SchemaDrill: Interactive Semi-Structured Schema Design*

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

Ying Yang, Beda Hammerschmidt,
Zhen Hua-Liu, Dieter Gawlick
Oracle
{ dieter.gawlick, zhen.liu, beda.hammerschmidt,
ying.y.yang }@oracle.com

ABSTRACT

Ad-hoc data models like Json make it easy to evolve schemas and to multiplex different data-types into a single stream. This flexibility makes Json great for generating data, but also makes it much harder to query, ingest into a database, and index. In this paper, we explore the first step of Json data loading: schema design. Specifically, we consider the challenge of designing schemas for existing Json datasets as an *interactive* problem. We present SchemaDrill, a roll-up/drill-down style interface for exploring collections of Json records. SchemaDrill helps users to visualize the collection, identify relevant fragments, and map it down into one or more flat, relational schemas. We describe and evaluate two key components of SchemaDrill: (1) A summary schema representation that significantly reduces the complexity of JSON schemas without a meaningful reduction in information content, and (2) A collection of schema visualizations that help users to qualitatively survey variability amongst different schemas in the collection.

ACM Reference Format:
William Spoth, Ting Xie, Oliver Kennedy and Ying Yang, Beda Hammerschmidt, Zhen Hua-Liu, Dieter Gawlick. 2018. SchemaDrill: Interactive Semi-Structured Schema Design. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Semi-structured formats like Json allow users to design schemas on-the-fly, as data is generated. For example, adding a new attribute to the output of a system logger does not break backwards compatibility with existing data. This flexibility facilitates the addition of new features and enables low-overhead adaptation of data-generating processes. However, because the data does not have a consistent underlying schema, it can be harder (and slower) to explore than simple tabular data. The logic of each and every query must explicitly account for variations in the schema like missing attributes. Performance also suffers, as there is no one physical data representation that is ideal for all schemas.

To address these problems, a variety of techniques [3, 7, 9, 13, 16] have arisen to generate schemas after-the-fact. The goal of these *semi-structured schema discovery* (S^3D) techniques is to propose a schema for collections of Json records. A common approach to this problem is to bind the Json records to a normalized relational representation, or in other words, to derive a set of flat *views* over the hierarchical Json data.

Existing automated approaches to this problem (e.g., [7, 9]) operate in a single-pass: They propose a schema and consider their job done. Unfortunately these techniques also rely heavily on general heuristics to select from among a set of schema design choices, as a clear declarative specification of a domain would be tantamount to having the schema already. To supplement domain-agnostic heuristics with feedback from domain experts, we propose a new *iterative and interactive* approach to S^3D called SchemaDrill.

SchemaDrill provides a OLAP-style interface (with analogs to roll-up, drill-down, and slice+dice) specialized for exploring collections of Json records. Every state of this interface corresponds to a relational view defined over the Json data. When ready, this view can be exported to a classical RDBMS or similar tool for simplified, more efficient data access. In this paper, we explore several design options for SchemaDrill, and discuss how each interacts with the challenges of S^3D.

1.1 Extracting Relational Entities

The first class of challenges we address involve the nuts and bolts of mapping hierarchical Json schemas to flat relational entities. Fundamentally, this involves a combination of three relational operators: Projection, Selection, and Unnesting.

Projecting Attributes. Json schema discovery can, naively, be thought of as a form of schema normalization [5], where each distinct path in a record is treated as its own attribute. Entities then, are simply groups of attributes projected out of the Json data, and the S^3D problem reduces to finding such groups (e.g., by using Functional Dependencies [7]).

Selecting Records. This naive approach fails in situations where the collection of Json records is a mix of different entity types that share properties. As a simple example, Twitter streams mix three entity types: tweets, retweets, and deleted tweets. Although each entity appears in distinct records, they share attributes in common. Hence, entity extraction is not just normalization in the classical sense of partitioning attributes, but rather also a matter of partitioning records by content.

Collapsing Nested Collections. Json specifies two collection types: Arrays and Objects. Typically the former is used for encoding nested collections and the latter for encoding tuples with named attributes. However, this is not a strict requirement. For example, latitude and longitude are often encoded as a 2-element array. Conversely, in some data sets, objects are used as way to index collections by named keys rather than by positions. Hence, simple type analysis can not distinguish between the two cases. This is problematic because treating a collection as a tuple creates an explosion of attributes that make classical normalization techniques incredibly expensive.

