Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data

Michael DiScala
Yale University
michael.discala@aya.yale.edu

Daniel J. Abadi
Yale University
dna@cs.yale.edu

ABSTRACT

Self-describing key-value data formats such as JSON are becoming increasingly popular as application developers choose to avoid the rigidity imposed by the relational model. Database systems designed for these self-describing formats, such as MongoDB, encourage users to use denormalized, heavily nested data models so that relationships across records and other schema information need not be predefined or standardized. Such data models contribute to long-term development complexity, as their lack of explicit entity and relationship tracking burdens new developers unfamiliar with the dataset. Furthermore, the large amount of data repetition present in such data layouts can introduce update anomalies and poor scan performance, which reduce both the quality and performance of analytics over the data.

In this paper we present an algorithm that automatically transforms the denormalized, nested data commonly found in NoSQL systems into traditional relational data that can be stored in a standard RDBMS. This process includes a schema generation algorithm that discovers relationships across the attributes of the denormalized datasets in order to organize those attributes into relational tables. It further includes a matching algorithm that discovers sets of attributes that represent overlapping entities and merges those sets together. These algorithms reduce data repetition, allow the use of data analysis tools targeted at relational data, accelerate scan-intensive algorithms over the data, and help users gain a semantic understanding of complex, nested datasets.

1. INTRODUCTION

Over the past two decades, self-describing semistructured data formats have become increasingly common solutions for storing and sharing information. As a recent example, NoSQL databases which implement key-value or document-based data models have gained traction among developers seeking to avoid rigid schemas. Human-readable key-value formats have emerged as the de facto message passing format used in modern APIs, allowing applications to interoperate with many services over standardized languages.

Although many semistructured formats exist, all typically support a set of common features. Notably, semistructured formats tend to inline attribute names within records instead of requiring clients to predefine their schemas. This allows records to contain arbitrary attributes with arbitrary types and allows records from within the same dataset to maintain different, potentially conflicting, attribute definitions.

Semistructured formats also tend to support deeply nested and hierarchical data structures. The popular JSON data format, for example, is recursively defined such that a key within a record may map to a nested record or even an array of nested records. Since most semistructured formats and their associated storage engines lack first-class support for modeling relationships across records (e.g. primary keys, foreign keys, joins, etc.), designers of semistructured data instead represent relationships with these nesting features and typically tend toward denormalized layouts.

The growing popularity of these formats has fueled interest in loading and processing semistructured data within relational database management systems (RDBMSes) so that more traditional business intelligence tools, query optimization techniques, and other tools from the relational world can be applied to these datasets [3, 4, 5, 6, 9, 21, 22, 23, 24, 27]. Efforts in this space typically propose mappings from semistructured formats to relational or mostly-relational structures that are compatible with standard SQL interfaces.

Once a mapping has been chosen, nearly any semistructured data can be represented within a relational framework. However, the schemas resulting from these mappings typically do not resemble anything a human designer would suggest. In some cases, this dissonance occurs because the mapping scheme is intended only to provide an efficient internal representation of semistructured data and does not attempt to provide a human-readable representation [3, 4, 9, 24, 21]. Even the mappings which attempt to generate human-readable schemas frequently fail to emulate human designers because they consider only the structure of input documents, rather than their values (e.g. for a key-value format, they consider the keys and their arrangement, but not their value distributions) [5, 23]. As a result, these mappings do not account for the denormalized data layouts that semistructured formats encourage. Even if an object is nested identically within two different records, these earlier strategies will doubly store the nested objects instead of creating a normalized, reference-based layout.

In this paper, we present a three-phase unsupervised machine learning algorithm that automatically designs a more traditional relational schema for an input dataset consisting of a set of related, denormalized, semistructured records. The first phase of the algorithm mines soft functional dependencies that exist within the semistructured dataset. It uses these dependencies to identify groups of attributes that are

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

ISBN 978-1-4503-3531-7/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2882903.2882924

likely to correspond to an independent semantic entity, and creates tables for these entities and associated foreign keys in parent tables. The second phase identifies entities found in the first phase with overlapping domains and merges them into a single entity structure. The final phase combines the intermediate results from the preceding two phases to produce a proposed schema that resembles a traditional normalized database schema (though not necessarily compliant with normalization schemes like BCNF, 3NF, or 4NF).

The advantages of this approach are two-fold. First, our algorithm simplifies the cognitively difficult task of exploring novel datasets by highlighting recurring structural and semantic patterns. Automated data exploration tools of this type are especially important while working with semistructured datasets, as these datasets are often semistructured precisely because their data sources are highly dynamic, and the set of attributes associated with any record is highly variable. Nonetheless, data exploration tools remain valuable even when the semistructured data originates from more ordered sources such as APIs, as these sources often lack extensive documentation and can be semantically inconsistent.

Second, our algorithm can significantly reduce the size of a dataset by eliminating redundancies. Whereas the denormalized data formats that we target store duplicate information whenever an instance of an entity is repeatedly referenced, normalized layouts avoid this repetition by storing instances of each entity a single time and replacing references to those instances with foreign keys. For semistructured formats that rely on inline attribute names, data size is further reduced by not repeatedly storing attributes names in the relational tables. The smaller size of our transformed datasets accelerates the scan-based queries that are especially common during an analyst's early interactions with a dataset.

Even as column-store technology reduces the need to normalize data in order to improve scan performance, scans of "dimension" tables corresponding to particular entities with a small number of rows can be performed much faster than scans of "fact" tables with a large number of rows. Hence, most column-stores continue to utilize separate tables for separate entities despite the overall reduced benefits of normalization. Furthermore, even when column-stores maintain physically denormalized data layouts, users still create logical views of the data in order to avoid losing the semantic connection to the actual entities within the dataset.

Therefore, the first two phases of our algorithm, which identify and match entities, are useful within both the column- and row- store contexts. The third phase of our algorithm, which analyzes the output of the preceding phases and decides which relations to normalize, is parameterized to target either a column- or row- store. If the final schema is destined for a row-store, phase 3 will normalize relations more aggressively. Conversely, if the schema is destined for a column-store, phase 3 will pre-join relations more frequently.

In Section 2, we summarize earlier works related to our algorithm. In Section 3, we explain each of our algorithm's three phases. In Section 4, we present our algorithm's performance on three semistructured datasets. In Section 5, we discuss our algorithm's limitations and areas for further investigation. In Section 6, we summarize our findings.

2. RELATED WORK

Functional dependencies: Largely due the close relationship between normalization and functional dependencies, functional dependency detection has been studied extensively [1, 2, 13, 14, 15, 18]. Mannila et al. prove that identifying the complete set of functional dependencies present within a relation is, in the worst case, exponential in the number of attributes [18]. Our work avoids this exponential search space by mining functional dependencies between pairs of attributes rather than arbitrary subsets of attributes. We note that this search space can be further reduced by applying the transitivity axioms that Bell presents [1].

Several of these dependency detection efforts attempt to identify "soft functional dependencies" or "approximate functional dependencies," which are similar to traditional functional dependencies, except that they are required to hold for most — but not all — data instances [13, 14, 15]. In particular, Ilyas et al. propose an approximate functional dependency detector which they use to improve query optimizer performance. We draw from their construction, applying it within the context of automated normalization and modifying it to function more robustly for skewed attributes.

Semistructured-to-relational mappings: Our work relates closely to previous mappings of semistructured data into relational data structures [4, 5, 6, 9, 21, 22, 23, 24, 27].

The most straightforward of these earlier efforts provide generic mapping rules that do not necessitate upfront analysis of the input dataset. One example, Chasseur et al.'s Argo, encodes JSON documents as ternary relations consisting of "object id," "attribute name," and "attribute value" columns [4]. Each attribute at the end of a nesting path through a record (i.e. a leaf attribute) is represented with a row in the ternary relation. Generic mappings of this sort achieve our goal of encoding semistructured documents in a relational system, but do not leverage the specific characteristics of the input dataset to achieve a more performant encoding or facilitate data exploration.

Other works are more similar to ours and generate schemas tailored to the input dataset. For example, Deutsch et al. present a language called STORED that maps semistructured data onto a combination of standard relational tables and an auxiliary data structure referred to as an "overflow graph"[1] [5]. They further present a data mining algorithm based on Wang and Liu's earlier work [25, 26], which identifies structures that appear frequently within the input dataset. They automatically translate these frequent structures into STORED mappings for the input data.

Shanmugasundaram et al. similarly present a mapping strategy based on structural analysis of the input dataset [23]. They design schemas for XML datasets by analyzing a graph of the DTD elements present in the input data; they apply a set of heuristics which dictate whether an element should be materialized as its own table or inlined within a parent element's table. Although their algorithm cannot operate directly on data that lacks a DTD specification, it can be applied within the schemaless context if the input dataset is first preprocessed by a schema discovery algorithm. One such discovery algorithm is Garofalakis et al.'s XTRACT system which produces a DTD from the arrangement of elements within a set of input XML documents [10].

Despite their overall differences, STORED's and Shanmugasundaram's approaches are similar in that they both extract their schemas from the original structures of the dataset and do not consider the distribution of data within

[1]This auxiliary structure limits STORED's ability to integrate with vanilla RDBMSes

those structures. In contrast, our approach ignores the original structure of the input and instead depends on patterns in the data (functional dependencies) to guide its schema generation. Our data-driven approach has three advantages over the prior structural approaches: 1) our analysis remains possible even when hierarchical structures are not present in the input dataset (e.g. flat CSV data files), 2) our algorithms can produce structures that were not present in the original encoding, and 3) our use of functional dependencies allows us to compress the input dataset through deduplication.

One previous approach to schema mapping that is sensitive to data distributions is Bohannon et al.'s LegoDB, which maps XML data to relational structures according to a search process that minimizes the cost of a user-specified workload [3]. LegoDB collects statistics about individual data attributes that it then feeds to a traditional query optimizer in order to estimate the cost of its workload for many candidate schemas. Since the system does not collect joint statistics about input attribute pairs, however, it cannot leverage functional dependencies as we do in our algorithm. Moreover, given that LegoDB requires users to define a query workload in advance of schema generation, it is less useful within the context of data exploration where queries are unlikely to be known in advance.

Another approach to schema mapping that is sensitive to data distributions is some very recent work by Pham et al. in the context of graph (RDF) data [19]. This work scans through a vertex-edge-vertex (SPO) dataset to find characteristic sets (CSs) consisting of edge-labels that co-occur as outgoing edges from multiple vertices. The CSs correspond to entities from which relations can be derived. Pham et al. use structural information in the graph (direct edges to non-literal vertices) to discover relationships between entities. In contrast, our algorithm does not rely on structural clues in the dataset to discover relationships.

