# Unsupervised String Transformation Learning for Entity Consolidation

Dong Deng[†]   Wenbo Tao[♠]   Ziawasch Abedjan[◇]   Ahmed Elmagarmid[♣]   Ihab F. Ilyas[♡]
Guoliang Li[‡]   Samuel Madden[♣]   Mourad Ouzzani[♣]   Michael Stonebraker[♠]   Nan Tang[♣]
[†]Rutgers University  [♠]MIT CSAIL  [◇]TU Berlin  [♣]QCRI, HBKU  [♡]University of Waterloo  [‡]Tsinghua University
{dongdeng, wenbo, stonebraker, madden}@csail.mit.edu,  abedjan@tu-berlin.de,  liguoliang@tsinghua.edu.cn
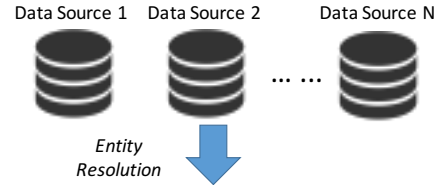{aelmagarmid, mouzzani, ntang}@hbku.edu.qa,  ilyas@uwaterloo.ca

*Abstract*—Data integration has been a long-standing challenge in data management with many applications. A key step in data integration is entity consolidation. It takes a collection of clusters of duplicate records as input and produces a single "golden record" for each cluster, which contains the canonical value for each attribute. Truth discovery and data fusion methods as well as Master Data Management (MDM) systems can be used for entity consolidation. However, to achieve better results, the variant values (i.e., values that are logically the same with different formats) in the clusters need to be consolidated before applying these methods.

For this purpose, we propose a data-driven method to standardize the variant values based on two observations: (1) the variant values usually can be transformed to the same representation (e.g., "`Mary Lee`" and "`Lee, Mary`") and (2) the same transformation often appears repeatedly across different clusters (e.g., transpose the first and last name). Our approach first uses an unsupervised method to generate groups of value pairs that can be transformed in the same way. Then the groups are presented to a human for verification and the approved ones are used to standardize the data. In a real-world dataset with 17,497 records, our method achieved 75% recall and 99.5% precision in standardizing variant values by asking a human 100 yes/no questions, which completely outperformed a state of the art data wrangling tool.

## I. Introduction

Data integration plays an important role in many real-world applications such as customer management, supplier management, direct marketing, and comparison shopping. Two key steps in data integration are entity resolution and entity consolidation. Entity resolution [13] produces clusters of records thought to represent the same entity from disparate data sources. Many resolution methods [8], [28], [29], [5] and systems have been developed in recent years, such as Tamr [25], Magellan [20], and DataCivilizer [9].

Entity consolidation takes as input a collection of clusters, where each cluster contains a set of duplicate records, and outputs a single "golden record" for each cluster, which represents the canonical value for each attribute in the cluster. As the attribute values of two duplicate records may not necessarily be redundant, we cannot simply choose an arbitrary record from each cluster as the golden record. For example, $r_4[Address] =$ "`5 th St, 22701 California`" and $r_5[Address] =$ "`3rd E Ave, 33990 California`" in two duplicate records in Table 1 refer to different addresses and are not redundant. Instead, they conflict with each other.



Fig. 1. An example

Probabilistic methods have been proposed to resolve conflicts in truth discovery and data fusion [10], [12], [30]. They can be adapted for entity consolidation. Master Data Management (MDM) systems leverage human-written rules for entity consolidation. However, to achieve better results, the variant values (values that are logically the same with different formats) in the same clusters should be consolidated before applying existing methods. For this purpose, in this paper, we propose a data-driven method to standardize the variant values. As an example, as shown in Figure 1, our method takes Table 1 as input and outputs Table 2. Afterwards, existing entity consolidation methods can take Table 2 as input and construct the golden records in Table 3.

| | |
|---|---|
| Mary Lee — M. Lee | 9th — 9 |
| Lee, Mary — Mary Lee | Street — St |
| Lee, Mary — M. Lee | 3rd — 3 |
| Smith, James — James Smith | Avenue — Ave |
| Smith, James — J. Smith | Wisconsin — WI |
| James Smith — J. Smith | California — CA |
| *and many more* | |

*Grouping Replacements by Our Unsupervised Method*

| | | | | |
|---|---|---|---|---|
| Lee, Mary — Mary Lee / Smith, James — James Smith | *Group 1* | 9th — 9 / 3rd — 3 | *Group 4* | |
| Lee, Mary — M. Lee / Smith, James — J. Smith | *Group 2* | Street — St / Avenue — Ave | *Group 5* | |
| Mary Lee — M. Lee / James Smith — J. Smith | *Group 3* | California — CA / Wisconsin — WI | *Group 6* | |

Fig. 2. Grouping candidate replacements

**Solution Overview.** We propose a data-driven approach to identify and standardize the variant values in clusters. Because the variant values usually can be transformed to each other (e.g., "Mary Lee"↔"Lee, Mary"), we use an unsupervised method to group all the value pairs in the same cluster (which we call *candidate replacements*) such that the value pairs in the same group can be transformed in the same way (i.e., they share a *transformation*). For example, Figure 2 shows 12 sample candidate replacements in Table 1, along with 6 groups generated by our unsupervised method. The details will be given in the following sections.

Since users usually are not willing to apply a transformation blindly, we ask a human to verify each group. The human browses the value pairs in a group and marks the group as either correct (meaning the transformation is valid, with most or all value pairs representing true variant values) or incorrect (meaning the transformation is invalid and no replacement should be made). The human is not required to exhaustively check all pairs; our method is robust to small numbers of errors as verified in our experiment.

Usually there is a budget for human effort. Therefore we rank the groups by their size and ask a human to check the groups sequentially until the budget is exhausted or the human is satisfied with the result. The reason for this is twofold. First, larger groups are more 'profitable' once they are approved by the human. Second, larger groups are more likely to be correct as the variant values often share common transformations that appear repeatedly across different clusters (e.g., both "Mary Lee"↔"Lee, Mary" and "James Smith"↔"Smith, James" can be transformed by transposing the first and last name). Finally, the approved groups will be used to perform the replacement in the clusters.

**Unsupervised Group Generation.** Clearly, to save human effort, it is desirable for the number of groups to be as small as possible. Thus our goal is to group all the value pairs such that the value pairs within the same group can be transformed in the

same way (i.e., share the same transformation), and the number of groups is minimized. To formally express the transformation (which describes how one string is transformed to another), we borrow a very powerful domain specific language (DSL) designed by Guwani et al [16]. The DSL is very expressive and has been used in production in Microsoft Excel. However, using this DSL, each value pair can be transformed in an exponential number of ways. Moreover, we will prove it is NP-complete to optimally group the value pairs in our criteria.

To alleviate this problem, we develop a simple and effective, data-driven greedy strategy, along with optimization techniques, to group the value pairs. However, this approach incurs a large upfront time cost as it generates all the groups at once. To address this issue, we design an incremental (i.e., top-k) algorithm which generates the next largest group with each invocation. It reduces the upfront cost by up to 3 orders of magnititude in our experiments. We compared with a baseline method that uses the data wrangling tool Trifacta and achieved better precision (all above 99%) and recall (improved by up to 0.3) in standardizing the data with less human effort. Note that instead of verifying the transformations (i.e., groups of value pairs) in our approach, the user was asked to write code (i.e., rules) in the baseline method.

In summary, we make the following contributions:

- We propose an unsupervised method to learn string transformations for entity consolidation.
- We prove it is NP-complete to optimally group the value pairs in our criteria. We design an algorithm to greedily group the value pairs, along with optimization techniques to make it more efficient.
- We develop an incremental algorithm to efficiently generate the groups. At each invocation it guarantees to produce the next largest group for a human to verify.
- We conducted experiments on three real-world datasets. In an address dataset with 17,497 records, by having a human confirm only 100 algorithm-generated groups, we achieved a recall of 75% and a precision of 99.5% for identifying and standarizing variant value pairs.

The rest of the paper is organized as follows. Section II defines the problem. Section III presents our framework. We introduce the DSL in Section IV and give our unsupervised string transformation learning algorithm, along with optimization techniques, in Section V. The incremental algorithm is presented in Section VI. Section VII discusses some implementation details. We report experimental results, review related work, and conclude in Sections VIII, IX, and X.

## II. PROBLEM DEFINITION

Entity consolidation assumes a collection of clusters of duplicate records and produces a "golden record" for each cluster that contains the canonical value for each attribute. In this paper, we focus on the variant value standardization problem in entity consolidation, which identifies and transforms the variant values to the same format, as formalized below.

*Definition 1:* Given a collection **C** of clusters where each cluster $C \in \mathbf{C}$ contains a set of duplicate records, the data

**Algorithm 1**: GOLDENRECORDCREATION(**C**)

---

**Input**: **C** : a set of clusters with $m$ columns
$\mathbf{C}^1, \mathbf{C}^2, \cdots, \mathbf{C}^m$;

**Output**: **GR**: a set of golden records, one for each cluster;

1 **begin**
2    **foreach** *column* $\mathbf{C}^i$ *in* **C** **do**
3      $\Phi$ = GeneratingCandidateReplacements($\mathbf{C}^i$);
4      $\Sigma_{trans}$ = UnsupervisedGrouping($\Phi$);
5      **while** *the budget is not exhausted* **do**
6        pick the largest group $\Sigma$ in $\Sigma_{trans}$;
7        **if** *a human confirms $\Sigma$ is correct* **then**
8          apply the replacements in $\Sigma$ to update $\mathbf{C}^i$;
9        remove $\Sigma$ from $\Sigma_{trans}$ and update $\Sigma_{trans}$;
10    **GR** = TruthDiscovery(**C**);
11    **return GR**
12 **end**

---

standardization problem is to update the clusters such that all the variant values (logically the same values in different formats) in the same cluster become identical.

