Reducing Ambiguity in Json Schema Discovery

William Spoth, Oliver Kennedy
University at Buffalo, SUNY
{wmspoth, okennedy}@buffalo.edu

Ying Lu, Beda Hammerschmidt, Zhen Hua Liu
Oracle
{ying.lu, beda.hammerschmidt, zhen.liu}@oracle.com

ABSTRACT
Ad-hoc data models like JSON simplify schema evolution and enable multiplexing various data sources into a single stream. While useful when writing data, this flexibility makes JSON harder to validate and query, forcing such tasks to rely on automated schema discovery techniques. Unfortunately, ambiguity in the schema design space forces existing schema discovery systems to make simplifying, data-independent assumptions about schema structure. When these assumptions are violated, most notably by APIs, the generated schemas are imprecise, creating numerous opportunities for false positives during validation. In this paper, we propose JXPLAIN, a JSON schema discovery algorithm with heuristics that mitigate common forms of ambiguity. Although JXPLAIN is slightly slower than state of the art schema extractors, we show that it produces significantly more precise schemas.

CCS CONCEPTS
• Information systems → Semi-structured data;

KEYWORDS
Json; JSON-schema; semi-structured; entity detection

ACM Reference Format:

1 INTRODUCTION
Record-level schema formats like JSON are the de-facto data representation for rapidly evolving applications like REST APIs, data loggers, and data portals. JSON data is easy to create programmatically, offering a path for flexible schema evolution, and allows easy nesting of collections and structures. However, when each record defines its own schema, by (self-)definition it is virtually impossible to detect data errors or structural changes in new data. For example, an operations engineer monitoring JSON log data may want to be warned when the structure of newly arriving events changes, as this may signify errors, or the addition of new event types. However, detecting such changes first requires a concise description of “typical” log data — a collection-level schema.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD '21, June 20–25, 2021, Virtual Event, China. © 2021 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-8345-1/21/06...$15.00
https://doi.org/10.1145/3448016.3452801

Figure 1: Example Json Data
Repeated attempts at inferring collection-level schemas [3, 5, 7, 18] from JSON records run into a common problem here: Nesting makes it impossible (in general) to assert a single, unambiguous schema from a set of example JSON records. Such techniques, which are typically designed to summarize JSON collections for human consumption and not for data validation, resort to overgeneralizing. The resulting schemas are descriptive (i.e., they have high recall), but achieve this descriptiveness by adopting the broadest, most permissive of the ambiguous interpretations of the input data (i.e., they have low precision). This generality makes the resulting schemas ill suited for use in data validation.

Example 1. Consider the two JSON records in Figure 1. Production schema discovery systems (e.g., Spark’s [3]) assume that objects in a collection are instances of a single entity. This assumption leads them to (correctly) assert that all records in the data must have an integer ts field and a string event field. However, it also leads them to (incorrectly) assume that variation between records is exclusively caused by optional fields. Intuitively, a record can not simultaneously be a login and a serve event, but existing schema discovery systems lack sufficient information to make this distinction. Thus, the proposed schema will also admit any of the following (invalid) records:

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

Because the user field and the files field are independently optional, the proposed schema will accept not only the two expected record-level schemas, but also records with both fields, or records with neither.

Existing approaches resolve ambiguity by assuming that the data conforms to three informal conventions [7]: (i) Collections contain a single entity type, (ii) JSON arrays always encode collections, and (iii) JSON objects always encode touples. These assumptions are sufficient for simple, homogeneous JSON collections, but break down on the complex nesting structures appearing in more heterogeneous data sources like web service APIs or JSON-formatted system logs.

In this paper, we develop a new JSON schema discovery system called JXPLAIN. In contrast to existing techniques (e.g., [3, 7]) that resolve ambiguity through data-independent heuristics, JXPLAIN’s heuristics resolve schema ambiguity on a per-instance basis. As we show, the resulting schemas capture the structure of JSON record collections with negligible loss relative to existing techniques (i.e., recall stays high), while simultaneously admitting a far narrower range of JSON records (i.e., precision improves) when compared to the same state-of-the-art systems.

Concretely, this paper makes the following contributions: (i) We identify forms of schema ambiguity that cause existing JSON schema discovery techniques to produce low-precision schemas, (ii) We
present Jxplain, a general framework for heuristically resolving this ambiguity, (iii) We show the feasibility of Jxplain by proposing specific heuristics for detecting and resolving these forms of ambiguity, (iv) We show experimentally that Jxplain with these heuristics creates schemas with significantly higher precision than state-of-the-art schema discovery, with negligible change in recall.

2 BACKGROUND AND NOTATION

The goal of JSON schema discovery is to re-construct a hidden ground truth schema — a description of a set of valid JSON records — from a finite collection of records sampled from this set. An ideal schema for this collection is a JSON object with any field types, e.g., strings or numbers. We add a question mark to a field name to mark it as optional; The resulting schema accepts object types with any valid type field value.

When clear from context, we abuse notation using types (e.g., $\tau$) to denote singleton schemas (i.e., $\{ \tau \}$) or complex. As summarized in Figure 2, a primitive JSON value is a boolean $\mathbb{B}$, a numeric value $\mathbb{R}$, or a string value $\mathbb{S}$, or the value null $\text{null}$. A complex JSON value is any array or object. A JSON array of type $[ \tau_1, \ldots, \tau_N ]$ is an ordered sequence of $N$ values with types $\tau_1 \ldots \tau_N$. A JSON object of type $\{ k_1 : \tau_1, \ldots, k_N : \tau_N \}$ is a collection of mappings from keys $k_1 \ldots k_N$ to values with types $\tau_1 \ldots \tau_N$. We define the kind of a type $\tau$ to be $\tau$ if it is primitive, or the symbol $O$ or $A$ if $\tau$ is an object or array respectively:

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

**Example 2.** The JSON object with ts 7 in Figure 1 has type:

$$\{ \text{ts : } \mathbb{R}, \text{event : } \mathbb{S}, \text{user : } \{ \text{name : } \mathbb{S}, \text{geo : } \{ \mathbb{R}, \mathbb{R} \} \} \}$$

The kind of the record is $O$. The field event has kind $S$.

If $\tau$ is an object (resp., array), we write keys($\tau$) to denote the set of keys mapped by the object (resp., the valid indices of the array; we also refer to these as keys). We write $\tau.k$ to denote the type of the value nested under key $k$. We refer to this as a field type of $\tau$. Denote by $p$ a path, a sequence of keys $p_1, \ldots, p_n$.

**Definition 1 (Schema).** A schema $S$ is a set of types $\tau \in S$. We say that $\tau$ is admitted by the schema if it is an element of the set.

