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

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

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

Wook-Shin Han*
Graduate School of AI
POSTECH, Republic of Korea
wshan@dblab.postech.ac.kr

ABSTRACT

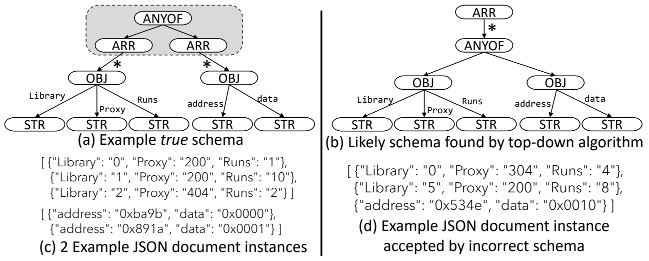
The schemalessness, one of the major advantages of JSON representation format, comes with high penalties in querying and operations by denying various critical functions such as query optimizations, indexing, or data verification. There have been continuous efforts to develop an accurate JSON schema discovery algorithm from a bag of JSON documents. Unfortunately, existing schema discovery techniques, being top-down algorithms, face challenges from the 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 46% better F1 score while also performing 2.11× faster on average against the state-of-the-art.

PVLDB Reference Format:
Joohyung Yun, Byungchul Tak, and Wook-Shin Han. ReCG: Bottom-Up JSON Schema Discovery Using a Repetitive Cluster-and-Generalize Framework. PVLDB, 17(11): 3538 - 3550, 2024.
doi:10.14778/3681954.3682019

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/postechdblab/VLDB2024_ReCG.

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 [2, 23, 30, 44] and big data analytics [5, 10, 16, 27, 31, 40, 41]. 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 [42]. Schemas are essential for many important use cases such as JSON validation [3, 34], JSON parsing acceleration [27, 31], migration of two data sources with different schemas [43], query formulation [15, 48], query optimizations [7, 9, 18, 33], fine-grained access control [42], JSON-to-relational data translation [13, 25, 46], and query answering [47]. 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.



(a) Example *true* schema

(b) Likely schema found by top-down algorithm

```
[ {"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. It may lead to incorrect results (b) due to lack of visibility.

Due to the importance of JSON schemas, the JSON schema discovery (JSD) problem and topics related to the JSON schema formalization have been investigated [3, 4, 6, 11, 16, 21, 34, 38, 42]. Despite many efforts, deriving correct JSON schemas from JSON documents remains difficult 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 [32, 39]. 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 [24]. 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 [38]. 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.

*Corresponding authors

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3682019

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, 38], starts processing from the root and it proceeds down to the subsequent child levels repeatedly. However, this top-down approach faces the following nontrivial challenges. When a new node is encountered, it can be difficult to determine the correct 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), it illustrates one possible case where the top-down algorithm failed 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 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. *Second, 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` [22] 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. *Third, 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. By using this bottom-up style approach, we can expect to reduce the uncertainty in making decisions and the reliance on heuristics.

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 42.57 ∼ 47.80% improved F1 score compared to the state-of-the-art technique. We observed this was due to 39.36 ∼ 46.84% improvements in recall and 15.55 ∼ 23.33% improvements in precision. In terms of the algorithm running time, `ReCG` outperformed the state-of-the-art by 2.11×.

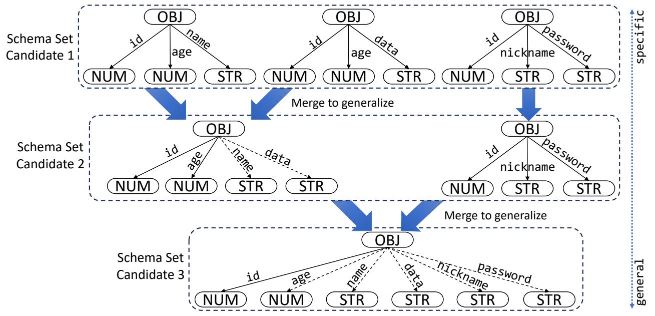


Figure 2: Schema sets with varying generality. More general schema sets are created by merging more specific sets.

J	::=	$P \mid O \mid A$	
P	::=	null | true | false | n | 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	::=	$[\ J_1, \ldots, J_n\ ]$	$(n \geq 0)$

Figure 3: Grammar of JSON documents

We make the following contributions. 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

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. The set of these numbers is denoted as **Number**.
- *String* type JSON document: double-quoted sequence of zero or more Unicode characters. The set of such double-quoted sequences is 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.

2.1.2 JSON Schema. *JSON schema*s are a set of strings in the Backus-Naur form grammar, introduced by Pezoa et al. [34]. 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. [38]. 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*, and the terminals are expressed in `consolas`.

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

Figure 4: BNF grammar of JSON schemas.

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* (look *key* : *JS* at the right-hand side of *prop*). *req* is used to list 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, 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, 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*.

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 and only if one or more of its derivations can validate against j.

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-labeled and edge-labeled trees. Instance trees are extensions from *JSON tree*s introduced by Hutter and Augsten et al. [16]. We define an instance tree as $I = (V_I, E_I, \Psi_I, \Lambda_I, \Gamma_I)$ with the following notations.

- V_I is a set of nodes.
- $E_I \subseteq V_I \times V_I$ is a set of edges.
- $\Psi_I : V_I \rightarrow \{obj, arr, prm\}$ maps each node to its node type.
- $\Lambda_I : V_I \rightarrow$ **Number** $\cup$ **String** $\cup\{$`true`,`false`,`null`$,\epsilon\}$ maps a node to its label. ϵ means an empty label.
- $\Gamma_I : E_I \rightarrow$**String** $\cup\{\epsilon\}$ maps each edge to its edge label.

We transform a JSON document to a JSON tree as follows. Primitive JSON documents each become nodes with the type of *prm*. The nodes are further labeled with the values of JSON documents. Objects become nodes of type obj (labeled as ϵ) with the nodes of its values as children. The edge connecting an object node to its child is labeled with the key corresponding to the value of the child. An array is converted to an *arr* (labeled as ϵ) node with the i-th element becoming the i-th child subtree.

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 being 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.

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

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 ϵ. 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

The 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 [22, 35, 36] 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 number of 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 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:

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

where $\mathcal{Z}$ is a set of schemas. 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. [19], 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 [22]. Let Σ 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.

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

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 $\mathcal{Z}$.

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

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.

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 ($j \in D$), $\mathcal{Z}$ a set of schemas, and $\mathcal{S}$ an element of $\mathcal{Z}$. We adopt the '$\models$' symbol such that $j \models \mathcal{S}$ means j satisfies $\mathcal{S}$ as defined by Pezoa et al. [34] (Also see §2.1.2). We extend '$\models$' to j and $\mathcal{Z}$, where $j \models \mathcal{Z} \iff \exists\, \mathcal{S} \in \mathcal{Z}\ s.t.\ j \models \mathcal{S}$.

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

