Efficient Record Linkage using a Double Embedding Scheme

Noha Adly
Department of Computer and Systems Engineering
Faculty of Engineering, Alexandria University
Alexandria, 21544 Egypt
noha.adly@alex.edu.eg

***Abstract*—Record linkage is the problem of identifying similar records across different data sources. The similarity between two records is defined based on domain-specific similarity functions over several attributes. In this paper, a novel approach is proposed that uses a two level matching based on double embedding. First, records are embedded into a metric space of dimension *K*, then they are embedded into a smaller dimension *K′*. The first matching phase operates on the *K′*-vectors, performing a quick-and-dirty comparison, pruning a large number of true negatives while ensuring a high recall. Then a more accurate matching phase is performed on the matching pairs in the *K*-dimension. Experiments have been conducted on real data sets and results revealed a gain in time performance ranging from 30% to 60% while achieving the same level of recall and accuracy as in previous single embedding schemes.**

Keywords- data cleaning; similarity matching; record linkage; embedding schemes

I. INTRODUCTION

The record linkage problem is to find similar records, across different data sources, that refers to the same real world entity, e.g. patient, customer or author. When record linkage is performed within the same source, the problem is referred to as duplicate detection. The record linkage arises in the context of data cleaning that usually precedes data analysis and mining. It is important when integrating two data sources into one and in improving the quality of data by comparing to more accurate sources. The major challenges in record linkage are reducing computational complexity while maintaining high recall and accuracy. Several techniques have been proposed in the literature, see [7, 16] for recent surveys.

A naïve approach for discovering matching records in two sources is to perform a nested-loop comparing each record in one source to all records in the second source. However, $O(N^2)$ comparisons is computationally infeasible, especially for large datasets. Further, performing an approximate matching between two records requires the computation of distance functions among textual attributes, which is an expensive factor in the cost.

Several techniques have been introduced to reduce the quadratic number of comparisons that are based on a two phase approach. Blocking [1, 2] use record attributes, or subsets of attributes as a blocking key (e.g. first 4 characters of the surname) to split the data into blocks, then a detailed comparison is carried out only between records that fall into the same block. Although blocking increase the speed of comparison, it can lead to an increased number of false negatives due to selection of a blocking step that places entries in the wrong blocks. Multiple runs using different blocking fields are performed to improve effectiveness. Sorted neighborhood [10], or merge/purge approach, relies on sorting records based on a sorting key, then moving a window of fixed size *w* sequentially where only records within *w* are paired with each other. This approach relies on the assumption that duplicate records will be close in the sorted list. Also, its effectiveness is dependent upon the comparison key. Similar to blocking, it has shown to be more effective with multi-passes, which causes an increase in the runtime. McCallum et al [18] proposed a cheap comparison metric to group records into overlapping clusters called canopies, then records within the same cluster are accurately compared. A recent work [21] exploited the semantic ambiguity of the data sources, using social network analysis, and applied relaxed matchers to less ambiguous data.

Reducing the complexity of record comparisons has been addressed by techniques such as feature subset selection algorithms [23]. Recent approaches [15, 20] have used embedding textual attributes into Euclidean space while preserving the distances between the record values and performed the comparison in the metric space which is much cheaper than string comparison. In [20] SparseMap was used to map strings in a private environment to Euclidean space then used multidimensional index based on KDTree to perform the comparison. Jin et al [15] used FastMap for converting strings then applied a similarity-join algorithm based on R-tree to compare the metric vectors.

This paper proposes a novel two-level matching scheme that exploits the advances developed in mapping records into a multidimensional Euclidean space while preserving the distances between the record values. It relies on a quick-and-dirty matching process that is employed first, to prune a large number of mismatches and produces a smaller set of pairs of records that are used as input to a more expensive matching process. Datasets are first embedded into a metric space of dimension *K*, which captures an accurate representation of the data. It is followed by a second embedding that converts vectors from *K* dimension to vectors of smaller dimension *K′*. The quick-and-dirty matching phase is performed on the output of the second embedding, with the goal of discarding a large number of true negatives while ensuring that the resulting potential matching pairs includes all true positives and zero false negatives, yet achieving this with a small cost. The returned results are ingested to the second level of matching, which applies a similarity function on the potential pairs, but in their *K*-dimension representation. The second

level matching is more expensive, but more accurate, and its goal is to refine the results of the first level by excluding false positives and any true negatives that have not been pruned in the first phase. With $K'<K$, the first level matching process is much cheaper than having the similarity function applied on all pairs in their K-representation. The expensive matching is performed only on the pairs detected in the first phase, which are much less, and cause an overall gain in time performance while achieving a high recall and accuracy.

Recently, a similar approach has been proposed in [27] in the context of private record linkage. They assumed the data already embedded in the metric space, which is then represented as a point in the complex plane where a relaxed matching is performed, detecting pairs likely matching. Then a more accurate matching phase is performed on the likely matching pairs only. The second embedding though is bound to the complex plane where $K'=2$, and is not contractive. In this paper, the second embedding is generalized and a proper K' is chosen according to the nature of the data and its size. It has been shown that as the data size increases, higher values of K' results in higher performance gains.

By selecting a proper embedding scheme in each phase, preserving the distances between the record values while guaranteeing contractiveness, it is ensured that results obtained are with a high recall and accuracy. The proposed matching scheme based on double embedding has been implemented and a set of experiments have been conducted on real data sets and compared with a scheme using a single embedding. Results showed that improvement in time performance ranging from 30% to 60% is achieved while maintaining the same level of recall and accuracy.

The remainder of the paper is organized as follows. A formal definition of the problem is presented in Section II. In Section III, we describe a single embedding matching scheme that is similar to previous schemes proposed and will be used for comparison. In section IV, the steps of the matching scheme based on double embedding are presented. In section V, the experiments conducted to evaluate the performance of the protocol are presented and results are discussed. Finally, Section VI concludes the paper.