*The first two authors contributed equally

Conference'17, July 2017, Washington, DC, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

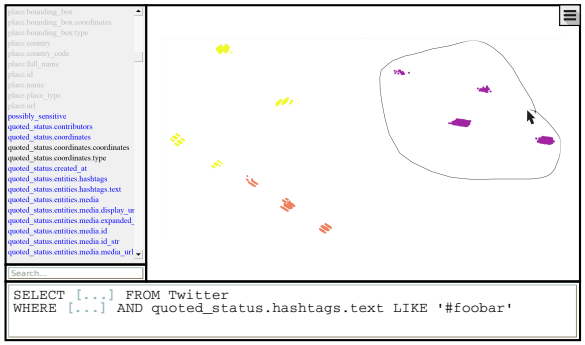


Figure 1: Prototype user interface for SchemaDrill.

1.2 Human-Scale S^{3}D

Even in settings where Json data is comparatively well behaved, it is common for it to have dozens, or even hundreds of attributes per record. Similarly, individual Json records can be built from any of the hundreds or thousands (or more) different permutations of the full set of attributes used across the entire collection. Bringing this information down to human scale requires simultaneously simplifying and summarizing.

Summarization. For the purposes of entity construction, the full set of attributes is often unnecessary. It is often possible to collapse multiple attributes together, or express attributes as equivalent alternatives. As an example, an address might consist of four distinct attributes city, zip code, street, and number when it could conceptually be expressed as just one.

Visualization. In addition to simplifying the underlying problem, it is also useful to give users a coarse "top-down" view of the schema process. Specifically, users need to (1) be able to see patterns of structural similarity between distinct schemas, and (2) understand how much variation exists in the data set as-is.

Iteration. By combining straightforward summarization and data visualization techniques, SchemaDrill helps users to quickly identify natural clusters of records and attributes that represent relational entities. SchemaDrill facilitates an *iterative* schema design process to allow human experts to better evaluate whether structures in the data indicate conceptual relationships between records or attributes, or are merely data artifacts.

1.3 Overview

Figure 1 shows the prototype interface of SchemaDrill. The pane on the left, discussed in Section 2, shows the schema of the currently selected JSON view, highlighting attributes and groups based on relevance. The pane on the right, discussed in Section 3, provides a top-down visual sketch of schemas in the currently selected JSON data, and allows users to interactively filter out parts of it. Finally in Section 3, we show examples of how SchemaDrill facilitates incremental, iterative exploration and mapping of JSON schemas.

2 SUMMARIZATION

The first component of SchemaDrill, the schema pane, shows the relational schema of the extracted view. Initially, this schema consists of one attribute for every path in the Json collection being summarized. Attributes may be deleted or restored, and sets of attributes may be unified.

The core challenge behind implementing this pane is that, depending on data set, the schema could consist of hundreds or thousands of attributes. This can be overwhelming for users who just wants to find account profiles appearing in a Twitter stream. To mitigate this problem, SchemaDrill presents the schema in a summarized form.

Specifically, attributes are grouped based on both correlations and anti-correlations between them. Groups of *correlated* attributes, or those that frequently co-occur in Json records are likely to be part of a common structure. Similarly, groups of *anti-correlated* attributes, or those that rarely co-occur in Json records are likely to represent alternatives (e.g., a Street Address vs GPS coordinates). We use correlations and anti-correlations between attributes to compact the schema for representation. Before describing the summary itself, we first formalize the problem.

2.1 Data Model

A Json object is an order-independent mapping from keys to values. A key is a unique identifier of a Json object, typically a string. A value may be atomic or complex. Atomic values in Json may be integers, reals, strings, or booleans. A complex value is either a nested object, or an *array*, an indexed list of values. For simplicity, we model arrays as objects by treating array indexes as keys. A Json record may be any value type, but for simplicity of exposition we will assume that all records are objects.