As shown in Figure 1, a solution to the data standardization problem will ideally take Table 1 as input and output Table 2. In this paper, we focus on the popular case of string values. Different tactics are needed for numerical values.

## III. THE FRAMEWORK

Our golden record construction framework is given in Algorithm 1. It takes a set of clusters **C** as input and outputs a golden record for each cluster. At each iteration it processes one column/attribute $\mathbf{C}^i$ in **C** by the following steps.

**Step 1: Generating Candidate Replacements.** A *replacement* is an expression of the form `lhs → rhs` where `lhs` and `rhs` are two different strings. A replacement states that the two strings `lhs` and `rhs` are matched (e.g., "Mary Lee"→"Lee, Mary"), and thus one could be replaced by the other at certain places[1] in $\mathbf{C}^i$. Since the values within the same cluster in $\mathbf{C}^i$ are potentially duplicates, one way to get candidate replacements is to enumerate every pair of non-identical values $v_j$ and $v_k$ within the same cluster in $\mathbf{C}^i$ and use them to form two candidates replacements $v_j \to v_k$ and $v_k \to v_j$. For example, consider the attribute `Name` in Table 1. 12 candidate replacements are generated from the two clusters and 6 of them are shown in Figure 2 on top-left. By the end of step 1, we have a set $\Phi$ of candidate replacements (Line 3).

**Step 2: Generating Groups of Replacements.** In this step, we partition the candidate replacements in $\Phi$ into groups such that the candidate replacements in the same group share a common transformation (which describes how one string transformed to another). We introduce a language to formally express transformations in Section IV-A. Note that each replacement group corresponds to a transformation; thus this step is essentially

---

[1]e.g., not all "St"s are "Street" in addresses; they can also be "Saint".

---

conducting unsupervised string transformation learning. For example, Figure 2 shows 6 groups generated from the 12 candidate replacements. Group 1 shares the transformation of transposing the first and last name; while group 2 objects take the initial of the first name and concatenate it with the last name. By the end of this step, we have a set $\Sigma_{trans}$ of groups, which is a partition of $\Phi$ (Line 4).

**Step 3: Applying Approved Replacement Groups.** The groups in $\Sigma_{trans}$ are ranked by their size in descending order. We sequentially present each group to a human expert for verification. The expert either rejects or approves a replacement group. If it is approved, the expert needs to further specify the replacement direction, i.e., either replacing `lhs` with `rhs` or the other way around. The verification stops once the budget is exhausted or the expert is satisfied with the results.

The reason for confirming the groups in decreasing size order is twofold. First, the more replacements there are in a group, the more places we can apply them to update the clusters once the group is approved by the human, i.e., the larger groups are more 'profitable' once they are approved. Second, the larger groups are more likely to be approved, as variant values often share common transformations that appear repeatedly across different clusters (e.g., transposing the first and last name). On the other hand, the transformations of value pairs in smaller groups are more peculiar and uncommon. Thus the value pairs in smaller groups are less likely to be variant values and get approved. Section VII-A gives the details. By the end of this step, $\mathbf{C}^i$ is updated (Lines 5-9).

**Running Truth Discovery.** Finally, after we process all the columns in **C** by the above steps, a truth discovery algorithm is applied on the updated clusters **C** to resolve any potential conflicts. In the end, we have the golden records (Line 10).

## IV. GENERATING GROUPS

We introduce the DSL in Section IV-A and discuss how to group the candidate replacements in $\Phi$ in Section IV-B.

### A. Transformation Programs

**Transformation Programs.** A *transformation program* (or program for short) captures how one string is transformed to another. We adopt the domain specific language (DSL) designed by Gulwani [15], [16] to express the programs. Here we give a high level description. In a nutshell, a transformation program takes a string **s** as input and outputs another string **t**. The DSL defines *position functions* and *string functions*, which all apply to the input string **s**.

A position function returns an integer that indicates a position in **s** based on a collection of pre-defined regular expressions (regexes). For example, Figure 3 on the left shows some example position functions and pre-defined regexes. Let the input string **s** be "Lee, Mary", as shown in Figure 4, we have $\mathbf{s}.P_A = 1$ as the $1^{st}$ match of the capital regex $\mathsf{T}_C$ in **s** is "L" and the beginning position of "L" is 1.

A string function returns either a substring of **s** or a constant string. The returned substring is located by two position functions. Thus the space of string functions is all pairs of possible

| example pre-defined **regexes** | example **position functions** | example **string functions** | an example **program** |
|---|---|---|---|
| capital regex: $T_C = $ [A-Z]+<br>lowercase regex: $T_l = $ [a-z]+<br>whitespace regex: $T_b = $ \s+<br>digital regex: $T_d = $ [0-9]+ | $P_A$ : beginning of the 1st match of $T_C$<br>$P_B$ : ending of the 1st match of $T_l$<br>$P_C$ : ending of the 1st match of $T_b$<br>$P_D$ : ending of the last (-1st) match of $T_C$ | $f_1$ : Substring $(P_A, P_B)$<br>$f_2$ : Substring $(P_C, P_D)$<br>$f_3$ : Constant (". ") | $\rho := f_2 \oplus f_3 \oplus f_1$ |
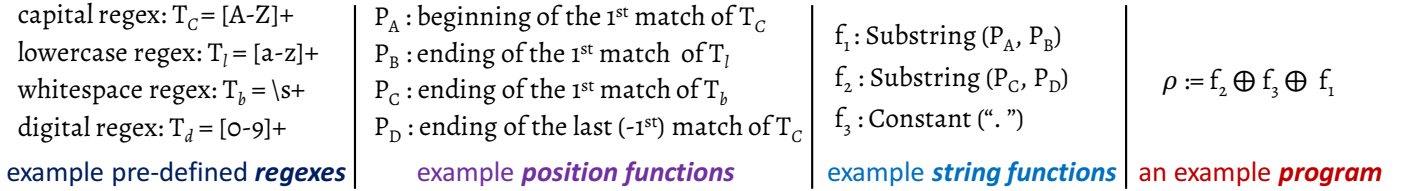
Fig. 3. An example program $\rho := f_2 \oplus f_3 \oplus f_1$
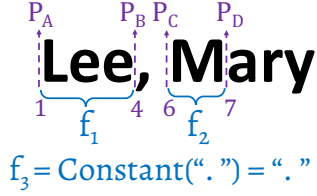
Fig. 4. Evaluating the example functions and program $\rho$ in Figure 3 on an input string "Lee, Mary"

Fig. 5. The graph for "Lee, Mary"→"M. Lee"

position functions. A program is defined as a sequence of string functions and its output **t** is the concatenation of the outputs of these string functions. Figures 3 and 4 show an example. For **s** = "Lee, Mary", we have **s**.$f_1$ = "Lee" and **s**.$f_2$ = "M". The program $\rho := f_2 \oplus f_3 \oplus f_1$ will produce **t** = $\rho($**s**$)$ = "M. Lee".

**Transformation Graph.** We say a program $\rho$ is consistent with a replacement **s** → **t** (or $\rho$ can express the replacement) iff $\rho($**s**$)$ produces **t**. Due to the many possible combinations of string functions, there are an exponential number of *consistent programs* for a given replacement **s** → **t**. Fortunately, all the consistent programs of a replacement can be encoded in a directed acyclic graph (DAG) in polynomial time and space [15]. Intuitively, each node in the graph corresponds to a position in **t**, each edge in the graph corresponds to a substring of **t** and the labels on the edge are string functions that return this substring when being applied to **s**.

*Example 1:* Figure 5 shows the transformation graph for "Lee, Mary"→"M. Lee". Some notations are borrowed from Figure 3. The string functions (edge labels) associated with each edge are also shown in the figure. For simplicity, we only show 5 out of all the 21 edges and ignore some edge labels in the figure. The edge $e_{4,7}$ corresponds to the substring "Lee". One of its label is $f_1$, as it returns "Lee" when being applied to **s** = "Lee, Mary" as discussed before. Substring$(P_A, P_E)$ is also a label of $e_{4,7}$, as **s**.$P_A = 1$, **s**.$P_E = 4$, and **s**.Substring$(1, 4)$ = "Lee".

Formally, the transformation graph is defined as below.

*Definition 2 (Transformation Graph):* Given a replacement **s** → **t**, its transformation graph is a directed acyclic graph $G(\mathbf{N}, \mathbf{E})$ with a set **N** of nodes and a set **E** of edges. There are $|\mathbf{t}|+1$ nodes, i.e., $\mathbf{N} = \{n_1, \ldots, n_{|\mathbf{t}|+1}\}$. There is a directed edge $e_{i,j} \in \mathbf{E}$ from $n_i$ to $n_j$ for any $1 \le i < j \le |\mathbf{t}| + 1$. Moreover, each edge $e_{i,j}$ is labeled with a set of string functions that returns $\mathbf{t}[i, j-1]$ when being applied to **s**.

