# Literature Reviews on Reducing Ambiguity in JSON Schema

CS 520: Data Integration, Warehousing, & Provenance

Pradeep Raj Thapaliya — Shubham Tiwari — Somanshu Gupta

## Abstract

*JSON data modelling binds the various data sources with single stream, when writing the data, JSON flexibility relies on automated schema discovery techniques. Ambiguity in the schema design forces existing schema discovery systems to make data independent assumptions about schema structure. When these assumptions are violated, generated schema are imprecise. JXPLAIN algorithm is implemented, JSON schema discovery algorithm with heuristics that creates a replica of common forms of ambiguity and produces significantly accurate schema.*

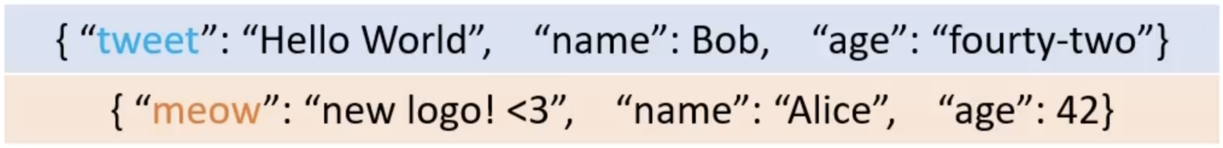
## Contents

## 1 Introduction

JSON schema are JSON documents that describe other JSON documents. It is used to define the structure of a JSON message and validate it. Example: - API request and response in JSON format.

### Why JSON Datasets?

1. Easy Data Provider
2. Schemeless
3. Structured
4. Write Friendly

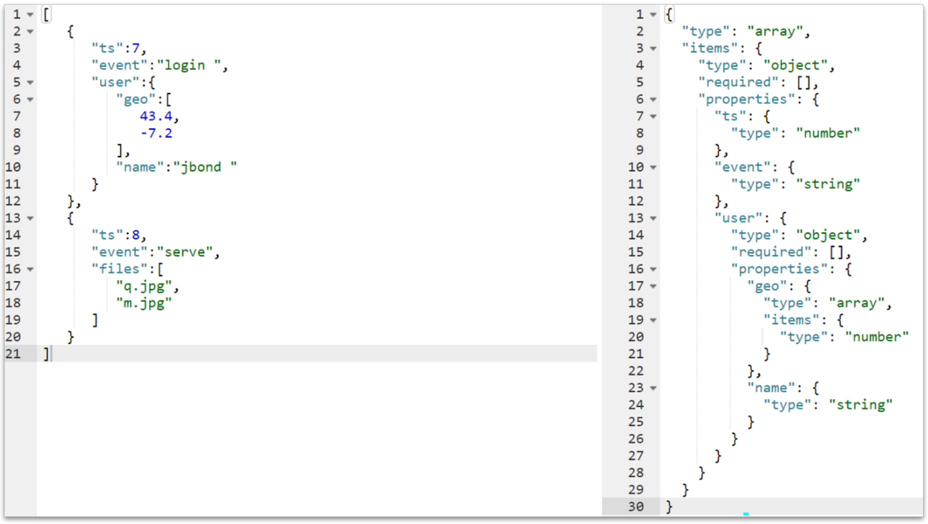


```
{ “tweet”: “Hello World”,   “name”: Bob,   “age”: “fourty-two”}
{ “meow”: “new logo! <3”,   “name”: “Alice”,   “age”: 42}
```

JSON is very easy to handle because We don’t need any migration script or schema update, its just changing the string. For Example, Twitter Re-branding.

### JSON Schema : -

1. Schema is self defining.
2. Every individual row is a schema.
3. It tells the types and columns it contains and encapsulates large grouping of related fields



```
[
    {
        "ts":7,
        "event":"login ",
        "user":{
            "geo":[
                43.4,
                -7.2
            ],
            "name":"jbond "
        }
    },
    {
        "ts":8,
        "event":"serve",
        "files":[
            "q.jpg",
            "m.jpg"
        ]
    }
]
```

```
{
  "type": "array",
  "items": {
    "type": "object",
    "required": [],
    "properties": {
      "ts": {
        "type": "number"
      },
      "event": {
        "type": "string"
      },
      "user": {
        "type": "object",
        "required": [],
        "properties": {
          "geo": {
            "type": "array",
            "items": {
              "type": "number"
            }
          },
          "name": {
            "type": "string"
          }
        }
      }
    }
  }
}
```

Figure 1: Example JSON with their Schema

## 1.1 Existing Schema Discovery Approach to Resolve Ambiguity

Consider two below JSON records to understand the method to resolve ambiguity.

**Listing 1** JSON example

```
{
    "ts": 7,
    "event": "login",
    "user": {
        "geo" : [43.4, -7.2],
        "name": "jbond"
    }
}
{
    "ts": 8,
    "event": "serve",
    "files": ["q.jpg"," m.jpg"]
}
```

Schema discovery system assume that objects in a collection are instances of single entity. This will give lead us to two assumptions: -

1. **Assumption - 1** : All records in a data must have an "integer" ts field and "string" event field.
2. **Assumption - 2** : Variation between records is exclusively caused by optional fields. But record cannot simultaneously be a login and serve event.

After above two assumptions, proposed schema will encourage any of the following " records (See listing 2).

**Listing 2** " Mappings

```
{
    "ts": 9,
    "event": "xyz",
    "user": {....},
    "files": [...]
}
{
    "ts": 10,
    "event": "wat",
}
```

As "user" field and "files" field are independently optional, proposed schema may accept any of the below

1. Records with both fields
2. Records with none of them

Existing approaches resolve the ambiguity through data independent heuristics and by assuming that data agrees with three informal conventions.