II. Problem Formulation

The process of identifying similar record pairs consists of building a classifier that takes as input a set of thresholds and accurately classifies pairs of records as *match* or *mismatch* according to a predefined matching rule. Without loss of generality, it is assumed that the input datasets R and S are represented as relations, and the schema of the two relations is the same $R(a_1, a_2,..a_n)$ and $S(a_1, a_2,..a_n)$.

Given a distance function d_i: $Dom(R.a_i) \times Dom(S.a_i) \rightarrow \mathbb{R}^+$ defined over domains of corresponding attribute of R and S, and matching thresholds $\theta_i \geq 0$, record linkage can be expressed as a join operator over R and S. A record pair (r,s) where $r \in R$ and $s \in S$, is a matching pair if $d_i(r.a_i, s.a_i) \leq \theta_i$ for all attributes $1 \leq i < n$. Then the join condition can be defined based on the following matching rule that returns *true* for matching record pairs and *false* for mismatching record pairs

$$MR(r,s) = \begin{cases} true & iff\ d_i(r.a_i, s.a_i) \leq \theta_i \quad \forall\ 1 \leq i \leq n \\ false & otherwise \end{cases}$$

The presented definition for the matching function is used by most of the record linkage approaches. The distance function d_i defines the similarity metric at the attribute level and is domain specific. In the domain of strings, there are a variety of metrics including the Edit distance, Smith-Waterman distance, Jaro distance, q-gram and others (refer to [5, 7] for a survey). In this work, the Edit distance a.k.a. Levenshtein distance, a common measure of textual similarity, is used although any other metric could be used. Formally, given two strings s_1 and s_2, their edit distance is the minimum number of insertion, deletions and replace operations of single characters that are needed to transform s_1 to s_2. For instance, the edit distance between Johnson and Jonsan is 2, as Johnson is obtained by adding h and replacing a by o. In the metric domain, the most common metric distance function used is the Minkowski metrics based on the L_p norms, $\|x\|_p = (\sum |x_i|^p)^{1/p}$, with $p \geq 1$. In this work, the Euclidean distance d_E (p=2), is used as the distance metric in the embedded space, although any other metric can be used.

III. Single Embedding Scheme

In this section we describe the Single Embedding Scheme, which consists of two steps. In the first step, strings are mapped to objects in a multidimensional Euclidean space, such that the mapped space preserves the original string distance. In the second step, a multidimensional similarity join is performed in the Euclidean space. This approach is similar to previous work such as [15, 20] and will be used for comparison. Several methods have been proposed to embed a set of objects in a metric space, including FastMap[8], SparseMap[13], MetricMap[25] and others (see [12] for a survey). Among those methods, SparseMap has been chosen because it has proven to be contractive when the original space is strings [12]. Contractiveness ensures that distances in the embedded space are a lower bound for distances in the original space, thus improving the quality of the embedding in terms of recall. In the following, the concept of the SparseMap technique is introduced and a description of how it is used to embed strings into Euclidean space follows. Next, the technique used to perform the similarity join to complete the matching process is described.

A. Embedding Strings to K-dimension Euclidean Space

SparseMap is an embedding method based on a class of embedding known as Lipschitz embedding [3]. Therefore, we first describe Lipschitz embedding followed by the heuristics introduced by SparseMap.

Lipschitz embedding defines a coordinate space where each axis corresponds to a reference set, drawn from the set of objects to be embedded. Given a set of objects O and a distance D in the original space, the embedding is defined in terms of a set S of subsets of O, $S=\{S_1, S_2,..S_k\}$ where S_i is a reference set. Given an object $o \in O$, the mapping F is defined as $F(o)=(D(o,S_1),\ldots,D(o,S_k))$, where $D(o,S_i)=\min_{x \in S_i}\{D(o,x)\}$. That is, the coordinate values of object o are the distances from o to the closest element in each set S_i. The method is based on the triangle inequality and exploits the fact that if $|d(o_1,x)-d(o_2,x)| \leq d(o_1,o_2)$, then the property can be extended

to subset S_i and the value $|d(o_1,S_i)-d(o_2,S_i)|$ is a lower bound on $d(o_1,o_2)$. By using a set S of subsets, we increase the likelihood that the distance $D(o_1,o_2)$ is captured adequately by the distance in the embedding space between $F(o_1)$ and $F(o_2)$ i.e. $d(F(o_1),F(o_2))$.

Linial et al [17] have shown that when d, the metric distance function used to compare the embedded object, is one of the Minkowski metrics L_p, a bound can be established on $d(F(o_1),F(o_2))$, provided that $k=\lfloor \log_2 N \rfloor^2$ and S_i is of size 2^j with $j=\lfloor (i-1)/\log_2 N+1 \rfloor$. Given $F(o)=(D(o,S_1)/q,....,D(o,S_k)/q)$, where $q=k^{1/p}$, it has been proved [17] that the embedding is *contractive* and the *distortion*, that is, the relative amount of deviation of the distance values in the embedding space with respect to the original distance, is guaranteed to be $O(\log N)$.

Lipschitz embedding is rather impractical for two reasons. First, due to the number and sizes of the subsets in S, $O(N^2)$ distance computations is needed to embed an object o, as the distance between o and practically all objects need to be computed, which is exactly what we wish to avoid. Second, the number $=\lfloor \log_2 N \rfloor^2$ of subsets, which is the number of coordinate values (dimensions) in the embedding is rather large. SparseMap [13] introduces heuristics to overcome the above limitations. The *Distance Approximation* heuristic approximates the distance between object o and subset S_i by computing $\hat{Y}(o, S_i)$, an upper bound on $d(o,S_i)$ by exploiting the partial vector that has been computed for each object.