The transformation graph can be built in $O(|\mathbf{s}|^2|\mathbf{t}|^2)$ time and there are $O(|\mathbf{t}|^2)$ edges in the graph. As a replacement has only one transformation graph, we refer to a replacement and
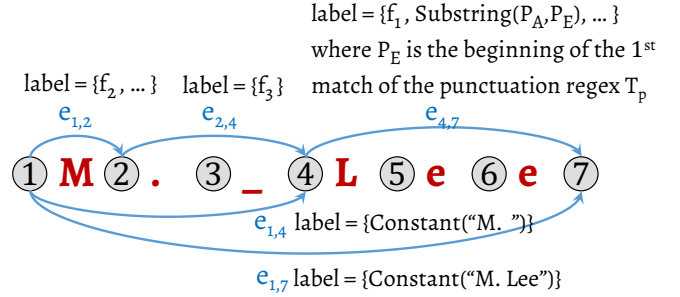
its transformation graph (or graph for short) interchangeably.

**Transformation Path.** Given a replacement **s** → **t**, a *transformation path* is a path in its graph from the *first node* $n_1$ to the *last node* $n_{|\mathbf{t}|+1}$, where each edge has only one label (i.e., a string function). Note that a transformation path uniquely refers to a consistent program. We use them interchangeably. For instance, two transformation paths in Figure 5 are:

$$\rho_1 := ① \xrightarrow{f_2} ② \xrightarrow{f_3} ④ \xrightarrow{f_1} ⑦ \text{ and } \rho_2 := ① \xrightarrow{\text{Constant ("M. Lee")}} ⑦$$

Next we discuss how to use transformation graphs to group the candidate replacements in $\Phi$.

*B. The Optimal Partition Problem*

As discussed before, given a collection of replacements, we aim to group them such that the replacements within the same group share a consistent program (i.e., their graphs share a transformation path) while the number of groups is minimum.

*Definition 3 (Optimal Partition):* Given a set $\Phi$ of replacements, the *optimal partition* problem is to partition $\Phi$ into disjoint groups $\Phi_1, \ldots, \Phi_n$ such that (i) the replacements in each group $\Phi_i$ share at least one consistent program; and (ii) the number of groups $n$ is minimum.

Unfortunately, we can prove the optimal partition problem is NP-complete by a reduction from the set cover problem (proof sketch: each transformation path corresponds to the set of graphs in $\Phi$ containing this path; the optimal partition problem is to find the minimum number of transformation paths such that the set of all graphs is covered). As it is prohibitively expensive to find the optimal partition, we employ a standard greedy strategy. For each replacement $\varphi \in \Phi$, we denote the transformation path in its graph that is shared by the largest number of the graphs in $\Phi$ as its *pivot path*. Then the replacements with the same pivot path are grouped together[2]. We discuss how to find the pivot path for a given replacement

---

[2]Note that two paths are considered to be the same if each pair of string functions in the two sequences are the same.

**Algorithm 2:** UNSUPERVISEDGROUPING($\Phi$)

**Input**: $\Phi$: a collection of candidate replacements.
**Output**: $\Sigma$: groups of replacements with the same transformation, where $\Sigma[\rho]$ contains all the replacements in $\Phi$ with $\rho$ as the pivot path.

1 **begin**
2     Build graphs **G** for all replacements in $\Phi$;
3     Build inverted index $I$ for all edge labels in **G**;
4     **foreach** *graph* $G \in$ **G** **do**
5        $\rho = \rho_{max} = \ell_{max} = \phi$ ;
6        SEARCHPIVOT($G$, $\rho$, **G**, $n_1$, $\rho_{max}$, $\ell_{max}$);
7        add $G$ to the group $\Sigma[\rho_{max}]$;
8     **return** $\Sigma$;
9 **end**

---

**Algorithm 3:** SEARCHPIVOT($\rho$, $\ell$, $n_i$, $\rho_{max}$, $\ell_{max}$)

**Input**: $G$: a transformation graph;
         $\rho$: a path in $G$ starting from $n_1$;
         $\ell$: the list of graphs in **G** containing $\rho$;
         $n_i$: the node at the end of $\rho$;
         $\rho_{max}$: the best path in $G$ found so far;
         $\ell_{max}$: the list of graphs in **G** containing $\rho_{max}$.

1 **begin**
2     **if** $n_i$ *is the last node in $G$* **then**
3        **if** $|\ell| > |\ell_{max}|$ **then**
4           $\rho_{max} = \rho$;
5           $\ell_{max} = \ell$;
6     **else**
7        **foreach** *edge $e_{i,j}$ from $n_i$ to $n_j$ in $G$* **do**
8           **foreach** *string function label $f$ on $e_{i,j}$* **do**
9              $\rho' = \rho \oplus f$;
10              $\ell' = \ell \cap I[f]$;
11              SEARCHPIVOT($G$, $\rho'$, $\ell'$, $n_j$, $\rho_{max}$, $\ell_{max}$);
12 **end**

---

in $\Phi$ in next section. In this way, we can partition the replacements in $\Phi$ into disjoint groups.

## V. SEARCHING FOR THE PIVOT PATH

We use **G** to denote the set of graphs corresponding to the candidate replacements in $\Phi$. Section V-A gives the pivot path search algorithm and Section V-B presents two optimizations.

### A. Pivot Path Search Algorithm

A naive method enumerates every transformation path $\rho$ in a graph $G$ and counts the number of graphs in **G** containing $\rho$. Then the pivot path is the transformation path contained by the largest number of graphs. For this purpose, given a path $\rho := f_1 \oplus f_2 \oplus \cdots \oplus f_m$, we first show how to get the list of graphs in **G** containing $\rho$.

We observe that if a graph $G$ contains $\rho$, every string function $f_1, f_2, \cdots, f_m$ of $\rho$ must appear in $G$ as a label. Thus we can build an inverted index $I$ with string functions as keys. The inverted list $I[f]$ consists of all the graphs $G \in$ **G**

---

**Algorithm 4:** EARLYTERMINATION

1 **begin**
    // add after Line 2 of Algorithm 2
2     **foreach** *graph* $G \in$ **G** **do** set $G_{lo}$ as 1;
    // add after Line 2 of Algorithm 3
3     **foreach** *graph* $G' \in \ell$ **do**
4        **if** $G'_{lo} < |\ell|$ **then** $G'_{lo} = |\ell|$;
    // add before Line 11 of Algorithm 3
5     **if** $|\ell'| > |\ell_{max}|$ *and* $|\ell'| \geq G_{lo}$ **then**
6 **end**

---

that have $f$ (i.e., $f$ is an edge label in $G$). Then given a path $\rho := f_1 \oplus \cdots \oplus f_m$, we can find the list of graphs in **G** that contain $\rho$ by taking the intersection $I[f_1] \cap \cdots \cap I[f_m]$.

However, since $\rho$ is a path, the edges corresponding to the string functions $f_1, f_2, \cdots, f_m$ are required to be adjacent in the graphs. To enable this, we also add the edge information to the entries of the inverted lists. In particular, the inverted list $I[f]$ consists of all triples $\langle G, i, j \rangle$ such that the edge $e_{ij}$ from $n_i$ to $n_j$ in $G$ has the label $f$. Then, when intersecting $I[f_1]$ with $I[f_2]$, only if an entry $\langle G, i_1, j_1 \rangle$ from $I[f_1]$ and another entry $\langle G, i_2, j_2 \rangle$ from $I[f_2]$ satisfy $j_1 = i_2$ (i.e., their edges $e_{i_1,j_1}$ and $e_{i_2,j_2}$ are adjacent), they produce a new entry $\langle G, i_1, j_2 \rangle$ in the result list. By doing so, one can verify that $I[f_1] \cap \cdots \cap I[f_m]$ is exactly the list of graphs in **G** containing $\rho$. Hereinafter, whenever we intersect two lists, we intersect them in the way we described above.

*Example 2:* Consider $\varphi_1$="Lee, Mary"→"M. Lee", $\varphi_2$="Smith, James"→"J. Smith", and $\varphi_3$="Lee, Mary"→"Mary Lee" and their transformation graphs $G_1$, $G_2$, and $G_3$. Using the string functions $f_1, f_2$, and $f_3$ in Figures 3-5, we have $I[f_1]=(\langle G_1, 4, 7 \rangle, \langle G_2, 4, 9 \rangle, \langle G_3, 6, 9 \rangle)$, $I[f_2]=(\langle G_1, 1, 2 \rangle, \langle G_2, 1, 2 \rangle, \langle G_3, 1, 2 \rangle)$, and $I[f_3] = (\langle G_1, 2, 4 \rangle, \langle G_2, 2, 4 \rangle)$. The path $f_2 \oplus f_3 \oplus f_1$ is contained by $\varphi_1$ and $\varphi_2$ as $I[f_2] \cap I[f_3] \cap I[f_1] = (\langle G_1, 1, 7 \rangle, \langle G_2, 1, 9 \rangle)$.

**The Search Algorithm.** As there are an exponential number of transformation paths in a graph, it is prohibitively expensive for the naive method to enumerate all of them. To alleviate this problem, we give a recursive algorithm to find the pivot path in a graph $G$. At a high level, the algorithm maintains a path $\rho$ in $G$ starting from the first node $n_1$ and the list $\ell$ of all graphs in **G** containing $\rho$. At each invocation, the algorithm will try to append a label (string function) $f$ on the outgoing edges of the last node $n_i$ in $\rho$ to the end of $\rho$ and update $\ell$ to the list of graphs containing the new path. After this, if $\rho$ does not reach the last node in $G$, the algorithm will be invoked again to further extend $\rho$. Otherwise $\rho$ reaches the last node and must be a transformation path; and $\rho$ is the pivot path if $\ell$ has the largest number of the graphs in **G**.

