Toward Data Cleaning with a Target Accuracy: A Case Study for Value Normalization

Adel Ardalan
University of Wisconsin-Madison
adel@cs.wisc.edu

Derek Paulsen
University of Wisconsin-Madison
dpaulsen2@wisc.edu

Amanpreet Singh Saini
University of Wisconsin-Madison
saini5@wisc.edu

Walter Cai
University of Washington
walter@cs.washington.edu

AnHai Doan
University of Wisconsin-Madison
anhai@cs.wisc.edu

ABSTRACT

Many applications need to clean data with a target accuracy. As far as we know, this problem has not been studied in depth. In this paper we take the first step toward solving it. We focus on value normalization (VN), the problem of replacing all string that refer to the same entity with a unique string. VN is ubiquitous, and we often want to do VN with 100% accuracy. This is typically done today in industry by automatically clustering the strings then asking a user to verify and clean the clusters, until reaching 100% accuracy. This solution has significant limitations. It does not tell the users how to verify and clean the clusters. This part also often takes a lot of time, e.g., days. Further, there is no effective way for multiple users to collaboratively verify and clean. In this paper we address these challenges. Overall, our work advances the state of the art in data cleaning by introducing a novel cleaning problem and describing a promising solution template.

1 INTRODUCTION

Data cleaning (DC) has been a long-standing challenge in the database community. Many DC problems have been studied, such as cleaning with a budget, cleaning to satisfy constraints but minimize changes to the data, etc. Recently, however, we have seen another novel DC problem in industry: cleaning with a target accuracy, e.g., with at least 95% precision and 90% recall. While pervasive, this problem appears to have received little attention in the research community.

In this work, we take the first step toward solving this problem. We focus on value normalization (VN), the problem of replacing all strings (in a given set) that refer to the same real-world entity with a unique string. VN is ubiquitous, and industrial users often want to do VN with 100% accuracy.

Example 1.1. To enable product browsing by brand on walmart.com, the business group at WalmartLabs asks the IT group to normalize the brands, e.g., converting those in Figure 1.a into those in Figure 1.b. If some brands are not normalized correctly, then customers may not find those products, resulting in revenue losses. So the business group asks the IT group to ensure that the brands are normalized with 100% accuracy.

Many enterprise customers of Informatica (which sells data integration software) also face this problem, in building business glossaries, master data management, and knowledge graph construction. In general, if even a small amount of inaccuracy in VN can cause significant problems for the target application, then the business group will typically ask the IT group to help perform VN with 100% accuracy.

In response, the IT group typically employs an algorithm to cluster the strings, then asks a user to verify and clean the clusters.

Consider the five brands in Figure 2.a. The IT group applies an algorithm to produce two clusters $c_1$ and $c_2$ (Figure 2.b). A user $U$ manually verifies and cleans the clusters, by moving “Vizio Corp” from cluster $c_1$ to $c_2$, producing the two clusters $c_3$ and $c_4$ in Figure 2.c. Finally, $U$ replaces each string in a cluster with a canonical string, producing the VN result in Figure 2.d.

Typically, a data scientist performs the "machine" part that clusters the strings, then a data analyst performs the "human" part that verifies and cleans the clusters. The IT group assures the business group that the resulting output is 100% accurate because a data analyst has examined it (assuming that he/she does not make mistakes).

While popular, the above solution has significant limitations. Typical limitations include:

First, there is no precise procedure that a user can follow to execute the "human" part. So users often verify and clean in an ad-hoc, suboptimal, and often incorrect fashion. This makes it impossible to understand the assumptions under which the solution reaches 100% accuracy and to formally prove it.

Second, the "human" part often incurs a huge amount of time, e.g., days. In contrast, the "machine" part often takes mere minutes. (In most cases that we have seen, users verified and cleaned using Excel, in a slow and tedious process.) So it is critical to develop a better solution and GUI tool to minimize the time of the "human" part.

Finally, it is difficult for multiple users to collaboratively verify and clean, even though this setting commonly occurs in practice.

In this paper we develop Winston, a solution for the above challenges. (In the movie "Pulp Fiction", Winston Wolfe is the fixer who cleans up messes made by other gangsters.) We first

Figure 1: An example of normalizing product brands

Figure 2: A popular solution in industry to perform VN with 100% accuracy.
define a set of basic operations on a GUI for users, e.g., selecting a value, verifying if a cluster is clean, merging two clusters, etc. Then we provide precise procedures involving these actions that users can execute to verify/clean clusters. We prove that if users execute these actions correctly, then the output has 100% accuracy.

To minimize the time of the "human" part, we adopt an RDBMS-style solution. Specifically, we compose the GUI operations with clustering algorithms to form multiple "machine-human" plans, each executes the VN pipeline end-to-end. Next, we estimate the total time a user must spend per plan, select the plan with the least estimated time, execute its machine part, then show the output of that part to the user so that he/she can verify and clean it using a GUI (following the sequence of user operations that this plan specifies).

Finally, we show how to extend our solution to effectively divide the verification and cleaning work among multiple users. Our solution appears highly effective. Section 8 shows that using the existing solution, a single user needs 29 days, 4.4 years, and 11.5 years to verify/clean 100K, 500K, and 1M strings, respectively. Winston drastically reduces these times to just 13 days, 9.6 months, and 1.3 years, using 1 user, and to 4.25 days, 2.2 months, and 3.5 months, using 3 users.

In summary, we make the following contributions:

- We formally define the novel data cleaning problem of VN with 100% accuracy. As far as we know, this paper is the first to study this problem in depth.
- We propose Winston, a novel RDBMS-style solution. Winston defines complex human operations and optimizes the human time of a plan. This is in contrast to traditional RDBMSs which define machine operations and optimize machine time.
- We describe extensive experiments (comparing Winston to a tool in a company, to the popular open-source tool OpenRefine, and to state-of-the-art string matching and entity matching solutions) that show the promise of our approach.

Overall, our work advances the state of the art in data cleaning by introducing a novel cleaning problem and describing a promising solution template. It also advances the state of the art in human-in-the-loop data analytics (HILDA) by showing that it is possible to develop an RDBMS-style solution to HILDA, by defining complex human operations, combining them to form plans, and selecting the plan with the lowest estimated human effort.

## 2 PROBLEM DEFINITION

In this section we define the problem of VN with 100% accuracy, and examine when we can reach this accuracy under what conditions. We first define

**Definition 2.1 (Value normalization).** Let V be a set of strings \( \{v_1, \ldots, v_n\} \). Replace each \( v \in V \) with a string \( s(v) \) such that \( s(v^1) = s(v^2) \) if and only if \( v_1 \) and \( v_2 \) refer to the same real-world entity, for all \( v_1, v_2 \in V \).

This problem is often solved in two steps: (1) partition V into a set of disjoint clusters \( V' = \{V_1, \ldots, V_m\} \), such that two strings refer to the same real-world entity if and only if they belong to the same cluster; (2) replace all strings in each cluster \( V_i \) with a canonical string \( s_i \).

In this paper we will consider only Step 1, which tries to find the correct partitioning of V. Step 2 is typically application dependent (e.g., a common method is to select the longest string in a cluster \( V_i \) to be its canonical string, because this string tends to be the most informative one).

**Gold Partition & Accuracy of Partitions:** Let \( U \) be a user who will verify and clean the clusters. To do so, \( U \) must be capable of creating a "gold", i.e., correct, partition \( V^* = \{V_1^*, \ldots, V_k^*\} \), such that two strings refer to the same real-world entity if and only if they are in the same cluster. For example, the two clusters \( c_3 \) and \( c_4 \) in Figure 2.c form the gold partition for the set of strings in Figure 2.a.

Our goal is to find the gold partition \( V^* \). But the partition that we find may not be as accurate. We now describe how to compute the accuracy of any partition. First, we define

**Definition 2.2 (Match and non-match).** A match \( v_1 = v_2 \) means "\( v_1 \) and \( v_2 \) refer to the same real-world entity", and is correct if this is indeed true. \( v_1 = v_2 \) and \( v_1 = v_3 \) are considered the same match. We define a non-match \( v_1 \neq v_2 \) similarly.

**Definition 2.3 (Set of matches specified by a partition).** A cluster \( V_i \) specifies the set of matches \( M(V_i) = \{v_{p1} = v_{q1} | p \in V_i, q \in V_i, p \neq q\} \). Partition \( V = \{V_1, \ldots, V_n\} \) specifies the set of matches \( M(V^*) = \cup_{i=1}^m M(V_i) \).

For example, cluster \( c_1 \) in Figure 2.b specifies three matches: \( M(c_1) = \{\text{Sony = Sony Corp}, \text{Sony = Vizio Corp}, \text{Sony Corp = Vizio Corp}\} \). Cluster \( c_2 \) specifies one match: \( M(c_2) = \{\text{Vizio = Vizio Inc}\} \). These two clusters form a partition \( V_{12} \), which specifies the set of matches \( M(c_1) \cup M(c_2) \). The accuracy of a partition is then measured as follows:

**Definition 2.4 (Precision and recall of a partition).** Let \( V^* \) be the gold partition of a set of strings V. The precision of a partition \( V \) is the fraction of matches in \( M(V^*) \) that are correct, i.e., appearing in \( M(V^*) \), and the recall of \( V \) is the fraction of matches in \( M(V^*) \) that appear in \( M(V) \).

Given the gold partition \( V_{34} = \{c_3, c_4\} \) in Figure 2.c, the precision of partition \( V_{12} = \{c_1, c_2\} \) in Figure 2.b is 2/4 = 50%, and the recall is 2/4 = 50%.