```
    In $D^+$ := A bag of JSON documents
    In $b$ := beam width
    Out $\mathcal{Z}_{D^+}$ := A discovered set of JSON schemas
 1: $s_{init} \leftarrow D^+$, $s_{goal} \leftarrow \emptyset$, $beam \leftarrow \{s_{init}\}$
 2: while true do
 3:     if $beam$.BEGIN().ISLEAFSTATE() then
 4:         $s_{goal} \leftarrow$ GETLOWEST($beam$, $key = MDLCost$)
 5:         break
 6:     $nextStageStates \leftarrow \emptyset$
 7:     for $s \in beam$ do
 8:         $childrenStates \leftarrow$ GENERATECHILDRENSTATES($s$)
 9:         $nextStageStates \leftarrow nextStageStates \cup childrenStates$
10:     SELECTLOWESTK($nextStageStates$, $k = b$, $key = MDLCost$)
11:     $beam \leftarrow nextStageStates$
12: return GETDERIVEDSCHEMAS($s_{goal}$)
```

We define JSON schema discovery (JSD) problem as the problem of deriving a set of schemas $\mathcal{Z}$ from a given set of JSON documents in such a way that the F1-score is maximized. Let D^+ and D'^+ be two sets of JSON documents that are comprised of documents accepted by the ground truth set of schemas $\mathcal{Z}_G$. Informally we call D^+ and D'^+ positive JSON document sets, and two may not necessarily be the same set. Also, let $\mathcal{Z}_{D^+}$ be a set of schemas derived from D^+. Similarly, we introduce a negative set D^- as a set of JSON documents that are not accepted by $\mathcal{Z}_G$.

$$\begin{aligned} \mathcal{Z} &= \underset{\mathcal{Z}_{D^+}}{\arg\max}\, F1(\mathcal{Z}_{D^+}, D'^+, D^-) \\ &= \underset{\mathcal{Z}_{D^+}}{\arg\max}\, \frac{2 \times Recall(\mathcal{Z}_{D^+}, D'^+) \times Precision(\mathcal{Z}_{D^+}, D'^+, D^-)}{Recall(\mathcal{Z}_{D^+}, D'^+) + Precision(\mathcal{Z}_{D^+}, D'^+, D^-)} \end{aligned} \tag{5}$$

In Equation 5, we define 'Recall' of a given schema set $\mathcal{Z}_{D^+}$ as:

$$Recall(\mathcal{Z}_{D^+}, D'^+) = \frac{|J'|}{|D'^+|} \tag{6}$$

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:

$$Precision(\mathcal{Z}_{D^+}, D'^+, D^-) = \frac{|J'|}{|J''|} \tag{7}$$

where $J'' = \{j | j \in \{D'^+ \cup D^-\}, j \models \mathcal{Z}_{D^+}\}$. 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

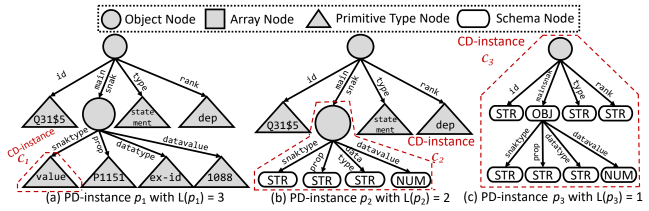


Figure 5: The concepts of PD-instances and CD-instances.

(through our schema 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 5 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 schema node types as shown in Figure 5 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., $L : \mathcal{P} \rightarrow \mathbb{Z}_{\geq 0}$.

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.

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. The stage of a state can also be computed by the following equation:

$$stage(s) = maxHeight(D^+) - L(p') + 1 \qquad (8)$$

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.

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 $childrenStates$
```

4.3 ReCG Algorithm

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 [22, 35, 36] (See §2.3) and its cost valuation technique that gives preference to the one requiring a smaller amount of bits to express the schemas 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. The schemas produced by ReCG must accept all JSON documents in the input dataset. Throughout the ReCG algorithm, we hypothesize that the set of schemas that shows low MDL cost will show a high F1 score.