Algorithm 3 gives the pseudo-code of the search algorithm. At each invocation, it takes six parameters: a graph $G$, a path $\rho$ in $G$ starting from the first node $n_1$, the list $\ell$ of graphs in **G** containing $\rho$, the node $n_i$ at the end of $\rho$, the best path $\rho_{max}$ (i.e., contained by the largest number of graphs in **G**) in $G$ found so far, and the list $\ell_{max}$ of graphs in **G**

TABLE IV
AN EXAMPLE OF SEARCHPIVOT

| | $\rho$ | $\ell$ | $n_i$ | $\rho_{max}$ | $\ell_{max}$ | $f$ |
|---|---|---|---|---|---|---|
| 1 | $\phi$ | $\{\langle G_1,1,1\rangle, \langle G_2,1,1\rangle, \langle G_3,1,1\rangle\}$ | $n_1$ | $\phi$ | $\phi$ | Constant("M. Lee") on $e_{1,7}$ |
| 2 | Constant("M. Lee") | $\{\langle G_1,1,7\rangle\}$ | $n_7$ | Constant("M. Lee") | $\{\langle G_1,1,7\rangle\}$ | — |
| 3 | $\phi$ | $\{\langle G_1,1,1\rangle, \langle G_2,1,1\rangle, \langle G_3,1,1\rangle\}$ | $n_1$ | Constant("M. Lee") | $\{\langle G_1,1,7\rangle\}$ | $f_2$ on $e_{1,2}$ |
| 4 | $f_2$ | $\{\langle G_1,1,2\rangle, \langle G_2,1,2\rangle\}$ | $n_2$ | Constant("M. Lee") | $\{\langle G_1,1,7\rangle\}$ | $f_3$ on $e_{2,4}$ |
| 5 | $f_2 \oplus f_3$ | $\{\langle G_1,1,4\rangle, \langle G_2,1,4\rangle\}$ | $n_4$ | Constant("M. Lee") | $\{\langle G_1,1,7\rangle\}$ | $f_1$ on $e_{4,7}$ |
| 6 | $f_2 \oplus f_3 \oplus f_1$ | $\{\langle G_1,1,7\rangle, \langle G_2,1,9\rangle\}$ | $n_7$ | $f_2 \oplus f_3 \oplus f_1$ | $\{\langle G_1,1,7\rangle, \langle G_2,1,9\rangle\}$ | — |

containing $\rho_{max}$. First, it checks whether the maintained path $\rho$ is a transformation path (Line 2). If $\rho$ is a transformation path and there are more graphs in $\mathbf{G}$ containing $\rho$ than $\rho_{max}$, the algorithm updates the best path $\rho_{max}$ found so far as $\rho$ and the list $\ell_{max}$ as $\ell$ (Lines 3 to 5). If $\rho$ is not a transformation path, it tries to extend $\rho$ with one more edge label (string function). Specifically, for each outgoing edge $e_{i,j}$ of the node $n_i$, which must be adjacent to $\rho$, and each string function label $f$ on $e_{i,j}$, it appends $f$ to the end of $\rho$ to get a new path $\rho'$ and intersects $\ell$ with the inverted list $I[f]$ to get the list $\ell'$ of graphs containing the new path $\rho'$ (Lines 7 to 10). Then the algorithm recursively invokes itself to examine the new path $\rho'$ (Line 11). When the recursive algorithm terminates, $\rho_{max}$ must be the pivot path of $G$. Initially, $\rho$, $\rho_{max}$, and $\ell_{max}$ are all empty while $\ell$ contains all the graphs in $\mathbf{G}$, as an empty path can be contained by any graph (Line 5 of Algorithm 2).

*Example 3:* Consider the graphs $\mathbf{G}$ in Example 2. We invoke SEARCHPIVOT to search the pivot path of $G_1$. As shown in the first row of Table IV, initially $\rho$, $\rho_{max}$, and $\ell_{max}$ are all $\phi$ and $\ell$ has all the graphs in $\mathbf{G}$. Next we go through every label on the edges starting from $n_1$.

For example, consider the label Constant("M. Lee") on $e_{1,7}$ as shown in Figure 5. We update the maintained path $\rho$ and list $\ell$, as shown in row 2 of Table IV, and invoke SEARCHPIVOT again with $n_i$ as the endpoint of $e_{1,7}$, i.e., $n_7$. Since $n_7$ is the last node in the graph, $\rho$ is a transformation path and we assign $\rho$ and $\ell$ to $\rho_{max}$ and $\ell_{max}$ respectively.

Next, we explore another edge starting from $n_1$. The maintained path $\rho$ and list $\ell$ are restored as shown in row 3 of Table IV. Consider the label $f_2$ on $e_{1,2}$. We update $\rho$ and $\ell$ to row 4 of Table IV and invoke SEARCHPIVOT again with $n_i$ as the endpoint of $e_{1,2}$, i.e., $n_2$. As $n_2$ is not the last node in $G_1$, we further go through the labels on the edges starting from $n_2$. Eventually, as shown in row 6 of Table IV , $\rho$ is extended to a transformation path $f_2 \oplus f_3 \oplus f_1$. Since the list $\ell$ has more graphs than $\ell_{max}$, we update $\rho_{max}$ and $\ell_{max}$ by $\rho$ and $\ell$ respectively. The algorithm continues to explore the graph but could not find any better transformation path. Finally it returns the pivot path $\rho_{max} = f_2 \oplus f_3 \oplus f_1$.

Algorithm 2 gives the pseudo-code of our unsupervised string transformation learning algorithm UNSUPERVISED-GROUPING. Each generated group $\Sigma[\rho]$ corresponds to a string transformation program $\rho$ written in our DSL.

### B. Improving Pivot Path Search

In this section, we introduce two optimizations to improve the efficiency of the pivot path search algorithm. Intuitively, intersecting two inverted lists cannot result in a longer list.

Thus if the length of $\ell$ is no longer than that of $\ell_{max}$, we can skip recursively invoking the algorithm to extend $\rho$ to a transformation path as it cannot result in any transformation path contained by more graphs in $\mathbf{G}$ than $\rho_{max}$ (i.e., cannot result in the pivot path). Next we discuss the details.

**Local Threshold-based Early Termination.** The length of the maintained list $\ell$ decreases monotonically as the maintained path $\rho$ is getting longer. This is because each time a label $f$ is appended to $\rho$, $\ell$ is updated to $\ell \cap I[f]$ and gets shorter. As we only need the pivot path – the one that is shared by the largest number of the graphs in $\mathbf{G}$, we can use $|\ell_{max}|$ as a (local) threshold: only if $|\ell| > |\ell_{max}|$, the algorithm is recursively invoked. To enable this, we add an if condition "$|\ell'| > |\ell_{max}|$" before Line 11 of Algorithm 3.

**Global Threshold-based Early Termination.** Once the maintained path $\rho$ becomes a transformation path (Line 2 in Algorithm 3), we know all the graphs in $\ell$ must contain $\rho$. Thus, for each graph in $\ell$, any path no better than $\rho$ must not be its pivot path. Specifically, we can use $|\ell|$ as a (global) threshold for those graphs in $\ell$. Then when searching for the pivot paths of the graphs in $\ell$, we can use this global threshold for early termination in the same way as the local threshold. To enable this in Algorithm 3, we can associate each graph $G$ with a global threshold $G_{lo}$. Then whenever a transformation path is found (Line 2 in Algorithm 3), the corresponding global threshold $G'_{lo}$ of the graph $G'$ in $\ell'$ will be updated to $|\ell'|$ if $|\ell'|$ is larger. Algorithm 4 shows how to enable the two early optimizations in SEARCHPIVOT.

*Example 4:* Continue with Example 3. At the $6^{th}$ row of Table IV, we have the local threshold of $G_1$ as $|\ell_{max}| = 2$. Then, as none of the edge labels of $G_1$ has an inverted list length longer than 2, we will not invoke SEARCHPIVOT any more and have the pivot path $\rho_{max} := f_2 \oplus f_3 \oplus f_1$. Moreover, as $\rho$ is a transformation path, we set the global threshold of $G_2 \in \ell$ as $|\ell| = 2$. Then, when we search for the pivot path of $G_2$, we can skip all edge labels with inverted list length shorter than 2, including Constant("J. Smith") on $e_{1,9}$.

## VI. INCREMENTAL GROUPING METHOD

We observe that the approach UNSUPERVISEDGROUPING in Section V-A partitions all the replacements upfront. This will incur a huge upfront cost, i.e., the users need to wait a long time before any group is generated. Moreover, due to the limited budget, many small groups will not be presented to the user for verification and it is unnecessary to generate them. To alleviate this problem, we propose an incremental algorithm (i.e., top-k algorithm) in this section. It produces the largest group at each invocation. Next we give the details.

## A. Largest Group Generation

We first give the intuition of largest group generation. We denote the pivot path that is shared by the largest number of graphs in $\mathbf{G}$ as the *best pivot path* $\rho_{best}$. Then the list $\ell_{best}$ of graphs containing $\rho_{best}$ must be the largest group. This is because no other path can be shared by more graphs than $\rho_{best}$. Next we show how to calculate $\rho_{best}$ and $\ell_{best}$.

Intuitively, for each graph $G$ in $\mathbf{G}$, we associate it with an upper bound and a lower bound of the number of graphs in $\mathbf{G}$ containing its pivot path. Let $\tau$ be the largest lower bound among all the graphs in $\mathbf{G}$. We visit each graph in $\mathbf{G}$ and invoke SEARCHPIVOT to find its pivot path. In this process, the lower and upper bounds of the graphs in $\mathbf{G}$ will become tighter and the largest lower bound $\tau$ will be updated accordingly. We stop once $\tau$ is no smaller than any upper bound of the unvisited graphs. Then the pivot path shared by the largest number of graphs so far must be the best pivot path $\rho_{best}$. This is because for the unvisited graphs, their pivot paths must be shared by no more than $\tau$ graphs, while one of the pivot paths in the visited graphs must be shared by no less than $\tau$ graphs.