**Actions & Their Verification Sets:** Henceforth, we use "action" and "operation" interchangeably. When a user \( U \) performs an action, \( U \) has implicitly verified a set of matches and non-matches, called a verification set. Formally,

**Definition 2.5 (User action and verification set).** We assume a GUI on which user \( U \) can perform a set of actions \( A = \{a_1, \ldots, a_r\} \). Each action \( a_i \) inputs data \( I_i \) and outputs data \( O_i \), both of which involve sets of strings in V. After correctly executing an action \( a_i \) on input \( I_i \), as a side effect, user \( U \) has implicitly verified a set \( Q(a_i, I_i) \) of matches and non-matches to be correct. We refer to \( Q(a_i, I_i) \) as a verification set.

To illustrate, suppose \( U \) has employed an algorithm to produce the partition \( \{d_1, d_2\} \) in Figure 3. Next, \( U \) uses a GUI to verify and clean these clusters. Call a cluster "pure" if all strings in it
We define \( Q \) Recall that the gold partition \( VN \) with 100% Accuracy: (because all three must be in the same gold cluster). Similarly, if \( v_i = v_j \) matches form a transitivity path from \( v_i \) to \( v_j \) is also correct (because all three must be in the same gold cluster). Similarly, if \( v_i = v_j \) and \( v_j \neq v_k \) are correct, then \( v_i \neq v_k \) is also correct.

**Definition 2.7 (Inferring matches).** We say that match \( v_i = v_j \) can be inferred from a verification set \( Q(G) \) if and only if there exists a sequence of strings \( v_{h1}, \ldots, v_{hl} \) such that the matches \( v_i = v_{h1}, v_{h1} = v_{h2}, \ldots, v_{hl} = v_j \) are in \( Q(G) \). We say that these matches form a transitivity path from \( v_i \) to \( v_j \). Similarly, we say that non-match \( v_i \neq v_j \) can be inferred from \( Q(G) \) if and only if there exists such a path, except that exactly one of the edges of the path is a non-match.

**VN with 100% Accuracy:** Recall that the gold partition \( V^* \) specifies a set of correct matches \( M(V^*) \). We say that it also specifies a set of correct non-matches \( N(V^*) \), which consists of all non-match \( v_i \neq v_j \) such that \( v_i = v_j \) is not a match in \( M(V^*) \). We define

**Definition 2.8 (Gold sequence of actions).** A sequence \( G \) of actions of user \( U \) is "gold" if and only if any match in \( M(V^*) \) or non-match in \( N(V^*) \) either already exists in the verification set \( Q(G) \) or can be inferred from \( Q(G) \).

It is not difficult to prove that executing a gold action sequence \( G \) will produce the gold partition \( V^* \). We now can define our problem as follows:

**Definition 2.9 (VN with 100% accuracy).** Let \( V \) be a set of strings. Let \( (X, Y) \) be a pair of machine/human algorithms, such that the machine part \( X \) can be executed on \( V \) to produce a partition \( V' \), then the human part \( Y \) can be executed by a user \( U \) on \( V' \) to produce a new partition \( V^+ \). Find \( X \) and \( Y \) such that (a) the action sequence executed by user \( U \) in part \( Y \) is a gold sequence, and (b) the total time spent by user \( U \) is minimized. Return the resulting partition \( V^+ \).

Thus, we reach 100% accuracy if the user executes a gold sequence \( G \) of actions. Then all correct matches and non-matches will have already been in the verification set of \( G \), or inferred from this verification set via match transitivity.

### 3 DEFINING THE HUMAN PART

As discussed, each VN plan \( (X, Y) \) consists of a machine part \( X \) and a human part \( Y \). In part \( X \) we apply an algorithm to the input strings to obtain a set of clusters \( V \), then in part \( Y \) we employ a user \( U \) to verify and clean \( V \). We now design part \( Y \); the next section designs part \( X \).

The key challenge in designing the human part \( Y \) is to ensure that it is easy for users to understand and execute, minimizes their effort, and is amenable to cost analysis. Toward these goals, we discuss the user setting, describe a solution called split and merge, then define a set of user operations that can be used to implement this solution.

We assume user \( U \) will work with a graphical user interface (GUI), using mouse and keyboard. \( U \) has a short-term memory (or STM for short). According to [36] each individual could remember \( 7 \pm 2 \) objects in his or her STM at each moment (a.k.a. Miller’s law). Thus we assume the STM capacity to be 7 objects. Finally, we assume that \( U \) can use paper and pen for those cases where \( U \) needs to keep track of more objects than can be fit into STM.

User \( U \) can clean the clusters output by the machine part in many different ways. In this paper, based on what we have seen users do in industry, we propose that \( U \) clean in two stages. The first stage splits the clusters recursively until all resulting clusters are "pure", i.e., each containing only the values of a single real-world entity (though often not all such values). The second stage then merges clusters that refer to the same entity.

**Example 3.1.** Suppose the machine part produces clusters 1-2 in Figure 4. The split stage splits cluster 1 into clusters 3-4, cluster 2 into clusters 5-6, cluster 6 into clusters 7-8, then cluster 8 into clusters 9-10 (see the solid arrows). The output of the split stage is the set of pure clusters 3, 4, 5, 7, 9, 10. The merge stage then merges clusters 3 and 5 into cluster 11, and clusters 4 and 9 into cluster 12 (see the dotted arrows). The end result is the set of clean clusters 11, 12, 7, 10.

### 3.1 The Split Stage

We now describe the split stage (Section 3.2 describes the merge stage). First, we define a dominating entity of a cluster \( c \) to be the one with the most values in \( c \) (henceforth we use "value" and "string" interchangeably). Formally,
Algorithm 1 Split Phase

Procedure Split(C)
Input: a set of clusters C = \{c_1, \ldots, c_p\}, output by machine
Output: a set of clean clusters D = \{d_1, \ldots, d_q\} s.t. \cup_c c \subseteq \cup_d d.
1: D \leftarrow \emptyset
2: for each cluster c \in C do D \leftarrow D \cup SplitCluster(c)
3: return D

Procedure SplitCluster(c)
Input: a cluster c
Output: a set of clean clusters G = \{g_1, \ldots, g_q\} \cup d \forall g \in g
1: if |c| = 1 then return \{c\}
2: if yes button \rightarrow at the end, user selects yes/no button
3: if yes button is selected then return \{c\}
4: if findCleanityValue(c) \rightarrow at the end, user knows c^* and g
   // or "mark values" button
5: if "clean merged cluster" button is selected then return \{c\}
6: return Merge(c) if g < 0.1 in this case
7: MarkValues(c, c^*, g)
   // at the end, user selects "create/merge new cluster"
   // or "create new cluster / clean old cluster" button
8: Move all marked values in c into a new cluster d
9: if "create/clean new cluster" button is selected then return \{c\}
10: return c \cup SplitCluster(d) if g \geq 0.5
11: else return SplitCluster(c) if \forall d \cup g < 0.5

Procedure MarkValues(c^*, c^*, g)
Input: a cluster c, dominating entity c^* and purity g of c
Output: a set of values in c will be selected by the user
1: Let L be the list of values in c, displayed on GUI.
2: if g > 0.5 then
3: for i = 1, \ldots, |c| do
4: if findTrack(c^*, i, g), if not markTrack(c^*, i, g) then select(c(i))
5: else for i = 1, \ldots, |c| do
6: if track(c^*, i, g), if markTrack(c^*, i, g) then select(c(i))

Definition 3.2 (Dominating entity). Let G be a partition of a cluster c into groups of values G_1, \ldots, G_p such that all values in each group refer to the same real-world entity and different groups refer to different entities. Then the dominating entity of c is the entity of the group with the largest size: G_k = arg max_{c \in G} |c_k|. Henceforth, we will use dom(c) (or e* when there is no ambiguity) to denote the dominating entity of c.

In Figure 4, dom(Cluster 1) and dom(Cluster 2) are Sony Corporation. Cluster 6 has three candidates; we break tie by randomly selecting one to be the dominating entity.

Let C be the set of clusters output by the machine part. Our key idea for the split stage is that if the machine part has been reasonably accurate, then any cluster c \in C is likely to be dominated by dom(c). If so, user U can clean c by moving all the values in c that do not refer to dom(c) into a new cluster d, then clean d, and so on.

Specifically, for each cluster c \in C, user U should (1) check if c is pure; if yes, stop; (2) otherwise find the dominating entity dom(c); (3) move all values in c that do not refer to dom(c) into a new cluster d; then (4) apply Steps 1-3 to cluster d (cluster c has become pure, so needs no further splitting). This recursive procedure will split the original cluster c into a set of pure clusters. It is relatively easy for human users to understand and follow, and as we will see in Section 5, it is also highly amenable to cost analysis.

Example 3.3. Given cluster 1 in Figure 4, user U splits it into the pure cluster 3, which contains only the values of the dominating entity Sony Corporation, and cluster 4, which contains all remaining values of cluster 1. A similar recursive splitting process applies to cluster 2. (Note that cluster 6 has three dominating-entity candidates, so we break tie randomly and select Dell to be the dominating entity.)

We now optimize the above procedure in three ways. First, if c is a singleton cluster, then we do not invoke the above splitting procedure, because c is already pure. Second, there are cases where the number of values referring to dom(c) is less than 50% of |c|. Formally, we define

Definition 3.4 (Cluster purity). The purity of a cluster c is the fraction of the values in c that refer to dom(c). Henceforth we will use p(c) (or a when there is no ambiguity) to denote the purity of c.

For example, in Figure 4, the purity of cluster 2 is 2/5 = 0.4 < 0.5. In such cases, instead of moving all values in c that do not refer to dom(c), as discussed so far, it is less work for the user to move the values that do refer to dom(c) into a new cluster d (e.g., for cluster 2, U should move "Sonny" and "SONY Corp", instead of the other three values).

Finally, if p(c) is below a threshold, currently set to 0.1, then c is very "mixed", with each entity having less than 10% of the values. In this case, we have found that instead of splitting c, it is often more effective to apply the Merge procedure described in Section 3.2 to c. This produces a set of pure clusters that are then fed to the merge stage.