A final approach related to our work is the xCurator system, which addresses our problems of schema generation, entity identification, and entity matching in the RDF context [12]. xCurator extracts entities by constructing and analyzing a structure graph from the dataset. Again, our algorithm differs in that it ignores structural information.

Schema matching: The second phase of our algorithm is closely related to earlier work on schema matchers, which identify mappings between independently designed schemas that model the same underlying domain. Rahm and Bernstein present an extensive survey of past approaches to this problem [20]. Despite the breadth of past work, earlier approaches are largely incompatible with our needs in phase 2: some require user-input (we intend our algorithm to be unsupervised), some rely on similarities in attribute names (we intend our algorithm to be entirely data driven), and some require rich schema information such as types or foreign-keys (we assume that this context will sometimes be unavailable). In this paper, we thus propose a matching scheme tailored to phase 2's restricted schema matching problem.

Our phase 2 algorithm borrows Li and Clifton's strategy of clustering attributes within a single schema into categories in order to reduce the number of potential matches the algorithm must consider [16].

3. ALGORITHM OVERVIEW

In this section, we provide a high-level overview of our algorithm, reserving more detailed discussion for later sections. We use a toy dataset of submissions to a journal in order to ground our description in concrete examples. Listing 1 shows a sample record from this dataset, which is encoded in the JSON format. The record includes the submission's title, author, and assigned reviewer, along with the author's and reviewer's associated institutions. A complete dataset would include many examples of this kind of submission record. While we expect that the structure of each record will vary, we assume that all of the records logically represent the same kind of object or entity.

```
{ 'submissionId': 1576,
  'title': 'Perils of DB',
  'author': {
    'email': 'js@e.pfl',
    'name': 'J. Snow',
    'institution': {
      'name': 'EPFL',
      'number': 1 } },
  'reviewer': {
    'email': 'dd@ca.uk',
    'name': 'D. Duck',
    'institution': {
      'name': 'Cambridge',
      'number': 14 }}}
```

Listing 1: Sample JSON

We begin constructing a relational schema for the data by temporarily transforming the raw data records from their semistructured format into a flat table akin to the "universal relation" model [7, 17]. In order to transform the JSON records from our journal submissions example, we create a table with a column for every attribute that originally contains an atomic value (e.g. a string or integer), where the column's name is the concatenated path through any nesting required to reach the atomic value. We then populate the table with a row for every record in the dataset by entering each record's values into their corresponding table column. Figure 1 illustrates this flattening process, as its first row contains the flattened form of the record shown in Listing 1 and its remaining rows represent additional submission records. Since our toy example does not rely on semistructured features more complex than nested attributes, this simple transformation procedure is sufficient for our needs here. We describe techniques for accommodating other semistructured features (e.g. creating relationship tables to model nested arrays of objects) in Appendix A.

This flat representation does not meet either of our primary goals: (1) offering users semantic insight into their datasets and (2) creating an approximately normalized schema for a dataset. This encoding nevertheless serves as a convenient starting point because it decouples our algorithm from a specific semistructured format and ensures that our results reflect the patterns implicit in the original data rather than the original data structure. We consciously ignore structural cues such as JSON's implicit object boundaries because these cues frequently derive from assumptions and needs that may not be borne out by the data itself (e.g. for development convenience, separate entities may be combined into a single JSON object). As our algorithm progresses, we refine this flat representation to expose more of the implicit structure of the data and thereby construct a more traditional relational schema.

In phase 1, we identify functional relationships between attributes in the dataset and leverage those relationships to decompose our initial table into a collection of smaller tables joined by primary-to-foreign key relationships. While the strict definition of a functional dependency from an attribute A to another attribute B requires that n-to-one relationships exist between *all* values of A and B, we adopt Ilyas et al.'s notion of a "soft" functional dependency which

subId	*title*	*authEmail*	*authName*	*authIName*	*authINum*	*revEmail*	*revName*	*revIName*	*revINum*
1576	Perils...	js@e.pfl	J. Snow	EPFL	1	dd@ca.uk	D. Duck	Cambridge	14
1680	Broken...	js@e.pfl	J. Snow	EPFL	1	nd@h.edu	N. Dame	Harvard	55
1559	Wins...	dd@ca.uk	D. Duck	Cambridge	14	js@e.pfl	J. Snow	EPFL	1

Figure 1: Collection of flattened records from the journal dataset (nested paths in column names shortened to save space)

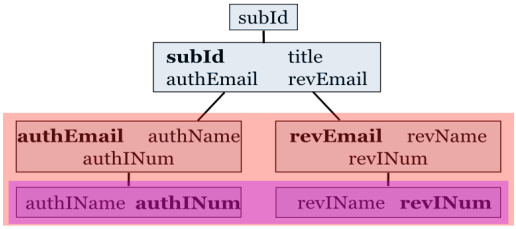

Figure 2: The ideal phase 1 attribute tree for the journal submission dataset, with parent attributes in bold and phase 2 entity matches highlighted in red and purple.

instead requires that the relationships exist for *most* values of A and B [14]. At the highest level, we identify these soft functional dependencies by considering pairs of attributes and observing how frequently the necessary n-to-one relationships exist between their values (described in Section 3.1.2). Even within the compact journal submission example, we would likely identify many functional relationships between pairs of attributes. Among others, we would identify that all attributes depend upon "subId" and that all attributes prefixed with "auth" depend upon "authEmail."

Based on an analysis of the identified dependencies, we group attributes that exhibit similar functional relationships and materialize these groups as new tables. In order to separate a group of attributes into their own table, every member of the group must depend upon some other attribute in the dataset. We refer to the attribute upon which all members depend as the *parent* of the group. For example, we might group the attributes prefixed with "auth" together because they share a common dependency on "authEmail." These attributes can be separated into their own table that is linked to the original through a primary-to-foreign key relationship on their parent attribute, "authEmail."

While this "auth" decomposition is valid, it is not the most expressive arrangement of the input: the dataset contains a chain of dependencies from "subId" to "authEmail" to "authINum" to "authIName," but the current decomposition reveals only functional relationships from "authEmail" to fields prefixed with "auth." We must further decompose our tables in order to expose the remaining functional relationships. To this end, we generate the attribute groups according to an algorithm that attempts to create the longest possible chains of groups from the identified functional dependencies.

The table structure resulting from this decomposition process can be modeled as a rooted tree in which every node represents a relation and every edge represents a primary-to-foreign key relationship. Figure 2 shows a potential attribute tree for the data contained in the journal submission example (ignore the purple and red boxes for now).

Phase 2 of the algorithm now begins. Unlike phase 1, which exclusively identifies relationships between attributes within a single record, phase 2 identifies relationships between *groups* of attributes *across* records. In particular, this phase searches for instances where the dataset contains multiple attribute groups that seem to represent similar entities in the dataset's domain model. As an example of this pattern, consider the group of journal submission attributes that contains "authIName" and "authINum" alongside the group that contains "revIName" and "revINum." Despite their distinct naming schemes, each group contains attributes describing a logical institution entity. When the fields are prefixed with "rev," the institution is associated with the reviewer, and when the fields are prefixed with "auth," the institution is associated with the author; regardless, the two groups always describe an institution. Moreover, the institutions that appear in the "revI" prefixed fields on some records will likely appear in the "authI" prefixed fields on others, as a single institution's professors may sometimes serve as reviewers and other times as authors.

We can therefore improve the relational design of our schema by merging these attribute groups together so that the resulting schema contains a single relation composed of all institutions, regardless of their author/reviewer classification. This merging operation eliminates redundant storage of records representing the same institutions in different tables, thereby reducing the overall size of the dataset. More importantly, it alerts users to the fact that some branches of the tree refer to an overlapping domain.

From a high level, phase 2 quantifies the overlap between the domains of subtrees within the attribute tree and declares that subtrees represent the same type of entities when their domains significantly overlap (described in Section 3.2). Phase 2 annotates the attribute tree with entity overlap information so that phase 3 can later merge the branches together. The purple and red boxes shown in Figure 2 represent the annotations we expect phase 2 to add to the journal submission attribute tree. The purple portion shows the earlier-described relationship between the "revI" prefixed fields and the "authI" prefixed fields. The red portion shows the overlapping relationship between the "rev" prefixed fields and the "auth" prefixed fields, which match because both groups contain attributes related to a person entity.

Phase 3 creates the final relational schema for the dataset by merging the subtrees identified by phase 2 and condensing long dependency chains (described in Section 3.3). Although these long chains reveal dependencies in the dataset, phase 3 condenses them because excessively normalized schemas are (1) less readable, (2) bloated by foreign keys, and (3) slow to query due to extensive JOIN operations.

Figure 3 shows one schema that the algorithm might create for the journal submission dataset. It contains three relations: one representing the root submission table, one representing people entities, and one representing institution entities. These relations are stitched together by primary-to-foreign key relationships that the algorithm introduces.

3.1 Phase 1 in-depth

In this section, we expand our earlier description of phase 1's mining of soft functional dependencies and its decomposition of the initial flat table into a tree of smaller tables.

subId	*title*	*revID*	*authID*	*ID*
1576	Perils...	3	1	1
1680	Broken...	5	1	3
1559	Wins...	1	3	4

email	*name*	*instID*	*ID*
js@e.pfl	J. Snow	1	1
dd@ca.uk	D. Duck	4	3
nd@h.edu	N. Dame	5	5

INum	*IName*	*ID*
1	EPFL	1
14	Cambridge	4
55	Harvard	5

Figure 3: The final relational schema generated for the data from the journal submission example.

3.1.1 Soft vs hard functional dependencies

We choose to mine soft functional dependencies instead of traditional functional relationships for two reasons. First, by mining soft relationships we allow our algorithm to operate on imperfectly curated datasets that contain a few erroneous records in violation of otherwise valid functional dependencies. Data inconsistencies of this type can arise in most any format, but are especially common within semistructured datasets because these formats often do not support schema level constraints (e.g. type or foreign key constraints).

Second, we favor soft functional dependencies because they philosophically align with our goal of helping users learn about their datasets. Even if they do not hold globally, soft functional relationships reflect the broad contours of the training dataset and can help users to understand the data structures prevailing within the majority of their data.

3.1.2 Identifying soft functional dependencies

We detect a soft functional relationship from a_1 to a_2 if the values of a_1 usually exhibit n-to-one relationships with values of a_2. Ilyas et al. [14] provide a light-weight formula to approximate the frequency of these n-to-one relationships:

FORMULA 3.1.1. *Let* $strength(a_1,\ a_2)$ *be:*

$$strength(a_1,\ a_2) \ = \ \frac{\#\ of\ unique\ values\ of\ a_1}{\#\ of\ unique\ (a_1,\ a_2)\ value\ pairs}$$

The strength measure ranges from 0 to 1, and is positively correlated with the degree to which a_1 predicts a_2. When individual values of a_1 pair with many values of a_2, the denominator grows and the strength measure approaches zero. Conversely, when a hard functional dependency exists and every value of a_1 pairs with a unique value of a_2, this measure equals one.

