Reducing Ambiguity in Json Schema Discovery

Anonymous Author(s)

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.

ACM Reference Format:
Anonymous Author(s). 2018. Reducing Ambiguity in Json Schema Discovery. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

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, offers 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.

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` *feld 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"]}
```

Figure 1: Example Json Data

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

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 tuples. These assumptions are sufficient for simple, homogeneous Json collections, but break down on the complex nesting structures appearing in more heterogenous 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 algorithm produces a generated schema with high *recall*: all records in the ground truth schema should be part of the generated schema, even if they do not appear in the sample. For schema validation, it is also critical that the algorithm produce a schema with high

Woodstock '18, June 03–05, 2018, Woodstock, NY
2018. ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

$$\tau \;:=\; \mathbb{B} \mid \mathbb{R} \mid \mathbb{S} \mid \textbf{null} \mid [\ \tau_1,\ \ldots,\ \tau_N\ ] \mid \{\ k_1 : \tau_1,\ \ldots,\ k_N : \tau_N\ \}$$

Figure 2: Json's Typesystem

precision: records not in the ground truth schema should not be part of the generated schema. Ideally, an algorithm would also produce a generated schema with a concise description. In this section, we adapt the notation of Baazizi et. al. [7] to allow us to define precision and recall more precisely.

Data values in Json are weakly typed and may be either primitive or complex. As summarized in Figure 2, a primitive Json value is a boolean value ($\mathbb{B}$), a numeric value ($\mathbb{R}$), a string value ($\mathbb{S}$), or the value `null` (**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 τ to be τ if it is primitive, or the symbol $\mathcal{O}$ or $\mathcal{A}$ if τ is an object or array respectively:

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

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

$$\{\ \texttt{ts} : \mathbb{R},\ \texttt{event} : \mathbb{S},\ \texttt{user} : \{\ \texttt{name} : \mathbb{S},\ \texttt{geo} : [\ \mathbb{R}, \mathbb{R}\ ]\ \}\ \}$$

The kind of the record is $\mathcal{O}$. The field `event` *has kind $\mathbb{S}$.*

If τ is an object (resp., array), we write $\texttt{keys}(\tau)$ 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 τ. Denote by $\mathbf{p}$ a *path*, a sequence of keys $\mathbf{p}_1, \ldots, \mathbf{p}_n$.

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

When clear from context, we abuse notation using types (e.g., τ) 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.

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

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

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

$$[\ \mathcal{S}_1, \ldots, \mathcal{S}_N\ ] \triangleq \{\ \{\ \tau_1, \ldots, \tau_N\ \} \mid \tau_i \in \mathcal{S}_i\ \}$$

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

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

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

2.1 Schema Discovery

We are given a collection of records with N types $\mathcal{R} = \{\ \tau_1, \ldots, \tau_N\ \}$ drawn from some hidden ground truth schema $\mathcal{S}_\mathcal{G}$. The schema discovery problem is to "merge" these types into a new schema definition (denoted $\texttt{merge}\ (\mathcal{R})$) that closely approximates $\mathcal{S}_\mathcal{G}$. This derived schema should have high precision, admitting only ground truth types (i.e., $\frac{|\texttt{merge}(\mathcal{R}) - \mathcal{S}_\mathcal{G}|}{|\mathcal{S}_\mathcal{G} \cup \texttt{merge}(\mathcal{R})|} \approx 0$) and high recall, admitting all ground truth types (i.e., $\frac{|\mathcal{S}_\mathcal{G} - \texttt{merge}(\mathcal{R})|}{|\mathcal{S}_\mathcal{G} \cup \texttt{merge}(\mathcal{R})|} \approx 0$). 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 $\mathcal{S}_\mathcal{G}$. This is analogous to the $\mathcal{L}$-reduction of Baazizi et. al. [7]. In other words we define:

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

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.*

$$\{\ \{\ \texttt{ts} \to \mathbb{S},\ \texttt{event} \to \mathbb{S},\ \texttt{user} \to \{\ \texttt{geo} \to [\ \mathbb{R}, \mathbb{R}\ ], \texttt{name} \to \mathbb{S}\ \}\ \},\ \{\ \texttt{ts} \to \mathbb{S},\ \texttt{event} \to \mathbb{S},\ \texttt{files} \to [\ \mathbb{S}, \mathbb{S}\ ]\ \}\ \}$$

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

$$\texttt{merge}_\mathcal{A}\ (\mathcal{R}) \triangleq [\ \texttt{merge}\ (\{\ \tau_i \mid [\ \tau_1, \ldots, \tau_N\ ] \in \mathcal{R}, i \in [N]\ \})\ ]^*$$

Example 4. *The field* `files` *in Figure 1 would be merged into a collection of strings: $[\ \mathbb{S}\ ]^*$, because all of its elements have kind $\mathbb{S}$.*

Objects as Tuples. Json objects are commonly used to encode tuples. Accordingly, variation between objects in a collection is assumed to be the result of optional fields. Typically, fields appearing in all input objects are mandatory (with keys $\texttt{keys}_\forall(\mathcal{R})$), while fields appearing in only some are optional (with keys $\texttt{keys}_\exists(\mathcal{R})$).