The algorithm for embedding strings into an Euclidean space of dimension K begins by combining the strings from the two datasets into one set O. It starts by building the reference sets $S=\{S_1, S_2,..S_k\}$. As suggested in [13], each set S_i is composed of any random strings of O of size 2^j where $j=\lfloor (i-1)/\lfloor \sqrt{K} \rfloor+1 \rfloor$. Thus, we get $\sqrt{K}$ reference sets of size 2, $\sqrt{K}$ of size 4, etc, up to size $2^{\lfloor \sqrt{K} \rfloor+1}$. Then it proceeds by computing the first coordinate for all objects, followed by the second coordinates, etc. The *EmbedString* algorithm is depicted in Figure 1.

```
Algorithm: EmbedString(O, K)
Input: O: a set of N strings
        K: dimensionality of Euclidean space
Output: SE[1,N][1,K] coordinates of the N strings

// Build K reference sets S={S1, S2,..Sk}
for i=1 to K
  Si ← 2^⌊(i-1)/⌊√K⌋+1⌋ strings randomly chosen from O
for i=1 to K
   ∀ oj ∈ O
     if (i==1)
     // 1st coordinate: the distance is the minimum
     // Edit distance between oj and every object in S1
        Ŷ(oj, S1) = min_{xεS1}{D(oj ,x)}
    else {
    // Get Euclidean distance between the (i-1) coordinates
    // of oj and all objects in Si
       Compute dE(Fi-1(oj), Fi-1(ot))  ∀ot ∈ Si
       Sort dE(Fi-1(oj), Fi-1(ot)) ∀ ot ∈ Si ascendingly
       Select the first σ objects and place them in set φ
       Compute D(oj,ot) for all ot ∈ φ
       Select or s.t. D(oj,or) = min_{ot ε φ} {D(oj,ot)}
       Ŷ(oj, Si) = D(oj,or)
    }
  SE[j,i]  = Ŷ(oj, Si)
```

Figure 1: Pseudo code of *EmbedString* embedding strings to Euclidean space

A drawback of the distance approximation heuristic is that it renders the mapping non-contractive. In this paper, the heuristic proposed by [12] is used in order to make SparseMap contractive. The heuristic suggests that, instead of computing the actual distance $D(o_j,o_t)$ for only a fixed number of objects σ, it does so for a variable number of objects in S_i. In particular, it first computes the approximate distances $d_E(F_{i-1}(o_j),F_{i-1}(o_t))$ for all $o_t \in S_i$ which are lower bounds on the actual distance value $D(o_j,o_t)$. Next, it computes the actual distance value of the object $o_t \in S_i$ in increasing order of their lower bound distances $d_E(F_{i-1}(o_j), F_{i-1}(o_t))$. Let $o_r \in S_i$ be the object whose actual distance value $D(o_j,o_r)$ is the smallest distance value so far. Once $D(o_j,o_r)$ is smaller than all distances $d_E(F_{i-1}(o_j),F_{i-1}(o_t))$ of all remaining elements in S_i, then $d(o_j,S_i) = D(o_j,o_r)$. Although this heuristic increases the number of distance computations, it was decided to adopt it in order to make the embedding contractive.

B. Similarity Join in Euclidean Space

After the two datasets have been mapped into the metric space, it is required to find pairs of objects whose distance in the Euclidean space is within a threshold δ. Many similarity-join algorithms can be applied [11, 14] and usually they employ a form of multidimensional index [9]. In this work, the KDTree index[9] has been used as it is considered one of the most prominent data structure for indexing multidimensional spaces and is designed for efficient nearest neighbor search [22].

KDTree is a binary tree in which every node is a k-dimensional point. Every internal node generates a splitting hyperplane that divides the space into subspaces. Points left to the hyperplane represent the left subtree of that node and the points right to the hyperplane represents the right subtree. The hyperplanes are iso-oriented and their direction alternates among the k possibilities. Building a KDTree is a $O(NlogN)$ operation.

The nearest neighbor algorithm (NN) aims to find the node in the tree which is nearest to a given input vector. This search can be done efficiently ($O(logN)$ by using the tree properties to quickly eliminate large portions of the search space. The search starts with the root node and moves down the tree recursively until it reaches a leaf and saves that node point as the *nearest*. The algorithm unwinds the recursion of the tree; if the current node is closer than the *nearest*, then it becomes the *nearest*. The algorithm checks whether there could be any points on the other side of the splitting plane that are closer to the search point than the *nearest*. This is done by intersecting the splitting hyperplane with a hypersphere around the search node that has a radius equal to the current nearest distance. If the sphere crosses the plane, there could be nearer points on the other side of the plane, so the algorithm must move down the other branch of the tree from the current node looking for closer points, following the same recursive process as the entire search. If the hypersphere does not intersect the splitting plane, then the algorithm continues walking up the tree, and the entire branch on the other side of that node is eliminated.

A variation of the NN algorithm [19] has been adopted that allows performing range searching. That is, given δ and a vector v, it is required to retrieve all nodes that are within distance δ from v. The variation is mainly that the initial distance is not reduced as closer points are discovered and all discovered points within δ are returned, not just the *nearest*.

The variation of the NN algorithm is applied in order to compare the vectors representing the two datasets SE_1 and SE_2; the used distance metric is the Euclidean distance. Specifically, the KDTree for one of the datasets is built, say SE_1. Then, for every vector in SE_2 the NN range search algorithm is applied to retrieve the nodes in SE_1 that are within distance δ.