When clear from context, we abuse notation using types (e.g., $\tau$) to denote singleton schemas (i.e., $\{ \tau \}$). We define the following three shorthands for compactly representing nested schemas:

**Optional Fields.** We add a question mark to a field name to mark it as optional; The resulting schema accepts object types with any subset of the fields marked optional.

$$\{ k_1 : \tau_1, \ldots, k_m : \tau_m, \ k'_1 : \tau'_1, \ldots, k'_m : \tau'_m \} \triangleq \{ k_1 : \tau_1, \ldots, k_m : \tau_m, k'_1 : \tau'_1, \ldots, k'_m : \tau'_m \} \ \cap \{ \tau_1, \ldots, \tau_m \} \subseteq [m]$$

**Schema Nesting.** We write $\{ k : [S, \ldots] \}$ (resp., $\{ S, \ldots \}$) to denote the schema admitting like-kind types with corresponding field types admitted by the schema $S$. That is:

$$\{ k_1 : S_1, \ldots, k_N : S_N \} \triangleq \{ k_1 : \tau_1, \ldots, k_N : \tau_N \} \ \cap \{ \tau_1 \in S_1 \}$$

**Collection Types.** We write $\{ * : [S] \}$ (resp., $\{ [S] \}$) to denote a collection schema that admits any object-kind type (resp., array-kind) who’s field types are drawn from $S$.

$$\{ * : [S] \} \triangleq \{ k_1 : \tau_1, \ldots, k_N : \tau_N \} \ \cap \ N \in \mathbb{N} \ \cap \tau_1, \ldots, \tau_N \in S$$

**2.1 Schema Discovery**

We are given a collection of records with $N$ types $R = \{ \tau_1, \ldots, \tau_N \}$ drawn from some hidden ground truth schema $S_G$. The schema discovery problem is to "merge" these types into a new schema definition (denoted $\text{merge}(R)$) that closely approximates $S_G$. This derived schema should have high precision, admitting only ground truth types (i.e., $|S_G \Land \text{merge}(R)| = 0$) and high recall, admitting all ground truth types (i.e., $|S_G \merge(R)| = 0$). We would also like the derived schema to have a compact representation, avoiding explicit type enumeration by using the shorthands defined above.

**Naive Discovery.** Naively, we might take the sample records to be the definitive set of types admitted by $S_G$. This is analogous to the $L$-reduction of Baazizi et al. [7]. In other words we define:

$$\text{merge}_{\text{naive}}(R) \triangleq \{ \tau_1, \ldots, \tau_N \}$$

This approach guarantees high precision, but (i) rejects types missing from the input (i.e., has low recall) and (ii) is not compact.

**Example 3.** Applied to the two records in Figure 1, naive discovery simply returns a set of the two distinct schemas:

$$\{ \{ \text{ts : } \mathbb{R}, \text{event : } \mathbb{S}, \text{user : } \{ \text{name : } \mathbb{S}, \text{geo : } \{ \mathbb{R}, \mathbb{R} \} \} \} \}$$

**Arrays as Collections.** JSON arrays are commonly used to encode nested collections. For array-kind records, existing algorithms discover the schema of the nested collection by recursively applying schema discovery to the union of the array’s elements.

The merge operation groups nested field types by their key and recursively merges groups. Defining $\{ k_1, \ldots, k_k \} \equiv \text{keys}_{\text{array}}(R)$, while fields appearing in only some objects are optional (with keys keys$_{\text{array}}(R)$).

$$\text{keys}_{\text{array}}(R) \equiv \bigcap \text{keys}(\tau) \ \cap \text{keys}_{\text{array}}(\tau) \equiv \bigcup \text{keys}(\tau) \ \cap \text{keys}_{\text{array}}(\tau)$$

**Standard Discovery.** The classical approach to schema discovery, used in production systems like Spark’s JSON data source [3] or Oracle’s JSON Data Guide [18], is modeled by Baazizi et al.’s $K$-reduction [5], defined formally as follows:

$$\text{merge}_K(R) \equiv \text{merge}_{\text{naive}}(\{ \tau : \tau \in R - O - A \}) \cup \text{merge}_A(\{ \tau : \tau \in R \cap A \}) \cup \text{merge}_O(\{ \tau : \tau \in R \cap O \})$$

This approach uses naive merge for primitive types and recursively merges arrays and objects as collections or tuples, respectively.
3 AMBIGUOUS SCHEMA EXTRACTION

In summary, existing schema discovery techniques decide how to interpret a collection of records by the kind of the records in the collection: Arrays are always collection-like, objects are always tuple-like, and collections always contain a single entity. We now highlight examples of JSON in the wild that that violate these assumptions, leading to imprecise schemas. A detailed description of all datasets discussed can be found in Section 7.

3.1 Arrays as Tuple-Like Structures

Example 5. Consider the user.geo field of Figure 1. Although encoded as an array, this field’s geospatial coordinates are actually a 2-element tuple and not a collection of numbers. 2D coordinates would be more precisely described by the schema: [R, R].

Many web service APIs including Twitter [32] and Yelp [35] follow the GeoJSON standard [11] and use 2-element arrays for coordinates. Similarly, arrays sometimes encode tuples in settings where JSON data is naively generated from CSV files [22]. In each case, treating all arrays as collections (e.g., [R, 1] instead of [R, R]) results in unnecessarily permissive schemas.

3.2 Objects as Collections

Example 6. Consider the pharmaceutical dataset [25] described in the experiments section, which has a collection-like object that maps drug names to prescription counts:

```json
{  "cms_prescription_counts":  [  "DOXAZOSIN MESYLATE": 26,  "MIDODRINE HCL": 12, ... ],  ...
```

Although encoded as an object, this field is a nested collection, where each element maps keys (drugs) to values (prescription counts). A better schema for this dataset would model it as a collection (i.e., {k → R}). We also observed collection-style objects in many other API datasets, including Yelp’s checkins dataset:

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

…as well as the matrix chat server event log [19]:

```json
{  "users": {"Alice": 100, "Bob": 100, ... }, ...
```

Typical schema discovery treats all objects as tuples, always assuming missing elements to be optional fields. In the example, the use of optional attributes is very verbose, as in each case the descendants share a schema. Furthermore, optional attributes can not describe new field names (e.g., new medications or user names) as they are added, as well reducing the schema’s recall.

3.3 Multi-Entity Collections

Log data and event-based web APIs are often composite streams of multiple data types. For example, GitHub provides API access to a stream of status updates, consisting of 49 event types, including:

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

Multi-entity collections can also be found in nested collections, like the New York Times article API [31], which includes a multimedia object array containing summary metadata:

```json
{  "multimedia": [  {"legacy": {"thumbnail": "03 Prose1 - thumbStandard.jpg", "xlarge": {"xlargewidth": 680, "xlargeheight": 450}, ...},  {"legacy": {"thumbnail": "03Prosel-thumbStandard.jpg", "thumbnailwidth": 75, "thumbnailheight": 75}, ... }], ...
```

Though object fields (e.g., type) are shared between all records, multiple tuple-like structures appear. Typical schema discovery produces a single unified schema spanning all records. Such schemas are unnecessarily permissive, admitting arbitrary mixtures of fields.

4 SYSTEM OVERVIEW

We now introduce Jxplain, a general framework for implementing ambiguity-aware schema discovery. To represent generated schemas, we use a subset of the JSON Schema specification captured by the following grammar $S_f$. Primitive types are explicit:

$$S_f ::= R | S | B | null$$

Tuple-like arrays and objects are defined in terms of nested schemas:

| ArrayTuple($S_f, S_f, ...$) |
| ObjectTuple($k : S_f, k : S_f, ...$, $k^7 : S_f, k^7 : S_f, ...$) |

Collection-like complex types are similarly defined in terms of a single nested schema, and a union type combines alternatives:

| ArrayCollection($S_f$) | ObjectCollection($S_f$) |
| Union($S_f, S_f, ...$) |

This grammar mirrors the schema shorthands given in Section 2 and its semantics follow naturally. We abuse notation and use expressions in this grammar interchangeably with schemas. As an example, Algorithm 1 implements $K$-reduction as presented in Section 2. Via helper functions (Algorithms 2 and 3), array-kind types are always interpreted as single-entity collections, while object-kind types are always interpreted as tuples. Both helper functions are parameterized by a recursive merge heuristic, $K$-reduction itself in this example. The central feature of $K$-reduction is its distributivity over union [7]:

$$merge_K(\mathcal{R}_1 \cup \mathcal{R}_2) = merge_K(merge_K(\mathcal{R}_1) \cup merge_K(\mathcal{R}_2))$$