$$\texttt{keys}_\forall(\mathcal{R}) \triangleq \bigcap_{\tau \in \mathcal{R}} \texttt{keys}(\tau) \qquad \texttt{keys}_\exists(\mathcal{R}) \triangleq \bigcup_{\tau \in \mathcal{R}} \texttt{keys}(\tau) - \texttt{keys}_\forall(\mathcal{R})$$

The merge operation groups nested field types by their key and recursively merges groups. Defining $\{\ k_1, \ldots, k_k\ \} \triangleq \texttt{keys}_\forall(\mathcal{R})$, $\{\ k_{k+1}, \ldots, k_N\ \} \triangleq \texttt{keys}_\exists(\mathcal{R})$, and $\mathcal{S}_i \triangleq \texttt{merge}\ (\{\ \tau.i \mid \tau \in \mathcal{R}\ \})$:

$$\texttt{merge}_\mathcal{O}\ (\mathcal{R}) \triangleq \{\ k_1 : \mathcal{S}_1\ \ldots\ k_k : \mathcal{S}_k,\quad k_{k+1}^? : \mathcal{S}_{k+1}\ \ldots\ k_N^? : \mathcal{S}_N\ \}$$

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 $\mathcal{K}$-reduction [5], defined formally as follows:

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

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:* $[\ \mathbb{R}, \mathbb{R}\ ]$.

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., $[\ \mathbb{R}\ ]^*$ instead of $[\ \mathbb{R}, \mathbb{R}\ ]$) 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:*

```
{"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, $\{\ * \rightarrow \mathbb{R}\ \}^*$*). We also observed collection-style objects in many other API datasets, including Yelp's* `checkins` *dataset:*

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

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

```
{"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 descendents share a schema. Furthermore, optional attributes can not describe new field names (e.g., new medications or user names) as they are added, actually 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:

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

```
{ "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:

```
{"multimedia": [
  {"legacy": [], ...},
  {"legacy": { "xlarge": "03Prose1-articleLarge.jpg",
               "xlargewidth": 600,
               "xlargeheight": 450}, ...},
  {"legacy": { "thumbnail": "03Prose1-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[1] captured by the following grammar $\mathcal{S}_\mathcal{J}$. Primitive types are explicit:

$$\mathcal{S}_\mathcal{J} := \mathbb{R} \mid \mathbb{S} \mid \mathbb{B} \mid \textbf{null}$$

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

$$\mid \textbf{ArrayTuple}(\mathcal{S}_\mathcal{J}, \mathcal{S}_\mathcal{J}, \ldots)$$

$$\mid \textbf{ObjectTuple}(k : \mathcal{S}_\mathcal{J},\ k : \mathcal{S}_\mathcal{J},\ \ldots,\ k^? : \mathcal{S}_\mathcal{J},\ k^? : \mathcal{S}_\mathcal{J},\ \ldots)$$

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

$$\mid \textbf{ArrayCollection}(\mathcal{S}_\mathcal{J}) \mid \textbf{ObjectCollection}(\mathcal{S}_\mathcal{J})$$

$$\mid \textbf{Union}(\mathcal{S}_\mathcal{J}, \mathcal{S}_\mathcal{J}, \ldots)$$

This grammar mirrors the schema shorthands given in Section 2 and its semantics follow naturally. We abuse notatation and use expressions in this grammar interchangeably with schemas.

As an example, Algorithm 1 implements $\mathcal{K}$-reduction as presented in Section 2. Via helper functions (Algorithms 2 and 3), array-kinded types are always intepreted as single-entity collections, while object-kinded types are always intepreted as tuples. Both helper functions are parameterized by a recursive merge heuristic, $\mathcal{K}$-reduction itself in this example. The central feature of $\mathcal{K}$-reduction is its distributivity over union [7]:

$$\textbf{merge\_K}(\mathcal{R}_1 \cup \mathcal{R}_2) = \textbf{merge\_K}(\textbf{merge\_K}(\mathcal{R}_1) \cup \textbf{merge\_K}(\mathcal{R}_2))$$

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

[1] https://json-schema.org/

Algorithm 1 **merge_K**($\mathcal{R}$)

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

1: **return** **Union**($\mathcal{R} \cap \{\ \mathbb{R}, \mathbb{S}, \mathbb{B}, \textbf{null}\ \}$,
merge_array_coll(**merge_K**, $\mathcal{R} \cap \mathcal{A}$),
merge_object_tuple(**merge_K**, $\mathcal{R} \cap \mathcal{O}$))

Algorithm 2 **merge_array_coll**(merge, $\mathcal{R}$)

In: merge: A recursive merge function
In: $\mathcal{R}$: A bag of array-kinded types
1: **return** **ArrayCollection**(merge({ $\tau.k \mid k \in \text{keys}(\tau), \tau \in \mathcal{R}$ }))

Algorithm 3 **merge_object_tuple**(merge, $\mathcal{R}$)

In: merge: A recursive merge function
In: $\mathcal{R}$: A bag of object-kinded types
1: $k_1, \ldots, k_k \leftarrow \text{keys}_\forall(\mathcal{R})$ $\quad k'_1, \ldots, k'_\ell \leftarrow \text{keys}_\exists(\mathcal{R})$
2: **return** **ObjectTuple**(
$k_1 \rightarrow \text{merge}(\{\ \tau.k_1 \mid \tau \in \mathcal{R}\ \}), \ldots,$
$k_k \rightarrow \text{merge}(\{\ \tau.k_k \mid \tau \in \mathcal{R}\ \}),$
$k'_1 \rightarrow \text{merge}(\{\ \tau.k_1 \mid k'_1 \in \text{keys}(\tau), \tau \in \mathcal{R}\ \}), \ldots,$
$k'_\ell \rightarrow \text{merge}(\{\ \tau.k'_\ell \mid k'_\ell \in \text{keys}(\tau), \tau \in \mathcal{R}\ \}))$

Algorithm 4 Jxplain ($\mathcal{R}$)

In: $\mathcal{R}$: A bag of types