Formally, as discussed in Section V-B, for a graph $G$, we update its global threshold $G_{lo}$ only if a transformation path in $G$ is found by SEARCHPIVOT. Thus we can use the global threshold $G_{lo}$ as the lower bound of $G$. We discuss how to initialize the upper bound $G_{up}$ in Section VI-B. Then we sort all the graphs in $\mathbf{G}$ by their upper bounds in descending order and visit them sequentially. As we only need the best pivot path $\rho_{best}$, it is unnecessary to find the pivot path of a graph $G$ if it is shared by no more than $\tau$ graphs in $\mathbf{G}$. For this purpose, when visiting a graph $G$ and searching for its pivot path, we use $\tau$ as a local threshold (recall Section V-B) in SEARCHPIVOT. Then SEARCHPIVOT either finds its pivot path shared by more than $\tau$ graphs or concludes its pivot path cannot be shared by more than $\tau$ graphs. In the latter case, we can assign $\tau$ as a tighter upper bound for $G$. In the former case, since SEARCHPIVOT already finds the pivot path $\rho_{max}$ of $G$, we can update its lower and upper bound to the number of graphs sharing $\rho_{max}$. Moreover, since $\rho_{max}$ is shared by more than $\tau$ graphs, the largest lower bound $\tau$ should also be updated. We stop whenever $\tau$ is no smaller than the upper bound of the currently visiting graph as the graphs are ordered by their upper bounds. Then all the graphs with lower bounds equal to $\tau$ form the largest group.

*Example 5:* Continue with Example 2. Initially the lower bounds of $G_1$, $G_2$, and $G_3$ are all 1 and the upper bounds are 2, 2, and 1 respectively (we will discuss this details in the next section). The largest lower bound $\tau = 1$. Then we invoke SEARCHPIVOT to find the pivot path of $G_1$ as discussed in Example 3. We find the pivot path $\rho_{max}$ of $G_1$ is shared by 2 graphs $G_1$ and $G_2$. Thus we update the lower bound of $G_1$ to 2 and the largest lower bound $\tau$ to 2. Next we visit the second graph $G_2$. Since its upper bound is 2, which is no larger than $\tau = 2$, we can stop and $\rho_{max}$ must be the best pivot path and the largest group consists of $G_1$ and $G_2$.

---

**Algorithm 5:** INCREMENTALGROUPING

```
// replace Line 4 of Algorithm 1
1 G = Preprocessing(Φ);
// replace Line 6 of Algorithm 1
2 Σ = GenerateNextLargestGroup(G);
// replace Line 9 of Algorithm 1
3 remove all the graphs in Σ from G and update G;
```

---

## B. Initializing the Upper Bounds

Next we discuss how to initialize an upper bound for a graph. We observe that the pivot path is a sequence of edge labels and the number of graphs sharing the pivot path is the intersection size of the inverted lists of these edge labels. Thus, for any graph, we can use the length of the longest inverted list among all its edge labels as an upper bound as the intersection cannot result in longer lists.

Clearly, we desire the upper bound to be as tight as possible to reach the stop condition earlier. To achieve a tighter upper bound, we have the following observation. The pivot path must cover the entire output string $\mathbf{t}$, i.e., it goes from the first node $n_1$ to the last $n_{|\mathbf{t}|+1}$. Thus, for any node $n_k$, one of the edges $e_{i,j}$ where $i \le k < j$ (i.e., $e_{i,j}$ "covers" $n_k$) must appear in the pivot path. Based on this observation, we can deduce an upper bound $ub[k]$ from any position $n_k$, which is the length of the longest inverted list among all the labels of the edge $e_{i,j}$ where $i \le k < j$, i.e., $e_{i,j}$ covers $n_k$. Since every value in $ub$ is an upper bound, we use the tightest one (i.e., the smallest value in $ub$) to initialize $G_{up}$.

*Example 6:* Continue with Example 2. For $G_1$, we have $ub[5] = 3$ as the label $f_1$ on $e_{4,7}$ has an inverted list $I[f_1]$ of length 3 as shown in Example 2. $ub[2] = 2$ as none of the labels of $G_3$ can produce the character '.'. Finally the upper bound of $G_1$ is initialized as $ub[2] = 2$.

## C. The Incremental Algorithm

Algorithm 5 shows the pseudo-code of our incremental algorithm. Instead of invoking UNSUPERVISEDGROUPING in our framework in Algorithm 1, we first invoke Algorithm 6 to preprocess the candidate replacements (Line 1). Then, while the budget is not exhausted, we invoke Algorithm 7 to produce the next largest group $\Sigma$ for a human to verify (Line 2) and update the graphs as necessary (Line 3).

Algorithm 6 takes the set $\Phi$ of candidate replacements as input. It creates the graphs $\mathbf{G}$ for $\Phi$ (Line 2), builds the inverted index $I$ (Line 3), and initializes the lower bounds (Line 5) and upper bounds (Lines 6-11) for the graphs in $\mathbf{G}$.

Each invocation of Algorithm 7 produces the next largest group. It first initializes the largest lower bound $\tau$ (Line 2). Then it sorts all the graphs in $\mathbf{G}$ by their upper bounds in descending order and visit them sequentially (Lines 3-4). It uses two variable $\rho_{best}$ and $\ell_{best}$ to keep the best pivot path found so far and the list of graphs in $\mathbf{G}$ containing $\rho_{best}$. When visiting a graph $G$, it first checks whether its upper bound is larger than $\tau$. If so, we can stop and return $\ell_{best}$ as $\rho_{best}$ must be the best pivot path (Line 5). Otherwise, it invokes SEARCHPIVOT to check if the pivot path of $G$ is contained

**Algorithm 6**: PREPROCESSING($\Phi$)

  **Input**: $\Phi$: a collection of candidate replacement.
  **Output**: **G**: the set of graphs corresponding to $\Phi$.

1  **begin**
2     Construct graphs **G** for all replacement in $\Phi$;
3     Build inverted index $I$ for all edge labels in **G**;
4     **foreach** *graph* $G \in$ **G** **do**
5         set $G_{lo}$ as 1;
6         **foreach** *edge* $e_{i,j}$ *in* $G$ **do**
7             **foreach** *string function label* $f$ *on* $e_{i,j}$ **do**
8                 **foreach** $i \leq k < j$ **do**
9                     **if** $ub[k] < |I[f]|$ **then**
10                       $ub[k] = |I[f]|$;

11         set $G_{up}$ as the smallest value in $ub$;

12     **return** **G***;*
13 **end**

---

**Algorithm 7**: GENERATENEXTLARGESTGROUP(**G**)

  **Input**: **G**: a set of transformation graphs.
  **Output**: $\ell_{best}$: the list of graphs in **G** containing the best path $\rho_{best}$ that shared by the largest number of graphs in **G**.

1  **begin**
2     let $\tau$ be the largest lower bound in **G**;
3     sort the graphs in **G** by upper bounds descendingly;
4     **foreach** *graph* $G \in$ **G** **do**
5         **if** $\tau \geq G_{up}$ **then** Break;
6         $\rho = \rho_{max} = \phi$ ;
7         initial $\ell_{max}$ with $\tau$ random graphs s.t. $|\ell_{max}|=\tau$;
8         SEARCHPIVOT($G$, $\rho$, **G**, $n_1$, $\rho_{max}$, $\ell_{max}$);
9         **if** $\rho_{max} \neq \phi$ **then**
10             update $G_{lo}$, $G_{up}$, and $\tau$ all as $|\ell_{max}|$;
11             $\rho_{best} = \rho_{max}$;
12             $\ell_{best} = \ell_{max}$;
13         **else**
14             $G_{up} = \tau$;

15     **return** $\ell_{best}$;
16 **end**

---

by more than $\tau$ graphs. For this purpose, it initializes $\ell_{max}$ with $\tau$ random graphs such that the local threshold $|\ell_{max}| = \tau$ (Lines 6-8). In this way, only if the maintained path $\rho$ is shared by more than $\tau$ graphs (i.e., $|\ell| > \tau$), SEARCHPIVOT will be recursively invoked and $\ell_{max}$ will be updated . Then, if SEARCHPIVOT finds a pivot path $\rho_{max}$, it updates $\rho_{best}$ and $\ell_{best}$ as $\rho_{max}$ and $\ell_{max}$ respectively. The lower bound $G_{lo}$, upper bound $G_{up}$, and the largest lower bound $\tau$ are all updated to $|\ell_{max}|$ (Lines 10-12). Note that SEARCHPIVOT may update the lower bounds of the other graphs. However, in this case, $\tau$ is still the largest lower bound as none of the updated lower bounds can exceed $|\ell_{max}|$. If SEARCHPIVOT does not find the pivot path, it means the pivot path in $G$ cannot be shared by more than $\tau$ graphs. Thus we update $G_{up}$ as $\tau$ (Line 14). Similarly, in this case, none of the updated lower

bound can be larger than $\tau$ and $\tau$ remains the largest lower bound. Finally, when the stop condition is satisfied, $\rho_{max}$ must be the best pivot path and $\ell_{max}$ must be the largest group and thus get returned (Line 15).

*Theorem 1:* Let $\Sigma_1, \cdots, \Sigma_m$ be the ordered replacement groups generated by the algorithm UNSUPERVISEDGROUP-ING, where $|\Sigma_1| < \cdots < |\Sigma_m|$. The algorithm GENER-ATENEXTLARGESTGROUP will return $\Sigma_i$ at its $i^{th}$ invocation.