Example 2.1. A fragment of Twitter Tweet encoded as Json

```
"tweet": {
  "text": "#SIGMOD2018 in Houston this year",
  "user": {
    "name" : "Alice Smith",  "id" : 12345,
    "friends_count": 1023,  ...
  },
  "retweeted_status": {
    "tweet": {
      "user": { ... }, "entities": { ... },
      "place": { ... }, "extended_entities": { ... }
    }, ...
  },
  "place": { ... }, "extended_entities": { ... },
  "entities": { "hashtags": [ "SIGMOD2018" ], ... },
  ...
}
```

Json objects are typically viewed as trees with atomic values at the leaves, complex values as inner nodes, and edges labeled with the keys of each child. Our goal is to identify commonalities in the structure of this tree across multiple Json records in a collection. To capture the structure, we define the *schema* S of a Json record as the record with all leaf values replaced by a constant $\perp$. A *path* P is a sequence of keys $P = (k_0, \ldots, k_N)$. For convenience, we will write paths using dots to separate keys (e.g., `tweet.text`). We say that a path appears in a schema (denoted $P \in S$) if it is possible to start at the root of S and traverse edges in order. If the value reached by

this traversal is $\perp$, we say that P is a terminal path of S (denoted $P \perp S$).

2.2 Paths as Attributes

Ultimately, our goal is to create a flat, relational representation suitable for use with an entire collection of Json records. The first step to reaching this goal is to flatten individual Json schemas into collections of attributes. We begin with a naive translation where each attribute corresponds to one terminal path in the schema. We write $S^\perp$ to denote the path set, or relational schema of Json schema S, defined as: $S^\perp = \{\ P \mid P \perp S\ \}$ Since keys are unique, commutative and associative, this representation is interchangeable[1] with the tree representation. Hence, when clear from context we will abuse syntax, using S to denote both a schema and its path set.

Example 2.2. The path set of the Json object from Example 2.1 includes the paths: (1) `tweet.text` (2) `tweet.user.friends_count` (3) `tweet.user.id` (4) `tweet.entities.hashtags.[0]` Each terminal appears in the set. Note in particular that single element of the array at `tweet.entities.hashtags` is assigned the key `[0]`.

Path sets make it possible to consider containment relationships between schemas. We say that S_1 is contained in S_2 iff $S_1^\perp \subseteq S_2^\perp$.

2.3 Schema Collections

We now return to our main goal, summarizing the schemas of collections of Json records. The starting point for this process is the schemas themselves. Given a collection of Json records, we can extract the set of schemas $\{\ S_1, \ldots, S_N\ \}$ of records in the collection, which we call the *source schemas*. One existing technique for summarizing these records, used by Oracle's JSON Data Guides [14, 15], is to simply present the set of all paths that appear anywhere in this collection. We call this the *joint schema* $\mathbb{S}$:

$$\mathbb{S}^\perp \stackrel{def}{=} \bigcup_i S_i^\perp$$

Observe that, by definition, each of the source schemas is contained in the joint schema. The joint schema mirrors existing schemes for relational access to JSON data like for example. However, the joint schema can still be very large, with hundreds, thousands, or even tens of thousands of columns[2]. To summarize them we need an even more compact encoding for sets of schemas.

A Schema Algebra. As a basis for compacting schema sets, we define a simple algebra. Recall that we are particularly interested in summarizing cooccurrence and anti-cooccurence relationships between attributes.

$$\mathbf{A} \ := \ \mathbf{P} \mid \emptyset \mid \mathbf{A} \wedge \mathbf{A} \mid \mathbf{A} \vee \mathbf{A}$$

Expressions in the algebra construct sets of schemas from individual attributes. There are two types of leaf expressions in the algebra: A single terminal path P represents a singleton schema ($\{\{P\}\}$), while $\emptyset$ denotes a set containing no schemas $\{\}$. Disjunction acts as union, merging its two input sets:

$$\{\ S_1, \ldots, S_N\ \} \vee \{\ S'_1, \ldots, S'_M\ \} \stackrel{def}{=} \{\ S_1, \ldots, S_N, S'_1, \ldots, S'_M\ \}$$

Disjunction models anti-correlation: The resulting schema set is effectively a collection of schema alternatives. For example, $P_1 \vee P_2$ indicates two alternative schemas: $\{P_1\}$ or $\{P_2\}$. Conjunction combines schema sets by cartesian product:

$$\{\ S_1, \ldots, S_N\ \} \wedge \{\ S'_1, \ldots, S'_M\ \} \stackrel{def}{=} \left\{\ S_i \cup S'_j \;\middle|\; i \in [1, N], j \in [1, M]\ \right\}$$

Conjunction models correlations: The resulting schema set mandates that exactly one option from the left-hand-side and one option from the right-hand-side be present. On singleton inputs, the result is also a singleton. For example, $P_1 \wedge P_2$ is a single schema that includes both P_1 and P_2. For inputs larger than one element, the conjunction requires one choice from each input. For example, $(P_1 \vee P_2) \wedge (P_3 \vee P_4)$ is the set of all schemas consisting of one of P_1 or P_2, and also one of P_3 or P_4.

