Cleansing Databases of Misspelled Proper Nouns

Arturas Mazeika
arturas@inf.unibz.it

Michael H. Böhlen
boehlen@inf.unibz.it

Faculty of Computer Science, Free University of Bozen-Bolzano
Dominikanerplatz 3, I-39100 Bozen-Bolzano, Italy

Abstract

The paper presents a data cleansing technique for string databases. We propose and evaluate an algorithm that identifies a group of strings that consists of (multiple) occurrences of a correctly spelled string plus nearby misspelled strings. All strings in a group are replaced by the most frequent string of this group. Our method targets proper noun databases, including names and addresses, which are not handled by dictionaries.

At the technical level we give an efficient solution for computing the center of a group of strings and determine the border of the group. We use inverse strings together with sampling to efficiently identify and cleanse a database. The experimental evaluation shows that for proper nouns the center calculation and border detection algorithms are robust and even very small sample sizes yield good results.

1 Introduction

The high-dimensional nature of the string space puts forward a number of problems that do not exist in the numeric domain. However, besides the added complexity, strings also offer unique opportunities. In this paper we describe a solution that takes advantage of the high-dimensional space to clean databases of proper nouns, i.e., strings that do not occur in dictionaries.

Since strings are elements of a high-dimensional space the distance between any two strings is typically large. An exception are misspelled strings, which tend to be located near correctly spelled strings. The combination of these two properties means that small hyper-spheres can be used to cluster a string database. The hyper-spheres are far from each other, and each hyper-sphere encloses the correctly spelled string and the nearby misspelled strings.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CleanDB, Seoul, Korea, 2006

Figure 1 illustrates the setting for strings george, sydney, and jacob, together with misspelling of these strings. We describe a solution to group misspellings of a string by identifying the border and center of a hyper-sphere.

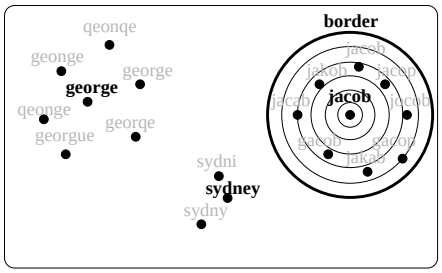


Figure 1: Database of Proper Nouns with Misspellings

The border detection algorithm is based on the *string proximity graph* (cf. Section 4.1), which captures the properties of proper noun databases with misspellings. The string proximity graph shows that in the immediate neighborhood of a string the number of strings is growing because of the misspellings. As we further increase the neighborhood the number of strings does not grow. There are no misspellings in this area and the other strings are further away because of the high-dimensional nature of the string space. The point at which the clusters stops to grow indicates the border of a group of misspelled strings.

The computation of the border and center is done in parallel. We start with a random string that has not yet been processed and identify all strings that are within distance one from this string. Next we adjust the center of the cluster and increase the radius. The adjustment of the center makes the method more robust, so that it also applies to groups of strings that are not far away from each other. As soon as an increase of the radius does not further increase the number of strings we have found a group and proceed with another string that has not yet been processed. The process stops when all the strings have been grouped.

The contributions of the paper are the following:

- We introduce a new cleansing technique for string data with typos. The solution is based on the (i) border detection and (ii) the center adjustment. The computation of the distance between strings is done with the help of q-grams of strings (substrings of length q). The center of the cluster is modeled as a bag of the

most frequent substrings of length q of the strings in the group. Thus, the center reflects the substrings that are common for the strings and neglects substrings that are the result of infrequent misspelling.

- We use inverse strings (IS) to determine close-by strings and to compute the border of the cluster. Inverse strings associate with each q-gram the string IDs that contain the q-gram. Even with inverse strings the computational complexity of the border detection is combinatorial wrt the length of the center string and radius of the cluster. We use sampling to approximate the border detection. This yields a linear complexity wrt the size of the sample.

- We provide experimental result for the border detection and data cleansing algorithms. We show that the border detection is robust and that even small sample sizes ensure good approximations of the border of clusters and a low cleansing error.

The organization of the paper is the following. Section 2 presents related work. Q-grams and inverse strings are reviewed in section 3. Border detection and computation of the center string are introduced in section 4. Approximation of the border with the help of IS data structure and sampling are described in section 5. Section 6 presents the algorithm of the cleansing of the data. We give an experimental evaluation in section 7. Finally, section 8 concludes the paper and offers future work.

2 Related Work

Fuzzy retrieval is the closest related work to our approach. Fuzzy retrieval algorithms get as input a string and threshold, and output strings that are within the given threshold. Chaudhuri et al. [2] introduce an algorithm that retrieves tuples that exactly match the query string with a high probability. Jagadish et al [10] and Ciaccia et al. [4] propose a family of index structures that support exact, prefix, and approximate queries on multi-string attributes. Jin et al. [12] propose an index structure that supports mixed types (string and non-string) of attributes for approximate retrieval.

Automatic spell checking techniques [13, 9] compare a potentially misspelled word with the words in a dictionary or a model based on the dictionary. They output a correction (or a set of corrections) for a given error threshold or r number of requested answers. If r is given the dictionary (or the model) is queried a number of times for different incremental thresholds until the size r is reached. In this paper we show how to automatically compute the threshold (border of the cluster).

