Tabularizing Large Scale Semi-Structured Data

by

William Spoth

December, 2021

A dissertation submitted to the
Faculty of the Graduate School of the
State University of New York at Buffalo
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science and Engineering

© William Spoth 2021

All Rights Reserved

Acknowledgments

First, I would like to express my deepest gratitude to my advisor Dr. Oliver Kennedy, for all the amazing opportunities, support, and guidance he has provided me with over the years. He has been a constant source of inspiration and academic mentor, allowing me to improve my skills and explore new ideas. He has always been more than generous with his time, and commitment to my success. I would also like to thank all the great faculty at the University at Buffalo I have had the pleasure of learning from. Their passion in multiple fields of study have impacted deeply the way I approach problem solving and my interests in life.

Second, I would like to thank Zhen Hua Liu for constantly listening and providing constructive feedback to my research as a subject matter expert. I would also like to thank all my friends and colleagues, for lending me their ears to discuss academics and anything else.

Finally, I thank my parents, Michael and Francina, for their unconditional support and love at every step in my life. Their teachings have shaped me into the person I am today, and their hard work and dedication to my success is the reason I am fortunate enough to be where I am today. To my future wife Emily, you are the most amazing person I have ever met and truly the love of my life. I am thankful for all the sacrifices you have made to allow me to focus on my studies, and I look forward to spending the rest of my life with you. May God bless us, our friends, and our family in all future endeavors.

Abstract

Data workflows now more than ever, consist of fusing various sources, implementations, and formats, in a process known as data federation. Numerous popular database services provide a SQL interface for issuing queries against foreign tables and types, as SQL continues to be the most popular human-data interface. The column and row structure at SQL's core, while great for querying tabular data such as database tables and CSV files. Becomes less manageable when dealing with hierarchal or sparse data, common to semi-structured data like JSON. For example, querying JSON with SQL, even with native type support, requires the use of inefficient and specialized functions. Users would much rather import their JSON data as a shredded native table, simplifying queries, enabling SQL reuse, and improving performance. Unfortunately, JSON has many incompatibilities when shredding tabularly, such as arrays of objects, heterogeneous datasets, embedded key-value tables, and multi-typed columns to name a few.

This dissertation first investigates the feasibility of previously proposed automatic truth finding solutions. Evaluating both performance and quality of output with the goal of materializing a quasi-relational schema. Prior related schema generating work using similarity matching such as functional dependency and clustering, we found diverges significantly from ground truth. Second, we outline a probabilistic approach to incorporate user feedback and view sharing during data exploration. Enabling multi-user collaboration and definition of localized views to narrow search space. Third, we propose and experimentally verify a novel clustering algorithm to specifically target JSON data, taking advantage of attribute occurrence. We compare state-of-the-art schema extractors against our novel clustering algorithm, and show magnitudes improvement reducing schema search space. Unfortunately the number of datasets with

known ground truth is limited. To address this we, quantify human infer-ability for datasets with unknown ground truth, we devised three metrics, precision, recall, and grouping, measuring specificity, validity, and conciseness respectfully. Finally, we leverage our algorithm to generate partition schema candidates, enabling RDBMS tabular import for datasets that could otherwise not. Greatly improving query performance through partition elimination optimizations, automating a vital error prone process.

Contents

Acknowledgments **iv**

Abstract **vi**

1 Introduction **1**

1.1 Motivation 1

1.2 Background 3

1.3 Contributions 3

2 Adaptive Schema Databases **5**

2.1 Adaptive Schemas 5

2.2 Adaptive Extraction and Discovery 10

2.3 Adaptive, Personalized Schemas 13

2.4 Explanations and Feedback 18

2.5 Adaptive Organization 22

2.5.1 Materializing Personalized Schemas 22

2.5.2 Shared Materializations 23

2.6 Adaptive Schema in Action 24

2.6.1 Deterministic Extraction 25

2.6.2 Non-Deterministic Extraction 26

2.6.3 Evaluation 29

3 Interactive Semi-structured Schema Generation **31**

3.1 Interactive JSON 31

3.1.1 Extracting Relational Entities 33

3.1.2 Human-Scale $\mathtt{S^3D}$ . . . 34
3.1.3 Overview . . . 35
3.2 Summarization . . . 36
3.2.1 Data Model . . . 36
3.2.2 Paths as Attributes . . . 38
3.2.3 Schema Collections . . . 39
3.2.4 Summarizing Schema Collections . . . 41
3.3 Visualization . . . 44
3.3.1 Schema Segmentation . . . 46
3.3.2 Schema Exploration . . . 49
3.3.3 An Iterative Approach . . . 50
3.4 Alternative Extractors . . . 51
3.5 Recursive Schemas . . . 52

4 Putting the Schema back in Schema-on-Read 54
4.1 JSON Inconsistencies . . . 54
4.2 Extractor Notation . . . 57
4.2.1 Schema Discovery . . . 60
4.3 Ambiguous Schema Extraction . . . 62
4.3.1 Arrays as Tuple-Like Structures . . . 62
4.3.2 Objects as Collections . . . 63
4.3.3 Multi-Entity Collections . . . 64
4.4 Extraction Overview . . . 65
4.4.1 Naive Implementation . . . 66
4.4.2 SchemaDrill . . . 67
4.4.3 Helper Heuristics . . . 69
4.5 Detecting Collections . . . 70
4.5.1 Key-Space Entropy . . . 71
4.5.2 The Similar Types Constraint . . . 72
4.5.3 Differentiating Tuples and Collections . . . 73
4.5.4 Entropy For Arrays . . . 75
4.6 Multi-Entity Collections . . . 76

4.6.1 Entity Discovery 76
4.6.2 Bimax 78
4.6.3 Greedy Merge 80
4.6.4 Implementation 82
4.7 Schema Generation 83
4.7.1 Recall 88
4.7.2 Schema Entropy 89
4.7.3 Entity Detection 91
4.7.4 Runtime 93
4.7.5 Results 94
4.8 Alternate Extraction Techniques and XML 96

5 Tabular Layout **99**
5.1 Partitioning 99
5.2 RDBMS JSON Integration 100
5.2.1 Binary JSON 101
5.2.2 JSON Tables 101
5.3 Sparse Columns 102
5.3.1 Null Space 102
5.3.2 Pivot Tables 103
5.3.3 Performance and Usability 104
5.4 Predictive Partitioning 107
5.4.1 Background 108
5.4.2 Partition Algorithms 109
5.4.3 Apache Parquet 111
5.4.4 Partition Query Rewriting 113
5.4.5 Partitioning Performance 115
5.5 Partitioning Conclusions 116

6 Conclusions and Future Work **119**
6.1 Conclusions 119
6.2 Future Work 121

Bibliography **134**

Chapter 1

Introduction

1.1 Motivation

Each year the amount of data generated increases substantially, to quote the International Data Corporation, "The amount of digital data created over the next five years will be greater than twice the amount of data created since the advent of digital storage" [45]. IOT, micro-service architecture, distributed systems, the rise of machine learning, SaaS, and more, all contribute to the increase in data volume, formats, sources, and sanity. ETL is further complicated when accounting for popular semi-structured data formats like JSON, defining no global file level schema. This lack of global schema definition on write speeds up development, enables seamless schema changes, and is a popular choice among developers. However, lacking a schema at read time greatly increases the time spent manually inspecting files to verify correctness, uncover join candidates, and often requires a schema to be generated post-write anyways. For analysts, this creates a complex trade-off between information gain,

preparation cost, and query performance. To make matters worse, analysts now commonly source thousands or millions of files, spanning multiple formats, from multiple sources, and of varying cleanliness. As we march further and further into the era of big data, the need for automatic data cleaning solutions is apparent, aiding analysts in sensible schema generation, data validation, and user insight.

This work first investigates the feasibility of previously proposed automatic truth finding solutions. Evaluating both performance and quality of output with the goal of materializing a quasi-relational schema. Prior related schema generating work using similarity matching such as functional dependency and clustering, we found diverges significantly from ground truth. Second we outline a probabilistic approach to incorporate user feedback and view sharing during data exploration. Enabling multi-user collaboration and definition of localized views to narrow search space. Third we propose and experimentally verify a novel clustering algorithm to specifically target JSON data, taking advantage of attribute occurrence. We compare state-of-the-art schema extractors against our novel clustering algorithm, and show magnitudes improvement reducing schema search space. Unfortunately the number of datasets with known ground truth is limited. To address this we, quantify human infer-ability for datasets with unknown ground truth, we devised three metrics, precision, recall, and grouping, measuring specificity, validity, and conciseness respectfully. Finally, we leverage these schemas to generate partition candidates, enabling RDBMS tabular import for datasets that could otherwise not. Greatly improving query performance through partition elimination, automating a vital error prone process.

1.2 Background

JSON is only the latest flavor of semi-structured data, ongoing research into importing XML and graph data has been conducted for decades. Entity discovery has been explored extensively in hierarchical [15–17, 30, 39, 44, 62, 67, 79, 81, 89], graph [5], and object-exchange model (OEM) [39] data. XML schema discovery in particular [15–17, 44, 62] has been explored extensively. Each format contains unique features, common patterns, and constraints, however all pose a significant loading problem. Notably, these prior works emphasize necessity of generating a schema prior to loading. In practice this often means non-trivial data rewriting and transformation to incorporate hierarchies, arrays, embedded objects, and more.

Specifically there exist a number of JSON extractors using various techniques. Prior works have defined extraction as a grammar [10, 12], applied various machine learning techniques [5, 67, 81, 89], and mined relationships [30, 61, 95]. While these works present potential solutions for a number of different applications and datasets, there is yet to be a single extraction technique that is capable of adequately mapping the enormous set of features JSON is capable of encoding.

1.3 Contributions

Loading semi-structured data like JSON into a tabular format is time-consuming, difficult, error prone, and has dramatic performance implications. While most RDBMS contain special types for JSON data, the functionality and performance is often considerably worse, and poses significant additional query steps when merging with other data. In this work, we provide a mechanism for personalized multi-source data clean-

ing pipelines that adapt to user input and can be shared. Introduce a case study exhibiting the shortcomings of current JSON schema extractors, machine learning results, and data loading techniques. We improve upon the current JSON schema extraction process, enabling importing for complicated JSON relationships that would otherwise need human intervention. We introduce a novel schema extraction algorithm to take advantage of particular characteristics of JSON datasets. Finally, we compare and discuss various shortcomings and performance issues of current JSON loading processes. Demonstrating how schema declaration impacts performance and usability.

Chapter 2

Adaptive Schema Databases

2.1 Adaptive Schemas

The rigid schemas of classical relational databases help users in specifying queries and inform the storage organization of data. However, the advantages of schemas come at a high upfront cost through schema and ETL process design. In this work, we propose a new paradigm where the database system takes a more active role in schema development and data integration. We refer to this approach as *adaptive schema databases* (*ASDs*). An ASD ingests semi-structured or unstructured data directly using a pluggable combination of extraction and data integration techniques. Over time it discovers and adapts schemas for the ingested data using information provided by data integration and information extraction techniques, as well as from queries and user-feedback. In contrast to relational databases, ASDs maintain multiple *schema workspaces* that represent individualized views over the data, which are fine-tuned to the needs of a particular user or group of users. A novel aspect of

ASDs is that probabilistic database techniques are used to encode ambiguity in automatically generated data extraction workflows and in generated schemas. ASDs can provide users with context-dependent feedback on the quality of a schema, both in terms of its ability to satisfy a user's queries, and the quality of the resulting answers. We outline our vision for ASDs, and present a proof of concept implementation as part of the Mimir probabilistic data curation system.

Classical relational systems rely on schema-on-load, requiring analysts to design a schema upfront before posing any queries. The schema of a relational database serves both a navigational purpose (it exposes the structure of data for querying) as well as an organizational purpose (it informs storage layout of data). If raw data is available in unstructured or semi-structured form, then an ETL (i.e., Extract, Transform, and Load) process needs to be designed to translate the input data into relational form. Thus, classical relational systems require a lot of upfront investment. This makes them unattractive when upfront costs cannot be amortized, such as in workloads with rapidly evolving data or where individual elements of a schema are queried infrequently. Furthermore, in settings like data exploration, schema design simply takes too long to be practical.

Schema-on-query is an alternative approach popularized by NoSQL and Big Data systems that avoids the upfront investment in schema design by performing data extraction and integration at query-time. Using this approach to query semi-structured and unstructured data, we have to perform data integration tasks such as natural language processing (NLP), entity resolution, and schema matching on a per-query basis. Although it allows data to be queried immediately, this approach sacrifices the navigational and performance benefits of a schema. Furthermore, schema-on-query

incentivizes task-specific curation efforts, leading to a proliferation of individualized lower-quality copies of data and to reduced productivity.

One significant benefit of schema-on-query is that queries often only access a subset of all available data. Thus, to answer a specific query, it may be sufficient to limit integration and extraction to only relevant parts of the data. Furthermore, there may be multiple "correct" relational representations of semi-structured data and what constitutes a correct schema may be highly application dependent. This implies that imposing a single flat relational schema will lead to schemas that are the lowest common denominator of the entire workload and not well-suited for *any* of the workload's queries. Consider a dataset with tweets and re-tweets. Some queries over a tweet relation may want to consider re-tweets as tweets while others may prefer to ignore them.

In this work, we propose ***adaptive schema databases*** (*ASDs*), a new paradigm that addresses the shortcomings of both the classical relational and the Big Data approaches mentioned above. ASDs enjoy the navigational and organizational benefits of a schema without incurring the upfront investment in schema and ETL process development. This is achieved by automating schema inference, information extraction, and integration to reduce the load on the user. Furthermore, instead of enforcing one global schema, ASDs build and adapt idiosyncratic schemas that are specialized to users' needs.

We propose the probabilistic framework shown in Figure 2.1 as a reference architecture for ASDs. When unstructured or semi-structured data are loaded into an ASD, this framework applies a sequence of data extraction and integration components that we refer to as an **extraction workflow** to compute possible relational

schemas for this data. Any existing techniques for schema extraction or information integration can be used as long as they can expose ambiguity in a probabilistic form. For example, an entity resolution algorithm might identify two possible instances representing the same entity. Classically, the algorithm would include heuristics that resolve this uncertainty and allow it to produce a single deterministic output. In contrast, our approach requires that extraction workflow stages produce non-deterministic, probabilistic outputs instead of using heuristics. The final result of such an **extraction workflow** is a **set of candidate schemas** and a probability distribution describing the likelihood of each of these schemas. In ASDs, users create **schema workspaces** that represent individual views over the schema candidates created by the extraction workflow. The schema of a workspace is created incrementally based on queries asked by a user of the workspace. Outputs from the extraction workflow are dynamically imported into the workspace as they are used, or users may suggest new relations and attributes not readily available to the database. In the latter case, the ASD will apply schema matching and other data integration methods to determine how the new schema elements relate to the elements in the candidate schemas, and attempt to synthesize new relations or attributes to match. Similar to extraction workflows, the result of this step is probabilistic. Based on these probabilities and feedback provided by users through queries, ASDs can incrementally modify the extraction workflow and schema workspaces to correct errors, to improve their quality, to adapt to changing requirements, and to evolve schemas based on updates to input datasets. The use of feedback is made possible based on our previous work on probabilistic curation operators [92] and provenance [8]. By modelling schemas as views over a non-relational input dataset, we decouple data representation from

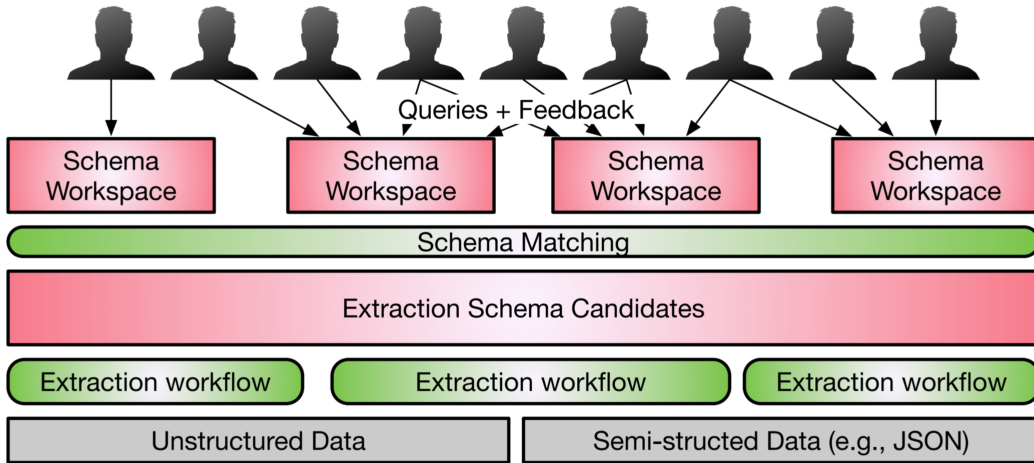


Figure 2.1: Overview of an ASD system

content. Thus, we gain flexibility in **storage organization** — for a given schema we may choose not to materialize anything, we may fully materialize the schema, or materialize selectively based on access patterns.

