Automatic Generation of Normalized Relational Schemas from Unannotated Data

Michael DiScala
Yale University
michael.discala@yale.edu

Categories and Subject Descriptors
H.2.4 [Database Management]: Relational Databases

General Terms
Systems, Data mining

Keywords
RDBMS, NoSQL, JSON, functional dependency, schema discovery

1. INTRODUCTION
Over the past two decades, self-describing semi-structured data formats have become increasingly common solutions for storing and sharing information: NoSQL databases which implement key-value or document-based data models have gained traction among developers seeking freedom from the burden of rigid schemas. Human-readable key-value formats have emerged as the defacto message passing format used in modern APIs, allowing applications to interoperate with many services over a standardized language.

In order to achieve simplicity and avoid schema declarations, these data formats and their associated storage engines tend to lack first-class support for modeling relationships across records. They do not allow primary or foreign key designations, joins among records, or relationship-based constraints. If users wish to model associations, they must encode links as ordinary key-value pairs and implement joining logic in the application layer. As a result of these limitations, the applications built atop these technologies tend to employ denormalized data models and to nest all related data within a single record. When a single data element is referenced by multiple records (as in a one-to-many-relationship), the shared element is stored repeatedly as an embedded attribute.

Since this denormalized layout stores all data relevant to a record physically close together on disk, certain classes of queries can be answered extremely efficiently. For example, queries that require access to all of the data related to a record benefit from this layout because a single sequential I/O operation is sufficient to bring all relevant attributes into memory. If all associated attributes were not embedded in this way and instead modeled by reference, the query’s performance would suffer as a result of the combined I/O and CPU overhead inherent in resolving the relationships.

Despite these performance gains in some scenarios, the denormalization of related data attributes does impose costs which reduce the performance of other queries. First, the redundant storage of shared data structures within their associated records can significantly increase the average width of individual records and the collective size of the relation. This unnecessary bloat negatively impacts queries involving full table scans, as they must repeatedly read identical information from disk. Second, denormalization requires the database to read all attributes related to a record from disk for all queries, even if the query only operates on a small subset of possible attributes. In contrast, if records are split among several relations, the query engine can selectively load the relevant fragments of the relation into memory thereby reducing I/O costs for queries that operate on a subset of available attributes.

These trade-offs between normalized and denormalized data layouts suggest that query performance can be improved by selectively normalizing attributes when the amount of redundancy and the width of the records are significant. Performing this selective normalization by hand, however, presents significant challenges to database administrators dealing with vast quantities of semistructured data. They must interpret the semantic meaning associated with each key and identify which columns will exhibit relationships that they can exploit in their choice of data layout. This process places significant demands on the database administrator, as the key-space may contain many hundreds of columns which may exhibit non-obvious relationships and correlations.

In place of this arduous process, we envision a relational database management system (RDBMS) which shields the user from complex decisions regarding their data’s physical layout. Users will provide our system with a sample of training data, from which it will learn an appropriate layout from the relationships it detects within its training set. When records are inserted into the database, they will be mapped into the learned layout. Users will express queries
over the logical view of the data originally supplied to the system, relying upon the RDBMS to transparently rewrite queries to operate on the true physical structures. Thus, our system will include three primary components: 1) an algorithm which learns an efficient layout for the data, 2) a loader which manipulates user data into the true physical layout of the database and 3) a query rewrite engine which translates requests to operate on the RDBMS’s underlying data structures.

This paper addresses the first component of this system and details an unsupervised machine learning algorithm which identifies candidates for normalization within input data sets. Informally, a normalization candidate is a group of attributes that is likely to represent redundancy within the original denormalized data set. Formally, we define:

**Definition 1.** A normalization candidate $N$ is defined by a set of member attributes, $C_n$, and a parent attribute, $P$, where $\forall c \in C_n : P \rightarrow c$. For a given data set, the member attributes of all normalization candidates must be disjoint.

**Definition 2.** The relationship $C_1 \rightarrow C_2$ denotes that a functional dependency exists from $C_1$ to $C_2$, or that every value of $C_1$ maps precisely to a single value of $C_2$.

The algorithm we present begins by mining likely functional dependencies between all column pairs. It proceeds by partitioning the set of all attributes into normalization candidates based on an analysis of the mined dependencies.