Efficient approximation of selectivity for a given string and edit distance (overlap threshold) is investigated in [11]. This provides important statistical information about the string data. In this paper we focus on precise computation of the center and the border of a cluster, though both our border detection and approximate selectivity solutions can be combined. Our border detection algorithm can query for approximate string selectivity, and use the result to detect border of the cluster. Then inverse strings can be used to cluster and cleanse the data.

There is a large body of work in the area of the similarity metrics for string attributes. Such measures include edit distance [8] q-grams, cosine similarity [6, 3, 7] and its variants [5, 14]. Ananthakrishna [1] proposes a textual similarity function for strings.

3 Background

3.1 Q-grams

Definition 3.1 [q-grams.] The q-grams of a string α are obtained by sliding a window of size q over the characters of α. Since at the beginning and at the end of the string we have fewer characters than q, we extend the string by prefixing it with $q-1$ occurrences of # and suffixing it with $q-1$ occurrences of \$. We assume that symbols # and \$ do not occur in the input strings.

Example 3.1 [q-grams.] Let α = george and q=2. The q-grams of string α are

$$B(\text{george}) = \{\#g^1, ge^1, eo^1, or^1, rg^1, ge^2, e\$^1\}.$$

In order to distinguish different occurrences of the same 2-gram we associate each q-gram with a sequence number (displayed as a superscript). For example, 2-gram ge^1 denotes the substring at the beginning of the string, and 2-gram ge^2 denotes the substring at the end of the string (positions 5–6 of the input string).

3.2 String Overlap

Overlaps of q-grams quantify the closeness of strings. The more two bags overlap, the closer the strings are to each other. We define the overlap of two strings as the number of q-grams they share.

Definition 3.2 [Overlap of strings α and β]. Let α and β be two strings. Then the overlap of the strings is

$$o(\alpha, \beta) = |B(\alpha) \cap B(\beta)|,$$

where $|X|$ denotes the cardinality of set X.

Our clustering strategy is based on the overlap between strings. We cluster strings together if they have a high overlap, and we assign strings to different clusters if the overlap between strings is low.

Example 3.2 [Overlap of strings.] Let α_1 = jacob, α_2 = jacop, β_1 = syndni, β_2 = syndny. Then

$$o(\text{jacob}, \text{jacop}) = |\{\#j^1, ja^1, ac^1, co^1\}| = 4$$

Since the overlap between the strings is high, we assign α_1 and α_2 to one cluster. Similarly, since $o(\text{sydny}, \text{sydni}) = 4$, β_1 and β_2 are clustered together. On the other hand, since $o(\text{sydny}, \text{jacob}) = 0$, strings $\alpha_1, \alpha_2, \beta_1, \beta_2$ are not put into one cluster.

3.3 Inverse Strings

Inverse strings associate with each q-gram κ all string IDs that contain κ as a q-gram.

Definition 3.3 [Inverse string.] Let $\alpha_1, \ldots, \alpha_n$ be a dataset and κ be a q-gram. The inverse string is the set of all strings (string IDs) that have κ as a q-gram:

$$IS(\kappa) = \{\alpha_i : \kappa \in B(\alpha_i)\}$$

Example 3.3 [Inverse string.] Let the input database consists of six strings: $\alpha_1 = jacob$, $\alpha_2 = jacop$, $\alpha_3 = jakob$, $\alpha_4 = sydny$, $\alpha_5 = sydni$, $\alpha_6 = sydney$. The bags of 2-grams for each string are:

$$\begin{aligned}
B(\alpha_1) = B(\text{jacob}) &= \{\#j^1, ja^1, ac^1, co^1, ob^1, b\$^1\}\\
B(\alpha_2) = B(\text{jacop}) &= \{\#j^1, ja^1, ac^1, co^1, op^1, p\$^1\}\\
B(\alpha_3) = B(\text{jakob}) &= \{\#j^1, ja^1, ak^1, ko^1, ob^1, b\$^1\}\\
B(\alpha_4) = B(\text{sydny}) &= \{\#s^1, sy^1, yd^1, dn^1, ny^1, y\$^1\}\\
B(\alpha_5) = B(\text{sydni}) &= \{\#s^1, sy^1, yd^1, dn^1, ni^1, i\$^1\}\\
B(\alpha_5) = B(\text{sydney}) &= \{\#s^1, sy^1, yd^1, dn^1, ne^1, ey^1, y\$^1\}
\end{aligned}$$

The inverse string structure for all 2-grams is:

$$\begin{aligned}
IS(\#j^1) &= \{\alpha_1, \alpha_2, \alpha_3\} & IS(\#s^1) &= \{\alpha_4, \alpha_5, \alpha_6\}\\
IS(ja^1) &= \{\alpha_1, \alpha_2, \alpha_3\} & IS(sy^1) &= \{\alpha_4, \alpha_5, \alpha_6\}\\
IS(ac^1) &= \{\alpha_1, \alpha_2, \alpha_3\} & IS(yd^1) &= \{\alpha_4, \alpha_5, \alpha_6\}\\
IS(co^1) &= \{\alpha_1, \alpha_2, \alpha_3\} & IS(dn^1) &= \{\alpha_4, \alpha_5, \alpha_6\}\\
IS(ob^1) &= \{\alpha_1, \alpha_2, \alpha_3\} & IS(ny^1) &= \{\alpha_4\}\\
IS(b\$^1) &= \{\alpha_1, \alpha_3\} & IS(y\$^1) &= \{\alpha_4, \alpha_6\}\\
IS(ak^1) &= \{\alpha_3\} & IS(ni^1) &= \{\alpha_5\}\\
IS(p\$^1) &= \{\alpha_2\} & IS(i\$^1) &= \{\alpha_5\}\\
IS(ko^1) &= \{\alpha_3\} & IS(ne^1) &= \{\alpha_5\}\\
IS(op^1) &= \{\alpha_2\} & IS(ey^1) &= \{\alpha_5\}
\end{aligned}$$

The inverse strings data structure pre-clusters strings. Intuitively, the example database consists of two clusters with data distributed around centers α_1 = jacob and α_5 = sydney. The inverse strings structure reflects the clusters: part of inverse strings consists of string IDs from the first cluster (cf. the first column), while the other parts consists of the IDs of the second cluster (cf. the second column).