## VII. IMPLEMENTATION DETAILS

### A. Applying Approved Groups

Once a replacement $\varphi$ is approved, we backtrack all the value pairs that generate $\varphi$ and make the change (i.e., replace one value with the other one). For this purpose, for each candidate replacement lhs $\rightarrow$ rhs, we build a *replacement set*, denote as $L[\text{lhs} \rightarrow \text{rhs}]$, to keep all the places where the replacement is generated from. In addition, after updating a value, the replacements generated from the value may change. For example, consider the three values $v_1 = r_1[\text{Name}]$, $v_2 = r_2[\text{Name}]$, and $v_3 = r_3[\text{Name}]$ in Table 1. They will generate 6 replacements. Suppose the replacement $v_1 \rightarrow v_2$ is approved and $v_1$ is replaced by $v_2$. Then, the replacement $v_1 \rightarrow v_3$ will become $v_2 \rightarrow v_3$. Moreover, the replacement $v_2 \rightarrow v_1$ no longer exists. Thus we also need to update the replacement sets after making changes to values.

**Building Replacement Sets.** Let $v_j^i$ be the cell value at the $i^{th}$ row and $j^{th}$ column in the given clusters. For each value pair $v_j^i$ and $v_k^i$ in the same cluster, except generating two candidate replacements $v_j^i \rightarrow v_k^i$ and $v_k^i \rightarrow v_j^i$, we also append an entry $(i, j)$ to $L[v_j^i \rightarrow v_k^i]$ and another entry $(i, k)$ to $L[v_k^i \rightarrow v_j^i]$.

**Updating Replacement Sets.** For each approved replacement lhs $\rightarrow$ rhs, if the users decide to replace lhs with rhs, for each entry $(i, j)$ in $L[\text{lhs} \rightarrow \text{rhs}]$, we replace the value $v_j^i$ (it must be lhs) with rhs. In addition, for each value $v_k^i$ within the same cluster as $v_j^i$, we update the replacement sets as follows. We remove the entry $(i, j)$ from $L[\text{lhs} \rightarrow v_k^i]$ and the entry $(i, k)$ from $L[v_k^i \rightarrow \text{lhs}]$, if $v_k^i$ is not identical to lhs. Moreover, we add the entry $(i, j)$ to $L[\text{rhs} \rightarrow v_k^i]$ and the entry $(i, k)$ to $L[v_k^i \rightarrow \text{rhs}]$, if $v_k^i$ is not identical to rhs. Note if a replacement set becomes empty in this process, which indicates the corresponding replacement no longer exists, we remove the corresponding replacement from $\Phi$. Since rhs must be an existing value in the give clusters, no new candidate replacements will be generated in this process. Similarly, in the case the users decide to replace rhs with lhs, we conduct the above process in the other way around.

### B. Refine Groups by Structures

We observe that using current DSL some replacements that share a common transformation may look very different syntactically. In this case, it is hard for the users to make a single decision. To alleviate this problem, we propose to refine the groups by their *structures*. The candidate replacements in $\Phi$ are grouped together only if they share both the same transformation and the same structure.

In general, the structure of a replacement is acquired by uniquely mapping the two sides of the replacement to two sequences of pre-defined character classes (e.g., numeric and lowercase character classes).

Formally, the *structure* of a replacement $\varphi$, denoted by STRUC($\varphi$), is based on decomposing each side of a replacement into a sequence of terms, drawn from the following:

● *Regex-based terms*:

(1) *Digits*: $\mathsf{T}_d$ = [0-9]+    (2) *Lowercase letters*: $\mathsf{T}_l$ = [a-z]+
(3) *Whitespaces*: $\mathsf{T}_b$ = \s+ (4) *Capital letters*: $\mathsf{T}_C$ = [A-Z]+

● *Single character terms*:

(5) The character cannot be expressed by regex-based terms, e.g., $\mathsf{T}_-$ for the character '-'

Clearly, each character in any string will fall in *one and only one* of the above terms, such that the string can be uniquely represented. Next we show how to map a replacement to its structure. Initially, the structure of $\mathbf{s}$, STRUC($\mathbf{s}$), is empty. We sequentially visit each character $\mathbf{s}[i]$ in $\mathbf{s}$ for $i \in [1, |\mathbf{s}|]$. If $\mathbf{s}[i]$ does not belong to any of the categories 1–4 above, i.e., $\mathbf{s}[i]$ is a single character term, we append $\mathsf{T}_{\mathbf{s}[i]}$ to STRUC($\mathbf{s}$); otherwise, suppose $\mathbf{s}[i]$ belongs to the category $x$ ($x \in [1,4]$), we append $\mathsf{T}_x$ ($\mathsf{T}_d$, $\mathsf{T}_l$, $\mathsf{T}_C$ or $\mathsf{T}_b$ depending on $x$) to STRUC($\mathbf{s}$) and skip all the consecutively subsequent characters in the same category. Finally, we obtain the structure STRUC($\mathbf{s}$) of $\mathbf{s}$. For example, the structures of $\mathbf{s} = $ 9 and $\mathbf{t} = $ 9th are respectively STRUC($\mathbf{s}$) = $\mathsf{T}_d$ and STRUC($\mathbf{t}$) = $\mathsf{T}_d\mathsf{T}_l$.

*Definition 4:* Two replacements $\varphi_1 : \text{lhs}_1 \rightarrow \text{rhs}_1$ and $\varphi_2 : \text{lhs}_2 \rightarrow \text{rhs}_2$ are structurally equivalent, denoted by STRUC($\varphi_1$) $\equiv$ STRUC($\varphi_2$), iff. STRUC($\text{lhs}_1$) = STRUC($\text{lhs}_2$) and STRUC($\text{rhs}_1$) = STRUC($\text{rhs}_2$).

As it is less time consuming to get the structure of a replacement than calculating the pivot path, we first group the replacements in $\Phi$ by their structures. Specifically, for each replacement $\varphi$ in $\Phi$, we compute its structure STRUC($\varphi$). All replacements in $\Phi$ are then partitioned into disjoint groups based on *structure equivalence*. For example, the two replacements 9 $\rightarrow$ 9th and 3 $\rightarrow$ 3rd will be grouped together, as they have the same structure $\mathsf{T}_d \rightarrow \mathsf{T}_d\mathsf{T}_l$. By the end, we have a set of structure groups $\Phi_1, \Phi_2, \cdots$, which is a partition of $\Phi$. Then for each structure group $\Phi_i$, we invoke UNSUPERVISEDGROUPING($\Phi_i$) as discussed in Section 5 to refine it into disjoint groups. To support the incremental grouping technique as discussed in Section 6, for each replacement $\varphi \in \Phi_i$, we use the structure group size $|\Phi_i|$ to initialize its upper bound. Then, whenever the first time a replacement in a structure group $\Phi_i$ is visited in Algorithm 7, we invoke PREPROCESSING($\Phi_i$) to build graphs and inverted index and recalculate tighter upper bounds. The rest remains the same.

## VIII. EXPERIMENTS

The goal of the experiments is to evaluate the effectiveness and efficiency of our proposed unsupervised methods for standarizing variant values and golden record construction.

**Datasets.** We used the following three real-world datasets.

TABLE V
THE DATASET DETAILS

| | AUTHORLIST | ADDRESS | JOURNALTITLE |
|---|---|---|---|
| avg/min/max cluster size | 26.9/1/159 | 5.8/1/1196 | 1.8/1/203 |
| # of distinct value pairs | 51,538 | 80,451 | 81,350 |
| variant value pairs % | 26.5% | 18% | 74% |
| conflict value pairs % | 73.5% | 82% | 26% |

● AUTHORLIST[3] contains 33,971 book records with 1,265 clusters identified by the ISBN. Typical attributes include book name, author list, ISBN, and publisher. We used the author list attribute in the experiment, which contains 51,538 distinct value pairs. See more dataset details in Table V.
● JOURNALTITLE[4] contains 55,617 records concerning scientific journals. Attributes include journal title and ISSN. We clustered the journals by their ISSN numbers, resulting in 31,023 clusters. We used the journal title attribute in the experiment, which contains 80,451 distinct value pairs.
● ADDRESS[5] contains 17,497 funding application records. Attributes include the applicant who started the application and the legal name, address, and Employer Identity Number (EIN) of the organization which applied for the funding. We clustered the applications by the EIN and used the address attribute in the experiment, which resulting in 3,038 clusters and 81,350 distinct value pairs.

**Setup.** We implemented our methods in C++, compiled using g++4.8, and conducted experiments on a server with 64 Intel Xeon CPU E7-4830 @2.13GHz and 128 GB memory.