1: $\mathcal{S}_\mathcal{A} \leftarrow \emptyset$ $\quad \mathcal{S}_O \leftarrow \emptyset$ $\quad A \leftarrow \mathcal{R} \cap \mathcal{A}$ $\quad O \leftarrow \mathcal{R} \cap \mathcal{O}$
2: **if** $|A| > 0$ **then**
3: **if** **is_collection**(A) **then**
4: $\mathcal{S}_\mathcal{A} \leftarrow$ **merge_array_coll**(Jxplain, A)
5: **else**
6: $A_1, \ldots, A_k \leftarrow$ **partition**(A)
7: $\mathcal{S}_\mathcal{A} \leftarrow$**Union**(**merge_array_tuple**(Jxplain, A_1), …, **merge_array_tuple**(Jxplain, A_k))
8: **if** $|O| > 0$ **then**
9: **if** **is_collection**(O) **then**
10: $\mathcal{S}_O \leftarrow$ **merge_object_coll**(Jxplain, A)
11: **else**
12: $O_1, \ldots, O_k \leftarrow$ **partition**(O)
13: $\mathcal{S}_O \leftarrow$**Union**(**merge_object_tuple**(Jxplain, O_1), …, **merge_object_tuple**(Jxplain, O_k))
14: **return** **Union**($\mathcal{R} \cap \{\ \mathbb{R}, \mathbb{S}, \mathbb{B}, \textbf{null}\ \}, \mathcal{S}_\mathcal{A}, \mathcal{S}_O$)

than the size of the input type bags, it is extremely amenable to distributed computation. Unfortunately, limiting ourselves to associative 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 Algorithm 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-kinded 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 (Algorithm 2 and its object analog). If the input bag's elements are tuples, Jxplain partitions the bag into individual entities and infers a schema for each individually (Algorithm 3 and its array analog).

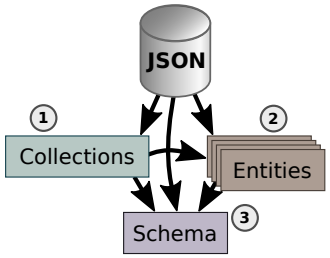

Figure 3: Stages of Extraction in Jxplain

4.2 Jxplain

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

Parallelization. We expect the two heuristics to need to see the entire input before producing an output, so the simplified Algorithm 4 is not an associative fold, and so not ammenable to distribution. Jxplain instead decouples these heuristics into separate computation 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 entities. 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 structures into either tuples or collections. Tuples bound the set of allowed 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 **ArrayTuple** 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

$\mathcal{K}$-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 $\mathcal{L}$-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-kinded types as inputs, before generalizing to arrays below.

$$\{ k_{1,1} : \tau_{1,1}, \ldots, k_{1,M_1} : \tau_{1,M_1} \}, \ \ldots, \ \{ k_{N,1} : \tau_{N,1}, \ldots, k_{N,M_N} : \tau_{N,M_N} \}$$

The goal is to mark the objects as (a) collection-like (i.e., **ObjectCollection**), 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., $\texttt{keys}(\tau)$) 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. Notably, 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:

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

For each key, JXPLAIN computes the probability that an object selected uniformly at random contains the key (P_k). The resulting Key-Space entropy ($\mathcal{E}_{\mathcal{K}}$) 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.,* $P_{\texttt{ts}} = 1$*) and have an entropy of* 0 *(*$= -1 \log 1$*). The* `user` *and* `files` *keys each appear in one record (e.g.,* $P_{\texttt{user}} = 0.5$*) and have entropies of* 0.35 *(*$= -\frac{1}{2} \log \frac{1}{2}$*). Combined, the total Key-Space entropy of the records is* $\mathcal{E}_{\mathcal{K}} = 0.70$ *(*$= 2 \cdot 0 + 2 \cdot 0.35$*).*

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:

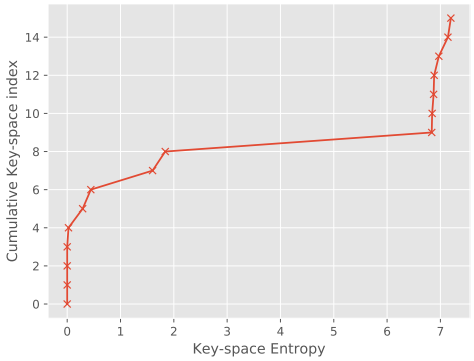

Figure 4: Yelp nested collection key-space entropy

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

Nulls are similar to anything, while primitive types are similar only to themselves and null. Like-kinded 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 subsumptive: If $\tau_1 \approx \tau_2$ and $(\tau_1 \cup \tau_2) \approx \tau_3$, then $\tau_1, \tau_2 \approx \tau_3$. 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-kinded 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.* JXPLAIN arbitrarily selects 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 }; $\mathcal{E}_\mathcal{T} = \mathcal{E}_\mathcal{K} = 0$
2: **for all** $\tau \in \mathcal{R}$ **do**
3: `RecordCount` += 1; `KindCount` = { $*$: 0 }
4: **for all** `key` $\in$ `keys`(τ) **do**
5: `KeyCnt[key]` += 1 ; `KindCount[kind(`τ`.key)]` += 1
6: **for all** (`kind` : `count`) $\in$ `KindCount` **do**
7: $\mathcal{E}_\mathcal{T}$ += $\frac{\texttt{count}}{|\texttt{keys}(\tau)|} \log\left(\frac{\texttt{count}}{|\texttt{keys}(\tau)|}\right)$
8: **if** $\mathcal{E}_\mathcal{T} > 0$ **then return** `Tuple`
9: **for all** (`key` : `count`) $\in$ `KeyCount` **do**
10: $\mathcal{E}_\mathcal{K}$ += $\frac{\texttt{count}}{\texttt{RecordCount}} \log\left(\frac{\texttt{count}}{\texttt{RecordCount}}\right)$
11: **if** $\mathcal{E}_\mathcal{K} \leq 1$ **then return** `Tuple` ▷ Threshold value
12: **else return** `Collection`

[$\mathbb{R}, \mathbb{R}$]), 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 P_ℓ, rather than the set of keys.