We found that Formula 3.1.1 works well most of the time. However, when a_2's distribution is significantly skewed toward a single value, it becomes possible that the strength value will be high as a result of coincidence rather than as a result of the dataset's semantics. The core problem is that a skewed value distribution of a_2 can produce patterns in the training data that resemble an n-to-one relationship where many values of a_1 map to the (skewed) modal value of a_2. We address this issue by modifying Formula 3.1.1 to discount the modal value of a_2:

FORMULA 3.1.2. *Let* v_{2freq} *be the modal value of* a_2 *in the dataset and let* $m(a_1, a_2)$ *be the number of unique* $(a_1,\ a_2)$ *value pairs where* a_2 *equals* v_{2freq}*.*

$$strength(a_1,\ a_2) \ = \ \frac{\#\ of\ unique\ values\ of\ a_1\ -\ m(a_1,a_2)}{\#\ of\ unique\ (a_1,\ a_2)\ pairs - m(a_1,a_2)}$$

Given a threshold, α, which corresponds to the allowable "softness" in the functional relationships, we use Formula 3.1.2 to discover soft functional dependencies in the dataset:

CONDITION 3.1.1. *Let* $strength(a_1,\ a_2)$ *be defined as in Formula 3.1.2, and* $density(a)$ *be the fraction of records where* a *is non-NULL. A soft functional dependency from* a_1 *to* a_2 *exists when:*

(1) $strength(a_1,\ a_2)$ *has a non-zero denominator*
(2) $strength(a_1,\ a_2) > \alpha$
(3) $density(a_1)\ \geq density(a_2)$

The threshold α should be set to a number close to 1 (0.99 in our experiments) in order to guarantee that a_1 reliably indicates a_2's value within the training set. This guarantee allows us to treat these relationships as approximate primary-to-foreign key links later in phase 1 and, consequently, is essential to our ability to decompose the initial flat table into many smaller relations.

3.1.3 Enumerating dependencies

For each pair of attributes, Condition 3.1.1 can be checked with algorithms that are either linear or linearithmic (n log n) in the size of the dataset. For simplicity, we identify dependencies by applying the conditions to every pair of attributes, in both directions ($O(N^2)$ comparisons where N is the number of attributes). This process enumerates all combinations of individual attributes that meet our conditions. However, this process cannot discover attributes that are functionally dependent on groups of attributes.

Our implementation can be optimized by leveraging the transitivity of functional relationships to avoid performing redundant dependency tests. Bell fully evaluates this strategy of optimizing dependency enumeration [1].

3.1.4 The longest path heuristic

After discovering dependencies, phase 1 forms groups of attributes. As described in Section 3, phase 1 forms these groups by identifying the longest chains of functional dependencies present in the dataset. Phase 1 employs this heuristic because it ensures that the attribute tree includes as many functional dependencies as possible and thus includes as much semantic information as possible. This strategy offers flexibility to the algorithm's later phases, allowing phase 2 to match entities within the tree's branches and phase 3 to selectively materialize the dependencies as relations.

Phase 1 begins its grouping process by constructing a dependency graph where nodes represent attributes and directed edges represent functional relationships (edges run from parent attributes to dependent attributes). The graph contains a node for every attribute in the training set and directed edges modeling each of the previously discovered dependencies. The graph also contains a special "ROOT" node that has outgoing edges to every other node. Figure 4(c) shows part of the journal submission example's graph.

For every attribute, a, phase 1 then computes $lp(a)$ which denotes the longest path through the dependency graph from "ROOT" to a. Ties are resolved according to the strength of the last functional dependency on the path, or are resolved randomly if the strengths are also tied.

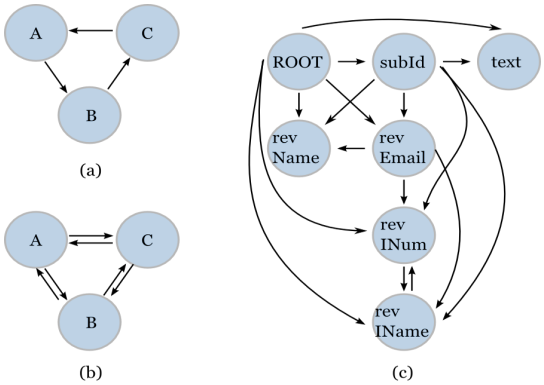

Figure 4: (a) An unlikely dependency graph cycle (b) A likely dependency graph cycle. (c) A subset of the dependency graph for the journal submission dataset.

Consider the value of $lp(revEmail)$ in Figure 4(c). There are two paths to "revEmail" in the dependency graph: "ROOT" → "revEmail" and "ROOT" → "subId" → "revEmail." Phase 1 chooses the latter path as $lp(revEmail)$ because it is longer.

In the general case, computing $lp(a)$ is an NP-hard problem and requires a brute-force examination of every possible path from "ROOT" to a. When the input is a directed acyclic graph (DAG), however, a linear time solution exists. Although we cannot assume that the phase 1 dependency graphs will be DAGs, the transitive nature of functional dependencies allows us to assume that the cycles in the graph will result from bidirectional dependencies between pairs of attributes as shown in Figure 4(b). Multi-node cycles similar to Figure 4(a) are only possible if a mistake was made in identifying functional dependencies, since multi-node cycles without bidirectional edges require that some functional dependencies do not exhibit the transitive property. We thus can usually transform our initial dependency graph into a DAG by removing bidirectional relationships. We remove a bidirectional relationship between attributes a_1 and a_2 according to the following rules:

1. If $descendants(a_1) - \{a_2\} = descendants(a_2) - \{a_1\}$, we merge the attributes together into a single node in the dependency graph. In this case, the attributes participate in the same functional dependencies so we treat them as a single unit. We would use this strategy to eliminate the cycle between "revIName" and "revINum" in Figure 4(c).
2. If the nodes cannot be merged, we break the cycle by removing the edge beginning at the node with fewer descendants in the graph.
3. If neither of the above conditions hold, we delete the edge with the lower strength, breaking ties randomly.

Once the longest paths have been identified, phase 1 forms attribute groups. For every attribute, it computes $parent(a)$ which denotes the attribute immediately preceding a on $lp(a)$. The algorithm then groups all attributes that share the same value of $parent(a)$. Throughout our remaining discussion, we refer to these groups by their shared parent attributes; we define A_{p1} to be the attribute group containing all attributes for which $parent(a)$ equals $p1$.

Finally, the algorithm splits the flat table into these attribute groups to form the tree structure that comprises the phase 1 output (as described above and shown in Figure 2).

3.2 Phase 2 in-depth

Phase 2 searches within the attribute tree to discover semantically equivalent entities that are embedded in multiple places within the dataset, as occurs in the algorithm overview's journal submission example with person entities (which exist as both authors and reviewers) and institution entities (which are nested underneath both authors and reviewers). In order to identify this behavior, phase 2 analyzes collections of attributes that — despite having different names (e.g "revIName" vs. "authIName") — have value distributions that suggest they represent the same entity.

We begin with Section 3.2.1 which explains why subtrees of the phase 1 attribute tree are a natural choice for searching for matching entities. We continue with Section 3.2.2 which presents our conditions for determining that two subtrees likely contain the same type of entities. For clarity, we first describe our algorithm in its most uncomplicated, but lowest performing variation. However, we describe four optimizations to this algorithm in Appendices B, C, D, E.

3.2.1 Entities and subtrees

We define entities as elements of the dataset's domain model that contain descriptive attributes and a unique identifier. For instance, the journal submission example includes person entities, which are uniquely identified by their emails and described by their names and institutional affiliations.

We classify attributes as "descriptive attributes" for a particular type of entity by examining the frequency with which the attributes change for a specific entity instance. On one hand, some attributes frequently hold different values throughout the dataset even when they are associated with the same entity. These frequently changing attributes typically model time-dependent information, where each record in the dataset contains a snapshot of the attribute's value. We are unconcerned with these kinds of attributes in phase 2, as they are not amenable to normalization and are not redundantly stored across repeated occurrences of an entity.

Instead, phase 2 searches for attributes that remain relatively fixed across all references to a specific entity; these attributes model concepts that are not time-dependent, such as a person's name. By virtue of their stability across references to single entity, a strong n-to-one relationship exists from an entity's unique identifier to its stable attributes. These stable attributes consequently tend to descend from their entity's unique identifier in the phase 1 attribute tree. Entities in a dataset thus correspond to subtrees of the phase 1 attribute tree:

DEFINITION 1. *If A_p is an attribute group in the tree with parent attribute p, then there exists a subtree beginning at A_p that contains all attribute groups A_{attr} such that the parent of A_{attr} equals p or one of p's descendants.*

For example, within the journal submission data, there is a perfect correspondence between the attribute tree's five subtrees and the domain model's 5 entities (1 submission entity, 2 person entities, and 2 institution entities)[2].

Since each semantic entity in the dataset usually corresponds to a subtree in the phase 1 attribute tree, entity matching reduces to a pairwise comparison of all subtrees ($O(N^2)$ in the number of subtrees). In many cases, this process identifies redundant matches because matching subtrees

[2]In general, not **every** subtree corresponds to an entity.

authEmail	*authName*	*authIName*	*authINum*	*revEmail*	*revName*	*revIName*	*revINum*
js@e.pfl	J. Snow	EPFL	1	js@e.pfl	J. Snow	EPFL	1
js@e.pfl	J. Snow	EPFL	1	js@e.pfl	J. Snow	EPFL	1
dd@ca.uk	D. Duck	Cambridge	14	dd@ca.uk	D. Duck	Cambridge	14

Figure 5: The result of joining the flat table on $authEmail = revEmail$.

often contain sub-subtrees that also match. Our algorithm exploits this redundancy to gain additional insight into the input data's semantics (discussed in Appendix C).

3.2.2 Detecting matching subtrees

Phase 2 uses a three step process to determine whether the subtrees S_1 and S_2 model the same underlying entity:

Step 1: This step identifies all pairs of training records where the subtrees seemingly describe the same entity. For example, when considering the person subtrees, $A_{authEmail}$ and $A_{revEmail}$, the algorithm must find all pairs of records where the author for the first record is the reviewer for the second. This process is essentially a join operation that returns records with matching values within their S_1 and S_2 subtrees. Given that the algorithm does not yet understand the relationships between S_1's and S_2's attributes, it cannot perform a direct subtree match. Instead, phase 2 approximates this join on the entire subtree by performing a join on the parent attributes of the two subtrees[3]. Due to the functional relationships from the parent attribute to the subtree's other attributes, this simpler join identifies all pairs of records that might contain the same entity. Figure 5 illustrates this process for the $A_{authEmail}$ and $A_{revEmail}$ subtrees. Appendix B outlines an optimization of this join.