Although we omit the proofs for conciseness, both $\wedge$ and $\vee$ are commutative and associative, and $\vee$ distributes over $\wedge$[3]. For conciseness, we use the following syntactic conventions: (1) When clear from context, a schema S denotes its own singleton set, and (2) We write P_1P_2 to denote $P_1 \wedge P_2$.

This schema algebra gives us a range of ways to represent schema sets. At one extreme, the set of source schemas arrives in what is effectively disjunctive normal form (DNF). One schema may be expressed as a conjunction of its elements, and the full set of source schemas can be constructed by disjunction. For example, the source schemas $\{P_1, P_2\}$ and $\{P_2, P_3\}$ may be represented in the algebra as $P_1P_2 \vee P_2P_3$.

At the other extreme, the joint schema is a superset of all of the source schemas. It too can be thought of as a schema set, albeit one that loses information about which attributes appear in which schemas. Hence, this joint schema set may be defined as the power set of all attributes in the joint schema.

$$2^{\mathbb{S}} = \bigwedge_{P \in \mathbb{S}} (P \vee \emptyset)$$

Observe that at a minimum, each of the source schemas must appear in this schema set ($S_i \in 2^{\mathbb{S}}$). However many other schemas appear in the resulting schema set as well.

2.4 Summarizing Schema Collections

These two extreme representations (the raw source schemas and the joint schema set) are bad, but for subtly different reasons. In both cases, the representation is too verbose. In the former case verboseness stems from redundancy, with significant overlap in variables between the source schemas. Conversely in the latter case it stems from imprecision, as the schema set encompasses schemas that do not appear in the source schemas. Of the two, the latter is more compact, in particular because each attribute appears no more than once. This is a distinct representational advantage because the joint schema can be displayed simply as a list.

We would like to preserve this only-once property. Our aim then, is to derive an algebraic expression (1) in which each attribute appears exactly once, and (2) that is as tight a fit to the original source schema set as possible. Ultimately, this problem reduces to polynomial factorization and the discovery of read-once formulas [6], a problem that is, in general, worse than P-time [4]. Hence, for this paper, approximations are required. We consider two approaches

[1] modulo empty objects or arrays

[2] One dataset [1] achieved 2.4 thousand paths through nested collections of objects.

[3] To be precise, the structure $\langle\ \{\{\mathbf{P}\}\}, \vee, \wedge, \emptyset, \{\{\}\}\ \rangle$ is a semiring.

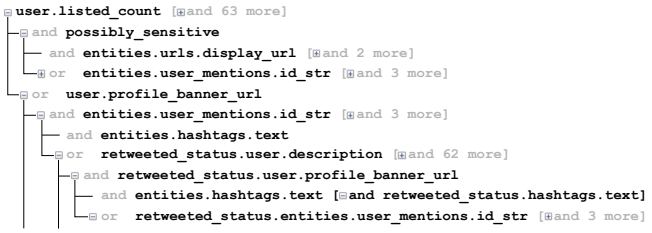


Figure 2: FP-Tree based schema summaries

and allow the user to select the most appropriate one for their needs. The first is based on Frequent Pattern Trees [10] (FPTrees), a data structure commonly used for frequent pattern mining. The second is to limit our search to read-once conjunctions of disjunctions of conjunctions, a form we call RCDC.

FP-Tree Summaries. An FP-Tree is a trie-like data structure that makes it easier to identify common patterns in a query. Every edge in the tree represents the inclusion of one feature, or in our case one attribute. Hence, every node in the tree corresponds to a set of paths (obtained by traversal from the root), and every leaf corresponds to one source schema. We observe that every node in an FP tree corresponds to a disjunction: For a node with 3 children, each subtree represents a different branch. Similarly, every edge corresponds to a conjunction with a singleton. Although the resulting tree may duplicate some attributes, duplications are minimized [10].