$$\mathcal{E}_\mathcal{K} = -\sum_\ell P_\ell \log P_\ell \qquad P_\ell = \frac{|\{\ i \mid i \in [N], \ell = M_i\ \}|}{N}$$

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 $\mathcal{S}_\mathcal{G}$ 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.

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

The entity discovery problem is to find approximately K schemas $\mathcal{S}$ that union to replicate $\mathcal{S}_\mathcal{G}$ 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.*

$$\mathcal{S}_1 = \{\ \{\ \texttt{ts} : \mathbb{R}, \texttt{event} : \mathbb{S}, \texttt{user} : \{\ldots\}\ \}, \{\ \texttt{ts} : \mathbb{R}, \texttt{event} : \mathbb{S}, \texttt{files} : [\ \mathbb{S}\ ]\ \}\ \}$$

$$\mathcal{S}_2 = \{\ \{\ \texttt{ts} : \mathbb{R}, \texttt{event} : \mathbb{S}, \texttt{user}^? : \{\ldots\}, \texttt{files}^? : [\ \mathbb{S}\ ]\ \}\ \}$$

[2] A limitation of manual schema curation: At time of writing, this page was out of date. $\mathcal{S}_1$ *encodes two distinct entities (one for each event type), while* $\mathcal{S}_2$ *uses a single entity with optional fields.*

At one extreme, we can consider each distinct source record to be a distinct entity (i.e., $K = N$, modulo duplicates). This approach is not only verbose, but can 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 $\mathcal{S}_\mathcal{G}$ 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`*. A business record missing 17 optional fields shares only 1 field in common with a photos record. However, between them there are only 6 (1 shared, 3 photo-only, 2 business-only) distinct fields. Thus, the Jaccard index considers this buisness more similar to a photo* ($\frac{1}{6} = 0.167$) *than to a business with all 20 attributes* ($\frac{3}{20} = 0.15$)*.*

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 set (k_{max}; line 4) and partitions the remaining records into three groups: (i) strict subsets of k_{max} ($\mathcal{K}_{sub}$), (ii) key-sets overlapping with k_{max} ($\mathcal{K}_{overlap}$), and (iii) key-sets disjoint with k_{max} ($\mathcal{K}_{disjoint}$). Preserving the original order within each partition, partitions are rearranged as $\mathcal{K}_{sub}, \mathcal{K}_{overlap}, \mathcal{K}_{disjoint}$, and the algorithm iteratively sorts the latter two. Although we omit field order here, it is sorted analogously.

The resulting sort order places subsets ($\mathcal{K}_{sub}$) closest, partially overlapping sets ($\mathcal{K}_{overlap}$) slightly further away, and fully disjoint subsets ($\mathcal{K}_{disjoint}$) 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 can not be applied directly. A naive Bimax-inspired

Algorithm 6 Bimax

In: $\mathcal{K}$: a list of sets of keys.
Out: $\mathcal{K}$ re-ordered with similar key-sets nearby.

1: Sort $\mathcal{K}$ in descending order of key-set size.
2: $i \leftarrow 1$
3: **while** $i < |\mathcal{K}|$ **do**
4: $\quad k_{max} \leftarrow \mathcal{K}[i]$
5: $\quad \mathcal{K}_i \leftarrow \{\ \mathcal{K}[j] \mid i \le j \le |\mathcal{K}|\ \}$
6: $\quad \mathcal{K}_{sub} = \{\ k \mid k \subseteq k_{max}, k \in \mathcal{K}_i\ \}$
7: $\quad \mathcal{K}_{disjoint} = \{\ k \mid k \cap k_{max} = \emptyset, k \in \mathcal{K}_i\ \}$
8: $\quad \mathcal{K}_{overlap} = \mathcal{K}_i - \mathcal{K}_{sub} - \mathcal{K}_{disjoint}$
9: $\quad$ Sort $\mathcal{K}[i, \ldots, |\mathcal{K}|]$ on $\mathcal{K}_{sub} < \mathcal{K}_{overlap} < \mathcal{K}_{disjoint}$
$\quad\quad$ ▷ Preserve existing order within each set
10: $\quad i \leftarrow i + |\mathcal{K}_{sub}|$

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: $\quad$ Repeat **Bimax** lines 4-10.
5: $\quad$ Add the cluster $\mathcal{K}_{sub}$ to $\mathcal{K}_{naive}$

clustering algorithm (Algorithm 7) builds entities out of the $\mathcal{K}_{sub}$ 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 record in the cluster must be a subset), which becomes less likely to appear in the input data as more optional fields appear.