1. Collection contains single entity type.
2. JSON arrays always encode collections.
3. JSON objects always encode tuples.

These above rules are sufficient for homogenous JSON collections but lack vital information for complex nesting structures like in heterogenous collections and data sources like web service API or JSON formatted system logs.

## 1.2 JXPLAIN Schema Discovery System Approach

This discovery system resolves schema ambiguity on a per-instance basis. We will have below contributions in this approach.

1. Identify forms of schema ambiguity that cause existing JSON schema discovery techniques to produce low-precision schema.
2. JXPLAIN, a framework for heuristically resolving ambiguity
3. Feasibility of JXPLAIN by proposing specific heuristics for detecting and resolving these forms of ambiguity.
4. Create schema with higher precision, considering negligible change in recall.

# 2 Background

The goal is to re-construct a hidden ground truth schema which is a description of set of valid JSON records from a finite collection of records sampled from set. the rules are : -

1. An ideal algorithm produces a generated schema with high recall. All records should be a part of generated schema irrespective of their appearance in sample.
2. Algorithm should produce a schema with higher precision and concise description.
3. Records not in the ground truth schema should not be a part of generated schema.

## 2.1 Notation

Let's try to understand below equation

$$\tau := B|R|S|null|[\tau_1.........\tau_N]|k_1 : \tau_1.........k_N : \tau_N$$

Data values in JSON are primitive or complex.

### 2.1.1 Primitive Values:

B = Boolean Value, R = Numerical Value, S = String Value, null - Null value

### 2.1.2 Complex Values:

1. JSON array : $[\tau_1, ..., \tau_N]$, this collection is a sequence of N values with types $\tau_1, \tau_2, ..., \tau_N$
2. JSON Collection Mappings: $\{k_1 : \tau_1, ..., k_N : \tau_N\}$, this is a mapping of keys to values where $k_1, k_2 \ldots k_N$ are the keys to values and $\tau_1, \tau_2, ...\tau_N$ are types of values.

$$\mathsf{kind}(\tau) = \begin{cases} \tau & \textbf{if } \tau \in \{\, \mathbb{B}, \mathbb{R}, \mathbb{S}, \textbf{null} \,\} \\ O & \textbf{if } \tau = \{\, k_1 : \tau_1, \ldots, k_N : \tau_N \,\} \\ \mathcal{A} & \textbf{if } \tau = [\, \tau_1, \ldots, \tau_N \,] \end{cases}$$

### 2.1.3 Example

In below JSON, we can write type of value as below

**Listing 3** ” Mappings

```
    {
     "ts" : R,
     "event" : S,
     "user" : {
         "name" : S,
         "geo" : [R R]
     }
    }
```

Note: - Kind of Record is O. Field event Kind: - S

### 2.1.4 Definition

A Schema S is a set of types $\tau$ belongs to s. In other words, $\tau$ is admitted by schema if it is an element of set. Singleton schema is represented by $\{\tau\}$.

### 2.1.5 Nested Schema Notations

1. Optional Fields
   Here question mark is added to a field name to mark it is optional. Resulting schema accepts object types with any subset of the fields marked optional.

$$\left\{\, k_1 : \tau_1, \ldots, k_n : \tau_n, k_1'^{?} : \tau_1', \ldots, k_m'^{?} : \tau_m' \,\right\} \triangleq$$
$$\left\{\, \left\{\, k_1 : \tau_1, \ldots, k_n : \tau_n, k_{\ell_1}' : \tau_{\ell_1}', \ldots, k_{\ell_p}' : \tau_{\ell_p}' \,\right\} \,\middle|\, \{\, \ell_1, \ldots, \ell_p \,\} \subseteq [m] \,\right\}$$

2. Schema Nesting
   { k : S,. . . ..} to denote the schema admitting like kind types with corresponding field types admitted by the schema S.

$$\{\ k_1 : \mathcal{S}_1, \ldots, k_N : \mathcal{S}_N\ \} \triangleq \{\ \{\ k_1 : \tau_1, \ldots, k_N : \tau_N\ \}\ |\ \tau_i \in \mathcal{S}_i\ \}$$
$$[\ \mathcal{S}_1, \ldots, \mathcal{S}_N\ ] \triangleq \{\ \{\ \tau_1, \ldots, \tau_N\ \}\ |\ \tau_i \in \mathcal{S}_i\ \}$$

3. Collection Types
   { *: S }* to denote collection schema that admits any object kinded type who's field types are drawn from S.

$$\{\ * : \mathcal{S}\ \}^* \triangleq \{\ \{\ k_1 : \tau_1, \ldots, k_N : \tau_N\ \}\ |\ N \in \mathbb{N}^0 \wedge \tau_1, \ldots, \tau_N \in \mathcal{S}\ \}$$
$$[\ \mathcal{S}\ ]^* \triangleq \{\ [\ \tau_1, \ldots, \tau_N\ ]\ |\ N \in \mathbb{N}^0 \wedge \tau_1, \ldots, \tau_N \in \mathcal{S}\ \}$$

## 2.2 Schema Discovery

Given a collection of records with N types $R = \{\tau_1, ..\tau_N\}$ drawn from some hidden ground truth schema SG. Schema discovery problem is to merge these types into a new schema denoted as merge(R). Also derived schema should have high precision, admitting only ground types and high recall, compact representation, avoiding explicit type enumeration.