Thus, $merge_K$ can be expressed as an associative fold operation. Considering that the encoded representation is typically smaller

Algorithm 1 merge_K(\mathcal{R})

In: $\mathcal{R}$: A bag of types

1: return Union($\mathcal{R} \cap \{R, S, B, null\}$),

merge_array_coll(merge_K($\mathcal{R} \cap \mathcal{A}$),
merge_object_tuple(merge_K($\mathcal{R} \cap \mathcal{O}$)))

1https://json-schema.org/
when deciding how to resolve ambiguity. We first describe the Jxplain Algorithm 4 (Algorithm 2 merge_array_coll(merge, \(R\))

In: \(merge\): A recursive merge function
In: \(R\): A bag of array-kindged types

1. return ArrayCollection(merge({ \(\tau.k \mid k \in \text{keys}(\tau) \in R \}))

Algorithm 3 merge_object_tuple(merge, \(R\))

In: \(merge\): A recursive merge function
In: \(R\): A bag of object-kindged types

1. \(k_1, \ldots, k_k \leftarrow \text{keys}_A(R)\)
2. \(k_1^?, \ldots, k_k^? \leftarrow \text{keys}_A(R)\)
3. return ObjectTuple(
   \(k_1 \mapsto \text{merge}(\{ \tau.k_1 \mid \tau \in R \}), \ldots,\)
   \(k_k \mapsto \text{merge}(\{ \tau.k_k \mid \tau \in R \}),\)
   \(k_1^? \mapsto \text{merge}(\{ \tau.k_1^? \mid k_1^? \in \text{keys}(\tau), \tau \in R \}), \ldots,\)
   \(k_k^? \mapsto \text{merge}(\{ \tau.k_k^? \mid k_k^? \in \text{keys}(\tau), \tau \in R \})\))

Algorithm 4 Jxplain (\(R\))

In: \(R\): A bag of types
1. \(S_A = \emptyset \) \(S_O = \emptyset \) \(A = R \cap A \) \(O = R \cap O \)
2. if |\(A| > 0\) then
3. \(\text{if is_collection}(A)\) then
4. \(S_A \leftarrow \text{merge_array_coll}(\text{Jxplain}, A)\)
5. else
6. \(A_1, \ldots, A_k \leftarrow \text{partition}(A)\)
7. \(S_A \leftarrow \text{Union}(\text{merge_array_tuple}(\text{Jxplain}, A_1), \ldots,\)
8. \(\text{merge_array_tuple}(\text{Jxplain}, A_k))\)
9. if |\(O| > 0\) then
10. \(\text{if is_collection}(O)\) then
11. \(S_O \leftarrow \text{merge_object_coll}(\text{Jxplain}, A)\)
12. else
13. \(O_1, \ldots, O_k \leftarrow \text{partition}(O)\)
14. \(S_O \leftarrow \text{Union}(\text{merge_object_tuple}(\text{Jxplain}, O_1), \ldots,\)
15. \(\text{merge_object_tuple}(\text{Jxplain}, O_k))\)
16. return Union(\(R \cap \{ \text{R}, \text{S}, \text{B}, \text{null} \}, S_A, S_O\))

than the size of the input type bags, it is extremely amenable to
distributed computation. Unfortunately, limiting ourselves to asso-
ciative folds limits the use of global statistics about the collection,
in turn limiting available strategies for resolving ambiguity.

4.1 Naive Implementation

Jxplain’s merge algorithm (sketched in simplified form as Algo-
rithm 4) relaxes this restriction, considering the data as a whole
when deciding how to resolve ambiguity. We first describe the
simplified algorithm, before discussing performance optimizations.
Broadly, two decisions need to be made: (i) Does a bag of array-
or object-kindged types encode a collection or a tuple?, and (ii) Given
a bag of tuples, are there multiple entities represented in the bag?
These decisions are encoded in the is_collection and partition
helper heuristics, which we discuss in greater detail below. If the
elements of the input bag are determined to be collections, nested
types are merged together to infer the collection-nested type (Algo-
rithm 2 and its object analog). If the input bag’s elements are

4.2 Jxplain

We now outline the details of Jxplain implemented on Apache
Spark, and in particular how we address bottlenecks in the simpli-
fied Algorithm 4 presented above.

Parallelization. We expect the two heuristics to need to see the en-
tire input before producing an output, so the simplified Algorithm 4
is not an associative fold, and so not amenable to distribution.
Jxplain instead decouples these heuristics into separate computa-
tion stages, each taking one pass over the data as illustrated in
Figure 3. Pass ① invokes the is_collection heuristic to determine
the set of paths at which a collection is present. Pass ② adapts the
partition heuristics to precompute a strategy for partitioning en-
tities. Finally Algorithm 4 runs as pass ③ to synthesize the schema.

Sampling. The need for multiple passes makes computing schemas
more expensive. One mitigation is to run Jxplain on only a small
sample of training data. As we show in the Section 7, entropy-based
collection detection is surprisingly robust (even a 1% sample is
often almost perfect). The notable exception is when the schema
involves a rare object field, array index, or collection-nested type.
To mitigate this problem, Jxplain can be used iteratively with the
following steps: (i) Derive a schema from a small sample of the
training data, (ii) Validate the remainder of the training data, (iii)
Add samples failing validation to the sample and repeat.

4.3 Helper Heuristics

Jxplain relies on two heuristics: is_collection and partition
in Algorithm 4. We now outline the design goals for both heuristics
before presenting two specific realizations in Sections 5 and 6.

Detecting Nested Collections. Following prior work in schema
extraction (e.g., [5, 7]), Jxplain focuses on collapsing nested struc-
tures into either tuples or collections. Tuples bind the set of al-
lowed fields (resp., positions) and allow each field to have distinct
types, creating a more precise schema when the set of fields is
stable. Collections do not restrict which fields are allowed and use
a single joint schema across all fields, creating a more compact
schema when fields share a common type. A compatible heuristic
needs to chose between these two strategies.

Concretely Jxplain expects this heuristic to be implemented as
a process that takes the full collection of JSON types as input and
produces a set of paths that should be interpreted as collections.

Multi-Entity Collections. We refer to each ObjectTuple or Ar-
RayTuple element in a schema as an entity. Prior work considers
two extremes when deciding how to extrapolate entities from collections of types. Reducing all input types to a single entity (as in \( K \)-reduction) with multiple optional fields, creates a high-recall, low-precision schema. Conversely, constructing one entity with no optional fields for each input type (as in \( L \)-reduction) creates a high-precision, low-recall schema. A compatible heuristic needs to select a point on the continuum between these two extremes.

Concretely, JXPLAIN expects this heuristic to be implemented as a process that takes a bag of tuple-like types as input and outputs a deterministic algorithm for partitioning these input types by entity.

5 DETECTING COLLECTIONS

Json objects and arrays can both encode nested collections or nested tuple-like structures. This section describes a default heuristic for JXPLAIN that distinguishes between these cases. We initially target object-kind types as inputs, before generalizing to arrays below.

\[
\begin{align*}
\{ k_{1,1} : r_{1,1}, \ldots, k_{1,M_1} : r_{1,M_1} \}, \ldots, \\
\{ k_{N,1} : r_{N,1}, \ldots, k_{N,M_N} : r_{N,M_N} \}
\end{align*}
\]

The goal is to mark the objects as (a) collection-like (i.e., Object-Collection), or (b) tuple-like (i.e., ObjectTuple). JXPLAIN’s default heuristic makes this decision relying 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.