In section two, we briefly cover earlier work related to this project. In section three, we detail the progression of the machine learning algorithm. In section four, we present the algorithm’s output on a sample data set and compare this output with an alternative that we designed by hand. In section five, we discuss opportunities for future work in this space.

2. RELATED WORK

In their paper outlining improvements to the statistics that RDBMSes maintain about related columns, Ilya et. al. propose a technique for identifying functional dependencies between pairs of columns based on statistics that the catalog already maintains for individual columns[1]. Their formulas provide the foundation for the functional dependency detection scheme outlined in section two of our paper, though significant extensions were required to accommodate our purposes.

3. ALGORITHM OVERVIEW

As its input, the algorithm expects a set of records where each record is a key-value map containing only atomic values. If the structure of the data does not originally meet this restriction, it must first be preprocessed into a suitable form (see section 2.3 for mappings from more complex data formats). As its output, the algorithm produces a valid set of normalization candidates which reflect the functional dependencies inherent in the data.

3.1 Phase 1: Schema Discovery and Functional Dependency Detection

Since the algorithm’s input does not include any schema information, the algorithm must begin with a complete pass through the data in order to identify the set of possible keys. Since there is no guarantee that every key will be present in every record, only a full examination the training set can ensure the discovery of the full keyset.

Once the keyset is known, the algorithm considers every pair of columns, performing an exhaustive search for functional dependencies. Since we assume that input data sets contain imperfections and noise, we adopt Ilya et. al.’s practice of searching for “soft” functional dependencies [1]. Whereas “hard” functional dependencies require that every value of $C_1$ exactly maps to a single value of $C_2$, “soft” functional dependencies impose the weaker constraint that $C_1$ maps to a single value of $C_2$ with very high probability[1].

The algorithm assumes that a “soft” functional dependency exists between a column $C_1$ and a distinct column $C_2$ if the pair satisfies a set of selection conditions. We use the following notation to express our selection criteria:

- $|C_i|$ the cardinality (number of unique values) of the column $C_i$, considering the full dataset, but excluding null values
- $||C_i||$ the magnitude (number of values) of the column $C_i$, considering the full data set, but excluding null values
- $\text{skew}(C_i)$ the frequency of the column’s most common non-null value divided by $||C_i||$
- $|C_1 \text{ wrt } C_2|$ the cardinality of the column $C_1$, within records where both $C_1$ and $C_2$ are non-null
- $||C_1 \text{ wrt } C_2||$ the magnitude of the column $C_1$, within records where both $C_1$ and $C_2$ are non-null
- $|C_1, C_2|$ the cardinality of the set of all $(C_1, C_2)$ value pairs, where both columns are non-null
- $||C_1, C_2||$ the magnitude of the set of all $(C_1, C_2)$ value pairs, where both columns are non-null

3.1.1 Condition 1: Non-trivial Relationships

If $|C_2|$ is one, the algorithm dismisses the functional relationship $C_1 \rightarrow C_2$ because $C_2$ will be considered dependent upon every other column in the data set.

3.1.2 Condition 2: Relationship Strength

The ratio \( \frac{|C_1 \text{ wrt } C_2|}{|C_1, C_2|} \) can be interpreted as a measure of the strength of the functional dependency from $C_1$ to $C_2$. If values of $C_1$ tend to appear alongside many different $C_2$ values in the training set, $|C_1, C_2|$ will tend to be much greater than $|C_1|$ and this ratio will be extremely low. At the other extreme, if a hard functional dependency exists, \( |C_1 \text{ wrt } C_2| \) and $|C_1, C_2|$ will be nearly the same size and this ratio will equal 1. Thus, we only identify functional dependencies where \( \frac{|C_1 \text{ wrt } C_2|}{|C_1, C_2|} \geq \epsilon_2 \).

\footnote{This ratio is taken exactly from Ilyas et. al.[1], although we have redefined the quantities involved to exclude nulls.}
In practice, an $\epsilon_2$ of 0.99 has generally been effective at identifying true dependencies. When $C2$ is highly skewed toward a single value, however, the probability that a column pair surpasses this threshold becomes significantly higher and an $\epsilon_2$ begins to detect a large number of false positives. This occurs because the probability that a $C1$ value maps to multiple $C2$ values becomes exceedingly small when $C2$ is predominantly a single value.