**Ground truth types:** - $|\text{merge(R)} - S_G| \,/\, |S_G \cup \text{merge(R)}| \approx 0$

OR

**Ground truth types:** - $|S_G - \text{merge(R)}| \,/\, |S_G \cup \text{merge(R)}| \approx 0$

### 2.2.1 Naïve Discovery

Sample records to be the definitive set of types admitted by SG. It returns a set of two distinct schemas.
**Definition:** $\text{merge}_{\text{naive}}(\mathcal{R}) \triangleq \{\ \tau_1, \ldots, \tau_N\ \}$
**Guarantees:** High Precision.
**Missing:** Reject types missing from input (low recall), no compactness.
**Example :**

### 2.2.2 Array as Collections

Recursively applying schema discovery to the union of the array's elements.
**Definition:** $\text{merge}_{\mathcal{A}}(\mathcal{R}) \triangleq [\ \text{merge}\ (\{\ \tau_i\ |\ [\ \tau_1, \ldots, \tau_N\ ] \in \mathcal{R}, i \in [N]\ \})\ ]^*$
**Example :**
Field "files" would be merged into a collection of strings: [ S] *, as all of its elements have kind S.

### 2.2.3 Objects as Tuples

Variation between objects in a collection is assumed to be the result of optional fields. Fields appearing in all input objects are mandatory with keys ($keys_{\exists}(R)$), fields are appearing in

**Listing 4** Naive Example

```
{
    {
    "ts" : S,
    "event" : S,
    "user":
        {
            "geo" : [R, R],
            "name" : S
        }
    },
    {
    "ts" : S,
    "event" : S,
    "files" : [S, S]
    }
}
```

**Listing 5** Array As a Collection

```
{
    "ts": 8,
    "event": "serve",
    "files": ["q.jpg"," m.jpg"]
}
```

only some are optional (with keys ($keys_{\forall}(R)$).

### 2.2.4 Standard Discovery

It uses naïve merge for primitive types and recursively merges array and objects as collections or tuples, respectively.

$$
\begin{aligned}
\mathtt{merge}_{\mathcal{K}}(\mathcal{R}) \;\triangleq\; & \mathtt{merge}_{\text{naive}}(\{\, \tau \mid \tau \in \mathcal{R} - \mathcal{O} - \mathcal{A} \,\}) \\
\cup\; & \mathtt{merge}_{\mathcal{A}}(\{\, \tau \mid \tau \in \mathcal{R} \cap \mathcal{A} \,\}) \\
\cup\; & \mathtt{merge}_{\mathcal{O}}(\{\, \tau \mid \tau \in \mathcal{R} \cap \mathcal{O} \,\})
\end{aligned}
$$

## 3 Ambiguous Schema Extraction

Interpreting collection of records by the kind of records in the collection are decided by schema discovery techniques. Collections always contain a single entity.

### 3.1 Arrays as Tuple-Like Structures

In below example geo field encoded as an array, as fields coordinates are 2-element tuple and not a collection of numbers. We denote 2D coordinates by schema: [R, R].

**Listing 6** Array as a Tuple-Like Structure

```
{
    "ts": 7,
    "event": "login",
    "user": {
            "geo": [43.4, -7.2],
            "name": "jbond"
        }
}
```

### 3.2 Objects as Collections

$1^{st}$*Approach* : – Maps keys (drugs) to values (prescription counts), field is a nested collection.

**Listing 7** First Approach : Object as a Collection

```
{
    "time": {
        "Thursday": {"15:00": 1},
        "Saturday": {"23:00": 1}
        },
    "business_id": "..."
}
```

$2^{nd} Approach : -$ Schema for above dataset would model it as a collection $\{*-> R\}^*)$

**Listing 8** Second Approach : Object as a Collection

```
{
    "cms_prescription_counts":
        {
            "DOXAZOSIN MESYLATE": 26,
            "MIDODRINE HCL": 12,
            ...
            }
    , ...
}
```

## 3.3 Multi-Entity Collections

Log data and event-based web APIs are often composite streams of multiple data types. Example: - GitHub provides API access to a stream of status updates, consisting various event types as below.

**Listing 9** Multi Entity Collection

```
{
    "payload":
    {
         "size":1,
         "head": "...",
         "commits": [
            { "distinct": true, "sha":"...", "message": "..."
                , ...}
        , ... ],
        "type": "PushEvent"
    }
}
{

    "payload":
    {
        "action": "opened",
        "issue": { ... },
        "Created_at":"2018-08-22T16:48:29Z", ...
    }
    "type": "IssuesEvent"

}
```

In Multi Entity Collection example, we can see object fields like "type" are shared between all records, multiple tuple-like structures appear as general schema discovery produces a single unified schema traversing all records and allow mixture of fields.

# 4 System Overview

## 4.1 Algorithms

### 4.1.1 K-reduction Algorithm

Array kinded types are interpreted as single-entity collections and object kinded types are always interpreted as tuples. It is implemented through helper functions and have parameter recursive merge heuristic, k-reduction.

**K-reduction distributivity: -** $merge\_K(R1UR2) = merge\_K(merge\_K(R1)Umerge\_K(R2))$.

**Helper Functions : -** merge_array_coll (merge, R) and merge_object_tuple (merge, R)

**Note: -** merge_K is expressed as an associative operation, here limiting to associative operation limits the use of global statistics about the collection, in turn limiting available strategy for resolving ambiguity.

### 4.1.2 JXPLAIN Merge Algorithm

Jxplain Merge algorithm makes two decisions: -

1. Does a bag of array- or object-kinded types encode a collection or a tuple?
2. Given a bag of tuples, are there multiple entities represented in the bag?

Also, decisions are encoded in two functions heuristics is_collection and partition.

**Collection Type Inference: -** If input elements are determined to be collections, nested types are merged to get collection nested type.

**Tuple Type Inference: -** If input elements are tuples, JXPLAIN partitions the bag into individual entities and produces a schema for each individually.

### 4.1.3 JXPLAIN Implementation on Apache Spark

JXPLAIN algorithm is not an associative operation. It decouples the heuristics into separate computation stages, each taking one pass over the data.