5.1 Key-Space Entropy

Variation between the key sets (i.e., \( \text{keys}(\tau) \)) of the input objects can be explained in two ways. If we believe that the input objects are tuple-like, keys that only appear in some objects are optional. Conversely, if we believe that the input objects are collection-like, variation is normal, as each collection maps a different set of keys. We would expect less variation in the former case, as any mandatory fields will be present in all tuples, and the number of fields of a tuple (dozens) is, in our experimental data, smaller than the domain of collections (hundreds or thousands). Thus, we expect the distribution of keys in tuple-like objects to be more limited. We quantify this variation through the corresponding entropy measure:

\[
E_K = -\sum_k P_k \log P_k
\]

For each key, JXPLAIN computes the probability that an object selected uniformly at random contains the key \( (P_k) \). The resulting Key-Space entropy \( (E_K) \) is a numerical value that captures variation in keys across the input objects, with higher values marking the objects as more collection-like.

Example 7. Consider the two records of Figure 1. The ts and event keys appear in both records (e.g., \( P_{ts} = 1 \)) and have an entropy of 0 (i.e., \( -\log 1 \)). The user and files keys each appear in one record (e.g., \( P_{user} = 0.5 \)) and have entropies of 0.35 (i.e., \( -\frac{1}{2} \log \frac{1}{2} \)). Combined, the total Key-Space entropy of the records is \( E_K = 0.70 \approx 2 \cdot 0 + 2 \cdot 0.35 \).

5.2 The Similar Types Constraint

As variation increases between the field types of an object, the resulting collection-like schema becomes both less precise and less concise. This suggests that we would like an entropy measure similar to key-space entropy for types, with a higher “type entropy” marking objects as more tuple-like. However, with optional fields and multiple levels of collection nesting, the number of distinct nested types grows exponentially and computing a type entropy score becomes prohibitively expensive. Instead, JXPLAIN adopts a constraint based on the following type similarity rule:

\[
\begin{align*}
\tau_1 \approx \tau_2 \iff \begin{cases}
\text{true} & \text{if } \tau_1 = \text{null} \text{ or } \tau_2 = \text{null} \\
\tau_1 = \tau_2 & \text{if } \text{kind}(\tau_1) \in \{ B, R, S \} \\
\forall i : \tau_1.i \approx \tau_2.i & \text{with } i \in \text{keys}(\tau_1) \cap \text{keys}(\tau_2)
\end{cases}
\end{align*}
\]

Nulls are similar to anything, while primitive types are similar only to themselves and null. Like-kind complex types are similar if nested elements at matching keys or positions are also similar. For a collection of objects to be collection-like, we require pairwise similarity for all objects in the collection. Similarity is not transitive: two objects with a dissimilar field can be similar to an object omitting this field. However, similarity is subtransitive: If \( \tau_1 \approx \tau_2 \) and \( (\tau_1 \cup \tau_2) \approx \tau_3 \), then \( \tau_1 \approx \tau_2 \approx \tau_3 \). A linear scan can accumulate a maximal object unioning all fields encountered, while checking for similarity to this maximal object, and by extension its components.

5.3 Differentiating Tuples and Collections

The input objects are considered tuples if (i) two nested values have dissimilar types, or (ii) the key-space entropy is below a threshold. Otherwise, the objects are considered to be nested collections. This process is summarized in Algorithm 5.

Selecting a Key-Space Entropy Threshold. While a threshold for marking a set of types as collection-like is required, we found that the precise value of this threshold is relatively unimportant for two reasons: First, we found optional fields to be rare. Second, Figure 4 shows the distribution of Key-Space entropy in the Yelp dataset introduced in Section 7: Each point is one complex-kind path with self-similar nested elements. Note the multi-modal distribution: Nearly all potential collections have a near-zero, or a very high entropy. Other datasets were similar. This suggests that the heuristic’s reliability is minimally sensitive to the precise threshold selected. Our experiments arbitrarily use a threshold of 1.

5.4 Entropy For Arrays

Like JSON objects, arrays allow nesting. Although used almost exclusively to represent nested collections, specific use-cases treat arrays more like tuples. For example, the Twitter API encodes coordinates as 2-element arrays (i.e., latitude and longitude with type
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 \}; \( E_T = E_K = 0 \)
2: for all \( r \in \mathcal{R} \) do
3: RecordCount += 1; KindCount = \{ * : 0 \}
4: for all key in keys(\( r \)) do
5: KeyCnt[key] += 1; KindCount[kind(\( r \).key)] += 1
6: for all (kind : count) in KindCount do
7: \( E_T += \frac{\text{count}}{\text{count}} \log \left( \frac{\text{count}}{|\text{keys}(T)|} \right) \)
8: if \( E_T > 0 \) then return Tuple
9: for all (key : count) in KeyCount do
10: \( E_K += \frac{\text{count}}{\text{count}} \log \left( \frac{\text{count}}{|\text{recordCount}|} \right) \)
11: if \( E_K \leq 1 \) then return Tuple \( \ast \) Threshold value
12: else return Collection

\[ \{ R, R \} \), while other applications encode rows of a CSV file as fixed-width arrays [22]. The problem of distinguishing collection-like and tuple-like arrays is analogous to objects. The type constraint maps naturally to arrays, while key-space entropy is computed from the distribution of array lengths \( R \_T \), rather than the set of keys.

\[ E_K = - \sum_{i} P_i \log P_i = \frac{\left| \{ i \mid i \in [N], i = M_i \} \right|}{N} \]

6 MULTI-ENTITY COLLECTIONS

The GitHub events protocol defines 49 event schemas on a human-curated documentation page\(^2\). By contrast, existing schema discovery produces one big schema with fields from all events. Nearly every field is optional, making the schema imprecise. Our aim is to recover a set of core entities with distinct schemas. We characterize entity discovery as follows: Assume some “ground-truth” JSON schema \( S_G \) that is a union of \( K \) ObjectTuple elements (\( K > 0 \)). We are given sets of keys (i.e., key sets) of \( N \) object-kinded records sampled from these entities, with the obvious extension to arrays.

\( \{ k_{1,1}, \ldots, k_{1,M_1} \}, \ldots, \{ k_{N,1}, \ldots, k_{N,M_N} \} \)

The entity discovery problem is to find approximately \( K \) schemas \( S \) that union to replicate \( S_G \) as closely as possible.

6.1 Entity Discovery

The primary challenge of entity discovery arises from a single source: A field present in only one of two training objects could be interpreted as (i) an optional field of a single entity, or (ii) a distinguishing feature separating two distinct entities.

Example 8. Both of these schemas admit every record in Figure 1.

\( S_1 = \{ \{ ts: R, event: S, user: \{ \ldots \} \}, \{ ts: R, event: S, files: [S] \} \} \)

\( S_2 = \{ \{ ts: R, event: S, user: \{ \ldots \}, files: [S] \} \} \)

A limitation of manual schema curation. At time of writing, this page was out of date.

\( S_1 \) encodes two distinct entities (one for each event type), while \( S_2 \) uses a single entity with optional fields.

At one extreme \( L \)-reduction considers each record to be a distinct entity (i.e., \( K = N \), modulo duplicates). This approach is not only verbose, but does not generalize beyond the specific schemas in the training data. Taking the latter approach as a default we can assume that all training objects are a single entity (i.e., \( K = 1 \)). This solution is concise, but over-generalizes, admitting many more types than the original schema would. Our challenge is to balance between these two extremes (i.e., \( 1 \leq K \leq N \)).

