Context-aware Approximate String Matching for Large-scale Real-time Entity Resolution

Peter Christen
Research School of Computer Science
The Australian National University
Acton ACT 2601, Australia
Email: peter.christen@anu.edu.au

Ross W. Gayler *
Veda
Melbourne VIC 3000, Australia

***Abstract*—Techniques for approximate string matching have been widely studied over several decades. They are required in many applications, including entity resolution, spell checking, similarity joins, and biological sequence comparison. Most existing techniques for approximate string matching used in entity resolution only consider the two strings that are compared. They neglect contextual information such as the frequency of how often strings occur in a database, the likelihood of the character edits between strings, or how many other similar strings there are in a database. In this paper we investigate if incorporating such contextual information into edit distance based approximate string matching can improve matching quality for real-time entity resolution. In this application, query records have to be matched in sub-second time to records in a large database that refer to the same entity. We evaluate our approach on two large real data sets and compare it to several baseline approaches. Our results show that considering edit frequency and the neighborhood size of a string can improve matching results, while taking string frequencies into account can actually make results worse.**

I. Introduction

In its basic form, approximate string matching is the task of comparing two strings (with symbols from a given alphabet) that are not the same. Either a similarity score or alternatively a positive distance, such as edit distance (ED) [16], between two strings is calculated. Similarities are often normalized into the $[0, 1]$ interval, where a similarity of 0 means two strings are completely different, 1 means they are the same, and a value in-between means they are somewhat similar [6].

Many different string matching techniques have been developed, some applicable for any type of strings, while others are designed for strings with specific characteristics. For example, the Jaro-Winkler comparator [6] is aimed at English personal name strings, the SoftTFIDF comparator [8] is for strings that contain several tokens (such as words separated by whitespace), and variations of ED such as the Smith-Waterman and Needleman-Wunsch [16] algorithms are aimed at comparing very long sequences such as those occurring in bioinformatics.

We focus on ED, which counts how many edit operations are required to convert one string into another [16]. For example, 'petra' can be converted into 'pedro' by substituting 't' with 'd' and 'a' with 'o', resulting in an ED of 2.

While approximate string matching is required in many application areas [16], our focus is on real-time entity resolution [4], [7], [15], [17], the task of matching query records to a database of entity records in sub-second time to retrieve the database records that refer to the same real-world entity. A major challenge in entity resolution is the lack of entity identifiers in the databases to be matched. Therefore, calculating the similarities between records requires the comparison of attributes that contain identifying information about entities, such as names and addresses for people and businesses, or names and descriptions of consumer products. Real-time entity resolution is crucial in various applications, such as when police officers need to identify a suspect within seconds when conducting identity checks based only on personal details, or when financial institutions have to evaluate the credit history of a person who applies for consumer credit.

The majority of existing string matching techniques used in entity resolution do not take any contextual information from the databases on which they are applied to into account, but rather consider a pair of strings in isolation. Possible contextual information includes the frequencies of two strings to be matched, the likelihood of edits between them, and the number of other strings that are highly similar to the two strings (i.e. their neighbors). Our hypothesis is that such contextual information – as collected from the database(s) to be queried or matched – can help to improve matching and better distinguish between true and false matches.

Only a few string matching techniques used for entity resolution make use of contextual information. SoftTFIDF [8] takes the frequency of string values into account when calculating the similarity between multi-word values. An alternative approach is to learn costs for different edit operations [5] (such as for character inserts, deletes, and substitutions) using a supervised classifier based on training data which are in the form of string pairs that correspond to true matches (refer to the same real-world entity) and true non-matches (refer to different entities). In our work, we estimate the likelihood of edit operations for true matches using a large string database by only considering pairs of strings that have a small ED.

As shown in Figure 1, from analyzing the string data sets we will use in our experiments in Section VI, we found that the vast majority of string pairs that have an ED of 1 correspond to true matches, while pairs with a large ED are most likely to be true non-matches. This is in line with studies on spelling and typing errors which showed that the majority of such errors result in an ED of only 1 or 2 [12], [16]. In our approach we consider frequency information of the edit operations between string pairs when matching query with database strings.

* Current email address: r.gayler@gmail.com

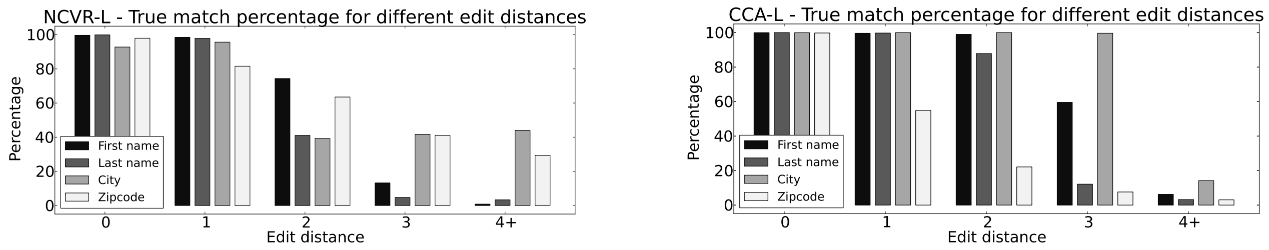


Fig. 1: True match (referring to the same entity) percentages for different edit distances between strings from the two large data sets used in the experiments in Section VI.