IV. DOUBLE EMBEDDING SCHEME

This section introduces the Double Embedding Scheme, which consists of four steps. The first step, combines the strings from the two datasets into one set O and maps them into Euclidean space of dimension K using the *EmbedString* algorithm presented in Section III.A. The second step involves mapping the embedded strings into a more compressed representation in the Euclidean space in dimension K', where $K'<K$. In this step, the FastMap embedding technique has been selected, because of its simplicity, efficiency and contractiveness. The third step performs the similarity join between the two sets in the K' dimensional space using the KDTree as described in Section III.B. The fourth and last step, takes its input as the potential matched pairs produced from the similarity join and compares the Euclidean distance of the corresponding objects in the K dimension space.

In the following, the technique used in embedding the datasets in K' dimension using FastMap is described, then the overall matching protocol is presented.

A. *Embedding K-dimension objects into K'-dimension*

FastMap[8] is a general embedding technique that is inspired by dimensionality reduction methods for Euclidean space based on linear transformation. Objects are mapped into points in K' dimensional space, where the coordinate values corresponding to these points are obtained by projecting them on K' mutually orthogonal directions, thereby forming the coordinate axes of the space in which the points are embedded. The projections are computed using the original distance function D. In our case, where the original space is K-dimension Euclidean space $D= d_E^K$. The coordinate axes are constructed one by one, where at each iteration, two objects (referred to as pivot objects) are chosen, a line is drawn between them that serves as the coordinate axis, and the coordinate value along this axis for each object o is determined by projecting o into this line.

For setting the K'-coordinate axis, pivot objects are chosen at each step to anchor the line that form the newly formed axis. To extract more distance information, FastMap attempts to identify a pair of pivot objects that are far away from each other. In order to avoid $O(N^2)$ distance computations to determine the farthest pair of objects, a heuristic is proposed in [8] for computing an approximation of the farthest pair of objects. This heuristic first arbitrary chooses one of the objects t. Next, it finds the object r which is farthest from t. Finally, it finds the object s which is farthest from r. The last step can be iterated a number of m times in order to obtain a better estimate. Although [8] indicated that setting m=5 provides good estimates, [13] has shown that negligible improvements are achieved for m>2.

Deriving the first coordinate for the N objects is obtained by projecting each object a on a line between pivots p_1 and p_2 $x_a=[D(p_1,a)^2+D(p_1,p_2)^2 - D(p_2,a)^2]/2D(p_1,p_2)$. To derive the ith coordinate, the (i-1)-dimensional hyperplane H, which is perpendicular to the line that forms the previous coordinate axis, is determined and all objects are projected onto H. The projection is performed by defining a new distance d_H that measures the distance between the projections of the objects on H. Let x_0^i be the ith coordinate for object o, $F_i(o) = \{x_0^1, x_0^2,\ldots,x_0^i\}$ be the first i coordinate value for $F(o)$, d_i be the distance function used in the ith iteration, and p_1^i and p_2^i be the two pivots chosen at iteration i, then

$$x_0^i = [d_i(p_1^i,o)^2+ d_i(p_1^i, p_2^i)^2 - d_i(p_2^i,o)^2]/2\, d_i(p_1^i, p_2^i)$$

The algorithm of embedding objects from the metric space of dimension K into dimension K' using FastMap is described in Figure 2.

```
Algorithm EmbedNum(SE, K')
Input: SE[1,N][1,K]:  coordinates of the N strings
       K': dimensionality of Euclidean space
Output: DE[1,N][1,K'] coordinates of the N strings in K' dimension

for (h=1 to K') {
 (p1,p2) = ChoosePivot(h);
 v = FastAproxDist(p1, p2, h);
 if (v==0)      // all inter-objects distances are zero
   DE[i][h] =0  ∀ i=1 to N
 else           // compute coordinate on this axis h
   for (i=1 to N) {
     x = FastAproxDist (i, p1, h);
     y = FastAproxDist (i, p2, h);
     DE[i][h]= (x^2 + v^2 – y^2)/ 2*v;
   }
}
```

Figure 2(a): Algorithm ***EmbedNum*** mapping numbers into K'

```
FastAproxDist (a, b, h) {
  v = d_E^K (SE[a],SE[b]);
  for (i=1 to h-1) {
    w = DE[a][i] – DE[b][i];
    v = |v^2-w^2|^1/2;
  }
  return v;
}
ChoosePivot(h) {
  // choose two pivots from objects represented in SE  on the h^th -dimension
  select an object and set it to be the second pivot b
  for (i=1 to m) {
   // FastAproxDist() is used to get distance between a and b
     Set a = farthest object from b;  Let p_a be index of a in SE
     Set b= farthest object from a;   Let p_b be index of b in SE
  }
  return(p_a , p_b);
}
```

Figure 2(b): Methods used with ***EmbedNum*** algorithm

In the method *FastApproxDist*(), since the distance v can be negative, the heuristic developed by [26] has been adopted, namely using the square root of the absolute value of (v^2-w^2). FastMap has the advantage of being simple and efficient as its cost is linear; it needs $O(2+2m)K'N$ distance

computations. It should be noticed that FastMap has not been chosen as the embedding technique in the first step because it has been proven in [12] that FastMap is not contractive when the original object space is not the Euclidean space.

B. *Overall Matching Scheme*

In this section, the pseudo-code of the overall matching scheme is illustrated. It starts by combining the strings from the two source datasets into one set O and maps them into Euclidean space of dimension K using the *EmbedString* algorithm, generating a global matrix $SE[N_1+N_2][K]$ containing the K-coordinates of all strings. SE is then split to $SE_1[N_1][K]$ and $SE_2[N_2][K]$ representing each dataset to be matched. This is followed by embedding the K-dimensional vector of each record in a more compressed representation in K'-dimension generating $DE_1[N_1][K']$ and $DE_2[N_2][K']$ for each source. Next, objects in their K'-representation are compared and returns the set P' including those pairs whose Euclidean distance is within a threshold δ', using the similarity join algorithm described in Section III.B. The set of matching pairs P' has been pruned such that most true negatives have been discarded. Further, it is ensured that all true positives are included with no false negatives, since the embedding used is contractive. Finally, all pairs in P' are compared in K dimension, and those pairs whose Euclidean distance is within threshold δ are extracted. The scheme is presented in Figure 3. It is worth mentioning that one advantage of the presented scheme is that it is open to many embedding schemes, as long as they are contractive, and any multidimensional similarity join algorithms. Also, it does not depend on specific similarity functions, whether in the string domain or the Euclidean space.