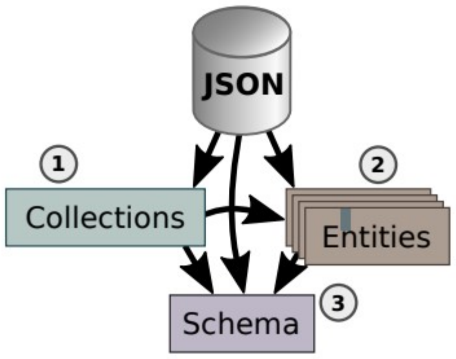


Figure 2: Stages of Extraction in Jxplain

1. Pass 1: - is_collection heuristic to determine set of paths at which collection is present.
2. Pass 2: - Partition Entities
3. Pass 3: - Synthesize Schema

The main problem with the JXPLAIN is, it needs to have the multiple passes to compute the schema which makes it more expensive. Because of this, it uses the sampling techniques, below steps will mitigate the multiple pass problem using sampling.

1. Derive a schema with small training data.
2. Validate the remainder of the training data.
3. Add samples failing validation to sample and repeat.

### 4.2 JXPLAIN Helper Heuristics

JXPLAIN relies on two heuristics: is_collection and partition.

#### 4.2.1 Finding Nested Collections

Whole collection of JSON types as input and produces a set of paths that should be interpreted as collections.

1. Nested Structures $(breakdowninto)->Tuples|Collections$.
2. **Tuples: -** Each field will have distinct type, which in turn creating more precise schema.
3. **Collections: -** No restrictions on fields and uses a single joint schema across all fields creating more compact schema when fields share a common type

#### 4.2.2 Multi Entity Collections

Tuple like types as inputs and outputs a deterministic algorithm for partitioning these input types by entity.

1. ObjectTuple or ArrayTuple element is considered as entity in schema.
2. Reducing I/P types to single entity also including multiple optional fields will create a low precision schema.
3. Single entity with no optional fields for each input type will create a high precision schema

## 5 JXPLAIN Default Heuristics

The goal is to mark the objects as:

1. Collection-Like (Object Collection).
2. Tuple Like (Object Tuple)

Default Heuristics makes this decision on a simple observation: In a collection, keys are more likely to vary than in a tuple, while nested types are likely to be more self-consistent. There are 4 methods or paradigm to achieve above result:

1. Key Space Entropy

2. Similar Type Constraints
3. Differentiating Tuples and Collections
4. Entropy for Arrays

## 5.1 Key Space Entropy

Variation between the key sets of the input objects depends on two conditions:

```
if (Input Objects == Tuple like)
{
    // Keys in some objects are optional
    // Variation will be less as mandatory fields
    // will be present in all tuples
}
if (Input Objects == Collection Type)
{
  //Normal Variation, as each
  //collections maps a different set of keys
}
```

**Key Space Entropy Formula:**

$$\mathcal{E}_{\mathcal{K}} = -\sum_{k} P_k \log P_k \qquad P_k = \frac{\left|\{\ i \mid i \in [N], j \in [M_i], k_{i,j} = k\ \}\right|}{N}$$

In above equation, JXPLAIN computes the probability that an object is selected uniformly at random contains key Pk Also, it captures the variation in keys across the input objects, with higher objects marking as collection type.

**Example:-** Consider below JSON, ts and event field appear in both records and user & files appear in one record , so below result will be produced as per key space entropy

**Listing 10** Json Example for Entropy

```
{
    "ts": 7,
    "event": "login",
    "user": {"geo": [43.4, -7.2], "name": "jbond"}
}
{
    "ts": 8,
    "event": "serve",
    "files": ["q.jpg"," m.jpg"]
}
```

$$P_{ts} = 1, soentropy = (-1 * \log(1)) = 0$$
$$P_{user} = 0.5, entropy = (-1/2 * \log(1/2)) = 0.35$$
$$EK = 0.70 = (20 + 20.35)$$

## 5.2 Similar Type Constraints

As we know, (Variation between field types of objects $\alpha$ 1 / Schema Precise/Concise). If we implement with higher "type entropy", marking the objects as more tuple like. However, with optional fields and multiple levels of collections nesting, number of distinct nested types grows exponentially, and type entropy score becomes more expensive. JXPLAIN follows below constraint rule:

$$\tau_1 \approx \tau_2 \triangleq \begin{cases} \textbf{true} & \textbf{if } \tau_1 = \texttt{null} \textbf{ or } \tau_2 = \texttt{null} \\ \tau_1 = \tau_2 & \textbf{if } \texttt{kind}(\tau_1) \in \{\, \mathbb{B}, \mathbb{R}, \mathbb{S} \,\} \\ \forall i : \tau_1.i \approx \tau_2.i & \textbf{with } i \in \texttt{keys}(\tau_1) \cap \texttt{keys}(\tau_2) \end{cases}$$

According to these constraint-based similarity rules,

1. NULLS are like anything.
2. Primitive types are similar only with themself or with NULL.
3. Complex types are similar only if nested elements at matching keys and positions are similar.

## 5.3 Differentiating Tuples & Collections

The input objects are considered tuples if below conditions are followed:

1. Two nested values have dissimilar types.
2. Key-Space Entropy is below threshold.

Process is explained in below algorithm.