When the result of this join is empty, we conclude that there is no possibility of S_1 and S_2 containing the same type of entity and skip the remaining two steps for this pair. This occurs whenever the subtrees' parent attributes have no common values, as in the example of $A_{authEmail}$ and $A_{authINum}$. When the result is non-empty, the joining records may reflect that 1) the two subtrees are related and the matches contain the same entities, or 2) the subtrees are unrelated and their parent attributes coincidentally overlap. The remaining two steps distinguish between these cases.

Step 2: This step analyzes the results from step 1 in order to identify correspondences between S_1's attributes and S_2's attributes. If the two subtrees truly represent the same underlying type of entity, we expect them to contain pairs of corresponding attributes. Consider, for example, the subtrees $A_{authEmail}$ and $A_{revEmail}$, which both model person entities. Their corresponding attributes can be mapped together as follows: "authEmail" maps to "revEmail," "authName" maps to "revName," "authINum" maps to "revINum," and "authIName" maps to "revIName." In this case, it is fairly trivial for a human observer to infer the appropriate mapping. In many other datasets, however, the mapping between subtrees will not be so obvious. For generality, we do not rely on information from attribute names.

Step 2 constructs attribute mappings by considering the frequency with which attributes from the two subtrees contain the same values on the same records in the step 1 join result. It tracks these frequencies in an attribute matrix that contains a cell for every pairing of S_1's and S_2's attributes.[4] We denote these pairs as (a_1, a_2). Every cell, $attrMatrix[a_1][a_2]$, is initialized to zero. The algorithm then iterates through every row, R, from the join result and increments $attrMatrix[a_1][a_2]$ by 1 if $R[a_1]$ equals $R[a_2]$.

For example, the matrix generated from the join in Figure 5 will have the value 3 in the cells for the pairs (revEmail, authEmail), (revName, authName), (revIName, authIName), and (revINum, authINum). All other cells contain zero.

From the attribute matrix we construct the mapping between S_1's and S_2's attributes. We attempt to map the S_1 attributes to the S_2 attributes that most frequently share their values across the join set. Formally, we define:

DEFINTION 2. *For every attribute a_1 in S_1, $match(a_1)$ equals the attribute a_2 in S_2 such that*

1. *$R[a_1] = R[a_2]$ for a threshold percentage of joined records:*

$$\frac{attrMatrix[a_1][a_2]}{\#\ records\ in\ join\ set} \geq \alpha^2$$

2. *$attrMatrix[a_1][a_2] \geq attrMatrix[a_1][a_2']$, for all other attributes a_2' in S_2*

Definition 2's first requirement excludes matches between attributes that coincidentally share the same value for some records in the join set. The threshold parameter α is the same α from Condition 3.1.1, which corresponds to the acceptable "softness" of the dataset. However, α^2 is used as the threshold here, since the errors multiply through a join. The *match* function is not guaranteed to be defined for all attributes; we refer to unmatched attributes as orphans. The definition's second requirement selects the best amongst multiple matches for an attribute that are above threshold. It allows *match* to map a single attribute from one tree to multiple attributes in the other when ties occur.

In Appendix E we describe an optimization to this process which clusters attributes based on their domain values. The algorithm leverages this clustering to improve accuracy and performance by excluding attribute matches across clusters.

Step 3: Finally, phase 2 analyzes the results from the previous two steps to determine whether the two subtrees likely model the same underlying entity. Phase 2 identifies an entity match only if the following three conditions hold:

1. *The subtrees' join result is at least (1-α)% the size of the original dataset.* This condition eliminates matches that result from extremely small, coincidental overlaps in the domains of the subtrees' parent attributes.
2. *The match function is defined for the majority of the attributes contained within the smaller subtree.* This condition allows us to identify subtree pairs that partially match, while also eliminating weak matches that may result from noise in the attribute matrix computations.

[3]When the parent attribute of a subtree is a merged attribute created during phase 1, we perform multiple joins and try all possible parent combinations. For example, if $parent(S_1)$ is the merged attribute "A, B" and $parent(S_2)$ is "C," we perform the join on both A = C and on B = C.

[4]In practice we do not compare every pair of attributes. See Appendix E for more details.

3. *The match function is defined for attributes other than the subtrees' parent attributes.* Parent attributes are guaranteed to match with each other since the subtrees are joined on these attributes. Thus, the *match* function provides no new information for parent attributes.

After identifying entity matches, phase 2 performs additional analysis to find partners for some orphaned attributes. The algorithm analyzes nested matches (e.g. the person and institution matches from the journal submission example) to find attribute matches that were overlooked during earlier phases. We outline this process further in Appendix D.

3.3 Phase 3 in-depth

Phase 3 produces a physical schema for the input data and a mapping of input attributes onto that physical schema. This phase begins by merging together the phase 2 entity matches. An entity's subtrees are merged in three steps:

1. Form groups of subtrees such that the members of each group are all phase 2 matches with one another. For example, if the phase 2 output contains the matches A-B, A-C, and B-C, then we form a group of all three trees.[5]
2. For each group of subtrees, designate the one with the most attributes as the canonical subtree for that group (resolve ties arbitrarily). Remove all non-canonical subtrees from the attribute tree.
3. Map each non-canonical subtree to its corresponding canonical subtree according to the pair's phase 2 *match* function. If the *match* function is undefined for any non-canonical attribute, add an attribute to the root of the canonical subtree and map the orphan to this new attribute. This ensures that every input attribute can be mapped to something in the physical schema.

Phase 3 then transforms the tree of canoncial subtrees into its final schema. The schema begins with a relation that corresponds to the root attribute group and contains all of the root attributes. The algorithm then performs a breadth-first traversal of the tree in order to build the complete schema. During each iteration of this traversal, the algorithm decides whether to create a new relation for the attribute group it's considering or to merge that group into an existing relation. As relations are created, the map of input attributes to physical attributes is updated accordingly.

The algorithm creates new relations if at least one of two conditions is satisfied. First, if the current attribute group corresponds to the root of a merged subtree, phase 3 always creates a new relation. This ensures that phase 3 can replace the original subtrees that were merged together with foreign key references into the newly created relation. Second, phase 3 creates a new relation if the cardinality of the current attribute group is at least ϵ% less than the cardinality of the relation it would otherwise be condensed into (the group's nearest ancestor in the tree for which a relation has already been created). By default we set ϵ% to 50%, but when a column-store is targeted, a larger ϵ is recommended. We choose to create relations for groups that correspond to significant cardinality shifts because the shifts usually signal the presence of a separate semantic entity. Even when they do not represent a new kind of entity, they represent a significant opportunity for normalization since they will reduce the size of the dataset if materialized as their own relation. This normalization and resulting data size reduction will generally improve row-store performance. However, column-stores usually perform better on denormalized data — this is why they benefit from a larger ϵ. Even though high ϵ values will reduce the number of entities that are revealed via the creation of separate tables, those entities that are hidden by a large value of ϵ can still be presented to the user separately from the final table schema.

If the attribute group under consideration does not meet either of these conditions, materializing the group as its own relation yields neither the benefits of entity identification nor significant redundancy elimination. The algorithm thus adds the attributes from the group to the relation corresponding to the group's nearest ancestor in the tree. This process concentrates the user's attention on the most important aspects of the dataset and may improve query performance by eliminating extra table joins.

Associations: The algorithm stitches together the created relations with primary-to-foreign key links. While the parent attributes would ideally be primary keys for their corresponding relations, this is often infeasible because parents are only required to be *approximately* unique within their relations. Instead, the algorithm adds an integer "id" attribute to each relation and designates "id" as the relation's primary key. Associations between relations are materialized as foreign keys from relations higher in phase 3's output tree to their children. By placing foreign keys in this way, phase 3 ensures that each relation can be deduplicated (if foreign keys were created in the opposite direction, every relation would have the same cardinality as the root relation).

Data loading: Each input record is loaded into the phase 3 schema by creating rows in phase 3's output relations according to the mapping of input attributes to physical attributes. Once all input records have been transformed, the phase 3 relations are deduplicated to reduce redundancy.

4. EXPERIMENTAL EVALUATION

We evaluate our algorithm with three real world datasets:

Dataset #1 — Flights: Our first dataset was collected by the US Bureau of Transportation Statistics and reports the timeliness of every non-stop flight within the United States. Each record corresponds to a single top-level flight entity and contains 64 attributes that collectively describe the flight's scheduled itinerary, any delays it experienced, and its affiliated airline. Within these flight objects, the dataset nests origin and destination airport entities.

Our training set includes 1.3 million flight records from the first months of 2014. The data is originally encoded in the CSV data format and, consequently, requires none of the preprocessing steps outlined in Appendix A.

Dataset #2 — Twitter: Our second dataset consists of 10 million records scraped from the JSON API of the social network Twitter, which allows its users to exchange messages called "tweets." Each record models a single top-level tweet entity and contains a short message along with more than 300 attributes of metadata. This metadata includes a user entity that identifies the author of the tweet and a place entity that specifies where the message was posted from.

10% of the experimental dataset models a special kind of tweet that users create by reposting a message that someone

[5]We simplify our implementation by assuming that subtree matches always behave transitively. While we expect this behavior to hold in most cases and observed no exceptions in our experiments, non-transitive matches are possible.

else previously authored. These special tweets, referred to as "retweets," are distinguishable because they embed the original tweet that they are derived from (this models the n-to-one relationship from retweets to original tweets).

This dataset contains arrays of nested objects. As described in Appendix A, our algorithm treats nested-object arrays as independent sub-instances of the original schema generation problem. In order to simplify our analysis of the Twitter results, we only focus on the first iteration of our algorithm, since this iteration contains the most important semantic entities (tweets, retweets, users, and places) and over 100 attributes on most records, while the subsequent sub-instances cover more fringe attributes and entities.

Dataset #3 — GitHub: Our third dataset was scraped from GitHub, which is a website allowing developers to collaborate on software projects managed by the Git version control system. At the top-level, records from this dataset contain a pull request entity which models a user's request to merge code across branches or different repositories. These pull request entities reference 7 embedded user entities, head and base commit entities (i.e. the new commit and the commit it's to be merged on top of), and head and base repository entities. Our training set contains 1M records scraped from GitHub's JSON API by the GHTorrent project [11].