Concretely, we investigate the following:

- We introduce our vision of ASDs, which enable access to unstructured and semi-structured data through personalized relational schemas.
- We show how ASDs leverage information extraction and data integration to automatically infer and adapt schemas based on evidence provided by these components, by queries, and through user feedback.
- We show how ASDs enable adaptive task-specific "personalized schemas" through schema workspaces which are probabilistic relational views over semistructured or unstructured input datasets.
- We illustrate how ASDs communicate potential sources of error and low-quality data, and how this communication enables analysts to provide feedback.
- We present a proof of concept implementation of ASDs based on the *Mimir* [66]

data curation system.

- We demonstrate through experiments that the instrumentation required to embed information extraction into an ASD has minimal overhead.

2.2 Adaptive Extraction and Discovery

An ASD allows users to pose relational queries over the content of semi-structured and unstructured datasets. We call the steps taken to transform an input dataset into relational form an **extraction workflow**. For example, one possible extraction workflow is to first employ *natural language processing* (NLP) to extract semi-structured data (e.g., RDF triples) from an unstructured input, and then shred the semi-structured data into a relational form. The user can then ask queries against the resultant relational dataset. Such a workflow frequently relies on heuristics to create seemingly deterministic outputs, obscuring the possibility that the heuristics may choose incorrectly. In an ASD, one or more modular information extraction components instead produce a *set* of possible ways to shred the raw data with associated probabilities. This is achieved by exposing ambiguity arising in the components of an extraction workflow. Any NLP, information retrieval, and data integration algorithm may be used as an information extraction component, as long as the ambiguity in its heuristic choices can be exposed. The set of schema candidates are then used to seed the development of schemas individualized for a particular purpose and/or user. The ASD's goal is to figure out which of these candidates is the correct one for the analyst's current requirements, to communicate any potential sources of error, and to adapt itself as those requirements change.

Extraction Schema Candidates. When a collection of unstructured or semi-structured datasets D is loaded into an ASD, then information extraction and integration techniques are automatically applied to extract relational content and compute candidate schemas for the extracted information. The choice of techniques is based on the input data type (JSON, CSV, natural language text, etc...). We associate with this data a **schema candidate set** $\mathcal{C}_{ext} = (\mathbf{S_{ext}}, P_{ext})$ where $\mathbf{S_{ext}}$ is a set of candidate schemas and P_{ext} is a probability distribution over these schemas. We use S_{max} referred to as the **best guess schema** to denote $\arg\max_{S \in \mathbf{S_{ext}}}(P(S))$, i.e., the most likely schema from the set of candidate schemas. Similar data models have been studied extensively in probabilistic databases [40], allowing us to adapt existing work on probabilistic query processing [80], while still supporting a variety of techniques for information extraction [31], natural language processing [32], data integration [35, 36, 49], and more.

Example 1 *As a running example throughout the paper, consider a JSON document (a fragment is shown below) that stores a college's enrollment. Assume that for every graduate student we store name and degree, but only for some students there is a record of the number of credits achieved so far. For undergraduates we only store a name, although several undergraduates were accidentally stored with a degree. A semi-structured to relational mapper may extract schema candidates as shown in Figure 2.2.*

```
{"grad":{"students":[
  {name:"Alice",deg:"PhD",credits:"10"},
  {name:"Bob",deg:"MS"}, ...]},
 "undergrad":{"students":[
  {name:"Carol"},{name:"Dave",deg:"U"}, ...]}}
```

Querying Extracted Data. We would like to expose to users an initial schema that allows D to be queried (i.e., the best guess schema S_{max}), while at the same time acknowledging that this schema may be inappropriate for the analyst, incorrect for her current task, or simply outright wrong. Manifestations of extraction errors appear in three forms: (1) A query incompatible with S_{max}, (2) An update with data that violates S_{max}, or (3) An extraction error resulting in the wrong data being presented to the user. The first two errors are overt and, thus easy to detect automatically. In both cases, the primary challenge is to help the user to determine whether the operation was correct, and if necessary, to repair the schema accordingly. Here, the distribution P_{ext} serves as a metric for schema suggestions. Given a query (resp., update or insert) Q, the goal is to compute $\arg\max_{S \in \mathbf{S_{ext}} \wedge S \models Q}(P(S))$, where the $S \models Q$ denotes compatibility between schema and query, i.e., the schema contains the relations and attributes postulated by the query. While S_{max} has the highest probability of all schema candidates in $\mathbf{S_{ext}}$, that does not imply that it has the highest probability with respect to the schema elements mentioned in the query. Thus, we use S_{max} as a generic best guess to enable the user to express queries at first, but then adapt the best schema over time. Note that the personalized schemas we introduce in the next section even allow queries to postulate new relations and attributes.

Detecting extraction errors is harder and typically only possible once issues with the returned query result are discovered. Rather, such errors are most often detected as a result of inconsistencies observed while the analyst explores her data. Thus, the goal of an ASD is to make the process of detecting and repairing extraction errors as seamless as possible. Our approach is based on pay-as-you-go or on-demand

Student

Name
Alice
Bob
Carol
Dave

(a) P = 0.19

Student

Name	Deg
Alice	PhD
Bob	MS
Carol	(null)
Dave	U

(b) P = 0.27

Undergrad

Name
Carol
Dave

Grad

Name
Alice
Bob

(c) P = 0.22

Undergrad

Name	Deg
Carol	(null)
Dave	U

Grad

Name	Deg	Credits
Alice	PhD	10
Bob	MS	(null)

(d) P = 0.32

Figure 2.2: Extracted Schema Candidate Set and Data

approaches to curation [49, 66, 91], and is the focus of Sections 2.4 and 2.6 below.

2.3 Adaptive, Personalized Schemas

An ASD maintains a set of **schema workspaces** $\mathcal{W} = \{W_1, \ldots, W_n\}$. Each workspace W_i has an associated mapped context $\mathcal{C}_i = (S_i, \mathcal{M}_i, P_i)$ where S_i is a schema, $\mathcal{M}_i$ is a set of possible schema matchings [14], each between the elements of S_i and one $S \in \mathbf{S_{ext}}$, and P_i assigns probabilities to these matches. In the future we will lift the restriction to schema matches and allow the relationship between the extracted schema and the schema of a workspace to be more complex than that (e.g., expressed as a schema mapping [36]). Users may maintain their own personal schema workspace or share workspaces within a group of users that have common interests (e.g., a business analyst workspace for the sales data of a company). We plan to provide version control style features for the schemas of workspaces including access to data through past schema versions and importing of schema elements from one workspace into another. Recent work on schema evolution [26] and schema versioning demonstrates that it is possible to maintain multiple versions of schemas in parallel

where data is only stored according to one of these schemas. Specifically, we plan to extend our own work on flexible versioning of data [68] and flexible schema extensions for SQL [56].

Importing Schema Elements. Initially, the schema of a workspace is created empty. When posing a query over an extracted dataset, the user can refer to elements from schema S_{max}, schema S_i, or new relations and attributes that do not occur in either. References of the first two types are resolved by applying the extraction workflow to compute the instances for these schema elements (and potentially mapping the data based on the matches in $\mathcal{M}_i$). If a query references schema elements from S_{max}, then these schema elements are added to the current workspace schema plus one-to-one matches with probability 1 between these elements in S_{max} and S_i are added to the matching $\mathcal{M}_i$.

A query may also remain agnostic to the specific schema elements it requires, and instead declaratively provide query goals in terms of higher-order logical primitives. That is, the user postulates the existence of schema elements (which is expressible in second-order logic) and part of their structure (e.g., attributes that are referenced) without having to fully qualify them. This constrains the schema workspace, but still allows the workspace to adapt and evolve over time. A concrete example of this idea can be found in the flexible schema data extensions for SQL [56] (FSD-SQL). FSD-SQL allows query authors to remain agnostic to the exact physical structure of inter-attribute relationships, automatically adapting the query structure as needed.

Example 2 *Consider the following FSD-SQL query, which returns all students in the PhD program:*

```
01 | SELECT name FROM Grad
```

```
02 | WHERE json_exists(deg == 'PhD')
```

The data initially shows that each student has only one degree — The best schema is one in which there is a 1-to-1 mapping between student and degree and **deg** *is an element of the Grad relation. Thus, the above query is equivalent to the classical SQL query:*

```
01 | SELECT name FROM Grad WHERE deg = 'PhD'
```

However, let's say that the data also contains the following student, registered for both programs.

```
{ name: "Eve", deg: ["MS", "PhD"] }
```

With this new data, it may be appropriate to represent the degree field by a many-to-one relationship, and the equivalent query becomes a more complex primary-key to foreign-key join:

```
01 | SELECT g.name FROM Grad g WHERE EXISTS (
02 |   SELECT * FROM GradDeg d
03 |   WHERE g.id = d.id AND d.deg = 'PhD' )
```

A language construct like **json_exists** *remains agnostic to which underlying representation is used, creating a query that is more resilient to schema evolution and supporting a broader range of possible schemas.*

Probabilistic Semantics of ASD Queries. Note that ASD queries are inherently probabilistic, as the result varies depending on the distribution of possible extractions. However, we do not have to overwhelm the user with full probabilistic query semantics. Instead, we apply the approach from [66,92] to return a deterministic best guess result

based on S_i and expose uncertainty through user interface cues [53] and through human-readable explanations generated on-demand.

Example 3 *Continuing with our running example, assume a user operating in workspace* W_1 *would like to retrieve all the names of students based on the enrollment JSON document. One option the user can take is to query the relations exposed by the best guess schema* S_{max}*. For instance, one way to express this query over the schema in Figure 2.2 is:*

```
01 | SELECT name FROM Undergrad UNION
02 | SELECT name FROM Grad
```

To process this query, the ASD would run the extraction workflow to create the relational content of the Undergrad *and* Grad *relations (it would be sufficient to create the projections of these relations on* name *only). The query is then evaluated over these extracted data. As a side-effect, by accessing these schema elements, the user declares interest in them and they are added to the schema workspace. Note that only accessed attributes are added to the workspace. If the workspace's schema was empty before, the resulting schema would be* $S_1 = \{\mathsf{Undergrad}(\mathsf{name}), \mathsf{Grad}(\mathsf{name})\}$*. Additionally, in* $\mathcal{M}_1$ *these elements are matched with their counterpart in* S_{max}*. If afterwards the user retrieves the degree of a graduate student then the* Grad *relation's schema would become* (name, deg)*.*

Declaring New Schema Elements. So far we have only discussed the case where a query refers to existing schema elements (either in the user schema or the extracted schema). If a query uses schema elements that are so far unknown, then this is interpreted as a request by the user to add these schema elements to the schema

workspace. It is the responsibility of the ASD to determine how schema elements in the extracted schema are related to these new elements. Any existing schema matching (and mapping discovery) approach could be used for this purpose. For instance, we could complement schema matching with schema mapping discovery [23, 88] to establish more expressive relationships between schema elements. Based on such matches we can then rewrite the user's query to access only relations from the extracted schema using query rewriting with views (a common technique from virtual data integration [41]) or materialize its content using data exchange techniques [36]. Again we take a probabilistic view by storing all possible matches with associated probabilities and choosing the matches with the highest probability for the given query.

Example 4 *Assume that a user would like to find names of students without having to figure out which relations in S_{max} store student information. A user may ask:*

```
01 | SELECT name FROM Student
```

Since relation Student *occurs in neither S_1 nor S_{max}, the ASD would run a schema matcher to determine which elements from S_{max} match with* Student *and its attribute* name*, for instance by probabilistically combining the* name *attribute of* Grad *and* Undergrad *as in the query from Example 3.*

In the example above, three Student(name) relations could reasonably be extracted from the dataset: One with just graduate students, one with just undergraduates, and one with both. Although it may be possible to heuristically select one of the available extraction options, it is virtually impossible for a single heuristic to cover all use cases. Instead, ASDs use heuristics only as a starting point for schema definitions. An ASD

decouples its information extraction heuristics from the space of possible extractions that could be emitted. In the next section, we present how these uncertain heuristic choices can be validated or corrected as needed in a pay-as-you-go manner [49, 66]. Note that we can use any existing schema matching algorithm to create schema matching $\mathcal{M}_i$ as long as it can be modified to expose probabilities. As we have demonstrated in previous work this assumption is reasonable — Mimir [66] already supports a simple probabilistic schema matching operator.

2.4 Explanations and Feedback

Allowing multiple schemas to co-exist simultaneously opens up opportunities for ambiguity to enter into an analyst's interaction with the database. To minimize confusion, it is critical that the analyst be given insight into how the ASD is presently interpreting the data.

Our approach to communicating the ASD's decisions leverages our previous work on: (1) explaining results of probabilistic queries and data curation operators in Mimir [53, 66, 92], and (2) provenance frameworks for database queries [8, 68]. We discuss the details of these systems along with our proof of concept implementation in Section 2.6. An ASD must be able to: (1) Warn the analyst when ambiguity could impact her interaction, (2) Explain the ambiguity, (3) Evaluate the magnitude of the ambiguity's potential impact, and (4) Assist the analyst in resolving the ambiguity. In this section, we explore how an ASD can achieve each of these goals in the context of three forms of interaction between the ASD and the outside world: Schema, Data, and Update.

Schema Interactions. Schema interactions are those that take place between the analyst and the ASD as she composes queries and explores the relations available in her present workspace. Recall that referencing a relation that does not exist in the workspace and extraction schema S_{max} does not necessarily constitute a problem since this triggers the ASD to add this relation to the workspace and figure out which relations in $\mathbf{S_{ext}}$ it could be matched with. However, it may be the case that no feasible match can be found. This either means that the user is asking for data that is simply not present in the dataset D or that errors in the extraction workflow or matching caused the ASD to miss the correct match. To explain the failure, we may provide the user with a summary of why matching with $\mathbf{S_{ext}}$ failed. For example, there are no relations with similar names in $\mathbf{S_{ext}}$.

Data Interactions. Data interactions happen when the ASD produces query results. Here, ambiguity can typically not be detected by static analysis and is often hidden behind multiple layers of aggregation and projection. For example, the query in Example 4 can have three distinct responses, depending on which relations from the extraction schema are matched against Student relation in the workspace that the analyst is currently using. At this level un-intrusive interface cues [53] are critical for alerting the analyst to the possibility that the results she is seeing may not be what she had in mind. The Mimir system uses a form of attribute granularity provenance [66, 92] to track the effects of sources of ambiguity on the output of queries. In addition to flagging potentially ambiguous query result cells and rows (e.g., attributes computed based on a schema match that is uncertain), Mimir allows users to explore the effects of ambiguity through both human-readable explanations of their causes and statistical precision measures like standard deviation and confidence scores. Linking results to

sources of ambiguity also makes it easier for the analyst to provide feedback that resolves the ambiguity. In Section 2.6 we show how we leverage Mimir to streamline data interactions in our prototype ASD.

Update Interactions. Finally, update interactions take place when the ASD receives new data, or changes to existing data. In comparison to the other two cases, there may not be an analyst directly involved in an update interaction like a nightly bulk data import. Thus, the ASD must be able to communicate the ambiguity arising from update interactions to analysts indirectly. The main problem with updates is that the extraction schema candidates $\mathbf{S_{ext}}$ and its probability distribution P_{ext} may get out of sync with the data it is describing. For example, when extracting JSON schemas, Oracle's DataGuides [57] transparently upgrade the type of a primitive-valued object to a singleton array if necessary for compatibility. Workspaces that have already imported mappings to the object expect it to be a primitive value.

However, rather than blocking an insertion or update which does not conform with the extraction schema outright, the ASD will represent schema mismatches as missing values when data is accessed through the out of sync schema. Alternatively, we can attempt to resolve data errors with a probabilistic repair. For example, an array of primitive values can be coerced into a primitive value by the probabilistic repair-key operation [6], allowing us to once again leverage probabilistic data curation systems like Mimir for explanations and feedback. However, because of the possibility that the schema is incorrect, feedback on these curation steps includes an additional two options for the analyst. In addition to repairing the potential data error, the analyst can choose to upgrade her workspace's schema to match the changes, or may choose to checkpoint her workspace and ignore new updates.