Basic User Operations: To implement the above solution, we define the following five basic user operations:

- **focus(a):** User U moves his or her focus to a particular object a on the GUI or on the paper, such as a cluster, a value within a cluster, a GUI button, a number on the paper, etc. Intuitively, user U will shift his or her attention from one object to another on the GUI or the paper, and that incurs a certain amount of time. This operation is designed to capture this physical action (and its cost).

- **select(a):** User U selects an object a on the GUI (e.g., a cluster, a value, a GUI button, etc.) by moving the mouse pointer to that object and clicking on it, or pressing a keyboard button (e.g., page up, page down). This operation is designed to capture this physical action (and its cost).

- **match(x,y):** Given two values, or a value and a real-world entity (in U’s short-term memory), U determines if they refer to the same real-world entity.

- **isPure(c):** U examines cluster c to see if it is pure (i.e., if it is clean). Specifically, we assume the values in c is listed (e.g., on the GUI) as a list of values L. User U reads the first value of L, maps it to an entity e, then scans the values in the rest of L. As soon as U sees a value that does not refer to e, the cluster is not pure, U stops and returns false. Otherwise U exhausts L and returns true.

- **findDom(c):** finds the dominating entity dom(c) and the purity p(c) of a cluster c. If |c| \leq 7, the size of the short-term memory (STM), then U does this entirely in STM. Specifically, U scans the list of values in c, maps each value into an entity, and keeps track of the number of times U has encountered a particular entity. Then U returns the entity with the highest count q as the dominating one, and g/|c| as the purity of cluster c. If |c| > 7 then U proceeds as above, but uses paper and pen to keep track of the counts of the encountered entities.

The Split Procedure: Algorithm 1 describes Split, a procedure that uses the above five operations to implement the split stage. Split takes the set of clusters output by the machine part, then applies the SplitCluster procedure to each cluster. We distinguish two kinds of procedures: GUI-driven and human-driven. Split and SplitCluster are GUI-driven, i.e., executed by the computer. A GUI-driven procedure, e.g., SplitCluster, may call human-driven procedures e.g., isPure, findDom, then pass control to user U to
execute those procedures. To distinguish between the two, we underline the names of human-driven procedures.

Algorithm 1 shows SplitCluster handles the corner case of singleton clusters (Step 1), then calls isPure and asks user \( U \) to take over (Step 2). At the end of this procedure, \( U \) would have selected either "yes" or "no" button, indicating whether the cluster is pure or not. In the former case, SplitCluster terminates, returning the pure cluster (Step 3). Otherwise, it calls findDom (Step 4), and so on. Note that at the end of findDom, user \( U \) knows the dominating entity \( e \) and the purity \( \alpha \), but the computer does not know these. Hence these quantities (and all quantities that only \( U \) know) are shown as underlined, e.g., \( \underline{e}, \underline{\alpha} \).

3.2 The Merge Stage

Given a set of pure clusters output by the split stage, the merge state merges clusters that refer to the same entity. Clearly, from each cluster we can select just a single representative value (say the longest string), then merge those (if we know how to merge those, we can easily merge the original clusters). For example, in Figure 4 the split stage produces clusters 3, 4, 5, 7, 9, and 10. To merge them, it is sufficient to consider merging the values "Sony Corp", "Lg", "SONY Corp", "Dell", "LG", and "Apple". Henceforth we will consider this simpler problem of merging \( n \) values \( v_1, \ldots, v_n \).

Naively merging by considering all pair takes quadratic time. To address this problem, we propose a two-step process. First, \( U \) does one pass through the list of values to do a "local merging" that merges matching values that are near one another. This reduces \( n \). Then \( U \) does "global merging" that considers all pairs (of the remaining values). Both steps will exploit the parallel processing capability of short-term memory (STM). We now describe these two steps.

Local Merging: This step uses STM to merge matching values that are near one another. Specifically, first the set of values is sorted. Currently we use alphabetical sorting, because matching values often share the first few characters (e.g., IBM, IBM Corp). Figure 5.a shows such a sorted list \( L \) (ignoring the arrows for now).

Next, user \( U \) processes the values in \( L \) top down. For each value, \( U \) stores it and the associated entity in STM. For the sake of this example, assume STM can only store three such pairs. Figure 5.b shows a full STM after \( U \) has processed the first three values of the list. Then when processing the 4th value, "Garmin", \( U \) needs to evict the oldest pair from STM to make space for "Garmin" (see Figure 5.c).

Then when processing the 5th value, "Ge", \( U \) realizes that its entity, \( e_2 \), is already in STM, associated with a previous value "GE". So \( U \) links "Ge" with "GE", and replaces the value "GE" in STM with the new value "Ge" (see Figure 5.d). Next, "IBM" will be stored in STM, displacing "Gamevice" (Figure 5.e), and so on. At the end, \( U \) has linked together certain matching values (see the arrows in Figure 5.a).

Algorithm 2 describes local merging, which uses previously defined user operations focus(a) and select(a) (see Section 3.1), as well as the following new user operation:

- **memorize(v):** \( U \) maps the input value \( v \) to an entity \( e \), then memorizes, i.e., stores the pair \((e, v)\) in STM. Specifically, if a pair \((e, v)\) is already in STM, \( U \) replaces it with \((e, v)\), then exits, returning \((e, v)\) (see Line 2 in Algorithm 2). Otherwise, \( U \) adds the pair \((e, v)\) to STM, "kicking" the oldest pair out of STM to make space if necessary.

Global Merging: After local merging, the original list of values is consolidated, i.e., from each set of linked values we again select just a single representative value (e.g., the longest one). This produces a new shorter list, e.g., consolidating the list in Figure 5.a produces the shorter list in Figure 6.a (ignoring the arrow).

Let \( L = [v_1, \ldots, v_n] \) be this new shorter list. Naively, user \( U \) can compare \( v_1 \) with \( v_2, \ldots, v_n \), then compare \( v_2 \) with \( v_3, \ldots, v_n \), etc.

A better solution however is to exploit the parallel processing capability of STM: read multiple values, say \( v_1, \ldots, v_k \), into STM all at once, then compare them all in parallel in \( v_{k+1}, \ldots, v_n \), etc.

Example 3.5. Consider again the list in Figure 6.a. User \( U \) can read the first two values, "Big Blue" and "GE", into STM, then scan the rest of the values and match them with these two in parallel (using a GUI, see Figure 6.b). If there is a match, e.g., "IBM Corp" and "Big Blue", then \( U \) checks off the appropriate box (see Figure 6.b). At the end of the list, \( U \) pushes a button to link the matching values. Next, \( U \) reads into STM the next two values, "Gamevice" and "Garmin", then match "IBM Corp" with these two. \( U \) detects no more matches, thus wrapping up global merge. The system uses the results of both local and global merges to produce the final clusters shown in Figure 6.c.

In practice, even though STM can hold 7 objects [36], we found that users prefer to read only 3 values at a time into STM. First, 7 values often take up too much horizontal space on the GUI (especially if the strings are long), making it hard for users to comprehend. Second, users want to reserve some STM capacity to read and remember the values in the rows.

As a result, we currently use \( k = 3 \) in our global merge procedure. Appendix A describes this procedure, which uses user operations focus(a), select(a), memorize(v), as well as the following new user operation:

```
1. Big Blue
2. GE
3. Gamevice
4. Garmin
5. IBM Corp

(a) Big Blue GE
(b) Gamevice
(c) IBM Corp
```

![Figure 5: An example of local merging.](image_5)

![Figure 6: An example of global merging.](image_6)
Sony and "Sonny" into work on the same dataset. well for our purposes. Then we discuss why existing SM/EM algorithms do not work values that refer to the same entity (e.g., "LG" and "Lg", or "Sony" "mixed" (e.g., Figure 8.b), then $U$ would need to split them up into pure clusters, before merging them. This incurs far more mouse clicks and thus far more work.

Of course, we do not know when to stop HAC. To address this problem, we introduce multiple HAC variations, each stopping at a different time, then try to select a good one. Specifically, to cluster $n$ values, we consider $n$ HAC variations, where the $i$-th variation, denoted HAC($i$), limits the cluster size to at most $i$. In each iteration, HAC($i$) finds the two clusters $c$ and $d$ with the highest similarity score, then merges them if $|c \cup d| \leq i$. Otherwise, HAC($i$) finds the two clusters with the next highest score, and merges them if the resulting size is at most $i$, and so on. HAC($i$) terminates when it cannot find any more clusters to merge.

Example 4.2. Consider applying HAC(2) to the values in Figure 7.a. HAC(2) first forms cluster $c_1$, then $c_2$, exactly as the normal HAC. Then normal HAC goes on to form cluster $c_3$ in Figure 7.a, but HAC(2) cannot, because $c_1$'s size exceeds 2. Instead, HAC(2) find the next two clusters with the highest similarity score. Suppose these are the singleton clusters for "Sony Corp" and "Sony Inc." Then HAC(2) merges them to form cluster $c_3$ in Figure 7.b. At this point HAC(2) cannot form any more cluster, because any resulting cluster size would exceed 2. So it stops, returning the clusters in Figure 7.b as the output.

HAC(1) produces the smallest but cleanest clusters (as they are singletons). As we increase $i$, HAC($i$) tends to produce bigger but less clean clusters. Typically, there exists an $i^*$ such that HAC($i^*$)'s clusters are still so clean that they help user $U$, but HAC($i^*+1$)'s clusters are already "too dirty" to help (e.g., $U$ would need to split them extensively before her or she can merge). This roughly corresponds to the point where we want HAC to stop. HAC($i^*$) thus is the "best" HAC variation for the current data set.