4.3.1 Algorithm Overview. At the high level, ReCG proceeds by processing CD-instances stage by stage in a bottom-up manner from the lowest level of the JSON document trees. At each level, it converts the instance nodes into schema nodes. In determining the schema node types, ReCG performs clustering of CD-instances to identify homogeneous, composite, and heterogeneous schema node types. These partially-formed schemas constitute one state, and other states are generated by iteratively generalizing this initial set of schemas into incrementally more general schema sets. These states belong to the same *stage*. The MDL cost for each state is calculated using the set of schemas derived so far and ReCG performs a beam search with them. For each selected state from the beam search, it repeats the above steps for the next level to generate subsequent stages until the top level of the input JSON documents is reached. The outline of ReCG is expressed in Algorithm 1. Time complexity analysis of ReCG algorithm is available in the technical report [1]. We refer interested readers to it for detailed analysis.

The complications of the ReCG algorithm mainly arise from the consideration of various candidates in deriving schema. For each stage in the search, ReCG enumerates candidate states that show varying generality in derived schemas and chooses the states that show low MDL cost. This process involves clustering similar CD-instances, and hierarchically merging these clusters to generate sets of schemas that incrementally get more general.

4.3.2 Illustrative Example. Figure 6 shows a schema discovery process of ReCG with three simplified JSON instances j_1^+, j_2^+ and j_3^+. The root node $s_{0,1}$ is initialized with input JSON document instances, or PD-instances with no nodes yet converted to schema nodes. ReCG's bottom-up schema discovery process starts from the nodes of the instance trees at the lowest level. 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

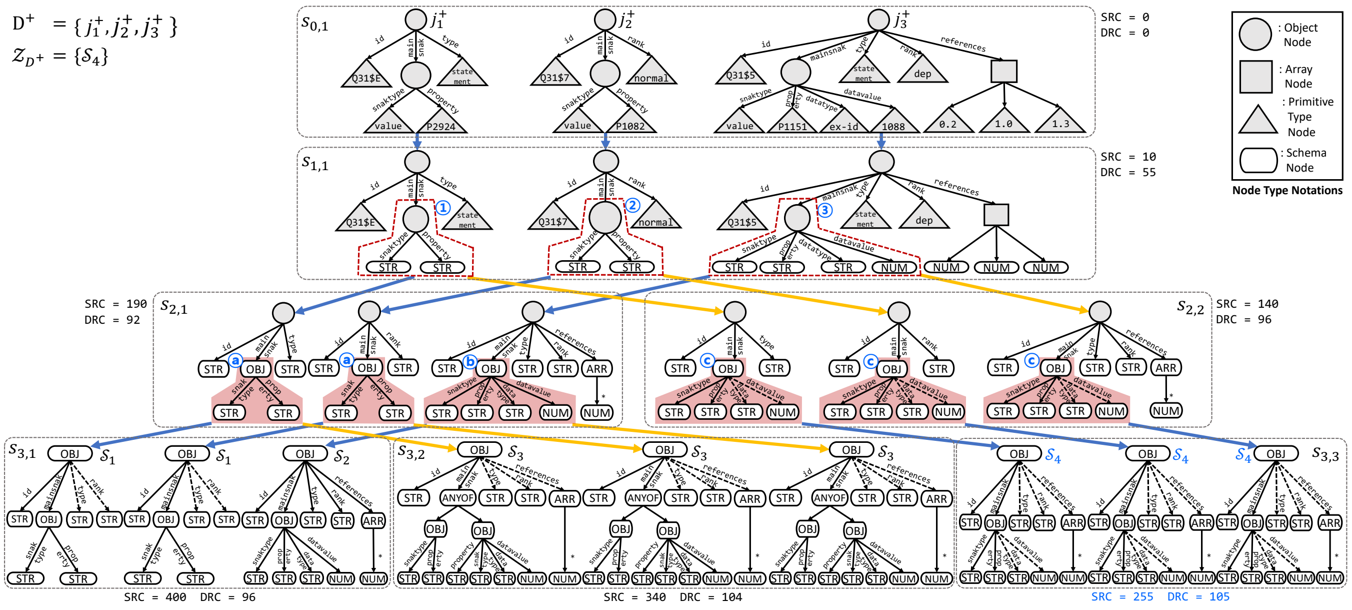


Figure 6: An illustration of ReCG's search space using JSON documents picked from the Wikidata dataset and simplified. The resolution of schema nodes proceeds from the leaf to the root (i.e., bottom-up) as the state transitions from top to bottom.

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.

The MDL cost of each state guides the search for the goal schema. It is calculated by summing the SRC for the schemas derived until that stage, and DRC with the partial instances accepted by the schemas. In Figure 6, $s_{0,1}$ shows an MDL cost of 0 as no schemas are derived yet. State $s_{1,1}$ shows 10 for SRC of derived schemas of STR and NUM, and 55 for DRC to express the primitive documents using these schemas. The goal state would be the leaf state with the least MDL cost, which is $s_{3,3}$ with the smallest MDL cost of 360 in the figure. All inputs j_1^+, j_2^+ and j_3^+ are derived into a single schema $\mathcal{S}_4$, and thus the set of discovered schema becomes $\mathcal{Z}_{D^+} = \{\mathcal{S}_4\}$.

4.4 Schema Node Type Resolution

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. Suppose 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 CD-instances attached to them (e.g., ①, ②, ③ of Figure 6) 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 schemas. On the other hand, there exists the most general 'singleton' schema that accepts 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 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* are given in §4.7.