Our contributions are (i) a novel approximate string matching approach that incorporates different pieces of evidence about the likelihood that a given query string should actually be an existing database string (i.e. the two strings are a correct match); (ii) an efficient index data structure that allows calculation of the required similarities and weights in real-time, and that can be updated at run-time as new strings are added to the index; and (iii) an experimental evaluation on two large real data sets illustrating the suitability of our approach to improve ranking when matching query records in real-time entity resolution.

II. Related Work

The literature on techniques for approximate string matching is vast. It ranges from theoretical work that investigates approaches to how ED can be calculated in near-linear times [1], to how approximate matching techniques can be implemented efficiently into relational databases [11]. Several comprehensive surveys are available [10], [16]. However, most work has concentrated on either developing novel string matching techniques that are domain specific, or efficient index structures to scale-up the matching of large numbers of strings in a database setting [3], [20]. Similarly, much work has been conducted on string matching within spell-checking systems [2], and for entity resolution [6].

Only a few works have considered using information about the frequencies of strings, as we do in our approach. The SoftTFIDF approach by Cohen et al. [8] takes the frequency of string values into account when calculating TF-IDF weights to be used in a two-level approximate matching technique aimed at multi-word strings such as paper titles in bibliographic data. SoftTFIDF assigns different weights to individual words, with larger weights given to words that are less common. Lange and Naumann [13] show why 'Arnold Schwarzenegger' is always a duplicate (as this is generally a very rare name), and they develop an approach where string values are grouped into three partitions (frequent, medium, and rare values) using a genetic partitioning algorithm and using a set of true matching values. A different classifier is then learned for each partition, resulting in significantly improved matching quality.

Work on similarity joins in the database community mostly considers static databases on which suitable efficient index structures can be built [14], [18], [20]. It is commonly assumed that these databases, and therefore the corresponding index structures, do not change over time. Our approach, on the other hand, can be updated dynamically by inserting new strings and updating the appropriate string and edit frequencies.

The idea of learning the characteristics of spelling errors to improve the accuracy of matching strings in entity resolution has been explored by Bilenko and Mooney [5]. Their approach is to learn the costs for different edit operations (like inserts or deletes) using a support vector machine (SVM) classifier. Unlike this approach, we use the likelihood of individual edits and edit sequences learned from a database, instead of the learned costs of different edit operations. More recently, Sukharev et al. [19] used very large genealogical databases of names to model and rank name variations using an approach based on a probabilistic language model and statistical machine translation techniques. This proposed approach outperformed both phonetic and approximate string comparisons techniques in a set of extensive experiments.

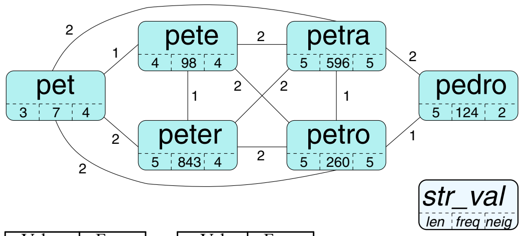
In the domain of entity resolution, few works have investigated real-time matching [4], [7], [15], [17]. Similar to our approach, these techniques commonly pre-calculate the similarities between strings during the build time of an index to speed-up query matching times. However, none of these techniques specifically considers contextual information within approximate string matching, and therefore these techniques are complementary to our proposed approach.

III. Problem Definition and Overview of Approach

We assume a database $\mathbf{D}$ that contains entity records r. Each record contains several attributes, with A the set of attributes used for string matching. The value in attribute $a \in A$ in record r is denoted as $r.a$. S_a is the set of all unique strings in a from $\mathbf{D}$, and $|r.a|$ is the length of string $r.a$ in characters. q is a query record that contains the same attributes as the records in $\mathbf{D}$. The value in attribute a of q is denoted with $q.a$, with either $q.a \in S_a$ or $q.a \notin S_a$. In the latter case, depending upon the application, if q is added to $\mathbf{D}$ then $q.a$ is added to S_a. We assume a function $dist_{ED}(s_i, s_j)$ which calculates the edit distance (ED) [16], counted as the number of character edits, between two strings s_i and s_j, with $s_i \neq s_j$. An ED can be converted into a normalized edit similarity as

$$sim_{ED}(s_i, s_j) = 1.0 - dist_{ED}(s_i, s_j)/max(|s_i|, |s_j|). \quad (1)$$

As can be seen from Figure 1, the vast majority of variations and errors of string pairs that refer to true matching records in the large data sets we use in our experiments in Section VI have an ED of 0 or 1. We use this property to develop a data structure (illustrated in Figure 2) to efficiently store and retrieve strings and their contextual information, as we describe in Section IV. We denote the maximum ED to be considered in our approach as m, with $m \geq 1$.