To find HAC($i^*$), we pair HAC(1), ..., HAC($n$) with Split and Merge to form $n$ end-to-end plans. Sections 5 and 6 show how to estimate the costs of these plans and find the one with the least estimated cost.

### 2.3 Limitations of SM/EM Solutions

We are now in a position to explain why existing string matching (SM) and entity matching (EM) solutions do not work well in our context. (Section 8 shows experimentally that Winston with HAC outperforms these solutions.)

At the core, VN is an SM problem. So SM solutions can be used in the machine part. EM solutions can also be used, by limiting each entity to be a string. Many such solutions have been developed, e.g., TransER [47], Magellan [32], Falcon [13], Waldo [44] (see Section 9).
These SM/EM solutions (e.g., Magellan, Falcon) typically output a set of matches. One way to use them is to ask user $U$ to verify certain matches, then infer even more matches using match transitivity. For example, given 5 string $a, b, c, d, e$, suppose a solution outputs $a = b$ and $a = c$ as matches. If $U$ has verified these matches, then we can infer $b = c$ as another match. A recent work, TransER [47], exemplifies this approach. A serious problem, however, is that we cannot guarantee 100% recall, as shown experimentally in Section 8. For example, no user verification and match transitivity on the outputs $a = b, a = c$ can help us infer $d = e$ (assuming this is also a correct match). Thus, these solutions are not appropriate for Winston.

Another way to use existing SM/EM solutions is to cluster the input strings in a way that respects the output matches. The work [43] describes multiple ways to do this. Continuing with the above example, given the output matches $a = b, a = c$, we can cluster the five input strings into, say, 3 clusters $\{a, b, c\}, \{d\}, \{e\}$. User $U$ can verify/clean these clusters, as discussed in the human part. A serious problem here, however, is that this approach often produces large mixed clusters, which are very time consuming for $U$ to clean, as shown experimentally in Section 8. With HAC, we solve this problem by modifying HAC to stop early to produce clean clusters (see Section 4.1). But there is no obvious way to modify the clustering algorithms in [43] to stop early such that they produce relatively clean clusters and the quality of these clusters can be estimated (e.g., see Section 5).

The above works provide no GUI, or very basic inefficient GUIs for user feedback, e.g., Falcon and TransER ask users to label string pairs as match/non-match. A recent work, Waldo [44], considers a far more efficient GUI which displays 6 strings so that a user can cluster all of them in one shot. As such, its "human" part is more similar to ours. But its "machine" part considers a very different optimization problem: namely minimizing crowdsourcing cost (e.g., clustering 6 strings incurs the same monetary cost, regardless of which human user does it). Thus, it cannot be used in Winston, which focuses on minimizing the effort of human users.

5 ESTIMATING PLAN COSTS

We now discuss estimating the cost of a plan, which is the total time user $U$ spends in the human part to clean the clusters output by the machine part. As we will see below, the key idea is to estimate the quality of these clusters, then use that to estimate the time needed to clean them.

Specifically, let $V = \{v_1, \ldots, v_n\}$ be the set of input values, and $p_1, \ldots, p_k$ be the plans that we will consider, where each plan $p_j$ applies $\text{HAC}(\lambda)$ to $V$ to obtain a set of clusters $C_j$, then employs a user $U$ to clean $C_j$ using Split and Merge. Let $C_j = \{c_1, \ldots, c_n\}$. Then the cost of $p_j$ (i.e., the time for $U$ to clean $C_j$) can be expressed as $\text{cost}(p_j) = \sum_{i=1}^{n}\text{time}(\text{SplitCluster}(c_i)) + \text{time}(\text{LocalMerge}(L)) + \text{time}(\text{GlobalMerge}(T))$, where $L$ is a list of values summarizing the output of SplitCluster, and $T$ is a list summarizing the output of LocalMerge. We now estimate these quantities.

Estimating the Cost of SplitCluster: We need to estimate $\text{SplitCluster}(c_i)$ for each cluster $c_i \in C_j$. To do this, we make two assumptions:

1. All clusters $c_1, \ldots, c_n$ produced by $p_j$ have the same cluster purity $\alpha_j$ (which is defined in Definition 3.4).
2. When we use SplitCluster to split a cluster $c_i$ (produced by $p_j$) into a pure cluster containing all values of the dominating entity and a "mixed" cluster containing all the remaining values, the "mixed" cluster also has purity $\alpha_j$. When we split this "mixed" cluster, the resulting "mixed" cluster also has purity $\alpha_j$, and so on.

These are obviously simplifying assumptions. However, they reflect the intuition that each plan $\text{HAC}(\lambda)$ produces clusters of a certain quality level, and that this quality level can be captured by a single number, $\alpha_j$, which is the purity of all the clusters.

Further, they allow us to efficiently estimate plan costs. Finally, Section 8 empirically shows that with these assumptions we can already find good plans.

Next, we use the above assumptions to estimate the cost of SplitCluster($c_i$). Suppose that we already know $\alpha_j$ (we show how to estimate $\alpha_j$ later in this section), and that $\alpha_j \geq 0.5$, then when splitting $c_i$ using SplitCluster, user $U$ creates two clusters: a pure dominating cluster $c_{i,w}$ of size $\alpha_j \psi_i$, where $\psi_i$ is the size of $c_i$, and a remainder cluster $c_{i,r}$ of size $(1 - \alpha_j)\psi_i$. If $\alpha_j > 1$, we assume that its purity is also $\alpha_j$. $U$ then splits $c_{i,r}$, etc. After $\beta_i$ splits, $U$ has created $\beta_i + 1$ clusters of sizes $\alpha_j \psi_i, \alpha_j (1 - \alpha_j)\psi_i, \ldots, \alpha_j (1 - \alpha_j)^{\beta_i} \psi_i, (1 - \alpha_j)^{\beta_i} \psi_i$ such that the last cluster has a single element. Thus we can estimate $\beta_i$ as $\lfloor \log_{1 - \alpha_j} \psi_i \rfloor$. Since we can split a cluster of size $\psi_i$ at most $\psi_i - 1$ times, we set $\beta_i = \min(\psi_i - 1, \lfloor \log_{1 - \alpha_j} \psi_i \rfloor)$.

Recall that Section 3 defines seven user operations: focus($a$), select($a$), match($x,y$), isPure($c$), findDom($c$), memorize($v$), and recall($v$) (see Table 1). Let $\rho_f, \rho_m, \rho_p, \rho_d, \rho_r, \rho_s$ be their costs (i.e., times), respectively. As we will see later, the cost $\rho_p$ of isPure($c$) is a function of $\psi_i$, the size of $c$, and $\alpha_j$, the purity of $c$. Hence, abusing notations, we will denote this cost as $\rho_p(\psi_i, \alpha_j)$ for cluster $c_i$.

Similarly, the cost $\rho_d$ of findDom($c$) is a function of the size of $c$, and will be denoted as $\rho_d(\psi_i)$ for cluster $c_i$. The remaining five costs ($\rho_f, \rho_m, \rho_s$, etc.) will be constants. We can now estimate the cost of SplitCluster($c_i$) as $\alpha_j \geq 0.5$ as $\text{cost}(\text{SplitCluster}(c_i)) = \sum_{\alpha_j \geq 0.5} \rho_p(\psi_i(1 - \alpha_j)^{\beta_j} \psi_i, \alpha_j) + \rho_f + \rho_m(1 - \alpha_j)^{\beta_j} \psi_i + \rho_r + \rho_s(1 - \alpha_j)^{\beta_j} \psi_i(\rho_m + (1 - \alpha_j)\rho_s) + \rho_p$.

Appendix B discusses deriving the above formula, and computing the cost for the cases $\alpha_j \in [0.1, 0.5]$ and $\alpha_j < 0.1$.

Estimating the Cost of LocalMerge: Recall that $\text{HAC}(\lambda)$ produces the set of clusters $C_j = \{c_1, \ldots, c_n\}$, and that $\beta_j$ is the total number of splits user $U$ performs in SplitCluster for each cluster $c_i$. Then at the end of the split phase, $U$ has produced a set of $r_j$ pure clusters, where $r_j = \sum_{i=1}^{n}(\beta_i + 1)$. Assuming that executing LocalMerge on any list will shrink its size by a factor of $\tau$ (currently set to 0.98), we can estimate the time of executing LocalMerge on the output of the split phase as $r_j \rho_r + r_j(1 - \tau)(3\rho_f + 2\rho_m) + \rho_f + \rho_s$ (see Appendix B for an explanation).

Estimating the Cost of GlobalMerge: GlobalMerge produces $r'_j = \tau r_j$ pure clusters to which user $U$ will apply GlobalMerge.
Recall that GlobalMerge takes the first three values in the input list \( L \), displays them in three columns, then asks \( U \) to go through the rest of the values of \( L \) and check a box if any value matches the values of the columns (see Figure 6), and so on. We assume that in each such iteration, for each column, \( \epsilon \) values will match, resulting in \( \epsilon \) checkboxes being marked. Then we can estimate the cost of GlobalMerge as \( \sum_{i=1}^{13|\Omega|} \left[ 3p_2 + (r' - 3(j - 1))r' - 3)\rho_1 + 3(r' - 1)(\rho_1 + \rho_2 + \rho_1) \right] \) (see Appendix B).

### Estimating the Cluster Purity \( \alpha_2 \):
Recall that we assume all clusters \( c_1, \ldots, c_n \) produced by HAC(\( \lambda \)) have the same cluster purity \( \alpha_2 \). Using set-aside datasets, we found that \( \alpha_2 \) could be estimated reasonably well using a power-law function \( a_b^k \) (where \( b \) is negative, see Table 1). To estimate \( a \) and \( b \), we compute \( \lambda_{10} \) and \( \lambda_{20} \). To compute \( \lambda_{10} \), we apply HAC(10) to the set of input values to obtain a set \( C_{10} \) of clusters. Next, we randomly sample 3 clusters of size 10 from \( C_{10} \) (if there are less than 3 such clusters, we select the three largest). Next, we show each cluster to user \( U \), ask him/her to identify all values referring to the dominating entity, then use those to compute the cluster purity. Finally, we take the average purity of these clusters to be \( C_{10} \). We proceed similarly to compute \( C_{20} \).