Example 2.3. Figure 2 illustrates a schema summary based on FP-Trees. Sequences of nodes with a single child are collapsed into single rows of the display (e.g., `user.listed_count` and 63 immediate descendents). A toggle switch allows these entities to be displayed to the user, if desired. Every level of the tree represents a set of alternatives. For example, `possibly_sensitive` never co-occurs with `user.profile_banner_url`.

RCDC Summaries. Our second visualization is based on [anti-]correlations. To construct this visualization, we begin with the joint summary. Recall that the joint summary has the form

$$(P_1 \vee \emptyset)\ (P_2 \vee \emptyset)\ (P_3 \vee \emptyset)\ (P_4 \vee \emptyset) \ldots$$

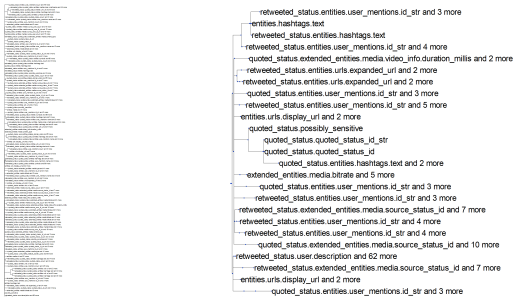
We create covariance matrix based on the probability of two attributes co-occurring in the schemas of one of our input Json records. Using this covariance matrix, hierarchical clustering [11], and a user-controlled threshold on the covariance, we cluster the attributes by parenthesizing. For example, if P_1 is clustered with P_2 and likewise P_3 and P_4. Because attributes within a group co-occur frequently we can approximate this formula by pushing down the optional $\emptyset$ disjunction and rewrite the schema as :

$$\approx (P_1P_2 \vee \emptyset)\ (P_3P_4 \vee \emptyset) \ldots$$

We now repeat the process with a new covariance matrix, based on the probability of two groups co-occurring. This time, however, we create clusters with a very negative covariance. Hence the resulting groups are unlikely to co-occur. Continuing the example, most frequently are $P_1P_2\emptyset$ and $P_3P_4\emptyset$. Approximating and simplifying, we get an expression in RCDC form.

$$\approx (P_1P_2 \vee P_3P_4 \vee \emptyset) \ldots$$

Figure 3: RCDC based schema summaries



(a) Without segmentation. **(b) With segmentation.**

Figure 4: Segmentation breaks up schema representations into manageable chunks.

As with the FP-Tree display, we use counts and an example attribute as a summary name for the group, and a toggle button to allow users to expand the group along either the OR or AND axes.

3 VISUALIZATION

Even within a mostly standardized collection of records like exported Twitter or Yelp data or production system logs, it is possible to find a range of schema usage patterns. Grouping by [anti-]cooccurrence is one step towards helping users understand these usage patterns, but is insufficient for three reasons: (1) Conceptually distinct fragments of the schema may share attributes in common (e.g., delete tweet records share attributes in common with tweet records). (2) Even if they do not co-occur, certain [groups of] attributes may be correlated (e.g., due to mobile phones, tweets with photos are also often geotagged). (3) There is no general way to differentiate Json objects and arrays being used to represent collections from those being used to represent structures (e.g., twitter stores geographical coordinates as a 2-element array). The second part of the SchemaDrill interface addresses these issues by presenting top-down visual surveys of the schema. These surveys help users to quickly assess variations in schema usage across the collection, to identify related schema structures, and to "drill down" into finer grained relationships.

3.1 Schema Segmentation

Specifically, we want to help the user to focus on specific parts of the joint schema; We want to allow the user to filter out, or segment the schema based on certain required attributes that we call *subschemas*. Specifically, a subschema s is a set of attributes, where s is contained in one or more source schemas. We define the s-segment of source schemas $S_1, \ldots, S_N$ to be the subset that contain s:

$$\textbf{segment}(s) \stackrel{def}{=} \{\ S_i \mid i \in [1, N] \wedge s \subseteq S_i\ \}$$

We are specifically interested in visual representations that can help users to identify subschemas of interest. By then focusing solely on the segments defined by these subschemas can significantly reduce the complexity of the schema design problem, as illustrated in Figure 4. Figures 4a illustrates the full schema summary as a tree, while Figure 4b shows a partial summary identified by the user using the lasso tool we describe shortly.