These steps are expressed in Algorithm 3.

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, only three contained homogeneous array types. Thus, we assume that arrays are heterogeneous by default. To derive array schemas, we generalize (i.e., all labels of edges connected to array elements are turned into '$*$') the arrays. Generalization reduces the array schema to the form of a single '$*$' edge per a unique subschema tree attached to it. 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

Algorithm 3 DeriveCandObjSchemaSets Function Definition

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

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. The CD-instance clustering described here needs to be performed *only once when the first child state is created for a state.*

4.5.1 Distance Measure. To perform effective clustering of CD-instances, we need to define a suitable distance measure. Although CD-instance is a tree, 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 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 [26]. 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:

$$\mathcal{D}(c_1, c_2) := 1 - \frac{|E(c_1) \cap E(c_2)| + |ES(c_1) \cap ES(c_2)|}{2|E(c_1) \cup E(c_2)|} \qquad (9)$$

Let us consider two CD-instances c_5, c_6 in Figure 7 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 [17, 45] 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 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 children schemas. 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 by:

(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 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

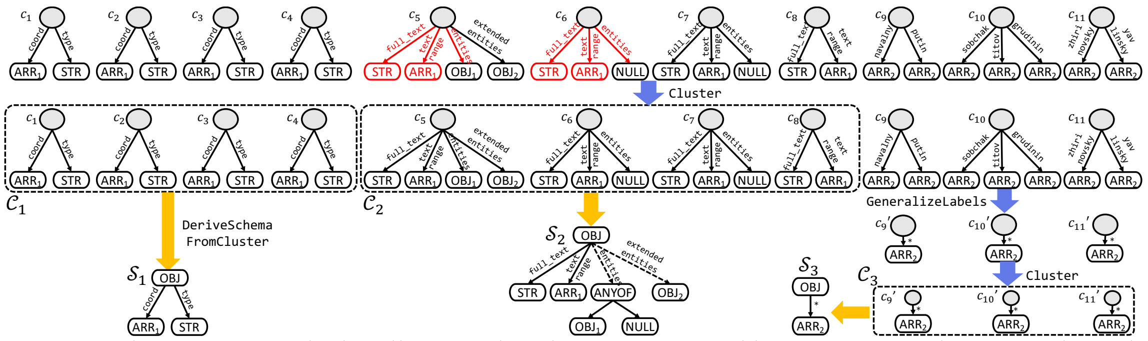


Figure 7: Clustering example that illustrates how homogeneous and heterogeneous schemas are derived.

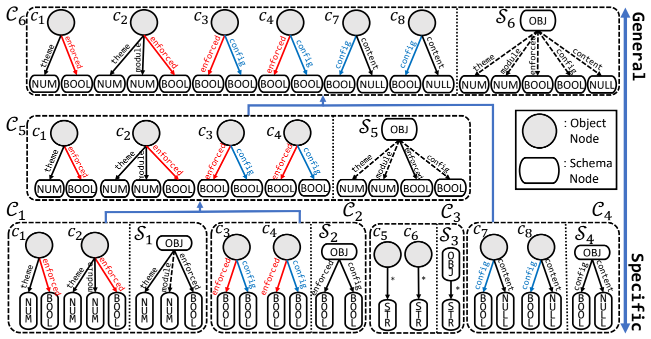


Figure 8: Example of repetitive generalization via merging in the simplified Twitter dataset.

or more distinct schemas exist, we derive another ANYOF node and assign that node to the edge's destination.

4.7 Repetitive Generalization of Schemas

The CD-instance clustering described in previous subsections produces the most specific schema set that ReCG can generate at a state. 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.

4.7.1 Hierarchical Merging. We generate sets of schemas with varying generality through hierarchical clustering where the 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: $c[l]$ points to the child node connected with the edge label l in CD-instance c_i and $E(c)$ is the set of edge labels of c.

$$\begin{aligned} \mathbf{E}^c(C) &:= \bigcup\nolimits_{c \in C} E(c) \\ \mathbf{S}^c(C) &:= \bigcup\nolimits_{c \in C} \bigcup\nolimits_{l \in E(c)} \{c[l]\} \\ \mathbf{T}^c[l](C) &:= \bigcup\nolimits_{c \in C} \{c[l]\} \end{aligned} \tag{10}$$

$\mathbf{E}^c$ 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.

$$\big(\mathbf{E}^c(C_1) \cap \mathbf{E}^c(C_2) - \{*\} \neq \emptyset\big) \wedge \big(\mathbf{T}^c[*](C_1) == \mathbf{T}^c[*](C_2)\big) \tag{11}$$

$$\big(\mathbf{S}^c(C_1) \subseteq \mathbf{T}^c[*](C_2)\big) \vee \big(\mathbf{S}^c(C_2) \subseteq \mathbf{T}^c[*](C_1)\big) \tag{12}$$

$$\mathbf{S}^c(C_1) \cap \mathbf{S}^c(C_2) \neq \emptyset \tag{13}$$

Equation 11 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 12 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 13 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:

$$\mathcal{D}^c(C_1, C_2) := \frac{|SRC(\mathcal{S}_1) + SRC(\mathcal{S}_2) - SRC(\mathcal{S}_m)|}{SRC(\mathcal{S}_m)} \tag{14}$$

Figure 8 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_4). 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_4, C_5) is viable for merge. They are merged to C_6. (C_3, C_6) is not a viable pair of merge, and thus the merging phase ends.