We now have three data points: \((1,1), (10, \lambda_{10}), \) and \((20, \lambda_{20})\), which we can use to estimate \( a \) and \( b \) in the function \( a_b^k \), using the ordinary least-squares method.

### Estimating the Costs of User Operations:
Finally, we estimate the costs of the seven user operations (see Table 1). The costs of focus(a) and select(a) measure the times user \( U \) focuses on an object and then selects it (e.g., by clicking a mouse button). After a number of timing with various users, we found that these times are roughly the same for most users, and we set them to be \( \rho_f = \rho_s = 0.5 \) seconds. Similarly, we found the times of memorize(v) and recall(v) to be roughly constant, at \( \rho_m = \rho_r = 0.4 \) seconds respectively (see Table 1).

The time \( \rho_m \) of match(x,y), however, while largely not dependent on \( x \) and \( y \), does vary depending on user \( U \). Further, estimating the time \( \rho_f \) of isPure(c) and time \( \rho_g \) of findDom(c) is significantly more involved. To determine whether a cluster \( c \) is pure, user \( U \) needs to examine at most \( a c \psi \) values in \( c \) (where \( \psi \) is the size of \( c \)) before he/she sees the first value not referring to \( \psi \).

To find the dominating entity \( e^c \) of cluster \( c \), we distinguish two cases. If \( \psi \leq |STM| \), then user \( U \) can execute findDom(c) entirely in \( U \)'s short-term memory. In this case the time is proportional to \( \psi \). Otherwise \( U \) needs to use paper and pen, and we found that the time roughly correlates to \( \psi^2 \). Thus, we model the time \( \rho_f(\psi) \) of findDom(c) as \( \eta_1 \psi \) if \( \psi \leq |STM| \) and as \( \eta_2 \psi^2 + \eta_3 \) otherwise.

All that is left is to estimate the cost \( \rho_m \) of match(x,y), and the parameters \( \gamma, \eta_f, \eta_g, \eta_1, \eta_2, \eta_3 \) of the cost models of isPure(c) and findDom(c). To do so, when running HAC(20) (to estimate cluster purity \( \alpha_2 \)), we also ask user \( U \) to perform a few match, isPure, and findDom operations, then use the recorded times to estimate the above quantities (see Appendix B). Altogether, the time it takes for users to calibrate cluster purity \( \alpha_2 \) and the cost models of user operations was mere minutes in our experiments (and was included in the total time of our solution).

### 6 SEARCHING FOR THE BEST PLAN
Recall that to cluster the values \( V = \{v_1, \ldots, v_N\} \), we consider \( n \) plan \( p_1, \ldots, p_n \), where each plan \( p_i \) applies HAC(\( \lambda \)) to \( V \) to obtain a set of clusters \( C_{ij} \), then employs a user \( U \) to clean \( C_{ij} \). We now discuss how to efficiently find the plan \( p_{\lambda_k} \) with the least estimated cost.

Naively, we can (1) execute HAC(\( \lambda \)) for each plan \( p_i \) to obtain \( C_{ij} \), (2) apply the cost estimation procedures in the previous section to \( C_{ij} \) to compute the cost of \( p_i \), then (3) return the plan with the lowest cost. Steps 2-3 take negligible times. Step 1 however applies HAC(1), \( \cdots \), HAC(n) separately to \( V \), which altogether can take a lot of time, e.g., 7.3 minutes for \(|V| = 480\) and 1.1 hours for \(|V| = 960\) in our experiments.

To address this problem, we have developed a solution to jointly execute HAC(1), \( \cdots \), HAC(n), such that executing a plan can reuse the intermediate results of executing a previous plan. Specifically, we first execute HAC(n), i.e., the regular HAC. Recall that each iteration \( Iter \) of HAC(n) merges two clusters. Let \( s(i) \) be the size of the largest cluster at the end of \( Iter \). Suppose there is a \( k \) such that \( s(i) \leq \lambda \) but \( s(i + 1) > \lambda \). Then we know that HAC(\( \lambda \)) can reuse everything HAC(n) has produced up to \( Iter \), but cannot proceed to \( Iter_{i+1} \). So at the end of \( Iter \), we save certain information for HAC(\( \lambda \)) (e.g., the merge commands so far, the value \( \lambda \)), then continue with HAC(n). Once HAC(n) is done, we go back to each saved point \( \lambda \) and resume HAC(\( \lambda \)) from there. This strategy enables great reuse, especially for high values of \( \lambda \), e.g., slashing the time for 960 values from 1.1 hours to 18 secs.

### Putting It All Together:
We can now describe the entire Winston system, as used by a single user \( U \). Given a set of values \( V \) to normalize, (1) Winston first calibrate the cluster purity \( \alpha_2 \) and the cost models. To do so, it runs HAC(10) and HAC(20), asks user \( U \) to perform a few basic tasks on sample clusters from these algorithms, then use \( U \)'s results to calibrate (see Section 5). (2) Winston runs the above search procedure to find a plan \( p_{\lambda_k} \) with the least estimated cost. (3) Finally, Winston sends the output clusters of \( p_{\lambda_k} \) to user \( U \) to clean, using procedures Split and Merge.

### 7 WORKING WITH MULTIPLE USERS
So far we have discussed how Winston works with a single user. In practice, however, multiple users (e.g., people in the same team) are often willing to jointly perform VN. We now discuss how to extend Winston to divide the work among such users, to speed up VN.

Consider the case of \( k \) users. Naively, we can divide the set of input strings into 3 equal parts, ask each user to apply Winston to perform VN for a part, then combine the three outputs to form a set of clusters. We can obtain a canonical string from each cluster, producing a new list of strings. Then we can divide this new list among \( k \) users, repeat the process, and so on. This naive solution however does not work well, because it often spreads matching strings, i.e., those belonging to a golden cluster, among all \( k \) users, causing much additional work in matching across the individual lists, in later steps.

Intuitively, strings within a golden cluster should be assigned to a single user, as much as possible. We have extended Winston to realize this intuition. In the extension, Winston first briefly interacts with each user to learn his/her profiles. Next, it uses these profiles to search a plan space to find a good VN plan. Next, it executes the machine part of this plan to produce a set \( C \) of clusters. Then it divides \( C \) among the users, such that each will have roughly the same workload. The intuition here is that a cluster in \( C \) captures many strings that belong to the same golden cluster, and is assigned to a single user. Next, Winston asks each user to use Split and Merge to clean the assigned clusters. Finally, it obtains the set of (cleaned) clusters from all users, then
repeatedly performs a distributed version of GlobalMerge until all clusters have been verified and cleaned. Appendix C describes the algorithm in detail, provides the pseudo code, and discusses cost estimation procedures for this version of Winston.

### 8 EMPIRICAL EVALUATION

We now evaluate Winston. Among others, we show that Winston can significantly outperform existing solutions, that it can leverage multiple users to drastically cut VN time, and that it can scale to large datasets.

#### 8.1 Existing Manual/Clustering Solutions

We first compare Winston with state-of-the-art manual and clustering solutions (Section 8.3 considers string/entity matching solutions). We use the four datasets in Table 2, obtained online and from VN tasks at a company. For each dataset we manually created all correct clusters, to serve as the ground truth. (We consider larger datasets later in Section 8.4.)

**The Existing Solutions:** We consider four solutions: Manual, Merge, Quack, and OpenRefine. Manual is the typical manual method that we have observed in industry. It can be viewed as performing the GlobalMerge method (Section 3.2). Merge is our own manual VN method, which performs LocalMerge then GlobalMerge.

Quack is a string clustering tool used extensively for VN at a company. It also uses HAC like Winston, but does not place a limit on the cluster size. We extended Quack by asking the user to clean the clusters using Split and Merge. Merge and Quack can be viewed as the two plans HAC(1) and HAC(n) in the plan space explored by Winston (where n is the number of values to be normalized).

OpenRefine is a popular open-source tool to wrangle data [1]. It uses several string clustering algorithms to perform VN [2]. Among these, the most effective one appears to be KNN-based clustering [2]. We extend this algorithm to work with Split and Merge (because the GUI provided by OpenRefine is very limited).

**Results:** Table 3 shows the times of Winston vs. the above four methods (in minutes), using a single user. For each method we measure the total time the user spends cleaning the clusters (for Winston this includes the calibration time). It is difficult to recruit a large number of real users for these experiments, because cleaning some datasets (e.g., Nickname) would take a few working days. So we use synthetic users and each data point here is averaged over 100 such users, see Appendix D. (We use real users to "sanity check" these results in Section 8.4.)

The table shows that Manual performs worst, incurring 3-6800 minutes. Merge performs much better, especially on the two large datasets, incurring 4-1961 minutes, suggesting that performing a local merge before a global merge is important. Merge is clearly the manual method to beat.

Quack is a bit faster than Merge on Nickname (1808 vs. 1961), but slower on the remaining three datasets. OpenRefine’s performance is very uneven. It is a bit faster than Merge on Citation, but far slower on the other three datasets.

| Name | Description | Sample Values |
|------|-------------|---------------|
| Nickname | Nicknames and issue of their types | "Cissy", "Fanny", "Frannie" |
| Citation | Article citations from Google Scholar and DBLP | caching technologies for web applications c moshammer 2001 |
| Life Stage | 199 Large scale staged products | "Custody", "Wetters", "Young Professionals" |
| Big Ten | Names of Big Ten conference colleges | University of Iowa, "UWas", "UM Twin Cities" |

**Table 2: Datasets for our experiments.**

In contrast, Winston performs much better than Merge. On Nickname it saves 7.5 hours of user time (see the last column). On Citation it saves 3.34 hours of user time. On Life Stage it is comparable to Merge, and on Big Ten it is only 3 mins worse (due to the overhead of user calibration time).