4 Cluster Computation

This section presents our clustering technique. First, we formalize the computation of the border b for each cluster (cf. Section 4.1). Second, we formalize the computation of center ζ of the cluster (cf. Section 4.2).

4.1 Border Detection

Assume a center string ζ of a cluster. The border detection algorithm aims to find the smallest radius that separates strings of this cluster from strings of other clusters. Since we compare strings with the help of overlaps, this border is the smallest overlap o that separates the cluster from other cluster.

The border is computed by examining $|C_d(\zeta)| = |\{\alpha : o(\alpha, \zeta) \geq d\}|$, i.e., the number of strings that have an overlap of at least d with ζ. Consider the following example.

Example 4.1 [Border detection.] We continue examle 3.3. Let ζ = jacob, q=2. We compute the database strings that have all 2-grams in common (overlap is $o = 6$) with jacob: $C_6(\text{jacob}) = \{\alpha_1\}$, the database strings that have all but one 2-gram: $C_5(\text{jacob}) = \{\alpha_1\}$. Similarly:

$$\begin{aligned}
C_4(\text{jacob}) &= \{\alpha_1, \alpha_2, \alpha_3\}\\
C_3(\text{jacob}) &= \{\alpha_1, \alpha_2, \alpha_3\}\\
C_2(\text{jacob}) &= \{\alpha_1, \alpha_2, \alpha_3\}\\
C_1(\text{jacob}) &= \{\alpha_1, \alpha_2, \alpha_3\}\\
C_0(\text{jacob}) &= \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6\}.
\end{aligned}$$

Figure 2 shows the size of $|C_d(\text{jacob})|$ as overlap o decreases (cf. Axis X from right to left). For large overlaps ($o = 5-6$) the size of the cluster increases. Then the cluster size stops to increase for a range of the overlaps ($o = 1-4$). This is an indication that the border of the cluster has been reached. As the overlap is further decreased the cluster starts to include points from other clusters resulting in a very fast increase of its size ($o = 0 - 1$).

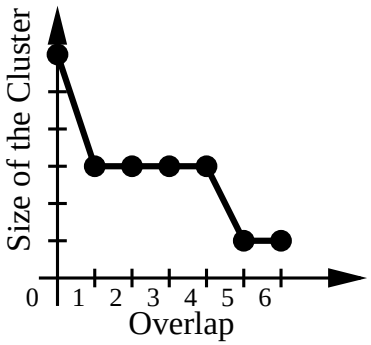


Figure 2: The String Proximity Graph

We compute the largest range of a constant size of the cluster (cf. $o = 1 - 4$ in Figure 2), and take the right end of the interval as the border.

The border detection algorithm takes a center string ζ and finds the border b of the cluster. We extend the notion of border detection for a bag of q-grams. Let $q = 2$. Then the following expressions are equivalent:

(i) b is the border for center string ζ = jacob

(ii) b is the border for the 2-grams $B(\text{jacob}) = \{\#j^1, ja^1, ac^1, co^1, ob^1, b\$^1\}$.

The extension of the border detection allows us to query for borders of centers that do not necessarily correspond to a database string (for example for a bag $\{\#j^1, ja^1, a\mathbf{X}^1, co^1, ob^1, b\$^1\}$). The motivation for this generalization comes from the computation of the center for a cluster and is discussed in detail in Section 4.2.

The following summarizes and defines the detection of the border.

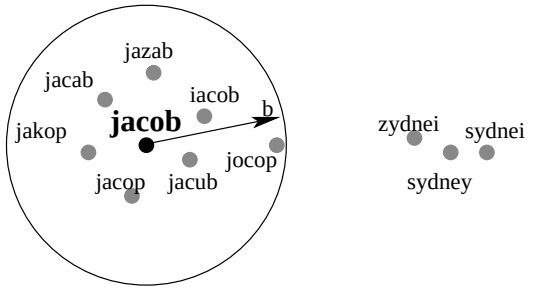
Definition 4.1 [Detection of the Border.] Let B be a (center) bag and $C_d(B) = \{\alpha : o(B(\alpha), B) \geq o\}$, $o = |B|, |B|-1, |B|-2, \ldots, 0$. Let $i_j, i_j+1, \ldots, i_j+k_{i_j}$ the longest sequence of unvarying sizes of the cluster:

$$|C_{i_j}(B)| = |C_{i_j+1}(B)| = \cdots = |C_{i_j+k_{i_j}}(B)|.$$

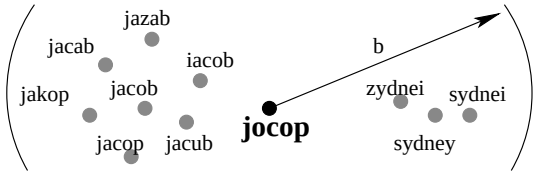
Then the border of the cluster with center B is $b = i_j$.