4.8 Limitations of ReCG

We identify a few current limitations of ReCG to use as a guide for improvements. First, ReCG may incorrectly classify JSON document instances from homogeneous schemas as outliers and turn them into generalized heterogeneous object schemas if the number of instances is below the level our clustering algorithm can group. Second, if excessively many clusters are formed at a stage, the performance can be significantly impacted due to high clustering and merging overheads.

Table 1: Statistics of 20 datasets used in experiments.

Dataset		Schema								Instance	
Type	Name	Height	$\|V_S\|$	# of OBJ nodes			# of ARR nodes		# of ANYOF nodes	$\|D^+\|$	avg($\|V_I\|$)
				Hom Obj	Het Obj	Com Obj	Hom Arr	Het Arr			
Real-life	NYT	6	92	9	0	0	0	3	14	10k	85.21
	Twitter	∞	∞	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 EVALUATION

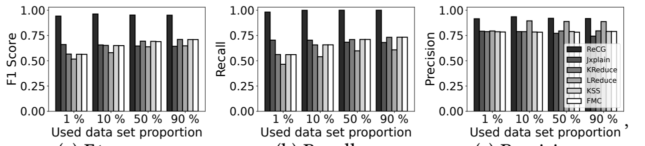
In this section, we present evaluation results about the accuracy, validity of MDL costs, scalability with dataset sizes, sensitivity of parameters, and the impact of design factors.

5.1 Experimental Setup

5.1.1 **Compared Techniques.** `ReCG` is compared against `five` existing techniques - `Jxplain`, `KReduce`, `LReduce`, `KSS` and `FMC`.

- `Jxplain` [38]: 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]: 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. `KReduce` is expected to be fast but also shows a low F1 score on datasets that do not conform to its assumptions.
- `LReduce` [6]: A variant of `KReduce` with the additional assumption that objects generated from different schemas have different edge labels. It tends to produce a more specific schema than `KReduce`.
- `KSS` [28]: A top-down style algorithm proposed by Klettke et al. and we refer to it as `KSS` for convenience in this work. Its assumptions are identical to those of `KReduce`. It builds a single schema tree by iteratively visiting each JSON document.
- `FMC` [20]: A top-down algorithm that also makes the same assumptions as `KReduce`. It first collapses identical schemas, and they are merged to build a general schema that can accept every JSON document accepted by the collapsed schemas.

5.1.2 **Hardware and Software Settings.** We conducted our experiments on a machine with an Intel Xeon E5-2680 v4 @2.40GHz CPU and 756 GB of RAM with the OS of Ubuntu 20.04. `ReCG` was implemented with C++ and was compiled with gcc 8.3.0. `Jxplain` [4], `KReduce` and `LReduce` [6] were implemented in Scala. `KSS` [28] and `FMC` [20] were written in C++.



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

Figure 9: Accuracy comparison with varying dataset sizes.

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.

Synthetic datasets are generated from schemas obtained from `JSON schema store` [29], 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` [37] and `json-schema-faker` [12].

Various characteristics of the datasets are presented in Table 1. It shows the characteristics of the ground truth schema set $\mathcal{Z}_G$, and its corresponding positive document set D^+ for each dataset. For $\mathcal{Z}_G$, we report the maximum height, 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 for instances.

We generated a negative document set D^- for each dataset from the ground truth schema set $\mathcal{Z}_G$. The size of D^- was made equal to the size of D^+ for each dataset. The generation process of a negative document set D^- followed the steps described below:

(1) Modification of $\mathcal{Z}_G$ into $\mathcal{Z}_G^-$, a set of schemas that can accept instances that $\mathcal{Z}_G$ rejects.
(2) Generation of synthetic document j^- from $\mathcal{Z}_G^-$.
(3) Validation of instance j^- against $\mathcal{Z}_G$. If it is not accepted by $\mathcal{Z}_G$, add it to D^-.
(4) Repeat (1) ~ (3) until $|D^-|$ becomes equal to $|D^+|$.

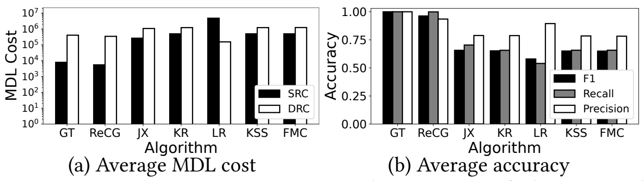
Generating $\mathcal{Z}_G^-$*:* We elaborate on the modification process of $\mathcal{Z}_G$ into schema $\mathcal{Z}_G^-$. The objective is to generate a negative schema set that could pose the highest difficulty for any 'schema under test' in accurately rejecting the negative JSON document instances generated from this. Since the heavier the modification to $\mathcal{Z}_G$, the more likely to be easy for any schema to reject the negative JSON documents, we decided to apply modification operation only once to a single node in $\mathcal{Z}_G$ in each modification. Thus, we randomly picked a target schema node v_S in $\mathcal{Z}_G$ and a modification operation from the set of possible operations predefined by the type of v_S.

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 the experiment of `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.