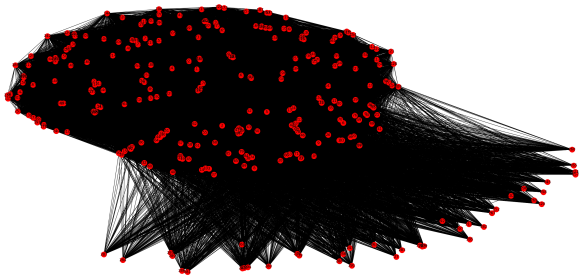


Figure 5: Coveriance Cloud for the full Yelp dataset.

Covariance Clouds. Our second visual representation, also like schema summaries, uses correlations and anti-correlations to communicate subschemas of interest. To generate a covariance cloud, we create a covariance matrix from the source schemas, using the appearance of each attribute as a variable. Based on a user-controllable threshold, we then construct a graph from the covariance matrix with every attribute as one node, and every covariance exceeding the threshold as an edge. The graph is then displayed to the user as a cloud using standard force-based layout techniques (e.g., those used by GraphViz [8]). Cliques in the graph represent commonly co-occurring subschemas that might form segments of interest. This includes every conjunctive group identified in the schema summary. However, unlike the schema summary, this visual representation more effectively captures subschemas with attributes in common.

KNN-PCA Clouds. While the first visualizaton works on simple schemas, we found that on more complex Json data like Twitter streams [17], or the Yelp open dataset [18] there were too many inter-attribute relationships, and the resulting visualizatons were noisy. An approach we settled on is a mixture of Principle Component Analysis (PCA) and K Nearest Neighbor clustering (KNN). As before, we treat each source schema as a feature vector with each attribute representing one feature. We then use PCA to plot our source schemas in two-dimensions. The resulting visualization illustrates relationships between source schemas, with greater distances in the visualization representing (approximately) more differences between the schemas. Hence, clustered groupings of schemas represent potentially interesting sub-schemas.

A key limitation with this visualization is that for more complex datasets the somewhat arbitrary choice of 2 dimensions can be too low. Conversely adding more dimensions directly through PCA makes the visualization more complicated and hard to follow. To mitigate these limitations, we use K-Nearest Neighbors (KNN) to colorize the PCA Cloud. In addition to using PCA, we do KNN clustering on the schemas using a user-provided K (number of clusters). Each cluster identified by PCA is assigned a different color. Combined, these two algorithms to provide users an initial insight into the potential correlations that exist in their dataset.

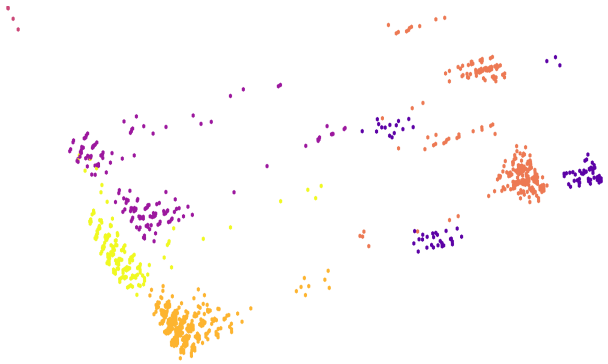


Figure 6: KNN-PCA cloud with K=6 on the Yelp dataset.

Example 3.1. Figures 1 and 6 show an example of the resulting view on Twitter and Yelp data. Note the much tighter clustering of attributes in the Twitter data: each cluster represents one particular, common type of tweet. Conversely, the Yelp schema includes a nested collection with, for example, hourly checkins at the business. The presence (or absence) of these terminal paths is more variable, and the resulting PCA cloud follows more of a gradient.

3.2 Schema Exploration

Now that we have shown the user potentially interesting subschemas, our next task is to help them to (1) narrow down on actually interesting subschemas, and (2) use those schemas to drill down to a segment of the schema data. For the first step, it is critical that the user develop a good intuition for what the visual representations encode. One way to accomplish this is to establish a bi-directional mapping between SchemaDrill's two data panes.