The ASD can also adjust the information extractor to adapt the schema candidate set $\mathcal{C}_{ext}$ and its probability distribution. However, this change could invalidate the matches of an existing workspace. For instance, consider an extracted schema that contains a relation Student(name,credits). If subsequent updates to the dataset insert many students without credits, then eventually the relation Student(name,credits) should be replaced with Student(name). If a workspace schema contains an attribute matched to Student.credits, then this attribute is no longer matched with any attribute from the extraction schema. When explaining missing values to the user, we plan to highlight their cause, whether they are the result of data that can not be cast to the current schema, or of an orphaned workspace attribute (it is no longer matched to any attribute in $\mathcal{C}_{ext}$).

User Feedback. We envision letting the user provide various kind of feedback about errors in query results such as marking sets of invalid attribute values and unexpected rows. Based on provenance we can then back-propagate this information to interpret these as feedback on matches and extraction workflow decisions. To determine a precise method for determining the best fixes based on such information is an interesting avenue for future work. Continuing with our running example, assume that the user when postulating the existence of a `Student` relation was only interested in the names of graduate students. If the ASD has matched the student relation against both `Undergrad` and `Grad`, then the result will also include undergraduates. To indicate that there is a problem, the user can mark some of the undergraduate names in the result as erroneous. The ASD can back-propagate these markers to the inputs using provenance (e.g., see [20, 25]). In the example, these back-propagated markers all annotate data that is in the result based on the match between `Student`

and `Undergrad`. Thus, they provide evidence against this schema match and the ASD may decide to remove the match.

2.5 Adaptive Organization

A classical RDBMS determines the physical organization of data based on its schema. This leads to excellent query performance and good storage utilization, because the schema information can be exploited to choose a beneficial physical design. For instance, using this approach it is not necessary to store schema information with each row, since all rows share the same structure and may be interpreted in the same way. However, this approach has the disadvantage that even small schema changes may necessitate physical reorganization of large amounts of data. Conversely, storing data in its native semi-structured or unstructured form does not enforce a fixed schema for the data and, thus, no physical reorganization is needed when the schema evolves. However, this flexibility comes at the cost of reduced query performance as the raw data needs to be decoded and transformed at runtime, and at the cost of higher storage requirements.

2.5.1 Materializing Personalized Schemas

From a user's perspective, data in a personalized schema is relational, i.e., SQL queries are treated as queries that access relational views. Under the hood, when a query accesses unstructured or semi-structured data, the data is transparently transformed into relational form, e.g., through an SQL-standard flatting operation for semi-structured data such as `JSON_TABLE()`. Thus, ASDs effectively decouple the data

format used for storage (the unstructured or semi-structured input datasets) from the data format used for querying (in the format defined by a schema workspace). Technically, it would be sufficient to just store the input datasets and generate data according to a workspace schema at query time. This corresponds to the second method mentioned above. Alternatively, we could materialize data according to a workspace schema eagerly after each change to the schema. This is the approach taken by traditional data warehousing. In ASDs any (partial) relational workspace schema is essentially a view over the semi- or unstructured input datasets. Thus, we have the freedom to materialize such a view just-in-time when it is requested or drop it if it is no longer deemed beneficial. Existing adaptive physical design and caching techniques [18, 28, 46, 63, 96] can be utilized to make such decisions. For example, these "views" can be materialized as in-memory data structures [57] or be stored as disk-resident materialized views and indexes as illustrated in [56].

Having this flexibility enables an ASD to adapt to usage patterns. During periods of frequent data evolution, materialized schemas require additional efforts to be kept up to date with the evolving schema and data. If this effort exceeds the performance benefit for queries, then these structures should be dropped. In ASDs this is not a problem, since these structures can be re-created from the input datasets at any point in time. In other words, in ASDs, the schema-based storage approach is merely an optional cache that we can fully control.

2.5.2 Shared Materializations

Since ASDs will host multiple schema snapshots from multiple workspaces over the same raw input data, it is advantageous to extensively share materializations across

schemas. The schemas discovered by ASDs can be used to guide performance tuning by caching content of relations that are used frequently and are stable (their schema and content has not been modified recently). In this regard, it will be important to determine differences among schema snapshots from the same and from different workspaces to identify opportunities for sharing and to update cached data through incremental materialized view maintenance techniques. Additionally, we can leverage techniques from revision control systems, such as, copy-on-write, storing change deltas, and materializing at large change boundaries, to organize the shared physical cache for schema snapshots.

Furthermore, we can cache the outputs of data extraction or projections over this output to benefit multiple workspace schema elements. This leads to interesting optimization problems because of the sharing of elements, a problem analogous to the view selection problem [63]. The difference to other automated approaches for tuning physical design is that the sharing of such elements by workspace schemas is encoded in their matchings, which simplifies the identification of sharing opportunities. It remains to be seen whether this information can be exploited to devise caching strategies that are fine tuned for ASDs.

2.6 Adaptive Schema in Action

We now outline a proof of concept ASD implementation, leveraging the Mimir [66, 91] system for its provenance and feedback harvesting capabilities. For this preliminary implementation, we chose to implement a probabilistic version of the normalizing relational data extractor for JSON data recently proposed by DiScala and Abadi [31].

This extractor uses heuristics based on functional dependencies (FDs) between the objects' attributes to create a narrow, normalized relational schema for the input data. We use this extractor as a proof of concept extraction workflow in our prototype.

2.6.1 Deterministic Extraction

The DiScala extractor [31] runs on collections of JSON objects with a shared schema. The first step in the extraction process is to create a functional dependency (FD) graph from the objects. The nodes in a dependency graph represent attributes and an edge from attribute A to attribute B denotes that a functional dependency $A \rightarrow B$ approximately holds. The objects are first flattened, discarding nesting structure and decomposing nested arrays, resulting in a single, very wide and very sparse relation. The extractor creates a FD graph for this relation using a fuzzy FD detection algorithm [47], originally proposed by Ilyas et. al., that keeps it resilient to minor data errors. Any subtree of this graph can serve as a relation, with the root of the tree being the relation's key (since it implies all attributes in this subtree). At this point, the original, deterministic DiScala extractor heuristically selects one set of relations upfront to use as a *final* target schema.

In a second pass, the extractor attempts to establish correlations between the domains of pairs of attributes. Relations with keys drawn from overlapping domains become candidates for being merged together. Once two potentially overlapping relations are discovered, the DiScala extractor uses constraints given by FDs to identify potential mappings between the relation attributes.

2.6.2 Non-Deterministic Extraction

The DiScala extractor makes three heuristic decisions: (1) Which relations to define from the FD graph, (2) Which relations to merge together, and (3) How to combine the schemas of merged relations. In an ASD, such heuristics are expected to serve only as a rough, first draft of the schema, and not as a final, correct representation of the data. Errors in the first of these heuristics appear in the visible schema, and as discussed in Section 2.2 are easy to detect and resolve. The remaining heuristics can cause data errors that are not visible until the user begins to issue queries. As a result, the primary focus of our proof of concept is on the latter two heuristics.

Concretely, our prototype ASD generalizes the DiScala extractor by providing: (1) Provenance services, allowing users to quickly assess the impact of the extractor's heuristics on query results, (2) Sensitivity analysis, allowing users to evaluate the magnitude of that impact, and (3) Easy feedback, allowing users to easily repair mistakes in the extractor's heuristics. We leverage a modular data curation system called Mimir [66, 91, 92] that encodes heuristic decisions through a generalization [52] of a common encoding of ambiguous and incomplete data called C-Tables [40, 48]. A relation in Mimir may include placeholders, or *variables* that stand in for heuristic choices. During normal query evaluation, variables are replaced according to the heuristic. However, the terms themselves allow the use of program analysis as a form of provenance, making it possible to communicate the presence and magnitude of heuristic ambiguity in query results, and also allow users to easily override heuristic choices.

Example 5 *Consider the task of mapping a source relation $R(A, B)$ to a target relation $R'(C)$. Mimir's schema matcher uses a query of the form:*

```
01 | SELECT CASE {C}
02 |            WHEN 'A' THEN A
03 |            WHEN 'B' THEN B
04 |            ELSE NULL
05 |          END AS C
06 | FROM R
```

where {C} denotes a variable with domain {'A', 'B', NULL} that selects between the possible input columns, or declares that there is no match.[1] *The resulting C-Table includes a labeled null for each output row that can take the values of* `R.A`, `R.B`, *or* `NULL`, *depending on how the model assigns variable values. Consider this example instance:*

R	A	B
	1	2
	3	4

R'	C
	{C}='A' ? 1 : ({C}='B' ? 2 : NULL)
	{C}='A' ? 3 : ({C}='B' ? 4 : NULL)

Conceptually, every value of $R'.C$ is expressed as deferred computation (a future). A valuation for $\{C\}$ allows R' to be resolved into a classical relation. Conversely, program analysis on the future links each value of $R'.C$, as well as any derived values back to the choice of how to populate C.

Heuristic data transformations, or *lenses*, in Mimir consist of two components. First, the lens defines a view query that serves as a proxy for the transformed data with variables standing in for deferred heuristic choices. Second, a model component abstractly encodes a joint probability distribution for each variable through two operations: (1) Compute the most likely value of the variable, and (2) Draw a ran-

[1] Currently, the schema matcher assumes that one target attribute can only be matched against one source attribute.

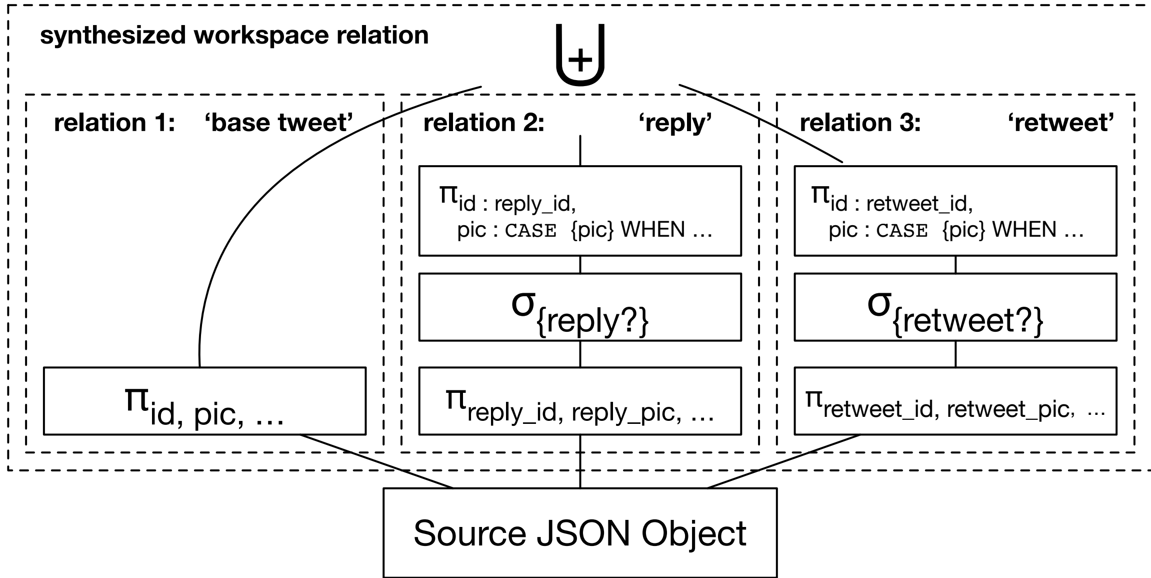


Figure 2.3: Structure of an extracted relation's merged view

dom sample from the distribution. Additionally, the model is required to provide a human-readable explanation of the ambiguity that the variable captures.

For this proof of concept we adopt an interaction model where the ASD dynamically synthesizes workspace relations by extending one extracted relation (the primary) with data from extracted relations containing similar data (the secondaries). Figure 2.3 illustrates the structure of one such view query: The primary and all possible secondaries are initialized by a projection on the source data. A single-variable selection predicate (i.e., {`relation?`}) reduces the set of secondaries included in the view. Second, a schema matching projection as illustrated in Example 5 adapts the schema of each secondary to that of the primary.

Our adapted DiScala extractor interfaces with this structure by providing models for relation- and schema-matching, respectively. We refer the reader to [31] for the

details of how the extractor computes relation and attribute pairing strength. The best-guess operations for both models use the native DiScala selection heuristics, while samples are generated and weighted according to the pairing strength computed by the extractor. By embedding the DiScala extractor as a lens, we are able to leverage Mimir's program analysis capabilities to provide feedback. When a lens is queried, Mimir highlights result cells and rows that are ambiguous [53]. On-request, Mimir constructs human-readable explanations of why they are ambiguous, as well as statistical metrics that capture the magnitude of the potential result error. Crucially, this added functionality requires only lightweight instrumentation and compile-time query rewrites [66].

2.6.3 Evaluation

Mimir and the prototype ASD are implemented in Scala 2.10.5 being run on the 64-bit Java HotSpot VM v1.8-b132. Mimir was used with SQLite as a backend. Tests were run multi-threaded on a 12x2.5 GHz Core Intel Xeon running Ubuntu 16.04.1 LTS. The primary goal of our experiments is to evaluate the overhead of our provenance-instrumented implementation of the DiScala extractor compared to the behavior of the classical extractor. We used a dataset, **TwitterM** consisting of 200 columns and **TwitterW** consisting of 400 columns of twitter JSON data. The extractor was run on 100,000 rows taken from Twitter's Firehose. Figure 2.4 shows the performance of the extractor. We show pre-processing time, the time necessary to compile and evaluate `SELECT * FROM workspace_relation`, and the same query against a table containing the output of the classical DiScala extractor. Note that both these queries return the same set of results, but the former is instrumented,

Dataset	Time		
	Precompute	ASD	Classic
TwitterM	214s (1.34)	1.26s (0.06)	0.31s (0.04)
TwitterW	625s (36.9)	1.49s (0.05)	0.28s (0.0012)

Figure 2.4: Overhead of the provenance-aware extractor (average of the trail runs, standard deviations are in parenthesis)

allowing it to provide provenance and feedback capabilities. Runtime for the instrumented extractor is a factor of 2 larger than the original, but is dominated by fixed compile-time costs.

Chapter 3

Interactive Semi-structured Schema Generation

3.1 Interactive JSON

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. 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.

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 [12, 30, 39, 55, 78] 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., [30, 39]) 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. Further, we explore several design options for SchemaDrill, and discuss how each interacts with the challenges of $\mathtt{S^3D}$.

3.1.1 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 [24], 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 $\mathtt{S^3D}$ problem reduces to finding such groups (e.g., by using Functional Dependencies [30]).

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.

3.1.2 Human-Scale $\texttt{S}^3\texttt{D}$

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

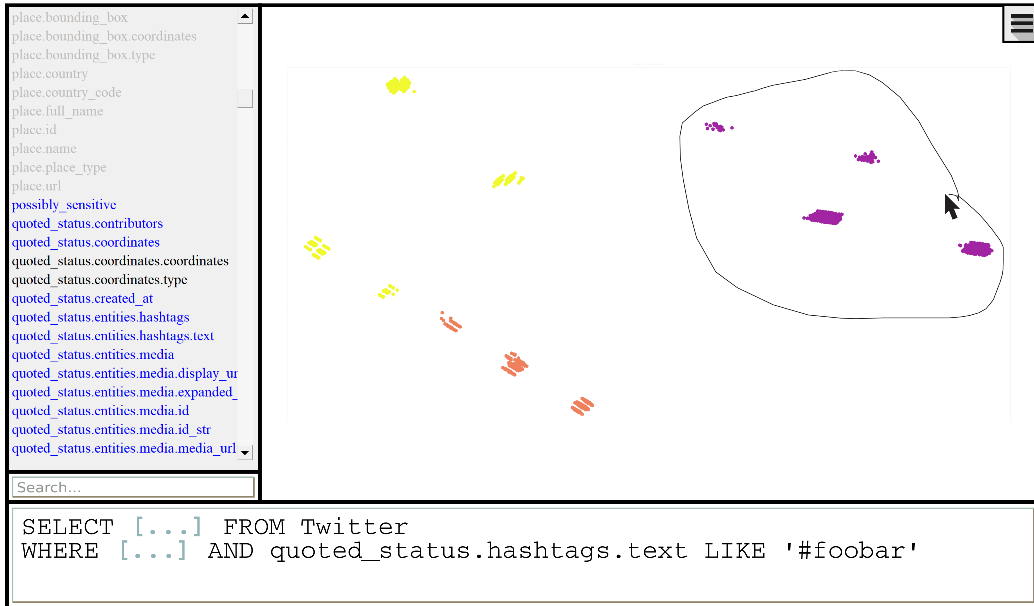


Figure 3.1: Prototype user interface for SchemaDrill.