4.2 Computation of the Center

The border detection algorithm provides a simple and effective strategy to compute clusters in string data. One starts with a string in the database and selects the border that separates the cluster from the other clusters. If the initial string was chosen close to the center of the cluster, the border detection will yield good and robust results (cf. $\zeta =$ jacob, Figure 3(a)). If one chooses the initial string close to the border, two separate clusters might be assigned to one cluster (cf. $\zeta =$ jocop, Figure 3(a)).



(a) Cluster with Center String $\zeta =$ jacob



(b) Cluster with Center String $\zeta =$ jocop

Figure 3: Border Detection for Different Center Strings

The computation of the exact center for a given bag of strings B is expensive. One needs to compute distances between all strings in B and choose the one that minimizes the sum of distances from the center to other strings from B. We transform all strings into the space of bags of q-grams, and find the center bag there. The following example illustrates the computation.

Example 4.2 [Computation of the center for a given set of bags.] We continue Example 4.1. Let α_1, α_2, and α_3 be a set of strings. Then the set of bags for the strings is the following:

$$\begin{array}{lll} B(\alpha_1) & = B(\text{jacob}) = & \{\#j^1, ja^1, ac^1, co^1, ob^1, b\$^1\} \\ B(\alpha_2) & = B(\text{jakob}) = & \{\#j^1, ja^1, ak^1, ko^1, ob^1, b\$^1\} \\ B(\alpha_3) & = B(\text{jacap}) = & \{\#j^1, ja^1, ac^1, ca^1, ap^1, p\$^1\}, \end{array}$$

Our aim is to find a bag that represents bags $B(\alpha_1)$, $B(\alpha_2)$, and $B(\alpha_3)$. We compute such a bag in the following way. We compute the overall histogram for the set of bags, and neglect the infrequent 2-grams. The histogram of all 2-grams is presented in Figure 4 with the 2-grams in the second row, and the number of occurrences of the 2-gram in the first row.

3	3	3	3	2	1
$\#j^1$	ja^1	ac^1	ob^1	$b\$^1$	co^1

1	1	1	1	1
ak^1	ko^1	ca^1	ap^1	$p\$^1$

Figure 4: Histogram of 2-grams

The size of the center bag of 2-grams is determined by the average size S of the input bags $B(\alpha_1)$, $B(\alpha_2)$, and $B(\alpha_3)$. Therefore, the center bag is the following:

$$B^c = \{\#j^1, ja^1, ac^1, ob^1, b\$^1, co^1\}.$$

Note that the center bag might consist q-grams that correspond to typos in the input dataset. These occurrences do not decrease the quality of clustering. In fact, the opposite holds, since we are looking for a center bag that represents all the strings in the cluster as precisely as possible.

The following formalizes the computation of the center bag for a set of input bags.

Definition 4.2 [Center bag.] Let $B_1, B_2, \ldots, B_k$ be a set of input bags. Let

$$A = \frac{|B_1| + |B_2| + \cdots + |B_k|}{k}$$

be the average size of bags $B_1, B_2, \ldots, B_k$. Let

$$h(\kappa) = |\{B_i : \kappa \in B_i\}|$$

be the histogram value of q-gram κ. Let $\kappa_1, \kappa_2, \ldots, \kappa_m$ be an ordered sequence of q-grams of $B_1 \cup B_2 \cup \cdots \cup B_k$ such that $h(\kappa_i) \geq h(\kappa_{i+1})$. Then the center bag B is the set of q-grams:

$$B = \{\kappa_1, \kappa_2, \ldots, \kappa_A\}.$$

5 Sampling of Inverse Strings

In this section we show how to use inverse strings to identify strings that have an overlap with the center string above

a given threshold. First, we develop a mathematical formula that shows how to identify strings of high overlap. The result has combinatorial complexity. Second, we approximate the computation of high overlap strings with a help of sampling.

The IS data structure allows to quickly identify database strings that have selected q-grams in common. For example, if one wants to find all string IDs that share all 2-grams with the string jacob, one needs to compute the following expression:

$$IS(\#j^1) \cap IS(ja^1) \cap IS(ac^1) \cap IS(co^1) \cap IS(ob^1) \cap IS(b\$^1)$$

Similarly, if one wants to identify strings that contain all but one 2-gram of jacob, one needs to compute the following:

$$\begin{aligned}
&IS(ja^1) \cap IS(ac^1) \cap IS(co^1) \cap IS(ob^1) \cap IS(b\$^1) \bigcup \\
IS(\#j^1) &\cap IS(ac^1) \cap IS(co^1) \cap IS(ob^1) \cap IS(b\$^1) \bigcup \\
IS(\#j^1) \cap IS(ja^1) &\cap IS(co^1) \cap IS(ob^1) \cap IS(b\$^1) \bigcup \\
IS(\#j^1) \cap IS(ja^1) \cap IS(ac^1) &\cap IS(ob^1) \cap IS(b\$^1) \bigcup \\
IS(\#j^1) \cap IS(ja^1) \cap IS(ac^1) \cap IS(co^1) &\cap IS(b\$^1) \bigcup \\
IS(\#j^1) \cap IS(ja^1) \cap IS(ac^1) \cap IS(co^1) \cap IS(ob^1)&
\end{aligned}$$

Definition 5.1 [Computation of strings of high overlap with the help of the IS data structure.] Let B be a center bag, such that $\kappa_1, \kappa_2, \ldots, \kappa_o \in B$, and o be the overlap threshold. Let

$$O(\kappa_1, \ldots, \kappa_o) = IS(\kappa_1) \cap IS(\kappa_2) \cap \cdots \cap IS(\kappa_o). \quad (1)$$

The IDs of strings that have at least o q-grams from B can be computed with the following equation:

$$\bigcup_{\kappa_1, \kappa_2, \ldots, \kappa_o \in B} O(\kappa_1, \ldots, \kappa_o) \quad (2)$$

where $\kappa_1, \kappa_2, \ldots, \kappa_o$ are different q-grams of B.