To address the issue of skew, we set $\epsilon_2$ to 0.99999 when skew$(C2) \geq 0.99$. We are currently pursuing a more flexible formulation of $\epsilon_2$ which is defined as a function of skew$(C2)$ as a replacement for this stopgap solution to skew handling.

3.1.3 Condition 3: Data Duplication
The algorithm requires that $\frac{|C1 \setminus C2|}{|C1 \setminus C2|} \leq \epsilon_3$ in order to ensure that some duplication of $C1$-$C2$ pairs exists. Since we assume that the output of the algorithm will be post-processed to decide which normalization candidates are translated into separate relations, we experiment with fairly high values of $\epsilon_3$ even though they accept functional dependencies that are not associated with significant redundancy in the data. In practice, we have found 0.90 to be an effective value for $\epsilon_3$.

3.1.4 Condition 4: Relationship Generality
True functional relationships should hold throughout the majority of the data set and should not reflect local trends. However, since we consider the relationship between $C1$ and $C2$ only where both values are non-null (explained in more detail in the next section), the above three conditions may identify relationships that do not characterize the entire data set. Figure 1 illustrates an example of this problem, where $C1$ is only defined for a small subset of $C2$'s potential values. If no further conditions are imposed and a relationship exists between $C1$ and $C2$ where the columns are both defined, the algorithm will likely identify a functional dependency from $C1 \rightarrow C2$ that does not hold globally throughout the data set.

The algorithm imposes two additional constraints in order to assure that only globally valid functional dependencies are identified. First, we require that $\frac{|C1 \setminus C2|}{|C1 \setminus C2|} \leq \epsilon_4$ so that functional relationships are not created between columns when $C1$ is significantly less dense than $C2$. In practice, an $\epsilon_4$ of 1.1 has been effective at eliminating these nonsensical dependencies. Second, we require that $\frac{|C2 \setminus C1|}{|C1 \setminus C2|} \geq \epsilon_5$ to ensure that a significant portion of the possible values of $C2$ are expressed where $C1$ is defined. In practice, we have found $\epsilon_5$ of 0.75 effective at eliminating local dependencies of this type.

3.1.5 Sparse columns and null exclusion
The various magnitude and cardinality measures that lie at the heart of the algorithm’s phase one computations all exclude null values. This exclusion is necessary in order to accurately mine functional dependencies for very sparse attributes. If null values were incorporated into these computations, all sparse attributes would suffer from the skew problem described above because the majority of their values would equal null. As a result, sparse attributes would likely be identified as dependent on many other columns, when in fact those other columns merely map to nulls. By ignoring null values when processing the data set and imposing condition 4, our algorithm remains sensitive to functional dependencies involving sparse columns without sacrificing accuracy for dense values.

3.2 Phase two: Candidate Selection
In phase two, the algorithm groups the attributes of the data set into normalization candidates based on the functional dependencies it discovers in phase one.

Within highly denormalized data sets, it is likely that phase one will identify a large number of functional dependencies because distinct data entities are colocated within a single record. Consider the data from Table 1, which stores Course entities (id and name) and Teacher entities (id and name) on a single record. The "Record ID" serves as a primary key for the entire data set and, consequently, all other columns are functionally dependent upon this attribute. The "Teacher ID" and "Course ID" attributes respectively serve as primary keys for the Teacher and Course entities. As a result, the "Teacher Name" attribute is functionally dependent upon "Teacher ID" and the "Course Name" attribute is functionally dependent upon "Course ID." These dependency relationships are graphed in Figure 2.

This dependency graph reveals that the attributes can be partitioned into many valid sets of normalization candidates. A single normalization candidate $N_1$ might exist with "ID" as its parent attribute and "Teacher ID", "Teacher Name", "Course ID", and "Course Name" as its member attributes. Or three normalization candidates might be defined:

$N_2$ parent: "Teacher ID", members: "Teacher Name"
$N_3$ parent: "Course ID", members: "Course Name"
$N_4$ parent: "ID", members: "Teacher ID", "Course ID"

In order to choose among these valid sets of normalization candidates, the algorithm applies heuristics to construct...
Table 1: A sample data set illustrating the candidate selection process.