Winston also outperforms both Quack and OpenRefine. Importantly, in all cases where Quack or OpenRefine performs worse than Merge, Winston is able to select a good plan which allows it to outperform Merge.

#### 8.2 Working with Multiple Users

We have shown that Winston outperforms existing manual and clustering methods. We now examine how Winston can leverage multiple users to reduce VN time. Table 4 shows that Winston can leverage multiple users to drastically cut the VN time, e.g., from 1512 minutes with 1 user to 412 with 9 users for Nickname, and from 1112 to 177 for Citation. The most significant reduction is achieved early, e.g., from 1 to 3-5 users. After that, adding more users still helps reduce the VN time, but only in a "diminishing-return" fashion.

#### 8.3 Limitations of SM/EM Solutions

We now compare Winston to existing string matching (SM) and entity matching (EM) solutions, specifically with TransER [47], Falcon [13], and Magellan [32].

**Comparing with TransER:** As discussed in Section 4.2, there are two main ways to use SM/EM solutions in our context. First, a solution can produce a set of matches M, employ a user U to verify certain matches in M, then use match transitivity to infer even more matches. The work [47] describes such a solution, which we call TransER.

The main problem, as discussed in Section 4.2, is that such solutions cannot guarantee 100% recall. Consider TransER, which matches strings using rule $\text{Jaccard}(g(u), g(v)) ≥ \alpha$. Assuming a perfect user U who does not make mistakes when verifying matches, Figure 9 shows the recall of TransER on our four datasets as we vary $\alpha$. It shows that to reach 100% recall, $\alpha$ must be set to less than 0.08. But that would produce a huge number of matches (almost the entire Cartesian product), which require a huge amount of effort from the user to verify. In such cases, it is not difficult to show that TransER would perform worse than Merge.

**Comparing with Falcon and Magellan:** The second way to use current SM/EM solutions is to produce the matches, then group them into clusters. To examine this approach, we use Falcon [13] and Magellan [32]. A recent work (name withheld for anonymous reviewing) has adapted Falcon to SM, and shown that it outperforms existing SM solutions. Thus, Falcon can be viewed as a state-of-the-art SM solution. Magellan, on the other hand, can be viewed as a state-of-the-art EM solution. To learn a matcher, both Falcon and Magellan require the user to label a

**Table 3: Winston vs four existing solutions.**

| Dataset | Manual | Merge | Quack | OpenRefine | Winston | Savings |
|--------|--------|-------|-------|------------|---------|--------|
| Nickname | 6800 | 1961 | 1808 | >10000 | 1512 | 7.5hrs |
| Citation | 6385 | 1513 | 1371 | 1280 | 1112 | 3.4hrs |
| Life Stage | 13 | 9 | 13 | 12 | 9 | 0hrs |
| Big Ten | 3 | 4 | 9 | 22 | 7 | 9min |

**Table 4: The times of Winston with multiple users.**
### 8.4 Additional Experiments

**“Sanity Check” with Real Users:** We want to “sanity check” our results so far using real users. Extensive checking is very difficult because it is hard to recruit real users for these time-consuming experiments. As a result, we carried out a limited checking. Specifically, we performed stratified sampling to obtain a Nickname sample of 316 values and a Citation sample of 343 values. On each sample we recruited multiple real users and asked them to perform Merge, Winston and 3Winston (i.e., Winston with 3 users), taking care to minimize user bias. The right side of Figure 10.a shows the results for Nickname. For comparison purposes, the left side of the figure shows the times with synthetic users. Figure 10.b shows similar results for Citation.

The figures show that “Simulation” approximates “Real User” quite well. In both cases, the ordering of the methods is the same. Further, the results show that Winston can do much better than Merge, and 3Winston in turn can do much better than Winston. While limited, this result with real users does provide some anecdotal support for our simulation findings.

**Finding Good Plans:** Table 7 shows that Winston finds good plans. Consider Nickname. Recall that we ran 100 synthetic users for this dataset. For each user $U_i$, Winston estimated the costs of all plans then selected plan $p^*$, the one with the least estimated cost. Knowing gold clusters, however, we can simulate how $U_i$ executes each plan and thus compute the plan’s exact cost. This allows us to find the rank of $p^*$ on the list of all plans sorted by increasing cost, as well as the time difference between $p^*$ and the best plan.

The first row of Table 7 shows this information. Here, Winston considered a space of 100 plans. For all 100 users, it selected the plan ranked 2nd. The difference between this plan and the best plan, however, is just 3–4 mins (over 100 users). The next two cells show the average/min/max times of the best plan, and the average/min/max difference in percentage. The remaining rows are similar. Thus, Winston did a good job. In many cases, it selected top-ranked plans, and most importantly, all the selected plans differ in time from the best plans by only 0–14% (see the last column).

**Scaling to Large Datasets:** Finally, we examine how Winston scales to large datasets. Table 8 shows the estimated cleaning time of Merge, Quack, Winston, and 3Winston, i.e., Winston with 3 users, for synthetic datasets of various sizes. The table shows that Merge is not practical, taking 29 days, 4.4 years, and 11.5 years for 100K, 500K, and 1M strings, respectively. Quack is better, but still incurs huge times.

Winston, in contrast, can reduce these times drastically, to just 13 days, 9.6 months, and 1.3 years, respectively. As discussed in Section 1, this is because Winston provides a better UI, so the
user can do more with less effort. Further, the machine part of Winston outputs clusters that are "user friendly", i.e., requiring little effort for the user to clean. Finally, Winston searches a large space of plans to find one with minimal estimated human effort. 3Winston does even better, cutting the times to clean 500K and 1M strings to just 2.2 and 3.5 months, respectively. These suggest that cleaning large datasets with Winston indeed can be practical, especially by dividing the work among multiple users.

9 RELATED WORK

Data Cleaning: Data cleaning has received enormous attention (e.g., [3, 5, 9, 11, 14, 16, 17, 20–24, 26, 30, 31, 33, 34, 37–39, 47]). See [10, 12, 15, 40] for recent tutorials, surveys, and books. However, as far as we can tell, no published work has examined the problem of cleaning with 100% accuracy, as we do for VN in this paper. Our work here shows that the problem of cleaning to reach a desired level of accuracy raises many novel challenges for data cleaning.

Value Normalization: Much work has addressed VN, typically under the name "synonym discovery". Most solutions use string/contextual similarities to measure the relatedness of values [8, 49], and employ various techniques, e.g., clustering, regular expressions, learning, etc. [35, 49] to match values. However, no work has examined verifying and cleaning VN results to reach 100% accuracy, as we do here.

Clustering: Our work is related to clustering (which we use in VN). Numerous clustering algorithms exist [18, 27, 48], but we are not aware of any work that has developed a human-driven procedure to clean up clustering output and tried to minimize the human effort of this procedure. Much work has also tuned clustering (e.g., [6, 7]), but for accuracy. In contrast, our work can be viewed as tuning clustering to minimize the post-clustering cleaning effort.

String/Entity Matching for the "Machine" Part: At the core VN is a matching problem, and hence string matching (SM) and entity matching (EM) solutions can be used in the "machine" part. Numerous such solutions have been developed (e.g., TransER, Falcon, Magellan, Waldo and more [13, 32, 44, 47]). We have discussed in Section 4.2 and experimentally validated in Section 8.3 that these methods do not work well for our context. The main reason is that they generate large mixed clusters that are very time consuming for users to clean. This result suggests that when we combine a machine part with a human part, it is important to develop the machine part such that it generates results that are "user friendly" for the user in the human part to work with.

User Interaction Techniques for the "Human" Part: Many recent works on string/entity matching and crowdsourcing solicit user feedback/action via GUIs to verify and further clean (e.g., CrowdDB, CrowdER, and more [19, 20, 44–47]). These works however allow only a limited range of user actions (e.g., asking users if two tuples match). A recent work, Waldo [44], considers more expressive user actions, such as showing six values on a single screen and allowing the user to cluster all six in "one shot". The above works differ from Winston in two important ways. First, the range of user actions that they allow is still quite limited. In contrast, Winston considers far more expressive user actions, such splitting a cluster, merging two clusters, etc. Second, the above works do not explicitly model the human effort of the user actions and do not seek to minimize this total human effort, as Winston does. For example, they model the cost of labeling a value pair or clustering six values to be a fixed value (e.g., 3 cents paid to a crowd worker), regardless of how much effort a user puts into doing it. As such, our work can be viewed as advancing the recent human-in-the-loop (HILDA) line of research, by considering more expressive user actions and studying how to optimize their human-effort cost using RDBMS-style techniques.

RDBMS-Style Cleaning Systems: Many cleaning works have also adopted an RDBMS-style operator framework, e.g., AJAX [22], Wisteria [23], Arnold [28], QuERy [4]. They however do not consider expressive human operations, modeling human actions at a coarse level, e.g, labeling a tuple, converting a dirty tuple into a clean one. In contrast, we model and estimate the cost of simple human operations, e.g., removing a value from a cluster, verifying if a cluster is clean, etc. Finally, current work typically optimizes for the accuracy and time of cleaning algorithms (while assuming a ceiling on the human effort). In contrast, we minimize the human effort, which can be a major bottleneck in practice.

Interactive Cleaning Systems: Another prominent body of work develops interactive cleaning systems (e.g., AJAX [22], Potter Wheel [41], Wrangler [29], Trifacta [26], ALIAS [42], and [25]. Such systems often try to maximize cleaning accuracy, or efficiently build data transformations/cleaning scripts, while minimizing the user effort. To the best of our knowledge, however, they have not examined the problem of VN with 100% accuracy. For example, active learning-based approaches such as [42] do not tell the user what to do (to reach 100% accuracy) if after using them the accuracy of the cleaned dataset is still below 100%.