Methodology: We have manually analyzed the raw data and documentation associated with each of the three datasets in order to gain insight into the semantics of their attributes and the relationships among their attributes. In Sections 4.1-4.3, we evaluate our algorithm's performance by comparing the output of each phase with the expectations developed during this analysis. In Section 4.4, we compare our algorithm with Argo [4] and an XML transformation algorithm from Shanmugasundaram et al. [23].

Due to the large number of attributes in all of these datasets, it is impossible to fit the full schema produced by our algorithm in the main part of this paper. However, Appendix F shows the final schema for the GitHub dataset.

Performance: Given that our algorithm is intended to be run once per dataset in an **offline** setting, we did not performance-optimize our implementation and do not present performance metrics in the body of the paper. In Appendix H, we provide details about our implementation, analyze its runtime, and describe the trade-offs of sampling.

4.1 Phase 1 results

Phase 1 produces attribute trees with many attribute groups for each of the datasets, with the Flights tree having the fewest at 22 and the GitHub tree having the most at 28.

Entities and stable attributes: As described in Section 3.2.1, each entity has a unique identifier and a set of descriptive attributes that are either stable or variable across repeated references to a specific instance of the entity. We expect entities to manifest in the phase 1 attribute tree as subtrees that begin with the entity's unique identifier and contain the entity's stable attributes. The phase 1 output for each of the experimental datasets generally shows this pattern. To illustrate, we consider the Flight dataset's origin airport entity which has the unique identifier "origin_airport_seq_id" and has descriptive attributes that specify the airport's short-code (e.g. "JFK") along with its location. These descriptive attributes are relatively stable across repeated references to the same airport and, consequently, form a subtree with the parent "origin_airport_seq_id." The subtree contains only attributes related to the origin airport.

In most cases, the entities contained within the experimental datasets behave similarly to this example and correspond to a single subtree in the phase 1 graph; the user entity from the Twitter dataset, however, is split across two subtrees. The first has "user_id" as its parent attribute and contains the user's basic information (e.g. screen name). The second has "user_profile_image_url" as its parent and contains attributes related to the visual design of the user's profile (e.g. text color and image URLs). This division occurs as a result of phase 1's strategy for assigning parent attributes: when choosing the parents for most of the user attributes, phase 1 finds that both the ID and URL are possible parents at a depth of one in the attribute tree; faced with a tie in the depth heuristic, phase 1 assigns each of the user attributes to the parent with which the attribute demonstrates the strongest functional relationship. The attributes for the user's basic information are most strongly dependent upon "user_id" and are correspondingly placed in that subtree; the visual attributes are most strongly dependent upon "user_profile_image_url" and are placed accordingly. This split suggests that the visual attributes associated with users behave as their own entity. Each *version* of a user's profile corresponds to an instance of this entity whose unique identifier is "user_profile_image_url" and whose descriptive attributes include the remaining visual attributes. Since these versions are neither unique to nor constant for a specific user, the algorithm produces two distinct subtrees that both attach to the attribute tree at the same depth.

The creation of this separate visual entity is another example of our phase 1 algorithm identifying semantic information missed during our manual analysis of the dataset. It is important to note that this semantic information is also absent in the original nesting of the JSON documents that comprise our training set, as those documents combine the user's basic information and visual attributes together in a single object. This example demonstrates the pitfalls of creating tables that exactly mimic the structures of the input data: even the designers of the input documents may be indifferent to or unaware of some important semantic details.

Entities and unstable attributes: Many entities from the experimental datasets contain unstable attributes that do not tend to retain their values across multiple references to the same instance of an entity. Due to the absence of functional relationships from an entity's unique identifier to its variable attributes, we expect unstable attributes to appear outside of their entity's subtree in the phase 1 output.

We observe this behavior in both the GitHub and Twitter datasets, which include several snapshot-in-time measurements. For example, GitHub's base repository entity includes "base_repo_forks_count" to track the number of times the base repository was forked, and Twitter's user entity includes "user_favourites_count" to track the number of tweets the user has saved to their favorites. These counting attributes vary over the lifetime of their associated entities and their values are only accurate as of the moment they were scraped from the GitHub and Twitter APIs. They consequently do not participate in functional dependencies and are placed in the root of their respective attribute trees.

There are, however, some exceptions, and in some cases these unstable attributes are placed in deeper attribute groups. For example, phase 1 places GitHub's unstable "head_user_

	Flights	*Twitter*	*GitHub*
# FDs from Formula 3.1.1	267	2497	3728
# FDs from Formula 3.1.2	112	1180	2603
Avg. skew of FD allowed by 3.1.1 but not by 3.1.2	97%	95%	91%

Table 1: Comparison of Formulas 3.1.1 and 3.1.2.

site_admin" attribute within the "head_repo_pushed_at" subtree. At first glance, there is no semantic justification for this placement. However, both attributes exhibit a hidden dependency on the record creation time, which creates a subtle, but strong, relationship between them. Since unstable attributes tend to be time dependent, phase 1 often places them within the subtrees of other time dependent attributes.

Skew performance: All three experimental datasets include attributes skewed toward a single value. As we described in Section 3.1.2, we expect these skewed attributes to frequently exhibit high values of Ilyas et al.'s strength formula (Formula 3.1.1) even when paired with semantically unrelated attributes in the dataset. We further expect that the modified strength formula we presented (Formula 3.1.2) will exclude these false pairings from the phase 1 output.

We observe this expected behavior across all three datasets. As shown in Table 1, Formula 3.1.2 excludes 30% to 60% of the column pairs that Formula 3.1.1 identifies as potential functional dependencies. These excluded column pairs tend to have dependent attributes that are highly skewed toward their modal value. On average, the dependent attributes from the excluded column pairs have the same value on 91% to 97% of all records in the dataset.

As a concrete example of the difference between the two strength formulas, consider the Flights dataset's "cancelled" and "cancellation_code" attributes. Both are NULL for the 95% of records that represent flights that were not canceled; for the remaining 5% of records, "canceled" contains the value 1.0 and "cancellation_code" contains one of the letters A though C. Both exhibit high values of Formula 3.1.1 with at least 10 other attributes in the dataset, the majority of which do not correspond to semantic relationships in the Flights domain model. However, these semantically unfounded relationships have near-zero values of our Formula 3.1.2 and the algorithm includes only the dependency from "cancellation_code" to "cancelled" in its output.

Attribute merging: As outlined in Section 3.1.4, we expect the algorithm to merge together attributes that contain the same information content in order to simplify its longest path computations. We observe correct attribute merges across all three datasets and observe no invalid merges.

A particularly interesting example of attribute merging occurs in the GitHub dataset where a pair of user entities are entirely merged together. The "base_user" and the "base_repo_owner" entities refer to the same user instance across every record in the dataset. As a result, their constituent fields always contain exactly the same value and are merged together. The two users are thus represented in the attribute tree by a single subtree of merged attributes.

4.2 Phase 2 results

Subtree matching: Our manual analysis of the experimental datasets identified several pairs of entities that had similar descriptive attributes and had good semantic basis to be matched and combined: Within the Flights dataset, the origin and destination airports should be matched. Within the Twitter dataset, the embedded retweet entity is really just a tweet, and should be matched with the top-level tweet entity. Similarly, the top-level user should match with the retweet's user, and the top-level place should match with the retweet's place. Within the GitHub dataset, there should be matches among all pairings of the 7 kinds of user entities, the 2 repository entities, and the 2 commit entities.

These expectations are summarized in Table 2's second column, which lists the number of matching entity pairs we manually identified in each dataset. The table's "Our algorithm" section compares phase 2's performance against those counts. It shows that phase 2 identifies all but one of the expected entity matches and also identifies semantically meaningful entity matches that we did not manually predict.

Phase 2 ignores the predicted entity match between the GitHub commit entities. We predicted this match because we noticed that the commits' unique identifiers — "head_sha" and "base_sha" — overlap and we assumed that they were each associated with a set of stable descriptive attributes describing associated user and repository entities. Our assumptions hold true for the head commit, which corresponds to a subtree that has "head_sha" as its parent and that embeds the head repository and head user entities. The base commits, however, are not stably associated with specific users or repositories. As a result, the base user and base repository attributes demonstrate weak functional relationships with "base_sha" and are not placed within the base commit's subtree. This leads the two commit entities to have dissimilar subtrees that do not match in phase 2. Given that the commits do not behave as we initially assumed, the phase 2 algorithm's output is a slightly more accurate reflection of the underlying data than our manual analysis. This is another example of our manual analysis being too sensitive to the attributes' original JSON nesting.

In addition to the 26 matches we anticipated during our analysis, seven matches were identified that we did not expect. For example, the algorithm reveals that Twitter's retweet entity contains an additional partial user to model whether the tweet is in reply to another member of the network. This partial user has only the fields "retweeted_status_in_reply_to_user_id," "retweeted_status_in_reply_to_user_id_str," and "retweeted_status_in_reply_to_user_name." Despite this sparsity of descriptive attributes, phase 1 allocates the partial entity its own subtree which matches with both the retweeted user and the top-level user during phase 2. These unexpected matches accurately reflect overlapping entities. Given that the in-reply-to user does not exist as its own object in the JSON input, this example represents another instance of our algorithm discovering semantic entities absent from the initial structure of the dataset.

There are no erroneous phase 2 matches found for any of the experimental data sets. All identified matches represent subtrees containing overlapping entities.

Attribute matching: Within each pair of matched entities, phase 2 highlights the pairs of semantically equivalent attributes that the two entities contain. In order to gauge phase 2's attribute matching performance, we consider three measures for each entity pair: 1) the number of attribute matches identified across the two entities, 2) the number of these attribute matches that accurately capture semantic equivalence, and 3) the number of these attribute matches do not accurately capture semantic equivalence and should

	# entity matches expected	Our algorithm				Shanmugasundaram et al.'s Algorithm			
		# expected found	# expected missing	# unexpected found	% unexpected meaningful	# expected found	# expected missing	# unexpected found	% unexpected meaningful
Flights	1	1	0	2	100%	0	1	0	N/A
GitHub	23	22	1	1	100%	5	18	0	N/A
Twitter	3	3	0	4	100%	2	1	0	N/A

Table 2: Summary of the entity matches found by both our algorithm and Shanmugasundaram et al.'s algorithm. Unexpected matches are considered "meaningful" if they have semantic underpinnings and correctly reflect overlapping domains.