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.

3.1.3 Overview

Figure 3.1 shows the prototype interface of SchemaDrill. The pane on the left, discussed in Section 3.2, shows the schema of the currently selected JSON view, highlighting attributes and groups based on relevance. The pane on the right, discussed in Section 3.3, 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 3.3, we show examples of how SchemaDrill facilitates incremental, iterative

exploration and mapping of JSON schemas.

3.2 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.

3.2.1 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 6 *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 $P \perp S$).

3.2.2 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 7 *The path set of the JSON object from Example 6 includes the paths:*

1. `tweet.text`
2. `tweet.user.friends_count`
3. `tweet.user.id`

[1]modulo empty objects or arrays

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 $S_1^{\perp} \subseteq S_2^{\perp}$.

3.2.3 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 [58, 69], is to simply present the set of all paths that appear anywhere in this collection. We call this the *joint schema* $\mathbb{S}$:

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

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

[2]One dataset [4] achieved 2.4 thousand paths through nested collections of objects.

anti-cooccurence relationships between attributes.

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

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

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

conventions: (1) When clear from context, a schema S denotes its own singleton set, and (2) We write $P_1 P_2$ 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 $P_1 P_2 \vee P_2 P_3$.

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.

3.2.4 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

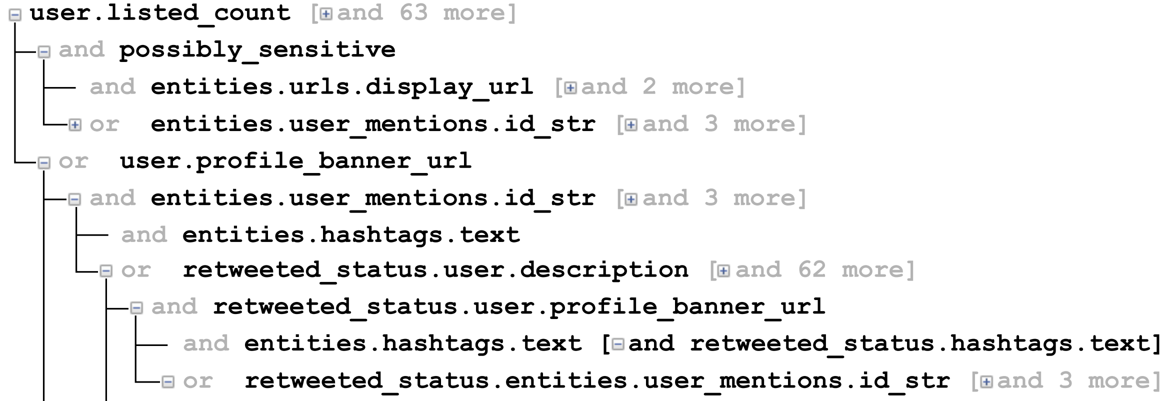


Figure 3.2: FP-Tree based schema summaries

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 [29], a problem that is, in general, worse than P-time [21]. Hence, approximations are required. We consider two approaches and allow the user to select the most appropriate one for their needs. The first is based on Frequent Pattern Trees [42] (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 [42].

Example 8 *Figure 3.2 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 correlations and anticorrelations. 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 a 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 [50], and a user-controlled threshold on the covariance, we cluster the attributes by parenthesizing. For example, clustering might group P_1 with P_2 and likewise P_3 with P_4. We can rewrite this formula as:

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

```
user.listed_count [⊞and 63 more]
entities.user_mentions.id_str [⊞and 3 more] [⊞or 3 more]
possibly_sensitive [⊟]
 or user.profile_banner_url
retweeted_status.user.description [⊞and 66 more]
```

Figure 3.3: RCDC based schema summaries

Observe that this formula omits schemas that the original formula captures (e.g., any schema including P_1 but not P_2). However, because clustering ensures that attributes within a group co-occur frequently, there are comparatively few such schemas.

We next repeat the process with a new covariance matrix built using the frequency of co-occurrence of *groups* (like P_1P_2). As before, we create clusters, but this time we cluster based on extremely negative co-variances. Hence, members of the resulting clusters are unlikely to co-occur. Continuing the example, let us assume that P_1P_2 and P_3P_4 are highly anti-correlated. Approximating and simplifying, we get an expression in RCDC form.

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

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.

3.3 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:

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

Figure 3.4: Segmentation breaking up schemas.

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.

3.3.1 Schema Segmentation

Specifically, we want to help the user to focus on particular 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*. We define a subschema s as a set of attributes, where s is contained in one or more source schemas. The s-segment of source schemas $S_1, \ldots, S_N$ to be the subset containing s:

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

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 3.4. Figures 3.4a illustrates the full schema summary as a tree, while Figure 3.4b shows a partial summary identified by the user using the lasso tool we describe shortly.

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 [34]). Cliques in the graph represent commonly co-occurring subschemas that might form segments of interest. This includes every

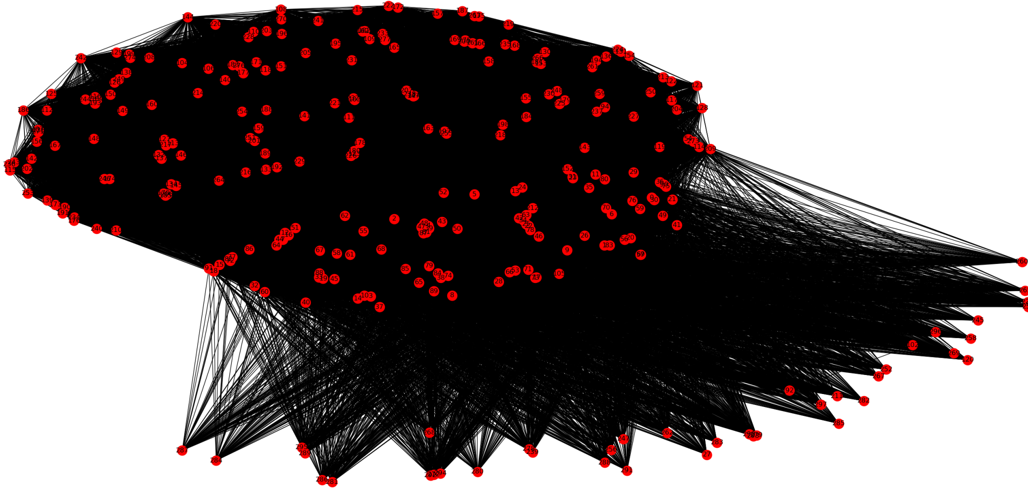


Figure 3.5: Coveriance Cloud for the full Yelp dataset.

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 [84], or the Yelp open dataset [93] 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

Figure 3.6: KNN-PCA cloud with K=6 on the Yelp dataset.

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.

Example 9 *Figures 3.1 and 3.6 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.

These graphs may not initially be intuitive to interpret, but they provide abstract bearings that map directly to phenomenon's that exist in the data. In lieu of a formal user study, we postulate that the information gained from each algorithm independently will benefit the exploration process that is agnostic to labeled information. This is in contrast to other tools [75], that largely utilize naming cues and attribute nesting to meet user needs.

3.3.2 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 3.1, 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 3.7. Users can select one or more attributes (or groupings) in the schema summary pane,

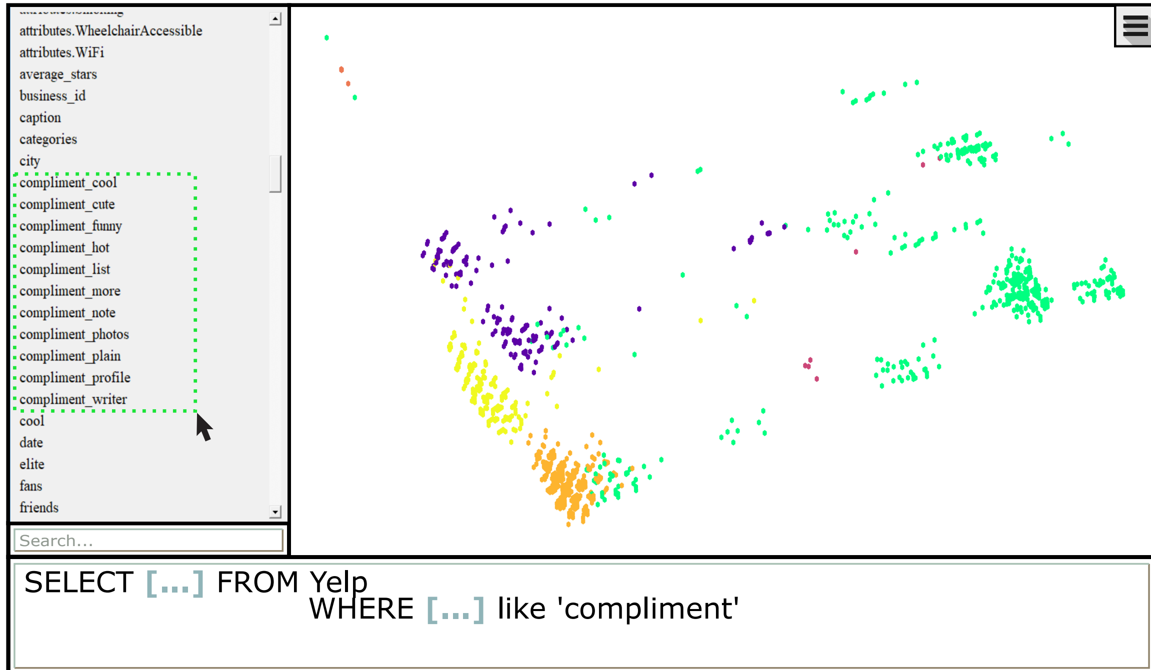


Figure 3.7: User-selected attributes shown in green.

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.

3.3.3 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.

3.4 Alternative Extractors

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 [39]. 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 [55, 58]. Key challenges in this space involve simply extracting schemas from large, multi-terabyte collections of JSON data [12], as well as managing ambiguity in the set of possible factorizations of a schema [10, 78]. The approach taken by Baazizi et. al. [12] in particular adopts a type unification model similar to ours, but lacks the conjunctive operator of our type-system. For non-hierarchical data, interactive tools like Wrangler [51] 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 [55] to quickly identify records that make use of sparse paths in the schema. Another approach is to normalize schema elements [30]. Although the resulting schema may not always be interpretable, this approach can result in substantial space savings.

Information Retrieval. From a more general perspective, the schema extraction problem which aims at making large datasets tractable for interactive exploration, is an instance of *categorization* problem that has been repeatedly studied in the literature. More precisely, attributes (metadata) of the datasets can be grouped into a hierarchy of "facets" (i.e., categories) [75] where the child-level facets are conditioned on the presence of the parent one. In our approach, we adopt the hierachical data visualization and focus more on the algorithmic essence of the problem: How to (1) balance between the preciseness and conciseness of the visualization and (2) respond to users' data exploration requests in a scalable way.

3.5 Recursive Schemas

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.

Chapter 4

Putting the Schema back in Schema-on-Read

4.1 JSON Inconsistencies

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. We propose SchemaDrill, a JSON schema discovery algorithm with heuristics that mitigate common forms of ambiguity. Although SchemaDrill is slightly slower than state of the art schema extractors, we

show that it produces significantly more precise schemas.

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 [7, 10, 12, 58] 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 10 *Consider the two JSON records in Figure 4.1. Production schema discovery systems (e.g., Spark's [7]) 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*

```
{"ts":7,"event":"login","user":{"geo":[43.4,-7.2],
                                "name":"jbond"}}
{"ts":8,"event":"serve","files":["q.jpg","m.jpg"]}
```
Figure 4.1: Example JSON Data

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":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 [12]: (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 response, we develop a new JSON schema discovery system called SchemaDrill. In contrast to existing techniques (e.g., [7, 12]) that resolve ambiguity through *data-independent* heuristics, SchemaDrill'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, we make 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 SchemaDrill, a general framework for heuristically resolving this ambiguity, (iii) We show the feasibility of SchemaDrill by proposing specific heuristics for detecting and resolving these forms of ambiguity, (iv) We show experimentally that SchemaDrill with these heuristics creates schemas with significantly higher precision than state-of-the-art schema discovery, with negligible change in recall.

4.2 Extractor 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 *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. [12] 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

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

Figure 4.2: **JSON's Typesystem**

summarized in Figure 4.2, 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 11 *The JSON object with* `ts` *7 in Figure 4.1 has type:*

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

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 $\mathcal{S}$.

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

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

4.2.1 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. [12]. In other words we define:

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

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

Example 12 *Applied to the two records in Figure 4.1, naive discovery simply returns a set of the two distinct schemas.*

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

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 13 *The field `files` in Figure 4.1 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 $\texttt{keys}_{\exists}(\mathcal{R})$).

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

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 $\mathcal{S}_i \triangleq \texttt{merge}(\{\ \tau.i \mid \tau \in \mathcal{R}\ \})$:

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

Standard Discovery. The classical approach to schema discovery, used in production systems like Spark's JSON data source [7] or Oracle's JSON Data Guide [58], is modeled by Baazizi et. al.'s $\mathcal{K}$-reduction [10], defined formally as follows:

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

This approach uses naive merge for primitive types and recursively merges arrays and

objects as collections or tuples, respectively.

4.3 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 4.7.

4.3.1 Arrays as Tuple-Like Structures

Example 14 *Consider the `user.geo` field of Figure 4.1. 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 [85] and Yelp [90] follow the GeoJSON standard [22] and use 2-element arrays for coordinates. Similarly, arrays sometimes encode tuples in settings where JSON data is naively generated from CSV files [64]. In each case, treating all arrays as collections (e.g., $[\,\mathbb{R}\,]^*$ instead of $[\,\mathbb{R}, \mathbb{R}\,]$) results in unnecessarily permissive schemas.