The computation of the strings of high overlap with the help of the IS data structure is expensive. Let $|B|$ be the size of the bag of q-grams, and o be the desired overlap threshold. Then the computational complexity of the computation is $o \cdot \binom{|B|}{o}$ number of set operations (cf. equation (2)). We approximate the computation of equation (2) with the help of sampling. We select a small sample of different o-tuples $(\kappa_1^i, \kappa_2^i, \ldots, \kappa_o^i)$, $i = 1, 2, \ldots, S$, where S is the size of the sample, and compute the union of the intersections:

$$\bigcup_{i=1}^{S} IS(\kappa_1^i) \cap IS(\kappa_2^i) \cap \cdots \cap IS(\kappa_o^i) \quad (3)$$

Example 5.1 [Computation of strings of high overlap with the help of the IS data structure and sampling.] We continue example 4.2. Let the center bag be $B = \{\#j^1, ja^1, ac^1, co^1, ob^1, b\$^1\}$ (the bag of string jacob). Let the overlap threshold be $o = 5$ (all 2-grams except one) and let the sample size be $S = 3$.

The computation of approximated strings is done in three steps. First, we generate $S = 3$ random 5-tuples from B:

$$\begin{aligned}
\kappa^1 &= (\kappa_1^1, \ldots, \kappa_5^1) = (ja^1, ac^1, co^1, ob^1, b\$^1) \\
\kappa^2 &= (\kappa_1^2, \ldots, \kappa_5^2) = (\#j^1, ac^1, co^1, ob^1, b\$^1) \\
\kappa^3 &= (\kappa_1^3, \ldots, \kappa_5^3) = (\#j^1, ja^1, ac^1, co^1, ob^1)
\end{aligned}$$

Second, we compute the intersections for the 5-tuples:

$$\begin{aligned}
U_1(\kappa^1) &= IS(ja^1) \cap IS(ac^1) \cap IS(co^1) \cap IS(ob^1) \cap IS(b\$^1) \\
&= \{\alpha_1, \alpha_2, \alpha_3\} \cap \{\alpha_1, \alpha_2, \alpha_3\} \cap \{\alpha_1, \alpha_2, \alpha_3\} \\
&\quad \cap \{\alpha_1, \alpha_2, \alpha_3\} \cap \{\alpha_1, \alpha_2, \alpha_3\} \cap \{\alpha_1\} \\
&= \{\alpha_1\}.
\end{aligned}$$

Similarly, $U_1(\kappa^2) = \{\alpha_1\}$ and $U_1(\kappa^3) = \{\alpha_1\}$. Finally, we compute the union:

$$U(\kappa^1) \cup U(\kappa^2) \cup U(\kappa^3) = \{\alpha_1\}.$$

Therefore, the approximate database strings with overalp $o = 5$ and higher to the center string jacob are $\{\alpha_1\}$.

6 Algorithm

This section presents the algorithm of our data cleansing method. The algorithm cleanses data in 4 steps. First the algorithm initializes the variables (cf. block 1, Figure 5), then it clusters the string data (cf. block 2), merges overlapping clusters (cf. block 3), and finally it replaces the strings of a cluster with the most frequent string of the cluster (cf. block 4).

```
Input:
  D = {α_1, α_2, ..., α_n}:database of strings
  q:size of q-grams
  S:sample size

Output:
  α_1, α_2, ..., α_n:cleansed strings

Body:
  1. Initialize the clustered strings
     Clustered_Strings=∅, Clusters = ∅
  2. Scan database strings. For each α ∈ D do
     2.1 If α ∈ Clustered_Strings then start a new iteration with the
         next DB string (go to step 2). Otherwise compute initial
         center bag:B = B(α), max border: b_m = |B|. Initialize
         the current cluster O = ∅
     2.2 For each overlap threshold o = b_m − 1, ..., 1 do
         2.2.1 Compute approximate strings with center bag B
               and overlap threshold o. For i = 1, 2, ..., S
               2.2.1.1 Generate κ_1, ..., κ_o o-tuple of q-grams
               2.2.1.2 Compute the overlap strings
                       O = O ∪ O(κ_1, ..., κ_o) (cf. Eq (1))
         2.2.2 Update the center of the cluster.
               2.2.2.1 For each α ∈ O, for each κ ∈ B(α) do
                       update histogram h[κ] ← h[κ] + 1
               2.2.2.2 Sort h[κ] in descdending order
               2.2.2.3 Compute the average length of the strings
                       A = Σ_{α∈B} len(α)/|B|
               2.2.2.4 Assign the top A q-grams of the histogram
                       to the center bag B
            2.2.3 Record the cluster for overlap o:
                  2.2.3.1 Cluster[o] = B
     2.3 Find the longest sequence i_b, i_b + 1, ..., i_b + Δ
         such that |Cluster[i_b]| = ··· = |Cluster[i_b + Δ]|
     2.4 Update the clustered strings
         Clustered_Strings = Clustered_Strings ∪ Cluster[i_b]
     2.5 Insert a new cluster to the set of clusters
         Clusters = Clusters ∪ {Cluster[i_b]}
     2.6 Empty h[κ], O, B
  3. Merge overlapping clusters. For each C_i, C_j ∈ Clusters do
     if C_i ∩ C_j ≠ ∅ then C_i ← C_i ∪ C_j
  4. Clean the clusters. For each cluster C_i ∈ Clusters do
     4.1 Find the most frequent string φ in C_i.
         Replace all strings α ∈ C_i with φ.
```