A natural solution is clustering similar records together through a classical algorithm like k-means. However, classical clustering presents two challenges. First, \( K \) may not be known ahead of time. Even if \( K \) is known, a more subtle issue arises when the entities of \( S_G \) have different numbers of attributes. This asymmetry poses a problem for classic measures of similarity where each field is weighted equally (e.g., the Jaccard index between key-sets).

Example 9. For example, the Yelp photos table has 4 mandatory fields, while the business table has 20, most of which are optional. Both photos and business share exactly one mandatory field: business\_id. While a business record missing 17 optional fields shares only 1 field in common with a photos record. Between them there are 6 distinct fields (1 shared, 3 photo-only, 2 business-only). Thus, the Jaccard index considers this business more similar to a photo (\( \frac{1}{6} = 0.167 \)) than to a business with all 20 attributes (\( \frac{1}{20} = 0.05 \)).

Bi-Clustering. Ideally, we could derive a distance measure that accounts for entity size, for example by reducing the weights of features in large entities. However to do this, we need to know the entities, which rather defeats the purpose. Thus, entity detection is an example of a bi-clustering problem [26], a problem where we need to simultaneously group records by feature co-occurrence, while also grouping features by co-appearance in records.

6.2 Bimax

JXPLAIN adopts a simple, yet surprisingly robust and commonly used bi-clustering technique called Bimax [24]. Originally targeted at gene expression analysis, Bimax eschews distance measures in favor of a simpler greedy subset/superset clustering strategy. Algorithm 6 summarizes the original Bimax algorithm, which greedily selects the largest key (\( k_{max} \); line 4) and partitions the remaining records into three groups: (i) strict subsets of \( k_{max} (K_{sub}) \), (ii) key-sets overlapping with \( k_{max} (K_{overlap}) \), and (iii) key-sets disjoint with \( k_{max} (K_{disjoint}) \). Preserving the original order within each partition, partitions are rearranged as \( K_{sub}, K_{overlap}, K_{disjoint} \), and the algorithm iteratively sorts the latter two. Although we omit field order here, it is sorted analogously.

The resulting sort order places subsets (\( K_{sub} \)) closest, partially overlapping sets (\( K_{overlap} \)) slightly further away, and fully disjoint subsets (\( K_{disjoint} \)) furthest away. The Bimax algorithm has several compelling advantages. First, subset relationships are independent of the size of each entity, mitigating the entity size skew problem. Second, the algorithm does not need to know the number of entities upfront, as it simply sorts records to put more similar records closer to one another. However, as we need the results clustered into entities, Bimax cannot be applied directly. A naive Bimax-inspired
clustering algorithm (Algorithm 7) builds entities out of the $K_{sub}$ sets constructed by the Bimax function, returning every such set as one cluster.

Unfortunately, this naive algorithm has a limitation: optional fields. Each cluster is seeded from a maximal record (of which every current entity (line 10) and continues with the next candidate.

Algorithm 6 Bimax

In: $K$: a list of sets of keys.
Out: $K$ re-ordered with similar key-sets nearby.
1: $K$ ← $K$[i]
2: $K_{sub} = \{ k | k \subseteq k_{max}, k \in K_i \}$
3: $K_{disjoint} = \{ k | k \cap k_{max} = \emptyset, k \in K_i \}$
4: $K_{overlap} = K_i - K_{sub} - K_{disjoint}$
5: Sort $K[i, \ldots, |K|]$ on $K_{sub} < K_{overlap} < K_{disjoint}$  
   » Preserve existing order within each set
   
   
   

Algorithm 7 Bimax-Naive

In: $K$: an ordered list of key-sets.
Out: $K_{naive}$: A set of key-set clusters.
1: $K$ ← $K$ in descending order of key-set size.
2: $i$ ← 1
3: while $i < |K|$ do
4:   $k_{max} ← K[i]$
5:   $K'_i = \{ K[j] | i \leq j \leq |K| \}$
6:   $K_{sub} = \{ k | k \subseteq k_{max}, k \in K'_i \}$
7:   $K_{disjoint} = \{ k | k \cap k_{max} = \emptyset, k \in K'_i \}$
8:   $K_{overlap} = K_i - K_{sub} - K_{disjoint}$
9:   Sort $K[i, \ldots, |K|]$ on $K_{sub} < K_{overlap} < K_{disjoint}$  
   » Preserve existing order within each set
10: $i$ ← $i + |K_{sub}|$

6.3 Greedy Merge

To avoid this blowup in the number of records required, we need a way to coalesce clusters together. Simply linking entities that share keys is insufficient, as a small number of fields (e.g., foreign keys) may be shared by multiple entities. JXPLAIN adopts a simple greedy heuristic summarized in Algorithm 8.

Proceeding in reverse insertion order (i.e., smallest-first), the algorithm iteratively selects candidate entities $K_{cand}$, each with maximal element $k_{cand}$. The algorithm then attempts to find a minimal set-cover for the maximal element among the maximal elements of the remaining entities (line 4). If such a cover exists, the algorithm removes the covering entities from further consideration (line 6), adds them to the candidate entity (line 7), synthesizes a new maximal element for the resulting set (line 8), and attempts to repeat the process. If no cover exists, the algorithm emits the current entity (line 10) and continues with the next candidate.

Example 11. Consider 4 entities discovered by Bimax-Naive over keys $A, B, C, D, E$, with maximal elements:

$$K_{naive} = \{ E_1 : \{ A, B, E \} \ E_2 : \{ B, C, E \} \ E_3 : \{ C, D, E \} \ E_4 : \{ B, D \} \}$$

The algorithm begins with the final (and smallest) entity $E_4$. The union of the maximal elements of $E_2$ and $E_3$ is a superset of $E_4$’s maximal element, so GreedyMerge links all three into a new candidate entity with (synthesized) maximal element $\{ B, C, D, E \}$ and removes $E_2$ and $E_3$ from further consideration. The only remaining entity, $E_1$, can not form a set cover over the joint $\{ E_2, E_3, E_4 \}$ entity. The final result is thus two entities: $E_1$ and the merged $\{ E_2, E_3, E_4 \}$ entity.

As an alternative view of GreedyMerge, consider an undirected graph with one node for each entity emitted by Bimax-Naive and one edge for every pair of nodes that share a field. Intuitively, optional fields manifest in this graph as regions of densely interconnected nodes. GreedyMerge collapses dense regions by iteratively identifying cycles and collapsing each into a single node. The critical feature of the algorithm is the order in which cycles are collapsed: in reverse order of discovery by the Bimax algorithm. This (i) ensures that entities are linked together with more similar nodes, since the Bimax order places similar entities together; and (ii) prioritizes merges of smaller entities.
We now evaluate Jxplain record open dataset of per-doctor Pharmaceutical event stream [15] collected over a period of 330 days, (ii) a 240,000 GitHub synthetic dataset: (i) A 3 million record sample of the experiments validate the following claims: Jxplain proxy for precision, in lieu of ground truth). As a secondary concern, generalizes beyond the example data from which it was derived each system for data validation: (i) How well the discovered schema (ii) Jxplain reduces, because it is a close analog to industry standard tech- 6.4 Implementation This entity discovery process needs to be run once on each tuple- types pass the data, a preprocessing step first compacts the dataset into a feature vector encoding that encodes the set of paths appearing in each record. Feature vector storage is flexible: Sparse feature vectors require less computation and combine opera- tions from Spark. Dense feature vectors can be faster and reduce memory overhead for Json with many mandatory fields. Jxplain defaults to a sparse encoding. The preprocessing step iterates over each record. For the root collection, it constructs a feature vector consisting of all paths in the record. For all other object-kinded collections, it unnests the collected objects and constructs feature vectors for the paths below each. The final result of the preprocessing step is a set of feature vectors for each object-kinded collection discovered in step 1. We observe that nested collections significantly increase the number of distinct feature vectors in each of their parents. As a further optimization, we modified the preprocessing step to retain only paths contained in an outer collection, but not in any collection nested within. Figure 5 illustrates the memory savings of removing nested collection features in the Yelp Dataset (see Section 7), while in the the Pharmaceutical dataset (also see Section 7) nearly all structural complexity arises from the nested collection, and this optimization reduces memory requirements to nearly nothing. Algorithms 7 and 8 output a set of feature vector sets that can be used to partition input types by entity.