Example 10. *Consider an entity with N optional fields, each independently present in any individual record with probability p. To have even a 50% chance of seeing a maximal record requires $\left(\frac{1}{p}\right)^N$ example records. For example, with 10 optional fields, each with appearing with probability 0.1, we would need to see (in expectation) 10 trillion records to see a maximal record.*

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

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: $\quad k_{cand} \leftarrow$ the maximal element of K_{cand}
3: $\quad$ **loop**
4: $\quad\quad$ Find minimal $\mathcal{K}_{cover} \subseteq (K_{naive} - \{\ K_{cand}\ \})$
$\quad\quad\quad$ s.t. $k_{cand} \subseteq \bigcup_{k \in K; K \in \mathcal{K}_{cover}} k$
5: $\quad\quad$ **if** K_{cover} exists **then**
6: $\quad\quad\quad \mathcal{K}_{naive} \leftarrow \mathcal{K}_{naive} - \mathcal{K}_{cover}$
7: $\quad\quad\quad K_{cand} \leftarrow K_{cand} \cup (\bigcup \mathcal{K}_{cover})$
8: $\quad\quad\quad k_{cand} \leftarrow k_{cand} \cup$ every new key in $\mathcal{K}_{cover}$
9: $\quad\quad$ **else break**
10: $\quad$ Add K_{cand} to $\mathcal{K}_{merge}$

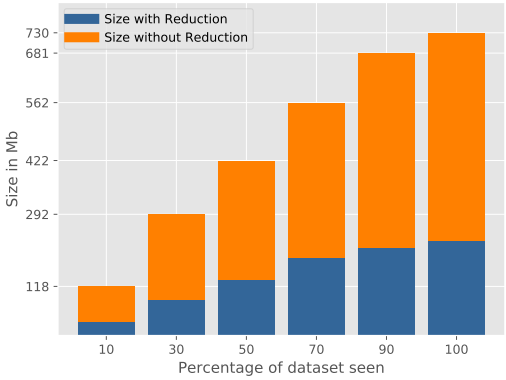

Figure 5: Memory comparison: Yelp

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:*

$$\mathcal{K}_{naive} = \{\ E_1 : \{A, B, E\}\quad E_2 : \{B, C, E\}\quad E_3 : \{C, D, E\}\quad E_4 : \{B, D\}\}$$

*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.

6.4 Implementation

This entity discovery process needs to be run once on each tuple-typed path appearing in the schema. Because Algorithm 7 requires multiple passes over 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 operations 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 discovery algorithm proposed by Baazizi et. al. [5, 7]. We chose $\mathcal{K}$-reduction, because it is a close analog to industry standard techniques 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) **$\mathcal{L}$-reduce**: Baazizi et. al.'s $\mathcal{L}$-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 sizes[3], 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 800 thousand 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 Times[4]. 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 `checkin` table 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

[3] The 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
[4] NYT 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.

		$\mathcal{K}$-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)

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 $\mathcal{K}$-reduce, we obtained a binary (JAR) implementation of $\mathcal{K}$-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 $\mathcal{K}$-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 Jxplain and $\mathcal{K}$-reduction both produce perfect schemas with a 1% sample. On the remaining datsets, even with only a 1% training set, schemas generated by Jxplain 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: Jxplain 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, Jxplain 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:

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

As with the Pharmaceutical dataset, Jxplain correctly identifies this structure as a two-level nested collection, allowing it to generalize to servers and keys not present in the sample.

Dataset		$\mathcal{K}$-reduce		Bimax-Merge		Bimax-Naive		$\mathcal{L}$-reduce	
		mean	std	mean	std	mean	std	mean	std
NYT	1%	21.21	0.71	14.70	1.02	14.70	1.02	8.67	0.06
	10%	21.13	0.00	17.94	0.00	18.05	0.27	10.56	0.00
	50%	23.00	0.93	18.74	0.40	18.59	0.17	11.58	0.01
	90%	23.46	0.00	18.94	0.00	18.67	0.00	11.82	0.00
Synapse	1%	248.87	19.86	175.34	11.70	176.14	10.62	8.64	0.07
	10%	749.22	21.56	656.24	49.99	662.60	40.62	10.80	0.01
	50%	1598.62	10.01	1459.84	54.93	1490.40	8.89	12.40	0.00
	90%	1974.35	16.14	1799.32	87.59	1848.00	16.65	12.99	0.00
Twitter	1%	279.63	2.64	147.01	23.82	96.21	5.00	11.37	0.01
	10%	496.26	3.99	166.90	23.96	130.91	6.57	13.97	0.01
	50%	518.06	5.13	212.72	22.07	154.81	1.47	15.67	0.00
	90%	526.95	1.28	235.14	10.74	156.01	0.00	16.27	0.00
Github	1%	85.78	0.63	25.88	0.11	25.84	0.15	10.48	0.02
	10%	92.24	0.98	28.62	0.88	153.87	1.79	12.38	0.00
	50%	93.72	0.80	29.42	0.53	155.48	1.07	13.60	0.00
	90%	94.12	0.00	29.69	0.00	156.02	0.00	14.01	0.00
Pharma	1%	1199.20	8.28	1199.20	8.28	1199.20	8.28	10.86	0.03
	10%	1801.80	15.74	1801.80	15.74	1801.80	15.74	14.03	0.01
	50%	2223.60	13.60	2223.60	13.60	2223.60	13.60	16.28	0.00
	90%	2369.20	3.31	2369.20	3.31	2369.20	3.31	17.11	0.00
WikiData	1%	2575.55	71.57	1506.07	62.84	1220.65	28.06	†	†
	10%	4131.32	70.77	2214.03	97.14	†	†	†	†
	50%	5969.92	48.10	4334.19	96.76	†	†	†	†
	90%	6890.20	27.73	5037.16	65.12	†	†	†	†
Yelp-Merged	1%	268.80	0.75	175.00	0.00	175.00	0.00	11.62	0.03
	10%	269.80	0.40	175.00	0.00	175.00	0.00	14.39	0.01
	50%	270.47	0.57	175.00	0.00	175.00	0.00	16.42	0.00
	90%	271.17	0.00	175.00	0.00	175.00	0.00	17.18	0.00
Yelp-Business	1%	50.40	1.36	44.62	2.80	38.62	0.49	9.92	0.03
	10%	52.20	0.98	47.21	2.04	39.89	3.17	12.49	0.01
	50%	54.00	0.00	49.88	2.62	46.01	0.00	14.27	0.01
	90%	53.60	0.80	49.61	0.80	45.61	0.80	14.91	0.00

Table 2: Schema Entropy: The number (log 2) of types accepted by the generated schema († ran out of resources).