10 CONCLUSIONS & FUTURE WORK

We have examined the problem of value normalization with 100% accuracy. We have described Winston, an RDBMS-style solution that defines human operations, combines them with clustering algorithms to form hybrid plans, estimates plan costs (in terms of human verification and cleaning effort), then selects the best plan.

Overall, our work here shows that it is indeed possible to apply an RDBMS-style solution approach to the problems of 100% accurate cleaning. Going forward, we plan to open source our current VN solution, explore other clustering algorithms for VN, and explore applying the solutions here to other cleaning tasks, such as deduplication, outlier removal, extraction, and data repair.

REFERENCES

[1] 2018. OpenRefine open-source tool. openrefine.org.
[2] 2018. The value normalization capabilities of OpenRefine. https://github.com/OpenRefine/OpenRefine/wiki/Clustering.
[3] Z. Abedjan et al. 2016. Detecting Data Errors: Where are we and what needs to be done? PVLDB 9, 12 (2016), 995–1004.
[4] H. Altwaijry et al. 2015. QuERy: A Framework for Integrating Entity Resolution with Query Processing. PVLDB 9, 3 (2015), 120–131.
[5] A. Arasu et al. 2011. Towards a Domain Independent Platform for Data Cleaning. IEEE Data Eng. Bull. 34, 3 (2011), 43–50.
[6] S. Basu et al. 2002. Semi-supervised Clustering by Seeding. In ICML.
Algorithm 3 Merge Phase

Procedure Merge(L)
Input: a list of values L representing output clusters of Split phase
Output: a set of clean clusters C of values in L

1. L = consolidated list of values from LocalMerge step
2. return GlobalMerge(L)

Procedure GlobalMerge(L)
Input: a list of values L sorted alphabetically
Output: a set of clean clusters C of values in L

1. while |L| > 1 do
2. if |L| < 3 then B ← L[1], L[2] else B ← L[1], L[2], L[3]
3. MarkValuesForGlobalMerge(B, L)
4. // at the end, user selects "global merge" button
5. for i = 1, ..., |B| do
6. Merge(B[i]) and values marked to match it into cluster C
7. return C

Procedure MarkValuesForGlobalMerge(B, L)
Input: a list B of values on columns, a list D of values on rows
Output: links among values in B and D that match

1. for each i = 1, ..., |B| do
2. if B[i] = r then (B[i], r) ← markLink(r)
3. if B[i] is not null then globalMerge(B[i], L)
4. else match(B[i])
5. for each j = 1, ..., |D| do
6. if D[j] is not null then globalMerge(D[j], L)

A DEFINING THE HUMAN PART

Algorithm 3 describes the Merge and GlobalMerge procedures (LocalMerge has been described in Section 3.2).

B ESTIMATING PLAN COSTS

In this section we describe the cost estimation formula for various procedures used in the human part of value normalization plans and how we have derived them.

SplitCluster Procedure: To estimate the cost of applying SplitCluster to a cluster c_i during the execution of the plan p_1, we consider the following three cases:

Case 1 (σ_0 ≥ 0.5): Recall that when σ_0 ≥ 0.5 we estimate the cost of applying SplitCluster to c_i as follows:

\[ \text{cost}_{\text{splitCluster}}(c_i; \sigma_0 ≥ 0.5) = \frac{[\alpha_i - \beta_i]}{[\alpha_i - \beta_i]^2 + \sigma_0} \]

Also recall that we go through β_i iterations of splitting c_i and at iteration j ∈ {1, ..., β_i} we split an impure cluster of approximate size (1 − α_i)^j/|β_i|, e.g. at iteration 1 we split the whole cluster of size (1 − α_i)^1/|β_i| = |β_i|. Each iteration corresponds to a (recursive) call of the SplitCluster Procedure. At each execution of SplitCluster, there are three lines (numbered 2, 4 and 7 in Algorithm 1) involving user operations and thus only these lines contribute to the cost of the procedure.

The cost of line 2 is captured by part q_0 of the above formula, it consists of the cost of isPure (executed on a cluster of size (1 − α_i)^j/|β_i|) and then focusing on and selecting "no" button. Part q_2 captures the cost of line 4: it consists of the cost of findingDom and then focusing on and selecting "mark values" button. Finally parts q_{3,1} and q_{3,2} of the above formula capture the cost of line 7: q_{3,1} is the cost of focusing on and selecting "create/clean new cluster" button. Part q_{3,2} in turn consists of going through the cluster values (line 3 in MarkValues pseudo code), focusing on each value, matching it with the dominating entity of the cluster and selecting the value.
if they match (i.e. for $1 - \alpha_j$ fraction of the values).

Case 2 ($\alpha_j \in [0.1, 0.5)$): We estimate the cost of applying SplitCluster to $c_j$ when $\alpha_j \in [0.1, 0.5)$ as follows:

$$
cost_{split\_cluster}(c_j; \alpha_j \in [0.1, 0.5)) = \sum_{m=1}^{\alpha_j} [\rho_t(1 - \alpha_j)^{-\gamma} \phi_j \alpha_j] + \rho_t + \rho_t(1 - \alpha_j)^{-\gamma} \phi_j \rho_t + \rho_t + ((1 - \alpha_j)^{-\gamma} \phi_j \rho_t + \rho_t + \rho_t + \alpha_j(\rho_t + \rho_t)].
$$

The derivation is very similar to the previous case. The only difference is the fraction of matching values at each execution of MarkValues which is $\alpha_j$ instead of $1 - \alpha_j$.

Case 3 ($\alpha_j < 0.1$): When $\alpha_j < 0.1$ we estimate the cost of applying SplitCluster to $c_j$ as follows:

$$
cost_{split\_cluster}(c_j; \alpha_j < 0.1) = \rho_t(\phi_j \alpha_j) + \rho_t + \rho_t + \rho_t
$$

$$
+ \rho_t(\phi_j \alpha_j) + \rho_t + \rho_t + \phi_j(1 - \gamma) [3 \rho_t(2 \rho_t + \rho_t)] + \rho_t + \rho_t + \rho_t + \rho_t + \rho_t + \rho_t
$$

$$
+ \rho_t(\phi_j \alpha_j) + \rho_t + \rho_t + \phi_j(1 - \gamma) [3 \rho_t(2 \rho_t + \rho_t)] + \rho_t + \rho_t + \rho_t + \rho_t + \rho_t + \rho_t + \rho_t
$$

Here $q_j$ is the cost of executing isPure on $c_j$ and then focusing on and selecting “no” button. $q_j$ is the cost of executing findDom and then focusing on and selecting “clean mixed cluster” button. $q_j$ is the cost of executing LocalMerge on $c_j$ and the rest of the formula is the cost of executing GlobalMerge on the results of the previous step. We will describe the costs of LocalMerge and GlobalMerge in the following sections.

**LocalMerge Procedure:** Recall that for a particular plan $p_k$ the split phase result consists of approximately $r_k$ pure clusters of input values. Thus the size of the input list $L$ to the LocalMerge is $r_k$. User $U$ goes through the values in $L$ and for each value, he or she first remembers it. For $1 - \tau$ fraction of the values in $L$, $U$ finds a value in his or her short-term memory (STM) in which case he or she (1) selects the current value, then focuses on and selects the value retrieved from STM and finally focuses on and selects the link button. Finally the user focuses on and selects “done local merging” button to proceed to the global merging. Adding up these costs gives us the cost formula $r_k \rho_t + r_k (1 - \tau)(3 \rho_t + 2 \rho_t) + \rho_t + \rho_t$.

**GlobalMerge Procedure:** GlobalMerge takes as input a list of values $L$ with approximate size $r_j$. The GlobalMerge consists of possibly several iterations and in each iteration, we assume that each of the three values displayed on the columns of the GUI would match approximately $\xi_j - 1$ values displayed on the rows, forming clusters of size $\xi_j$. Thus the number of iterations of GlobalMerge would be approximately $g_j = r_j / (3 \xi_j) = [1 / (3 \xi_j)]$.

At iteration $j \in \{1, \ldots, g_j\}$ the user sees $r_j - 3(j - 1)\xi_j$ values remained to be matched, three of which are displayed on the columns and the rest on the rows of the GUI. The user first memorizes the three values on the columns (with total cost of $3 \rho_t$). Then for the $r_j - 3(j - 1)\xi_j - 3$ values on the rows the user recalls each value. Lastly for each of the three columns the user focuses on and selects checkboxes for $\xi_j - 1$ rows. Finally the user focuses on and selects “global merge” button to finish the current round. Adding up these costs would give us the cost formula $\sum_{j=1}^{[1/3\xi_j]} [3 \rho_t + (r_j - 3(j - 1)\xi_j - 3) \rho_t + 3(\xi_j - 1)(\rho_t + \rho_t) + \rho_t + \rho_t]$.

**Estimating the Costs of User Operations:** We now describe how we estimate the cost $p_m$ of the match operation, the parameters $\eta$ and $\gamma$ of the isPure operation cost function and the parameters $\eta_1$, $\eta_2$ and $\eta_3$ of the findDom operation cost function during the calibration stage.

To estimate $p_m$ we first pick three pairs of random values of $V$. We then ask the user $U$ to match each pair and, depending on whether they match or not, to select a “yes” or “no” button. For the $k$th pair we measure the time $t_{m,k}$ it takes from when we show the screen containing the pair of values and the buttons to $U$ till he or she selects one of the buttons. During this time the user matches the values shown on the screen, then focuses on one of the buttons and selects it. Hence the time we measure is equal to $p_m + \rho_t + p_t$ where $p_m$ is our estimated cost of the $k$th match operation. We then calculate the $p_m(k) = t_{m,k} - (\rho_t + p_t)$ and estimate $p_m$ to be the average of $p_m(k)$, i.e. $p_m = \sum_{k=1}^{3} p_m(k) / 3$.