7 EXPERIMENTS

We now evaluate Jxplain against the $\mathcal{K}$-reduction schema dis- covery algorithm proposed by Baazizi et. al. [5, 7]. We chose $\mathcal{K}$- reduction, because it is a close analog to industry standard tech- niques for schema discovery like Spark’s Json data source and Oracle’s Json Data Guides.

Our primary interest is in the utility of schemas extracted by each system for data validation: (i) How well the discovered schema generalizes beyond the example data from which it was derived (i.e., recall), and (ii) How few types the discovered schema admits (a proxy for precision, in lieu of ground truth). As a secondary concern, we are also interested in the runtime overheads of the (admittedly more complex) Jxplain schema discovery process. Concretely, our experiments validate the following claims:

(i) Jxplain produces schemas that are significantly more precise (i.e., admit fewer types) than $\mathcal{K}$-reduction, …

(ii) …while not incorrectly rejecting types that are legitimately part of the schema,

(iii) A clustering strategy based on Bimax bi-clustering is preferable to a standard technique like k-means,

(iv) The merge step described in Section 6 is critical for creating compact schemas, and

(v) The overhead of the additional steps required by Jxplain is not prohibitive.

Experiments were run over twelve real-world datasets, and one synthetic dataset: (i) A 3 million record sample of the GitHub event stream [15] collected over a period of 330 days, (ii) a 240,000 record open dataset of per-doctor Pharmaceutical prescription statistics [25], (iii) 800,000 Twitter tweets, (iv) 150,000 events taken from a Matrix Synapse server [19], (v) an archived list of New York Times (NYT) articles from 2019, (vi) a JSON dump from Wikidata of 1.7 million Wikipedia articles, (vii-xii) and the six individual schemas of the 7.5 million record Yelp Open Dataset [35]. For each dataset, we test on a 1%, 10%, 50%, and 90%-uniform random sample of the data. We reserve 10% of the data as a testing set.

We compare four algorithms: (i) $\mathcal{K}$-reduce: Baazizi et. al.’s $\mathcal{K}$- reduction, representing the state-of-the-art in schema extraction, (ii) Bimax Naive: A single pass of Jxplain (Algorithm 4) with the naive adaptation of Bimax (Algorithm 7), (iii) Bimax-Merge: A single pass of Jxplain with the Bimax Merge algorithm (Algorithm 8), and (iv) $L$-reduce: Baazizi et. al.’s $L$-reduction, a trivial schema extractor that accepts only exact types encountered in the input.

The GitHub dataset has a large number of entities of wildly varying sizes4, and its complex nesting structure creates extremely high memory requirements from both extractors. The Pharmaceutical dataset is the smallest, but also contains a collection-like object with 2397 distinct keys. This results in the largest number of distinct types across all of our datasets; Nearly every record has a unique type. Our Twitter dataset is a collection of 900 thou- sand tweet objects, containing the recursive schemas for retweets, deleted tweets, and quoted tweets, as well as a multitude of object arrays, and geo type tuple arrays. NYT contains all 70 thousand 2019 articles archived by the New York Times4. Wikidata is a dump of 1.7 million Wikipedia articles. These records closely resemble HTML and XML, with large and deeply nested arrays of objects. Additionally each Wikidata entity participates in their “Linked Data Interface” [36], where each entity attribute is represented as an integer key for reference linking. Finally, the Synapse dataset is the events table from a multi-year deployment of the Matrix Synapse open source chat server [19]. Matrix follows a complex state management protocol and this table is an immutable history of all state update events, including what appear to be 36 revisions to the protocol’s Json schema over the deployment period. The Yelp Business dataset makes extensive use of optional fields, nested collections, and soft functional dependency relationships, such as hair salon attributes having extremely high positive correlation with the by_appointment attribute. Additionally, Yelp’s check in ta- ble contains a multiply nested collection of checkins per hour of the week, with an outer object containing keys for each day of the week and inner objects containing keys for each hour of the day. Leaf values contain checkin counts, and hours or days with no checkins are omitted. This is analogous to a pivot table with two indexes: day and hour, with checkins as a value. This high variability poses a significant challenge while attempting to use common clustering algorithms, and motivated our use of Bimax. Finally, we create a synthetic Yelp-Merged dataset by combining the six schemas of the Yelp open dataset. This is a useful test of the Bimax-Merge algorithm, as (i) its 6 tables give us a well-defined ground truth for entity clustering, (ii) it contains attributes with common name collisions such as “name” that are not intended to

--

4The official documentation describes 49 event types, of which our trace contains 10, due to some events being used internally (which is unlisted) or exceedingly rare

4NYT is provided as the payload of a small number of Json records, each nested in a Json array. Our experiments combine the array contents into one root collection.
be shared between tables, and (iii) all entities can be joined through three foreign key fields, none of which appear in all entities.

All experiments were run using Apache Spark 2.3.4 and Scala version 2.11.8. Runtime testing was performed on 4x20-core 2.40-GHz Intel Xeon E7 processors with 1 TB of RAM and running on CentOS-7 linux. All results shown are the result of 5 trials with training/testing data uniformly sampled from the source data for each trial. To evaluate K-reduce, we obtained a binary (JAR) implementation of K-reduction from Baazizi et. al. Results shown are for this implementation with the following two caveats. First, during experimentation, we observed that schemas from the binary release diverged from the reference paper [5], producing schemas with some redundant union types. To ensure a fair performance/validation comparison, we added a post-processing step to reduce redundancy in several generated schemas. Second, the binary release of K-reduction timed out while processing the Pharmaceutical dataset; we omit performance numbers and evaluate recall and schema entropy on an equivalent, manually derived schema.

### 7.1 Recall

We evaluate claim (ii) by reserving a uniform 10% sample of the data as a testing set, and generate schemas from 1%, 10%, 50%, and 90%-uniformly sampled subsets of each dataset. Table 1 shows the fraction of the records in the testing set accepted by the generated schema. The table omits datasets where JXPplain and K-reduction both produce perfect schemas with a 1% sample. On the remaining datasets, even with only a 1% training set, schemas generated by JXPplain accept nearly every row from the testing set. False negatives in the 1% test result are almost exclusively optional attributes that (i) do not appear in the training set, or (ii) are present by chance in every record of the training set making them (falsely) appear to be mandatory. By a 10% sample the overwhelming majority of exceptions are accounted for in the generated schema. We note two particular outliers: JXPplain has better recall on both the Pharmaceutical and Synapse datasets, particularly with smaller sample sizes. As noted above, the Pharmaceutical dataset is dominated by a large collection-like object mapping medications to prescription counts. Even on the 1% sample, JXPplain correctly identifies this as a collection, which in turn allows it to generalize the schema to drugs not in the 1% sample. The Synapse dataset illustrates a similar problem, as many event records contain a signatures field of the following form:

```python
"signatures": { <url>: { <key>: <signature> } }
```

As with the Pharmaceutical dataset, JXPplain correctly identifies this structure as a two-level nested collection, allowing it to generalize to servers and keys not present in the sample.
Table 2: Schema Entropy: The number (log 2) of types accepted by the generated schema († ran out of resources).