4.3.2 Objects as Collections

Example 15 *Consider the pharmaceutical dataset [73] 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 [60]:

```
{"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, as well reducing the schema's recall.

4.3.3 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 [83], 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.

4.4 Extraction Overview

We now introduce SchemaDrill, 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 4.2 and its semantics follow naturally. We abuse notatation and use expressions in this grammar interchangeably with schemas.

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

As an example, Algorithm 1 implements $\mathcal{K}$-reduction as presented in Section 4.2. 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 [12]:

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

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

4.4.1 Naive Implementation

SchemaDrill'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

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($\{\ \tau.k \mid k \in \texttt{keys}(\tau), \tau \in \mathcal{R}\ \}$))

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

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

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, SchemaDrill partitions the bag into individual entities and infers a schema for each individually (Algorithm 3 and its array analog).

4.4.2 SchemaDrill

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

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

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

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

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. SCHEMADRILL instead decouples these heuristics into separate computation stages, each taking one pass over the data as illustrated in Figure 4.3. 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

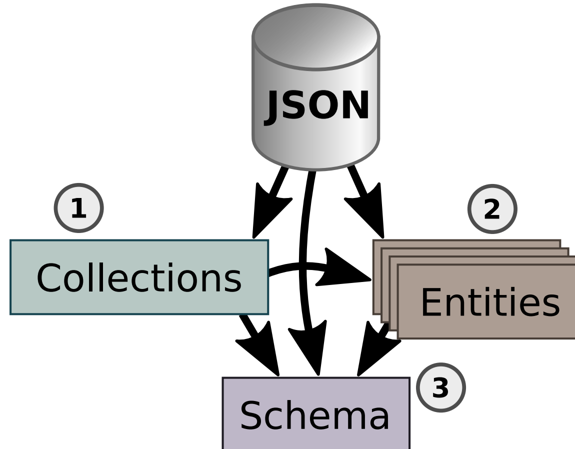


Figure 4.3: **Stages of Extraction in SchemaDrill**

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 SchemaDrill on only a small sample of training data. As we show in the Section 4.7, 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, SchemaDrill 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.

4.4.3 Helper Heuristics

SchemaDrill 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 4.5 and 4.6.

Detecting Nested Collections. Following prior work in schema extraction (e.g., [10, 12]), SchemaDrill 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 SchemaDrill 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, SchemaDrill 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.

4.5 Detecting Collections

JSON objects and arrays can both encode nested collections or nested tuple-like structures. This section describes a default heuristic for SchemaDrill that distinguishes between these cases. We initially target object-kinded types as inputs,

before generalizing to arrays below.

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

The goal is to mark the objects as (a) collection-like (i.e., **ObjectCollection**), or (b) tuple-like (i.e., **ObjectTuple**). SchemaDrill'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.

4.5.1 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. 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:

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

For each key, SchemaDrill 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

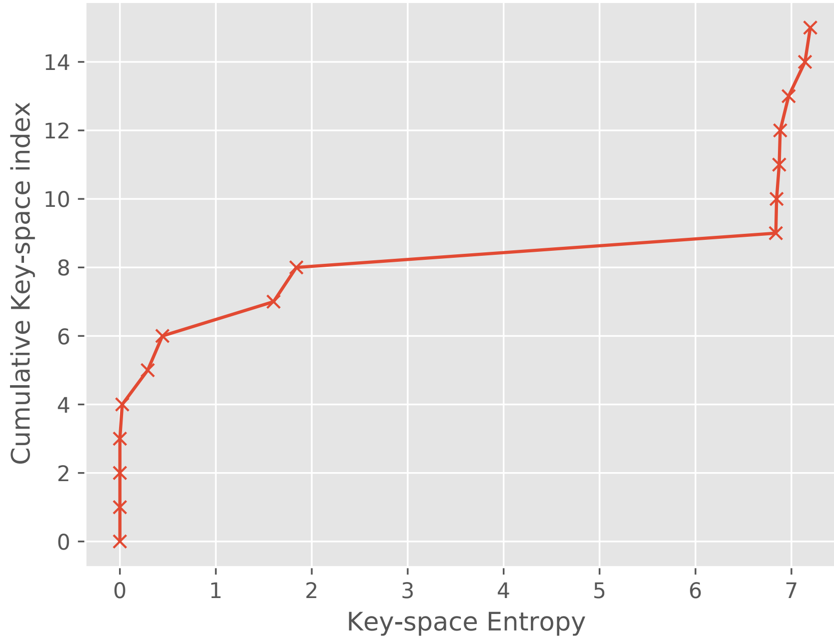


Figure 4.4: **Yelp nested collection key-space entropy**

numerical value that captures variation in keys across the input objects, with higher values marking the objects as more collection-like.

Example 16 *Consider the two records of Figure 4.1. 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* $\mathcal{E}_{\mathcal{K}} = 0.70$ *(*$= 2 \cdot 0 + 2 \cdot 0.35$*).*

4.5.2 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, SchemaDrill adopts a constraint based on the following type similarity rule:

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

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.

4.5.3 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

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: **end for**
7: **for all** (`kind` : `count`) $\in$ `KindCount` **do**
8: $\mathcal{E}_\mathcal{T}$ += $\frac{\texttt{count}}{|\texttt{keys}(\tau)|} \log \left(\frac{\texttt{count}}{|\texttt{keys}(\tau)|} \right)$
9: **end for**
10: **end for**
11: **if** $\mathcal{E}_\mathcal{T} > 0$ **then return** `Tuple`
12: **end if**
13: **for all** (`key` : `count`) $\in$ `KeyCount` **do**
14: $\mathcal{E}_\mathcal{K}$ += $\frac{\texttt{count}}{\texttt{RecordCount}} \log \left(\frac{\texttt{count}}{\texttt{RecordCount}} \right)$
15: **end for**
16: **if** $\mathcal{E}_\mathcal{K} \leq 1$ **then return** `Tuple` ▷ Threshold value
17: **else return** `Collection`
18: **end if**

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 4.4 shows the distribution of Key-Space entropy in the Yelp dataset introduced in Section 4.7: 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.* Our experiments arbitrarily use a threshold of 1.

4.5.4 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 $[\,\mathbb{R}, \mathbb{R}\,]$), while other applications encode rows of a CSV file as fixed-width arrays [64]. 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.

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

4.6 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.

4.6.1 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.

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

Example 17 *Both of these schemas admit every record in Figure 4.1.*

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

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

$\mathcal{S}_1$ encodes two distinct entities (one for each event type), while $\mathcal{S}_2$ uses a single entity with optional fields.

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

A natural solution is clustering similar records together through a classical algorithm like k-means. However, classical clustering presents two challenges. First, K may not be known ahead of time. Even if K is known, a more subtle issue arises when the entities of $\mathcal{S}_\mathcal{G}$ 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 18 *For example, the Yelp* `photos` *table has 4 mandatory fields, while the* `business` *table has 20, most of which are optional. Both* `photos` *and* `business` *share exactly one mandatory field:* `business_id`*. While a business record missing*

17 optional fields shares only 1 field in common with a photos record. Between them there are 6 distinct fields (1 shared, 3 photo-only, 2 business-only). Thus, the Jaccard index considers this 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 [74], a problem where we need to simultaneously group records by feature co-occurrence, while also grouping features by co-appearance in records.

4.6.2 Bimax

SchemaDrill adopts a simple, yet surprisingly robust and commonly used bi-clustering technique called Bimax [72]. 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.

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: $\quad \mathcal{K}_i \leftarrow \{ \mathcal{K}[j] \mid i \leq j \leq |\mathcal{K}| \}$
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 $\mathcal{K}_{sub} < \mathcal{K}_{overlap} < \mathcal{K}_{disjoint}$
▷ Preserve existing order within each set
10: $\quad i \leftarrow i + |\mathcal{K}_{sub}|$
11: **end while**

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 $\mathcal{K}_{naive}$
6: **end while**

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 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 19 *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.*

4.6.3 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. SchemaDrill adopts a simple greedy heuristic summarized in Algorithm 8.

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

Example 20 *Consider 4 entities discovered by* ***Bimax-Naive*** *over keys $A, B, C, D, E,$*

Algorithm 8 GreedyMerge

In: $\mathcal{K}_{naive}$: The output of **Bimax-Naive**.
Out: $\mathcal{K}_{merge}$: A list of merged key-set clusters.

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

with maximal elements:

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

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

As an alternative view of **GreedyMerge**, consider an undirected graph with one node for each entity emitted by **Bimax-Naive** and one edge for every pair of nodes that share a field. Intuitively, optional fields manifest in this graph as regions of

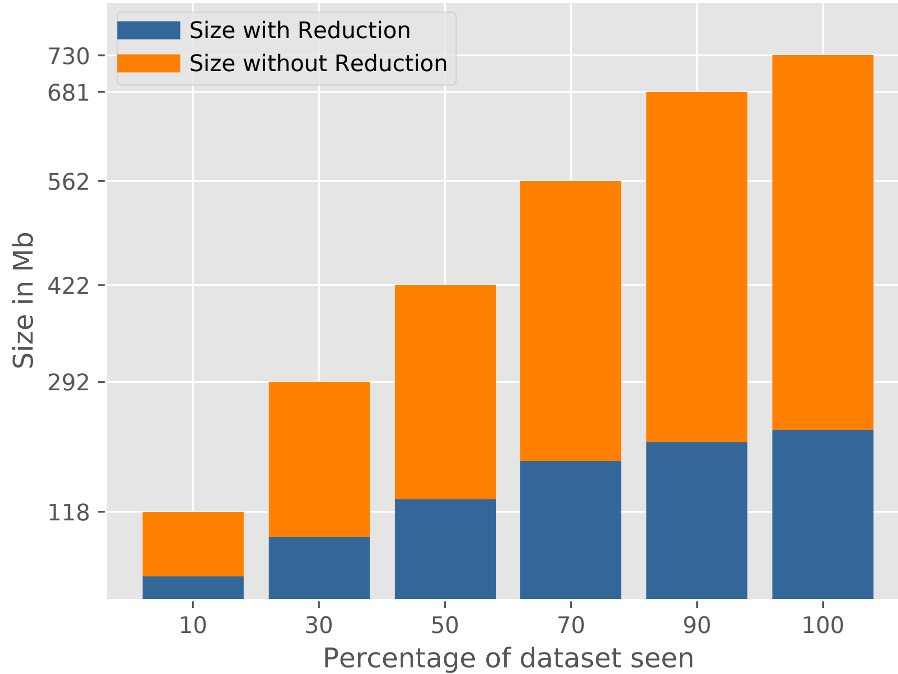


Figure 4.5: **Memory comparison: Yelp**

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.

4.6.4 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. SchemaDrill 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 4.5 illustrates the memory savings of removing nested collection features in the Yelp Dataset (see Section 4.7), while in the the Pharmaceutical dataset (also see Section 4.7) 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.

4.7 Schema Generation

We now evaluate SchemaDrill against the $\mathcal{K}$-reduction schema discovery algorithm proposed by Baazizi et. al. [10,12]. 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.

Dataset		$\mathcal{K}$-reduce mean	$\mathcal{K}$-reduce std	$\mathcal{K}$-reduce max	**Bimax-Merge** mean	**Bimax-Merge** std	**Bimax-Merge** max	Bimax-Naive mean	Bimax-Naive std	Bimax-Naive max	$\mathcal{L}$-reduce mean	$\mathcal{L}$-reduce std	$\mathcal{L}$-reduce 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 4.1: **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)**

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) SchemaDrill schema discovery process. Concretely, our experiments validate the following claims:

(i) SchemaDrill 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 4.6 is critical for creating compact schemas, and

(v) The overhead of the additional steps required by SchemaDrill 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 [38] collected over a period of 330 days, (ii) a 240,000 record open dataset of per-doctor **Pharmaceutical** prescription statistics [73], (iii) 800,000 **Twitter** tweets, (iv) 150,000 events taken from a Matrix **Synapse** server [60], (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 [90]. 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 SchemaDrill (Algorithm 4) with the naive adaptation of Bimax (Algorithm 7), (iii) **Bimax-Merge**: A single pass of SchemaDrill 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" [94], where each entity attribute is represented as an integer key for reference

[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.

linking. Finally, the Synapse dataset is the `events` table from a multi-year deployment of the Matrix Synapse open source chat server [60]. 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 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 [10], 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.

4.7.1 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 4.1 shows the fraction of the records in the testing set accepted by the generated schema. The table omits datasets where SchemaDrill 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 SchemaDrill 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: SchemaDrill 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, SchemaDrill 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, SchemaDrill correctly identifies this structure as a two-level nested collection, allowing it to generalize to servers and keys not present in the sample.

4.7.2 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 4.2; 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. SchemaDrill's tighter schemas are largely due to partitioning entities. In the Yelp and GitHub datsets especially, mixed

		$\mathcal{K}$-reduce		Bimax-Merge		Bimax-Naive		$\mathcal{L}$-reduce	
Dataset		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 4.2: **Schema Entropy: The number (log 2) of types accepted by the generated schema († ran out of resources).**

	Yelp					
Schema	Bus	Ckn	Pho	Rev	Tip	Usr
$\mathcal{K}$-reduce	197	132	297	292	296	277
Bimax-Merge	0	0	0	0	0	0
k-means	0	0	106	109	107	124

	Github													
Schema	Com	Cre	Del	For	Gol	IsC	Iss	Mem	Pub	Pul	PRR	Psh	Rel	Wat
$\mathcal{K}$-reduce	798	825	827	732	823	675	709	810	830	383	370	815	760	829
Bimax-Merge	0	0	2	0	0	0	34	0	5	0	0	0	0	6
k-means	127	126	124	115	128	4	0	139	121	0	1	136	143	120

Table 4.3: **Minimum symmetric difference from ground-truth schema (lower is better)**

entity types pose a challenge for $\mathcal{K}$-reduce, which emits only +a single entity with many optional fields. Conversely, SchemaDrill 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.

4.7.3 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

	$\mathcal{L}$-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 4.4: **Entity predictions with 90% training data († $\mathcal{L}$-reduce and Bimax-Naive ran out of resources on Wikidata)**

(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 4.3 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 4.4 illustrates the effectiveness of the **BiMax-Merge** optimization of the **BiMax-Naive** algorithm (Algorithms 7 and 8 in Section 4.6, 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, SchemaDrill 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.

4.7.4 Runtime

Finally we evaluate claim (v) by comparing the runtime of SchemaDrill against that of $\mathcal{K}$-reduce. We omit the Pharmaceutical dataset runtime where the official binary implementation times out, as previously noted. In general, SchemaDrill needs to do more work to create a more precise schema, so we do not expect it to

	1%		10%		50%		90%	
Dataset	*$\mathcal{K}$-reduce*	*Bimax-Merge*	*$\mathcal{K}$-reduce*	*Bimax-Merge*	*$\mathcal{K}$-reduce*	*Bimax-Merge*	*$\mathcal{K}$-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 4.5: **Runtime (milliseconds) by discovery algorithm and training set size. (†: $\mathcal{K}$-reduce times out on the Pharma dataset)**

outperform $\mathcal{K}$-reduce; We aim here simply to assess the added overhead. Table 4.5 shows schema discovery performance, varying the proportion of each dataset used in order to generate the schema to show scaling behavior. Performance scales linearly for both extractors. SCHEMADRILL 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 SCHEMADRILL 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 SCHEMADRILL picks-out complex nested structures.

4.7.5 Results

Table 4.2 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 4.1 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 4.4 makes clear the importance of the Bimax merge heuristic for creating compact, descriptive schemas. SchemaDrill produces goldilocks schemas with the benefit of both extractor implementations proposed in related work [10, 12]. Lastly, although SchemaDrill 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 SchemaDrill over $\mathcal{K}$-reduce in general, while by contrast, SchemaDrill produces far fewer false negatives.

4.8 Alternate Extraction Techniques and XML

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. [10, 12]. They propose a grammar for concisely describing sets [12] and bags [10] of JSON types (on which we base our grammar in Section 4.2), 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 [37]. Unfortunately, requiring the merge operation to be commutative and associative limits it to local-decision making: Schema properties like the number of entities (Section 4.6) or whether an object encodes a tuple or collection (Section 4.5) 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. SchemaDrill 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 [15–17, 30, 39, 44, 62, 67, 79, 81, 89], graph [5], and object-exchange model (OEM) [39] data. XML schema discovery in particular [15–17, 44, 62] 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 con-

cept 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 [16], a problem related to entity discovery in SchemaDrill. Their approach relies on the node's ancestors and predecessors to disambiguate entity types, inferring a selector for each entity based on these factors, rather than based on the entity's attributes as in **Bimax-Merge**.

A range of machine-learning-based techniques have been proposed for discovering such linkages [67,79,81,89]. However, as we discuss in Section 4.6, such approaches are vulnerable to skew in entity size, and the inherent ambiguity of the entity discovery problem. SchemaDrill adopts a more robust approach based subset relationships and field overlap. We note one approach [5] 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 [82]: 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 [30, 61, 95] between nodes in graph [61], XML [95], or JSON [30] 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

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

differentiate between tuple- and collection-like reference/nesting structures. Like the extractor of Baazizi et. al., [30], 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 [9, 11]. 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 [95] 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 SchemaDrill.

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" [76] to make XPath queries resilient to small schema changes, while the Prism Workbench [27] encodes prior versions of a relational schema as views. GraphQL [86] 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.

Chapter 5

Tabular Layout

5.1 Partitioning

In many cases datasets contain natural boundaries by which data may be compartmentalized and later leveraged in queries. Time, location, identifiers, are natural candidates for such an optimization. Example 21 shows a common date bound query selecting all orders placed after a given date, selecting orders for the month of January. For this example we partitioned our data by month, Figure 5.1 illustrates the primary benefit of partitioning, enabling all other month partitions to be excluded. Additionally, indexes within each partition become smaller and more efficient, improving query performance and enabling our data to remain cached longer. Devising an upfront partitioning scheme is critical to scaling query performance, and is a vital yet often overlooked tool due to upfront intricacies.

Example 21 *Date bound query, candidate for partition optimization.*

```
01 |  SELECT item_no, item_desc, order_placed
```

Orders		
item_no	item_desc	order_placed

Orders (Partitioned on order_placed)		
item_no	item_desc	order_placed
		2021-01-01
		2021-01-31
		2021-02-01
		2021-02-14
		2021-02-07
		2021-04-03
		2021-04-10
		2021-04-02

Figure 5.1: Data layout comparing Orders table with and without partitioning. Green indicates data scanned in query example 21.