Value	Freq
pedro	124
pet	7
pete	98

Val	Freq
peter	843
petra	596
petro	260

String pair	Edit	Freq
pedro, petra	(d $\leftrightarrow$ t, o $\leftrightarrow$ a)	114
pedro, petro	(d $\leftrightarrow$ t)	1839
pet, pete	($\epsilon \leftrightarrow$ e)	991
pet, peter	($\epsilon \leftrightarrow$ e, $\epsilon \leftrightarrow$ r)	84
pet, petra	($\epsilon \leftrightarrow$ r, $\epsilon \leftrightarrow$ a)	123
pet, petro	($\epsilon \leftrightarrow$ r, $\epsilon \leftrightarrow$ o)	148
pete, peter	($\epsilon \leftrightarrow$ r)	780
pete, petra	(e $\leftrightarrow$ r, $\epsilon \leftrightarrow$ a)	42
pete, petro	(e $\leftrightarrow$ r, $\epsilon \leftrightarrow$ o)	67
peter, petra	(e $\leftrightarrow \epsilon$, $\epsilon \leftrightarrow$ a)	145
peter, petro	(e $\leftrightarrow \epsilon$, $\epsilon \leftrightarrow$ o)	231
petra, petro	(a $\leftrightarrow$ o)	811
peta, pete	(a $\leftrightarrow$ e)	151
peta, petra	($\epsilon \leftrightarrow$ r)	780

Fig. 2: An example of our proposed approach, with six strings and edges of ED 1 and 2 between them (i.e. $m = 2$). The frequencies of values and edits are assumed to have been counted from a large string database. The empty character is denoted by ϵ. '*peta*' is an unknown query string, as described in the example in Section III.

We define an *edit path*, e, as a sequence of one or more (up to m) character edits and a function $path(s_i, s_j)$ which returns all possible edit paths between s_i and s_j. Each individual edit is a character pair of the form (c_{from}, c_{to}), with $c_{from}, c_{to} \in (\Sigma \cup \epsilon)$, where Σ is the alphabet from which strings $s \in S$ are generated ($s \in \Sigma^*$) and ϵ is the empty character (i.e. $|\epsilon| = 0$). For example, the edit path that changes 'pete' into 'peter' (see Figure 2) is an insert of 'r' which corresponds to $path(\textit{'pete', 'peter'}) = [(\epsilon, r)]$, while the change from 'peter' to 'petra' would be one of the two edit sequences $[(e, \epsilon), (\epsilon, a)]$ or $[(e, r), (r, a)]$. We denote the set of all edit paths up-to a maximum length m extracted from a set of strings S as $E = \{path(s_i, s_j) : s_i, s_j \in S : dist_{ED}(s_i, s_j) \le m\}$.

In the remainder of this section we consider one attribute only, and drop the subscript a if it is not required in the context (i.e. if applicable to any attribute). We discuss how to combine the similarity calculations from several attributes in Section V. We use the shorthand $s_q = q.a$. We now consider the following three properties of strings in S_a, and then define the problem we aim to solve:

- $f_s(s) = |\{r \in \mathbf{D} : r.a = s\}|$ is the frequency of occurrence of string $s \in S_a$ in attribute a in $\mathbf{D}$.
- $f_e(e) = |\{s_i, s_j \in S : dist_{ED}(s_i, s_j) \le m : e \in path(s_i, s_j), e \in E\}|$ is the frequency of occurrence of edit path e, counted as the number of times e occurs in string pairs in S that have an ED of up-to m.
- $n(s_i, m) = |\{s_i, s_j \in S : dist_{ED}(s_i, s_j) \le m\}|$ is the number of neighboring strings s_j of s_i that have an ED up-to m.

Definition 1 (Context-aware approximate string matching): Given query string s_q and a set of strings S from database $\mathbf{D}$ (potentially $s_q \in S$), we want to find the string $s_i \in S$ with the highest propensity $p_{s_q \to s_i}$ that s_q actually should be s_i (i.e., s_q is a mistyped or corrupted version of s_i) where we define

$$p_{s_q \to s_i} \propto Pr(s_q \to s_i | d_{q,i}, f_s(s_i), f_e(e_{q,i}), n(s_i, d_{q,i})), \quad (2)$$

with $d_{q,i} = dist_{ED}(s_q, s_i)$, and $e_{q,i}$ the most likely edit path that converts s_q into s_i.

Equation (2) takes into account relevant properties of the database string s_i and query string s_q within the context of attribute a in $\mathbf{D}$ where s_i occurs. Most existing approximate string matching techniques used in entity resolution do not take any contextual information into account, and only few consider one of the three properties of strings given above [5], [8], [13]. We are not aware of any techniques that consider all three properties as we do in our proposed approach.

The conditional probability in Equation (2) is an unknown function of the string properties we have hypothesized are relevant (such as $d_{q,i}$, $f_s(s_i)$, etc). If we had the conditional probabilities and string properties, we could assume a parameterized functional form and learn the mapping from string properties to conditional probability. An alternative approach would be to assume a very restrictive form (such as naive Bayes) and directly calculate the estimated conditional probabilities.

We have taken a modified version of the second approach by defining terms that we assume capture the qualitative effects of string properties, and assumed a functional form for combining those terms, as we will discuss in Section V. The resulting value is a heuristic score that we expect to be monotonically related to the true conditional probability.

As an example, following Figure 2 and assuming $m = 1$, for an unknown query string 'peta', we have both 'pete' and 'petra' with ED 1. 'petra' is nearly six times more common than 'pete' ($f_s(\textit{'petra'}) = 596$ and $f_s(\textit{'pete'}) = 98$), the edit path between 'peta' and 'petra' is more frequent ($f_e([(\epsilon, r)]) = 780$) than the one between 'pete' and 'peta' ($f_e([(a, e)]) = 151$), and 'petra' has one neighbor with an ED of 1 ($n(\textit{'petra'}, 1)) = 1$) while 'pete' has two ($n(\textit{'pete'}, 1)) = 2$). Assuming independence between the different factors in Equation (2), as we will discuss in Section V, an example calculation of the score of 'peta' $\to$ 'petra' is (showing $f_s()$, $f_e()$ and $n()$) $\frac{596}{596+98} \times \frac{780}{780+151} \times \frac{1}{1} = 0.719$, while for 'peta' $\to$ 'pete' it is $\frac{98}{596+98} \times \frac{151}{780+151} \times \frac{1}{2} = 0.012$. Therefore, based on this contextual information, 'petra' is predicted by the model as more likely to be the actual matching string for 'peta' than 'pete'. Assuming only ED would be used, both 'pete' and 'petra' would have an ED of 1 with 'peta'.