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

Dataset	ReCG			Jxplain			KReduce			LReduce			KSS			FMC		
	Recall	Precision	F1	Recall	Precision	F1	Recall	Precision	F1	Recall	Precision	F1	Recall	Precision	F1	Recall	Precision	F1
NYT	1.00	1.00	1.00	Runtime Error			1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	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	0.90	1.00	0.95	1.00	1.00	1.00	1.00	1.00	1.00
Github	1.00	1.00	1.00	0.48	1.00	0.64	1.00	1.00	1.00	0.98	1.00	0.99	1.00	1.00	1.00	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	0.27	1.00	0.43	0.86	1.00	0.92	0.86	1.00	0.92
Wikidata	1.00	1.00	1.00	Time Out			0.47	1.00	0.64	0.00	1.00	0.01	0.47	1.00	0.64	0.47	1.00	0.64
Yelp	1.00	0.70	0.82	0.97	0.77	0.86	1.00	0.36	0.53	0.92	1.00	0.96	1.00	0.36	0.53	1.00	0.36	0.53
VK	1.00	0.99	1.00	0.99	1.00	0.99	0.99	1.00	1.00	0.97	1.00	0.98	1.00	1.00	1.00	1.00	1.00	1.00
Iceberg	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	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	0.60	1.00	0.75	0.81	1.00	0.90	0.81	1.00	0.90
ETH	1.00	1.00	1.00	0.87	0.34	0.49	1.00	0.33	0.49	1.00	0.50	0.67	1.00	0.31	0.47	1.00	0.31	0.47
GeoJSON	1.00	0.96	0.98	1.00	0.71	0.83	1.00	0.42	0.59	1.00	0.56	0.72	1.00	0.42	0.59	1.00	0.42	0.59
ThaiMovies	0.99	0.99	0.99	0.85	1.00	0.92	0.99	0.95	0.97	0.99	1.00	1.00	0.99	0.92	0.95	0.99	0.92	0.95
RDB	1.00	0.79	0.88	Time Out			0.53	0.85	0.65	0.35	0.86	0.50	0.53	0.85	0.65	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	0.22	0.97	0.36	0.34	0.93	0.50	0.34	0.93	0.50
HelmChart	1.00	0.95	0.97	1.00	0.43	0.61	0.57	1.00	0.72	0.33	1.00	0.49	0.57	1.00	0.72	0.57	1.00	0.72
Dolittle	1.00	1.00	1.00	0.33	1.00	0.50	0.67	0.79	0.72	0.67	1.00	0.80	0.67	0.79	0.72	0.67	0.75	0.71
Drupal	1.00	0.93	0.97	0.06	0.82	0.12	0.01	0.12	0.01	0.00	1.00	0.01	0.01	0.12	0.01	0.01	0.12	0.01
DeinConfig	1.00	0.92	0.96	1.00	0.93	0.96	0.33	1.00	0.50	0.16	1.00	0.27	0.33	1.00	0.50	0.33	1.00	0.50
Ecosystem	1.00	1.00	1.00	Runtime Error			0.28	1.00	0.44	0.26	1.00	0.41	0.28	1.00	0.44	0.28	1.00	0.44
Plagiarize	1.00	0.72	0.84	Time Out			0.31	1.00	0.47	0.18	1.00	0.31	0.31	1.00	0.47	0.31	1.00	0.47



(a) Average MDL cost (b) Average accuracy

Figure 10: Average MDL cost and accuracy of the ground truth schema, and the schemas found by six algorithms for 20 datasets. The proportion of the used dataset is 10%. (GT: ground truth, JX: Jxplain, KR: KReduce, LR: LReduce)

Figure 9 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, ϵ=0.5, *minPts*=5%, and sampleSize=500. Overall, ReCG gave superior F1 scores against competitors irrespective of the data set proportions. F1 score of ReCG was higher than Jxplain, KReduce and LReduce by 46%, 45%, 60%, 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 9 (b) and (c).

According to our investigation, the lower F1 score of Jxplain, KReduce and LReduce 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 type of heterogeneous object schema. In the cases of KReduce and LReduce, they do not support heterogeneous object schemas. As a result, Jxplain, KReduce and LReduce all show low precision on ETH, GeoJSON, and Drupal (only for Jxplain) 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 more general than the ground truth schema and thus shows low precision. *Second, it is the inability to correctly derive a schema node.* Jxplain, KReduce, and LReduce 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. This causes these algorithms to show low recall in Iceberg, Wikidata datasets. Also, KReduce and LReduce show low recall on Drupal for this reason. *Third,* Jxplain*,* KReduce *and* LReduce *do not handle composite object schemas.* They 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. Also for LReduce, it partitions the objects using the set of edge labels of objects and derives a homogeneous schema from each partition. This results in a more specific schema than the other two. *Fourth,* Jxplain*,* KReduce *and* LReduce *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. LReduce and Jxplain only assume ANYOF schema nodes that have two or more homogeneous object schema nodes as children. KReduce finds a single homogeneous object schema, and both LReduce and Jxplain partition the objects into multiple homogeneous object schemas, resulting in low recall.

5.3 MDL Cost Analysis

We compared and analyzed MDL costs of the schemas produced by ReCG and competitors to observe the quality of produced schemas. The measurements were made on all 20 datasets and the input J_{input} was comprised of 10% of D^+. Figure 10 shows the average of MDL costs and accuracy measures. The numbers from the ground truth schema are also plotted together for reference.

Figure 10 (a) shows that ReCG obtained the least average MDL cost. ReCG's MDL cost came out to be 3.8×, 4.9× and 14.2× smaller than Jxplain, KReduce, and LReduce, respectively, while maintaining high F1 score of 0.95. Jxplain, KReduce and LReduce received about 29.5%, 31.1% and 37.4% less F1 score than ReCG, respectively. Among the components of MDL costs, the SRC of ReCG were superior – 48.26×, 91.08× and 879.53× smaller than Jxplain, KReduce and LReduce, respectively. This indicates that the schema discovered by ReCG is 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 15.9% 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.