7.2 Schema Entropy

In lieu of ground truth, we support claim (i) by measuring how restrictive the generated schema is. Specifically: our next experiment measures the number of possible types admitted by the output schema, a measure we refer to as *schema entropy*. Schema entropy is computed by treating each optional path as a binary decision, taking into account mandatory and locally mandatory paths. For collections (i.e., $\{ * \rightarrow \tau \}^*$ or $[\tau]^*$) we range over the active domain of the matched object, or over arrays of length up to the longest present in the data. Additionally, typing and notation are held consistent between implementations. Intuitively, given a high Recall, accepting fewer types indicates a more precise schema.

The results for each dataset are illustrated in Table 2; We test schemas generated from 1%-, 10%-, 50%-, and 90%- uniform samples taken from each dataset. Datasets with single-type schemas are omitted. We include $\mathcal{L}$-reduce — the number of distinct types in the training set — as a lower bound. Jxplain's tighter schemas are largely due to partitioning entities. In the Yelp and GitHub datsets especially, mixed entity types pose a challenge for $\mathcal{K}$-reduce, which emits only +a single entity with many optional fields. Conversely, Jxplain detects the entities correctly and partitions schemas, greatly reducing the number of admitted types. This detection is challenging on the merged Yelp dataset, where foreign keys like `business_id` and `user_id` are shared across entities. On datasets with one underlying schema and no functional dependencies like Yelp Photos, our output schema is identical to $\mathcal{K}$-reduce. Finally, note that schema entropy is stable across sample sizes: Even with only 10% of each dataset, both generators produce virtually the same schema that would be generated from the full dataset.

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 inferrable. The synthetic **Yelp-Merged** has ground-truth by definition, while the **GitHub** event trace includes a "type" attribute with 14 distinct types that we use as a ground truth. Concretely, we compare **Bimax-Merge** against $\mathcal{K}$-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 $(\mathcal{S}_i, \mathcal{G}_j)$ where each $\mathcal{S}_i$ is the schema derived for one cluster, and each $\mathcal{G}_j$ is the schema for one ground-truth entity: $D(\mathcal{S}_j, \mathcal{G}_j) = |\mathcal{S}_i - \mathcal{G}_j| + |\mathcal{G}_i, \mathcal{S}_j|$. Note that $\mathcal{K}$-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, $\mathcal{K}$-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 who's 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, Jxplain identified multiple entities in Yelp's `business` fields, separating out hair salons, which nearly always have, and are nearly always indicated by the presence of a `by_appointment` field.

7.4 Runtime

Finally we evaluate claim (v) by comparing the runtime of Jxplain against that of $\mathcal{K}$-reduce. We omit the Pharmaceutical dataset runtime where the official binary implementation times out, as previously noted. In general, Jxplain needs to do more work to create a more precise schema, so we do not expect it to outperform $\mathcal{K}$-reduce; We aim here simply to assess the added overhead. Table 5 shows schema discovery performance, varying the proportion of

	Yelp						Github													
Dataset	Bus	Ckn	Pho	Rev	Tip	Usr	Com	Cre	Del	For	Gol	IsC	Iss	Mem	Pub	Pul	PRR	Psh	Rel	Wat
K-reduce	197	132	297	292	296	277	798	825	827	732	823	675	709	810	830	383	370	815	760	829
Bimax-Merge	0	0	0	0	0	0	0	0	2	0	0	0	34	0	5	0	0	0	0	6
k-means	0	0	106	109	107	124	127	126	124	115	128	4	0	139	121	0	1	136	143	120

Table 3: Minimum symmetric difference from ground-truth schema (lower is better)

	$\mathcal{L}$-reduce		Bimax-Naive		**Bimax-Merge**	
Dataset	mean	std	mean	std	mean	std
Twitter	79191.0	73.7	72.8	1.6	8.4	1.6
NYT	3627.0	12.0	5.0	0.0	1.0	0.0
Synapse	8127.0	14.3	97.0	2.6	35.0	1.9
Github	16533.7	16.8	10.0	0.0	10.0	0.0
Pharma	141177.0	61.9	1.0	0.0	1.0	0.0
Wikidata	†	†	†	†	31.0	13.4
Yelp-Merged	148242.0	44.6	40.8	4.1	8.0	1.3
Yelp-Business	30809.0	56.5	33.2	1.2	2.6	0.8
Yelp-Checkin	108229.0	51.6	1.0	0.0	1.0	0.0
Yelp-Photos	1.0	0.0	1.0	0.0	1.0	0.0
Yelp-Review	1.0	0.0	1.0	0.0	1.0	0.0
Yelp-Tip	1.0	0.0	1.0	0.0	1.0	0.0
Yelp-User	9142.0	17.0	1.0	0.0	1.0	0.0

Table 4: Extractor entity predictions with 90% training data († $\mathcal{L}$-reduce and Bimax ran out of resources on Wikidata)

each dataset used in order to generate the schema to show scaling behavior. Performance scales linearly for both extractors. Jxplain for all tests was not using entropy approximation and thus required a full second pass over the dataset, which we see reflected in the runtimes of Yelp, Synapse, and NYT, being approximately 2-3 times slower. We observe that Jxplain has a particularly hard time with more complicated datasets (e.g. Twitter, Github). This is the result of large, nested object arrays that need to be decoded, stored, and pivoted for recursive entity extraction, something $\mathcal{K}$-reduce does not attempt. However, the value of this overhead is especially clear in the NYT data set, as Jxplain picks-out complex nested structures.