	Flights	*Twitter*	*GitHub*
# attribute pairs	15	47	79
# correct attribute pairs	15	46	79
# incorrect attribute pairs	0	1	0
# missing attribute pairs	0	15	11

Table 3: Summary of attribute pairs found in each dataset.

not have been identified. The first three rows of Table 3 show the sum of these performance measures across all of the entity pairs found in each dataset. The numbers reflect that phase 2 correctly identifies equivalent attribute pairs in every dataset. Of the 141 attribute matches made across all three datasets, phase 2 only makes one incorrect match (it incorrectly matches two of Twitter's boolean fields).

While phase 2 rarely matches attributes incorrectly, it sometimes omits matches between semantically related attributes. We consider phase 2 to have missed a relevant attribute pair when an entity match contains an orphan attribute that has a semantically equivalent attribute elsewhere in the tree. The final row of Table 3 shows the number of these missing attribute pairs from each of the experimental datasets. In total there were 26 attributes across the datasets that could have been matched but were not. Of these 26, 24 were due to the fact that phase 2 only searches for matching attributes within matching subtrees of the attribute tree we construct in phase 1. In these 24 cases, the matching attribute was located outside of the relevant subtree being searched. This was usually due to the attribute being unstable, and phase 1 placing the attribute in the root instead of a particular subtree. The two remaining missed matches were a side-effect of the one incorrect match in the Twitter dataset — the two attributes that were incorrectly matched were not available for their correct partners.

4.3 Phase 3 results

Schema generation: For all three datasets, phase 3 creates a relational schema that contains a root relation that includes columns for (1) descriptive attributes of the top-level entity, (2) unstable attributes associated with embedded entities, and (3) foreign keys linking to the other relations in the final schema. As expected, the number of relations in the phase 3 schemas is significantly lower (50%-65%) than the number of phase 1 attribute groups. The Flights schema contains 10 relations, the Twitter schema contains 8, and the GitHub schema contains 8. For each of the three datasets, 4 of these relations result from phase 2 entity matches. The other relations result from large cardinality changes.

Phase 3's assumption that large drops in cardinality may signal the presence of an embedded entity frequently holds across all three datasets. For example, phase 3 detects GitHub's milestone entities due a large cardinality drop. In other cases, however, the relations due to cardinality are less semantically significant and instead represent opportunities for normalization. For example, the GitHub schema includes a relation off of the root that contains four semantically unrelated attributes that phase 3 creates because it has only 328K unique rows, while the root has 1M rows.

4.4 Comparison to Alternative Approaches

We compared our approach to two alternative approaches for mapping nested key-value data to relational schemas:

(1) Argo is significantly simpler than our algorithm because it does not perform entity identification and instead proposes a generic relational encoding of arbitrary JSON documents [4]. As described in Section 2, Argo stores documents in ternary relations that contain rows for every key in every document (each row stores the document ID, the key's name, and the key's value). Argo preserves type information by creating three ternary relations differentiated by the type of their "value" columns (boolean, text, and doubles).

(2) Shanmugasundaram et al. describe the shared inlining algorithm which generates a relational schema by analyzing a graph of the XML DTD associated with its input[6]. The DTD graph contains nodes for all elements in the input DTD and represents the nesting relationships between those elements with edges. The algorithm creates a relation for all nodes in the DTD graph with an in-degree of 0 (these are accessible only if made into relations) or with an in-degree greater than 1 (these are elements that are "shared" among multiple parent elements). Within our datasets, the algorithm inlines all other nodes into their nearest ancestors.

Note that the shared inlining approach requires XML DTDs, but none of our experimental datasets originally include this schema information. For the JSON-encoded GitHub and Twitter datasets, we generated DTDs with elements corresponding to the original nesting structure of the documents. For the CSV-encoded Flights dataset, we generated a DTD with a single element that contains all attributes.

Entity matching: Table 2 compares the number of entity matches identified by our algorithm and the number of matches identified by the shared inlining algorithm. We omit Argo from this part of the comparison, since it does not perform entity identification or matching.

While our algorithm identifies all three expected entity matches within the Twitter dataset, the shared inlining algorithm only matches the top-level user to the retweeted user and the top-level place to the retweeted place. The match between the top-level user and retweeted-user is identified because the two entities both appear beneath a "user" key in the original dataset ("user" and "retweeted_status.user" respectively). The generated DTD represents these keys

[6]We do not discuss their additional hybrid inlining algorithm because it produces a single relation for our datasets.

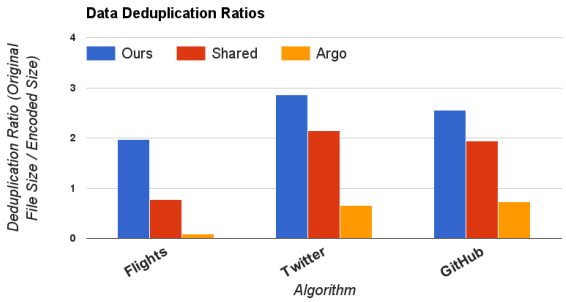

Figure 6: Deduplication ratios by algorithm by dataset.

with a single <user> element that may be nested within either a <root> element or a <retweeted_status> element. Since <user> may be nested within multiple elements, its in-degree in the DTD graph is greater than 1 and the shared inlining algorithm creates a separate relation to model the matched user entities. Similarly, the top-level place and retweeted place occur beneath a common key, enabling the shared inlining algorithm to merge them into one relation.

Since the top-level tweet and the retweeted-status do not appear beneath a common key in the input dataset, the shared inlining algorithm is not sensitive to their semantic relationship and cannot identify this expected entity match. Moreover, the shared inlining algorithm cannot detect any of the unexpected entity matches discovered by our algorithm because those matches involve groupings of input attributes that are not captured by the original JSON nesting.

The shared inlining algorithm similarly omits entities from GitHub dataset. The algorithm only successfully identifies 5 of the 23 expected entity matches — one match from the head repository to the base repository and four among pairings of the user entities. Only the user entities that are nested beneath the same key within the original JSON document are matched by the shared inlining algorithm. Since the seven user entities present in the dataset appear beneath four distinct keys ("milestone_creator," "user," "owner," and "assignee"), the shared inlining algorithm cannot identify the remaining 17 matches that occur among the dataset's users.

The shared inlining algorithm is unable to detect any entity matches from the flat Flights DTD.

Data deduplication: As described above, an advantage of converting nested data into relational data is the deduplication of records that are in multiple nested locations. Thus, we loaded each of the experimental datasets into the three proposed schemas and measured each schema's data size reduction ratio (the size of the input file divided by the size of the schema's relations). Figure 6 shows these results.

The Argo encoding carries significant storage overhead because it creates a large number of tuples (1 per key per document) that each require a 22-byte tuple header, an 8-byte document identifier ("objid"), and a variable length string to store key names. As a result of this storage overhead, the Argo encoding expands all of the input datasets.

The shared inlining algorithm exhibits better storage performance than Argo because it creates many fewer tuples (1 to 4 per document, depending on the dataset) and because it stores key names as column names. This latter optimization allows the algorithm to reduce the data size by roughly 2x for the Twitter and GitHub datasets. However, the Flights dataset does not reduce in size because its original CSV encoding already avoided redundant storage of key names. The shared inlining algorithm instead causes the Flights dataset to expand as a result of tuple headers and document ids.

Our algorithm achieves significantly more data size reduction for all three experimental datasets. Similar to the shared inlining algorithm, our algorithm stores key-names as column names which dramatically reduces the size of the Twitter and GitHub datasets. However, our algorithm achieves more data reduction for these datasets and for the Flights dataset because it uses foreign-key references to eliminate the duplication inherent in the entity matches it finds.

Query performance: In Appendix G, we analyze the runtimes of 5 queries to further compare and contrast these three approaches to schema generation.

5. FUTURE WORK

Our algorithm has two limitations which represent opportunities for future work. First, our algorithm does not support functional dependencies with multiple attributes on the left-hand side (e.g. functional dependencies in which one attribute depends on 2+ attributes). To support this class of dependency, our algorithm requires three extensions:

Dependency enumeration: Since it is computationally intractable to evaluate Condition 3.1.1 for all possible functional dependencies when multiple LHS attributes are allowed, a more intelligent exploration of the search space will be required. Flach et al. present a promising solution to this exploration problem which we could leverage [8].

Attribute tree construction: Our attribute tree construction algorithm creates a directed graph from the mined functional dependencies. If functional dependencies with multiple LHS attributes are allowed, we must either devise a strategy for modeling these attributes as merged nodes that are compatible with our existing algorithm or must extend the algorithm to handle the resulting hyper-graphs directly.

Entity matching: The entity matching algorithm identifies matches by joining subtrees on their parent attributes. Since subtrees will potentially have many parents when multiple LHS attributes are allowed, a new process for joining subtrees will be required. The naive approach simply joins subtrees on all combinations of their parent attributes; since this could greatly bloat the space of subtree matches, investigation of more efficient strategies will likely be needed.

The second limitation of our work is that we entirely ignore all structural information associated with our input datasets. While we contend that structural information is often misleading or otherwise unreliable, our approach of completely disregarding structural cues is heavy-handed. We expect that future research into a hybrid solution that integrates our data-driven approach with structural cues will yield higher quality schemas. We further expect that structural cues can also be leveraged to more intelligently explore the functional dependency search space.

6. CONCLUSION

We have presented an algorithm that identifies the structures implicit in semistructured datasets and materializes those structures as relational schemas. Our experiments show that our algorithm generates reasonable schemas for datasets from disparate domains, irrespective of their input format. While our algorithm's schemas often contradicted our expectations, the algorithm's schemas captured the patterns of the input data better than our manual analysis.

Acknowledgments This work was sponsored by the NSF under grant IIS-1527118. We thank Wenbo Tao, Lambros Flokas, and the anonymous SIGMOD 2016 reviewers for their insightful feedback on earlier versions of this manuscript.
Downloads Our experimental datasets are available at http://s3.amazonaws.com/discala-abadi-2016/index.html.