### Dataset

<table>
<thead>
<tr>
<th></th>
<th>K-reduce</th>
<th>Bimax-Merge</th>
<th>Bimax-Naive</th>
<th>(L)-reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
<td>std</td>
<td>mean</td>
<td>std</td>
</tr>
<tr>
<td><strong>Yelp</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Merged</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1%</td>
<td>21.21</td>
<td>0.71</td>
<td>14.70</td>
<td>1.02</td>
</tr>
<tr>
<td>10%</td>
<td>21.13</td>
<td>0.00</td>
<td>17.94</td>
<td>0.00</td>
</tr>
<tr>
<td>50%</td>
<td>23.46</td>
<td>0.06</td>
<td>18.94</td>
<td>0.00</td>
</tr>
<tr>
<td>90%</td>
<td>24.88</td>
<td>19.86</td>
<td>175.34</td>
<td>11.61</td>
</tr>
<tr>
<td><strong>Naive</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1%</td>
<td>48.10</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>10%</td>
<td>49.61</td>
<td>0.07</td>
<td>49.61</td>
<td>1.07</td>
</tr>
<tr>
<td>50%</td>
<td>50.40</td>
<td>16.42</td>
<td>50.40</td>
<td>15.01</td>
</tr>
<tr>
<td>90%</td>
<td>52.20</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td><strong>Github</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1%</td>
<td>10.56</td>
<td>212.72</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>10%</td>
<td>10.48</td>
<td>2575.55</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>50%</td>
<td>10.48</td>
<td>97.14</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>90%</td>
<td>10.48</td>
<td>1801.80</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

### Health Insurance

<table>
<thead>
<tr>
<th></th>
<th>K-reduce</th>
<th>Bimax-Merge</th>
<th>Bimax-Naive</th>
<th>(L)-reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
<td>std</td>
<td>mean</td>
<td>std</td>
</tr>
<tr>
<td><strong>Yelp</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Merged</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1%</td>
<td>1199.20</td>
<td>8.28</td>
<td>1199.20</td>
<td>8.28</td>
</tr>
<tr>
<td>10%</td>
<td>1199.20</td>
<td>8.28</td>
<td>1199.20</td>
<td>8.28</td>
</tr>
<tr>
<td>50%</td>
<td>1199.20</td>
<td>8.28</td>
<td>1199.20</td>
<td>8.28</td>
</tr>
<tr>
<td><strong>Naive</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1%</td>
<td>1199.20</td>
<td>8.28</td>
<td>1199.20</td>
<td>8.28</td>
</tr>
<tr>
<td>10%</td>
<td>1199.20</td>
<td>8.28</td>
<td>1199.20</td>
<td>8.28</td>
</tr>
<tr>
<td>50%</td>
<td>1199.20</td>
<td>8.28</td>
<td>1199.20</td>
<td>8.28</td>
</tr>
</tbody>
</table>

### Bimax-Merge vs. Bimax-Naive

<table>
<thead>
<tr>
<th></th>
<th>K-reduce</th>
<th>Bimax-Merge</th>
<th>Bimax-Naive</th>
<th>(L)-reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
<td>std</td>
<td>mean</td>
<td>std</td>
</tr>
<tr>
<td><strong>Yelp</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Merged</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1%</td>
<td>280.41</td>
<td>1.07</td>
<td>175.00</td>
<td>0.98</td>
</tr>
<tr>
<td>10%</td>
<td>281.78</td>
<td>1.07</td>
<td>175.00</td>
<td>0.98</td>
</tr>
<tr>
<td>50%</td>
<td>281.78</td>
<td>1.07</td>
<td>175.00</td>
<td>0.98</td>
</tr>
<tr>
<td>90%</td>
<td>281.78</td>
<td>1.07</td>
<td>175.00</td>
<td>0.98</td>
</tr>
<tr>
<td><strong>Naive</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1%</td>
<td>1199.20</td>
<td>1.07</td>
<td>175.00</td>
<td>0.98</td>
</tr>
<tr>
<td>10%</td>
<td>1199.20</td>
<td>1.07</td>
<td>175.00</td>
<td>0.98</td>
</tr>
<tr>
<td>50%</td>
<td>1199.20</td>
<td>1.07</td>
<td>175.00</td>
<td>0.98</td>
</tr>
<tr>
<td>90%</td>
<td>1199.20</td>
<td>1.07</td>
<td>175.00</td>
<td>0.98</td>
</tr>
</tbody>
</table>

### Conclusion

7.3 Entity Detection

Clustering Accuracy. We evaluate claim (iii) based on the two datasets (Yelp-Merged and Github) for which ground truth information for entities is available or inferable. The synthetic Yelp-Merged has ground-truth by definition, while the Github event trace contains a “type” attribute with 14 distinct types that we use as a ground truth. Concretely, we compare Bimax-Merge against K-reduce, and clustering using k-means. For k-means, we used the ground-truth value of k (which would not be available in practice) and Euclidean distance. We compute the symmetric set difference for each pair \((S_i, G_j)\) where each \(S_i\) is the schema derived for one cluster, and each \(G_j\) is the schema for one ground-truth entity: \(D(S_i, G_j) = |S_i - G_j| + |G_j - S_i|\). Note that K-reduce does not perform entity detection and so produces only one cluster.

Table 3 reports, for each ground-truth entity, the difference from the most similar cluster (i.e., the cluster that corresponds to the ground truth entity). Smaller values are better. Observe that for k-means, only a handful of entities do very well: this clustering algorithm tends to create multiple clusters for entities with many attributes, while starving smaller ones (even with an ideal k value). As expected, K-reduce over-describes each entity, while not describing any single entity well. Bimax-Merge has a near perfect description of every individual entity; A few minor errors arise in the four Github entities whose fields are a subset of another entity.

Conciseness. Table 4 illustrates the effectiveness of the Bimax-Merge optimization of the Bimax-Naive algorithm (Algorithms 7 and 8 in Section 6, respectively). The table supports claim (iv) by comparing the number of output entities identified by both the optimized and unoptimized algorithms. For the purposes of this experiment, we disable nested collection detection for the Pharmaceutical dataset and consider only entities at the root level of the Github and Yelp datasets (ignoring nested collections). The merge heuristic has minimal impact while selecting entities for the Github schema, but significantly reduces entities in both Yelp-Merged and Pharmaceutical datasets due to optional fields. Github entities have few optional fields. Conversely many fields in the Yelp dataset are optional, as are the fields of the pharmaceutical dataset with collection detection disabled. For an entity with optional fields, the Bimax-Naive algorithm needs to see at least one object with all optional fields present; Bimax-Merge lifts this requirement. There is a small error that arises on the Yelp dataset due to a soft functional dependency that is so rarely violated it is possible to miss even when training on 90% of the data. As a result, K-reduce needs to do more work to create a more precise schema, so we do not expect it to outperform K-reduce; We aim here simply to assess the added overhead. Table 5 shows schema discovery performance, varying the proportion of each dataset used in order to generate the schema to show scaling behavior. Performance scales linearly for both extractors. JXPLAIN...
### Table 3: Minimum symmetric difference from ground-truth schema (lower is better)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Yelp-Review</th>
<th>Yelp-Merged</th>
<th>Yelp-Checkin</th>
<th>Yelp-Photos</th>
<th>Yelp-Tip</th>
<th>Yelp-User</th>
<th>Github</th>
</tr>
</thead>
<tbody>
<tr>
<td>‐reduce</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
</tr>
<tr>
<td>Bimax-Naive</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
</tr>
<tr>
<td>Bimax-Merge</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
</tr>
</tbody>
</table>