Overview of Proposed Approach: The real-time matching of query strings s_q with strings in the set S consists of two phases. In the first, described in the following section, we build an efficient index data structure (named *EDIndex*) by pre-calculating the ED for all string pairs $s_i, s_j \in S$ that have $dist_{ED}(s_i, s_j) \le m$. We build a graph connecting all such pairs, and as shown in Figure 2, we store the lengths (len)

Algorithm 1: ***Generate EDIndex for attribute*** a

```
Input:
- $\mathbf{D}$: Database containing string values
- $a$: Attribute to use
- $m$: Maximum ED to consider
Output:
- $\mathbf{S}_a$: Inverted index of strings in attribute $a$ (frequencies and record identifiers)
- $\mathbf{P}_a$: EDIndex graph of string pairs from $a$ with ED $\leq m$
- $\mathbf{E}_a$: Inverted index of edit paths and their frequencies from $a$
1:  $\mathbf{P}_a$ := [], $\mathbf{E}_a$ := []; $\mathbf{C}_a$ := {}
2:  $\mathbf{S}_a$ := GenInvIndex($\mathbf{D}$, $a$)     // Generate inverted index of all strings from $a$
3:  $\mathbf{L}_a$ := GenLengthIndex($\mathbf{S}_a$)   // Generate one list of strings per string length
4:  for $l \in \mathbf{L}_a$ do:                  // Loop over string lengths
5:      if $m = 1$ then:               // Maximum one edit in a string
6:          $\mathbf{I}$ := GenHalfLenIndex($\mathbf{L}_a$, $l$, $m$)
7:      else if $m = 2$ then:              // Maximum two edits in a string
8:          $\mathbf{I}$ := GenThreePartIndex($\mathbf{L}_a$, $l$, $m$)
9:      else:                              // More than two edits in a string
10:         $\mathbf{I}$ := GenQGramIndex($\mathbf{L}_a$, $l$, $m$)
11:     $\mathbf{C}_a$ := $\mathbf{C}_a \cup$ GenCandPairs($\mathbf{I}$)   // Generate string pairs from this length
12: for $(s_i, s_j) \in \mathbf{C}_a$ do:           // Check ED for all candidate string pairs
13:     if $dist_{ED}(s_i, s_j) \leq m$:        // Insert string pair into EDIndex
14:         Add $(s_i, s_j)$ to $\mathbf{P}_a$ and $path(s_i, s_j)$ to $\mathbf{E}_a$
```

and frequencies ($freq$) of string values, and the number of their neighbors ($neig$). We also store all edit paths in E up-to length m and their frequencies in a separate data structure.

In the second phase, described in Section V, we assume a stream $\mathbf{Q}$ of query records q that are to be matched (over several attributes $a \in A$) with the strings in the EDIndex. For each $q \in \mathbf{Q}$, we build an accumulator [3] M_q that consists of a list of matched candidate records $r \in \mathbf{D}$ ranked according to the predicted likelihood that they match with q. We describe several approaches to efficiently calculate Equation (2).

IV. Building the Edit Distance String Index

Assuming a database $\mathbf{D}$ of records, for each attribute $a \in A$ to be used in string matching we build an EDIndex. For this we generate (i) the inverted index $\mathbf{S}_a$ of unique strings and their frequencies (i.e. how many records in $\mathbf{D}$ have a certain value s in attribute a), (ii) the graph $\mathbf{P}_a$ of string pairs in $\mathbf{S}_a$ that have an ED of m or less, and (iii) the inverted index $\mathbf{E}_a$ of edit paths between these pairs and their frequencies. In the following we discuss each of these in detail.

Algorithm 1 illustrates the steps involved in building the EDIndex for one attribute a. First we initialize the required data structures, including $\mathbf{C}_a$ which is the set of candidate string pairs that are to be checked if their ED is m or less. In line 2 we build the inverted index $\mathbf{S}_a$ where each unique string in a from database $\mathbf{D}$ becomes a key. We store in $\mathbf{S}_a$ the frequency of each unique string, as well as a list of the identifiers of the records that have that string.

As we are only interested in string pairs with an ED of m or less, we employ several efficient filtering techniques to reduce the number of pairs that have to be compared [2], [3], [10], [11], [20]. Specifically, we only have to compare a string of length l with strings of lengths l to $l + m$, as any larger length difference requires more than m inserts, leading to an ED larger than m. In line 3 of Algorithm 1 we build the length index $\mathbf{L}_a$ which inserts all strings of the same length into one list, and in line 4 we loop over the different string lengths.