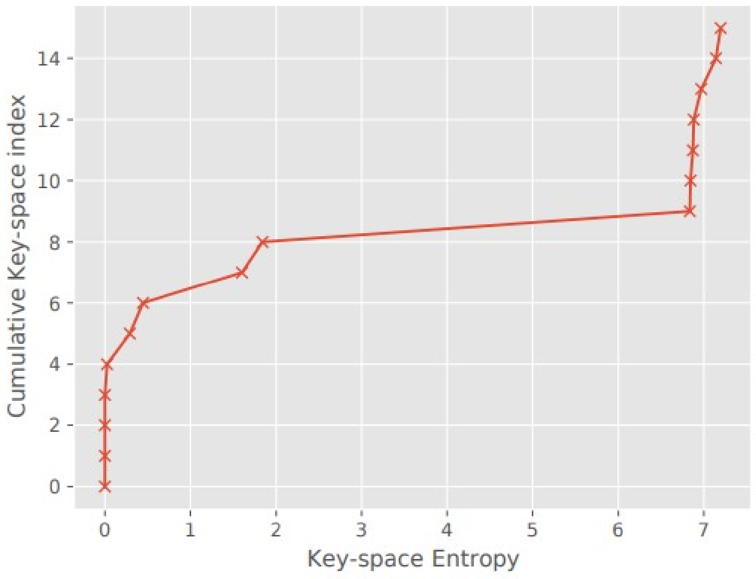
**Algorithm 5** Collection Detection Heuristic

**In:** $\mathcal{R}$ (a bag of object-kinded record types)
**Out:** A designation: `Collection` or `Tuple`

```
1:  RecordCount = 0;  KeyCount = { ∗ : 0 };  $\mathcal{E}_\mathcal{T} = \mathcal{E}_\mathcal{K} = 0$
2:  for all $\tau \in \mathcal{R}$ do
3:      RecordCount += 1;   KindCount = { ∗ : 0 }
4:      for all key ∈ keys($\tau$) do
5:          KeyCnt[key] += 1 ;   KindCount[kind($\tau$.key)] += 1
6:      for all (kind : count) ∈ KindCount do
7:          $\mathcal{E}_\mathcal{T} \mathrel{+}= \frac{\text{count}}{|\text{keys}(\tau)|} \log\left(\frac{\text{count}}{|\text{keys}(\tau)|}\right)$
8:  if $\mathcal{E}_\mathcal{T} > 0$ then return Tuple
9:  for all (key : count) ∈ KeyCount do
10:     $\mathcal{E}_\mathcal{K} \mathrel{+}= \frac{\text{count}}{\text{RecordCount}} \log\left(\frac{\text{count}}{\text{RecordCount}}\right)$
11: if $\mathcal{E}_\mathcal{K} \leq 1$ then return Tuple                ▷ Threshold value
12: else return Collection
```

Precise value of key-space is unimportant because of two reasons:

1. Optional fields are rare.
2. In below graph, each point is one complex-kinded path with self-similar nested elements and very high entropy in case of multi-model distribution.



## 5.4 Entropy For Arrays

Array allows nesting, some use cases treat arrays more like tuples.
**Example: -** Twitter API encodes coordinates as 2-element arrays (Latitude and Longitude

with type [R, R]), or CSV file may be formatted as analogous to objects The type constraint maps naturally to arrays, while key-space entropy is computed from the distribution of array length Pl , rather than set of keys.

# 6 Entity Discovery

| Collection Type | Goal | Entity Discovery Challenge |
|---|---|---|
| Multi Entity | Recover entities with distinct schema | Optional field of a single entity, Separating Entities by distinguishing features. |

Existing schema discovery systems produces one big schema with all fields from all events. Given: - Set of keys, N object-kinded records

$$\{ k_{1,1}, \ldots, k_{1,M_1} \}, \qquad \ldots, \qquad \{ k_{N,1}, \ldots, k_{N,M_N} \}$$

Entity Discovery Problem: - Find K schemas S that union to replicate SG ( SG is "ground-truth" JSON schema) as closely as possible. Let's assume we have two JSON schemas admit every record as shown below: -

**Listing 11** Example for Schema Discovery Problem

```
S1 = {
        { "ts": R, "event" : S, "user": {..} },
        { "ts": R, "event": S, "files": [S] }
    }
S2 = {{ "ts" : R, "event" : S, "user?" : {..}, "files?" : [S]}  }
```

Here in above two schemas S1 encodes two distinct entities (one for each event type) , S2 uses single entity with optional fields.

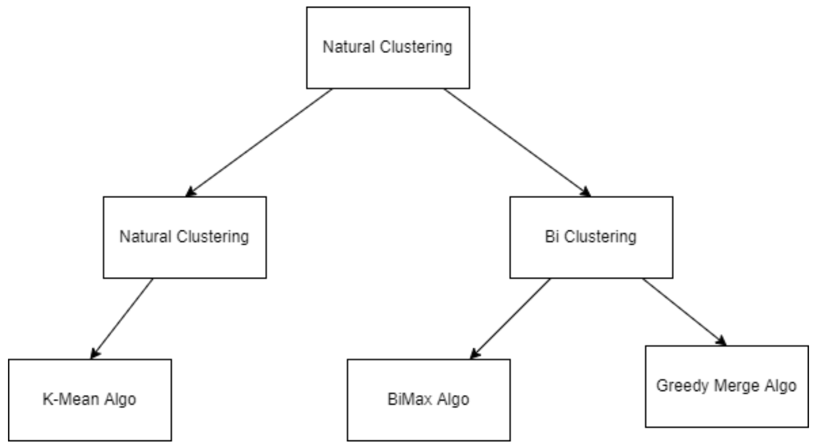
## 6.1 Available Solutions

**Extreme 1: -** L-reduction considers each record to be a distinct entity (K=N), it does not generalize beyond the specific schemas in the training data.

**Extreme 2: -** All training objects are single entity (K = 1), it is concise but admits many more types than original schema would.

**Solution: -** Balance between Extreme 1 and Extreme 2 that is 1 ¡= K ¡= N

### 6.1.1 Clustering



### 6.1.2 Natural Clustering

A solution is to cluster similar records together through k-means. It presents two challenges.