To map from visual survey to schema summary, we provide users with a lasso tool. As illustrated in Figure 1, users can select regions of the KNN-PCA Cloud and the corresponding schemas within that region. Doing so identifies the maximal subschema contained in all subschemas and regenerates the schema summary pane based only on the segment containing the maximal subschema. The maximal subschema itself is also highlighted in the schema summary pane. On the Covariance cloud, the lasso tool behaves similarly, selecting attributes explicitly rather than a maximal subschema.

The reverse mapping is achieved using highlighting, as illustrated in Figure 7. Users can select one or more attributes (or groupings) in the schema summary pane, and the KNN-PCA Cloud (resp., Covariance Cloud) is modified to highlight schemas in the corresponding segment (resp., to highlight the attributes). In conjunction with their prior knowledge of their tasks and ideal use cases, we use this approach to perform the initial schema segmentation.

In either case, after selecting a set of attributes or schemas, the analyst may choose to drill down into the selected segment, regenerating both views for the now restricted collection of schemas.

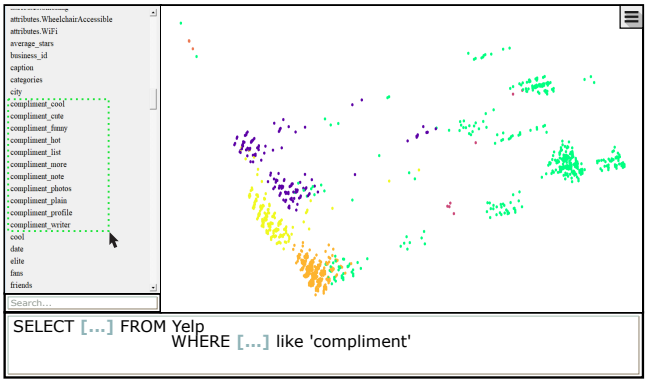


Figure 7: Points containing user selected attributes are highlighted green.

3.3 An Iterative Approach

At any time in our exploration pipeline analysts may stop where they are, take the knowledge they gained about their dataset, and restart the process from the beginning. Through exploring the Twitter dataset we found retweets and quoted tweets to have a high correlation, with this knowledge we can then start back at the beginning and depending on whether our task allows us to merge these attributes, we can then choose a more appropriate K value for KNN. In addition, attributes that have no relationship to core attributes, such as a users profile_image_url being present, can easily be pruned to compress our summary output. By incrementally learning about their dataset, an analyst can converge on the views required for their tasks.

4 RELATED WORK

Schema Extraction. Schema extraction for Json, as well as for other self-describing data models like XML has seen active interest from a number of sources. An early effort to summarize self-describing *hierarchical* data can be found in the LORE system's DataGuides [9]. DataGuides view schemas begin with a forest of tree-shaped schemas and progressively merge schemas, deriving a compact encoding of the forest as a DAG. Although initially designed for XML data, similar ideas have been more recently applied for Json data as well [13, 14]. Key challenges in this space involve simply extracting schemas from large, multi-terabyte collections of Json data [3], as well as managing ambiguity in the set of possible factorizations of a schema [2, 16]. For non-hierarchical data, interactive tools like Wrangler [12] provide an interactive frameworks for regularizing schemas.

Physical Layout. While schemas play a role in the interpretability of a Json data set, they can also help improve the performance of Json queries. One approach relies on inverted indexes [13] to quickly identify records that make use of sparse paths in the schema. Another approach is to normalize schema elements [7]. Although the resulting schema may not always be interpretable, this approach can result in substantial space savings.

5 FUTURE WORK

One challenge that we will need to address in SchemaDrill is coping with nested collections. At the moment, the user can manually merge collections of attributes that correspond to disjoint entites. However, we would like to automate this process. One observation is that a typical collection like an array has a schema with the general structure:

$$(P_1 \vee P_1P_2 \vee P_1P_2P_3 \vee \ldots) \;=\; (P_1 \wedge (\emptyset \vee P_2 \wedge (\emptyset \vee P_3 \wedge (\ldots))))$$

The version of this expression on the right hand side is notable as its closure over the semiring $\langle\ \{\{\mathbf{P}\}\}, \vee, \wedge, \emptyset, \{\{\}\}\ \rangle$ would indicate that the semiring is "quasiregular" or "closed", an algebraic structure best associated with the Kleene star. Hence, we plan to explore the use of the Kleene star to encode nested collections in our algebra. A key challenge in doing so is detecting opportunities for incorporating it into a summary, a more challenging form of the factorization problem.