7. REFERENCES

[1] S. Bell. Dependency mining in relational databases. In *Qualitative and Quantitative Practical Reasoning*. 1997.
[2] D. Bitton, J. Millman, and S. Torgersen. A feasibility and performance study of dependency inference [database design]. In *Proc. of ICDE*, 1989.
[3] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From xml schema to relations: a cost-based approach to xml storage. In *Proc. of ICDE*, 2002.
[4] C. Chasseur, Y. Li, and J. M. Patel. Enabling json document stores in relational systems. In *WebDB*, 2013.
[5] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with stored. In *SIGMOD*, 1999.
[6] F. Du, S. Amer-Yahia, and J. Freire. Shrex: managing xml documents in relational databases. In *VLDB*, 2004.
[7] R. Fagin, A. O. Mendelzon, and J. D. Ullman. A simplied universal relation assumption and its properties. *ACM Trans. Database Syst.*, Sept. 1982.
[8] P. A. Flach and I. Savnik. Database dependency discovery: A machine learning approach. *AI Commun.*, 12(3), 1999.
[9] D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing xml data in a relational database. In *Inria Research Report*, 1999.
[10] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. Xtract: A system for extracting document type descriptors from xml documents. In *SIGMOD*, 2000.
[11] G. Gousios. The ghtorrent dataset and tool suite. In *Conference on Mining Software Repositories*, 2013.
[12] O. Hassanzadeh, S. H. Yeganeh, and R. J. Miller. Linking semistructured data on the web. In *WebDB*, 2011.
[13] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 1999.
[14] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, 2004.
[15] J. Kivinen and H. Mannila. Approximate inference of functional dependencies from relations. *Theoretical Computer Science*, pages 129 – 149, 1995.
[16] W.-S. Li and C. Clifton. Semint: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data & Knowledge Engineering*, 2000.
[17] D. Maier, J. D. Ullman, and M. Y. Vardi. On the Foundations of the Universal Relation Model. *ACM Trans. Database Syst.*, 9(2):283–308, June 1984.
[18] H. Mannila and K.-J. Räihä. Algorithms for inferring functional dependencies from relations. *DKE*, 1994.
[19] P. Minh-Duc, P. Linnea, E. Orri, and P. Boncz. Deriving an emergent relational schema from rdf data. In *WWW*, 2015.
[20] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDBJ*, 10(4):334–350, 2001.
[21] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of xml documents. In *WebDB*, pages 47–52, 2000.
[22] J. Shanmugasundaram, E. Shekita, J. Kiernan, R. Krishnamurthy, E. Viglas, J. Naughton, and I. Tatarinov. A general technique for querying xml documents using a relational database system. *SIGMOD Rec.*, pages 20–26, 2001.
[23] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying xml documents: Limitations and opportunities. pages 302–314, 1999.
[24] D. Tahara, T. Diamond, and D. J. Abadi. Sinew: A sql system for multi-structured data. In *SIGMOD*, 2014.
[25] K. Wang and H. Liu. Schema discovery for semistructured data. In *KDD*, 1997.
[26] K. Wang and H. Liu. Discovering typical structures of documents: A road map approach. In *SIGIR*, 1998.
[27] H. Zhang and F. W. Tompa. Querying xml documents by dynamic shredding. In *DocEng*, 2004.

APPENDIX

A. FLATTENING INPUT DATA

As described above, input datasets must be flattened before phase 1. We accomplish this with three transformations:

Objects nested within objects: Objects nested inside of one another are flattened by creating a column for each of the leaf values contained in the objects. Each column is labeled with the concatenation of the attribute names that appear along the leaf's corresponding nesting path. Label collisions are resolved by appending the depth of the leaf within the original document to the concatenated name.

Arrays of scalar values: Scalar arrays (e.g. arrays of integers) are casted to strings for the algorithm's computations. When loaded into the final schema, the arrays are stored instead using the target database's preferred storage scheme (e.g. 1st-class array types or separate tables).

Arrays of nested objects: Objects containing arrays of other objects require the most involved transformations because they model either one-to-many or many-to-many associations, but our algorithm searches for one-to-one or many-to-one relationships. In order to overcome this incompatibility, each nested array can be treated as an independent sub-instance of the original schema generation problem. If the array models a one-to-many relationship, a foreign key is added from the sub-instance's root back to the schema for its parent instance. If the array models a many-to-many relationship, a linking object that joins the sub-instance to the schema for its parent instance must be introduced.

B. IMPROVING PHASE 2 PERFORMANCE

Although the join operation described in Section 3.2.2 effectively identifies record pairs with matching parent values, in many cases its computation will be expensive. In the worst case, when both subtrees' parent attributes contain a single shared value, the size of the join set is the square of the number of records in the training set. To reduce the size of the join and the cost of the subsequent operations, we preprocess each subtree before performing any join operations. For each subtree, we compute a deduplicated table that contains the unique permutations of the subtree's values, along with the frequency of each permutation in the training set. The result of this process for the $A_{authEmail}$ subtree is:

authEmail	*authName*	*authINum*	*authIName*	*freq*
js@e.pfl	J. Snow	1	EPFL	2
dd@ca.uk	D. Duck	14	Cambridge	1

Instead of performing subtree joins as self joins on the original flat table, we perform joins between the subtrees' deduplicated tables. Since all other attributes within the subtree exhibit soft functional dependencies on the parent attributes, the parents will be nearly unique within these deduplicated tables. Consequently, the upper bound on the size

of the condensed join result is approximately the size of the training set and, in many cases, significantly smaller.

Every row in the condensed join set will contain a frequency column from each of the deduplicated tables. The product of these frequencies reflects the number of times that a row would repeat in the uncondensed join set. As a result, to compute the attribute matrices from the condensed join sets, we must modify phase 2 to increment cells by the product of the frequency columns, instead of by 1.

C. REDUNDANT PHASE 2 MATCHES

As described in Section 3.2.2, our matching algorithm attempts to match entire subtrees from the phase 1 attribute tree. In other words, it combines all attributes that descend from a particular node in the attribute tree into a single bucket, before attempting to match with a different bucket of attributes from a different subtree. As described in step 3 above, at least 50% of the attributes in the smaller bucket have to match with attributes in the larger bucket for the two subtrees to be considered a match. Therefore, once we discover that two subtrees are a match, additional matches will likely be found in subtrees of those subtrees.

On one hand, it is totally unnecessary to compare subtrees of matching subtrees. For example, as shown in Figure 2, once we have discovered that the subtree corresponding to "author" matches with the subtree corresponding to "reviewer" (shown in red), it is unsurprising that the nested institution information within the author and reviewer trees also matches (shown in purple). Checking the two institution subtrees therefore seems like a waste of effort.

We still check these sub-subtrees for a match because the inner match can reveal interesting insights into the relationship between the sub-subtrees and their parent subtrees. In particular, if the size of the join set between the two sub-subtrees is at least 50% larger than the join set between the two parent subtrees, we conclude that the entities represented by the sub-subtrees are semantically separate from the entities represented by their parent subtrees. We assume this because the join set's size can only increase significantly if either (1) some records from the training set do not join for the parent match but do join for the inner match or (2) the entity set represented by the inner match has significantly lower cardinality than the entity set represented by the parent match. Since both behaviors indicate the inner match behaves differently than the outer match, we show sub-subtree matches if the join set size increases sufficiently.

D. ATTRIBUTE TREE IMPROVEMENT

Due to the functional relationships entities typically exhibit, we expect phase 1 to isolate all of the stable attributes associated with an entity inside of a single subtree whose parent is the entity's unique identifier. Nevertheless, phase 1 sometimes produces suboptimal trees that place attributes outside of their entity's subtree. Although these erroneous placements do not reflect a correct semantic understanding of the data, they typically cannot be detected algorithmically because they do reflect the functional structures identified by phase 1. In the special case that these suboptimal placements occur within nested subtree matches, however, the phase 2 output provides a chance to correct these errors.

Matches between subtrees that begin at higher levels in the attribute tree cover more of the full tree and, consequently, are more likely to identify matching attribute pairs. Thus, if there is an orphaned attribute a_{orphan} in a subtree match, and the subtree match is nested within a match at a higher point in the tree, and the higher subtree match pairs a_{orphan} with another attribute, then there is likely an error within the original attribute tree. These errors are fixed by moving the attribute paired with a_{orphan} into the lower subtree.

E. ATTRIBUTE CLUSTERING

The algorithm above compares every pair of subtrees to see if there is a match, and during this comparison operation, compares every pair of attributes across these two subtrees. To reduce the space of both sets of comparisons, we analyze the domain of each attribute before we perform subtree matching, and cluster attributes into groups that contain similar values. The algorithm then only attempts to match subtrees whose parent attributes belong to the same cluster, as these are the only subtrees whose domains are likely to overlap. Similarly, when comparing two subtrees, phase 2 only considers pairs of attributes in same cluster.

We form these attribute groups according to a process inspired by Li and Clifton's clustering techniques [16]. For each text based attribute in the dataset we compute: 1) the maximum, minimum, variance, and average of the length of its values, 2) its proportion of numeric characters, 3) its proportion of punctuation characters, and 4) its proportion of whitespace characters. We use these statistics as the features for a bisecting k-means clustering instance. We choose K programmatically by computing clusterings with progressively increasing values of K and choosing the final clustering by analyzing the within-cluster-sum-of-squares:

$$\sum_{c \in clusters} \sum_{vector \in c} distance(vector, centroid(c))^2$$

We continue adding clusters until this function plateaus or increases from a previous value. Although lower K values limit the impact of this optimization, we favor lower values because they minimize the risk that matching attributes are erroneously separated into distinct clusters.

F. THE GITHUB SCHEMA

Interpreting the phase 3 diagrams: Figure 7 shows the phase 3 schema that our algorithm produces for the GitHub dataset. Each box in the diagram represents a relation in the final schema. For all relations:

- Attribute names that are written in upper-case letters are either primary or foreign keys inserted by phase 3.
- Foreign keys point to the relation whose header matches their name (e.g. the key "HR_ID" points to the relation with the header "HR").
- Relations joined by foreign key references are connected with lines (arrows point into the referenced relation).

For relations that do not participate in entity matches, a one-to-one relationship exists from input attributes to physical attributes in the phase 3 schema (these are the single-column boxes in the diagram). Each lower-cased cell in these relations represents an input attribute that phase 3 included directly in its physical schema. Consider the "MS1" relation in the diagram: "ms_title" is an input attribute mapped to a physical attribute of the same name.