The values of the embedding parameters K and K' and the similarity thresholds δ and δ' affects the performance, which will be explored by experimental evaluation in Section V. A heuristic for selecting the parameters based on samples from the datasets is proposed and its application is validated against the experimental results and showed to be very effective. The heuristic and its application will be presented in an extended version of this paper.

Algorithm: ***DoubleEmbedding***(Set_1, Set_2, K, δ, K', δ')
Input: Set_1, Set_2: two datasets of Strings of size N_1 and N_2
K, K': dimension of first and second embedding
δ, δ' : threshold of first and second embedding
Output: P: set of matching pairs

Combine Set_1 and Set_2 into one set O
Embed O using *EmbedString*(O, K) and get $SE_1[N_1][K]$, $SE_2[N_2][K]$
Embed SE using *EmbedNum*(SE,K') and get $DE_1[N_1][K']$ and $DE_2[N_2][K']$
// perform similarity join between DE_1 and DE_2
Build the KDtree T for DE_1
Set $P' = \{\}$;
for i=1 to N_2
  $P' = P' \cup$ NNSearch($DE_2[i]$, T, δ')
// compare the pairs in P' in K dimension
Set $P = \{\}$;
$\forall$ pairs $(p_{i,j}) \in P'$
  if $d_E^K(SE_1[i], SE_2[j]) \leq \delta$
    $P = P \cup (p_{i,j})$

Figure 3: Pseudo code of ***DoubleEmbedding*** Algorithm

V. EXPERIMENTS

In order to evaluate the potential benefits of the proposed solution, a set of experiments has been conducted on real datasets with different sizes, with the following goals:

- Tune the parameters of the Single Embedding Scheme (SES) while analyzing the quality of the embedding and evaluating the effectiveness of the resulting matching scheme. Several parameters, namely K and δ, are varied experimentally and the distortion of the embedding and the effectiveness of the matching are measured. The goal is to reach a reasonable selection of K and δ and validate them against previous published results.
- Tune the parameters of the Double Embedding Scheme (DES) while analyzing the efficiency and effectiveness of the matching protocol in comparison with the SES.
- Analyzing the efficiency of the DES while varying the size of the datasets, when compared to the SES. Also, the time performance is compared to record matching performed in the original string space.

In the experiments, a real dataset has been used, representing British Columbia voters' list containing 34,264 records of voters' names and addresses. This data is available at http://www.rootsweb.com/~canbc/vote1898. Only the first name and last name fields were used in the experiments. Removing all duplicates from the original set resulted in 29,299 distinct records. From such dataset, two datasets are generated where we controlled and identified the percentage of similar records between each set pair. Three different sizes of datasets pairs were generated, namely with each set containing 4,000, 10,000 and 20,000 records respectively in order to evaluate the scalability of the proposed solution. Throughout the experiments the threshold θ in the string space was set to 2.

Efficiency is measured by the total execution time needed to perform the embedding, indexing and matching. Effectiveness of the scheme is analyzed in terms of:

- Recall: the ratio of the number of matched records pairs generated by the matching protocol to the total number of true matched record pairs. This metric has been sometimes referred by others [4, 6, 15, 24] as pairs completeness.
- Accuracy: the percentage of the correctly classified pairs [6, 24]. It is defined as the number of pairs correctly classified as matches or non-matches to the total number of pairs. This metric has been sometimes referred by others [1, 18, 20] as precision.

The platform used for these experiments was a PC with an Intel dual-Core Duo processor 2.2 GHz and 3GB of memory. The protocol was implemented using Java and tested under Windows XP. In the implementation, the SecondString library [5] has been used for similarity matching and Levenshtein distance has been used. The library is available at http://secondstring.sourceforge.net/. For indexing the embedded space, the KDTree implementation available at http://www.cs.wlu.edu/~levy/kd has been used. The Java source was modified in order to implement the nearest neighbor using range search as described in Section III.B.

A. *Selection of Parameters for First Embedding*

The selection of a small dimension K would result in a miss representation of the data, hence distance would not be preserved and similar pairs and dissimilar pairs will not be distinguished. However, setting K to a high value results in high cost, both for embedding and matching, and we risk the curse of dimensionality. For the selection of K, samples of 4000 records from the datasets have been embedded in different dimensions and the quality of the embedding is evaluated with respect to the *stress* [12], measuring the distortion of the embedding defined as

$$stress = \frac{\sum_{o_1,o_2}(\delta(F(o_1),F(o_2)) - d(o_1,o_2))^2}{\sum_{o_1,o_2} d(o_1,o_2)^2}$$

Also, the recall and accuracy were recorded. Results when varying K for three dataset sizes (4K, 10K and 20K) are shown in Figure 4. As expected, increasing K results in lower stress values and higher recall and accuracy. Results revealed that with K set to 25 and higher, very small variation in the stress is obtained and very small improvement in the recall and accuracy are reached. Therefore, in the remaining of the experiments, K is set to 25. These results are similar to the results obtained by [13] and [20].