7.5 Results

Table 2 illustrates the potential data intricacies lost from a merge all strategy like $\mathcal{K}$-reduce. Between over generalization and over specificity, generalization is the only realistic option, as demonstrated in Table 1 through $\mathcal{L}$-reduce's unusably low validation scores. We demonstrate Bimax-Naive as a realistic middle ground, based on these two metrics. However, when considering human-schema-interaction we need to limit the number of user-facing schema choices. Table 4 makes clear the importance of the Bimax merge heuristic for creating compact, descriptive schemas. Jxplain produces goldilocks schemas with the benefit of both extractor implementations proposed in related work [5, 7]. Lastly, although Jxplain is slower than $\mathcal{K}$-reduce, the amount of data required to create high-precision, high-recall schemas is not large — at most 10% of the original data in our experiments.

Row rejection was overwhelmingly due to missing attributes for both $\mathcal{K}$-reduce and Bimax-Merge algorithms. We devised a greedy algorithm to obtain an upper bound of the number of schema edits needed to achieve 100% recall across one or more entities. Using 1% training data, we find that both algorithms produce schemas that require relatively few manual edits to achieve perfect recall for simpler datasets, typicaly on the order of tens of edits. More complex datasets require hundreds to thousands of edits for **both** $\mathcal{K}$-reduce and Bimax-Merge: Bimax-Merge does better on datsets with collection-like objects (e.g., Synapse and Pharma), where $\mathcal{K}$-reduce struggles with new keys. The reverse is true on datasets with rare, or rarely missing attributes that appear in multiple entity types. For example, retweets and quoted tweets share many fields — Bimax-Merge has to see one example of the attribute for each entity, while $\mathcal{K}$-reduce only needs one example outright. Thus, we assert that the human intervention necessary to deal with false positives is no less feasible for Jxplain over $\mathcal{K}$-reduce in general, while by contrast, Jxplain produces far fewer false negatives.

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**.

	1%		10%		50%		90%	
Dataset	*K-reduce*	*Bimax-Merge*	*K-reduce*	*Bimax-Merge*	*K-reduce*	*Bimax-Merge*	*K-reduce*	*Bimax-Merge*
NYT	3355.2	5285.4	3445.4	6939.2	4102.6	10657.6	4710.0	14214.2
Synapse	3325.6	5568.0	1823.0	8900.8	2848.4	12715.4	3867.0	20233.6
Github	17881.0	77995.2	24022.6	278932.4	37990.0	703276.8	49191.2	931849.2
Twitter	11758.6	41710.8	16111.6	195733.2	19719.8	429832.4	17604.6	550703.6
Pharma	†	5816.2	†	8811.2	†	18901.0	†	29389.0
WikiData	24585.8	110090.4	62286.0	300846.6	213958.2	827083.8	422793.8	1379410.2
Yelp-Business	3485.8	6301.2	2022.2	9560.8	3309.0	19680.0	4732.8	27386.6
Yelp-Checkin	3682.6	5156.4	2777.4	7695.2	7395.8	14267.0	13234.8	20631.0
Yelp-Photos	3138.2	4338.8	1253.0	4742.4	1500.8	6222.2	1776.2	7703.8
Yelp-Review	8658.8	14789.4	9306.0	17630.8	15118.2	32696.8	21976.6	49642.4
Yelp-Tip	3497.2	4793.2	1835.2	5789.4	2518.6	9589.2	3225.4	13899.4
Yelp-User	6482.0	14355.4	5421.6	20480.8	7612.6	34600.2	10590.4	51143.2
Yelp-Merged	10179.0	22177.8	14475.4	36418.8	28217.6	92197.8	44950.8	141905.0

Table 5: Runtime (milliseconds) by discovery algorithm and training set size. (†: *K*-reduce times out on the Pharma dataset)

A range of machine-learning-based techniques have been proposed for discovering such linkages [23, 28, 29, 34]. However, as we discuss in Section 6, such approaches are vulnerable to skew in entity size, and the inherent ambiguity of the entity discovery problem. 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[5]. 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. These 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 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.

[5]These approaches are, in effect, simply normalizing their inputs

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 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: 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 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. Co-reference detection (i.e., matching entities that appear at different paths) is another area where Jxplain can be improved. Co-reference detection is critical for alternative hierarchical data representations. For example, we plan to explore the use of Jxplain to create schemas for filesystem directory structures. Finally, as we noted at the start, Json's flexible self-describing schemas are optimized for creating data. We believe that schemas extracted by Jxplain can be used for read-optimized data layouts.

REFERENCES

[1] Z. Abedjan, L. Golab, F. Naumann, and T. Papenbrock. *Data Profiling*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.

[2] P. Andritsos, R. J. Miller, and P. Tsaparas. Information-theoretic tools for mining database structure from largedata sets. In *SIGMOD Conference*, pages 731–742. ACM, 2004.

[3] Apache Spark. Spark documentation: Data sources: Json files. https://spark.apache.org/docs/latest/sql-data-sources-json.html, 2018.