1. K may not be known, if it is known entities of $S_G$ may have different no of attributes.
2. When each field weighted equally, asymmetry poses a problem for classic measures of similarity.

| Dataset | Mandatory Fields | Shared Mandatory Field | Missing Optional Fields |
|---|---|---|---|
| Yelp Photos | 4 | Business_id | |
| Business | 20 | Business_id | 17 |

Above two datasets have 6 distinct fields between them (1 - shared, 3 photos-only, 2 business only). Thus, Jaccard index considers this business more like a photo (1/6 = 0.167) than to a business with all 20 attributes (3/20 = 0.15).

### 6.1.3 Bimax Algorithm

Bi-clustering derive a distance measure that accounts for entity size, by reducing the weights of features in large entities. It presents below problem:
**Entity Detection: -** A problem where we need to simultaneously group records by feature co-occurrence, while also grouping features by co-appearance in records.

**Working of Bimax Algorithm**

1. $K_{max}$ = Largest Key Set
2. $K_{max}(K_{sub}), K_{max}(K_{overlap}), K_{max}(K_{disjoint})$ are the partition groups of remaining records.
3. Order of partition = $K_{sub}K_{overlap}K_{disjoint}$

**Advantages of Bimax Algorithm**

1. Subset relationships are independent of the size of each entity.
2. Knowledge of number of entities is not required, as it simply sorts records to put more similar records closer to one another.

As we need the results clustered into entities, so bimax cannot be applied directly, so we can build entities out of Ksub sets constructed by bimax function returning every set as one cluster. This process is known as naïve bimax version.

**Algorithm 7** Bimax-Naive

```
In: $\mathcal{K}$: an ordered list of key-sets.
Out: $\mathcal{K}_{naive}$: A set of key-set clusters.
1: Sort $\mathcal{K}$ in descending order of key-set size.
2: $i \leftarrow 1$
3: while $i < |\mathcal{K}|$ do
4:     Repeat Bimax lines 4-10.
5:     Add the cluster $\mathcal{K}_{sub}$ to $\mathcal{K}_{naive}$
```

**Note: -** In naïve algorithm, each cluster is seeded from maximal record ( every record in the cluster must be a subset ) which may not occur in input data due to more optional fields.

### 6.1.4 Greedy Merge Algorithm

We need a way to combine all clusters, as linking entities that share keys is insufficient due to small number of fields (FK) may be shared by multiple entities.

**Algorithm 8** GreedyMerge

```
In: $\mathcal{K}_{naive}$: The output of **Bimax-Naive**.
Out: $\mathcal{K}_{merge}$: A list of merged key-set clusters.
1:  for $K_{cand} \in \mathcal{K}_{naive}$ do          ▷ In reverse order of insertion
2:      $k_{cand} \leftarrow$ the maximal element of $K_{cand}$
3:      loop
4:          Find minimal $\mathcal{K}_{cover} \subseteq (K_{naive} - \{ K_{cand} \})$
                s.t. $k_{cand} \subseteq \bigcup_{k \in K; K \in \mathcal{K}_{cover}} k$
5:          if $K_{cover}$ exists then
6:              $\mathcal{K}_{naive} \leftarrow \mathcal{K}_{naive} - \mathcal{K}_{cover}$
7:              $K_{cand} \leftarrow K_{cand} \cup (\bigcup \mathcal{K}_{cover})$
8:              $k_{cand} \leftarrow k_{cand} \cup$ every new key in $\mathcal{K}_{cover}$
9:          else break
10:     Add $K_{cand}$ to $\mathcal{K}_{merge}$
```

**Example: -** Consider 4 entities discovered by Bimax-Naïve over keys A, B, C, D, E with maximal elements:

$$K = \{E1 : \{A, B, E\}E2 : \{B, C, E\}E3 : \{C, D, E\}E4 : \{B, D\}\}$$

**Solution: -**

1. Select smallest Entity E4.
2. Max(E2) U Max (E3) = Superset of E4 maximal element
3. New Candidate Key = $\{B, C, D, E\}$
4. Remove $E_2$ and $E_3$ from further consideration.
5. Entity E1 cannot form a set cover over the joint $\{E1, E2, E3\}$.
6. Result = $E1\&\{E2, E3, E4\}$.

## 7 Experiments

In order to perform the experiment on a designed algorithm, there should be some sort of comparison with some other industry-standard algorithm. Here, The performance of JXPLAIN is evaluated using the K-reduction schema discovery algorithm. While performing experiments, it mainly focuses on two paradigms, which are

1. How well does the discovered schema generalizes beyond the example data, and
2. How few types does the schema admit

It also focuses on the runtime process and resource overhead while performing the schema discovery process. After performing experiments, it validates the following claims:

1. JXPLAIN produces schemas that are significantly more precise (i.e., admit fewer types) than K-reduction.
2. while not incorrectly rejecting types that are legitimately part of the schema,
3. A clustering strategy based on Bimax bi-clustering is preferable to a standard technique like k-means,
4. The merge step described in Section 6 is critical for creating compact schemas, and
5. The overhead of the additional steps required by Jxplain is not prohibitive.

To perform the experiments, it uses various real-world datasets from Github, yelp, Twitter, etc. The following table explains the source, amount of data, and data characteristics that are used in this experiment.