Depending upon the value of m we can use special filtering approaches [11]. If $m = 1$, then the single edit between two strings can only affect the first or the second half of a string, but not both. The *GenHalfLenIndex* function (line 6) returns an inverted index $\mathbf{I}$ with half-length strings as keys and all corresponding strings as values. If $m = 2$ then we can similarly split a string into three partitions, as two edits can only affect two of the three. The *GenThreePartIndex* function (line 8) generates such an inverted index. For values $m > 2$ we generate a positional q-gram index [11] using the *GenQGramIndex* function (line 10), which returns an inverted index $\mathbf{I}$ with positional q-grams as keys and all strings that have a certain positional q-gram in the corresponding index list. In line 11 we generate all candidate string pairs from $\mathbf{I}$ and add them to the candidate set $\mathbf{C}_a$.

In lines 12 to 14 of Algorithm 1, all candidate string pairs in $\mathbf{C}_a$ are compared using an optimized ED function $dist_{ED}()$. ED is generally calculated using a dynamic programming algorithm [16] which has a quadratic complexity in the length of the two compared strings. As we are only interested in string pairs with a maximum ED of m, we can truncate (abandon) an ED calculation once it is known that the minimum distance between the two strings will be larger than m. This reduces the complexity of the function $dist_{ED}()$ to compare s_i and s_j from $O(|s_i| \times |s_j|)$ to $O(min(|s_i|, |s_j|) \times m)$. All string pairs with an ED of m or less are added to the EDIndex graph $\mathbf{P}_a$ and the edit paths between them are added to $\mathbf{E}_a$ in line 14.

Once built for each attribute $a \in A$, the EDIndex is ready to be queried. Depending upon the application, query records potentially have to be inserted into $\mathbf{D}$ and the EDIndex after being matched. Such dynamic updates only require that new string values, their frequencies, and edit paths are added to or updated in the EDIndex data structures. This allows our approach to be adaptive to new string values, potentially leading to improved matching.

V. Context-aware Real-Time String Matching

The matching of query records $q \in \mathbf{Q}$ over the attributes $a \in A$ to be used in the matching is based on the built EDIndex graphs $\mathbf{P}_a$, the string inverted indexes $\mathbf{S}_a$, and the edit path inverted indexes $\mathbf{E}_a$. To efficiently calculate Equation (2), we follow an approach analogous to naive Bayes and assume the different pieces of contextual information (string frequencies, edit path frequencies, and the neighborhood size of strings) are independent from each other. We therefore calculate the propensity score of Equation (2) as:

$$p_{s_q \to s_i} = sim_{ED}(s_q, s_i) \times w_s(s_i) \times w_e(e_{q,i}) \times w_n(s_i), \quad (3)$$

where $sim_{ED}(s_q, s_i)$ corresponds to Equation (1) and $e_{q,i}$ is the most likely edit path between s_q and s_i. In practice this independence will likely not hold, as for example frequently occurring strings might have more close neighboring strings than rare strings if more typographical mistakes are being made with the common string. In future work we aim to investigate approaches that take such dependencies into account. We now describe each of the three weight components of Equation (3).

String frequencies: $w_s(s_i)$ is a weight for the likelihood of s_i occurring, and we investigate two ways of calculating $w_s(s_i)$. We denote with $N_{q,m}$ the set of strings in the EDIndex graph $\mathbf{P}_a$ that are neighbors of s_q with an ED up-to m, i.e. the strings connected to s_q (directly or indirectly) in the graph $\mathbf{P}_a$. If s_q is a new unknown string we first find its neighbors $N_{q,m}$ in $\mathbf{P}_a$, and also calculate all edit paths between s_q and its neighbors in $N_{q,m}$.

One approach to calculate $w_s(s_i)$ is to give larger weights to strings that are more common, i.e. $w_s(s_i) = f_s(s_i)/(\sum_{s_j \in N_{q,m}} f_s(s_j))$. The idea behind this approach is that it is more likely that a query string is a variation of a more frequent string in its neighborhood $N_{q,m}$ than a rare string.

The second approach follows the SoftTFIDF [8] string matching approach which assigns weights to strings based on their term-frequency (TF) and inverse document frequency (IDF) weights. The idea is that rare strings are given larger matching weights as they are more unique. Such an approach has shown to be successful in earlier frequency-based string matching work [13]. In our application TF is 1 assuming an attribute value is seen as a 'document' and a specific string only occurs once in a value. We calculate $w_s(s_i)$ based on the IDF of s_i as $w_s(s_i) = log((\sum_{s_j \in N_{q,m}} f_s(s_j))/f_s(s_i))$.

Edit path frequencies: The weight $w_e(e_{q,i})$ for the edit path $e_{q,i}$ between s_q and s_i is calculated based on the frequency of occurrence $f_e(e_{q,i})$ of $e_{q,i}$ in $\mathbf{E}_a$. For edit paths of lengths 2 and longer (assuming $m > 1$), we take the frequency of the most common path. Then, $w_e(e_{q,i}) = f_e(e_{q,i})/(\sum_{s_j \in N_{q,m}} f_e(e_{q,j}))$, i.e. the weight for $e_{q,i}$ is based on the frequencies of all edit paths between s_q and all its neighbors in $N_{q,m}$. The idea is that the more common an edit path is the more likely s_q is a variation of the corresponding neighboring string s_i.

String neighborhoods: Similar to edit path frequencies, the weight of the neighborhood $w_n(s_i)$ is calculated based on the number of neighbors $N_{i,m}$ of string s_i in the EDIndex $\mathbf{P}_a$ as $w_n(s_i) = 1/|N_{i,m}|$. The more neighbors (with a maximum ED of m) a string has, the less likely it is the actual match for the query string.