For the rest of the parameters above we use the results of HAC(20) we have previously run on the input dataset during the calibration phase. Denote the results of HAC(20) as $C_{20}$. First we pick three random non-singleton clusters $c_1$, $c_2$ and $c_3$ (of different sizes if possible) from $C_{20}$. Then we show each $c_k$ and ask $U$ to select a "yes" button if $c_k$ is pure and a "no" button otherwise. We record the time $t_{p,k}$ it takes from when we show $c_k$ to $U$ till one of the buttons is selected. We also record which button is selected. Using the same timing analysis we described for $p_m$, we form three equations of the form $a(c_k) \phi_j \gamma_k + \eta_0 = t_{p,k} - (\rho_t + p_t)$ where $k \in \{1, 2, 3\}$, $a(c_k)$ is the purity of $c_k$ and $\phi_j$ is the size of $c_k$. However since we don’t know the purity of $c_k$, $U$ we use the button $U$ has selected to guess the purity of $c_k$: if $U$ has selected the “yes” button, we set $a(c_k) = 1$, otherwise we set $a(c_k) = a(20)^b$ (we have already estimated $b$ and $d$ during the calibration of the purity function). Finally we use ordinary least-squares method to solve the system of three equations above to estimate the parameters $\eta$ and $\gamma$.

To estimate $\eta_1$ we first pick three clusters $c_1'$, $c_2'$ and $c_3'$ from $C_{20}$ such that $|c_1'| = |c_1| \leq |STM|$. We then show each $c_k'$ to $U$ and ask him/her to find the dominating entity of $c_k'$ and then select a value in $c_k'$ which refers to dom($c_k'$). For each $c_k'$ we measure the time $t_{p,k}$ it takes from when it is shown to $U$ till he/she selects the value referring to dom($c_k'$). Using the same timing analysis as above we obtain three equations of the form $t_{p,k} = \eta_1(\phi_j \gamma_k + \eta_0) / \phi_j$.

To estimate $\eta_2$ and $\eta_3$, we follow a similar process: we first pick three clusters $c_1''$, $c_2''$ and $c_3''$ from $C_{20}$ such that $\phi_j'' = |c_1''| > |STM|$. We then show each $c_k''$ to $U$ and ask him/her to find the dominating entity of $c_k''$ and then select a value in $c_k''$ which refers to dom($c_k''$). Using the same timing analysis as above we obtain three equations of the form $t_{p,k} = \eta_2(\phi_j \gamma_k + \eta_0) + \rho_t + \rho_t + \rho_t$. Finally we use ordinary least-squares method to solve the system of three equations above to estimate the parameters $\eta_2$ and $\eta_3$.

C WORKING WITH MULTIPLE USERS

We now describe cWinston, which extends Winston to work with multiple users. Assuming $k$ users want to collaborate to normalize a set $V$ of input values, cWinston goes through four main stages. In the first stage, it shows each user a few clusters of values in $V$ and asks them to perform some basic operations on them. cWinston then uses the results of these operations to tune the purity function parameters and user operation cost models for each user (the same way as Winston). Next, it takes the average
of purity function and cost model parameters to create a single purity function and a single cost model for each user operation. In the second stage, cWinston uses the above purity function and user operation cost models to find the best VN plan. To do so, it uses the same plan space searching procedure as Winston (see Section 6) to find the best plan. It then executes the machine part of the best plan to obtain a set C of clusters.

In the third stage, cWinston partitions C into k subsets of roughly the same number of values. It then assigns each subset to one of the users and asks them to clean their respective subsets of clusters using Split and Merge algorithms.

In the last stage, cWinston starts by collecting the results of Split-Merge from all the users and for each user, it creates a list of representative values of the clean clusters he or she has produced. It then picks the longest list and divides it into k chunks of roughly the same size. Next, cWinston asks each user to merge one of these chunks with the rest of the lists using the GlobalMerge procedure (see Section 6). It then collects the results from all the users and repeats this stage (i.e., takes representative values from the merged clusters, divides the largest list into k chunks, and so on) until all of the lists are verified/merged. Algorithm 4 shows the pseudocode of cWinston.

### Algorithm 4 cWinston

**Procedure:** cWinston($V, U$)

**Input:** a set $V$ of representative value sets, a set $U = \{u_1, \ldots, u_k\}$ of users

**Output:** a set of clean clusters $S$

1. for each $u_i \in U$ do
2.  $Ay \leftarrow$ tune purity function and cost model parameters for $u_i$
3.  $Ay_* \leftarrow$ search plan space to find the best plan using $Ay$
4.  $C \leftarrow$ run $H(c)^2$ on $V$
5.  Divide $C$ into $C' = \{C'_1, C'_2, \ldots, C'_k\}$ s.t. $|C'_i| = |C|/k, S_i \leftarrow 0$
6.  for each $u_i \in U$ do //each user $u_i$ executes Winston on $C'_i$
7.  $D_i \leftarrow$ Split($C'_i$); $E_i \leftarrow$ list of representative values of clusters in $D_i$
8.  LocalMerge($E_i$); $L_i \leftarrow$ Consolidated list of values of clusters in $D_i$ merge
9.  $S_i \leftarrow S_i \cup \text{GlobalMerge}(L_i)$
10. if $|U| = 1$ then return $S_i$ //effectively, (single-user) Winston
11. else return MultiUserMerge($S, U$)

**Procedure:** MultiUserMerge($S, U$)

**Input:** a set $S$ of clean clusters, a set $U = \{u_1, \ldots, u_k\}$ of users

**Output:** a set of clean clusters $S'$

1. $D \leftarrow 0$ //representative values for the sets of clusters in $S$
2. $S' \leftarrow 0$ //flattened $S$
3. for each $S_i \in S$ do
4.  $D_i \leftarrow$ a set of values representing the clusters in $S_i$, $S'_i \leftarrow S_i$
5. while each $|D| > 0$ do
6.  $D' \leftarrow \arg\max_{D_i} (|D_i|)$; $D' \leftarrow D' \cup D_i$; $D_i \leftarrow 0$ //match
7.  Divide $D'$ into $L = \{L_1, L_2, \ldots, L_k\}$ s.t. $|L_i| = |D'|/k$
8.  all users perform merge with their respective column values in parallel
9.  for each $E_i \subseteq L, D \leftarrow M \cup \text{GroupedMerge}(E_i, copy(D')) \\ union matches
10. for each $D_j \in D$ do
11.  for each $(v, w) \in M$ do
12.    Merge the clusters in $S' \subseteq \delta$ which $v$ and $w$ refer to, and
13.    set $v$ to refer to the new cluster
14. return $S'$

**Procedure:** GroupedMerge($L, u$)

**Input:** a list $L$ of column values, a set $D$ of representative value sets, a user $u$

**Output:** a set $M$ of matches

1. $M \leftarrow \emptyset$
2. while each $|L| > 0$ do //while there are still column values left
3.  $r \leftarrow \max(|L|, |L_i|)$; $L \leftarrow r$ values from $L$
4.  for each $b \in B$ do //merges $(i,b)$
5.    for each $D_j \in D$ do $M \leftarrow M \cup \text{SetMerge}(B, D_j, u)$
6.  $L \leftarrow L \cup B$
7.  return $M$

**Procedure:** SetMerge($B, D_j, u$)

**Input:** a set $B$ of column values, a set $D_j$ of representative value sets, a user $u$

**Output:** a set $M'$ of matches

1. $D_j \leftarrow \{v \mid u \land \text{similarity to the values in } B, M' \leftarrow \emptyset$
2. for each $b \in B$ do
3.  if $(b, v_0) \in B$ then
4.    $M' \leftarrow (M' \cup \{b, v_0\}) \land b \in B \land matches v$
5.  if $|M| = 3$ then break
6.  $D_j \leftarrow D_j \setminus \{v_0\}$
7.  return $M'$

The innermost sum corresponds to the number of rows examined per each set of three representative values during each scan. At iteration $i$, there are $k - i$ lists of representative values left to appear on the rows. Each time $u$ scans one of these lists, he or she matches on average $3^i$ rows, each of which requires a button click. Additionally, $u$ has to click the merge button, hence the term $(3^i + 1)(\rho_{F,u} + \rho_{P,u})$. To account for the number of rows examined in each list before finding the number of rows left in the list by finding the number of entities removed from the list at the end of this iteration, i.e., $|D_j|((|R_i - 1| + 3i))\xi$, and then subtracting this value from $|D_j|$. We also assume that only a $\mu$ proportion of these rows need to be investigated before finding matches, hence the term $\mu|D_j|(|1 - (R_i - 1) + 3i)|\xi$.

### D EMPIRICAL EVALUATION

#### Generating Synthetic Users:
We use a deterministic model of a user, i.e., we use constant values for $\rho_F, \rho_P, \rho_m, \rho_r, \rho_{F,u}, \rho_{P,u}, \gamma_0 \in [0.8,1.2], \rho_{F} \geq \rho_{P} = 0.5$. We then assume a range of values for each $p_m \in [0.3,0.5], \gamma \in [0.1,0.4], \gamma_0 \in [0.5,1]$ and $\gamma_1 \in [0.2,0.4]$. Next, we generate a random
simulated user by uniformly randomly sampling a number from each of the above ranges and assigning these values to the corresponding parameters of the cost model. Finally, we assign the remaining parameters as $\rho_z = \rho_r$, $\eta_2 = \eta_1/(\lvert STM \rvert \times 100)$ and $\eta_3 = 0.99\eta_1/\lvert STM \rvert$. 