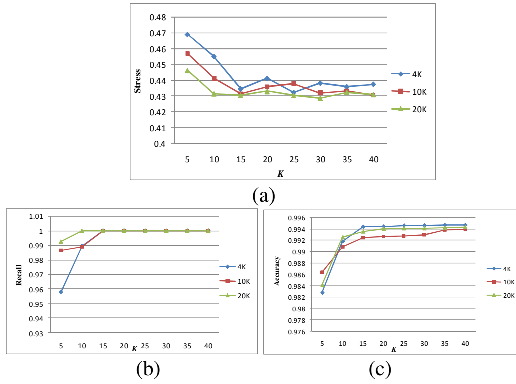
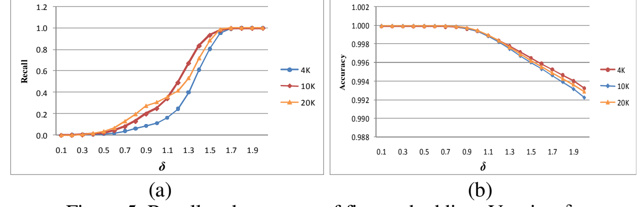


Figure 4: Stress, recall and accuracy of first embedding, varying K

Another important parameter that affects the performance and the effectiveness of the matching protocol is the threshold δ. We ran a set of experiments on the full datasets 4K, 10K and 20K while varying δ from 0.1 to 2. It should be noticed that there is no need to set δ higher than 2 since the mapping used is contractive. Since we knew which records were the true matches, therefore we could compute the recall and accuracy. Results are shown in Figure 5.

As expected, increasing δ results in improving the recall (Fig 5(a)) as large value of δ ensures that all true matches are included in the results returned from the matching protocol, at the cost of an increase in the matching time. The recall reaches good values close to 1 for δ larger than 1.6. At δ=1.8 the recall reaches 100% for all the three data sets. However, increasing δ results in a decrease in the accuracy as larger values of δ results in more false positives returned. However, it is noticed from Figure 5(b) that the decrease in accuracy is very small, ranging from 0.5% to 0.6% at δ=1.8. On the basis of these experiments, the chosen embedding parameters were for K to be set to 25 and for δ to be set at 1.8.



(a) (b)

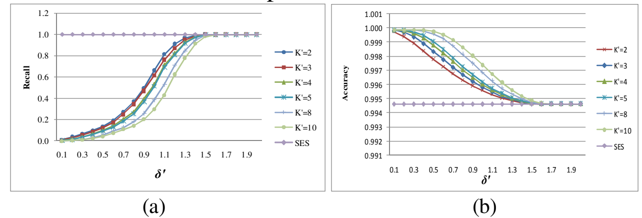
Figure 5: Recall and accuracy of first embedding, Varying δ

B. *Selection of Parameters of Second Embedding*

In this section, the effectiveness of the proposed Double Embedding Scheme (DES) is evaluated, measured in terms of the recall and accuracy as well as its efficiency measured in terms of the total execution time. The two parameters affecting the performance of the double embedding are K' and δ'. A set of experiments has been conducted varying K' from 2 to 10 and varying δ' from 0.1 to 2 for each K'. Again, δ' does not need to be larger than 2 since the FastMap algorithm used in the second mapping is contractive. The experiments are repeated for the three datasets in order to demonstrate the scalability of the protocol and to observe the effect of variation of the parameters K' and δ' when the data size increases.

Figure 6 shows the recall and accuracy for the 4K dataset, varying δ' from 0.1 to 2 for different values of K'. The results of the Single Embedding Scheme (SES) are also shown for comparison, with K=25 and δ=1.8.

As expected, increasing δ' results in an increase in the recall as larger values of δ' increases the number of true positives returned from the matching protocol. High recall values are reached for δ' >1.4 for all K'. The primary reason is that the embedding used is contractive and provides a good distance/similarity preservation. The accuracy on the other hand decreases as δ' increases, since more false positives are returned. However, it reaches the accuracy of SES for δ'>1.4. The same results were obtained for the 10K and 20K datasets, not shown for space constraint.



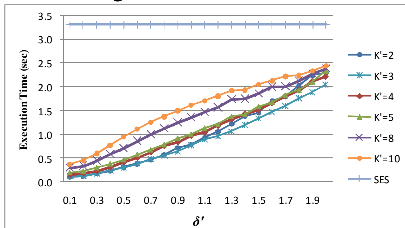
(a) (b)

Figure 6: effectiveness of DES for 4K dataset

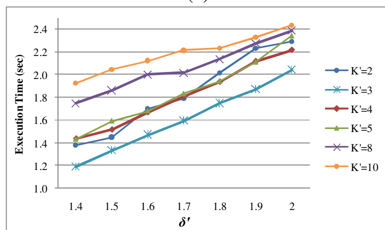
Figure 7(a) shows the execution time of DES while varying δ' and K'. It is observed that the cost increases as δ' increases. This is expected as while matching in the KDTree, less pruning is done as there are more nodes to be retrieved for larger δ'. Also, this increase is due to the increase in the number of potential matching pairs generated from the first level matching. However, for all δ' and K', the cost of the DES is substantially lower than SES, resulting in a minimum of 30% improvement for all K'. The effect of the variation of K' on the cost is somehow complex since it consists of three components. The first component is the embedding time, which increases as K' increases. The second component is the cost of indexing, that is building the KDtree for one of

the embedded sets, then applying the nearest neighbor algorithm using range search for the second set. This cost also increases with the increase of K'. The third component, is the final stage of matching, which consists of computing the Euclidean distance between the set of matching pairs resulting from indexing and searching the KDtree. This cost is dependent on the number of matching pairs returned, which decreases as K' increases. This decrease is attributed to a more accurate representation of the embedded records, hence more accurate matching pairs are obtained.