**Table 1** Datasets and Characteristics

| SN | Data Source | # of Tuples | Data Characteristics |
|---|---|---|---|
| 1 | GitHub | 3M | Event Stream Dataset collected over 330 days<br>A large number of entities of wildly varying sizes<br>Complex nesting structure |
| 2 | Pharmaceutical | 240K | Open dataset of pre-doctor Pharmaceutical Prescription Statistics.<br>Highly collection-like object with 2397 distinct Key (largest number of keys among all other datasets that are used for experiments)<br>Almost every record has different types |
| 3 | Twitter | 800K | Tweets dataset<br>Recursive schemas for retweets, deleted tweets and quoted tweets<br>Multitude of object array and geo type tuple array |
| 4 | Matrix Synapse | 150K | Events dataset<br>Char server dataset |
| 5 | NYT | | Archival list of New York Times Articles from 2019 |
| 6 | Wikipedia | 1.7M | Wikidata, collected from Wikipedia articles<br>Closely resembles HTML and XML<br>A deeply nested array of objects<br>Linked data interface (Link to the articles) |
| 7 | Yelp | 7.5M | Six-individual, different schema from Yelp Open Dataset.<br>Extensive use of optional fields<br>Nested Collection<br>Soft Functional Dependency relationship |

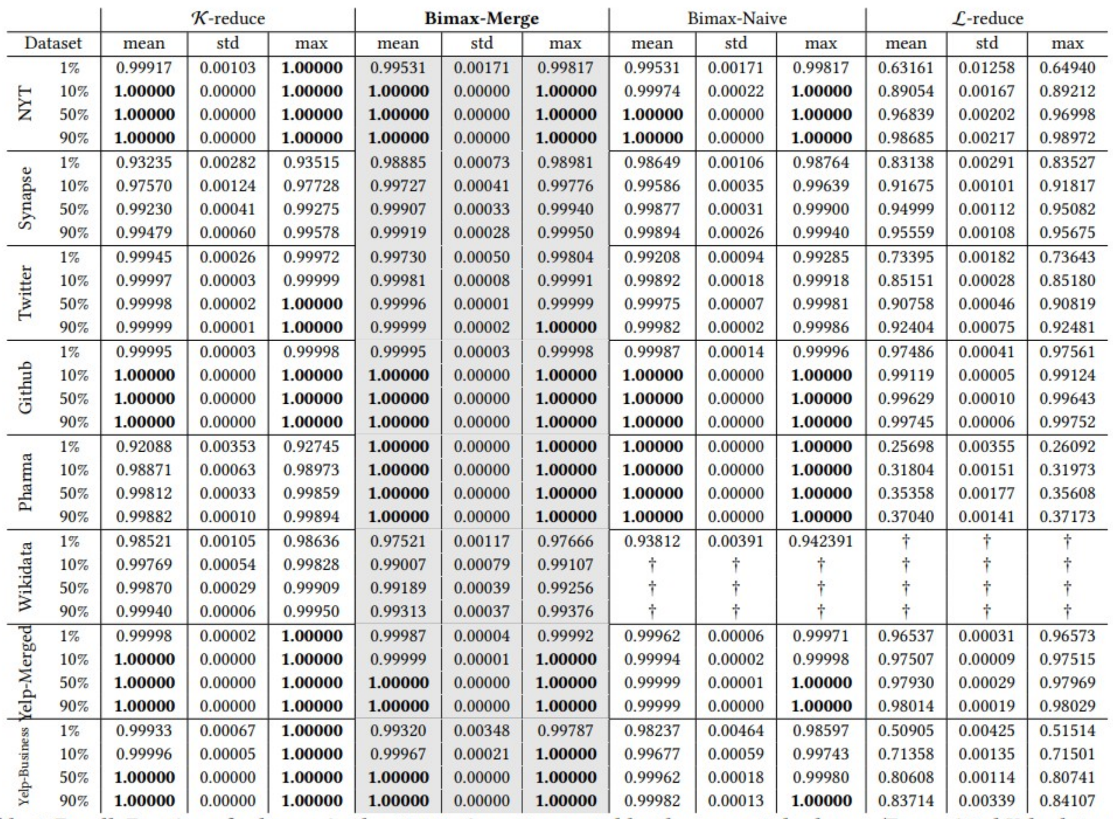
Among them, 10% dataset is used for the testing, and for each test it takes 1%, 10%, 50%, and 90% uniform random samples.

In this paper, the author compared four different algorithms, which are

1. K - Reduce
2. Bimax Naive
3. Bimax Merge
4. L - Reduce

To run the experiment using the above algorithms, they perform all the computations using the system with the specifications as 4 x 20 Core2.40 GHz Intel Xeon E7 Processor, 1 TB RAM, CentOS-7 Linux Operating System.

The output of the experiment can be shown in the following table.