<table>
<thead>
<tr>
<th>Record ID</th>
<th>Teacher ID</th>
<th>Teacher Name</th>
<th>Course Name</th>
<th>Course ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>Mr. Smith</td>
<td>History</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>Mr. Smith</td>
<td>Honors History</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>Mr. Smith</td>
<td>Painting</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>222</td>
<td>Mr. Jones</td>
<td>Gym</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>222</td>
<td>Mr. Jones</td>
<td>History</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2: A graph of the functional dependencies present in Table 1

The set which exposes as many functional dependencies as possible. Rather than choosing \( \{N_1\} \) from the previous example, the algorithm instead prefers outputs similar to \( \{N_2, N_3, N_4\} \). This latter output reveals the full dependency chains \( \text{"ID"} \rightarrow \text{"Teacher ID"} \rightarrow \text{"Teacher Name"} \) and \( \text{"ID"} \rightarrow \text{"Course ID"} \rightarrow \text{"Course Name"} \)–which are not evident from the other valid output.

By fully expanding these dependencies in its output, the algorithm provides a flexible foundation for normalization decisions. In translating the normalization candidates produced by the algorithm into relational tables, one may either create a separate table for every candidate or selectively collapse candidates into their parents according to a more extensive normalization policy.

3.2.1 Graph construction

Before forming normalization candidates, the algorithm first constructs a directed graph representing the functional dependencies identified in phase one. Vertices in the graph represent attributes in the data set’s key space. An edge exists from vertex \( V_1 \) to vertex \( V_2 \) if a functional dependency is detected between the corresponding attributes in the data set. The graph also contains a special root vertex that behaves as if it were a unique key present in every record; a directed edge from the root exists to all other vertices. We compute the transitive closure of the graph to ensure that all indirect relationships are transformed into direct relationships.

3.2.2 Parent selection

Operating on the graph from the previous section, we form normalization candidates by iterating over every attribute in the data set and choosing parents for each attribute independently. Attributes which are assigned to the same parent are combined into a normalization candidate in the algorithm’s final output.

In order to expose the deepest dependency chains possible, we select parents according to a simple greedy algorithm: The parent of a column \( C_1 \) is the predecessor in the dependency graph with the highest incoming degree. Ties are broken according to the strength of the relationship (\( \frac{|C_1\cap \{C_2\}|}{|C_1|} \)).

The heuristic employed by the algorithm is motivated by the observation that one plus the number of incoming edges at any given vertex \( V_1 \) is an upper bound on the longest dependency chain ending at that attribute (see proof below). Thus, at every step the algorithm chooses parents capable of producing the longest chains.

**Proof.** Suppose on the contrary that a vertex \( V \) has \( m \) incoming edges in the final dependency graph, that a dependency chain containing \( k \) vertices begins at \( V_{start} \) and ends at \( V \), and that \( k \) is greater than \( m + 1 \).

The sparsest initial graph representing this dependency structure contains only the edges on the path from \( V_{start} \) to \( V \). Each node has at most one incoming and one outgoing edge. After the transitive closure is computed on this graph, \( V \) must have \( k - 1 \) incoming edges because \( V \) is accessible from \( k - 1 \) other nodes in the graph. This represents a contradiction, however, because our assumptions require both \( k - 1 = m \) and \( k - 1 > m \).

Although this greedy algorithm provably fails to produce the most dependent paths in some cases\(^3\), in practice it has proven effective at generating reasonable sets of normalization candidates. In future work, we plan to evaluate an alternative parent selection algorithm that computes the longest path from the special root node to every vertex in the dependency graph and chooses parents which lie along these paths.

3.2.3 Merging nodes

As described in the previous section, our parent selection heuristic is highly sensitive to the number of incoming edges at every vertex. Consequently, we attempt to simplify the graph structure by merging equivalent nodes before performing parent selection. A group of nodes can be considered equivalent if bidirectional functional dependencies exist between all combinations of them and their descendants intersect completely. In this circumstance, we remove the original nodes from the dependency graph and add a new super-node representing the group of attributes. The set of predecessors

\(^3\)Consider a dependency graph where a node, \( C \), has two predecessors, \( A \) and \( B \). There is a path of length \( k \) from the root to \( A \), which implies that \( A \) has \( k \) incoming edges. There are \( k \) paths of length two from the root to \( B \), which implies that \( B \) has \( k + 1 \) edges. Consequently, \( C \) will erroneously select \( B \) as its parent.
to the supernode equals the union of the predecessors to the original nodes and the set of descendants equals the union of all descendants of the original nodes.