**Metrics.** To evaluate the efficiency of our unsupervised methods, we report the runtime for generating the groups. To evaluate the effectiveness, we report the *precision*, *recall*, and *Matthews correlation coefficient* (MCC) of standardizing variant values. Specifically, we first randomly sampled 1000 non-identical value pairs for each dataset and manually labeled each value pair as variant value pairs (i.e., they refer to the same value) or conflict value pairs (i.e., they refer to different values). We then ran our algorithm on the three datasets. After confirming a certain number of replacement groups generated by our methods and applying the approved ones to update the clusters, we checked the 1000 sample value pairs. Then true positives are the variant value pairs that become identical after updating, false negatives are the variant value pairs that remain non-identical after updating, false positives are the conflict value pairs that become identical after updating, and true negative are the conflict value pairs that remain non-identical after updating. We count the numbers of true positives (TP), false negatives (FN), false positives (FP), and true negatives (TN) and have precision as $\frac{TP}{TP+FP}$, recall as $\frac{TP}{TP+FN}$, and
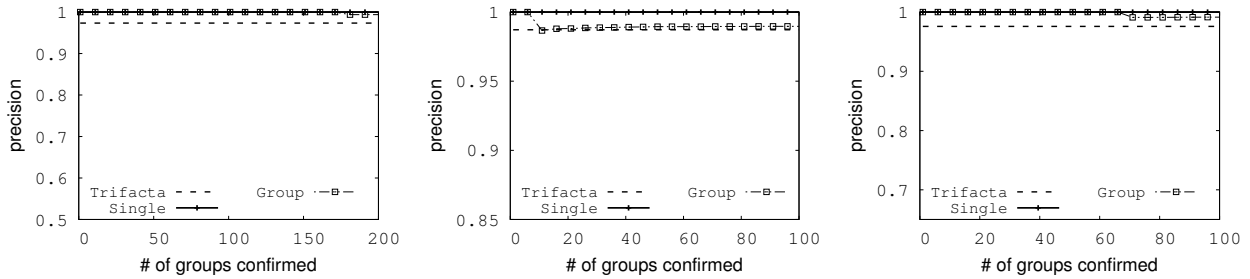
$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}.$$

MCC returns a value in $[-1, 1]$. The larger the better. We did not use the F1-score as the sizes of the positive class and negative class were quite different, which has bias to the precision or recall [18]. The MCC is known to be a balanced metric even if the classes are of very different sizes [4].

[3]http://www.lunadong.com/fusionDataSets.htm
[4]https://rayyan.qcri.org/
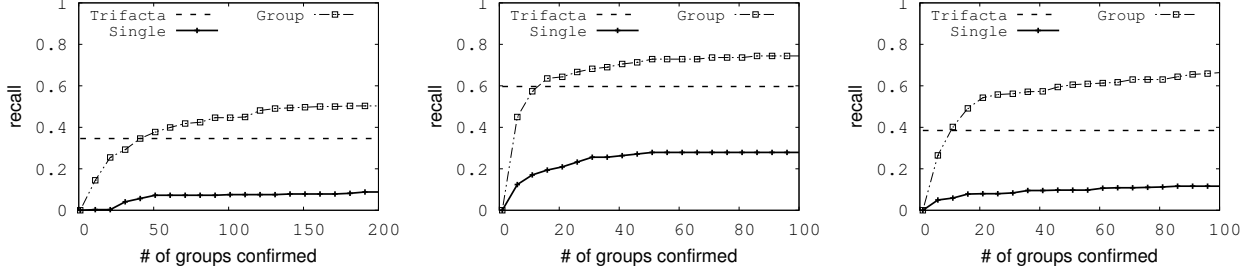[5]https://catalog.data.gov/

(a) AuthorList     (b) Address     (c) JournalTitle

Fig. 6. The precision of standardizing variant values by confirming a certain number of replacement groups
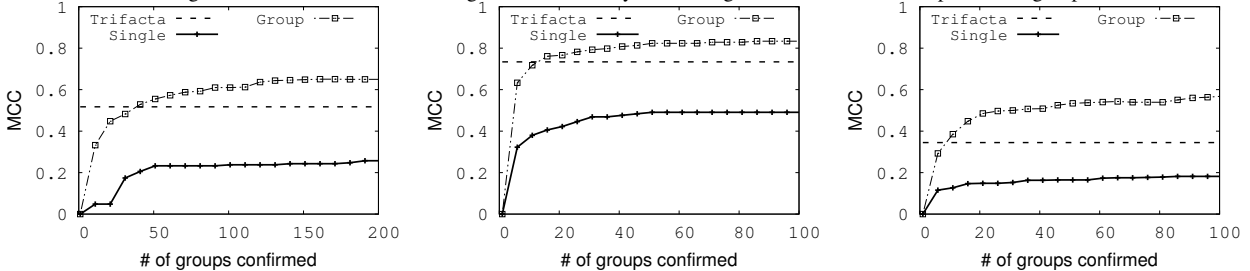


(a) AuthorList     (b) Address     (c) JournalTitle

Fig. 7. The recall of standardizing variant values by confirming a certain number of replacement groups



(a) AuthorList     (b) Address     (c) JournalTitle

Fig. 8. The MCC of standardizing variant values by confirming a certain number of replacement groups

### A. Effectiveness of Standardizing Data

We implemented two methods for standardizing variant values. (i) Single does not group the candidate replacements – each candidate replacement will be a group by itself. (ii) Group groups the candidate replacements in Φ by our proposed unsupervised methods: candidate replacements share the same pivot path (which corresponds to a transformation program) and structure will be grouped together.

We used Trifacta as our baseline method. Trifacta is a commercial data wrangling tool derived from DataWrangler [19]. It can apply some syntactic data transformations for data preparation, such as the regex-based replacing and substring extracting. Specifically, for each of the three datasets, we asked a skilled user to spend 1 hour on standardizing the dataset using Trifacta. Note that the user spent less than 20 minutes evaluating the groups in Single and Group in any of the experiments. Eventually, the user wrote 30-40 lines of wrangler code. For example, the following two lines of code were written to deal with groups C and E in Table 4.

REPLACE *with*: '' *on*: '({any}+)' and REPLACE *with*: '\$2 \$3. \$1' *on*: '({alpha}+), ({alpha}+) ({alpha}.)'

The first rule removes all the contents between a pair of parentheses, including the parentheses themselves, such as "(edt)" and "(author)". The second rule changes the name formats. Note that many string transformation learning methods and tools have been proposed in recent years. How-

ever, all of them are semi-supervised and cannot be used or adapted for our problem. See more details in Section IX.

Figures 6, 7, and 8 show the results, where the $x$-axis represents the number of groups confirmed by a human and $y$-axis represents the corresponding precision, recall, and MCC of standardizing variant values as previously defined. The dotted lines are the results of the baseline Trifacta. With regard to recall, Group consistently achieved the best performance. Specifically, Group surpassed the recall of Trifacta and Single by up to 0.3 and 0.5 respectively. For example, on the JOURNALTITLE dataset, the recall of Group, Trifacta, and Single were respectively 0.66, 0.38, and 0.12. This is because, compared with the one-by-one verification in Single, the batch confirmation in Group is more effective in standardizing more data. For Trifacta, the users had to observe the data and write code. The code only covers a fraction of the data, whereas our unsupervised method judiciously presents the most frequent and 'profitable' groups for the user to verify.

All the methods achieved very high precision as they all had a human in the loop. Specifically, Single achieved 100% precision, while Group and Trifacta achieved precision above 99% and 97%. This is because the one-by-one checking of Single is more fine-grained than the batch verification of Group, while Trifacta applied the code globally, which may introduce some errors. Nevertheless, the batch verification in Group and the human-written code in Trifacta were very

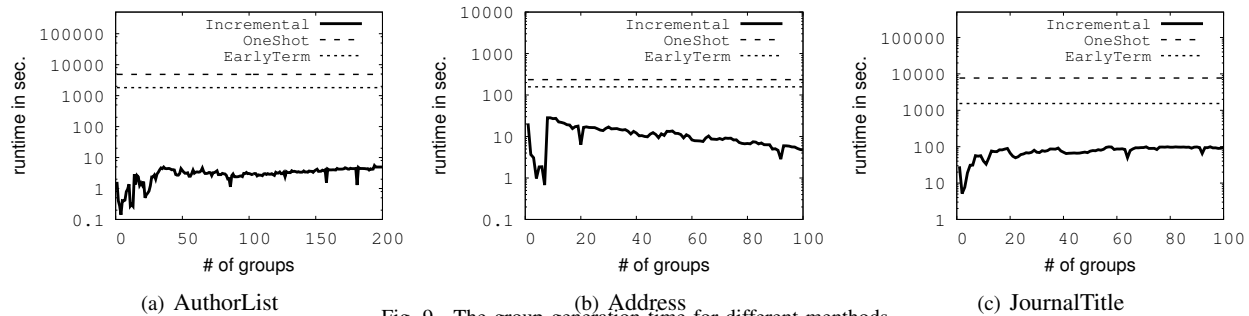| | | | |
|---|---|---|---|
| (a) AuthorList | (b) Address | (c) JournalTitle |

Fig. 9. The group generation time for different menthods

effective with regard to precision. Overall, Group achieved the best MCC. It outperformed Trifacta and Single by up to 0.2 and 0.4 respectively. For example, on JOURNALTITLE, the MCC of Group, Trifacta, and Single was respectively 0.57, 0.34, and 0.18, for the same reasons as discussed above.

Note that for all three datasets, the user spent less than 20 minutes in confirming the groups. Though Single took a little less human time than Group, its performance was much worse than that of Group as discussed above. In total, the user approved 70, 39, and 22 groups in Group out of the 200, 100, and 100 groups presented in AUTHORLIST, ADDRESS, and JOURNALTITLE. The denied groups were mostly because of the logic. For example, one group in AUTHORLIST transposes the authors' order and thus got denied.

The evaluation is based on only one user. In the future, we will conduct larger scale experiments. Existing methods [3], [15], [17] for string transformation are all semi-supervised, i.e., they need user-provided examples. Moreover, they are limited in learning the string transformation from homogeneous data, one at a time. In contrast, our method is unsupervised. We generate a large number of potentially dirty examples (candidate replacements) from the heterogeneous data and learns the string transformations (replacement groups) all at once. As the input data in entity consolidation usually come from different sources with different formats and thus are heterogeneous, the semi-supervised methods cannot be used or adapted for entity consolidation and we did not compare with them.