Query matching The actual matching of a query record q with records r from $\mathbf{D}$ works as follows.

(i) For each attribute $a \in A$, the attribute value $s_q = q.a$ is queried on the corresponding EDIndex graph $\mathbf{P}_a$, and all candidate strings from $\mathbf{P}_a$ that are within a maximum query ED m_q (with $m_q \geq m$) are retrieved into the set N_{q,m_q} of neighboring strings. For $m_q > m$ we recursively traverse $\mathbf{P}_a$ to retrieve all strings $s_i \in N_{q,m_q}$ with $dist_{ED}(s_q, s_i) \leq m_q$.

(ii) Using the information from $\mathbf{P}_a$ and $\mathbf{E}_a$, based on Equation (3) we calculate $p_{s_q \rightarrow s_i}$ for all $s_i \in N_{q,m_q}$. We then retrieve from $\mathbf{S}_a$ the identifiers of all records that have a certain s_i, and insert these identifiers together with their $p_{s_q \rightarrow s_i}$ values into an accumulator [3] M_q, whereby, in line with adding the similarities of several compared attributes in record linkage [6], for a certain record r its $p_{s_q \rightarrow s_i}$ values obtained from the different attributes $a \in A$ are summed.

(iii) Before being returned, M_q is sorted according to the summed $p_{s_q \rightarrow s_i}$ values with the records that have the highest values ranked at the beginning of M_q. We implement various optimizations in this accumulation phase following techniques developed for scaling-up similarity search in information retrieval [3], such as restricting the inclusion of new records into M_q according to minimum similarity criteria.

TABLE I: Number of records and strings in different attributes from the four data sets used in the experiments.

	NCVR-L	NCVR-S	CCA-L	CCA-S
Records	8,261,838	448,134	6,900,163	689,928
First names	239,743	30,058	169,382	32,330
Middle names	342,774	36,799	90,463	15,920
Last names	339,730	51,450	336,235	74,100
Cities	797	748	72,773	18,648
Zip codes	913	842	4,691	3,165

Compared to our approach which considers contextual information, traditional ED based approximate string matching as used in entity resolution only considers $sim_{ED}(s_q, s_i)$ as given in Equation (1). In the following section we experimentally evaluate how contextual information can help improve string matching for real-time entity resolution.

VI. Experiments and Discussion

We evaluated our approach using two large real data sets that contain personal information, with their characteristics shown in Table I. Both data sets contain realistic information about a large number of people, including small changes such as corrections of name errors and variations. We implemented a prototype of our approach using Python 2.7 and ran experiments on two Ubuntu servers with Intel Xeon CPUs running at 2.4 GHz and 2.0 GHz, respectively. The second server was used for the experiments on the confidential database described below. To facilitate repeatability, the programs (but no confidential data) are available from the authors.

The first data set we used is a public North Carolina Voter Registration (NCVR) data set[1], which contains the names and addresses of over 8 million voters. Each record includes a voter registration number, providing us with the true match status of record pairs. This data set contains realistic information about a large number of people, including small changes such as corrections of name errors and variations, as well as completely different values in the last name and address attributes (that correspond to genuine changes that occurred when people married or moved address). The number of true duplicate records in the full NCVR data set is below 2% of all records. To investigate a data set with a higher duplicate rate, we therefore generated a smaller sub-set where half of the individuals were represented by two or more records, and the remainder by only one record. We name the full and small data sets as NCVR-L and NCVR-S, respectively.

The second data set is a confidential consumer database (named CCA-L), which consists of two components. (i) The person component contains records for some tens of millions of consumers containing the current name, address, and other personal details such as gender. Each consumer has a unique identifier in this database. We used around 7 million records from this person component. (ii) The transaction component consists of a log file containing some millions of queries against the person component. These transaction records contain the same name and address attributes as the person component for the request made against the person component and the response returned from this database (i.e. the best matching record in the person component based on a proprietary matching