A further step to increase the capabilities of SchemaDrill is to incorporate type information in the summarization. This adds an extra layer of information an analyst can extract from our system, as well as the ability to identify and correct schema errors. As a long term goal we will provide capabilities for linking views, for example by defining functional dependencies. The goal is to create full entity relationship diagrams. In particular, one interesting way to identify potential relationships that exist between entities is by leveraging the overlap between segments.

ACKNOWLEDGMENTS

This work was supported by NSF Awards IIS-1750460, ACI-1640864, SaTC-1409551, and by a gift from Oracle. 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 Oracle.

REFERENCES

[1] Roam Analytics. 2017. Prescription-based prediction. https://www.kaggle.com/roamresearch/prescriptionbasedprediction. (2017).
[2] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Counting types for massive JSON datasets. In *DBPL*. ACM, 9:1–9:12.
[3] Mohamed Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Schema Inference for Massive JSON Datasets. In *EDBT*. OpenProceedings.org, 222–233.
[4] Peter Bürgisser. 2001. The Complexity of Factors of Multivariate Polynomials. In *FOCS*. IEEE Computer Society, 378–385.
[5] E. F. Codd. 1979. Extending the Database Relational Model to Capture More Meaning. *ACM Trans. Database Syst.* 4, 4 (dec, 1979), 397–434. https://doi.org/10.1145/320107.320109
[6] Nilesh Dalvi and Dan Suciu. 2013. The Dichotomy of Probabilistic Inference for Unions of Conjunctive Queries. *J. ACM* 59, 6, Article 30 (jan, 2013), 87 pages. https://doi.org/10.1145/2395116.2395119
[7] Michael DiScala and Daniel J. Abadi. 2016. Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data. In *SIGMOD Conference*. ACM, 295–310.
[8] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. 2001. Graphviz - Open Source Graph Drawing Tools. In *Graph Drawing*. 483–484.
[9] Roy Goldman and Jennifer Widom. 1997. DataGuides: Enabling Query Formulation and Optimization in SemistructuredDatabases. In *VLDB*. Morgan Kaufmann, 436–445.
[10] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining Frequent Patterns without Candidate Generation. In *SIGMOD Conference*. ACM, 1–12.
[11] Stephen C. Johnson. 1967. Hierarchical clustering schemes. *Psychometrika* 32, 3 (01 Sep 1967), 241–254. https://doi.org/10.1007/BF02289588
[12] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2011. Wrangler: interactive visual specification of data transformation scripts. In *CHI*,

Desney S. Tan, Saleema Amershi, Bo Begole, Wendy A. Kellogg, and Manas Tungare (Eds.). 3363–3372. https://doi.org/10.1145/1978942.1979444

[13] Zhen Hua Liu and Dieter Gawlick. 2015. Management of Flexible Schema Data in RDBMSs - Opportunities and Limitations for NoSQL -. In *CIDR*. www.cidrdb.org.

[14] Zhen Hua Liu, Beda Christoph Hammerschmidt, Doug McMahon, Ying Liu, and Hui Joe Chang. 2016. Closing the functional and Performance Gap between SQL and NoSQL. In *SIGMOD Conference*. ACM, 227–238.

[15] Oracle. 2017. Database JSON Developer's Guide: JSON Data Guide. https://docs.oracle.com/cloud/latest/db122/ADJSN/json-dataguide.htm. (2017).

[16] William Spoth, Bahareh Sadat Arab, Eric S. Chan, Dieter Gawlick, Adel Ghoneimy, Boris Glavic, Beda Hammerschmidt, Oliver Kennedy, Seokki Lee, Zhen Hua Liu, Xing Niu, and Ying Yang. 2017. Adaptive Schema Databases. In *CIDR*.

[17] Twitter. [n. d.]. Decahose stream. https://developer.twitter.com/en/docs/tweets/sample-realtime/api-reference/decahose. ([n. d.]).

[18] Inc. Yelp. 2018. Yelp Open Dataset: An all-purpose dataset for learning. https://www.yelp.com/dataset. (2018).