```
02 | FROM   Orders
03 | WHERE  order_placed >= '2021-01-01'
04 |   AND  order_placed <  '2021-02-01'
```

5.2 RDBMS JSON Integration

Most major RDBMS support JSON as a first class data type, where each cell is a JSON record [71]. This functionality provides a convenient mechanism for loading, processing, and preparing JSON for storage. However, PostgreSQL for example requires special built functions to interact with JSON data, which themselves are slow, clunky, and dependent on specific designated formats. To make matters worse, indexing attributes proves to be a complex and nuanced process. Lastly, JSON data, even as a native data type, is treated as a black box. Subsequent down stream operators are unoptimized, with the potential for joins against JSON data to be sub-optimal.

Fortunately, databases are excellent at optimizing tabular data, so by ditching the JSON type and directly shredding our JSON data we can: (1) improve query performance using less specialized indexes, (2) avoid JSON specific functions, and (3) enable the optimizer to rewrite queries, greatly improving query performance.

5.2.1 Binary JSON

BSON is a binary encoding for JSON records [19], and is supported in PostgreSQL, MongoDB, and most RDBMS. JSON stored as a native string requires every query to be re-parsed, with BSON JSON is parsed once then efficiently traversed each time after. While this format is preferable to the raw string alternative, it remains a row level optimization, unable to gather table level statistics. Lacking table level statistics makes enforcing types, optionality, and primary key constraints far more complicated. Additional binary encodings have been proposed and used in practice [54], but ultimately are still opaque compared to tabular data and still require specialized functions for access.

5.2.2 JSON Tables

Through the use of specialized functions, JSON can be projected into in-memory tables. While this abstraction provides a powerful mechanism to project JSON data into a more usable form. It still lacks table level statistics critical for efficient join and read optimizations. Additionally, fetching deeply nested attributes requires multiple lookups that are difficult to order or predict, even when using BSON. Instead of using special JSON or BSON data types, we propose to first flatten our JSON records. Storing data in a flat tabular format like any other table. This allows us to take full

advantage of decades worth of database optimizations regardless of JSON supported features. As previously discussed in Chapter 4, JSON records have many nuances, like containing thousands of unique attributes, making direct import often impossible. Our goal is to use the various techniques proposed in chapter 4 to regularize our data, making direct native import for JSON practical and efficient.

5.3 Sparse Columns

As discussed in Section 4.3.2, frequently key-value stores will be embedded within JSON records, what we refer to as *collections*. For example, in our pharmacy dataset the proportion of nulls is over 98%, spread across thousands of keys, as depicted in fig. 5.2. While certain cues may be taken from this distribution, in general this key-value pattern would create hundreds to thousands of additional columns using a direct import strategy. As in chapter 4, we will use our notion of key-space entropy to detect these collections, and devise a more appropriate loading strategy.

5.3.1 Null Space

Calculating the impact of nulls on physical storage space is not straight forward. In practice various database implementations apply on-disk optimizations such as eliminating trailing nulls, storing nulls in bitmaps, etc. However, worst case each null can add an additional byte of data, and considering a direct JSON to table mapping can create thousands of sparse columns, this create a significant performance issue. By normalizing collections detected by key-space entropy, we see in fig. 5.3 that we dramatically decrease the number of columns and nulls.

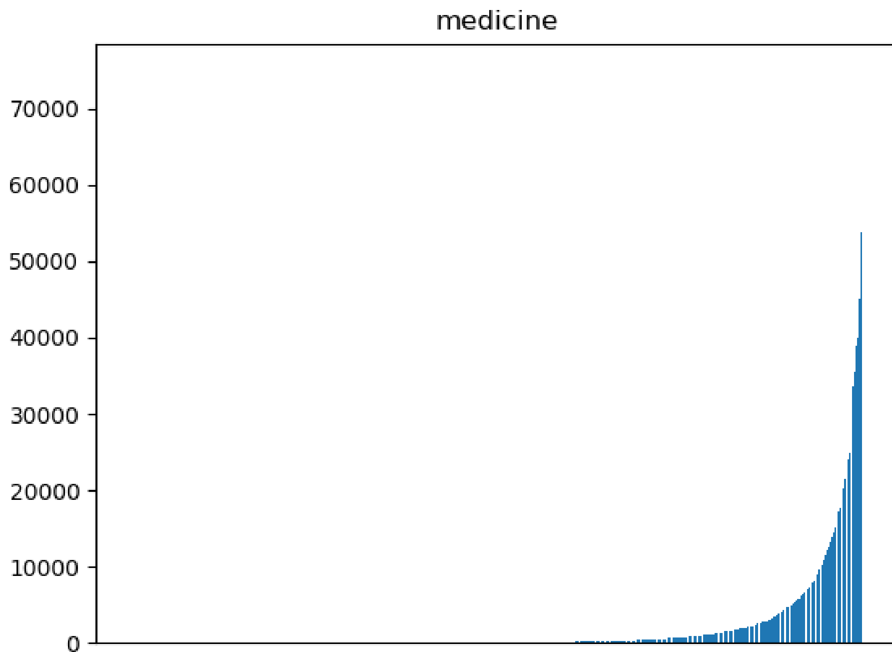


Figure 5.2: Distribution of prescription key occurrences.

5.3.2 Pivot Tables

After reflecting on how data within collections is likely to be accessed, we made the natural observation that a pivot table would be a great option for such same typed key value attributes. This allows us to normalize our proportionally large collections away from our core attributes, improving such queries. Additionally, pivot tables are a frequently used concept beyond the area of databases, enabling easier use by even beginners. Pivoting also allows for the potential use of more efficient specialized compression algorithms. To create our pivot table we simply add a foreign key between our core schema and the pivot, and create the pivot table such that it has an id column, key column, and value column. Further, we can implement an

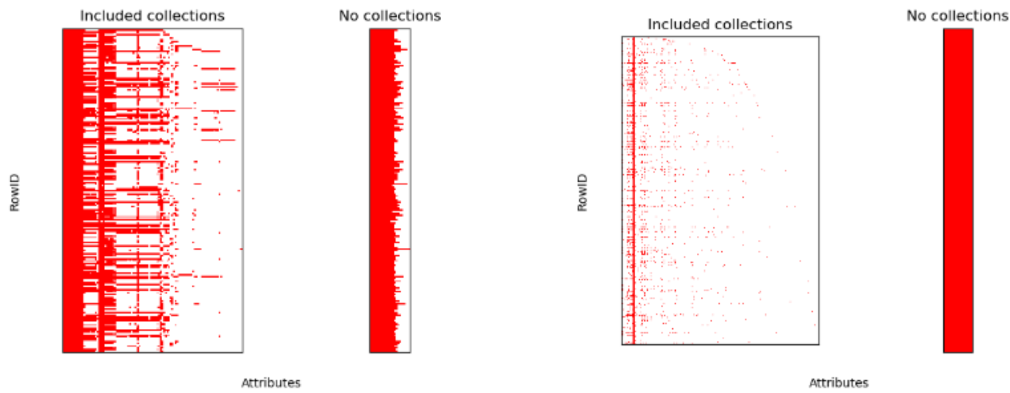


Figure 5.3: Yelp attributes (left) and Pharma attributes (right) tabulated with and without collections. White space is null.

Entity–attribute–value model to save space given certain key distributions. Finally, this normalized form can be used by the optimizer to create more efficient joins in complex queries.

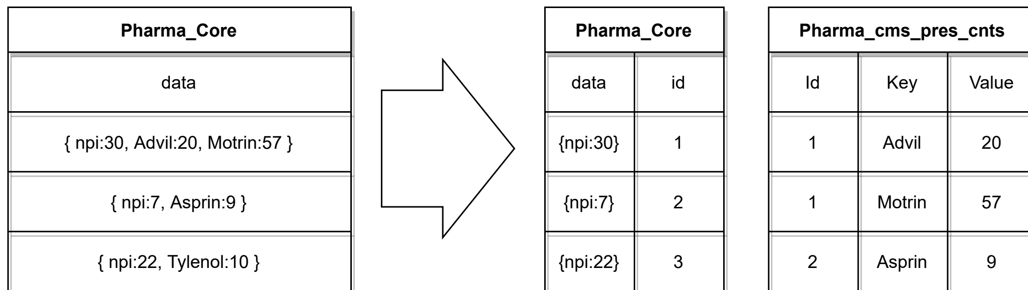


Figure 5.4: Example pivot table transformation.

5.3.3 Performance and Usability

Currently implemented, JSON functions are burdensome compared to the expressive power of SQL. Example 22 is a sample record from our Pharma [73] dataset. Of note is the outer attribute NPI mapping to a unique doctor identifier, and the

collection of prescriptions as an embedded key-value store. Using our collection detection method outlined in chapter 4, we detect likely embedded key-value pairs, choose an optimal pivoted layout, and rewrite incoming queries. Many popular RDBMS support a JSON to in-memory table function, often named JSON_Table. This is a powerful function enabling complex projection, and is our target comparison operator to rewrite. Simply, by keeping track of which JSON paths we have relationally shredded into a pivot, we check whether any projected attribute appears in the query. As depicted in fig. 5.5 and fig. 5.4, we pre-pivoted attribute *cms_prescription_counts* into three columns, **foreign_key**, **key**, and **value**. When rewriting, we preform a natural join between our main and foreign table, then add a selection filter on the **key** column with the predicate being our target attribute value. From *cms_prescription_counts.* our predicate is , so we select all keys. If of example we were projecting *cms_prescription_counts.acetaminophen*, we would add clause **WHERE Pharma_cms_prescription_countsk̇ey = ácetaminophen´**. This simple rewrite allows us to covertly boost query performance, unbeknownst to the user.

Example 22 *Sample JSON record from our Pharma dataset.*

```
{"cms_prescription_counts":
   {"DOXAZOSIN MESYLATE": 26,
    "MIDODRINE HCL": 12, ... },
 "npi":1 }
```

Common queries a user may issue are, ”how many prescriptions did each provider write”, ”how my prescriptions of certain drugs”, etc. Example 23 reflects this sample query using PostgreSQL’s JSON pivot function. This query requires a lateral join, multiple casts, and does not handle recursive embeddings. Not only is this process

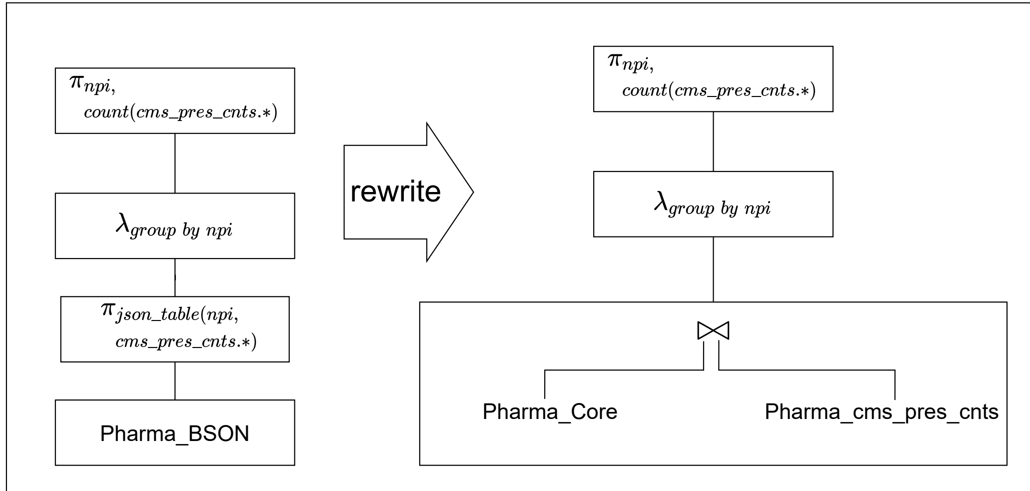


Figure 5.5: Example pivot query rewrite.

tedious, verbose, and error prone, but has a significant performance draw back. Section 5.3.3 shows the performance of various aggregation and filter queries over our pharmacy dataset using (1) native JSON format, (2) BSON format, and (3) a pre-pivoted format detected by our entropy metric. Unsurprisingly, we find by detecting this pattern and pre-pivoting and storing our data relationally. We gain a 5x to 10x performance improvement over BSON and JSON respectfully. Additionally, by skipping otherwise required casts, and gaining other inherent performance benefits to tabular storage. We see even queries targeting only specific drugs also gain a substantial performance benefit. Thus, in the best case scenario we successfully predict the way in which users will query their data. Gaining a substantial performance benefit, and in the worst case we only double our query performance.

Example 23 *PostgreSQL JSON pivot query.*

```
01 |  select (data->'npi')::text as npi, sum(value::text::int)
02 |  from pharma_json, json_each(pharma_json.data->'
        cms_prescription_counts')
```

Format	Agg (mean)	Std	Select (mean)	Std
JSON	21545	491.7	1934	218.2
BSON	9273	31.4	1020	2.9
Pivot	1636	6.4	423	1.0

Table 5.1: Pharma dataset group by aggregate and selection query performance (ms).