*B. Efficiency of the Grouping Algorithms*

In this section, we evaluate the efficiency of our grouping methods. We implemented three methods. (i) OneShot uses the vanilla UNSUPERVISEDGROUPING method to generate groups as discussed in Section V-A. (ii) EarlyTerm improves OneShot by the two early termination techniques as discussed in Section V-B. (iii) Incremental uses our incremental grouping method to generate groups as discussed in Section VI. We reported the group generation time for these methods. Figure 9 shows the results. In the figure, the two dotted lines for OneShot and EarlyTerm show their upfront costs. The solid line for Incremental gives the runtime of GENER-ATENEXTLARGESTGROUP at each invocation.

We can see from the figure that Incremental achieved the best performance. It improved the upfront cost of EarlyTerm by up to 3 orders of magnitude, while EarlyTerm outperformed OneShot by 2-10 times. For example, for the AU-THORLIST dataset, the upfront cost for OneShot, EarlyTerm, and Incremental were respectively 4900 seconds, 1800 seconds, and 1.6 seconds. This is because the two optimizations

TABLE VI
PRECISION IMPROVEMENT FOR MC

| | AUTHORLIST | ADDRESS | JOURNALTITLE |
|---|---|---|---|
| before | .51 | .32 | .335 |
| after | .65 | .47 | .840 |

in EarlyTerm can avoid a lot of unnecessary invocations of SEARCHPIVOT in finding the pivot path compared to OneShot. In addition, Incremental only generates the largest group at each time and thus can skip many unnecessary candidate replacements in Φ. Note that all these three methods had the same effectiveness for standardizing variant values as they are guaranteed to produce the same groups.

*C. Improvement on Entity Consolidation*

In this section, we evaluate the effectiveness of our algorithm in assisting truth discovery. For this purpose, we collected ground truth for 100 random clusters for each dataset. For AUTHORLIST, we used the same manually created ground truth as the previous work [10]. For JOURNALTITLE and ADDRESS, we manually searched for the ISSN in www.issn.cc and the EIN in www.guidestar.org to create the ground truth for each cluster. We used the dataset without any normalization except converting all characters to lowercase.

We first used the majority consensus (MC) to generate the golden values for each cluster and then compared the golden values with the ground truth. If they refer to the same entity, we increase TP (true positive) by 1; otherwise, we increase FP (false negative) by 1. Note that if there are two values with the same frequency, MC could not produce a golden value. Next we processed the original dataset with our algorithm and reran MC to create the golden values. We reported the precision before and after using our techniques. Table VI shows the results. We observe that our method indeed helped improve the precision of MC. In particular, on JOURNALTITLE, MC produced a precision of 33.5% before using our algorithm. After processing JOURNALTITLE with our algorithm, MC produced a precision of 84%, which is an improvement of over 40%. This is attributed to our effective variant value standardizing method, which correctly consolidated most of the duplicate values. On the other datasets, the improvement was less dramatic but still significant.

## IX. RELATED WORK

**String Transformations.** FlashFill [15] and BlinkFill [23] proposed to use program synthesis techniques [24] to learn a consistent transformation in a pre-defined DSL from a few user-provided input-output examples. DataXFormer [2] proposes to search string transformations from web tables, webforms, and knowledge bases based on the user-provided examples. Similarly, He et al. [17] developed a search engine

to find transformation programs from large scale codebases. Arasu et al. propose to learn string transformation rules from given examples [3]. DataWrangler [19] (*a.k.a.* Trifacta) has limited string transformation functionality such as regex-based replacing, string splitting, substring extraction, etc. Tao et al. [26] studied synonym discovering in similarity join.

**Entity Consolidation.** Entity consolidation aims at merging duplicate records [6], [11]. Entity consolidation is typically user-driven. For example, Swoosh [5] provides a unified interface that relies on the users to define the Merge function to specify how to merge two duplicate records. The conventional wisdom for entity consolidation is to use a Master Data Management (MDM) product [1]. MDM systems include a match-merge component, which is based on a collection of human-written rules. However, it is well understood that MDM solutions do not scale to complex problems, especially ones with large numbers of clusters and records.

**Truth Discovery and Data Fusion.** Truth discovery and data fusion [27], [22], [10], [12], [30], [21] can be used for entity consolidation. Given a set of claims made by multiple sources on a specific attribute, truth discovery and data fusion decide whether each claimed value is true or false and compute the reliability of each source. Solutions to these problems include models that use prior knowledge about the claims or source reputation [30], methods that consider the trustworthiness of a source as a function of the belief in its claims and the belief score of each claim as a function of the trustworthiness of the corresponding sources [22], [21], methods that consider other aspects such as source dependencies and truth evolution [10], [12]. In addition, there are approaches that try to resolve data conflicts by optimizing data quality criteria, such as data currency [14] and data accuracy [7], which selects the most recent value and the most accurate value, respectively. These works solve a different problem. They can be used to compute golden records. However, standardizing the variant values to the same canonical format using our method before applying them can improve their performance.

## X. CONCLUSIONS

In this paper, we proposed an unsupervised string transformation learning method for entity consolidation. Instead of directly applying existing solutions for entity consolidation, we first standardize the variant data. Specifically, we first enumerate the attribute value pairs in the same cluster. Then, we employ an unsupervised method to group value pairs that can be transformed in the same way. Finally we confirm the groups with a human and apply transformations in the approved groups to standardize the variant data. Experiments on real-world datasets show that our solution can effectively standardize variant values and significantly improve the performance versus a state of the art data wrangling tool. However, more research are needed for this work. Some open problems are to deal with data types other than strings and other languages, support data updates, and utilize source information.

## REFERENCES

[1] Informatica. https://www.informatica.com.
[2] Z. Abedjan et al. Dataxformer: Leveraging the web for semantic transformations. In *CIDR*, 2015.
[3] A. Arasu, S. Chaudhuri, and R. Kaushik. Learning string transformations from examples. *PVLDB*, 2(1):514–525, 2009.
[4] P. Baldi, S. Brunak, Y. Chauvin, C. A. F. Andersen, and H. Nielsen. Assessing the accuracy of prediction algorithms for classification: an overview. *Bioinformatics*, 16(5):412–424, 2000.
[5] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 18(1):255–276, 2009.
[6] J. Bleiholder and F. Naumann. Data fusion. *ACM Comput. Surv.*, 41(1):1:1–1:41, Jan. 2009.
[7] Y. Cao, W. Fan, and W. Yu. Determining the relative accuracy of attributes. In *SIGMOD*, pages 565–576, 2013.
[8] S. Das et al. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *SIGMOD*, pages 1431–1446, 2017.
[9] D. Deng et al. The data civilizer system. In *CIDR*, 2017.
[10] X. L. Dong, L. Berti-Equille, and D. Srivastava. Truth discovery and copying detection in a dynamic world. *PVLDB*, 2(1):562–573, 2009.
[11] X. L. Dong and F. Naumann. Data fusion: Resolving data conflicts for integration. *Proc. VLDB Endow.*, 2(2):1654–1655, Aug. 2009.
[12] X. L. Dong, B. Saha, and D. Srivastava. Less is more: Selecting sources wisely for integration. *PVLDB*, 6(2):37–48, 2012.
[13] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.
[14] W. Fan, F. Geerts, N. Tang, and W. Yu. Inferring data currency and consistency for conflict resolution. In *ICDE*, 2013.
[15] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011.
[16] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.
[17] Y. He, X. Chu, K. Ganjam, Y. Zheng, V. R. Narasayya, and S. Chaudhuri. Transform-data-by-example (TDE): an extensible search engine for data transformations. *PVLDB*, 11(10):1165–1177, 2018.
[18] L. A. Jeni, J. F. Cohn, and F. D. la Torre. Facing imbalanced data-recommendations for the use of performance metrics. In *ACII*, pages 245–251, 2013.
[19] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Wrangler: interactive visual specification of data transformation scripts. In *SIGCHI*, pages 3363–3372, 2011.
[20] P. Konda et al. Magellan: Toward building entity matching management systems. *PVLDB*, 9(12):1197–1208, 2016.
[21] Q. Li, Y. Li, J. Gao, B. Zhao, W. Fan, and J. Han. Resolving conflicts in heterogeneous data by truth discovery and source reliability estimation. In *SIGMOD*, pages 1187–1198, New York, NY, USA, 2014. ACM.
[22] T. Rekatsinas, M. Joglekar, H. Garcia-Molina, A. G. Parameswaran, and C. Ré. Slimfast: Guaranteed results for data fusion and source reliability. In *SIGMOD*, pages 1399–1414, 2017.
[23] R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *PVLDB*, 9(10):816–827, 2016.
[24] A. Solar-Lezama. The sketching approach to program synthesis. In *APLAS*, 2009.
[25] M. Stonebraker et al. Data curation at scale: The data tamer system. In *CIDR*, 2013.
[26] W. Tao, D. Deng, and M. Stonebraker. Approximate string joins with abbreviations. *PVLDB*, 11(1):53–65, 2017.
[27] D. A. Waguih and L. Berti-Equille. Truth discovery algorithms: An experimental evaluation. *CoRR*, abs/1409.6428, 2014.
[28] J. Wang, G. Li, J. X. Yu, and J. Feng. Entity matching: How similar is similar. *PVLDB*, 2011.
[29] S. Whang and H. Garcia-Molina. Entity resolution with evolving rules. *PVLDB*, 3(1):1326–1337, 2010.
[30] X. Yin, J. Han, and S. Y. Philip. Truth discovery with multiple conflicting information providers on the web. *TKDE*, 20(6):796–808, 2008.