[1]Available from: `ftp://alt.ncsbe.gov/data/`

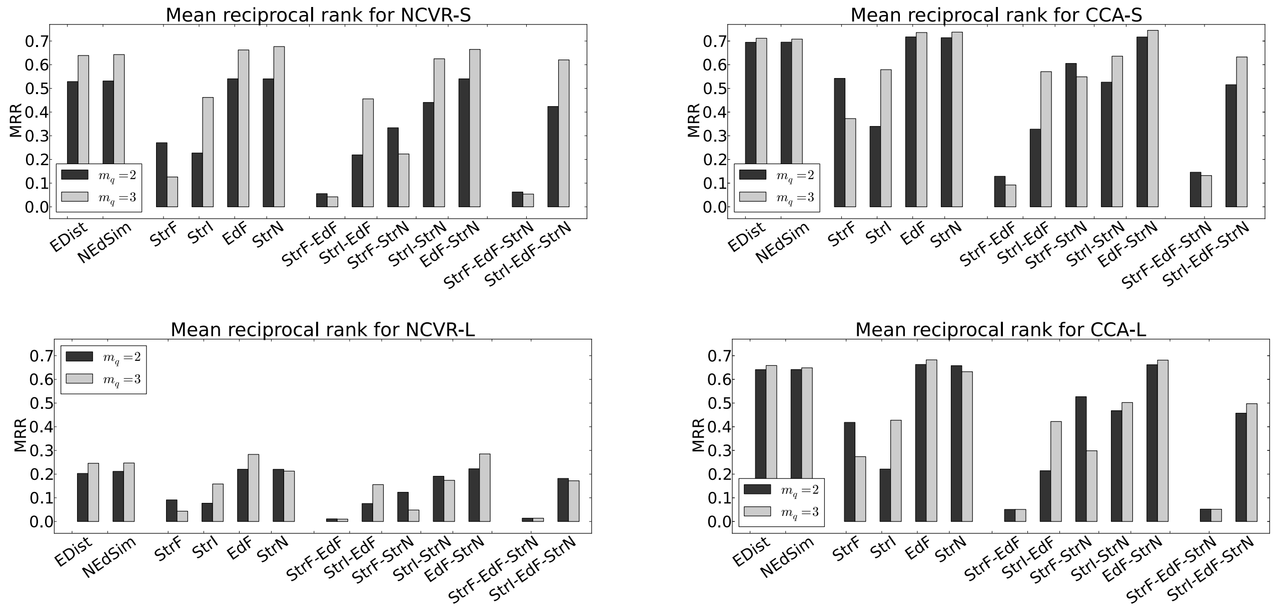


Fig. 3: Mean reciprocal rank (MRR) results for the four data sets.

approach). Many attribute values in these response records have typographical variations from the corresponding values in the requested records, as well as full attribute value changes. To investigate the scalability of our approach, we generated a smaller sub-set (named CCA-S) which is around 10% of the size of CCA-L.

To better understand the characteristics of these data sets, we analyzed the distribution of ED for pairs of records that correspond to true matches (i.e. refer to the same entity) and true non-matches (pairs referring to different entities). Figure 1 shows the resulting distributions on the two large data sets. We included exact matches (i.e. ED of 0) and grouped all pairs with ED of 4 or larger into one bin. As can be seen, the majority of string pairs with an ED of 1 or 2 correspond to true matches, while for larger ED the pairs more likely correspond to true non-matches. This indicates that building an EDIndex graph with a maximum ED of $m = 1$ or $m = 2$, respectively, will directly connect the majority of true matching string pairs. We therefore believe that only small values are required for m. Additionally, many strings with an ED larger than m are connected via another string, as for example 'pet' and 'petra' via 'pete' as can be seen in Figure 2.

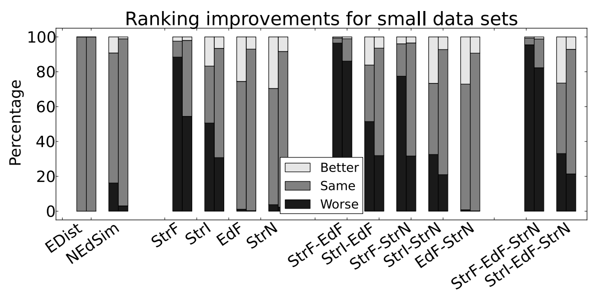
We conducted experiments by first building the EDIndex on 99% of the records in the NCVR data sets, and used the remaining 1% for querying. For the CCA data sets we built the index on the person component and used the transaction component for querying. When building the EDIndex we set $m = 2$, and for querying we used $m_q = [2, 3]$. Larger values for m_q increased query times significantly (as more candidate records were retrieved) and also led to worse matching results (as the larger candidate sets included more false matches).

As baseline techniques, we use the raw ED for ranking (named 'EDist' in the result plots), where candidate records with smaller ED values are ranked first; and the normalized edit similarity ('NEdSim') from Equation (1). Neither of these takes contextual information into account. We name the four contextual components discussed in Section V as 'StrF' (string frequency), 'StrI' (string inverse document frequency), 'EdF' (edit path frequency), and 'StrN' (string neighborhood). We investigated different combinations of these weights being included or excluded in the calculation of Equation (3), and how these affect the ranking of true matching records.

For the set $\mathbf{Q}$ of query records q that have at least one true match in the EDIndex, we calculated the *mean reciprocal rank* (MRR) [9] as $mrr = 1/|\mathbf{Q}| \sum_{q=1}^{|\mathbf{Q}|} 1/rank_q$, where $rank_q$ is the top rank of the first true matching candidate record in M_q for q. Note that the precision of all approaches will be the same because the set of candidate records returned only depends upon the maximum ED parameter m_q used to query the EDIndex. Only the ranking of candidate records will differ between the various approaches. We also report for the different approaches the percentages of queries where ranking was better, the same, or worse compared to the 'EDist' baseline, and the average overhead in query times over this baseline to investigate the cost of taking contextual information into account.

From the MRR results in Figure 3 we can see that generally the two baseline approaches perform well, and only considering edit frequencies and neighborhood size into the weighting calculations in Equation (3) can achieve improvements (for all four data sets). As the ranking improvement plot (left side in Figure 4) shows, for the NCVR-S data set edit frequencies and string neighborhood can improve the ranking in up-to 30% of all queries, while for the CCA-S data set the improvement is up-to around 10%. Surprisingly, incorporating string frequency or string inverse document frequency led to worse ranking results. We plan to conduct further experiments to investigate these unexpected outcomes.

With the larger data sets we obtained lower MRR results (as can be seen in Figure 3 especially for NCVR-L), which is because there are likely more records in a larger data set that have the same string values leading to more false matches.



