACM, New York, NY, USA, 1415–1420.

[33] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: a fast JSON parser for data analytics. *Proc. VLDB Endow.* 10, 10 (jun 2017), 1118–1129. https://doi.org/10.14778/3115404.3115416

[34] Zhen Hua Liu, Beda Hammerschmidt, and Doug McMahon. 2014. JSON data management: supporting schema-less development in RDBMS. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. ACM, New York, NY, USA, 1247–1258.

[35] Jason McHugh and Jennifer Widom. 1999. Query optimization for XML. In *VLDB*, Vol. 99. 315–326.

[36] Fionn Murtagh and Pedro Contreras. 2012. Algorithms for hierarchical clustering: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 2, 1 (2012), 86–97.

[37] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. 2016. Foundations of JSON Schema. In *Proceedings of the 25th International Conference on World Wide Web* (Montréal, Québec, Canada) *(WWW '16)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 263–273.

[38] J. R. Quinlan and R. L. Rivest. 1989. Inferring Decision Trees Using the Minimum Description Length Principle. *Inf. Comput.* 80, 3 (mar 1989), 227–248.

[39] J. Rissanen. 1978. Paper: Modeling by Shortest Data Description. *Automatica* 14, 5 (sep 1978), 465–471. https://doi.org/10.1016/0005-1098(78)90005-5

[40] Filipa Alves dos Santos, Hugo André Coelho Cardoso, João da Cunha Costa, Válter Ferreira Picas Carvalho, and José Carlos Ramalho. 2021. DataGen: JSON/XML Dataset Generator. (2021).

[41] ShowtimesTH. 2024. https://showtimes.everyday.in.th/api/v2/. Accessed: 2024-01-29.

[42] William Spoth, Oliver Kennedy, Ying Lu, Beda Hammerschmidt, and Zhen Hua Liu. 2021. Reducing Ambiguity in Json Schema Discovery. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. ACM, New York, NY, USA, 1732–1744.

[43] Daniel Tahara, Thaddeus Diamond, and Daniel J. Abadi. 2014. Sinew: a SQL system for multi-structured data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. ACM, New York, NY, USA, 815–826.

[44] New York Times. 2024. Article Search. https://developer.nytimes.com/docs/articlesearch-product/1/overview. Accessed: 2024-01-29.

[45] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler. 2018. Albis:{High-Performance} File Format for Big Data Systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 615–630.

[46] Twitter. 2024. Data dictionary: Standard v1.1. https://developer.twitter.com/en/docs/twitter-api/v1/data-dictionary/object-model/tweet. Accessed: 2024-01-29.

[47] Santiago Vargas, Utkarsh Goel, Moritz Steiner, and Aruna Balasubramanian. 2019. Characterizing JSON Traffic Patterns on a CDN. In *Proceedings of the Internet Measurement Conference* (Amsterdam, Netherlands) *(IMC '19)*. ACM, New York, NY, USA, 195–201.

[48] Lanjun Wang, Shuo Zhang, Juwei Shi, Limei Jiao, Oktie Hassanzadeh, Jia Zou, and Chen Wangz. 2015. Schema management for document stores. *Proc. VLDB Endow.* 8, 9 (may 2015), 922–933. https://doi.org/10.14778/2777598.2777601

[49] Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data migration using datalog program synthesis. *Proc. VLDB Endow.* 13, 7 (mar 2020), 1006–1019. https://doi.org/10.14778/3384345.3384350

[50] Wikibase. 2024. JSON. https://doc.wikimedia.org/Wikibase/master/php/docs_topics_json.html. Accessed: 2024-01-29.

[51] Erik Wilde. 2018. Surfing the API Web: Web Concepts. In *Companion Proceedings of the The Web Conference 2018* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 797–803. https://doi.org/10.1145/3184558.3188743

[52] Yi-Pu Wu, Jin-Jiang Guo, and Xue-Jie Zhang. 2007. A linear dbscan algorithm based on lsh. In *2007 International Conference on Machine Learning and Cybernetics*, Vol. 5. IEEE, 2608–2614.

[53] Gongsheng Yuan, Jiaheng Lu, Zhengtong Yan, and Sai Wu. 2023. A Survey on Mapping Semi-Structured Data and Graph Data to Relational Data. *ACM Comput. Surv.* 55, 10, Article 218 (feb 2023), 38 pages. https://doi.org/10.1145/3567444

[54] Qin Yuan, Ye Yuan, Zhenyu Wen, He Wang, and Shiyuan Tang. 2023. An Effective Framework for Enhancing Query Answering in a Heterogeneous Data Lake. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval* (, Taipei, Taiwan,) *(SIGIR '23)*. ACM, New York, NY, USA, 770–780.

[55] Xuan Zhou, Julien Gaugaz, Wolf-Tilo Balke, and Wolfgang Nejdl. 2007. Query relaxation using malleable schemas. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (Beijing, China) *(SIGMOD '07)*. ACM, New York, NY, USA, 545–556.