3.3 Operating on complex input formats

Although the algorithm’s flat key-value input format does not incorporate all of the features seen in the most popular semistructured data formats, the algorithm still can mine normalization candidates from these more complex formats if they are appropriately pre-processed. The most common features missing from our flat structure should be handled as follows:

3.3.1 Nested objects

Before operating on data containing objects nested within other objects, it is necessary to denote all relationships by concatenating the names along the nesting path. For example, if the original data set contains JSON objects with users nested inside the root object ‘{ user: { name: 'Jason'}, ....}’, the input to the algorithm should be ‘userdata: 'Jason’

We chose to flatten nested objects according to this scheme so that we can identify relationships between all fields of the input, regardless of their original clustering. We intentionally ignore the grouping of columns implied by their original nesting because the nested layout of the input format may not accurately reflect the boundaries of the functional relationships that we use to identify candidates for normalization (see section 4.1.2 for experimental validation of this assumption).

3.3.2 Arrays of atomic values

Arrays of atomic objects can be handled simply by interpreting them as strings. For example, the array attribute [1, 2, 3] can be cast to the atomic value “{1, 2, 3}”. Once cast, these values can be treated as if they were originally flat values.

3.3.3 Arrays of objects

Objects containing arrays of other objects require more aggressive preprocessing than the other scenarios. In their original form, these objects frustrate the algorithm because they reflect either one-to-many or many-to-many associations, rather than the one-to-one or many-to-one associations that the algorithm uses for its heuristic search. Nevertheless, the algorithm can still be adapted to handle these types of relationships if each nested array is considered a subinstance of the original normalization problem. Consider a data set containing two records: {article-title: 'one', comments: [ {text: 'one'} ]} and {article-title: 'two', comments: [ {text: 'two'} ]}. Two separate iterations of the algorithm can be run to select normalization candidates. The first operates on records containing only top-level article attributes: {article-title: 'one'} and {article-title: 'two'}. The second operates on records containing only comment attributes: {text: 'one'} and {text: 'two'}.

After running the algorithm, the user will have a set of normalization candidates for the article attributes and a set of normalization candidates for the comment attributes. In order to form a unified schema, the user must model the relationship between these two sets. If many articles reference the same comment attributes (a many-to-many relationship), an intermediate linking object which associates the root articles relation with the root comments relation should be introduced. If instead the data reflects a one-to-many relationship, the root comments relation should contain a back pointer to the root articles relation.

4. EXPERIMENTAL RESULTS

In order to validate the performance of the proposed algorithm, we considered a sample data set from Twitter which includes 9,901,087 tweets. The original data was encoded in the JSON format and contained approximately 300 possible keys at various levels of nesting. For simplicity, we restricted our analysis to consider only keys contained in the root object or at a single level of nesting. We preprocessed the data set as described in section 3.3.1 to generate input suitable for the algorithm. We computed the necessary statistics about attributes and attribute pairs by loading the data set into a Postgres node and expressing our statistics in SQL.

Before performing any analysis beyond extracting the key set from the data, we attempted to form an appropriately normalized schema from an examination of the attribute names and their definitions in the Twitter API. Throughout the remainder of this section we compare our hand normalization (see Appendix B) with the output of the algorithm (see Appendix A).

4.1 Splitting large entities

The algorithm’s output contains many small groups of attributes, whereas the manual grouping includes only five large attributes.

4.1.1 Retweets

The manual normalization of attributes lays out most retweeted_" attributes along a straight dependency chain of length three (root → retweeted_status_id → retweeted_status_user_id). All fields related to a specific retweet (e.g., its contents, its associated user, etc) are contained within the retweeted_status_id group. All fields related to a user attached to a retweet are included in the retweeted_status_user_id group. In contrast, the algorithm outputs a deeper, branched dependency structure.