Figure 5: Data Cleansing Algorithm

Block 2 (cf. Figure 5) clusters the string data. It starts with a non clustered string α (block 2.1) and computes string IDs that have overlap with the center string (cf. Figure 2) for different overlap thresholds. For each overlap o the algorithm computes the strings of high overlap (block 2.2.1), and adjusts the center bag of the cluster (cf. block 2.2.2, Section 4.2). Then the method detects the border of the cluster (block 2.3), inserts the newly found cluster (block 2.4), and removes the IDs of clustered strings from the database (block 2.5). Four data containers are used to implement the clustering step: histogram of q-grams for the current cluster $h[\kappa]$ (cf. definition 4.2), center bag B, set of strings that have overlap o and higher wrt the center bag B (the container increases as o decreases), and set of strings for each overlap threshold o (the container is not affected by the increase of o). All containers are main memory data structures and are implemented as sorted associated containers for fast point-queries.

Block 3 merges overlapping clusters and block 4 cleanses clusters with the most frequent string of the cluster (the reasoning is that most of the strings are entered correctly, and the data consists only of a smaller number of strings with typos). Alternatively, one can identify the string ζ that shares the largest number of q-grams with the center bag, and use string ζ as the correct string for cleansing.

The intersection of inverse strings $IS(\kappa_1) \cap \cdots \cap IS(\kappa_d)$ (Block 2.2.1.2) is the most expensive part of the algorithm. We implemented and tested four different approaches of the computations of the intersection. Let $\kappa_1, \kappa_2, \ldots, \kappa_o$ be a sequence of the q-grams of a center string (in some random order). Then the implemented strategies are the following:

(i) Scan all inverse strings simultaneously, i.e., let $i = (i(\kappa_1), i(\kappa_2), \ldots, i(\kappa_o))$ be an index vector that scans $(IS(\kappa_1), IS(\kappa_2), \ldots, IS(\kappa_o))$. If all the components of index i point to the same string ID, then the cluster size is incremented, and all components of i are incremented. Otherwise, only index $i(\kappa_i)$ is incremented, if $IS(\kappa_i)$ contains the smallest string ID. Note that we require that inverse strings are ordered according to the string ID.

(ii) Organize the computation of the intersection as a sequence of intersections of two inverse strings, for e.g.:

$$((IS(\kappa_1) \cap IS(\kappa_2)) \cap IS(\kappa_3)) \cap IS(\kappa_4)$$

The strategy can be formalized in the following way. Let $IN_{i+1} = IN_i \cap IS(\kappa_{i+1})$, $i = 2, \ldots, o$, $IN_1 = IS(\kappa_1)$, then

$$IS(\kappa_1) \cap IS(\kappa_2) \cap \cdots \cap IS(\kappa_o) = IN_o(\kappa_o)$$

(iii) The same strategy as (ii) though the sequence is sorted started with the smallest inverse string, i.e., $|IS(\kappa_i)| \leq |IS(\kappa_{i+1})|$.

(iv) Similar strategy to (ii), though intersections are organized into a bushy tree:

$$\Big(\big(IS(\kappa_1) \cap IS(\kappa_2)\big) \cap \big(IS(\kappa_3) \cap IS(\kappa_4)\big)\Big)$$

The following recurrent equations formalizes the computation:

$$\begin{aligned} IN_i^0 &\leftarrow I(\kappa_i) \\ IN_{i+1}^j &\leftarrow IN_{2i-1}^{j-1} \cap IN_{2i}^{j-1} \\ IN_{\lfloor o/2^j \rfloor}^j &\leftarrow IN_{\lfloor o/2^j \rfloor}^j \cap IN_{\lfloor o/2^{j-1} \rfloor}^{j-1} \text{ iff } 2^j \not| o \end{aligned}$$

where $i = 1, 2, \ldots, \lfloor o/2^j \rfloor$, $j = 1, \ldots, \log_2 o$. Then the intersection can be rewritten as follows:

$$IS(\kappa_1) \cap \cdots \cap IS(\kappa_o) = IN_1^{\log_2 o}.$$

The results on different datasets has showed that strategy (ii) outperformed the other strategies by at least 30%. Therefore, we used strategy (ii) in our experiments. However, other alternatives might be more beneficial for distributed environment and in connection with caching techniques (cf. strategy (iv)).