We found that the MDL costs had a strong negative correlation with F1 scores. The correlation came out to be a strong negative correlation of -0.77 between MDL and F1 score. This was contributed the most by the strong negativity with *Recall* (-.82) rather than with *Precision* (-.41). 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.

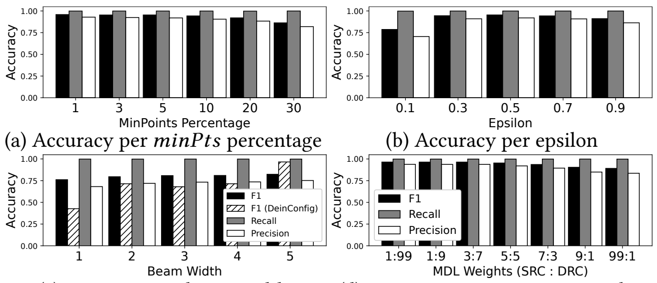
5.4 Scalability with Dataset Size

We compared the scalability of techniques by varying data sizes of 10%, 50%, and 100%. Table 3 shows the algorithm execution time measurements. Overall, ReCG outperformed Jxplain by a significant margin (1.54× to 2.11×) on average but KReduce was faster than ReCG by about 121%. 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. This is because Jxplain determines objects' heterogeneity and performs clustering of objects per a distinct labeled path of edges in input JSON documents. Determination of heterogeneity and clustering of objects are the main operations of Jxplain, which comprise 93.6% of total runtime on average. In addition, the number of distinct labeled path of edges are related to the total number of distinct keys. The correlation between the number of distinct keys per dataset and the speedup of ReCG compared to Jxplain was as high as 0.83. Specifically, Jxplain performed faster than ReCG in datasets of ETH which showed a small distinct number of keys of 54. On the other hand, ReCG outperformed Jxplain for the rest of the datasets, which had 7842 distinct number of keys on average.

KReduce, KSS and FMC showed faster execution than ReCG. This is because ReCG spends time in performing clustering and merging to derive various sets of schemas, which incurs the main overhead. LReduce showed smaller execution time than ReCG on 10% and 50% of the datasets, but similar runtime to ReCG on 100% datasets.

Jxplain exhibited exponential growth in runtime for datasets of Dolittle and Drupal. It also 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.



(a) Accuracy per *minPts* percentage (b) Accuracy per epsilon
(c) Accuracy per beam width (d) Accuracy per MDL cost weights

Figure 11: Parameter sensitivity to accuracy.

5.5 Parameter Sensitivity

5.5.1 DBSCAN Parameter: minPts. *minPts* is a parameter of DBSCAN that specifies the minimum number of neighbors within ϵ distance for a point to be considered in a cluster. Figure 11 (a) shows the accuracy change (average of 20 datasets) as we vary the *minPts* from 1% up to 30% of the total number of points to be clustered. The ϵ (See §5.5.2) and beam width were set to 0.5 and 3, respectively.

5.5.2 DBSCAN Parameter: Epsilon. It is another threshold parameter of DBSCAN that controls the radius within which the points are considered ϵ-neighberhood. Figure 11 (b) shows the average accuracy from all 20 datasets with *minPts* of 5% and beam width of 3. There existed a nonlinear relationship between ϵ and accuracies of the discovered schema with the highest accuracy at ϵ = 0.5.

5.5.3 Beam Width. We measured the effect of various beam widths on the accuracy and the average results over 20 datasets are shown in Figure 11 (c) under ϵ=0.1 and *minPts*=1%. The F1 score improvement was 8.13% as the beam width increased from 1 to 5. A similar degree of accuracy improvements was observed in several of our datasets including Twitter, Github, Yelp, VK, AdonisRC, HelmChart, DeinConfig.

5.5.4 MDL Cost Weights. We examined the effect of varying weights of *SRC* and *DRC* in computing the MDL cost as in $MDLCost(\mathcal{Z}, D) = \alpha SRC(\mathcal{Z}) + \beta DRC(\mathcal{Z}, D)$ where $\alpha + \beta = 1$. We set 7 different ratios for $\alpha : \beta$ and measured the accuracies of the discovered schemas as in Figure 11 (d). The value of DRC is roughly two orders of magnitude larger than SRC at 5:5 (See Figure 10 (a)). Thus, increasing DRC's weight beyond 5:5 ratio does not affect the overall accuracy much. However, larger weights on SRC affect the precision. At the ratio of 99:1 where SRC values grow to be comparable to DRC in magnitude, the precision and F1 score decrease to 9.2% and 6.5%, respectively, compared to the case with 5:5 ratio.

5.6 Impact of Design Factors to Accuracy

We compared the contribution of two key components of ReCG: bottom-up schema generation and MDL cost model for guiding the search. We implemented two versions of ReCG. In one version, ReCG is modified to use Jxplain's key-space entropy [38] to guide its search instead of our MDL cost model. The key-space entropy is

Table 3: Algorithm execution time comparison of ReCG against competitors. The third subcolumn under each algorithm is the relative speedup of each competitor against ReCG.