### Table 4: Entity predictions with 90% training data († ‐reduce and Bimax-Naive ran out of resources on Wikidata)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Yelp-Review</th>
<th>Yelp-Merged</th>
<th>Yelp-Checkin</th>
<th>Yelp-Photos</th>
<th>Yelp-Tip</th>
<th>Yelp-User</th>
<th>Github</th>
</tr>
</thead>
<tbody>
<tr>
<td>‐reduce</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
</tr>
<tr>
<td>Bimax-Naive</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
</tr>
<tr>
<td>Bimax-Merge</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
</tr>
</tbody>
</table>

8 RELATED WORK

Over the past two decades, there have been numerous attempts at structure detection for semi-structured data. Each implementation aims at creating a summary schema that is concise, descriptive, prescriptive, generalizable, and interpretable. The closest work to ours is an algebraic exploration of scalable schema extraction by Baazizi et al. [5, 7]. They propose a grammar for concisely describing sets [7] and bags [5] of JSON types (on which we base our grammar in Section 2), and propose fusion operators that combine sentences in this grammar (i.e., schemas). The primary contribution of this work is to address the issue of scalability by ensuring that the merge operators are commutative and associative, admitting distributed execution through typical fan-in aggregation (implemented in Spark). Frozza et al. also present a similar approach [14]. Unfortunately, requiring the merge operation to be commutative and associative limits it to local-decision making: Schema properties like the number of entities (Section 6) or whether an object encodes a tuple or collection (Section 5) dictate the behavior of the fusion operator, but can not be inferred from just two types. Instead, the Baazizi algorithm asks users to completely define the behavior of the merge operator in a data-independent way. Jxplain can be viewed as an extension of the Baazizi algorithm that infers the “correct” merge operator for each path in a pre-processing step.

Entity discovery has been explored extensively in hierarchical [8–10, 13, 16, 17, 21, 23, 28, 29, 34], graph [2], and object-exchange model (OEM) [16] data. XML schema discovery in particular [8–10, 17, 21] has been explored extensively. However, solutions in this space rely heavily on the contextual signal provided by node labels, which are not available in JSON data. Additionally XML data models have no concept of arrays, relying on sibling nodes sharing a label. In practice this detection can be very fragile, particularly when node identifiers are ambiguous. Overall XML schemas impose a different set of data constraints than JSON. Notably, Bex et al. address ambiguous node identifiers in XML [9], a problem related to entity discovery in Jxplain. Their approach relies on the node’s ancestors and predecessors to disambiguate entity types, inferring a selector for each entity based on these factors, rather than based on the entity’s attributes as in Bimax-Merge.

A range of machine-learning-based techniques have been proposed for discovering such linkages [23, 28, 29, 34]. However, as we
Table 5: Runtime (milliseconds) by discovery algorithm and training set size. (†: \(K\)-reduce times out on the Pharma dataset)

discuss in Section 6, such approaches are vulnerable to skew in entity size, and the inherent ambiguity of the entity discovery problem. \textsc{jxplain} adopts a more robust approach based subset relationships and field overlap. We note one approach [2] in particular uses a clustering mechanism similar to Bimax, but relies on information loss over data values rather than Bimax’s use of field-set-containment. Data values make this approach more expressive, but also limit its scalability. A related challenge that our approach does not (yet) address is co-reference detection [30]: Identifying entities that appear at multiple paths (e.g., Twitter’s API can include user information for the user making a post, as well as any users tagged in the post).

Alternative approaches to schema discovery rely on functional dependencies [13, 20, 37] between nodes in graph [20], XML [37], or JSON [13] data. These techniques attempt to discover functional dependencies [1] between fields (out-edges, children, descendents); each set of fields related by a functional dependency is treated as an independent entity. A key limitation in these approaches is that they still need to differentiate between tuple- and collection-like reference/nesting structures. Like the extractor of Baazizi et al. [13], each scheme makes upfront assumptions about which structural elements (e.g., JSON arrays) encode collections. Adjacent recent work has explored utilizing Human-in-the-Loop schema inference and parameterization [4, 6]. Our system offers improvements over this model by automating away many of these decisions through alternate entropy and schema signals. This heuristics may often be reliable, but present serious performance limitations on corner cases like the pharmaceutical dataset or geographical coordinate arrays. A further limitation is that these approaches are often designed to operate on flattened, relational representations (with [37] a notable exception) of the complex structure. Existing nesting structures are removed early, losing a significant source of signal about the intended schema structure. However, functional-dependency-based approaches to schema discovery are orthogonal to our own entity discovery strategies and can, in principle be integrated into \textsc{jxplain}.

Our work is partially motivated by ensuring up-to-date documentation for web APIs. A related, orthogonal issue is querying out-of-date schemas. Snodgrass et al. propose “neighborhood queries” [27] to make XPath queries resilient to small schema changes, while the Prism Workbench [12] encodes prior versions of a relational schema as views. GraphQL [33] is increasingly being used as an API for access to structured data resources, and addresses the same problem by allowing API consumers-specific schemas defined as GraphQL queries. This approach still requires a stable schema for graph entities, but does avoid schema changes made purely for optimization purposes or to facilitate certain API requirements.

9 CONCLUSION

When JSON data evolves beyond encoding glorified csv documents, JSON’s support for tuple and collection nesting make maintaining schemas incredibly difficult. Relying on users to create, maintain, and publish usable documentation or precise schemas is often wishful thinking [22]. This, in turn, creates opportunities for erroneous data to sneak through systems. Existing tools for automated schema discovery provide only coarse-grained, permissive schema summaries of a dataset. We have presented \textsc{jxplain}, a JSON schema discovery system that adopts a more nuanced, ambiguity-aware approach, aiming to create tight (high-precision), descriptive (high-recall) schemas without overwhelming users. In contrast to prior systems for similar tasks: \textsc{jxplain} avoids two assumptions about how JSON records are created: (i) Which nested structures encode nested collections, and (ii) How many entities appear in a collection of JSON objects (nested or not). To avoid these assumptions \textsc{jxplain} adds several pre-processing passes to typical schema discovery. Although these passes add a non-trivial overhead to the extraction process, the resulting schemas are tighter and more compact, especially for complex JSON data models, reducing the amount of manual tuning required to refine them.

Future Work. In addition to the pre-processing stages we introduced in this paper, further optimizations are possible. For example, many orthogonal schemes [13] use data values and specifically functional dependencies between data values to recover different families of entity structures. Additionally, we plan to explore the use of \textsc{jxplain} in filesystem directory structures extraction. Finally, schemas extracted by \textsc{jxplain} can be used to generate read-optimized data layouts.

10 ACKNOWLEDGEMENTS

This work is supported in part by NSF Award IIS-1750460 and Awards from Oracle University Relations. The conclusions and opinions in this work are solely those of the authors and do not represent the views of the National Science Foundation or the Oracle Corporation.
REFERENCES