7 Experiments

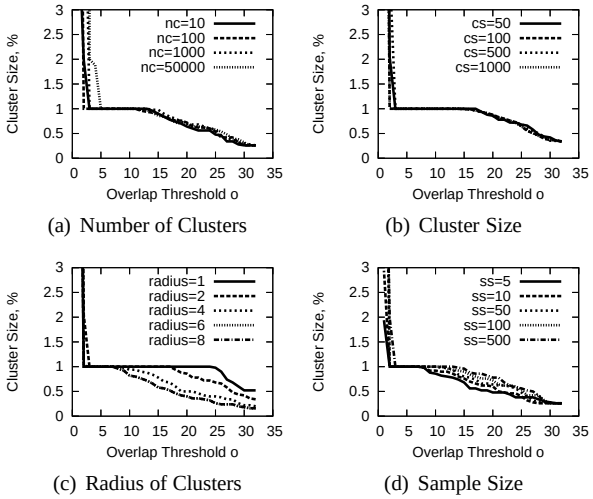
We organize the experiments in two sub-sections. First, we evaluate border detection criteria (cf. Section 7.1) and then we evaluate our cleansing method (cf. Section 7.2). We use synthetic datasets with different parameters in our experiments. Three classes of databases were generated in the experiments: (i) a class of databases with different number of clusters (nc), (ii) a class of databases with different cluster sizes (cs), and (iii) a class of databases with different radius of clusters ($radius$). All datasets were generated in the following way. First we generated nc number of center strings far away from each other. Then for each center string we generated cs number of strings in e edit distance[1] from the center string, where $0 \leq e \leq radius$.

7.1 Border Detection

Figure 6 shows the experiments for our border detection algorithm for different number of clusters (cf. Figure 6(a)), cluster sizes (cf. Figure 6(b)), radius of the cluster (cf. Figure 6(c)), and sample size (cf. Figure 6(d)). All figures varies overlap from around $o = |B| = 35$ to $o = 1$ (cf. Axis X from right to left in Figure 6). Y axis reports the fraction of the size of the cluster that is covered by the overlap threshold o. There are three intervals of overlaps in the graphs: an interval $I_<$ of overlaps o that does not cover the entire cluster (cf. interval 35–17, Figure 6(b)), interval $I_=$ of overlaps that cover exactly the cluster (cf. rage 16–4, Figure 6(b)), and interval $I_>$ of overlaps that

[1] edit distance between string α and string β is the smallest number of character- insertions, deletions, and substitutions required in order to get string α from string β.

cover more strings than there are in the cluster (cf. range 3–0, Figure 6(b)). The border detection works if there is a (relatively long) interval of overlaps that covers the cluster exactly.



(a) Number of Clusters (b) Cluster Size

(c) Radius of Clusters (d) Sample Size

Figure 6: Border Detection

Border detection algorithm successfully identifies borders of clusters provided a sufficient sample size.

The robustness of the algorithm is not affected by the cluster size (cf. Figure 6(b)). Indeed, the length of interval $I_=$ depends on the distance between the borders of the clusters and does not depend on the cluster size.

The robustness of the border detection is almost invariant to the number of clusters (cf. Figure 6(a)). As the number of clusters increases from 10 to $50,000$ the start of interval $I_=$ shifts from 16 to 13. However, the impact of the shift is negligible compared to the length of $I_=$, and therefore the border detection ensures robust results.

Radius of clusters (cf. Figure 6(c)) and sample size (cf. Figure 6(d)) impacts more significantly the robustness of border detection. The length of $I_=$ proportionally decreases as the radius decreases (by two for each decrease in radius). Decrease of the sample size lowers the shape of the curve, decreases the length of $I_=$, and in turn decreases the robustness of the border detection. However, we want to have the sample size as small as possible, since the smaller sample size means a lower computational time of data cleansing.

Figure 6(d) confirms that very small samples can be used to approximate the border detection robustly (cf. $ss = 10$ with the total number $\binom{35}{17} \approx 4.5 \times 10^9$ of intersection computations (cf. equation (1)) for the overlap threshold $o = 17$!)

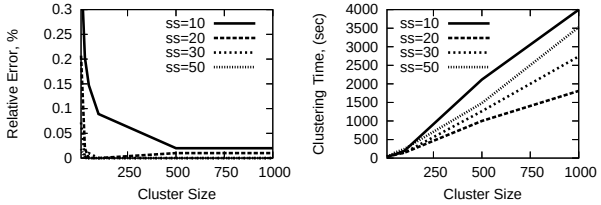
The default parameters in the series of experiments were: length of strings $l \approx 30$, number of cluster $nc = 100$, cluster size $cs = 50$, sample size $ss = 100$, cluster radius $radius = 3$.

7.2 Cleansing

We evaluate our cleansing algorithm for different cluster sizes (cf. sub-section 7.2.1) and different number of clusters (cf. sub-section 7.2.2). Two measurement are recorded for the experiments: relative error (recorded in relative number of misclustered strings compared to the total number of strings in the clusters) and clustering time (seconds).

7.2.1 Different Cluster Sizes

As the cluster size increases, the relative clustering error decreases (cf. Figure 7(a)). This is because the border detection algorithm is very effective, and the number of correctly clustered strings increases vs. the total number of strings in the cluster.