Used dataset proportion	ReCG			Jxplain			KReduce			LReduce			KSS			FMC		
	Avg runtime	Stdev	$\frac{\text{ReCG}}{\text{ReCG}}$	Avg runtime	Stdev	$\frac{\text{ReCG}}{\text{Jxplain}}$	Avg runtime	Stdev	$\frac{\text{ReCG}}{\text{KReduce}}$	Avg runtime	Stdev	$\frac{\text{ReCG}}{\text{LReduce}}$	Avg runtime	Stdev	$\frac{\text{ReCG}}{\text{KSS}}$	Avg runtime	Stdev	$\frac{\text{ReCG}}{\text{FMC}}$
10%	1489.62 ms	116.30	1.00	3130.67 ms	206.18	0.48	883.99 ms	34.89	1.69	1030.81 ms	58.16	1.45	168.32 ms	12.80	8.85	267.74 ms	73.37	5.56
50%	6571.55 ms	547.28	1.00	10128.23 ms	429.52	0.65	2689.82 ms	130.62	2.44	4892.20 ms	333.83	1.34	856.21 ms	45.51	7.68	1776.20 ms	557.19	3.70
100%	12571.91 ms	448.65	1.00	26479.15 ms	790.02	0.47	5006.07 ms	340.87	2.51	12603.57 ms	761.72	1.00	1748.04 ms	81.56	7.19	4959.49 ms	1644.84	2.53

Table 4: Impact of MDL cost model and bottom-up style to the overall accuracy of ReCG.

Method	Recall	Precision	F1
ReCG (Key-space entropy as cost model)	1.00	0.83	0.89
ReCG (Top-down schema generation)	1.00	0.88	0.92
ReCG	1.00	0.92	0.95

Jxplain's function to determine the heterogeneity of objects. In the second version, we modified ReCG to apply *top-down* schema generation. It starts to derive the schema trees from their roots by transforming the instance nodes into schema nodes. In determining the child schema types top-down, it heuristically sets the schema node type to its instance type.

As in Table 4, both versions reported lowered precisions and F1 scores. Between these, the impact of MDL cost model was larger. The version with key-space entropy cost model showed 10.6% and 6.2% drop in precision and F1, respectively. This was because the key-space entropy cost model was less effective as a search guide.

6 RELATED WORK

JSON Schema discovery Techniques: Baazizi et al. [4] proposed KReduce, made of schema type inference followed by schema fusion. KReduce assumes the input bag of JSON documents is from a single schema, record types (or objects) are homogeneous and array types are heterogeneous. LReduce [6] is another algorithm that follows the same principle as KReduce, but object schemas are fused only if two nodes have the same set of edge labels. Thus, LReduce can discover ANYOF nodes with multiple object schemas as children, but two children always have different sets of edge labels.

Spoth et al. [38] 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. To further recognize different object types, Jxplain used a Bimax & GreedyMerge clustering based only on the set of keys. However, it did not assume the presence of composite object schemas and also did not cluster heterogeneous objects.

Klettke et al. proposed a top-down JSD discovery algorithm [28]. It linearly iterates the input set of JSON documents and updates a single schema that can accept all the instances seen. It updates the schema by adding schema nodes from the top to the bottom.

Frozza et al. is another top-down JSD algorithm that has a similar structure as KReduce. 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. The algorithm lacks handling of heterogeneous objects and homogeneous array schemas.

Schema Discovery for Semi-structured Data Types: XTRACT [22] addresses the problem of inferring DTD (Document Type Definition) from XML documents in regular expressions. XTRACT utilizes the MDL principle 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 DTD inference problem in XML leverages the MDL principle for measuring regular expressions for a single element, as an element's DTD is completely independent of the DTDs for other elements [22]. However, in JSON schema discovery, we must infer tree-structured schemas with arbitrary heights as a whole. Additionally, our MDL costing must account for the homogeneity and heterogeneity of object and array schemas.

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 [14] 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. 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).

7 CONCLUSION

We presented a novel bottom-up algorithm for the JSON schema discovery problem, called ReCG. We hypothesized that bottom-up processing could avoid problems of top-down approach 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 the most likely candidate by the MDL criteria. Our evaluation revealed that ReCG outperformed the state-of-the-art competitor by 46% in terms of F1 score and showed 2.11× better performance.

ACKNOWLEDGMENTS

We appreciate Dr. Divesh Srivastava's comment on the need for experimental justification based on the MDL cost. This work was partly supported by the National Research Foundation of Korea grant funded by the Korea government (MSIT) (No. NRF-2021R1A2 B5B03001551 and No. NRF-2021R1A5A1021944) and Digital Innovation Hub project supervised by the Daegu Digital Innovation Promotion (DIP) Agency grant funded by the Korea government (MSIT and Daegu Metropolitan City) in 2024 (No.DBSD1-08).

REFERENCES

[1] 2024. Technical Report. Retrieved July 15, 2024 from https://sites.google.com/dblab.postech.ac.kr/recg-technical-report

[2] 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.

[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 July 15, 2024 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] Julien Delarue. 2014. Flash profile. *Novel techniques in sensory characterization and consumer profiling* (2014), 175–206.

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

[16] 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.

[17] 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.

[18] 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.

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

[20] 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)*. IEEE, 356–363.

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

[22] 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.

[23] 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

[24] 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.

[25] 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.

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

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

[28] Meike Klettke, Uta Störl, and Stefanie Scherzinger. 2015. Schema extraction and structural outlier detection for JSON-based NoSQL data stores. (2015).

[29] Mads Kristensen. 2017. *SchemaStore.* Retrieved July 15, 2024 from https://github.com/SchemaStore/schemastore

[30] 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.

[31] 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

[32] 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.

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

[34] 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.

[35] 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.

[36] 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

[37] 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).

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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

[43] 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

[44] 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

[45] 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.

[46] 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

[47] 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.

[48] 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.