Figure 7(b) shows the total execution time for δ' larger than 1.4. It is plotted separately in order to show the effect of varying K' more closely. δ' is chosen from 1.4 to 2.0 since this is the range δ' will be chosen from to achieve a good recall. It is observed that the cost of DES gives improvement for $\delta'>1.4$ ranging from 30% to 64% than SES for all values of K'. The lowest cost is achieved with $K'=3$, which achieves the best balance in the cost of the indexing versus the number of potential matching pairs. For $K'=2$, the cost is higher than $K'=3$ because the number of detected pairs in the first matching phase is much higher than for $K'=3$ (2.2m pairs versus 1.7m pairs for $\delta'=1.5$). Hence, the pair matching cost is higher, which increases for larger values of δ'. For $K'=$ 4 and 5, the cost is quite similar. It is higher than that of $K'=3$ because the increase in the indexing and matching cost (KDTree) is higher than the decrease in the matching pair cost. They are higher than $K'=2$ for δ' smaller than 1.5, then they outperform $K'=2$ since the increase in their indexing and matching cost is lower than its increase in the matching pair cost. When K' is set to 8 and 10 the cost gets higher as the indexing and matching cost increases.



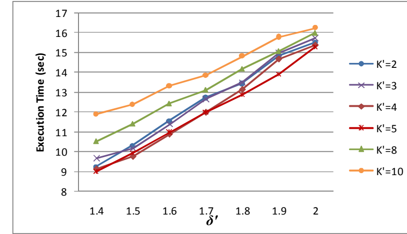
(a)



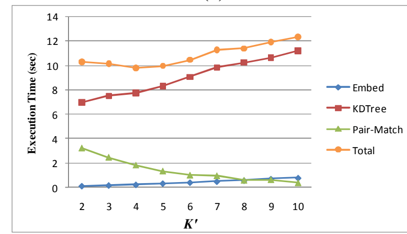
(b)

Figure 7: Cost of DES for 4K dataset

Running the same experiments on the 10K dataset revealed that although the improvement in the cost decreases as δ' increases, it ranges from 30% to 60% for all K' for $\delta'>1.4$. Figure 8(a) shows the cost for the 10K dataset for $\delta'>1.4$ and Figure 8(b) shows the breakdown of the total cost for $\delta'=1.5$ for all K'. It is noticed that for $\delta'=1.5$, the total cost reaches its minimum at $K'=4$, where the increase in the embedding time and the KDTree is lower than the decrease in the pair matching time. From Figure 8(a), it is observed that $K'=4$ and 5 yield the best balance between the three components. The cost of $K'=2$ and 3 is higher because the matching pair time is dominant. The cost of $K'=8$ and 10 are higher than all where the indexing and matching (KDTree) increase is more dominant.



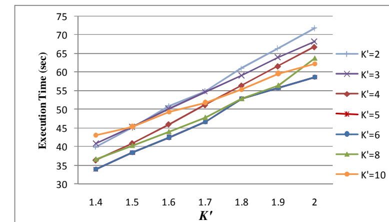
(a)



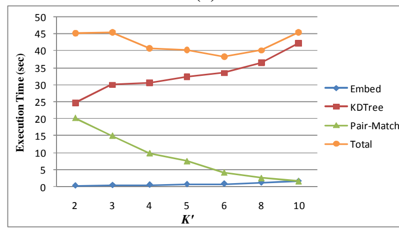
(b)

Figure 8: cost of DES for 10K dataset

Figure 9(a) shows the cost for the 20K dataset for $\delta'>1.4$. Figure 9(b) shows that for $\delta'=1.5$, the total cost reaches its minimum at $K'=6$, where the increase in the embedding time and the KDTree is lower than the decrease in the pair matching time. From Figure 9(a), it is observed that $K'=6$ yields the best balance between the three cost components, followed by $K'=5$, then $K'=8$. The cost of $K'=2$ and 3 experience the highest cost, especially for δ' larger than 1.4 where the matching pair time is dominant. The cost of $K'=10$ is high, specially for small value of δ', where the indexing and matching (KDTree) increase is more dominant. As δ' increases, K' set to 10 shows lower cost than K' set to 4 or smaller, as the increase in the matching pair time is minimal compared to smaller K'.



(a)



(b)

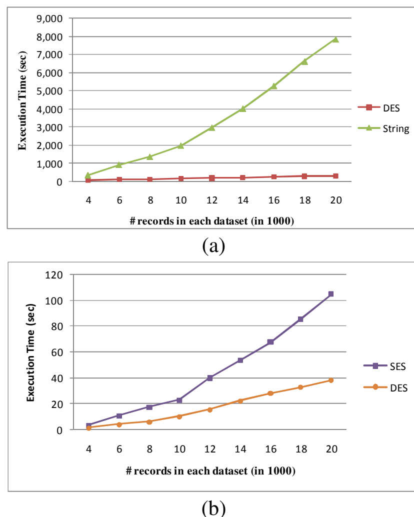
Figure 9: Cost of DES for 20K dataset

The above experiments show that the selection of K' affects the improvement of the cost, and is dependent on the

size of the dataset. As the size of the dataset increases, larger values of K' yields lower cost. However, it is shown that for δ' set to 1.5, the worst selection of K' would result in a 40% improvement, while an optimum selection can lead to improvement ranging from 50% to 60% over SES.

C. *Effect of Datasize Variation*

To evaluate the scalability of DES, its run time is compared with the run time of matching records in the original space, varying the datasets from 4K to 20K, shown in Figure 10(a). It is obvious that matching strings requires by far more time due to $O(N^2)$ string distance computations and the difference is more dramatic as the data size increases. The scalability of the protocol is studied also in comparison with SES as shown in Figure 10(b). The parameters used for SES were K=25, δ=1.8, and for DES δ' =1.5 and K'=3 for N=4K, K'=4 for N<10K, K'=5 for N<16K and K'=6 for N<20K. The results show that DES outperforms SES, especially for large datasets, showing improvement ranging from 59% to 64%.