(a) Relative Error (b) Clustering Time

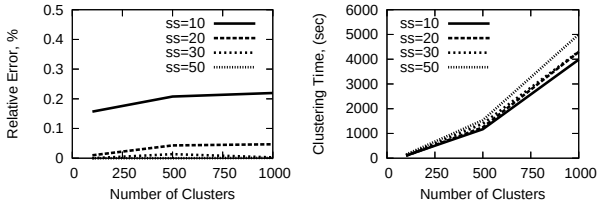
Figure 7: Different Cluster Sizes

The clustering time increases linearly as the number of strings per cluster increases (cf. Figure 7(b)). However a lower sample size does not necessarily mean a faster clustering time. This is because inadequately small sample size increases the number of total clusters, and in turn increases the number of iterations of the algorithm.

The default parameters in this series of experiments were: length of strings $l \approx 30$, number of cluster $nc = 100$, cluster radius $radius = 3$.

7.2.2 Different Number of Clusters

The relative error increases very slightly as the number of strings increases, (cf. Figure 8(a)). This is because the sharp borders between the inverse strings of different clusters gets blurred as the number of clusters increases. Note that our sampling technique is very effective: even a very small increase of the sample size (cf. $ss = 10$ and $ss = 20$) significantly reduces the relative error.



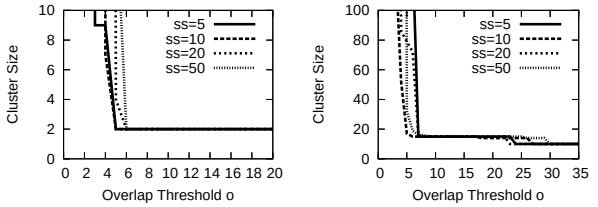
(a) Relative Error (b) Clustering Time

Figure 8: Different Cluster Sizes

The clustering time (cf. Figure 8(b)) increases linearly as the number of clusters increases. In contrast to the cluster size experiment (cf. Section 7.2.1) the clustering time for smaller samples does not exceed the clustering time for larger samples, since the number of clusters is very large compared to the size of the clusters.

7.3 Real World Data

This section evaluates the border detection algorithm for real world company names (database with around 15 character long strings) and company addresses (database with around 30 character long strings). Both databases consists of clusters that are far away from each other and a small number of strings within the clusters (cf. Figure 9). There is a large range of overlap levels for which the cluster size is constant (cf. o=[20–7] for the company names and o=[22–7] for the company addresses), and therefore our border detection algorithm detects the border correctly even for very small sample sizes. Our clustering algorithm detected small clusters (1–3 strings per cluster) for the company names and larger clusters (3–30 strings per cluster) for company addresses.



(a) Company Names (b) Company Addresses

Figure 9: String Proximity Graph for Real World Data

Intuitively, our data cleansing algorithm produces good cleansing results for string data with large distances between centers of clusters and small distances within the clusters. Examples of such datasets are databases of company names and company addresses. Our data cleansing algorithm is less applicable for natural language databases. In such databases two strings that are close to each other might have a very different meaning and therefore should be assigned to different clusters (for example “air” and “aim”, or “spouse” and “mouse”). In natural language databases spelling based and dictionary based techniques are more appropriate. For proper noun databases typically no dictionaries exists and the proposed solution is the preferred choice.

8 Conclusions and Future Work

In this paper we present our results of a new data cleansing algorithm. Data cleansing is done in two steps. First, the string data is clustered by identifying center and border of hyper-spherical clusters, and second, the cluster strings are cleansed with the most frequent string of the cluster. Clustering starts with a non-clustered string and computes the border b of the cluster. All strings within the overlap threshold b from the center of the cluster are assigned to one cluster. Experiments show that the border detection is robust provided a sufficient sample size.

There are a number of research directions for future work. One can further progress the IS data structure. Our investigation indicates that very few q-grams of the center strings are sufficient to identify strings of the cluster. An algorithm that robustly finds the identifying q-grams of the cluster is an interesting challenge.

The data cleansing method is robust if the distance between the clusters is large compared to the diameters of the clusters. In order to improve the precision for databases with small distances between the clusters one can introduce a number of string representatives for each cluster.

References

[1] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *VLDB*, pages 586–597, 2002.

[2] S. Chaudhuri, K. Ganjam, V. Ganti, and R .Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, pages 313–324, 2003.

[3] S. Chaudhuri, V. Ganti, and L. Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *ICDE:*, pages 227–239, 2004.

[4] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[5] W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string metrics for matching names and records. In *Data Cleaning Workshop in Conjunction with KDD*, 2003.

[6] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[7] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an rdbms for web data integration. In *WWW*, pages 90–101, 2003.

[8] D. Gusfield. *Algorithms on strings, trees and sequences: Computer science and computational biology*. Cambridge University Press, Cambridge, UK, 1997.

[9] V. J. Hodge and J. Austin. A comparison of standard spell checking algorithms and a novel binary neural approach. *TKDE*, 15(5):1073–1081, 2003.

[10] H. V. Jagadish, N. Koudas, and D. Srivastava. On effective multi-dimensional indexing for strings. In *SIGMOD*, pages 403–414, 2000.

[11] L. Jin and C Li. Selectivity estimation for fuzzy string predicates in large data sets. In *VLDB*, pages 397–408, 2005.

[12] L. Jin, C. Li, N. Koudas, and A. K. H. Tung. Indexing mixed types for approximate retrieval. In *VLDB*, pages 793–804, 2005.

[13] K. Kukich. Technique for automatically correcting words in text. *ACM Comput. Surv.*, 24(4):377–439, 1992.

[14] S. Sahinalp, M. Tasan, J. Macker, and Z. Ozsoyoglu. Distance based indexing for string proximity search. In *ICDE*, 2003.