| | | $\mathcal{K}$-reduce | | | **Bimax-Merge** | | | Bimax-Naive | | | $\mathcal{L}$-reduce | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dataset | | mean | std | max | mean | std | max | mean | std | max | mean | std | max |
| NYT | 1% | 0.99917 | 0.00103 | **1.00000** | 0.99531 | 0.00171 | 0.99817 | 0.99531 | 0.00171 | 0.99817 | 0.63161 | 0.01258 | 0.64940 |
| | 10% | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | 0.99974 | 0.00022 | **1.00000** | 0.89054 | 0.00167 | 0.89212 |
| | 50% | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | 0.96839 | 0.00202 | 0.96998 |
| | 90% | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | 0.98685 | 0.00217 | 0.98972 |
| Synapse | 1% | 0.93235 | 0.00282 | 0.93515 | 0.98885 | 0.00073 | 0.98981 | 0.98649 | 0.00106 | 0.98764 | 0.83138 | 0.00291 | 0.83527 |
| | 10% | 0.97570 | 0.00124 | 0.97728 | 0.99727 | 0.00041 | 0.99776 | 0.99586 | 0.00035 | 0.99639 | 0.91675 | 0.00101 | 0.91817 |
| | 50% | 0.99230 | 0.00041 | 0.99275 | 0.99907 | 0.00033 | 0.99940 | 0.99877 | 0.00031 | 0.99900 | 0.94999 | 0.00112 | 0.95082 |
| | 90% | 0.99479 | 0.00060 | 0.99578 | 0.99919 | 0.00028 | 0.99950 | 0.99894 | 0.00026 | 0.99940 | 0.95559 | 0.00108 | 0.95675 |
| Twitter | 1% | 0.99945 | 0.00026 | 0.99972 | 0.99730 | 0.00050 | 0.99804 | 0.99208 | 0.00094 | 0.99285 | 0.73395 | 0.00182 | 0.73643 |
| | 10% | 0.99997 | 0.00003 | 0.99999 | 0.99981 | 0.00008 | 0.99991 | 0.99892 | 0.00018 | 0.99918 | 0.85151 | 0.00028 | 0.85180 |
| | 50% | 0.99998 | 0.00002 | **1.00000** | 0.99996 | 0.00001 | 0.99999 | 0.99975 | 0.00007 | 0.99981 | 0.90758 | 0.00046 | 0.90819 |
| | 90% | 0.99999 | 0.00001 | **1.00000** | 0.99999 | 0.00002 | **1.00000** | 0.99982 | 0.00002 | 0.99986 | 0.92404 | 0.00075 | 0.92481 |
| Github | 1% | 0.99995 | 0.00003 | 0.99998 | 0.99995 | 0.00003 | 0.99998 | 0.99987 | 0.00014 | 0.99996 | 0.97486 | 0.00041 | 0.97561 |
| | 10% | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | 0.99119 | 0.00005 | 0.99124 |
| | 50% | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | 0.99629 | 0.00010 | 0.99643 |
| | 90% | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | 0.99745 | 0.00006 | 0.99752 |
| Pharma | 1% | 0.92088 | 0.00353 | 0.92745 | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | 0.25698 | 0.00355 | 0.26092 |
| | 10% | 0.98871 | 0.00063 | 0.98973 | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | 0.31804 | 0.00151 | 0.31973 |
| | 50% | 0.99812 | 0.00033 | 0.99859 | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | 0.35358 | 0.00177 | 0.35608 |
| | 90% | 0.99882 | 0.00010 | 0.99894 | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | 0.37040 | 0.00141 | 0.37173 |
| Wikidata | 1% | 0.98521 | 0.00105 | 0.98636 | 0.97521 | 0.00117 | 0.97666 | 0.93812 | 0.00391 | 0.942391 | † | † | † |
| | 10% | 0.99769 | 0.00054 | 0.99828 | 0.99007 | 0.00079 | 0.99107 | † | † | † | † | † | † |
| | 50% | 0.99870 | 0.00029 | 0.99909 | 0.99189 | 0.00039 | 0.99256 | † | † | † | † | † | † |
| | 90% | 0.99940 | 0.00006 | 0.99950 | 0.99313 | 0.00037 | 0.99376 | † | † | † | † | † | † |
| Yelp-Merged | 1% | 0.99998 | 0.00002 | **1.00000** | 0.99987 | 0.00004 | 0.99992 | 0.99962 | 0.00006 | 0.99971 | 0.96537 | 0.00031 | 0.96573 |
| | 10% | **1.00000** | 0.00000 | **1.00000** | 0.99999 | 0.00001 | **1.00000** | 0.99994 | 0.00002 | 0.99998 | 0.97507 | 0.00009 | 0.97515 |
| | 50% | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | 0.99999 | 0.00001 | **1.00000** | 0.97930 | 0.00029 | 0.97969 |
| | 90% | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | 0.99999 | 0.00000 | **1.00000** | 0.98014 | 0.00019 | 0.98029 |
| Yelp-Business | 1% | 0.99933 | 0.00067 | **1.00000** | 0.99320 | 0.00348 | 0.99787 | 0.98237 | 0.00464 | 0.98597 | 0.50905 | 0.00425 | 0.51514 |
| | 10% | 0.99996 | 0.00005 | **1.00000** | 0.99967 | 0.00021 | **1.00000** | 0.99677 | 0.00059 | 0.99743 | 0.71358 | 0.00135 | 0.71501 |
| | 50% | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | 0.99962 | 0.00018 | 0.99980 | 0.80608 | 0.00114 | 0.80741 |
| | 90% | **1.00000** | 0.00000 | **1.00000** | **1.00000** | 0.00000 | **1.00000** | 0.99982 | 0.00013 | **1.00000** | 0.83714 | 0.00339 | 0.84107 |

**Table 1: Recall: Fraction of schemas in the 10% testing set accepted by the generated schema. (For omitted Yelp datasets $\mathcal{K}$-reduce, Bimax-Merge, and Bimax-Naive obtain 100% validation for all training sizes) († Indicates system ran out of resources)**

## 8 Critique & Conclusion

Jxplain avoids two assumptions concerning the creation of JSON records:

1. Which hierarchical structures encode nested collections, and
2. How many entities exist in a JSON object collection (nested or not)

From these experiments and results, it seems that JXPLAIN could have a strong influence on the Ambiguous Aware JSON Schema Discovery. Because It has explained what factors we should care about during the discovery process. While describing the system for the design process, they have explained with good reasonable examples and there is strong evidence for every claim.

During these experiments, they have reserved 10% of the data for the testing. And more promising things about this experiment is they have used 12 different real world datasets from github, twitter, yelp, etc. and one synthetic yelp merged dataset. Even with the very low amount of the data they are able to achieve high recall.

Along with the variety of datasets, It mainly explains two concepts or two techniques in order to solve the ambiguity in the schema discovery process in a more precise and efficient way, the first one is Parallelization for extraction and the second one is Sampling for discovery. Parallelization concepts explain that it decouples the heuristics into separate computation stages.

**Note :-** Jxplain adds many pre-processing rounds to conventional schema discovery to avoid these assumptions. Although these passes add a significant amount of cost to the extraction process, the resultant schemas are tighter and more compact, particularly for complicated Json data models, lowering the amount of human tweaking necessary to optimize them.