```
03 |   group by npi;
```

5.4 Predictive Partitioning

RDBMS partitioning is traditionally by column value, where every partition enforces the same schema. For example, Example 21 utilizes the value in the order_placed column for partition pruning. While this approach can generate huge performance boosts, it caters to data already in relational format, and is best left as an advanced optimization. However, instead of partitioning by column value, we can partition by attribute occurrence. This allows us to give the illusion of one contiguous data block, while achieving performance as if it was manually segmented. We can then store in metadata whether columns in particular pages or files are present, allowing us to skip portions of data where no column exists in the query. Pruning based on metadata is deployed by various map-reduce systems, cloud based databases, and by various research projects. Extending this optimization to heterogeneous data allows us to create a dense representation of otherwise sparse data, taking advantage of signals relayed through JSON attribute occurrence.

5.4.1 Background

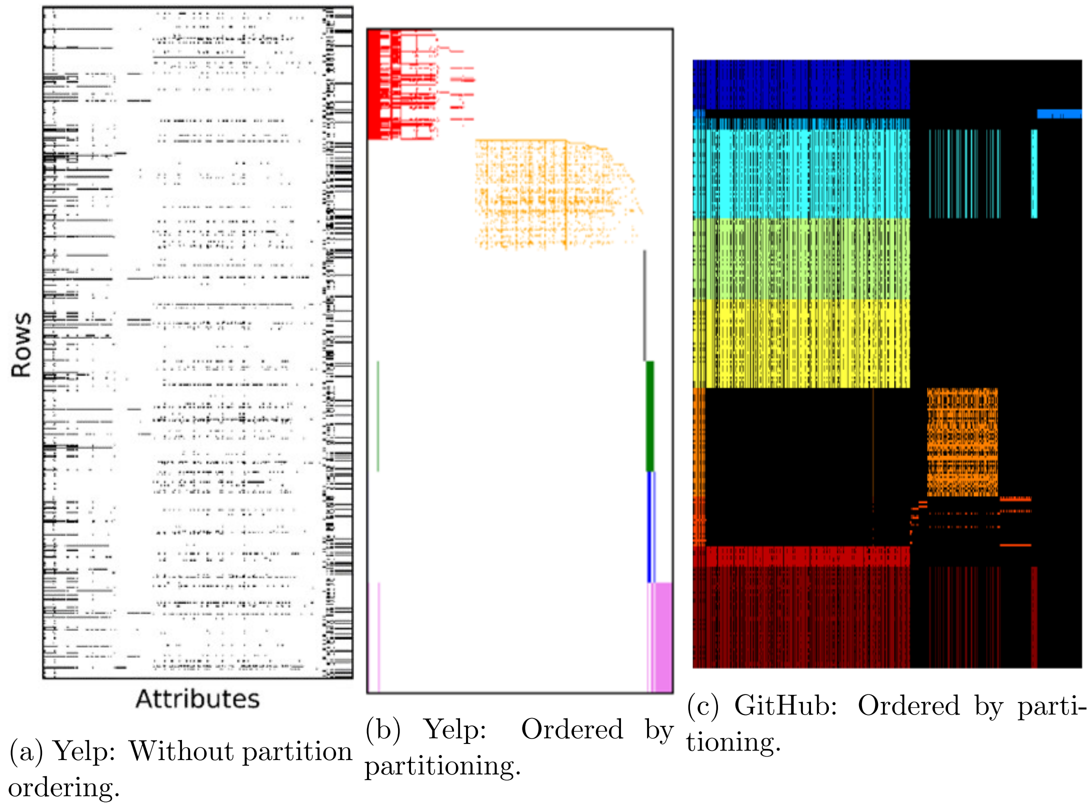
Schema generation utilizing values is too computationally and memory intensive for most large real world datasets. These systems instead rely on sampling for larger inputs, which has blatant issues when used for schema generation. Luckily, JSON attribute occurrence provides useful insight into data generation and use cases, while requiring a fraction of the cost. This allows us and similar systems to scale to large datasets through the use of distributed systems [9–12]. Additionally, we can use our schema generation process outline in Chapter 4 as a preliminary starting point to narrow down more expensive techniques. Using the attributes present in any schema as a filter, we can omit partitions from our scans. Similar optimizations have been used to great success in systems like NoDB [3] and Apache Spark. Although map-reduce systems may gain other benefits from a priori partition information.

Another distinction between JSON and RDBMS data is the role it fills. Due to a number of reasons relational data is generally dense and unchanging. It is more likely that a user will change their data requirements than change their table layout. In contrast, frequently JSON will be a mix of multiple partially or barely overlapping schemas. In fact a number of products are built around this exact behavior, storing outputs from various JSON sources in a single location [77]. At query time this has profound impact on the amount of data required for scanning. While tools like Apache Spark, Hadoop, and Impala are frequently the go-to JSON querying tool. Full RDBMS support is a clear user demand, requiring complex multi-system integrations to create a less than ideal product, usually with limited support. In addition to user demand, storing JSON data natively in an RDBMS enables the full range of ACID transactions, largely unsupported in native JSON formats.

5.4.2 Partition Algorithms

Partitioning aims to take advantage of inherent segregations within a dataset or table, allowing portions of computation to be skipped. Examining our datasets for which we have ground-truth, Yelp and GitHub, we visualize attribute occurrence in fig. 5.6. Figure 5.6a is an random ordering of our Yelp dataset, while Figure 5.6b sorts by our partition predictions. Visually, we see these groups are largely disparate, and likely a query would only pertain to a single grouping. Further, looking at Figure 5.6c, we also see large differences between partitions, however there's apparent significant overlap. This overlap in attribute naming can be intentional, sharing foreign keys, primary keys, corresponding to the same underlying component, or can be accidental, think how common the attribute "name", "address", etc. appear in datasets.

In Chapter 3 we explore a number human interpretable techniques, principal component analysis, various clustering techniques, frequent pattern trees, and more. From this analysis, we concluded classification using existing techniques appears to require a human-in-the-loop for both tuning and interpretation. Chapter 4 we introduce a new clustering algorithm, Bimax-Merge, which aims to take advantage of specific JSON characteristics which other algorithms weight less. Recent works have also explored the use of a decision tree algorithms, such as FPGrowth, for JSON clustering [2, 33, 43]. FPGrowth [43] requires users to set threshold parameters to generate frequent sets, and association rules. While the use of cost functions [33] dramatically improves the end user's interface, as we find in 3, parameter tuning has a prohibitively expensive human cost. Additionally, FPGrowth produces a varying number of frequent sets based on the threshold selection. Figure 5.7 depicts how this parameter impacts frequent set predictions. In testing we found tuning these parameters to



(a) Yelp: Without partition ordering.

(b) Yelp: Ordered by partitioning.

(c) GitHub: Ordered by partitioning.

Figure 5.6: Visualization of Yelp dataset partitioning.

generate ground truth frequent sets to be impractical. This is due to each partition effectively having a unique profile based on multiplicity, and either had no attribute set generated or an extremely limited subset. So much so that even mandatory attributes were excluded due to varying row multiplicities between partitions. Using this approach to detect potential key candidates has promising potential, however for the purpose of generating tabular partitions we find a similar merge algorithm such as the one outline in Chapter 4 would be necessary.

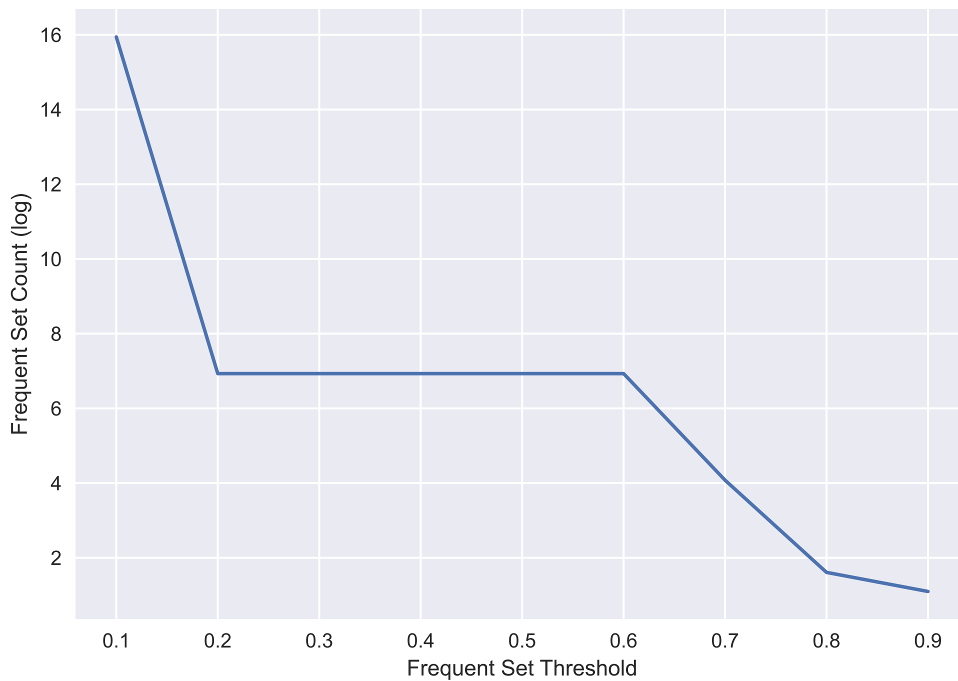


Figure 5.7: Yelp frequent set counts by threshold (log scale).

5.4.3 Apache Parquet

Apache Parquet [70] is a popular column oriented file format, frequently used for JSON due to its hierarchical encoding. Column oriented formats offer a significant performance benefit over row-wise storage when querying a small percentage of columns. While column oriented formats are not perfect for every use case and are not directly in our primary objective path. They are such a staple in the JSON ecosystem, we would be amiss to not investigate potential benefits of our partitioning scheme. Parquet is no stranger to utilizing partitioning as a main performance strategy. Declaring a partition key enables the logical and physical storing of simi-

lar records, and can be used to optimize parallel computation while avoiding invalid partitions. Additionally for our preliminary testing purposes, we can easily store our JSON data without the headache of creating relational DDL, while still not incurring the desegregation of performance due to sub-optimally encoded nulls. This is in part due to the efficient run length encoding, meta-data blocks, and data chunk format deployed by Parquet. Our queries were executed using Apache Spark SQL version 2.3 over our Yelp and GitHub ground truth datasets. We issued a number of aggregate and filter queries using three different strategies (1) no partitioning, (2) Spark's partition by key optimization, and (3) manual partition targeting and storage. From Figure 5.8 we can see parquet inherently does a good job invalidating entire null chunks by default, however some performance benefit is still gained by avoiding unnecessary reads and checks. Additionally, we see manual partitioning which would be the theoretic best performance gains even greater benefit. This is not entirely surprising considering a number of optimizer checks are skipped, and meta-data blocks can be read efficiently. While this is not a deep dive into partitioning performance benefits by partitioning in Apache Spark and Parquet, it is evidence that partitioning even under otherwise thought to be optimal querying format can still be improved.

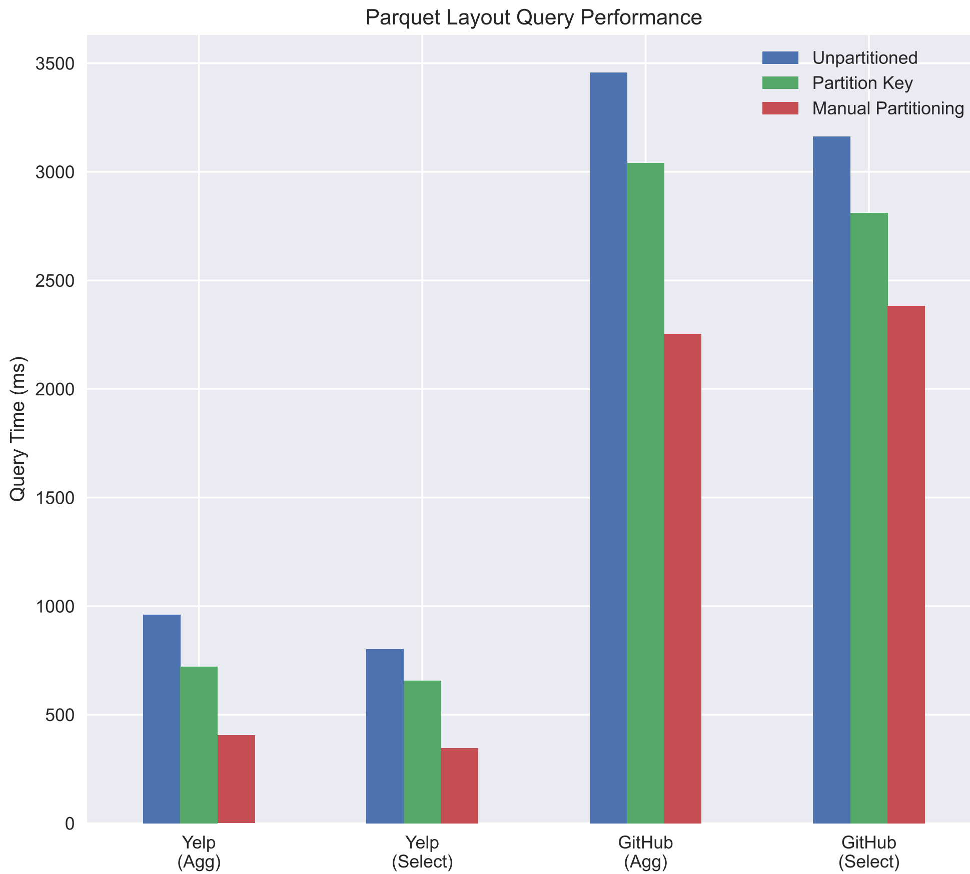


Figure 5.8: Performance of various queries using Apache Spark with various Parquet partitioning strategies.

5.4.4 Partition Query Rewriting

JSON datasets frequently contain thousands of unique attribute paths. This poses a significant initial loading hurtle, considering most RDBMS have a hard column capacity, often in the hundreds. We find this case often occurs in two scenarios, when datasets contain embedded key-value collections, and/or multiple conceptually differ-

ent schemas are contained in the same dataset. From our analysis, correcting these two issues allows us to import a much greater range of JSON use cases, across multiple RDBMS without significant modifications. We are able to hide the partitioning mechanism from the user by rewriting their queries to map to predefined partitions. This rewrite is also compatible with our pivot rewrite from Section 5.3.3.

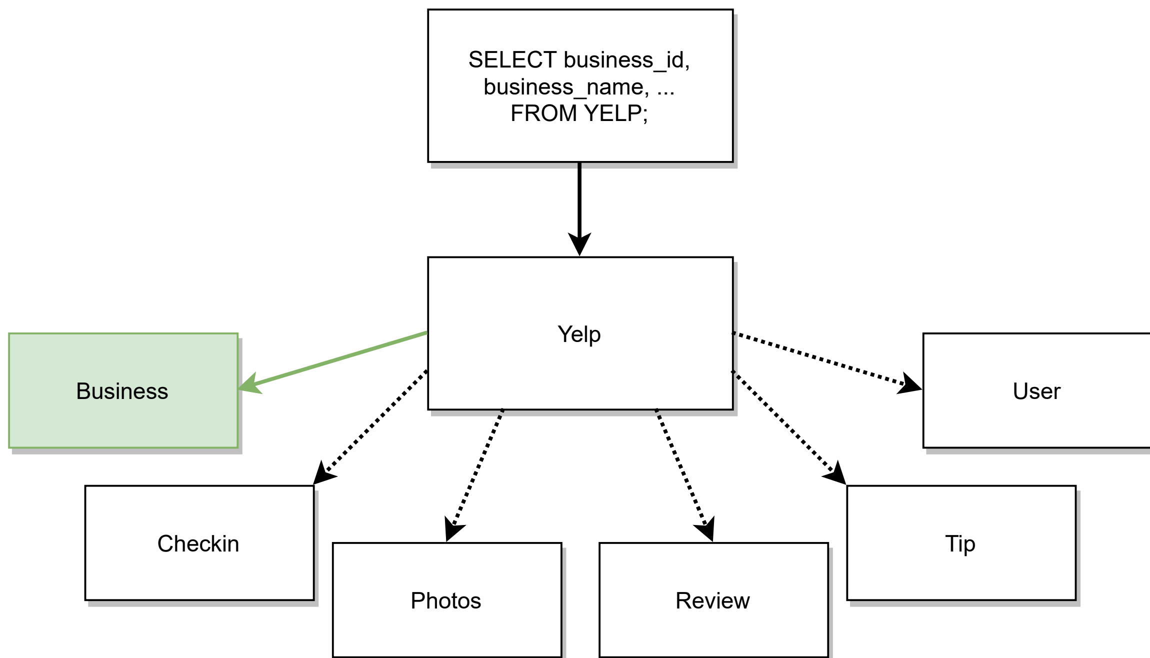


Figure 5.9: Partition selection based on projection pushdown.

Shown in Figure 5.9, we can use projection pushdown as a means to eliminate partitions. Such that a partition is only included, given a query predicate exists. Additionally, this optimization can be applied to self-referential joins, and given queries that span multiple partitions, nulls can be interpolated without an explicit read.

5.4.5 Partitioning Performance

Storing JSON in a partitioned tabular format has significant performance benefits over using native JSON types. For example, even though BSON does not require re-parsing, it still incurs at least one additional lookup due to attribute variations between records. More efficient data reads, advanced typing, better compression, all contribute to a performance improvement, beyond just ease of use. We test performance issuing various aggregate queries over our two ground-truth Yelp and GitHub datasets. We use three different layout strategies, native JSON, tabular unpartitioned, and tabular partitioned. When querying over our native JSON format, we use a JSON _table function to project only the required columns into a queryable form. Unpartitioned tabular format is the sparse table of every schema merged together. For example the schema contains all Yelp attributes across every record type. Finally, partitioned refers to our dense partitioned dataset, using the output from our Bimax-Merge algorithm. Our tests were run using a popular RDBMS system and their JSON function implementation. Unsurprisingly using the native JSON type produces the longest query times. Largely in part due to the overhead from re-parsing. Second, we see tabular unpartitioned has a performance benefit over native JSON. However, this is still far from ideal, requiring a full table scan and capturing many null values. Last, we see a significant performance increase by using partition elimination to only include rows from relevant partitions. Additionally, each table a more dense data representation, avoiding large null encodings.

5.5 Partitioning Conclusions

In conclusion, reformatting JSON data into a semi-relational format based on data cues can have substantial performance benefits. Detecting pivot opportunities can provide users more convenient query interfaces, skip the need for the inefficient explode operator, and avoid costly parses. Additionally, by separating collections which make up large portions of this data from core attributes, we can improve the performance of queries over the core relation. Next, we explore the performance benefit of using a JSON clustering algorithm for partitioning, reducing the amount of data scanned. Gaining a substantial benefit to query performance, all without the need to burden the user. Lastly, RDBMS often contain limiting factors, hindering direct JSON loading. By predictively pivoting collections, we substantially reduce the total number of columns for a direct translation. Then by enabling multiple partitions to have disjoint schemas, we can further reduce the number of columns, increasing the likelihood a direct JSON to table translation will be possible. This translation improving usability, and performance, especially for complex workflows.

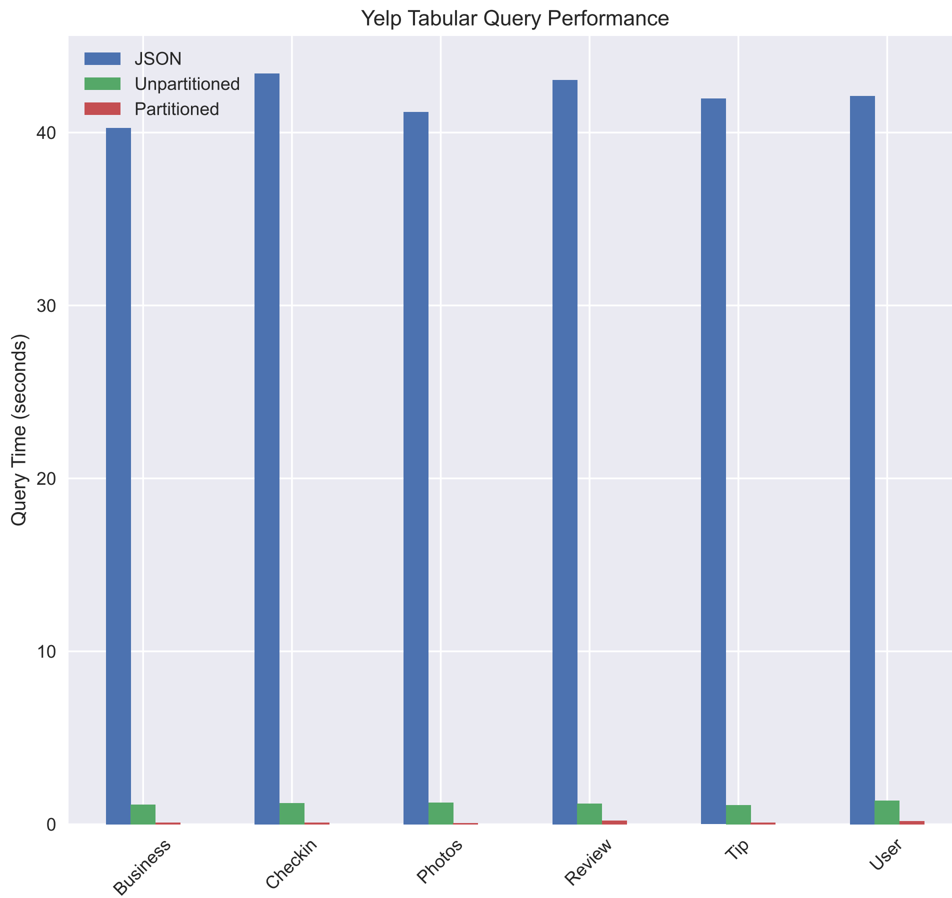


Figure 5.10: Performance of Yelp queries.

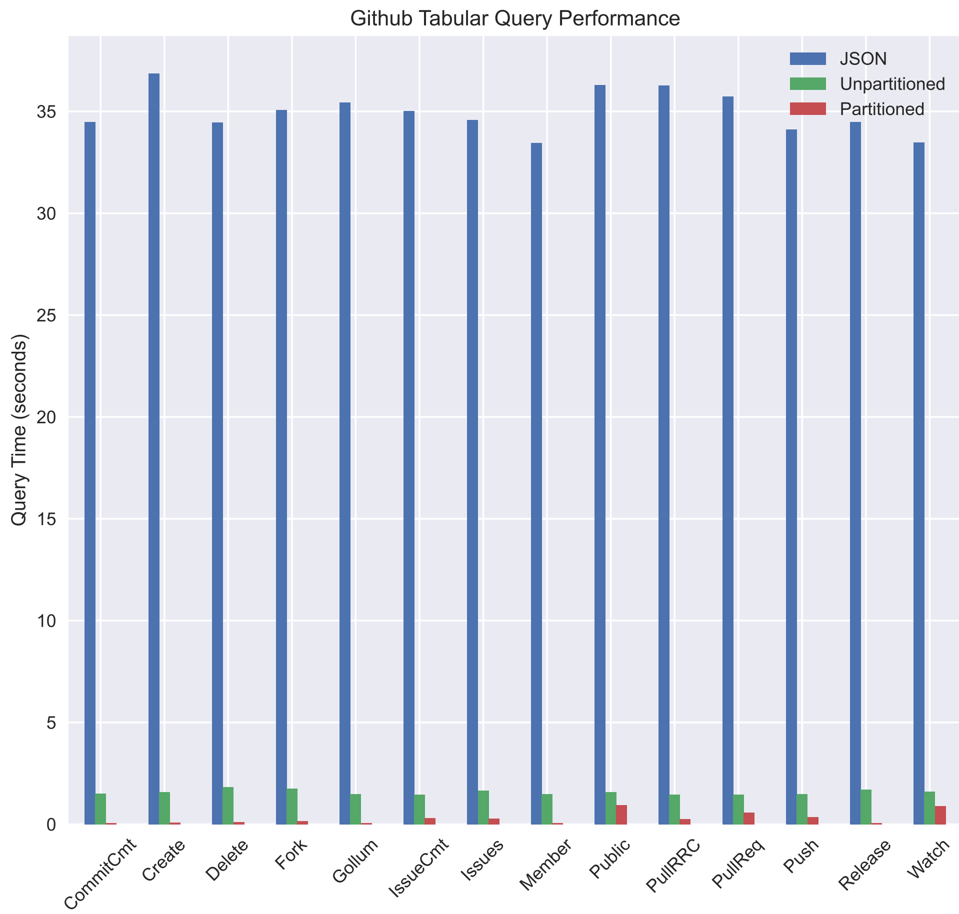


Figure 5.11: Performance of GitHub queries.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

JSON's flexible schema-on-read properties make it a developer favorite to store various types of data, and likely is here to stay. Today almost every major cloud based database offering has a number of connectors, integrations, or hacks to make importing JSON data possible due to high customer demand. We see this in offerings such as AWS Spectrum, Athena, and Aurora, to name a few, supporting everything from other JSON stores to raw file types. There is a clear need for full RDBMS JSON integration, beyond support for advanced string or BSON types and functions. In this dissertation, we investigated how this lack of direct native tabular storage impacts usability. Leading to frequent errors, performance degradation, and missing functionality. Worse yet, directly loading JSON into a tabular format is generally all but impossible. The amount of human effort to create a post hoc JSON ETL plan is infeasible, as JSON cleaning problems often scale with the size of the data

due to variations in records. These include embedded key-value stores, multi-typed attributes, and heterogeneous datasets to name a few.

First, we investigated how JSON work flows are often context dependent, particularly when joining with other relations. We propose and implement a process for query and data sharing, creating individual sanitized views to be reused and shared. Additionally, we implement a previously proposed method for data mining, to aid the users discovery process.