[4] M. A. Baazizi, C. Berti, D. Colazzo, G. Ghelli, and C. Sartiani. Human-in-the-loop schema inference for massive json datasets. In *EDBT*, 2020.

[5] M. A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani. Counting types for massive JSON datasets. In *DBPL*, pages 9:1–9:12. ACM, 2017.

[6] M.-A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani. Parametric schema inference for massive json datasets. *The VLDB Journal*, pages 1–25, 2019.

[7] M. A. Baazizi, H. B. Lahmar, D. Colazzo, G. Ghelli, and C. Sartiani. Schema inference for massive JSON datasets. In *EDBT*, pages 222–233. OpenProceedings.org, 2017.

[8] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise dtds from XML data. In *VLDB*, pages 115–126. ACM, 2006.

[9] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML schema definitions from XML data. In *VLDB*, pages 998–1009. ACM, 2007.

[10] P. Bohannon, J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. Legodb: Customizing relational storage for XML documents. In *VLDB*, pages 1091–1094. Morgan Kaufmann, 2002.

[11] H. Butler, M. Daly, A. Doyle, S. Gillies, T. Schaub, and T. Schaub. The GeoJSON Format. RFC 7946, Aug. 2016.

[12] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: The prism workbench. *Proc. VLDB Endow.*, 1(1):761–772, aug, 2008.

[13] M. DiScala and D. J. Abadi. Automatic generation of normalized relational schemas from nested key-value data. In *SIGMOD Conference*, pages 295–310. ACM, 2016.

[14] A. A. Frozza, R. dos Santos Mello, and F. de Souza da Costa. An approach for schema extraction of JSON and extended JSON document collections. In *IRI*, pages 356–363. IEEE, 2018.

[15] GitHub, Inc. Github developer: Webhooks. https://developer.github.com/webhooks/.

[16] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445. Morgan Kaufmann, 1997.

[17] J. Hegewald, F. Naumann, and M. Weis. Xstruct: Efficient schema extraction from multiple and large XML documents. In *ICDE Workshops*, page 81. IEEE Computer Society, 2006.

[18] Z. H. Liu, B. C. Hammerschmidt, D. McMahon, Y. Liu, and H. J. Chang. Closing the functional and performance gap between SQL and nosql. In *SIGMOD Conference*, pages 227–238. ACM, 2016.

[19] Matrix.org. Matrix: An open network for secure, decentralized communication. https://matrix.org/docs/projects/server/synapse.

[20] R. J. Miller and P. Andritsos. Schema discovery. *IEEE Data Eng. Bull.*, 26(3):40–45, 2003.

[21] J. Min, J. Ahn, and C. Chung. Efficient extraction of schemas for XML documents. *Inf. Process. Lett.*, 85(1):7–12, 2003.

[22] M. L. Möller, N. Berton, M. Klettke, S. Scherzinger, and U. Störl. jHound: Large-scale profiling of open JSON data. In *BTW*, volume P-289 of *LNI*, pages 555–558. Gesellschaft für Informatik, Bonn, 2019.

[23] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *ACM International Conference on Management of Data (SIGMOD 1998)*, 1998.

[24] A. Prelic, S. Bleuler, P. Zimmermann, A. Wille, P. Bühlmann, W. Gruissem, L. Hennig, L. Thiele, and E. Zitzler. A systematic comparison and evaluation of biclustering methods for gene expression data. *Bioinformatics*, 22(9):1122–1129, 2006.

[25] Roam Analytics. Prescription-based prediction. https://www.kaggle.com/roamresearch/prescriptionbasedprediction, 2017.

[26] F. Robardet, Célineand Feschet. Effcient local search in conceptual clustering. In *Discovery Science*, pages 323–335, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[27] R. T. Snodgrass, C. E. Dyreson, F. Currim, S. Currim, and S. Joshi. Validating quicksand: Temporal schema versioning in tauxschema. *Data Knowl. Eng.*, 65(2):223–242, 2008.

[28] W. Spoth, T. Xie, O. Kennedy, Y. Yang, B. Hammerschmidt, Z. H. Liu, and D. Gawlick. SchemaDrill: Interactive semi-structured schema design. In *HILDA*, 2018.

[29] R. M. Svetlozar Nestorov, Serge Abiteboul. Inferring structure in semistructured data. 1997.

[30] M. Szymczak, S. Zadrozny, A. Bronselaer, and G. D. Tré. Coreference detection in an XML schema. *Inf. Sci.*, 296:237–262, 2015.

[31] The New York Times. The new york times article archive api. https://developer.nytimes.com/docs/archive-product/1/overview.

[32] Twitter. Decahose stream. https://developer.twitter.com/en/docs/tweets/ sample-realtime/api-reference/decahose.

[33] M. Vogel, S. Weber, and C. Zirpins. Experiences on migrating restful web services to graphql. In *ICSOC Workshops*, volume 10797 of *Lecture Notes in Computer Science*, pages 283–295. Springer, 2017.

[34] L. Wang, S. Zhang, J. Shi, L. Jiao, O. Hassanzadeh, J. Zou, and C. Wangz. Schema management for document stores. *PVLDB*, 8(9):922–933, May 2015.

[35] Wikibase. Wikibase entity data. https://www.mediawiki.org/wiki/Wikibase/EntityData.

[36] Yelp, Inc. Yelp open dataset: An all-purpose dataset for learning. https://www.yelp.com/dataset, 2018.

[37] C. Yu and H. V. Jagadish. XML schema refinement through redundancy detection and normalization. *VLDB J.*, 17(2):203–223, 2008.