(a)

(b)

Figure 10: Cost of DES vs string matching and SES

VI. CONCLUSIONS

This paper introduced a novel scheme for record linkage based on double embedding of the data, aiming at improving the efficiency. A two level matching is proposed, with the first level performing a fast and inaccurate matching, ensuring high recall while the second level performs a more expensive matching, on a smaller set of pairs, to improve the accuracy. Experimental evaluation on real datasets revealed that, by using contractive embedding techniques that preserve the distance between records values, the suggested scheme outperforms the single embedding scheme achieving gains in time performance ranging from 30% to 60%, while achieving the same level of recall and accuracy. Future work will address scenarios with more than two parties and different data types such as DNA sequence, etc.

REFERENCES

[1] A. Al-Lawati, D. Lee, P. McDaniel, Blocking-aware Private Record Linkage, *IQIS* 2005.

[2] R. Baxter, P. Christen, A comparison of fast blocking methods for record linkage, *In KDD Workshop on Data Cleaning, Record Linkage and Object Consolidation*, 2003

[3] J. Bourgain, On Lipschitz Embedding of Finite Metric Spaces in Hilbert Space, *Israel Journal of Mathematics*, no. 1-2, 1985

[4] P. Christen, Automatic record linkage using seeded nearest neighbour and support vector machine classification. *In Proc. of 14th ACM SIGKDD Intl Conf. on Knowledge Disc. and Data Mining*, Aug 2008

[5] W. Cohen, P. Ravikumar, S. Fienberg. A comparison of string distance metrics for matching names and records. *In KDD Workshop on Data Cleaning, Record Linkage and Object Consolidation*, 2003

[6] M. Elfeky, V. Verykios, A. Elmagarmid: TAILOR: A Record Linkage Toolbox. *In Proc. of ICDE*, 2002

[7] A. Elmagarmid, G. Panagiotis, S. Verykios, Duplicate Record Detection: A Survey, *IEEE TKDE*, Vol. 19, no. 1, 2007

[8] C. Faloutsos, K. Lin. FastMap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. *SIGMOD Record*, 24(2):163–174, June 1995.

[9] V. Gaede, O. Günther, Multidimensional access methods, *ACM Computing Survey*. 30, 2 , June 1998

[10] M. Hernandez, S. Stolfo, Real-world data is dirty: data cleansing and the merge/Purge problem, *Data Mining and Knowledge Discovery* 2(1):9-37, 1998.

[11] G. Hjaltason, H. Samet, Incremental distance join algorithms for spatial databases, *ACM SIGMOD*, 1998

[12] G.R. Hjaltason, H. Samet, Properties of Embedding Methods for Similarity Searching in Metric Spaces, *IEEE TPAMI*, Vol. 25, 2003

[13] G. Hristescu, M. Farach-Colton, Cluster-preserving embedding of proteins, Technical Report, Rutgers Univ., Piscataway, 1999.

[14] E. Jacox, H. Samet, Metric space similarity joins, *ACM Transactions on Database Systems*. 33, 2 Jun. 2008

[15] L. Jin, C. Li, S. Mehrotra, Efficient Record Linkage in Large Data Sets, *DASFAA* , 2003.

[16] N. Koudas, S. Sarawagi, D. Srivastava, Record Linkage: Similarity Measures and Algorithms, *ACM SIGMOD,* 2006.

[17] N. Linial, E. London, Y. Rabinovich. The geometry of graphs and some of its algorithmics application, Combinatorica, vol 15, 1995

[18] A. McCallum, K. Nigam, L. Ungar, Efficient clustering of high-dimensional data sets with application to reference matching. *Proc. 6th ACM SIGKDD Intl Conf. on Knowledge Disc. Data Mining*, 2000

[19] A. Moore, "An Introductory Tutorial on Kd-Trees." Extract from Efficient Memory-based Learning for Robot Control (Technical Report 209). Computer Laboratory, University of Cambridge, 1991.

[20] M. Scannapieco, I. Figotin, E. Bertino, A. Elmagarmid, Privacy preserving schema and data matching. *In Proceedings of the 2007 ACM SIGMOD Intl Conference on Management of Data* 2007

[21] W. Shen, P. DeRose, L. Vu, A. Doan, R. Ramakrishnani, Source-aware Entity Matching: A Compositional Approach. *In Proc. of 23rd Intl Conf. on Data Eng., ICDE*, April, 2007

[22] D. Talbert, D. Fisher, An empirical analysis of techniques for constructing and searching k-dimensional trees. *In Proc. of 6th ACM SIGKDD Intl Conf. on Knowledge Disc. and Data Mining*, Aug 2000

[23] V. Verykios, A. Elmagarmid, E. Houstis, Automating the approximate record-matching process. *Info. Sciences*, 126, July 2000

[24] V. Verykios, M. Elfeky, A. Elmagarmid, M. Cochinwala, S. Dalal. On the accuracy and completeness of the record matching process. *In Proc. of the 2000 Conf. on Information Quality*, Oct.2000.

[25] J. Wang, X. Wang, K. Lin, D. Shasha, B. shapiro and K. Zhang, An Index Structure for Data Mining and Clustering, *Knowledge and Information Systems*, vol. 2, no. 2, May 2000

[26] J. Wang et al, Evaluating a Class of Distance-Mapping Algorithms for Data Mining and Clustering", *Proc. ACM SIGKDD Int'l Conf. Knowledge Discorvery and Data Mining*,, Aug 1999

[27] M. Yakout, M. Atallah, A. Elmagarmid, Efficient Private Record Linkage, *In Proc. of 25th Intl Conf. on Data Eng., ICDE*, April 2009