In most cases, the more extensive branching of the "retweet_" attributes reflects true functional relationships that were not acknowledged in the manual layout.

- "text" → "retweeted_status_id": although not initially obvious, this relationship reflects that the Twitter interface provides a default value for the "text" attribute when the record represents a retweet. Users likely retweeted many statuses within our dataset without modifying the "text" attribute, leading the algorithm to learn that certain "text" values are associated with certain "retweeted_status_id" values.

- "retweeted_status_user_profile_link_color" → "retweeted_status_user_default_profile": this example has no discernible explanation in the semantics...
of either field, yet does represent a true functional dependency in the data. Within our training set, the default profile attribute is true if, and only if, the color attribute equals "0084B4."

- "retweeted_status_user_time_zone" $\rightarrow$ "retweeted_status_user_utc_offset": this dependency is less interesting than the previous two, as it likely could have been identified with a more careful manual examination of the attribute names. However, its absence from the manual layout does speak to the difficulty of performing these groupings by hand.

In other cases, the algorithm creates less meaningful branches in the retweet structure. Consider the group of attributes with "retweeted_status_user_name" as their parent. The algorithm determines that these attributes are functionally dependent upon the name of the retweeted user because all three attributes are moderately skewed (97%, 95%, 93%) booleans and the "retweeted_status_user_name" field is nearly a primary key on the data (539,991 unique "retweeted_status_user_name"s exist among 569,328 unique "retweeted_status_user_id"s). As a result, it is very likely that each value of the user name column maps to at most one value of each of these booleans, simply by chance.

Regardless of whether these small groups are semantically meaningful, it is unlikely that normalizing out such narrow entities would compress the data set enough to justify the additional CPU costs that normalization imposes. If this is determined to be the case (either through experimentation or simply through a policy that dictates when normalization candidates should be accepted), these attributes can always be incorporated into the dependency structure alongside their parent keys. When this process is used to remove some of narrower groups from the algorithm's output, the retweet dependency path is revealed to be very similar to the manually derived dependency path. At its core, the algorithm's dependency path is nearly root $\rightarrow$ retweeted_status_id $\rightarrow$ retweeted_status_user_id. The principal difference is that the algorithm's version more extensively decomposes theretweeted_status_user_* fields.

4.1.2 Users
In the manual layout, the majority of the "user_*" attributes are grouped together and are directly referred to from the root (implying a dependency path root $\rightarrow$ user_id). As in the case of the "retweet_*" fields, the algorithm's output is deeper than the manual representation. In large part, this more extensive structure results again from functional dependencies not included in the manual version (e.g. the relationship between timezone and utc_offset reemerges and several URL fields are found to be interdependent).

Aside from these distinctions, which have been already explored more closely in the previous section, the "user_*" attributes are interesting in the algorithm's output because they are split into two broad categories that are independently associated with the root object. The first group connects to the root through the merged node "user_id, user_id_str, user_screen_name" and principally contains the user attributes that are unlikely to change (e.g. creation time, the user's Twitter url, the user's location, and the user's time zone). The functional dependency underlying this choice of parent is fairly straightforward. A single value of "user_id" is unlikely to ever map to more than one value of these more-or-less immutable attributes which necessarily implies an N to 1 relationship.

The second group of "user_*" attributes connects through the merged node "user_profile_image_url, user_profile_image_url_https" and principally contains fields regarding the visual display of the user's profile (e.g. colors and image urls). Since an individual user can redesign their profile many times, it is logical that these visual fields are not functionally dependent on user_id. Consequently, it is likely more appropriate to store them separately so that redundant copies of the immutable user fields do not need to be stored each time that a user appears in the data set with a novel design.

Although both chains of attributes include some fields outside of the immutable vs visual classification, the algorithm's output identifies a significant shortcoming of the manual layout.

4.2 Oversensitivity
The algorithm's choice of "user_url" as the parent of "place_country" reflects an instance where our search for the deepest dependency path leads to suboptimal layouts. Perhaps because users tend to post statuses from within a single country, the data set contains a strong functional dependencies from several infrequently updated user attributes (e.g. user_id or user_url) to "place_country." Unsurprisingly, the dependency "place_id" $\rightarrow$ "place_country" also exists. Although it would be more intuitive to group place_country with the other "place_*" attributes, the algorithm chooses to group it with the most deeply nested attribute, and thus chooses "user_url" as its parent attribute.