Second, after gathering a number of real world JSON datasets, we extended the scope of our data exploring algorithms, in particular targeting sanitizing JSON's heterogeneous properties. We implemented a number of common and specialized pattern mining and data exploring algorithms, and built an interactive user interface to convey these discovered patterns. From our observations, it appeared existing algorithms produced clustering results that were too error prone for a fully automated system, but useful for guiding user attention. Further investigation revealed to us the highly optional nature of JSON was skewing cluster results, leading us to develop our notion of *collections*, or embedded key-value stores.

In Chapter 4 we further our notion of collections, defining entropy metrics to efficiently detect this data pattern. We then created a novel clustering algorithm, specifically designed to take advantage of JSON's attribute optionality. Using this as a signal, we create natural partitions in the data which we experimentally show have minimal to no impact on validation, while reducing ambiguity by several magnitudes. This ambiguity being a fundamental data cleaning problem users face during the JSON ETL process, currently being accomplished by human elbow grease.

Finally, in Chapter 5 we create and apply a partitioning strategy generated from

our algorithm developed in Chapter 4. We devise a pivot table strategy for collections which improves performance and queryability. Next we apply our partitioning scheme and show significant query performance. Using our query rewriting strategy, this process is completely hidden from the user. All while improving performance and making no significant system modifications.

JSON data is here to stay, and the demand for multi-format queries is ever growing. Instead of creating one off partial solutions, poorly performing plugins, and domain specific languages, we should first look to reuse existing RDBMS functionality. In part, this means defining tabular relational mappings, promoting usability, scalability, and query reuse.

6.2 Future Work

There are a number of natural extensions and future applications of this work. First, while we captured a large number of previously unexplored patterns, there are still a number of unaddressed additional JSON patterns. For example recursion is a common pattern in particular schemas that we deemed as a whole problem on its own. Detecting recursion is notoriously computationally expensive and infeasible, and while our work does not inherently fail on such input, it does not attempt to be a general solution to capturing this pattern. Likewise, embedded foreign key dependencies and self-referential attributes are an additional avenue. A whole list of such patterns including enumerated types could take advantage of our base partitions to reduce computation and improve parallelization. In general, our approach can be used as a preprocessing step, for more computationally expensive operations.

Second, there are some situations where native JSON or sparse storage may provide a significant space saving. In this paper partitioning and collection detection significantly improves density. This however may not always be the case, and the benefits of partially storing some sparse or known unused columns in sparse form may be beneficial.

Finally, our process proposed in Chapter 4 is currently offline. Our roll-up merge algorithm can be reused to compute additional partitions, however this may be expensive and other methods may be more desirable. While most RDBMS systems enable a default partition which could be used for row overflow that does not obey partition rules, an online version would be worth exploring.

Bibliography

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

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[3] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: Efficient query execution on raw data files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 241–252, New York, NY, USA, 2012. Association for Computing Machinery.

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

[5] 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.

[6] L. Antova, C. Koch, and D. Olteanu. $10^{(10^6)}$ worlds and beyond: Efficient representation and processing of incomplete information. *The VLDB Journal*, 18(5):1021–1040, Oct. 2009.

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

[8] B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Reenactment for read-committed snapshot isolation. In *CIKM*, 2016.

[9] 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.

[10] 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.

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

[12] 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.

[13] S. Balakrishnan, A. Y. Halevy, B. Harb, H. Lee, J. Madhavan, A. Rostamizadeh, W. Shen, K. Wilder, F. Wu, and C. Yu. Applying webtables in practice. In *CIDR*. www.cidrdb.org, 2015.

[14] P. A. Bernstein, J. Madhavan, and E. Rahm. Generic schema matching, ten years later. *PVLDB*, 4(11):695–701, 2011.

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

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

[17] 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.

[18] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *SIGMOD*, 2005.

[19] BSON. Bson specification. https://bsonspec.org/, 2021.

[20] P. Buneman, S. Khanna, and W.-C. Tan. On Propagation of Deletions and Annotations through Views. In *PODS*, pages 150–158, 2002.

[21] P. Bürgisser. The complexity of factors of multivariate polynomials. In *FOCS*, pages 378–385. IEEE Computer Society, 2001.

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

[23] M. J. Cafarella, D. Suciu, and O. Etzioni. Navigating extracted data with schema discovery. In *WebDB*, 2007.

[24] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4(4):397–434, dec, 1979.

[25] G. Cong, W. Fan, F. Geerts, J. Li, and J. Luo. On the complexity of view update analysis and its application to annotation propagation. *TKDE*, (99):1–1, 2011.

[26] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo. Automating the database schema evolution process. *VLDB Journal*, 22(1):73–98, 2013.

[27] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: The prism workbench. *pVLDB*, 1(1):761–772, Aug. 2008.

[28] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic sql tuning in oracle 10g. In *VLDB*, 2004.

[29] N. Dalvi and D. Suciu. The dichotomy of probabilistic inference for unions of conjunctive queries. *J. ACM*, 59(6):30:1–30:87, jan, 2013.

[30] 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.

[31] M. DiScala and D. J. Abadi. Automatic generation of normalized relational schemas from nested key-value data. In *SIGMOD*, 2016.

[32] X. L. Dong, E. Gabrilovich, K. Murphy, V. Dang, W. Horn, C. Lugaresi, S. Sun, and W. Zhang. Knowledge-based trust: Estimating the trustworthiness of web sources. *pVLDB*, 8(9):938–949, 2015.

[33] D. Durner, V. Leis, and T. Neumann. Json tiles: Fast analytics on semi-structured data. In *Proceedings of the 2021 International Conference on Manage-*

ment of Data, SIGMOD/PODS '21, page 445–458, New York, NY, USA, 2021. Association for Computing Machinery.

[34] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. Graphviz - open source graph drawing tools. In *Graph Drawing*, pages 483–484, 2001.

[35] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *TKDE*, 19(1):1–16, 2007.

[36] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *TCS*, 336(1):89–124, 2005.

[37] 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.

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

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

[40] T. Green and V. Tannen. Models for incomplete and probabilistic information. In *Current Trends in Database Technology*, pages 278–296. 2006.

[41] A. Y. Halevy. Answering queries using views: A survey. *VLDBJ*, 10(4):270–294, 2001.

[42] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD Conference*, pages 1–12. ACM, 2000.

[43] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, may 2000.

[44] 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.

[45] IDC. Data creation and replication will grow at a faster rate than installed storage capacity, according to the idc global datasphere and storagesphere forecasts. https://www.idc.com/getdoc.jsp?containerId=prUS47560321, 2021.

[46] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.

[47] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, 2004.

[48] T. Imieliński and W. Lipski, Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, Sept. 1984.

[49] S. R. Jeffery, M. J. Franklin, and A. Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD*, 2008.

[50] S. C. Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, Sep 1967.

[51] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Wrangler: interactive visual specification of data transformation scripts. In D. S. Tan, S. Amershi, B. Begole, W. A. Kellogg, and M. Tungare, editors, *CHI*, pages 3363–3372, 2011.

[52] O. Kennedy and C. Koch. PIP: A database system for great and small expectations. In *ICDE*, 2010.

[53] P. Kumari, S. Achmiz, and O. Kennedy. Communicating data quality in on-demand curation. In *QDB*, 2016.

[54] Z. Liu, B. Hammerschmidt, D. McMahon, Y. Liu, and H. Chang. Closing the functional and performance gap between sql and nosql. pages 227–238, 06 2016.

[55] Z. H. Liu and D. Gawlick. Management of flexible schema data in rdbmss - opportunities and limitations for nosql -. In *CIDR*, 2015.

[56] Z. H. Liu and D. Gawlick. Management of flexible schema data in rdbmss-opportunities and limitations for nosql-. In *CIDR*, 2015.

[57] Z. H. Liu, B. Hammerschmidt, D. McMahon, Y. Liu, and H. J. Chang. Closing the functional and performance gap between SQL and NoSQL. In *SIGMOD*, 2016.

[58] 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.

[59] A. Maedche and S. Staab. Learning ontologies for the semantic web. In *ICSW*, 2001.

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

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

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

[63] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. *SIGMOD Rec.*, 30(2):307–318, May 2001.

[64] 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.

[65] A. Nandi and H. V. Jagadish. Assisted querying using instant-response interfaces. In *SIGMOD*, 2007.

[66] A. Nandi, Y. Yang, O. Kennedy, B. Glavic, R. Fehling, Z. H. Liu, and D. Gawlick. Mimir: Bringing ctables into practice. Technical report, The ArXiv, 2016.

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

[68] X. Niu, B. Arab, D. Gawlick, Z. H. Liu, V. Krishnaswamy, O. Kennedy, and B. Glavic. Provenance-aware versioned dataworkspaces. In *TaPP*, 2016.

[69] Oracle. Database json developer's guide: Json data guide. https://docs.oracle.com/cloud/latest/db122/ADJSN/json-dataguide.htm, 2017.

[70] Parquet. Parquet format. https://parquet.apache.org/, 2021.

[71] PostgreSQL. Postgresql json functions. https://www.postgresql.org/docs/current/functions-json.html, 2021.

[72] 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.

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

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

[75] G. Smith, M. Czerwinski, B. Meyers, D. Robbins, G. Robertson, and D. S. Tan. Facetmap: A scalable search and browse visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):797–804, Sept. 2006.

[76] 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.

[77] Solar Wings. Loggly. https://www.loggly.com/, 2021.

[78] W. Spoth, B. S. Arab, E. S. Chan, D. Gawlick, A. Ghoneimy, B. Glavic, B. Hammerschmidt, O. Kennedy, S. Lee, Z. H. Liu, X. Niu, and Y. Yang. Adaptive schema databases. In *CIDR*, 2017.

[79] 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.

[80] D. Suciu, D. Olteanu, C. Ré, and C. Koch. Probabilistic databases. *Synthesis Lectures on Data Management*, 3(2):1–180, 2011.

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

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

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

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

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

[86] 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.

[87] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

[88] K. Wang and H. Liu. Schema discovery for semistructured data. In *KDD*, volume 97, pages 271–274, 1997.

[89] 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.

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

[91] Y. Yang. On-demand query result cleaning. In *VLDB PhD Workshop*, 2014.

[92] Y. Yang, N. Meneghetti, R. Fehling, Z. H. Liu, and O. Kennedy. Lenses: An on-demand approach to etl. *VLDB*, 8(12):1578–1589, 2015.

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

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

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

[96] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. Db2 design advisor: Integrated automatic physical database design. In *VLDB*, 2004.

ProQuest Number: 28869383

INFORMATION TO ALL USERS
The quality and completeness of this reproduction is dependent on the quality
and completeness of the copy made available to ProQuest.

Distributed by ProQuest LLC (2022).
Copyright of the Dissertation is held by the Author unless otherwise noted.

This work may be used in accordance with the terms of the Creative Commons license
or other rights statement, as indicated in the copyright statement or in the metadata
associated with this work. Unless otherwise specified in the copyright statement
or the metadata, all rights are reserved by the copyright holder.

This work is protected against unauthorized copying under Title 17,
United States Code and other applicable copyright laws.

Microform Edition where available © ProQuest LLC. No reproduction or digitization
of the Microform Edition is authorized without permission of ProQuest LLC.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346 USA