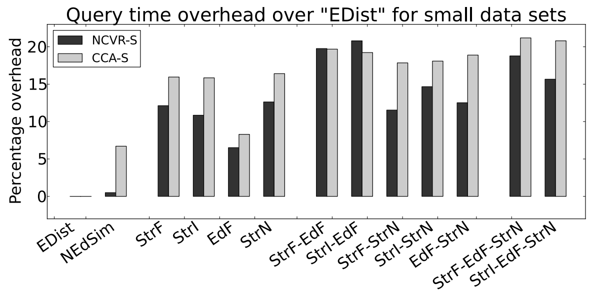


Fig. 4: Query ranking improvements (left) and query time percentage overheads (right) for the small data sets with $m_q = 3$. In the ranking improvement plot, the left bars show the results for NCVR-S and the right bars the results for CCA-S. Average times for matching a query record ranged from around 0.3 to 4 sec depending upon the data set size and the value of m_q.

With regard to timing, with our prototype we measured average query times ranging from around 300 $msec$ to around 4 sec depending upon data set size and the value of m_q. Larger values of m_q lead to more candidate records being retrieved, making querying more time consuming. However, as the right-hand side of Figure 4 shows, the overhead of taking contextual information into account is less than 20% compared to the baseline query time required.

VII. Conclusions and Future Work

We have investigated if contextual information about strings, as calculated from large databases, can improve the quality of edit distance based approximate string matching in the application of real-time entity resolution, where it is crucial that true matching records are ranked highly in the set of returned database records. We specifically investigated the frequency of string values, the frequency of character edits between two strings, and the neighborhood size of strings in a database that are similar to a given string. Our experimental evaluation on two large real-world data sets showed that taking edit frequency and string neighborhood into consideration can lead to improvements in matching quality, however both string frequency and string inverse document frequency resulted in lower matching quality.

In future work, we aim to investigate the reasons behind these somewhat unexpected outcomes and explore alternative weighting approaches to improve matching quality. We plan to learn the weights for the different contextual components from training data. We will also aim to improve query times by incorporating further query optimization techniques. Our approach so far assumes that the generated index data structures fit into main memory; to allow our approach on very large databases, investigating the suitability of disk-based index approaches is another avenue for future work.

While this work has mainly addressed the comparison step of real-time entity resolution, our approach can be integrated with existing other frameworks for real-time entity resolution [4], [7], [15], [17] to improve overall matching quality.

VIII. Acknowledgments

This research was funded by the Australian Research Council, Veda, and Funnelback Pty. Ltd., under Linkage Project LP100200079.

References

[1] Andoni, A., and Onak, K. Approximating edit distance in near-linear time. *SIAM Journal on Computing 41*, 6 (2012), 1635–1648.

[2] Bast, H., and Celikik, M. Efficient fuzzy search in large text collections. *ACM TOIS 31*, 2 (2013), 10.

[3] Bayardo, R., Ma, Y., and Srikant, R. Scaling up all pairs similarity search. In *WWW* (Banff, 2007).

[4] Bhattacharya, I., and Getoor, L. Query-time entity resolution. *Journal of Artificial Intelligence Research 30* (2007), 621–657.

[5] Bilenko, M., and Mooney, R. J. Adaptive duplicate detection using learnable string similarity measures. In *SIGKDD* (Washington, 2003).

[6] Christen, P. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer, 2012.

[7] Christen, P., Gayler, R., and Hawking, D. Similarity-aware indexing for real-time entity resolution. In *CIKM* (Hong Kong, 2009).

[8] Cohen, W., Ravikumar, P., and Fienberg, S. A comparison of string distance metrics for matching names and records. In *SIGKDD workshop on Data Cleaning, Record Linkage and Object Consolidation* (Washington, 2003).

[9] Craswell, N. Mean reciprocal rank. In *Encyclopedia of Database Systems*. Springer, 2009, pp. 1703–1703.

[10] Ferragina, P. String algorithms and data structures. *arXiv preprint arXiv:0801.2378* (2008).

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

[12] Kukich, K. Techniques for automatically correcting words in text. *ACM Computing Surveys 24*, 4 (1992), 377–439.

[13] Lange, D., and Naumann, F. Frequency-aware similarity measures: Why Arnold Schwarzenegger is always a duplicate. In *CIKM* (Glasgow, 2011).

[14] Li, C., Lu, J., and Lu, Y. Efficient merging and filtering algorithms for approximate string searches. In *IEEE ICDE* (Cancun, 2008).

[15] Liang, H., Wang, Y., Christen, P., and Gayler, R. Noise-tolerant approximate blocking for dynamic real-time entity resolution. In *PAKDD* (Tainan, 2014).

[16] Navarro, G. A guided tour to approximate string matching. *ACM Computing Surveys 33*, 1 (2001), 31–88.

[17] Ramadan, B., and Christen, P. Forest-based dynamic sorted neighborhood indexing for real-time entity resolution. In *ACM CIKM* (Shanghai, 2014), pp. 1787–1790.

[18] Sarawagi, S., and Kirpal, A. Efficient set joins on similarity predicates. In *ACM SIGMOD* (Paris, France, 2004), pp. 743–754.

[19] Sukharev, J., Zhukov, L., and Popescul, A. Parallel corpus approach for name matching in record linkage. In *IEEE ICDM* (Shenzhen, China, 2014).

[20] Xiao, C., Wang, W., and Lin, X. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB 1*, 1 (2008).