5. FUTURE WORK
In order to confirm that our algorithm performs well for general data sets, we plan to analyze its output on several other large bodies of denormalized data. We also plan to evaluate the alternative parent selection algorithm alluded to in Section 3.2.2.

Although the algorithm presented here represents a substantial step toward the adaptive RDBMS described in the introduction, significant work still remains. In the immediate future, we plan to automate the process of translating normalization candidates into optimal schemas. This automation will require the development of a normalization policy that decides whether the reduction in data size justifies the materialization of a normalization candidate. We expect to experiment with several policies on several data sets in order to understand their general behavior. We also plan to implement the full RDBMS outlined in our introduction.

6. CONCLUSION
Our experiments show that our algorithm can successfully design normalized schemas for originally denormalized data sets, relying only upon a sufficiently large training set. Although the schemas it produced on our training set did not closely conform to our expectations, in most cases the algorithm's output reflected relationships that we failed to
anticipate. Since these relationships may be inherently interesting to users, our algorithm may have applications as a standalone data exploration tool. More critically to our larger project, the normalization candidates that the algorithm identifies will serve as a strong foundation for designing schemas that appropriately leverage the tradeoffs between normalized and denormalized layouts.

7. ACKNOWLEDGMENTS
Many thanks to Professor Abadi for his guidance and insight throughout the course of this project.

8. REFERENCES
Appendix A: Normalization candidates generated from the algorithm

Attributes are color coded according to their associated attribute in the root table. Keep in mind that attributes may be indirectly associated with the root through a chain of other attributes.

Note: When column names wrap to multiple lines, they’re indented with “-----”

PARENT: ROOT
favorited
id
id_str
in_reply_to_screen_name, in_reply_to_user_id, in_reply_to_user_id_str
in_reply_to_status_id
in_reply_to_status_id_str
place_full_name, place_id, place_url
retweet_count
retweeted_status_favorited
retweeted_status_in_reply_to_screen_name, retweeted_status_in_reply_to_user_id,
----- retweeted_status_in_reply_to_user_id_str
retweeted_status_in_reply_to_status_id
retweeted_status_in_reply_to_status_id_str
retweeted_status_retweet_count
retweeted_status_retweeted
retweeted_status_truncated
retweeted_status_user_favourites_count
retweeted_status_user_followers_count
retweeted_status_user_friends_count
retweeted_status_user_listed_count
retweeted_status_user_protected
retweeted_status_user_statuses_count
source
text
truncated
user_description
user_favourites_count
user_followers_count
user_friends_count
user_id, user_id_str, user_screen_name
user_listed_count
user_name
user_profile_background_image_url, user_profile_background_image_url_https
user_profile_image_url, user_profile_image_url_https
user_protected
user_statuses_count

PARENT: place_full_name, place_id, place_url
place_name
place_place_type
Appendix B: Manually normalized schema
id
id_str
in_reply_to_screen_name
in_reply_to_status_id
in_reply_to_status_id_str
in_reply_to_user_id
in_reply_to_user_id_str
retweet_count
place_id
retweeted_status_id
retweeted_status_id_str
retweeted_status_user_followers_count
retweeted_status_user_friends_count
retweeted_status_user_favourites_count
retweeted_status_user_listed_count
retweeted_status_user_statuses_count
retweeted_status_retweet_count
source
text
truncated
user_favourites_count
user_followers_count
user_friends_count
user_id
user_listed_count
user_statuses_count

PARENT: retweeted_status_id
retweeted_status_id_str
retweeted_status_in_reply_to_screen_name
retweeted_status_in_reply_to_status_id
retweeted_status_in_reply_to_status_id_str
retweeted_status_in_reply_to_user_id
retweeted_status_in_reply_to_user_id_str
retweeted_status_possibly_sensitive
retweeted_status_retweeted
retweeted_status_source
retweeted_status_truncated
retweeted_status_favorited
retweeted_status_user_id

PARENT: place_id
place_country
place_country_code
place_full_name
place_name
place_place_type
place_url