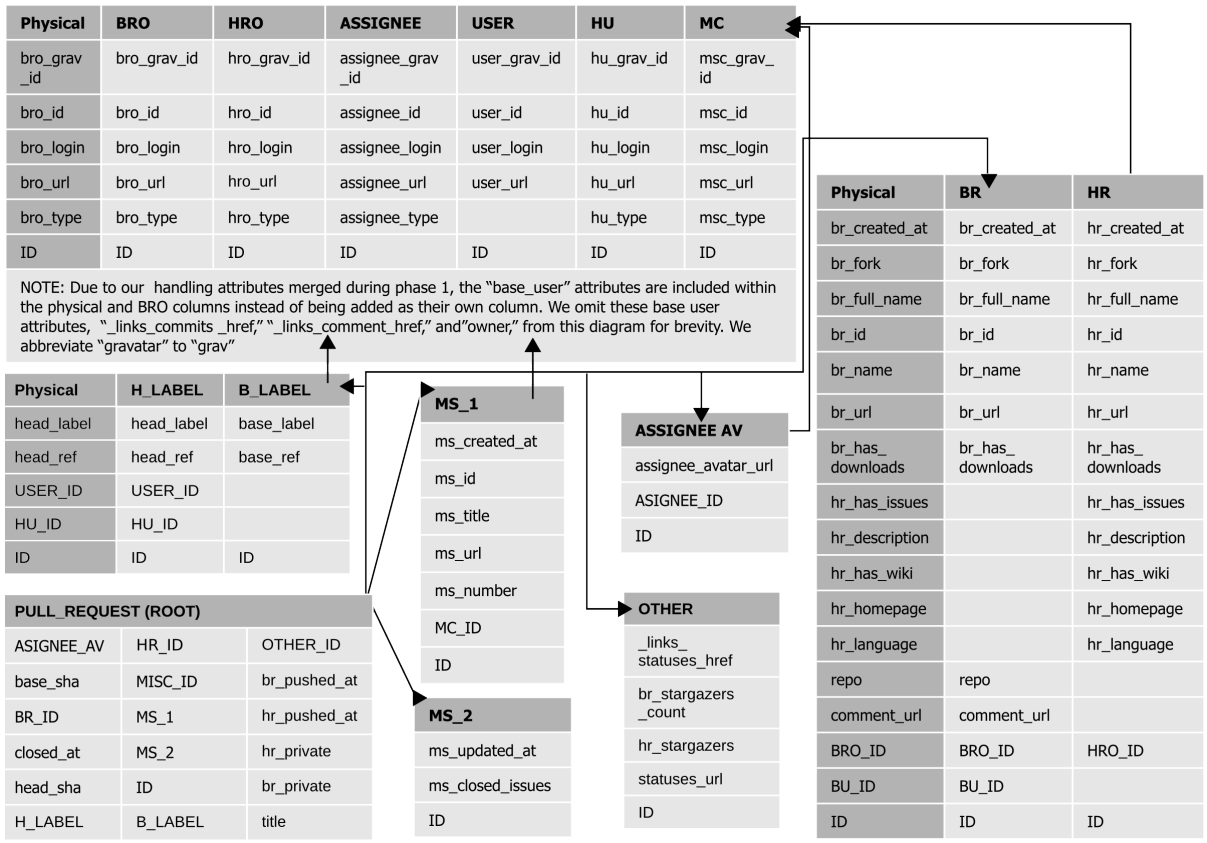

Figure 7: The phase 3 schema for the GitHub dataset. For the ROOT relation, we omit many attributes due to lack of space. We use the following abbreviations: "BR" is "base_repo," "BRO" is "base_repo_owner," "HR" is "head_repo," "HRO" is "head_repo_owner," "MS" is "milestone," "MC" is "milestone_creator," "BU" is "base_user" and "HU" is "head_user."

For relations that do participate in phase 2 entity matches, an n-to-one relationship exists from input attributes to physical attributes in the phase 3 schema (these are the multi-column boxes in the diagram). These relations include a dark gray column labeled "physical" that lists the names of the physical attributes in the phase 3 schema. Each additional column represents a distinct attribute group that was merged onto the relation. These columns show the mapping from that groups' input attributes to the physical schema: each attribute contained within the group appears on the same row as the physical attribute that it was mapped onto. Consider the relation containing "BR" (base_repo) and "HR" (head_repo) at the right of the figure: the first row indicates that input attributes "br_created_at" and "hr_created_at" both map onto the physical attribute "br_created_at."

Phase 1 attribute trees: Although we lack sufficient space to include the complete phase 1 attribute tree, the diagram shows nearly all of the original attribute groups because the phase 1 tree forms the basis of the phase 3 schema.

Phase 2 matches: The schema diagram shows all of GitHub's entity and attribute matches. Note that a single entity match may be represented by multiple relations if the matched entities contained several attribute groups.

G. QUERY PERFORMANCE COMPARISON

In Table 4, we present the runtimes of 5 queries over our algorithm's schema, the shared inlining schema, and Argo's schema. While the queries do not comprise a complete benchmark, they offer a view into the kinds of queries that may benefit from our algorithm's schema. All measurements represent cold-cache runtimes averaged over three runs. The queries were run on PostgreSQL on an Amazon EC2 instance with 8 vCPUs and 61 GiB of memory (this machine also was used for the performance evaluation in Appendix H).

Query #	*Our Algorithm*	*Shared-inlining*	*Argo*
1	0.7	28.7	559.1
2	32.7	95.1	1,635
3	1.9	5.6	6.7
4	25.2	37.1	36.9
5	23.9	16.3	35.5

Table 4: Runtimes in seconds for Appendix G's queries.

Query #1: This query projects a list of unique (user_id, user_login) pairs from the GitHub dataset, including pairs from all 7 kinds of user entities. This query requires only a sequential scan when expressed over our algorithm's schema because it merges all 7 users into a single relation. In contrast it must be expressed as the UNION of 4 queries in the shared inlining schema and as the UNION of 7 queries in Argo's schema. As a result, when run over our schema the query is 41x faster than it is over the shared inlining schema and 810x faster than it is over the Argo schema.

Query #2: This query identifies the average age of each Twitter user (current time - creation time). Argo's schema requires UNIONs and JOINs to express this query, which

causes the query's runtime over Argo's schema to exceed the other schemas' runtimes by more than an order of magnitude. In contrast, both our algorithm's schema and the shared inlining schema store Twitter users as a single relation and implement this query as an aggregate over that single relation. Despite this similarity, on our schema this query runs 2.9x faster than on the shared inlining schema. This disparity exists because our schema contains only 5.9M user records while the shared inlining schema contains 20M user records. Since the shared inlining schema does not deduplicate its merged entities, its query performance usually benefits less from the entities it identifies than our algorithm.

Query #3: This query calculates the average flight time for all flights departing from "Anchorage, AK". Since the Flights dataset is significantly smaller when encoded in our schema than when encoded in either the Argo or shared inlining schemas, this query runs fastest over our schema (2.9x faster than shared inlining and 3.5x faster than Argo).

Query #4: This query calculates the number of pull requests and the number of unique pull requests authors per repository in the GitHub dataset. Since the query accesses many different entities in the dataset, all three schemas must perform JOINS and must read a large subset of the dataset from disk in order to answer this query. As a result, the runtime is relatively even across all of the schemas (our schema is only 1.4x faster than the other two schemas for this query).

Query #5: This query calculates the number of unique pull-request authors per repository in the GitHub dataset. This query demonstrates an advantage of the shared inlining schema over our algorithm's schema: since shared inlining includes the document ID on all records, repositories can be joined to authors without accessing the root pull-request data. In contrast, our algorithm must join the repository and user entities through the root relation. As a result, our schema is 1.4x slower than the shared inlining schema (although our schema still is 1.5x faster than Argo's).

H. IMPLEMENTATION & PERFORMANCE

Implementation: Given that performance is not a focus of this paper, we did not build a production-grade, highly optimized implementation of our algorithm. We instead implement the algorithm as a Ruby application which delegates as much computation as possible to a Postgres server. The Ruby application loads the input dataset into a single Postgres relation and then queries this relation for the values it requires (the attribute cardinalities required by phase 1 are performed via COUNT DISTINCT queries, and the subtree matching required by phase 2 is done via JOINS). These queries are performed in parallel to improve performance.

The decision to push this logic into Postgres simplifies our software, but couples our implementation to the peculiarities of the Postgres query optimizer. While the optimizer often chooses performant execution plans, on some inputs it selects plans that are significantly slower than equivalent alternatives. In particular, the optimizer struggles with queries involving strings with long common prefixes (e.g. URLs on the same domain or Twitter messages with identical content). When the optimizer chooses to sort the dataset to implement a COUNT DISTINCT or JOIN on these string attributes with common prefixes, the time required to compare the common prefixes becomes significant. In some cases we observe an order of magnitude slowdown compared to the same operation implemented with a hashing strategy.

Performance: For the Twitter dataset, which was the largest of the three we considered, phase 1 required 31 hours to complete, phase 2 required 3.7 hours, and phase 3 required 3.61 minutes. Although these times collectively seem high, we reiterate that they reflect only an upper bound on runtime since we did not thoroughly optimize our implementation. Many opportunities for optimization exist, including 1) eliminating comparisons of attribute pairs in phase 1 when functional relationships can be inferred from their transitive properties and 2) applying a hashing strategy to reduce time spent comparing frequently-occurring string prefixes.

Even without these implementation improvements, our algorithm's runtime can be reduced by sampling the input dataset. In order to explore sampling's feasibility, we ran phases 1 and 2 over a random subset of 1 million Twitter records (10% of the full dataset). Since phase 3's runtime was already short relative to the other phases, we continued with the full dataset in this stage. Over this random sample, phase 1 required 2.7 hours and phase 2 required 5.1 minutes.

The phase 1 output from the sampled data is generally similar to the phase 1 output from the entire dataset. Within specific subtrees, there is some shifting of parent attributes and some movement of attributes deeper into the same tree (downward movement is common because the sampled dataset includes less entity repetition and it is thus easier for functional dependencies to achieve a high strength value). Since these modifications occur within the same subtree, they do not affect phase 2 attribute matches and generally do not cause semantically important shifts in the final schema.

There are, however, 6 attributes that switch subtrees in the sampled attribute tree and no longer descend from their original parents. These movements negatively impact the semantic meaning of the attribute tree and interfere with subsequent matching in phase 2. Specifically, these changed attributes cause the sampled phase 2 output to exclude 3 of the 4 unexpected entity matches that were originally identified. These matches are between subtrees containing 2-3 attributes each. They are excluded because their constituent attributes do not form subtrees in the sampled attribute tree and are instead merged together. Despite the omission of these entity matches, we still believe that sampling is a viable strategy since phase 2 still identifies all 3 expected entity matches and 1 of the 4 unexpected matches.

The sampled phase 2 output additionally includes two matches that were not identified from the full dataset. The first match occurs between two small subtrees containing skewed boolean attributes. While some of the booleans are semantically related, the overall entity-match is not semantically meaningful and should not have been identified. The second match occurs between a subtree rooted at "retweeted_status_in_reply_to_status_id" and the root relation. Since both parent attributes contain Tweet IDs, their domains overlap frequently. However, this match also lacks semantic meaning because the attributes beneath the in-reply-to ID do not form a coherent entity and are instead introduced due to noise in the phase 1 tree. These erroneous matches introduce some noise into the phase 2 output, but only affect a few fringe attributes in the final schema.

While we recommend that users run our algorithm over their full datasets for the best results, we find that especially time-sensitive users can trade some output quality for time-savings by running our algorithm over a random sample of their data.